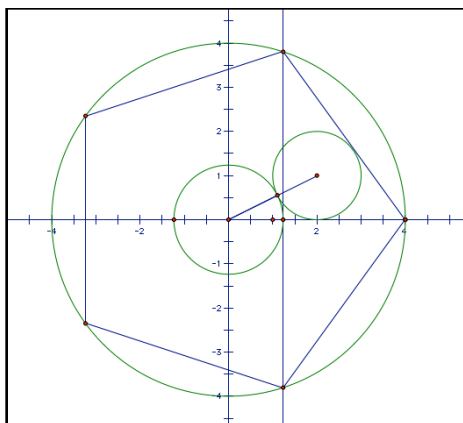


# Technology for Mathematics



**Factor** $[x^6 - 1]$

$$(-1 + x)(1 + x)(1 - x + x^2)(1 + x + x^2)$$

**Factor** $[x^4 - 2]$

$$-2 + x^4$$

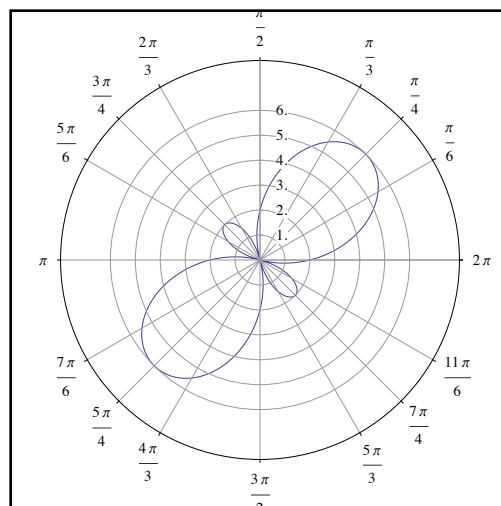
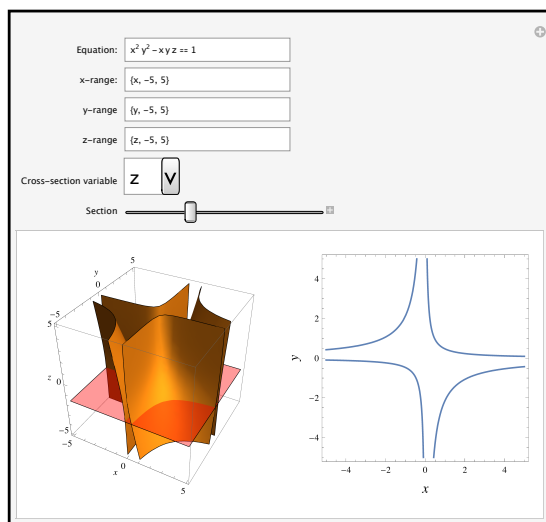
**Factor** $[x^4 - 2, \text{Extension} \rightarrow \{2^{1/4}, i\}]$

$$-(2^{1/4} - x)(2^{1/4} - ix)(2^{1/4} + ix)(2^{1/4} + x)$$

In[22]:= **TraditionalForm** $[\text{LogicalExpand}[\neg (p \wedge q) \Leftrightarrow (p \vee q)]]$   
 Out[22]/TraditionalForm=  
 $(p \wedge \neg q) \vee (q \wedge \neg p)$

**Reduce** $[x^2 + y^2 < 25 \ \&\& \ y > x - 1, \{x, y\}]$

$$\left(-5 < x \leq -3 \ \&\& \ -\sqrt{25 - x^2} < y < \sqrt{25 - x^2}\right) \ || \ \left(-3 < x < 4 \ \&\& \ -1 + x < y < \sqrt{25 - x^2}\right)$$



Dr. Christopher Moretti

The Geometer's Sketchpad is a registered trademark of Key Curriculum Press.

Mathematica is a registered trademark of Wolfram Research, Inc.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/3.0/> or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

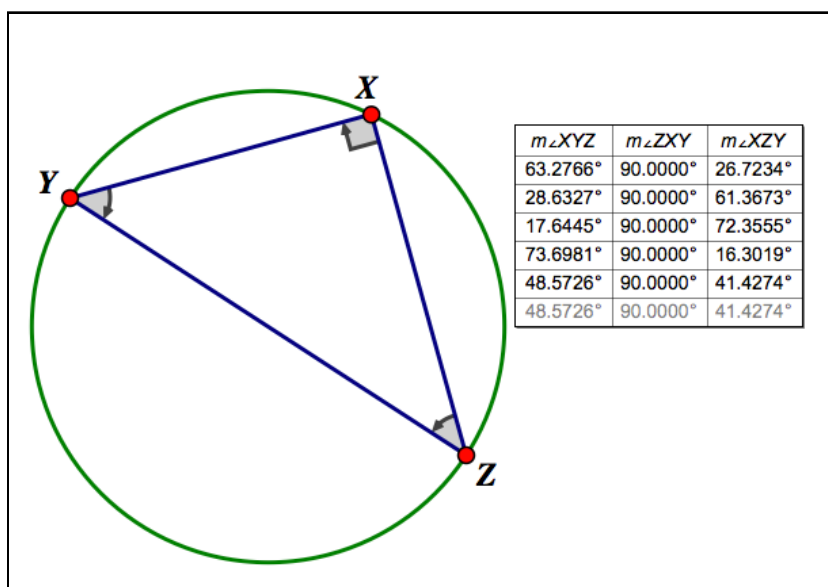
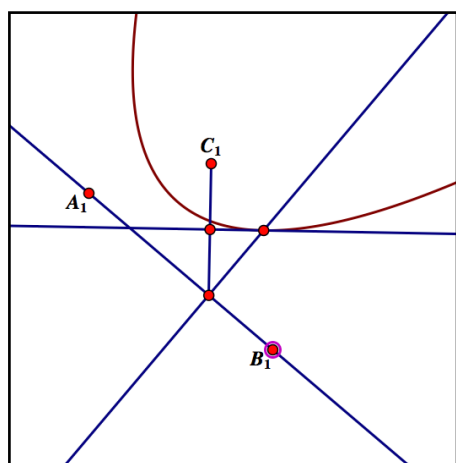
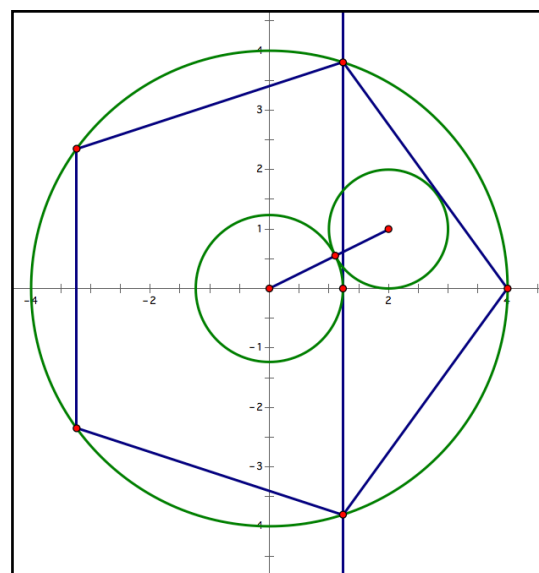
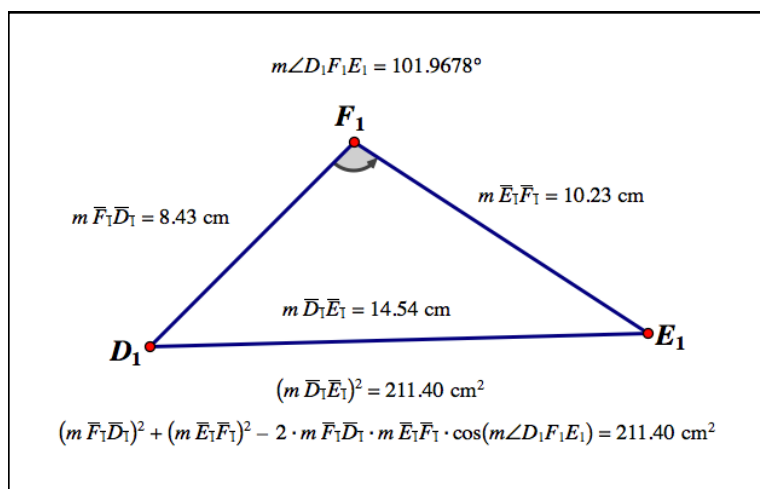
Last edit January 7th, 2018



Chapter 1 - Geometer's Sketchpad .....	5
<i>Section 1.1 - An Introduction</i> .....	6
<i>Section 1.2 - Basic Constructions, Measurements, and Marking</i> .....	6
<i>Section 1.3 – Advanced Constructions</i> .....	19
<i>Section 1.4 – Complex Constructions and Investigations</i> .....	30
<i>Section 1.5 – Constructing Loci</i> .....	36
<i>Section 1.6 – Custom Tools</i> .....	44
<i>Section 1.7 - Straightedge and Compass Constructions (optional)</i> .....	48
Chapter 2 - Mathematica Basics .....	77
<i>Section 2.1 - An Introduction</i> .....	78
<i>Section 2.2 - Working with Mathematica Notebooks</i> .....	80
<i>Section 2.3 - Basic Mathematica Functions</i> .....	90
<i>Section 2.4 - Basic Graphing in Mathematica</i> .....	99
<i>Section 2.5 - Algebraic Computations and Manipulations</i> .....	123
<i>Section 2.6 - Solving Equations and Inequalities</i> .....	137
<i>Section 2.7 - Defining Functions</i> .....	153
Chapter 3 - Additional Pre-Calculus Mathematica Topics .....	160
<i>Section 3.1 - An Introduction</i> .....	161
<i>Section 3.2 - The Basics of Lists</i> .....	161
<i>Section 3.3 - Basic Statistics</i> .....	179
<i>Section 3.4 - Basic Interactive Manipulations</i> .....	193
<i>Section 3.5 - Advanced Graph Control</i> .....	201
<i>Section 3.6 - Importing and Exporting from Mathematica</i> .....	206

<i>Section 3.7 - Additional Graphing Commands .....</i>	<i>210</i>
<i>Section 3.8 - Optimization .....</i>	<i>220</i>
<i>Section 3.9 - Logic in Mathematica.....</i>	<i>231</i>
Chapter 4 - Elementary Mathematica Programming .....	247
<i>Section 4.1 - An Introduction.....</i>	<i>248</i>
<i>Section 4.2 - Types of Variables .....</i>	<i>256</i>
<i>Section 4.3 - Conditionals and Loops.....</i>	<i>269</i>
<i>Section 4.4 - Formatting and Presentation .....</i>	<i>283</i>
<i>Section 4.5 - Random Generators.....</i>	<i>298</i>
<i>Section 4.6 - Default Values and Program Options .....</i>	<i>308</i>
<i>Section 4.7 - Raw Graphics.....</i>	<i>318</i>
Chapter 5 - Mathematica in Analytic Geometry & Calculus .....	331
<i>Section 5.1 - An Introduction.....</i>	<i>332</i>
<i>Section 5.2 - Analytic Geometry.....</i>	<i>332</i>
<i>Section 5.3 - The Computations of Calculus.....</i>	<i>347</i>
<i>Section 5.4 - Applications of the Derivative.....</i>	<i>365</i>
<i>Section 5.5 - Applications of the Integral.....</i>	<i>379</i>
<i>Section 5.6 - Geometric Computation and Regions .....</i>	<i>395</i>
<i>Section 5.7 - Into the Third Dimension .....</i>	<i>409</i>
<i>Section 5.8 - Calculus Computations for Three or More Dimensions.....</i>	<i>424</i>
<i>Section 5.9 - Differential Equations .....</i>	<i>435</i>

# Chapter 1 - Geometer's Sketchpad



## Section 1.1 - An Introduction

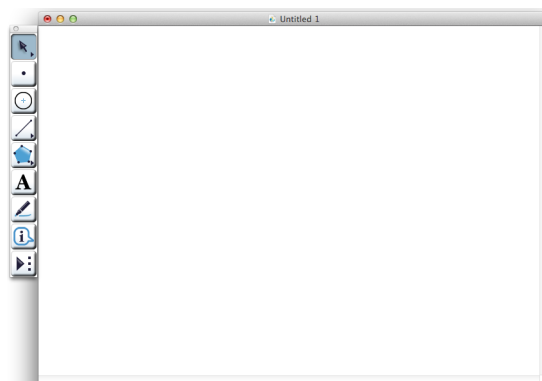
Geometer's Sketchpad is a program specifically designed for investigating classical geometry (that is, geometry in the flat plane as opposed to curved surfaces like a sphere). It provides a set of virtual Euclidean tools (meaning those based on a compass and straightedge) and allows you to sketch all sorts of lines, angles, points, and circles. If that were the limit of its ability, then it would simply be a digital reproduction of a regular compass and ruler. But it has two additional features (at least that we will discuss) which take it beyond the pencil-and-paper approach. First, it can numerically measure many geometric and numerical quantities like slope, angular measure, and coordinates. Second, you can drag or deform the objects you define in the plane and all the associated measurements will automatically be updated. Imagine dragging the vertex of an angle around and watching the numbers for the measure and slopes of the lines change as you drag (hard to do on a piece of paper). It is this interactivity between the user and defined objects that makes Geometer's Sketchpad such a useful tool in geometric investigations.

The limitations of Geometer's Sketchpad are a consequence of its geometric approach. At the introductory level we will use it at, you are more or less restricted to Euclidean tools and constructions (bisecting angles, constructing circles, and so on) and the graphs of functions. Because the "plane" for Geometer's Sketchpad is really an approximation of the true plane, distances and angles are approximated numerically rather than giving exact values (so you will usually see 1.414 rather than  $\sqrt{2}$ ). Keeping these sorts of restrictions in mind, however, Geometer's Sketchpad is still an excellent tool for geometric investigation.

Geometer's Sketchpad recently went from version 4.x to version 5.x. As Sketchpad has picked up many new features in version 5, documents saved from version 5 will not open in version 4. To save a version 4 compatible sketch, you need to select that option in the save dialog.

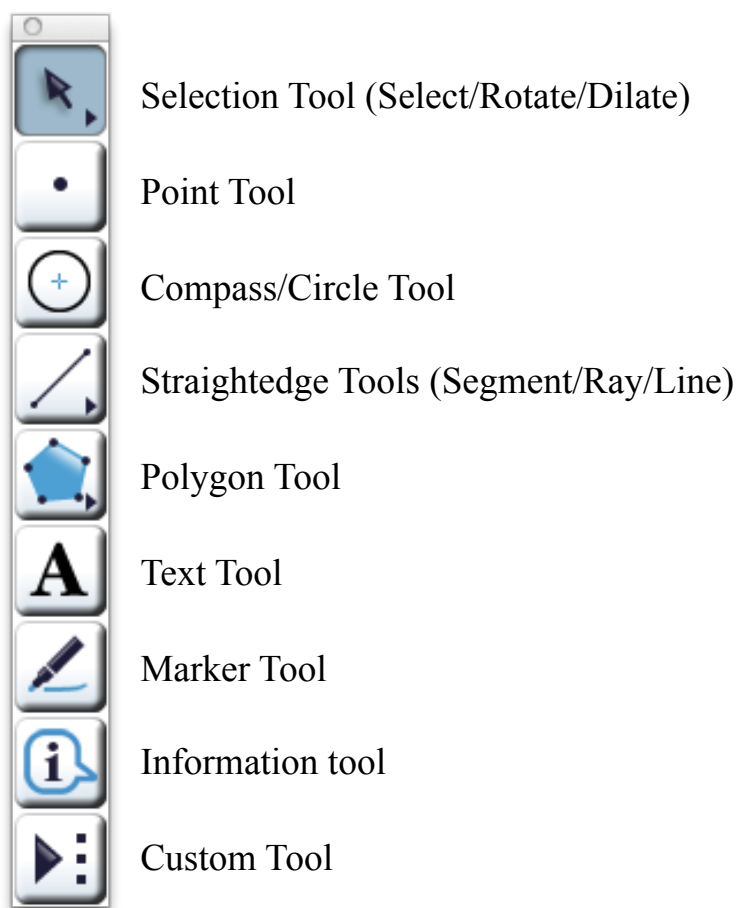
## Section 1.2 - Basic Constructions, Measurements, and Marking

When you first start Geometer's Sketchpad, you have a blank canvas on which to work (called a "sketch"). This new sketch looks something like this:



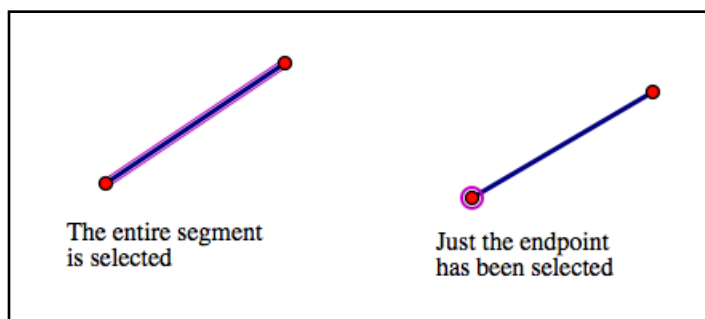
*A blank sketch in Geometer's Sketchpad*

Of great importance is the toolbar in the upper left corner – these are the tools that you use to create basic objects in the plane. The toolbar looks like this:



The functions of the different buttons are as follows:

**Selection Tool:** This allows you to both select and drag objects. Selecting is done with a single click; multiple objects can be selected by simply clicking one object after the other (this is somewhat counterintuitive as it does not work like selections in most other programs, where a click on a second object deselects the first). To drag an object, simply click on it and drag it. This is actually somewhat more complicated than it sounds, because some objects (like line segments) are actually multiple objects bound together (for a line segment there are the endpoints and the actual segment). Dragging the individual objects rather than the whole can have very different effects. Selecting an entire segment by clicking in its middle will enable you to drag the entire segment as a whole; dragging one of its ends will actually change the segment as the other endpoint will stay fixed.



*the difference between selecting a segment and just an endpoint*

To deselect an object or group, either click on them a second time or click on the background, which will deselect everything. If you click and hold on the Selection tool you will get a pop-out menu allowing you to select the Rotate and Dilate tools - these will be discussed in more detail in Section 1.3.

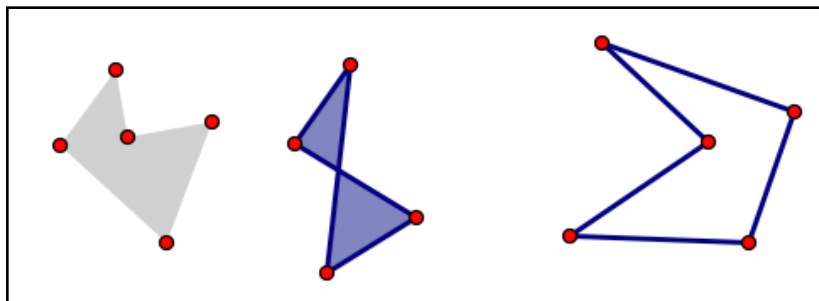
**Point Tool:** This allows you to create points in the plane. When this tool is selected, simply click anywhere in the plane to make a point (points can also be created by giving coordinates, which we will talk about later). You can also “attach” a point to an object like a circle or a line by hovering the Point tool over it until you see the object turn red and then clicking. Such a point will be stuck to the object but can be slid along it.

**Compass/Circle Tool:** This tool allows you to create circles, mimicking the use of a compass in geometry. Creating circles is done by clicking to lay down two points. Where you click first will be the center of the circle, and where you click second will be the endpoint for the circle’s radius. These two points will be visible along with the newly-created circle.

**Straightedge Tools:** This tool is used to create lines, rays, and segments. All three are created the same way – a click-drag. The initial click determines the first point for the object, and where you let go of the mouse after dragging is a second point (you can also just click twice as if you were creating points with the Point tool - as you move off the first point the straight object you are making will follow the cursor). For lines and segments the order doesn’t really matter, but for rays the order makes a big difference (as the ray from A through B is different than the ray from B through A). If you click and hold down the button on the toolbar a popup menu will allow you to select either the Line, Segment, or Ray tool.

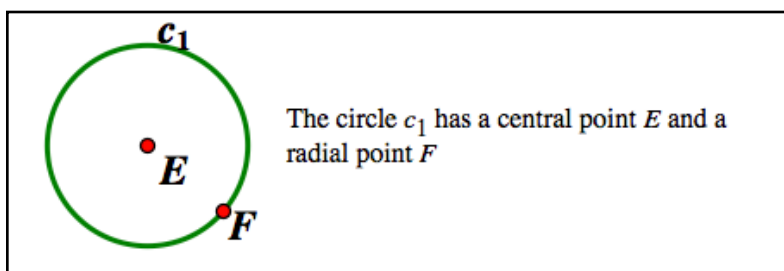
**Polygon Tool:** This tool allows you to create polygonal figures. To create a polygon with this tool, simply click to lay down the corners of the polygon one at a time. Either double clicking a point or laying down a new point on top of an old one will complete the polygon. The default version of this tool only creates the area within

the polygon (its “interior”). By clicking and holding on the Polygon tool you will see two other versions of it; one that creates both the sides and interior, and one that creates only the sides.



*polygons created with the three versions of the polygon tool*

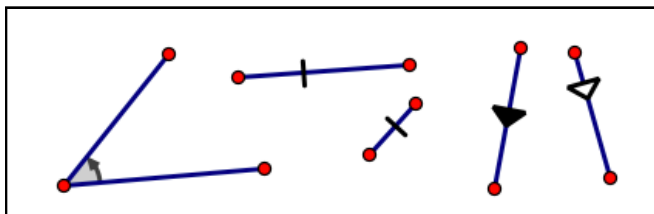
**Text Tool:** This tool has three uses. The first use is to show or hide the label of an object (for example, a newly drawn line segment may have the name “ $j$ ”, even if not shown). To show or hide the label of an object, simply click on it with this tool. The second use is to create captions. Simply double-click in an empty area of the plane, and you will get a box in which you can enter any text you like. Click outside the text box to finish the entry. These text boxes may be dragged around with the selection tool. You can copy the label for a given object directly into the text by simply clicking on a given object or label; the label will insert at the cursor and will link directly back to the object if you click on it and hold. The third use is to bring up a window with detailed information on an object (the “Properties” window). This window is brought up by double-clicking on the object with the text tool (alternatively, you can right-click an object and select “Properties” from the menu). This window includes the label of the object as well as the object’s “parents” and “children”. The parents of an object are the things that define it (the parents of a line segment are its endpoints, for example), and its children are what it helps to define (the midpoint of a line segment is a child of the segment).



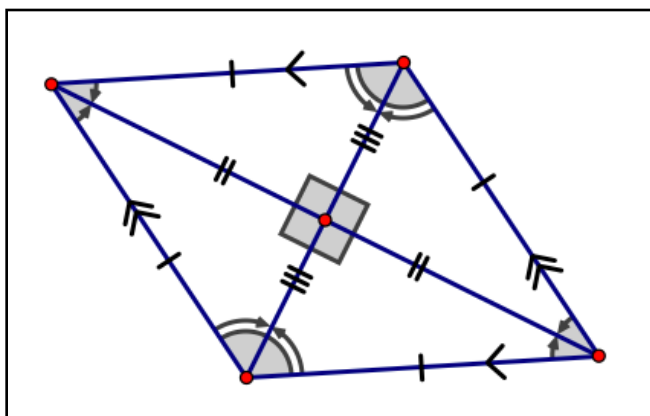
*labels revealed and inserted into a text box*

**Marker Tool:** The purpose of the Marker tool is to help you highlight important details in your sketch. Its most direct use is to simply let you freehand draw in a sketch. Simply select the tool, move it to the sketch, and click and hold down the mouse

button to draw directly on the sketch (when you release the mouse button, the entire mark you put down will be a selectable object). More importantly, the Marker tool allows you to have Sketchpad make standard geometric notations on angles (curved arcs from one side to the other), crossbars to denote equal lengths, or arrows to denote parallel lines. Repeatedly clicking a mark on an angle, segment, or line will add to the number of strokes, up to a limit of 4. Marks are created by clicking on a straightedge object or by clicking one side of an angle and dragging to the other side of it. By right-clicking on a mark you can bring up its properties window, which allows you to control the mark style, arrowheads, and the number of strokes. One thing to be careful of is that while geometric marks typically denote either congruency or parallelism, marks in Sketchpad can be applied to any object whether they have these properties or not. So you can mark line segments of different lengths with the same crossbar mark, and so on (although you shouldn't).



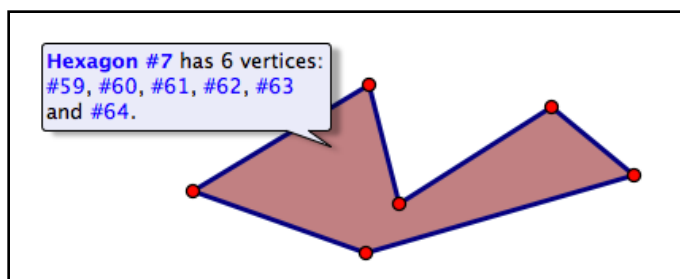
*several marked objects, including two segments that probably shouldn't use the same mark*



*a fully marked rhombus, including marks for congruent lengths, congruent angles, and parallel sides*

Information Tool: The information tool allows you to create “thought bubbles” for an object by clicking on them. These thought bubbles generally tell you what the type of an object is (segment, polygon, etc.) and what the parents of the object are. You can have multiple bubbles up at once, although all of them will fade away when another tool is selected.



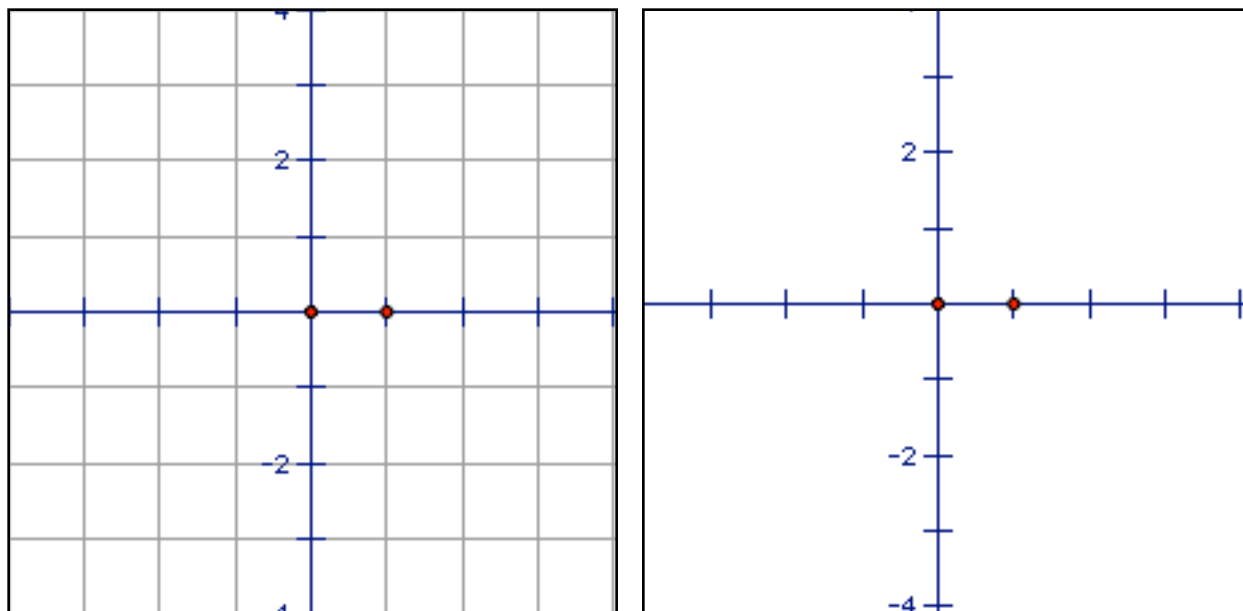


*a descriptive “thought bubble” made with the information tool*

**Custom Tool:** This button allows you to create or access your own custom tools in Sketchpad - ones that are not included with the program. We will discuss custom tools in Section 1.6.

It is definitely worth your time to play with the different tools for building these objects. In particular, you should experiment with the effects of dragging different parts of more complex objects like circles. For example, you might make a circle and see what happens when you drag its center, its edge, and the point you used to define its radius. If your sketch gets too cluttered, you can always close it and start a new one (using the Close and New commands from the **File** menu) or delete objects by selecting them and hitting the delete key (or using a Cut command, just as in a word processor).

One thing you should have noticed by now is that by default a sketch is completely blank. This emphasizes that in classical geometry there is no real notation of location other than points. In most of mathematics we make use of a reference system, usually by using Cartesian coordinates (when you include coordinates, you’ve moved from classical geometry to analytic geometry). These coordinates are already present in Geometer’s Sketchpad, although by default they are not shown. They can be revealed by selecting Define Coordinate System from the **Graph** menu. The coordinate axes appear along with a coordinate grid (the grid can be hidden or shown using Show Grid or Hide Grid from the **Graph** menu). The axes look something like this:



*Cartesian coordinate systems with and without grid lines*

Note the two points on the axes. The point at  $(0,0)$  represents the center of the coordinate system, and this point can be dragged around. Dragging this point will move the axes and will change the coordinates of many of the objects you've created. The second point at  $(1,0)$  represents the coordinate unit (the basic notion of scale). Dragging this point to the right or left will change the units on both axes and the coordinates for many of the objects you've defined (making the unit larger will make all the coordinates smaller in size as they will be "closer" to the origin). You can also force the coordinate unit to match the length of an existing line segment by selecting "Define Unit Distance" from the Graph **Menu**. You can create points using coordinates rather than the Point tool. Simply choose Plot Points from the Graph menu, and a window will pop up allowing you to define the points by giving x- and y- coordinates (or even in polar coordinates if the Grid Form in the Graph menu is set to polar). You also have the option to make your points "free" or "fixed". Free points are independent of the coordinate system, so if you move the coordinate axes they will stay in their location onscreen (but their coordinates will change). Points made by the Point tool are usually free points. Fixed points are tied to the coordinate system, so if you move the axes these points will move right along with them (so their coordinates will never change).

There are two important things to remember about coordinates in Geometer's Sketchpad. First, even if the axes aren't shown they are still there – so points always have coordinates, even if you can't see the axes they are measured by. Second, the coordinates in Geometer's Sketchpad are only approximate, not exact. This is a drawback of working on a computer screen. The plane used by Sketchpad is really a discrete grid of dots, not the truly continuous and seamless object we use in geometry. This means that coordinates will always be approximate, and all the measurements based on coordinates (like length, area, etc.) will be approximations as well.

Once you have coordinates, you are ready to take many different kinds of measurements. The measurements you make are generally placed in the upper left corner of your sketch (although they can be moved with the Selection tool) and will automatically be updated to reflect any changes you've made to the objects or coordinates. To measure something, simply select it and select the appropriate option from the **Measure** menu. Remember, some measurements require selecting more than one object, which can be done by simply clicking on one object and then another. Things you will find in the **Measure** menu include:

**Distance:** This measures the distance between two objects. Not only will this measure the distance between points, but also the distance from a point to a line. Note that the distance measured is always done as if you put a ruler to the screen and measured the physical distance, which is independent of coordinates. The units and the precision of measurements can be changed in the Sketchpad preferences.

**Coordinate Distance:** This measures the distance between two appropriate objects (2 points, a point and line, etc.) using the current coordinate system. If the scale of the coordinate system has been changed, this will not match the answer given by measuring the Distance. If your construction is based on or involves a "unit length" you will want to use "Define Unit Length" to set the proper unit length and then measure all distances and segment lengths using coordinate distances (for segments, the length is the coordinate distance between the endpoints).

**Length:** This measures the length of a line segment (physical length, without regard to a coordinate system).

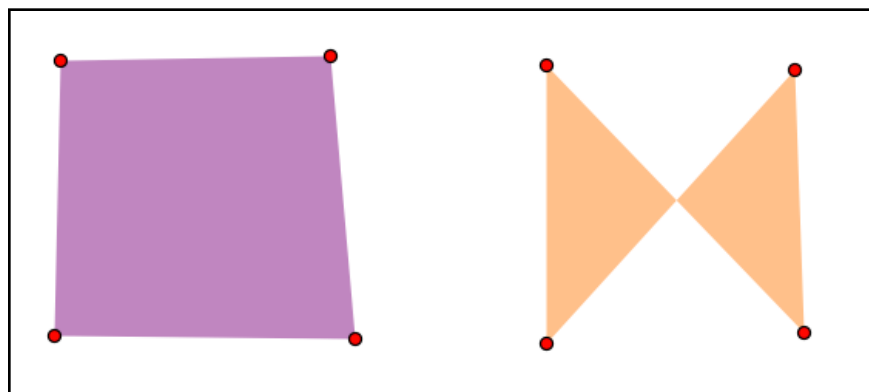
**Slope:** This measures the slope of a ray, line, or line segment.

**Radius:** This will measure the radius of a selected circle or circles.

**Circumference:** This measures the distance around a selected circle or set of circles.

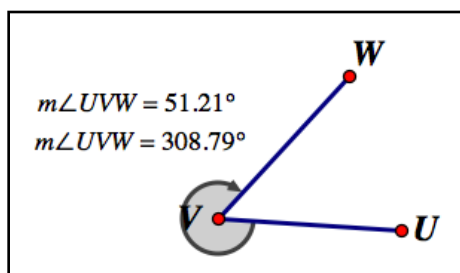
**Area:** This gives the area of a circle or polygon. See the following description of perimeter for the care that must be taken with polygons.

**Perimeter:** This gives the distance around a polygon. Before you can measure the perimeter, you must construct the interior of the polygon and have it selected (this construction is covered in Section 1.3 if you haven't built your polygon with the appropriate version of the Polygon tool). This is important because the order in which you select the points can change the polygon – if you have the corners of a square, selecting them in different orders can yield both a square and an hourglass shape, and these have different perimeters.



*two different polygons created from the same four points using different selection orders*

**Angle:** This gives the measure of an angle. To measure an angle, select either the 3 points that define it, the two sides that define it, or the angle's mark (if it has one) and then choose Angle from the **Measure** menu. If you select 3 points to measure the angle, the order in which you select the 3 points that make up the angle makes a big difference – usually angles ABC and BAC are quite different. If you use 3 points or 2 sides to measure an angle, Sketchpad will assume the angle is a “simple” one - where “simple” means one whose measure is from  $0^\circ$  to  $180^\circ$ . To measure a non-simple angle, either use the marker tool to mark the angle as being greater than  $180^\circ$  by marking it the long way around and use the mark to measure it or right-click on the angle's mark and select “Properties” - this will let you set the angle to be reflex, clockwise, or counter-clockwise.



*3 points defining both a simple and reflex angle*

**Arc Angle:** This will give the measure of the central angle of a selected circular arc.

**Arc Length:** This gives the length of a selected circular arc.

**Ratio:** This gives the ratio of the lengths of two selected line segments (the order of selection makes a difference).

**Circumference:** This measures the distance around a selected circle.

**Radius:** This measure the radius of a selected circle.

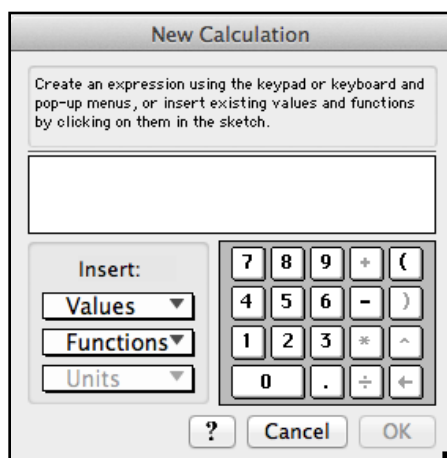
**Value of Point:** Given a point P on a segment from A to B, this measures how far along the segment P is from A to B. The value 0 means  $P=A$ , the value 1 means  $P=B$ , .75 means P is 3/4 of the way from A to B, and so on.

**Coordinates:** This reveals the coordinates of a selected point or points.

**Equation:** This gives the equation of a circle or line.

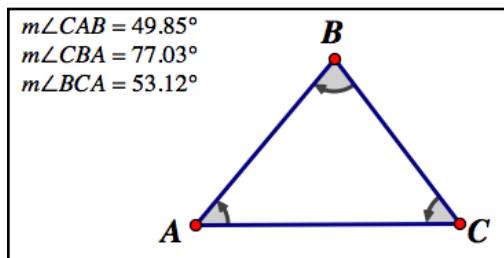
Try creating several different objects and making different measurements of them. Also try dragging the objects around or otherwise altering them and watch what that does to the corresponding measurements. Note that although by default your measurements will appear along the left edge of the screen they may be selected and dragged around wherever appropriate.

The measurements you make can also be combined by a calculator, which is available by selecting “Calculate...” from the **Number** menu. This will bring up a small calculator:



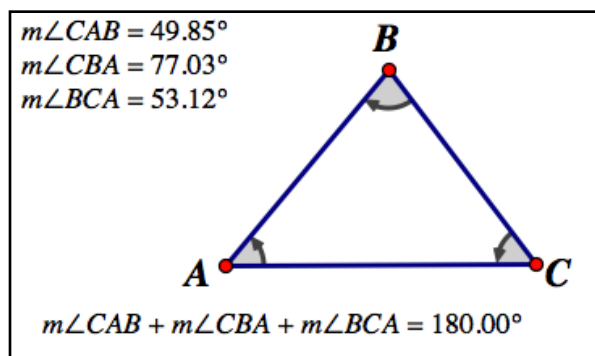
*the Sketchpad calculator*

You can use this calculator to create a new measurement based on old ones. To take a measurement and use it as part of a larger calculation, simply click on the measurement and it will be inserted by name into the calculator at the cursor's current position. Hitting “OK” will create the new combined measurement in the sketch. For example, create a triangle ABC by using the segment tool to create segments AB, BC, and CA. Mark the 3 angles and then measure them:



*a triangle and its angles*

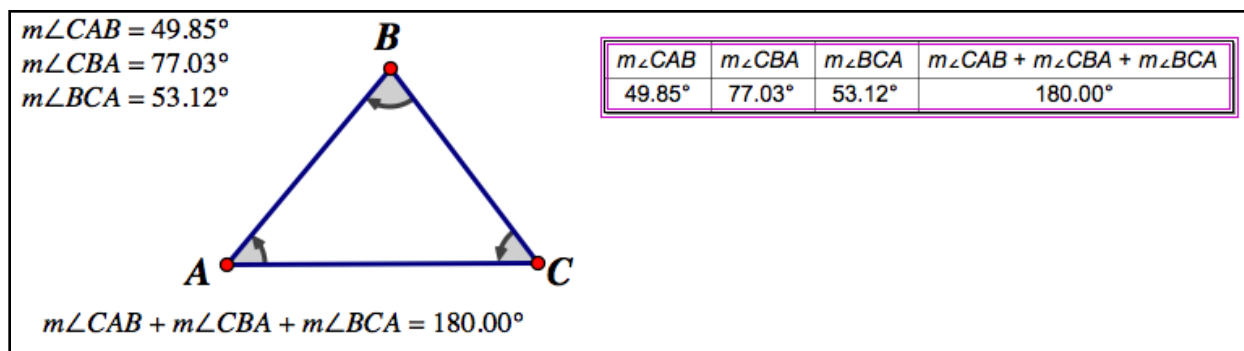
Create a new measurement in the calculator by clicking on the first angle measurement, then “+”, then the second angle measurement, then “+”, then the final measurement, and then “OK”:



*the sum of the three angles, after dragging the new measurement below the triangle to make it easier to see*

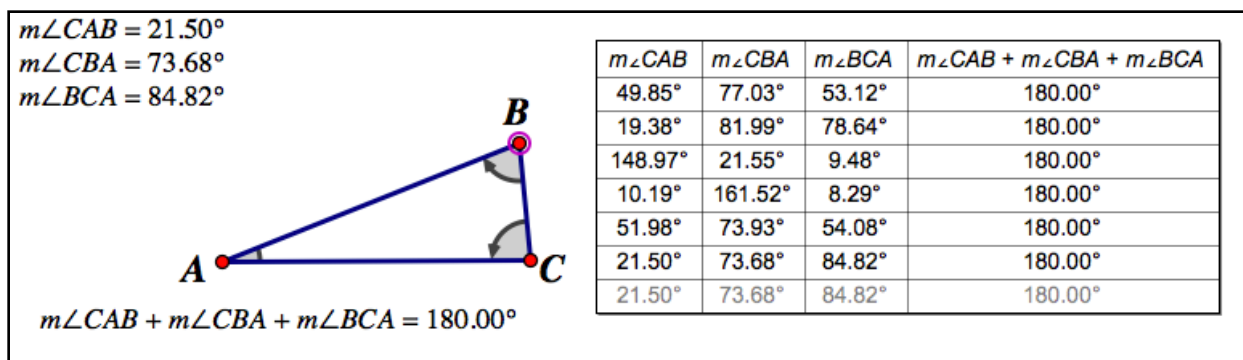
Now take the point B and drag it around - you will see the individual angles change, but the sum will always be  $180^\circ$ .

This kind of interactive measurement is at the heart of what makes Sketchpad such a valuable tool in geometry, and we will look at several similar geometric investigations in the Section 1.4. It's worth noting that you can make a table from any given set of measurements as well. To make a table, simply select the measurements you want to be columns in the table and then select “Tabulate” from the **Number** menu. For example, selecting the 4 measurements above and selecting Tabulate gives:



*creating a table from our angle measurements*

Double-clicking the row in the table permanently adds the current measurement to the table. So by double-clicking the current table row, then moving B, and repeating, we get a table of several different measurements:

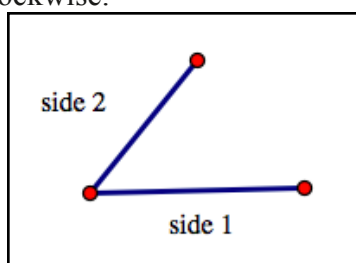


*a table that covers a whole set of angular measurements*

Having a whole range of measurements makes the conclusion “the sum of the angles in a triangle is always  $180^\circ$ ” a lot easier to draw.

## Section 1.2 Homework – Basic Constructions and Measurements

- 1) Explain the functions of the basic tools located in the Sketchpad toolbar.
- 2) How do you select the Line, Segment, or Ray tools in the toolbar?
- 3) What does it mean for an object to be a parent? To be a child?
- 4) Explain the three versions on the Polygon tool.
- 5) Create an angle similar to the one below by using the segment tool (you will need to click twice to create the point which is the vertex). Using the Marker tool, see what happens when you click-drag from side 1 to side 2 clockwise and counter-clockwise. Undo these, and try from side 2 to side 1 counter-clockwise.



*the angle for problem 5*

- 6) How do you create axes in a sketch? How do you move the origin and change the scale of the axes?
- 7) In a sketch, create axes and then create two points at coordinates  $(-1,2)$  and  $(3,6)$ . Measure the distance between these points and the slope of the segment which joins them.
- 8) Create a circle using the Circle tool. Find the coordinates of the center and radial point. Find the length of the radius as well as the perimeter, area, and equation of the circle.
- 9) Let A be the point  $(1,1)$ , B the point  $(4,5)$ , and C the point  $(6,-1)$ . Find the measures of the angles ABC, ACB, and BAC. Find the lengths of the segments AC, AB, and BC.
- 10) Create a circle in the plane. What is the difference between dragging the center as opposed to

dragging the radial point?

- 11) Create the point (3,2) in a sketch. Find polar coordinates of this point. (Hint: Switch coordinate systems using the **Graph** menu).
- 12) Create a mark on a segment. Right-click the mark, and explain the functions of what you see in the popup menu.
- 13) Create a circle and find the following measurements: area, circumference, radius, and area divided by circumference. Make a table from these measurements, deforming the circle each time to get a new measurement row. Make a conjecture from this table.
- 14) Go the Sketchpad preferences menu, and determine what angular measurement systems can be used. Does changing the “precision” affect measurements that already exist?



## Section 1.3 – Advanced Constructions

By this point you should be familiar with constructing basic objects using the toolbar buttons in Geometer's Sketchpad – points, lines, segments, circles, and so on. When really working with geometry, you need more than just the creation of basic objects. If you have two lines, you may need their point of intersection. Given a line and a point, you may need to drop a line from the point which meets the first line at right angles. Geometer's Sketchpad gives you the tools to do many of these “advanced” constructions where you use a set of preexisting objects to define a new one. This is the basic idea behind “parent” and “child” objects mentioned in Section 1.2 (under the Text tool, which can give you information about parent and child objects). The parent objects are the ones you are using as inputs for your construction, and the child object is the output of your construction.

The advanced constructions are found for the most part under the **Construct** menu (as you probably guessed). These constructions and their parents are:

**Point on an Object:** After selecting an object, this command will create an extra point on it. This is a “free” point in the sense that it can be dragged around on the object. This is a very useful tool for measuring angles which are defined by intersecting lines – it takes three points to make an angle, and the intersection point is just one of them. By creating points on the other lines you will have the three you need to define and measure the angle. In general this construction is a more precise way to bind a point to an object like a line or circle than using the “sticky click” of the Point tool.

**Point at Intersection:** This will create points wherever two selected objects cross. If the objects cross more than once (say, like when a line crosses a circle), all the intersection points are constructed. Intersection points come up all the time in geometry – in fact, most points in geometric constructions arise from intersecting objects like lines and circles.

**Point at Midpoint:** This creates the midpoint of a line segment, which appears in many geometric problems (particularly those using perpendicular bisectors).

**Line/Segment/Ray:** This will draw the line or segment or ray through the chosen points, essentially mimicking the use of a straightedge. When constructing a ray, remember that the order you chose the points makes a big difference – the ray from A through B is different than the ray from B through A. One common use of this construction to construct is a segment between two points in that are already in a line. When you build a segment inside of an existing line the portions of the line outside the segment will become dashed so you can easily see both parts; you can change the appearance of the line by right-clicking on the line and selecting “Solid” instead of “Dashed”.

**Perpendicular Line:** Given a selected line and point, this builds the line through the point perpendicular to the selected line.

**Parallel Line:** Given a selected line and point, this builds the unique line through the point parallel to the selected line. Geometer's Sketchpad works in Euclidean geometry, so this makes good sense (budding non-Euclidean geometers will have to find another program).

**Angle Bisector:** This will create the line which bisects (cuts in half) the selected angle.

To create the bisector you will need to either select 3 points which define the angle (order selection makes a difference here) or the 2 sides of the angle (if the angle consists of 2 segments with a common endpoint). You cannot create an angle bisector by selecting an angle's mark.

**Circle by Center and Point:** Given two selected points (say A chosen first and B chosen second), this builds the circle centered at A whose radius would be the segment AB.

The selection order makes a difference in this construction. This construction mimics the use of a compass where the point of the compass is placed at one point, the foot another, and a circle is spun out.

**Circle by Center and Radius:** Given a selected point and line segment, this creates the circle centered at the point whose radius is the same as the length of line segment.

The nice thing about this construction is that the segment does not have to contain the "central" point, which eliminates any need for copying the line segment. This construction mimics the second use of a compass, where the arms of the compass are used to measure a segment, and then the point of the compass is placed at another point and a circle spun out.

**Arc on a Circle:** Given a selected circle and two points on the circle, this construction builds the circular arc from one point to the other. The constructed arc always runs counterclockwise from the first selected point to the second, so the selection order can make a difference.

**Arc Through 3 Points:** Given selected points A, B, and C, this creates the circular arc from point A through point B which ends at point C. The selection order can make a difference here, although the various arcs will all be part of the same circle.

**Interior (of a Circle or Polygon):** This creates the interior of the selected circle or polygon. Selection order plays a big role in working with polygons (see measuring the perimeter of a polygon in Section 1.2), and a polygon's interior must be constructed before measuring its perimeter or area.

**Locus:** A locus is the most complicated of all constructions you will find in Geometer's sketchpad. In geometry, a locus is a set of points which is defined by some geometric conditions (for example, a circle is a locus because it is all points a fixed distance from a given point). We will discuss loci in more detail in Section 1.5.

Two important constructions are in the **Number** menu:

**New Parameter:** This allows you to define a constant with a given value. For example, you could define the parameter "a" to have an initial value of 2. This will create the statement "a=" followed by a box with the value 2 in it. You can click in the box and change the value of the parameter, and any definitions which use that parameter will automatically update to match.

**New Function:** This allows you to create an formula. With a formula selected, "Plot Function" will become available in the **Graph** menu. A formula may include one or more parameters - to use a parameter as part of a formula, simply move the cursor in the calculator to the appropriate place and then click on the parameter in your sketch.

Three important constructions can be found in the **Graph** menu:

**Plot value on axis:** If a segment is selected, this lets you create a point on the segment which is a given fraction from one endpoint to the other (this is the reverse of measuring the "Value of a Point"). If you enter the value .25, you will have created a point which is one fourth of the way across the segment. If you have selected either the x- or y-axis in a plot, this will let you create a point at a specific x- or y-coordinate.

**Plot New Function:** This brings up the calculator and lets you create a new function and its graph. You can create Cartesian graphs of the form "y=..." or "x=..." and polar graphs of the form "r=..." and " $\theta$ =...". Graphing a function will bring up the coordinate axes if they are not already present. Graphs can also be intersected with other Sketchpad objects (although for other graphs this is restricted to graphs of the same "type", i.e. two "y=" graphs can be intersected, but not a "y=" and "x="). When you select a graph, a new construction "Point on Function Plot" will become available on the **Construct** menu. This will let you create a point which can be slid along a graph.

**Plot Parametric Curve:** Given 2 formulas in x, selecting "Plot Parametric Curve" will bring up a window which will let you combine the two formulas into a parametric graph. The first function selected will give the x-coordinate, and the second will give the y-coordinate. You also get to pick the domain for the variable. The default domain is 0 to 1, but in practice you will most likely need to use a larger range of values to see a representative portion of the graph.

In addition to the constructions there are also different transformations of objects you can apply in Sketchpad, and these can be found under the **Transform** menu:

**Mark Center:** When applied to a point, this point becomes the central focus for the Dilate and Rotate transformations. Mark center can also be achieved by double-clicking on a given point with the selection tool. When marking a center you will see an animation on the point that resembles an archery target growing and shrinking.

**Mark Mirror:** When applied to a line, segment, or ray, the object becomes the mirror through which other objects can be reflected. Mark mirror can also be achieved by double-clicking on the appropriate object with the selection tool. When marking something as a mirror, you will see an animation that involves two squares growing and shrinking on the object.

**Translate:** This brings up a window which allows you to create a copy of the selected object which has been translated (moved but otherwise is unchanged). You can select the direction of the translation in either Cartesian or polar coordinates. Unlike the other constructions, this is found under the **Transform** menu.

**Rotate:** This will bring up a window in which you can enter an angle to rotate an object through. This will create a new object (the original is still present), and the rotation is done about the most recently marked center. A draggable version of Rotate is available through the Rotate Tool, available under the Selection Tool's popup menu. you can use the Rotate transformation as a protractor; to build an angle  $\theta$  with the segment AB as one side, mark A as the center, and then rotate the point B (as well as the segment AB if you want) through  $\theta$ . From a pure straightedge-and-compass perspective this is cheating (just as many uses of a protractor is "cheating" in classical geometry) as it allows you to build angles (like say  $20^\circ$ ) which cannot be created with the Compass and Straightedge tools alone.

**Dilate:** This will bring up a window in which you can set the dilation ratio for the selected object about the previously marked center. A ratio greater than 1 in size represent an expansion of the object away from the marked center, and a ratio less than 1 in size represents a contraction towards the marked center. A positive ratio means the newly created object will be on the same side of the center as the original, and a negative ratio will put the newly dilated object across the center from the original object. Before selecting the "Dilate" button, a green pre-image of the dilated shape will show you the position of the object-to-be.

**Reflect:** Choosing reflect will create the reflection of the selected object about the previously marked mirror. The Reflect transformation only works about a mirror -

that is, a line, segment, or ray. Reflections through a point can be obtained by using Dilate with a ratio of -1.

The children of these constructions and transformations will automatically be updated if you change the parent objects. For example, if you have a line segment and have constructed the midpoint, then dragging one of the ends of the segment will change the midpoint as well – Sketchpad will automatically update the midpoint's location to take into account the new position and length of the segment.

You can use sequences of these constructions to create other ones, such as the perpendicular bisector of a line segment. This particular example can be done by both a long sequence of constructions (mimicking what you would do with a compass and straightedge) or a shorter sequence using some of the advanced constructions to skip a few steps. Using a compass and straightedge, you would use this sequence of steps to form the perpendicular bisector of a segment:

- 1) Sketch the segment AB you wish to find the bisector for.
- 2) Spread the points of the compass so they are farther apart than half the length of AB.
- 3) Use that compass width to build circles centered at the endpoints A and B.
- 4) Locate the two points of intersections of the circles, and draw the line through those two points (the circles will cross because of how wide you've spread the compass points).
- 5) The line you just created is the perpendicular bisector of AB.

Each of these steps translates directly into Geometer's Sketchpad as follows:

- 1) Sketch in the segment AB with the Segment tool.
- 2) Sketch in another segment CD off to the side somewhere, taking care that the length of CD is at least half of that of AB.
- 3) Use the "Circle by Center and Radius" construction to build circles centered at A and B whose radius is the same size as CD.
- 4) Select both circles and use the "Point at Intersection" construction to mark where the circles cross. Use the "Line" construction to sketch in the line through those points.
- 5) The line you just created is the perpendicular bisector of AB.

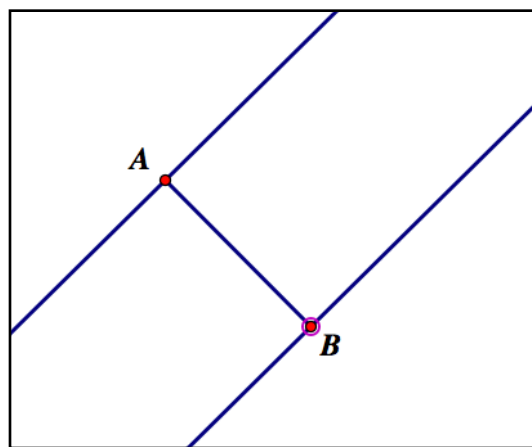
Notice the direct parallel between the physical construction and the one in Sketchpad. This is one of Sketchpad's major strengths – the ability to easily translate these "classic" geometric constructions into the virtual world of the computer. Try dragging around one end of the line segment AB, and watch the perpendicular bisector and the circles automatically change to match what you are doing. Try making the line segment AB more than twice as large as the segment CD you used as the radii – the two circles will no longer touch, and the bisector will vanish.

The shorter method for this construction using Sketchpad uses the literal definition of perpendicular bisector – the line through the midpoint of the segment which is perpendicular to it:

- 1) Sketch in the segment AB with the Segment tool.
- 2) Select the segment and use the “Point at Midpoint” construction to make the midpoint.
- 3) Select the segment AB and the midpoint, and use the “Perpendicular Line” construction.
- 4) The newly created line is the perpendicular bisector.

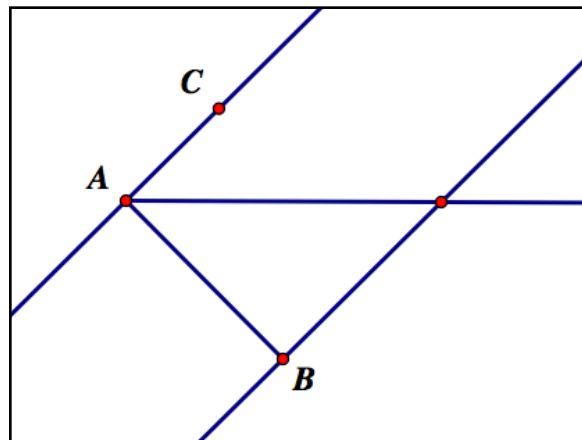
This approach is quicker – it takes fewer steps and the steps themselves are less involved. You can actually try both constructions on the same screen using the same base segment AB and see that the lines created by the two methods are the same.

As another example of a more complex construction, consider the problem of constructing a square from a given line segment AB (in other words, a square which has AB as a side). The best way to approach this is to think of how you might construct it using a regular compass and straightedge, and then try to translate the actions into Sketchpad constructions. To identify the square all you need to do is find its 4 corners – and you already have two of them (the endpoints A and B). Since a square has 4 right angles, the sides of the square must be perpendicular to the original segment AB. So you can create the perpendiculars at both A and B using the “Perpendicular Line” construction, which would give a picture like this:



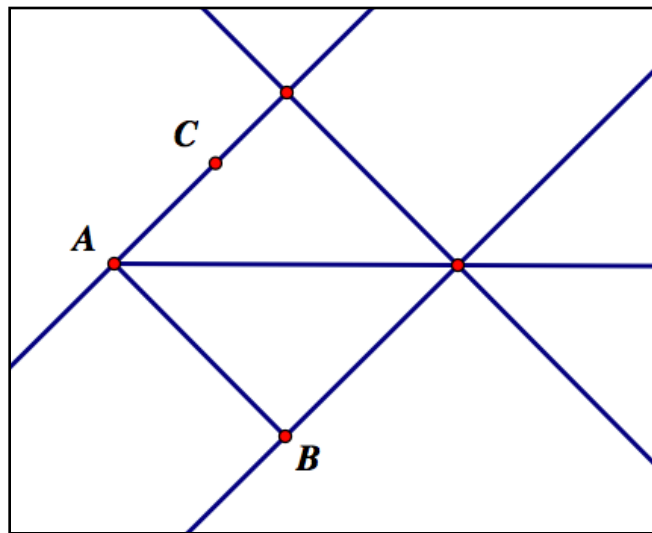
*perpendiculars created at A and B*

The question is, how do you locate the points on these perpendiculars which are the missing corners? There are several different ways you can approach this. One way is to use the fact that the diagonals of the square bisect the angles of the square. So if you create the angle bisector of one of the right angles, where the bisector meets the perpendicular will be a third corner of the square. This is illustrated in the following picture.



*bisecting a right angle to find a corner*

To find the last corner of the square, you could bisect the other right angle or create a perpendicular line at the newly discovered third corner.



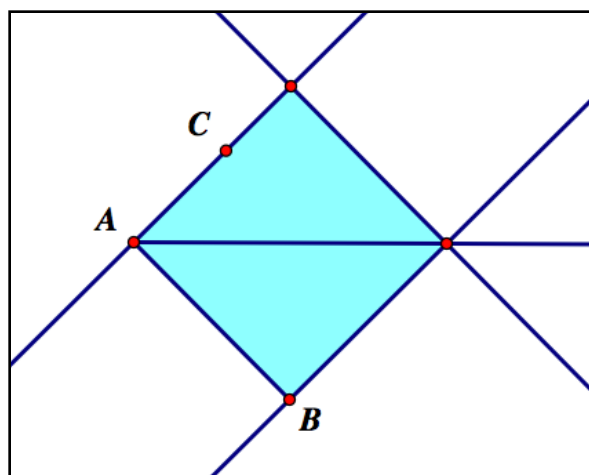
*locating the fourth corner with a perpendicular line*

How would you translate these steps into Sketchpad constructions?

- 1) Using the Segment tool, draw in the original segment AB.
- 2) Select the endpoints A and B as well as the segment AB, and then use the “Perpendicular Line” construction. By selecting both A and B as parents you will create the two perpendiculars at once (you could do each one separately if you wanted to).
- 3) Select the perpendicular at A and use the “Point on an Object” construction (call this point C for reference). Slide this point so it is on the same side of the segment AB you

- want your square to be on. The points C, A, and B in that order define one of the right angles of the square-to-be.
- 4) Select the points C, A, and B (in that order) and use the “Angle Bisector” construction to create a diagonal of the square. This diagonal will cross the perpendicular through B created in step 2.
  - 5) Select the angle bisector and the perpendicular through B. Use the “Point at Intersection” to create a new point (say D) which is a third corner of the square.
  - 6) Select the new point D and the perpendicular through B. Use the “Perpendicular Line” construction to create a perpendicular line through the point D. This line will close off the square shape.
  - 7) Select the line you just made and the perpendicular through A. Use the “Point at Intersection” construction to create the fourth (and last!) corner of the square. Call this point E.
  - 8) Select the four corners A, B, D, and E in that order. Use the “Polygon Interior” construction to create the square itself.

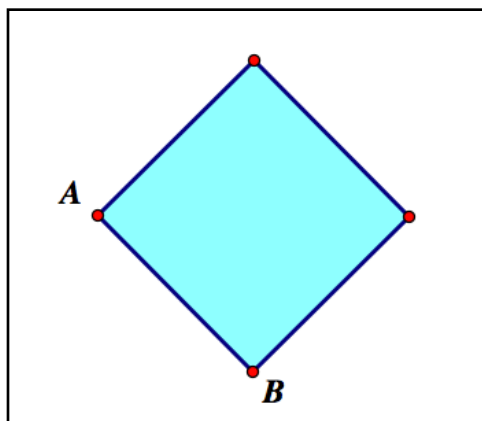
Using this method, your result should look like this:



*the square, complete with polygonal interior*

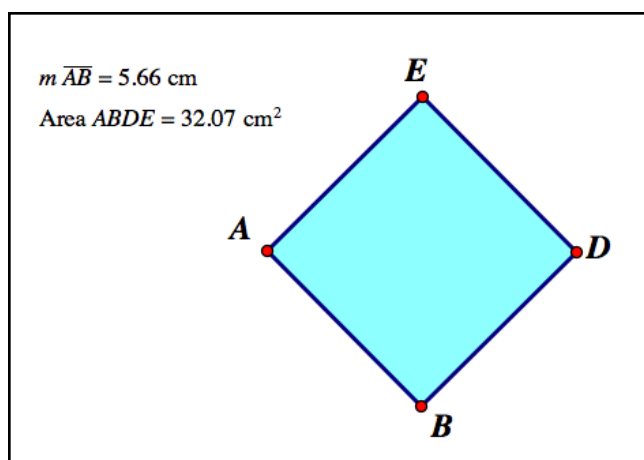
Notice how the picture looks pretty cluttered? This is because of all the lines and points we needed for the individual steps in making the square. If you don't want to show all the intermediate parts of the construction you can clean up this picture quite a bit by “hiding” the lines and points that are not part of the square. You do this by selecting the object you wish to hide and then selecting the “Hide” command from the **Display** menu. After constructing segments for the sides of the square (since we will be hiding the lines that currently form the border on three sides) and hiding all the lines we made in the construction, the labels, and the extra point C, the sketch now looks like this:





*the square and its interior with extraneous elements hidden*

This picture is much simpler, cleaner, and is much easier to use in investigations. Try dragging one of the corners A or B of the square – since all of the elements of our square are children of these points, the entire square will change to accommodate your dragging. You can use this to investigate things like the formula for the area of a square. To do this, select the segment AB and measure its length. Then select the polygon's interior and measure its area. Now you can compare the length of AB with the area directly.



*a quick comparison of length vs. area*

Even better, as you drag either A or B both of these measurements will automatically update. Using these measurements together with Tabulate gives you a very simple way to generate data to investigate the relationship between area and side length for a square. In this case, a few simple measurements would be enough to discover the relationship  $\text{Area} = (\text{side})^2$  (allowing for some roundoff error, of course). It is this type of investigation that is Sketchpad's forte, and we will look at more examples of this in the next section.

## Section 1.3 Homework – Advanced Constructions

- 1) How do you select multiple objects in Sketchpad?
- 2) Construct the line through (2,3) and (5,6). What is its equation? Construct the line through (4,3) and (6,-1). What is its equation? Construct the point of intersection of these lines and find its coordinates.
- 3) Construct the line segment from (1,2) to (7,8). What are the coordinates of its midpoint?
- 4) Build the perpendicular bisector of the previous segment. Construct the circle whose center is the midpoint and whose radius is given by either endpoint. Construct the intersection points of the perpendicular bisector and the circle and measure their coordinates.
- 5) Construct the line through the points (-1,5) and (3,3). Construct the line through (-4,0) which is parallel to this line. What is the equation of the parallel line?
- 6) Let A be the point (0,0), B the point (2,4), and C the point (3,1). Construct the arc ABC. Measure the arc's length and arc angle.
- 7) With the above points, is the arc ABC the same as the arc BCA? What does this tell you?
- 8) If A is the point (0,0), B is the point (4,0), C the point (5,1), and D the point (2,2), find the perimeter and area of the polygon ABCD.
- 9) For the same points as in problem 8, find the perimeter and area of the polygon ACBD.
- 10) Let A be the point (-1,2), B the point (1,4), and C the point (3,1). Sketch in the angle ABC with segments and find its measure. Construct the bisector of this angle. Make a point D on the bisector in the interior of the angle, and find the measure of the angle ABD. Mark the two parts of the angle appropriately.
- 11) Create a circle of radius 3 centered at (2,1). Find its area, circumference, and equation. Translate the circle 5 units to the right and 2 units up. Find the area, circumference, and equation of this new circle. Explain how you did the translation.
- 12) Create the segment from (-3,-1) to (1,2). Create the square above this line segment. Find its perimeter, area, and the coordinates of its corners. Mark the sides and angles of the square appropriately.
- 13) Create the circle centered at (1,1) with a radial point at (1,3). Mark the point (2,0) as a center for transformation. Create new circles via dilations with ratios of 2, 3, 1/2, -1, and -2, and find their equations.
- 14) Create a circle centered at (1,3) with a radial point at (2,2). Reflect this circle about the line  $y=2x$  and measure its equation. (hint: to graph the line, just determine 2 points on it).
- 15) Create 3 points and use these to create a triangular interior. Mark a fourth point as a center for transformations. Drag the interior with dilate tool (the rightmost in the selection pop-up menu), and describe the effects.
- 16) Repeat problem 16 but use the Rotate tool, which is in the middle of the popup menu.
- 17) Create the circle with center (3,2) and radial point (1,1). Rotate this circle 30 degrees counterclockwise about the point (2,6), and find its equation.
- 18) Graph  $\sin(x)$  and  $\cos(x)$  on the same set of axes.
- 19) Graph  $y = x^2$  and the unit circle together. Find the coordinates of their intersection points. (Hint: graph the unit circle geometrically as a circle of radius 1 centered at the origin).

- 20) Let  $A=(-3,4)$  and  $B=(7,1)$ . Plot the value .9 on the segment AB, and find the coordinates of that point.
- 21) Create parametric graph whose first coordinate is given by  $10\sin(3x)$  and whose y coordinate is  $5\cos(2x)$ . Make sure you use a large enough domain to see the entire curve.
- 22) Create the graph of  $y = x^2 - 4x + 1$ . Estimate the minimum distance from the point (1,3) to this graph by letting P be a point on the graph, finding the coordinate distance from P to (1,3), and sliding P across the curve to get a minimum distance.
- 23) Create the parameters m and b with initial values -2 and 5 respectively. Plot the function  $y = mx + b$ . Change the values for m and b, and see what effect they have on the curve.

## Section 1.4 – Complex Constructions and Investigations

In this section we will look at more examples of complex constructions in Sketchpad which can be used to illustrate or investigate many of the ideas and formulae which you may encounter in geometry. The key to making these constructions is to take things one step at a time. The main parts in figuring out these constructions are:

- 1) Read the construction thoroughly, making sure you understand all of its parts and what (if anything) you are trying to measure.
- 2) Write down how you would approach the problem in geometry (bisect this angle, create a point where these lines cross, and so on).
- 3) Translate your geometric steps into Sketchpad steps. In many cases a single geometry step may require several Sketchpad steps. This is very common when dealing with angles, where you may have to define extra points to determine the angle for other constructions (as in the case of creating the angle bisector in the square construction in Section 1.3).
- 4) Follow your Sketchpad outline, and take any measurements that are needed at the end. If you are truly investigating something, dragging one of the parent objects for your construction will give you more data.

If you follow these steps, most of the constructions and investigations you do in Sketchpad shouldn't cause too much trouble. Let's take a look at some examples:

Example 1: Three non-collinear points determine a circle.

One of the standard problems in geometry is finding the circle which runs through three distinct non-collinear points (if the three points are collinear, there won't be a circle going through them). The geometric approach to finding the circle uses the fact that the perpendicular bisector of a circle's chord must go through the circle's center. The construction steps run something like this:

- 1) Determine the three points A, B, and C.
- 2) Form the segments AB and BC. Since A, B, and C are non-collinear, B will actually be the corner of an angle.
- 3) Form the perpendicular bisectors of the segments AB and BC.
- 4) Locate the unique intersection point D of the two bisectors. This is the circle's center.
- 5) Using the distance from D back to any of the three points A, B, or C to give a radius, sketch in the circle centered at D. This is the only circle through A, B, and C.

Now comes the task of translating the steps into Sketchpad:

- 1) Use the Point tool to create points A, B, and C.

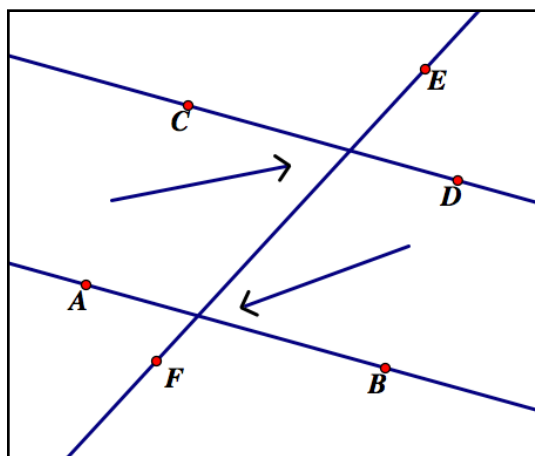
- 2) Select the points A and B, then use the “Segment” construction to make the segment AB. Repeat for the points B and C.
- 3) Select the segment AB. Use the “Point at Midpoint” construction to bisect it. Then select the midpoint and the segment and use the “Perpendicular Line” construction to create the perpendicular bisector.
- 4) Repeat step 3 for the segment BC to make its perpendicular bisector.
- 5) Select the two bisectors from steps 3 and 4, then use the “Point at Intersection” construction to create a point D which is the center of the circle.
- 6) Select the point D first and then any of the points A, B, and C. Use the “Circle by Center and Point” construction to make the unique circle through A, B, and C.

Notice that some of the steps from geometry (“create the perpendicular bisector”) require multiple constructions in Sketchpad (“create the midpoint, then the perpendicular line using the midpoint and the segment”). This is quite common when dealing with bisectors (of segments or angles). Try dragging one of three points A, B or C around and watch your constructed circle adjust to match. What happens if you drag the three points so they are more or less collinear?

Not only does this illustrate the process of finding a circle geometrically, but can give you some insight into finding it algebraically as well. Think of the process of finding the equations of the bisectors and the intersection point. Using this technique in algebra makes it significantly easier to find the equation of a circle (as opposed to the “plug the 3 points into the general equation of a circle and solve the resultant 3 by 3 system of equations”, which is often a lot messier).

#### Example 2: Investigating Angles on Opposite sides of a Transversal

Suppose that AB and CD are parallel lines, and EF is another line which crosses AB between A and B and CD between C and D. What is the relationship (if any) is there between the interior angles on opposite sides of the line EF (EF is called the transversal)?



*What is the relationship between these angles?*

Since this is an investigation and not a construction, we just need to create a picture in Sketchpad similar to the one above and measure the two angles. This can be done using the following sequence of steps:

- 1) Sketch in the line AB with the Line tool.
- 2) Create a third point C not on AB, and then use the “Parallel Line” construction to create the parallel line in the picture.
- 3) Select the line through C and use the “Point on Object” construction to make the fourth point D. If you want, you can slide D around on the line to match it with the picture.
- 4) Use the line tool to create the line EF, taking care that it passes between the points on both lines.
- 5) Mark the two angles indicated in the picture (this will automatically create the appropriate intersection points).
- 6) Measure the two angles to begin your investigation.

As is, this will measure those two angles from your picture, which doesn’t exactly give you a lot of data for your investigation. So select the two measurements, use Tabulate from the **Number** menu, and create a table of several measurements.

Once you’ve figured out the relationship between the two angles, move the line EF so that is entirely to one side of the points A, B, C, and D. Does this change the relationship between the angles? If so, can you find the new relationship?

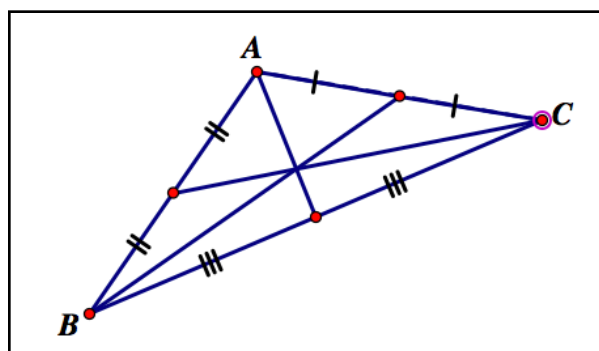
### Example 3: Centroid of a Triangle

Take any triangle ABC in the plane. Since its three sides are line segments, they all have midpoints. Sketch in the segment from each corner of the triangle to the midpoint of the opposite side (these segments are called “medians” of the triangle). Is there anything special about these three segments?

Since this is an investigation rather than a construction, we only have to create the triangle and the three segments in Sketchpad. You could do this with the following steps:

- 1) Using the Point tool, create the points A, B, and C.
- 2) Select all three points, and use the “Segment” construction. This will create all three sides of the triangle at once.
- 3) All three segments created in the previous step should still be selected. Use the “Point at Midpoint” construction to create all three midpoints simultaneously.
- 4) Select the point A and the midpoint of BC, and use the “Segment” construction to create the line joining A and the midpoint.

- 5) Repeat step 4 for the points B and C (and sides AC and AB respectively) to complete the setup.
- 6) For completeness, place congruency marks on each bisected side.



*the centroid of a triangle*

Once you've completed the construction, you will most likely have noted that the three segments all cross at the same point. Drag one of the corners A, B, or C around. Even as you deform the triangle, all three segments will run through the same point. As it turns out this is always true in a triangle, and this special point where the three segments coincide is called the "centroid" of the triangle.

#### Example 4 – Constructing an Equilateral Triangle

The goal here is to construct an equilateral triangle off of any given segment AB. As this is a construction rather than an investigation, it is best to start with the physical construction and then translate the steps over to Sketchpad. The geometric construction usually runs something like this:

- 1) Sketch the base segment AB.
- 2) Construct a circle at A whose radius is AB.
- 3) Construct a circle at B whose radius is AB.
- 4) A and B together with either intersection point of the two circles are the corners of an equilateral triangle.

These steps translate into Sketchpad as follows:

- 1) Use the Segment tool to create the segment AB.
- 2) Select A and the segment AB and use the "Circle by Center and Radius" construction to create the first circle.
- 3) Select B and the segment AB and use the "Circle by Center and Radius" construction to create the second circle.
- 4) Select the two circles and use the "Point at Intersection" construction. This will give two points, one on each side of the segment AB.

- 5) Select one of the intersection points together with A and B. Use the “Segment” construction to create all three sides of the triangle at once.
- 6) Mark all the sides as being congruent, and mark the angles as being congruent.

To verify that your triangle is equilateral, select the three sides and select “Length” from the Measure menu. This will measure all three sides at once, and they should all have the same length. If you drag the point A or B around, the triangle will change but the three side lengths will be the same. If you wanted to measure the area or perimeter of your triangle, you would first have to create the interior of the polygon by selecting all three points and the “Polygon Interior” construction. When dealing with only three points, the selection order doesn’t matter.

## Section 1.4 – Complex Constructions and Investigations

- 1) Draw a circle in Sketchpad. Create a point D on the circle, and create the line through that point which is perpendicular to the radius at D. What is special about this line? Slide the point around the circle to see what happens.
- 2) Draw a circle in Sketchpad, and construct its diameter. Measure the circumference of the circle and the length of its diameter. Find the ratio (circumference)/(diameter length). By dragging the radial point of the circle to change its size, compute the ratio for 4 additional circle sizes (list the diameter length and ratio for each). Make a conjecture about what happens.
- 3) Create a set of geometric and Sketchpad instructions for inscribing a hexagon inside a given circle. (Hint: The length of the hexagon’s side is the same as the radius of the circle).
- 4) Create a set of geometric and Sketchpad instructions for inscribing an octagon inside a given circle that goes not use any transformations from the **Transform** menu. (Hint: If you can make  $45^\circ$  angles at the center of the circle, where those angles intersect the circle at the corners of an octagon. And you can make  $45^\circ$  angles by bisecting easily made angles).
- 5) Construct a pentagon (5 sided polygon) in the plane so that its sides intersect only at its corners. Find the sum of the 5 interior angles of the pentagon. Do this for 4 other pentagons as well (just drag the corners). Find the pattern. Use this pattern to determine how many degrees there would be in each interior angle of a regular pentagon (a regular polygon is one where all the sides and angles are equal).
- 6) Give a set of Sketchpad instructions for creating a parallelogram from a pair of segments AB and AC.
- 7) Create a parallelogram using your instructions from 6 (give the coordinates of the original points A, B, and C). Construct the segments that join opposite corners of the parallelogram. Is it true that the segments bisect each other? Support your answer with measurements from Sketchpad.
- 8) Sketch a triangle and construct the segments from each vertex to the midpoint of the opposite side. What is special about these 3 line segments?
- 9) Sketch a triangle, and create the angle bisectors of each angle. Is there anything special about them?



- 10) Create 3 parameters  $a$ ,  $b$ , and  $c$  with values 1,4,-3. Plot the function  $y = ax^2 + bx + c$ . By changing the values of the 3 parameters, explain what  $a$ ,  $b$ , and  $c$  control in the graph.
- 11) Create the graph of  $y = x + 2\sin(x)$ , a point  $P$  on the graph, and a point  $Q$  not on the graph. Slide  $P$  on the graph until the distance from  $P$  to  $Q$  is a minimum. Construct the perpendicular line to  $PQ$  at  $P$ . What is special about this line? Repeat this for several different points  $Q$  to make sure you have the right idea.

## Section 1.5 – Constructing Loci

In geometry, a locus (plural loci) is a set of points which satisfy some geometric criteria. For example, a circle is a locus because it is the set of all points which are a fixed distance from a fixed point (the fixed point being the center, and the distance being the length of the radius). Many of the classical curves in geometry are defined as loci, including parabolas, circles, ellipses, and hyperbolas. These types of curves can be built in Sketchpad through the use of the “Construct Locus” command found in the **Construct** menu.

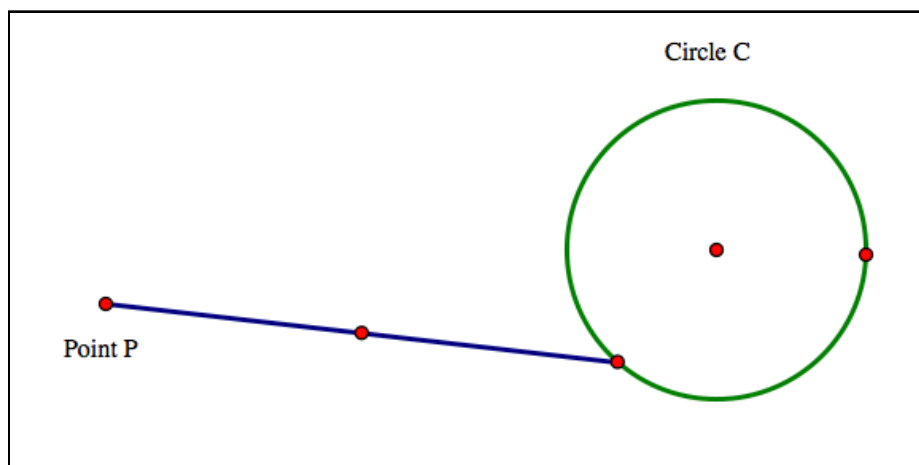
The construct locus command requires 3 things: a path (which can be a segment, ray, line, function plot, circle, or even another locus), a point P which is on the path and can freely slide along it, and some object Q (usually a point) which is a child of P. The Construct Locus command creates all possible objects Q (or at least an approximation to “all” of them) as the point P slides along the path. If the child Q which sketches out the locus is a point, Sketchpad will try to connect the children to form a continuous curve.

As an example, draw a circle C and a point P in the plane. You might ask yourself “Which points in the plane are halfway from the point to a point on the circle?” This would be a locus of points. Your starting picture might look something like this:



*A circle and a point - what points are half-way in-between?*

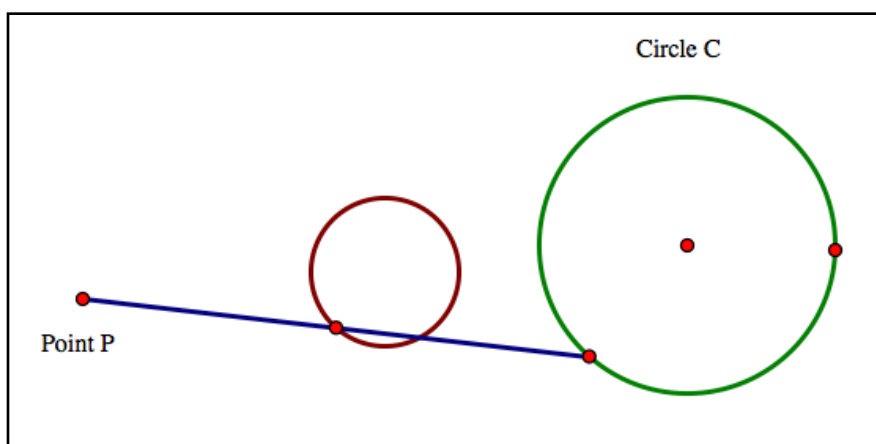
Before we can go any further, we need to determine exactly what is meant by a point halfway between P and C. The easiest definition would be to draw a line segment from P to a point on C, and then take the midpoint of that segment as a “halfway point”. So we would construct a new point on C, construct the segment, and take the midpoint:



*a single point halfway from P to C*

The reason for constructing a new point on C rather than using the radial point already there is that the radial point cannot be moved without changing the circle, whereas a new point is free to slide all the way around the circle without changing it.

We now have a single point “halfway” from P to C. We would now like to find “all” those possible points as the end of the line segment slides all the way around the circle. To do this, select the circle, select the endpoint of the segment on the circle, and then the midpoint (path-free point-child). Then use the Construct Locus command, and you should get this:

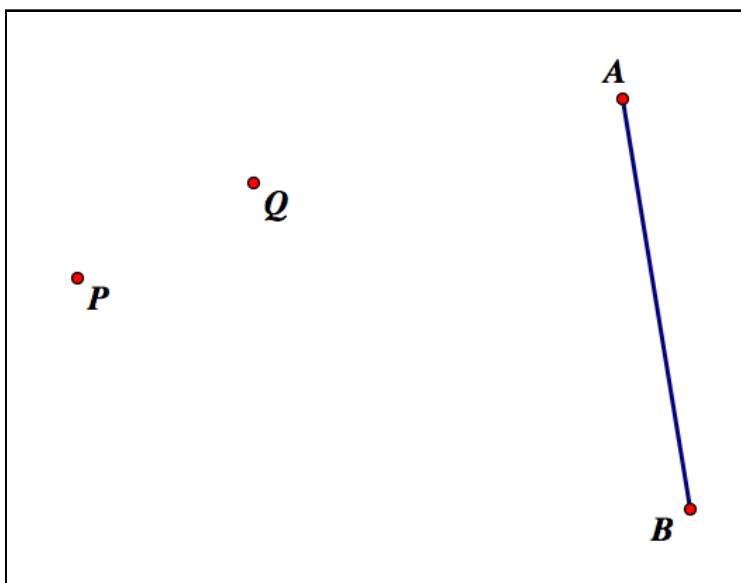


*the locus of all halfway points is another circle*

The highlighted set of points (which Sketchpad has connected into a single curve) is the locus of all points halfway from P to C. In this case, it appears that the locus is actually another circle. If you are not convinced that the locus is a circle, Mark the point P as the center for a dilation and then Dilate the circle C by a ratio of  $1/2$  - this should give the same set of points.

As another example of a curve defined as a locus, consider ellipses. The formal definition of an ellipse is as follows: “Let P and Q be distinct points, and let d be a number larger than the

distance from P to Q. Then the ellipse determined by P, Q, and d is the set of all points R such that the distance  $|PR|$  + the distance  $|QR| = d$ .” In this definition, the points P and Q are called the foci of the ellipse. To define an ellipse using the Locus construction, we will need the parent objects P, Q, and a line segment AB whose length is d. So create a sketch with 2 points and a line segment:

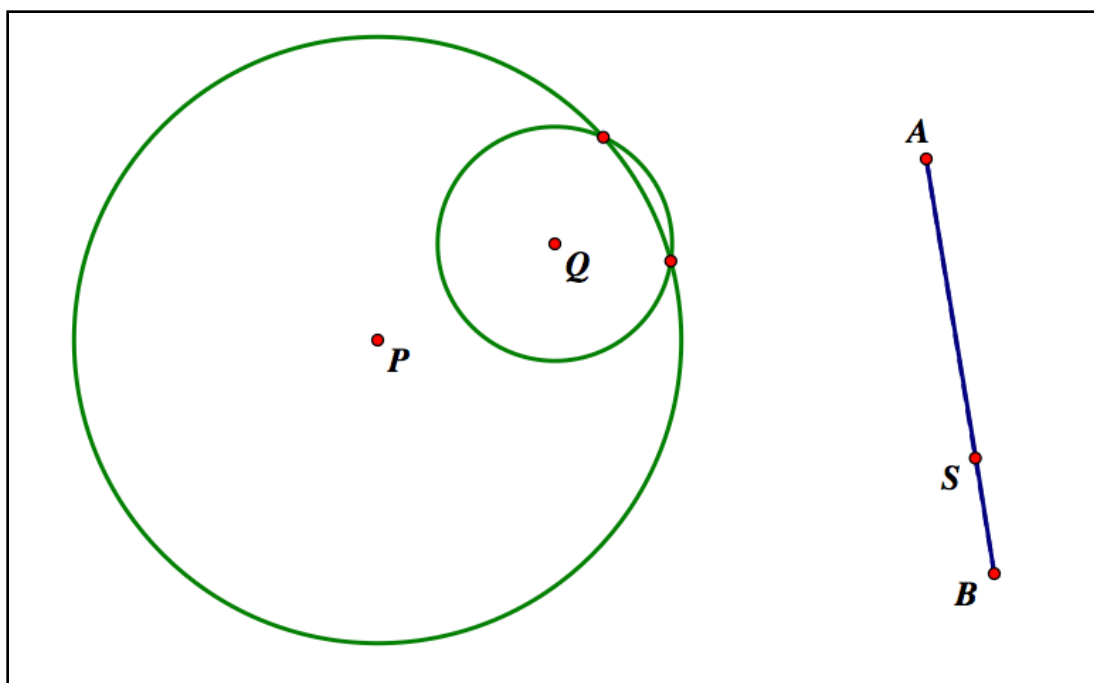


*the parent objects for an ellipse - 2 foci and a length greater than the distance in between*

Next we will need to determine how to find the point(s) R on the ellipse. The distance from R to P plus the distance from R to Q has to be the length of the line segment - so if we split the segment into 2 pieces by placing a point S somewhere in the middle of the segment AB (forming two subsegments AS and SB), the top piece could be the length  $|PR|$  and the bottom the length  $|QR|$ , and the two will automatically add up to d. If R is a distance  $|AS|$  from P, it must be on the circle centered at P with a radius  $|AS|$ . If R is a distance  $|SB|$  from Q, it must be on the circle centered at Q with radius  $|SB|$ . So to find the point(s) R, we can follow this procedure:

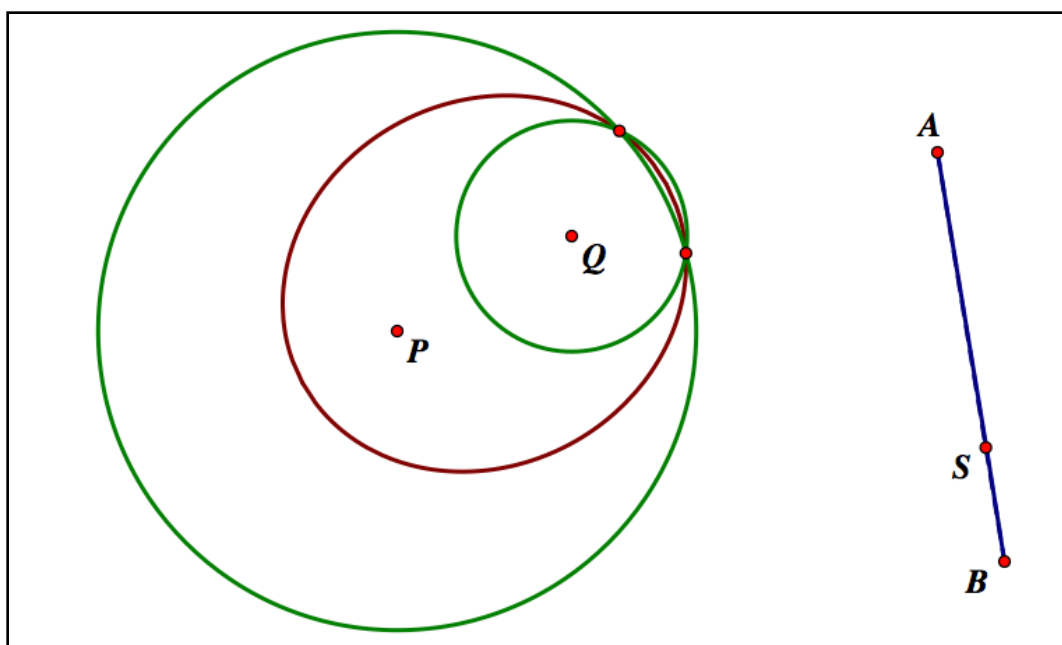
- 1) Sketch the points P and Q and the segment AB.
- 2) Create a new point S on AB.
- 3) Form the segments AS and SB using the Construct Segment command.
- 4) Construct a circle centered at P with radius the segment AS.
- 5) Construct a circle centered at Q with radius the segment SB.
- 6) Locate the point(s) R at the intersection of the two circles using the Construct Intersection command.

At this point, your sketch should look something like this:



*two points  $R$  for which  $|PR| + |RQ| = |AB|$*

To construct the full ellipse, select the full segment  $AB$  (not just one of the parts - this may require multiple clicking to “uncover”  $AB$  as the selection), the point  $S$ , and the one of the points where the circles cross (so the selection is “path”-“free point”-“children”, as before), and the use Construct Locus command. you will get one-half of the ellipse. Repeat for the other intersection point and you will get the whole ellipse:

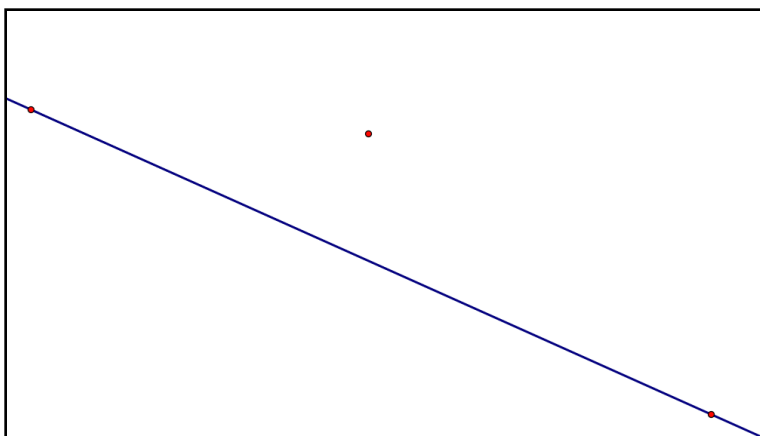


*the full ellipse as a pair of loci*

You can even drag any of the parent points A, B, P, and Q around and the loci will automatically adjust to match. If you move P close to Q, you may notice the ellipse begins to look irregular and have sharp corners. This is an artifact of not plotting enough points (moving P close to Q here is analogous to dividing by numbers close to 0 - small roundoff errors can play a big role). You can alleviate this somewhat by selecting the loci one at a time, right-clicking them to bring up their Properties window, selecting the Plot tab, and increasing the the number of samples (say from 500 to 1000 or 10000).

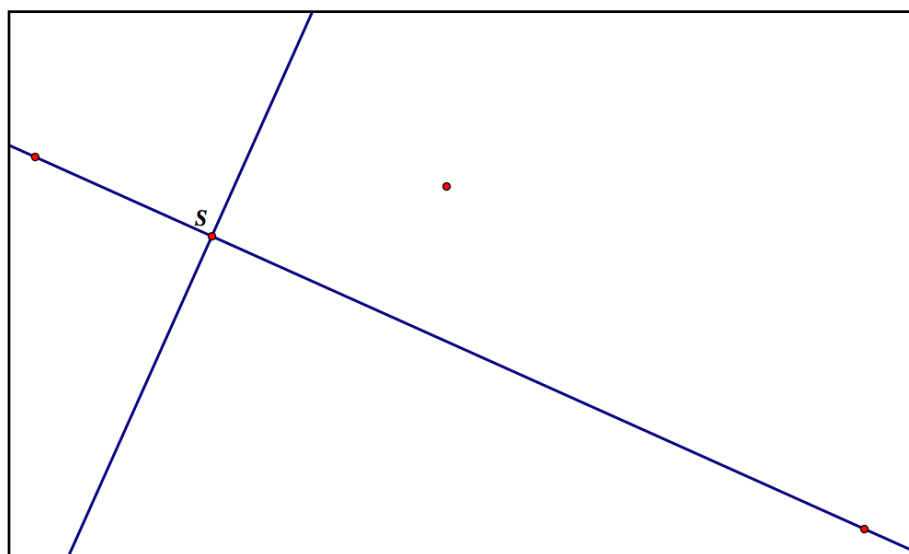
As our last example of the creation of a locus, consider the parabola. The geometric definition of a parabola starts with a point (called the focus) and a line not containing the point (called the directrix). The parabola determined by the point and line are the set of all points R which are the same distance to the point as they are to the line (remember, the distance from a point to a line is measured along the perpendicular).

The first step in constructing a parabola is to create a point P and a line D which does not contain P:



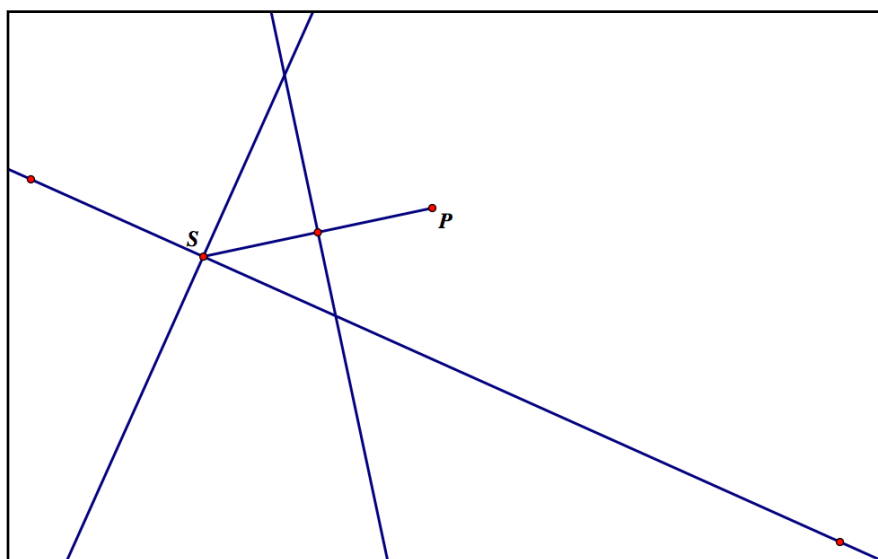
*the directrix and focus P for the parabola-to-be*

Create a free point S on the line, which is the only path available so far. Our goal is to construct a single point R on the parabola created from this free point S, and then we will use the Construct Locus command. The point R on the parabola must be on the line through S which is perpendicular to the directrix (remember, the distance from a point to a line is always measured on the perpendicular). Adding the free point and the perpendicular line, our picture now looks like this:



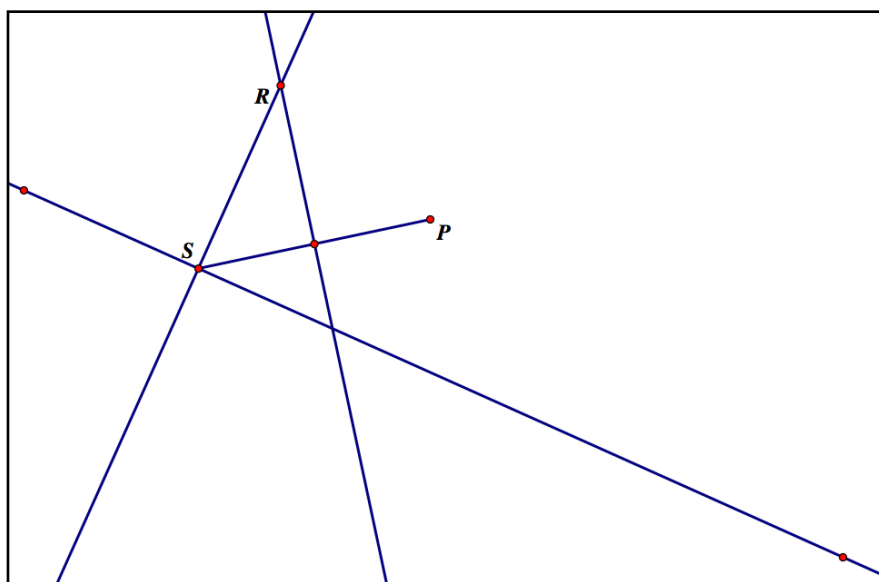
*a free point S on the directrix and the perpendicular through it*

If the point R is the same distance to the focus P as it is to the free point S, it must be on the perpendicular bisector of the segment PS. So sketch this perpendicular bisector in using the Construct Segment, Construct Midpoint, and Construct Perpendicular Line commands:



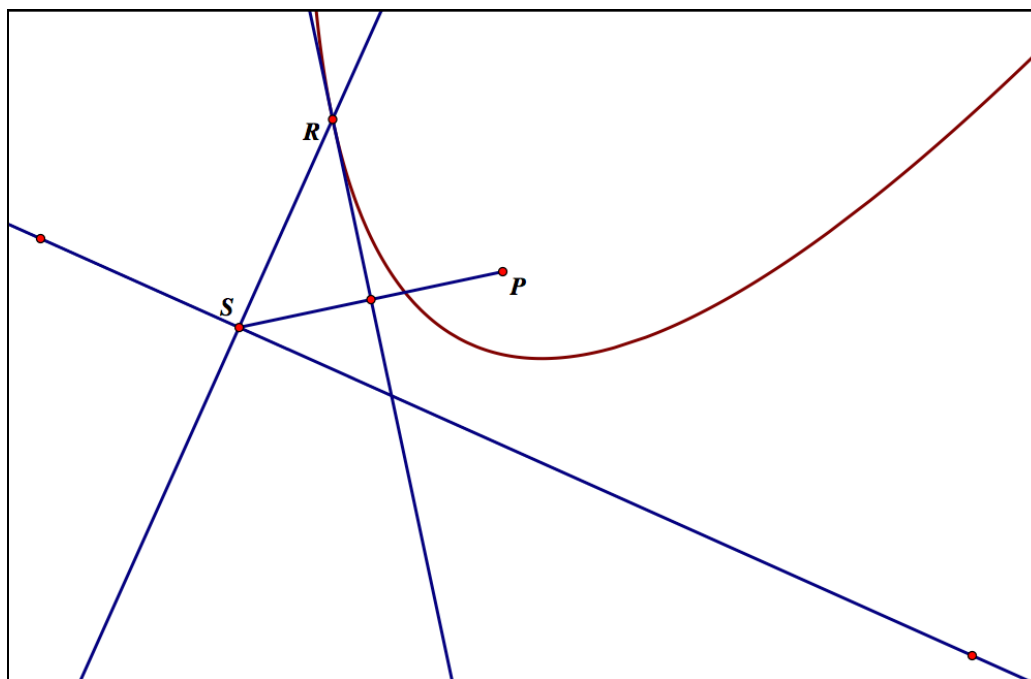
*all points equidistant from P and S are on the perpendicular bisector of PS*

If the point R on the parabola is on the perpendicular line through S and the perpendicular bisector of PS, it must be where the 2 lines cross, so construct the intersection:



*the point R is on the parabola*

To finish the construction of the parabola, select the directrix, then the free point  $S$ , and finally the intersection point  $R$  (path-free point-child), and use Construct Locus:



*the parabola revealed*

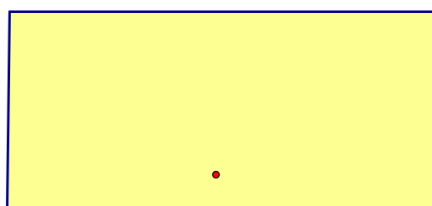
As with the ellipse construction, the parent objects (the focus and directrix) can be moved and the parabola will automatically update.



## Section 1.5 Homework – Constructing Loci

- 1) Repeat the first construction in this section, but rather than selecting the midpoint as the child or the locus, select the line segment. What happens?
- 2) Construct the ellipse with foci  $(1,2)$  and  $(4,0)$  with a “total length” of 5 units. Print out the graph of the ellipse with the coordinate grid shown.
- 3) What happens to an ellipse as the foci move towards one another? (note - as the foci get very close, the ellipse may suddenly deform and get sharp corners - this is a numerical error on the part of Sketchpad analogous to division by 0 and is not what the question is about)
- 4) What happens to the ellipse construction as the foci move farther apart?
- 5) Construct the parabola whose focus is at  $(1,1)$  and whose directrix is the line through  $(0,-3)$  and  $(3,1)$ . Print out the graph with the coordinate grid shown.
- 6) What happens to a parabola as its focus moves towards its directrix? As it moves away?
- 7) Sketch a triangle ABC. Construct the locus of all points R which are the midpoints of segments from A to BC. What is this locus?
- 8) Find the locus of points which are twice as far from  $(1,1)$  as they are from  $(6,3)$ . What kind of object is this locus? (This is tricky. As a hint, you can modify the ellipse approach by using a line rather than a line segment. Let the distance from one fixed point on the line to a free point be the “distance to  $(6,3)$ ”. On the same line you should be able to build a segment twice as long using a circle centered at the free point with the fixed point as a radial point. Then continue with an ellipse-style construction).

Additional Exercise: Take a rectangular sheet of wax paper, roughly 3 times long as it is wide. Designate one of the long sides as the “edge”. Using dark ink, mark a point P centered on this long side about 1/4 of the way up the narrow side of the sheet:



*the “edge”*

Fold the “edge” up so that it passes through the point P, and sharply crease the paper (there are many different ways to do this, each with a different “slope”). Do this to create 30 or 40 different creases. What shape do the creases make on the paper, and how does it relate to one of the constructions in this section?

## Section 1.6 – Custom Tools

As you have worked in Geometer's Sketchpad, there have probably been times where you found yourself the same set of constructions over and over again. For example, many of the constructions we have performed have used the perpendicular bisector of a line segment. While not a difficult set of steps (select segment, Construct Midpoint, select segment and midpoint, Construct Perpendicular Line), you may have found yourself wishing there was a simple tool or construction for creating the perpendicular bisector. Fortunately, Sketchpad allows you to build your own tools via the "Custom Tool" at the bottom of the toolbar.

To create a new tool, first make an example of the desired construction. Then select the construction's parent objects, and whatever part of the construction you want to appear as the final child or result of the construction (this can include multiple objects and marks). Then click on the Custom Tool button, and select "Create New Tool" from the popup menu. A dialog will appear asking you to name the tool (or an error message will appear, which usually means you have not selected all the parent objects of the construction). The tool's name will then appear in the popup menu you will get from the Custom Tool button, and selecting the tool name will allow you to use the tool. The tool you define will be available in all currently open documents. To make the tool permanently available, first select "Choose Tool Folder" from the Custom tool popup menu. This will let you designate a folder in which custom tools can be saved. Then simply save your sketch into the folder you designated.

As our first example of a custom tool, let's create a perpendicular bisector tool. There are two ways to think of the parents of this new tool - either the two points which define the segment, or the segment itself. We will use the segment itself for our construction. To create the tool, follow these steps:

- 1) Sketch in a segment AB.
- 2) Select the segment and use the Construct Midpoint command.
- 3) Select the midpoint and segment and then use the Construct Perpendicular Line command.
- 4) Select the segment and the bisector only, and under the Custom Tools popup menu, select Create New Tool.
- 5) In the dialog box you will see, call the new tool "Create Perpendicular Bisector".

The new bisector tool should now appear in the Custom Tools popup menu.

To use the tool, create a new segment, then select "Create Perpendicular Bisector". The cursor should change to a white arrow with a black edge to indicate the new tool is operational. Clicking on the segment (which will become red when the cursor is over it) should automatically create the perpendicular bisector. Note that the intersection point does not appear - since we selected only the bisector as the child when we created the tool, only the bisector shows when we

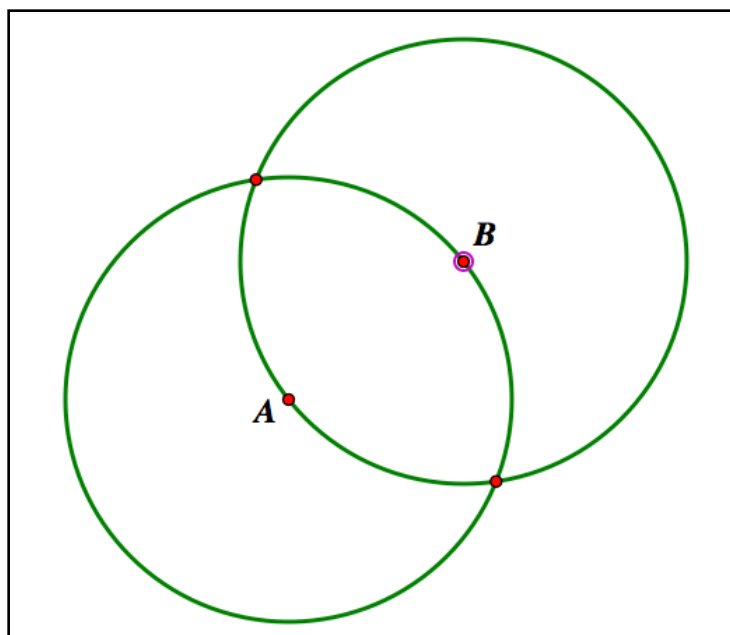
use it. Had we wanted both the bisector and midpoint to appear when the tool is used, both should have been selected when we made the tool.

With the Create Perpendicular Bisector tool still active (again, you can tell by the cursor), click on some empty space in the sketch. A point will appear, and if you drag the cursor it will sketch out both the segment and the bisector. This was why the segment was chosen as the tool's parents rather than the two points - the new tool can be used both on existing segments and as you sketch new ones. One of the first things you need to decide when making a new tool is what the parents should be. In a choice between say a segment and its two endpoints as parents, the segment may give a better tool but the choice of points may make the tool easier to construct - the choice will often depend on what your own preferences are.

As another example of making a new tool, let's create one for building equilateral triangles. The parents for this tool will be two points, and for the children we'll want the third point of the triangle, the sides, and the interior. The name of this tool will be "Right Equilateral Triangle", for reasons that we will see during the construction. To build the tool, follow these steps:

- 1) Sketch in the points A and B, with A the first point you make and B the second.
- 2) Sketch the circle centered at A with radial point B.
- 3) Sketch the circle centered at B with radial point A.
- 4) Construct the intersection points of the circles.

The picture at this point should look something like this:

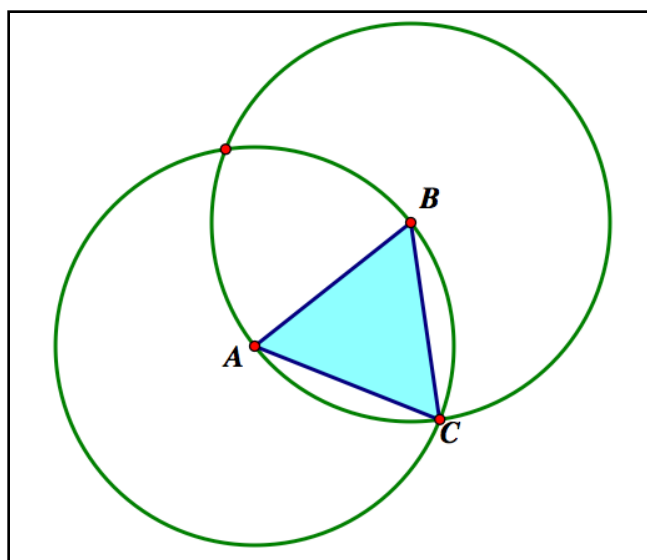


*steps in building an equilateral triangle*

At this point we have two choices for where the third corner of the triangle should be - where the circles intersect. Both are equally valid, but let's choose the point that would be on the right side of the segment as you walk from A to B and call it C. This is why in step 1 we noted the order of the points and why we'll call this the "Right Equilateral Triangle" (the "Left" triangle makes perfect sense as well, but would be a different construction. Continuing the steps, we have:

- 5) Construct the segments AB, AC, and BC.
- 6) Select the points A, B, and C and use the Construct Interior command.
- 7) Select A, B, the segments AB, AC, BC, and the interior.
- 8) Use the Create New Tool command from the Custom Tools button, and name it "Right Equilateral Triangle".

This should give you the following picture:



*the equilateral triangle on the right of AB together with its sides*

Selecting this tool will make it easy to create equilateral triangles for future construction. If you want to save this or any other custom tool for future use, remember to designate a tools folder and then save your sketches to that folder.

## Section 1.6 Homework – Custom Tools

For the homework in this section, use your custom tool to create several examples of the object. In addition, if you click and hold the Custom Tool button, on the resulting menu there will be a choice for "Show Script View". Right-clicking the Script View will let you print out a detailed step-by-step description of your custom tool.

- 1) Create a Left Equilateral Triangle tool (including the appropriate marks on sides and angles).
- 2) Create a Right Square tool (including appropriate marks on sides and angles)

- 3) Create a Parallelogram Tool whose inputs are 3 points (including marks on appropriate sides and angles).
- 4) Create a Circle through 3 points tool.
- 5) Create a Rhombus tool whose inputs are 3 points A, B, C, with AB being one side of the rhombus and the line through BC containing another side.
- 6) Create a second version of the Rhombus tool from problem 5 which includes marks to represent angle and side congruency.
- 7) Create a right hexagon tool.

## Section 1.7 - Straightedge and Compass Constructions (optional)

As we have gone through Geometer's Sketchpad in the previous sections every so often we've referred to "straightedge and compass constructions" and how certain things in Sketchpad relate to them or go beyond them. For this final section we will go back and make these references and distinctions more clear. To do so we will need to go through and develop some of the mathematics behind such constructions as well as the corresponding Sketchpad techniques, so this section will be somewhat more technical than the earlier ones.

First, what do we mean by a "straightedge and compass" construction? In classical Greek geometry (the kind going back to Euclid) people had to be very clear about what kinds of physical tools they were allowed to use in a given construction or demonstration - having different tools available could enable you to do different sorts of constructions or prevent them from being done. The most basic common toolset was the straightedge and compass. A straightedge is simply that - a tool for drawing a line through two given points. The simplest kind of straightedge is an unmarked ruler (the "unmarked" part is important - having marks on a ruler lets you do additional constructions). The original compass was what you might call a "rope compass" - basically a piece of string and a piece of chalk. This would let you draw a circular arc by wrapping the string around the chalk at a given point, pinning a spot on the string with a finger, and then swinging the chalk around. This is much simpler than the modern compass you probably used back in high school. Modern compasses (the kind with a point and pencil) allow you to lift the compass and have it maintain that length without collapsing as you move to a new position on the paper. If you tried to lift a rope compass the chalk would fall out and you would lose the precise spot where the string was held down. A rope compass lets you draw circles but a modern compass lets you draw circles and copy lengths - so it seems like you could do more with a straightedge and modern compass than you could with a straightedge and rope compass. Surprisingly it turns out that the things you can build with a straightedge and rope compass together is exactly the same as what you can build with a straightedge and modern compass (usually the straightedge and rope compass constructions are more involved and have many more steps, but they do let you build the same things). This is why most high school geometry classes use the modern compass even when their ancient counterparts only had rope ones - they let you do the same things, so why not use the tool that's easier?

Once you understand the capabilities of the basic tools you can formally define a straightedge and compass construction: A geometric figure can be constructed by straightedge and compass if you can build it from two starting points (usually the points  $(0,0)$  and  $(1,0)$ ) in a finite number of steps, where each step involves the use of a straightedge, either type of compass, or intersecting two things previously built.

With the notion of a straightedge and compass construction defined in the physical world it is straightforward to bring the idea of a straightedge and compass construction into Sketchpad. We will call a point in Sketchpad a straightedge and compass point (abbreviated as "a SC point")

as reading “straightedge and compass” over and over will quickly get tiring) if it can be built in Sketchpad starting with the points  $(0,0)$  and  $(1,0)$  in a finite number of steps, where each step is uses one of the 4 basic constructions “Construct Line/Segment/Ray”, “Construct Circle by Center + Point”, “Construct Circle by Center + Radius”, and “Construct Intersection” with objects built in previous steps. Note that the first three steps correspond precisely to the basic tools straightedge, rope compass, and modern compass.

A simple example of an SC point is the point  $(2,0)$ . We can build the point  $(2,0)$  in the following sequence of steps:

- 1) Plot the points  $A=(0,0)$  and  $B=(1,0)$ .
- 2) Construct the ray  $AB$  starting at  $A$  through  $B$  (the positive  $x$ -axis)
- 3) Construct the circle  $c_1$  centered at  $B$  through  $A$
- 4) Intersect  $c_1$  and the ray  $AB$ . One intersection will be  $(0,0)$  - the other is the point  $(2,0)$ .

Obviously  $(2,0)$  this is a very simple example but it illustrates the basic idea behind a SC point - you have to show how to build it from the two base points  $(0,0)$  and  $(1,0)$  using only the 4 basic constructions at each step.

Once you have the notion of SC points you can easily define other SC objects:

- 1) A line, ray, or segment is SC if it goes through two SC points.
- 2) A polygon is SC if its vertices are all SC points.
- 3) A circle is SC if its center point is SC and it either goes through a SC point or its radius is equal in length to an SC segment.
- 4) An angle  $ABC$  is SC if the three points that define it are SC.
- 5) An angular measure (like  $60^\circ$ ) is SC if it is the measure of an SC angle.

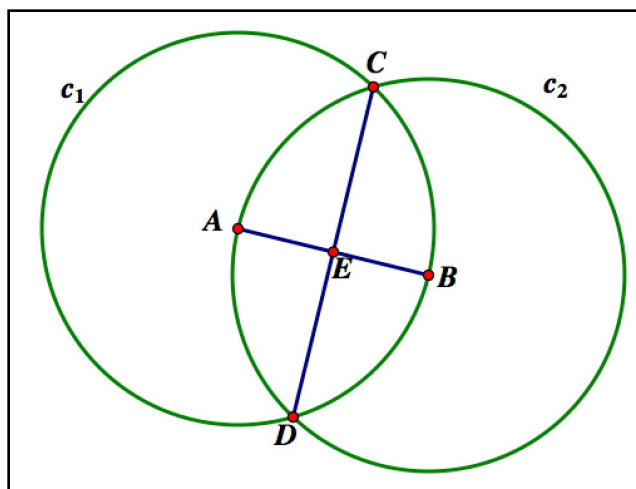
In addition to SC objects we can also deem certain Sketchpad constructions (like Construct Midpoint, etc.) to be SC constructions. A Sketchpad construction is SC if whenever its inputs/parents are SC objects, so is the result/child. That essentially says that a Sketchpad construction is SC if you could duplicate its effects using a sequence of just the four allowable straightedge and compass steps. This is important because if you stick to the basic constructions of “Line/Ray/Segment”, “Circle by Center + Point”, “Circle by Center + Radius”, and “Intersection” most constructions would have dozens of steps and be very complicated. By certifying a construction like “Construct Angle Bisector” as SC that means you can use it in any of your SC constructions as well and that will make many constructions much shorter and easier to understand. The downside is that means you have to go through and show such a construction is SC before you can use it in other SC constructions. So before going any further we need to certify a few of the common Sketchpad constructions as SC.

**SC construction 1:** “Construct Midpoint” is a SC construction and  $60^\circ$  is a SC angle.

This was done in Section 1.3 as an example of translating straightedge and-compass style constructions into Sketchpad but we'll repeat here as an example:

Let A and B be SC points. We find the midpoint of the segment AB as follows:

- 1) Construct the circle  $c_1$  with center A and radial point B.
- 2) Construct the circle  $c_2$  with center B and radial point A.
- 3) Intersect the circles  $c_1$  and  $c_2$  to get points C and D.
- 4) Construct the segment CD.
- 5) Intersect the segments DC and AB to get a point E. E is the midpoint of AB.



*the midpoint of a segment is SC*

Note that in this picture ABC would be an equilateral triangle so all of its angles would measure  $60^\circ$  - therefore without doing any additional work we know that the measure  $60^\circ$  is SC.

**SC construction 2:** “Construct Perpendicular Line” is a SC construction.

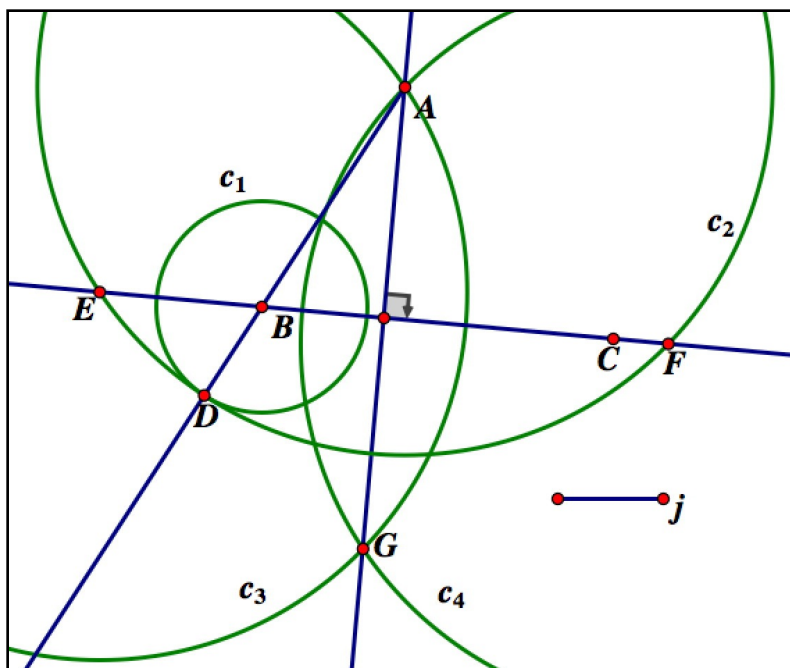
To drop a perpendicular from a SC point A to an SC line BC we need to consider two cases - the base where A is off the line BC and the case where A is on the line BC.

If A is off the line, we can use the following construction:

- 1) Let  $j$  be the segment from  $(0,0)$  to  $(1,0)$ .
- 2) Construct the ray starting at A through B.
- 3) Construct the circle  $c_1$  at B with radius equal to  $j$ .
- 4) Construct the intersection of  $c_1$  with the ray AB. Let D be the point of intersection on the opposite side of the line BC from A.
- 5) Construct the circle  $c_2$  centered at A through the point D.
- 6) Construct the intersections of  $c_2$  and the line BC to get points E and F.



- 7) Construct the circle  $c_3$  centered at E through A.
- 8) Construct the circle  $c_4$  centered at F through A.
- 9) Construct the intersection of  $c_3$  and  $c_4$ . One of these will be A, the other a new point G.
- 10) Construct the line AG. AG runs through A and is perpendicular to BC.

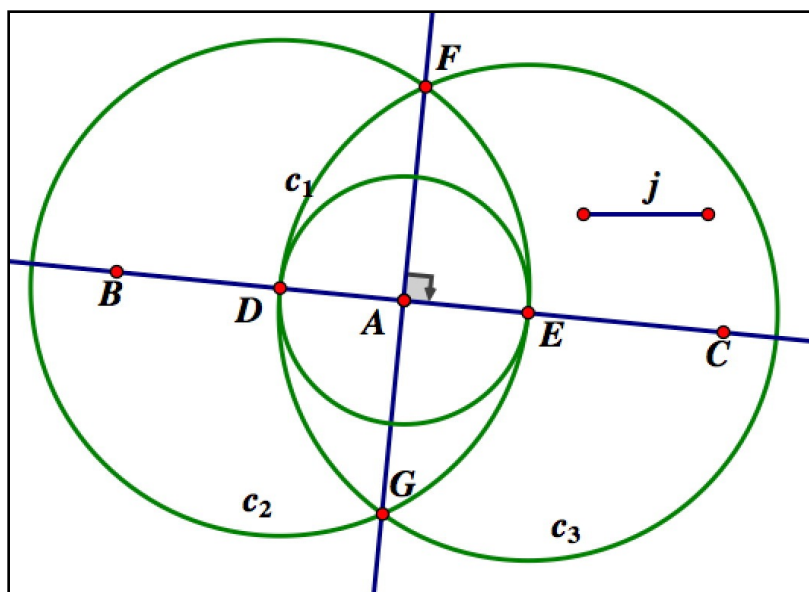


*dropping a perpendicular from A to BC is SC*

If you don't see why you can't just drop steps 3 and 4 and the point D and instead letting the circle  $c_2$  be centered at A and through B, consider what would happen if the line AB by sheer luck was very close to being perpendicular to the line BC. The two intersection points you would get when intersecting  $c_2$  with the line BC would be so close together you might have trouble seeing they were two different points - by building the point D across the line from A and using it to build  $c_2$  we can guarantee we hit the line BC at two points far enough apart to make selecting them easy.

If A is on the line BC, we can use this construction:

- 1) Let  $j$  be the segment through (0,0) and (1,0).
- 2) Construct the circle  $c_1$  centered at A with radius equal to  $j$  (i.e. a circle of radius 1).
- 3) Construct the intersection of  $c_1$  with the line BC - call these points D and E.
- 4) Construct the circle  $c_2$  centered at D through E.
- 5) Construct the circle  $c_3$  centered at E through D.
- 6) Construct the intersections of the circles  $c_2$  and  $c_3$  - call these points F and G.
- 7) Construct the line through F and G. The line FG runs through A and is perpendicular to BC.



*building a perpendicular to the line BC at a point A on the line*

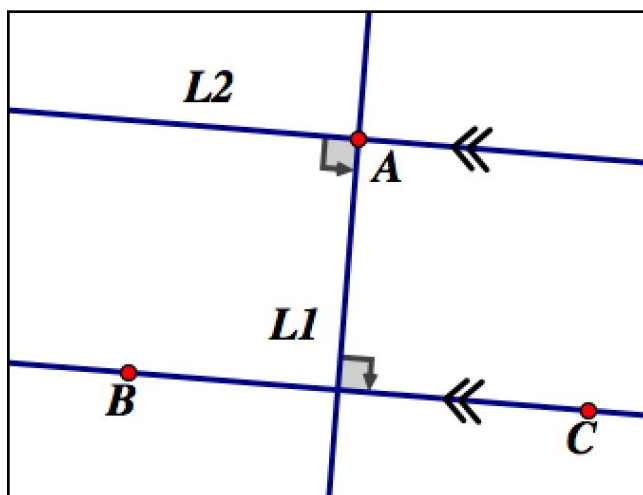
In this second case instead of building the segment  $j$  first you could let  $c_1$  be the circle centered at  $A$  through the farther of  $B$  and  $C$ . But since we don't know which point is farther away in your sketch we can't draw a picture of it - building the segment  $j$  takes the arbitrariness out of the construction and lets everyone start at the same place. An extra advantage of this approach is that we could easily create a custom tool from it if we wanted to.

**SC construction 3:** “Construct Parallel Line” is SC.

For this construction we will avoid the “standard” construction of a parallel line and use one with the newly-known-to-be-SC perpendicular line construction. This perpendicular-based construction is much shorter since we can use the Sketchpad command rather than doing everything from scratch.

Let  $A$  be an SC point and  $BC$  an SC line.

- 1) Construct the line  $L_1$  through  $A$  perpendicular to  $BC$ .
- 2) Construct the line  $L_2$  through  $A$  perpendicular to  $L_1$ .  $L_2$  runs through  $A$  and is parallel to  $BC$ .

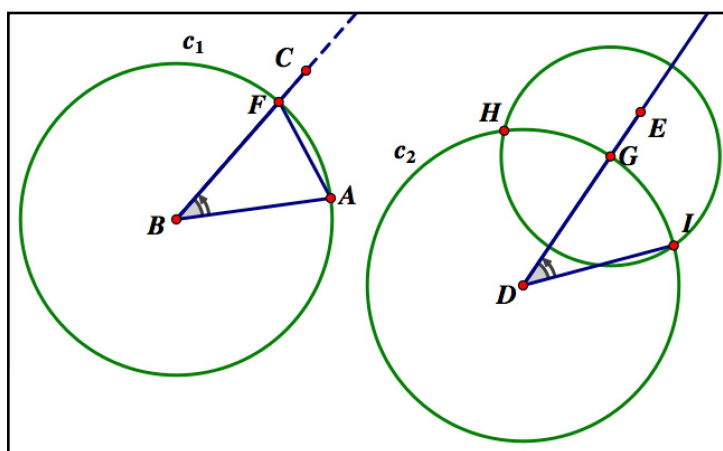


*building a line through A parallel to the line BC is simple if you can build perpendicular lines*

**SC construction 4:** A SC angle can be copied onto a SC segment

Let ABC be a SC angle and DE a SC segment. We can copy ABC onto DE (that is, create an angle congruent to ABC and that has DE as a side) as follows:

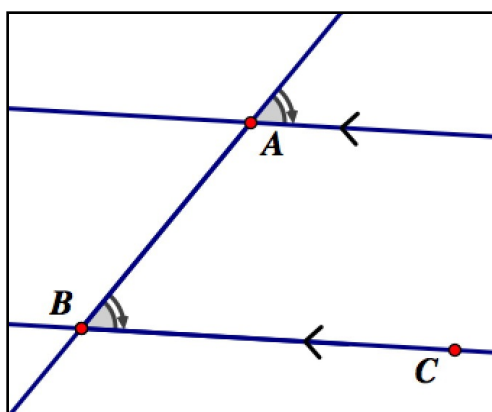
- 1) Construct the ray BC from B through C.
- 2) Construct the circle  $c_1$  centered at B through A.
- 3) Construct the intersection of  $c_1$  and the ray BC - call this point F.
- 4) Construct the segments BF and AF.
- 5) Construct the ray DE.
- 6) Construct the circle  $c_2$  centered at D with radius equal to the segment BA.
- 7) Construct the intersection of  $c_2$  with the ray DE - call this point G.
- 8) Construct the circle  $c_3$  centered at G with radius equal to the segment AF.
- 9) Construct the intersection of  $c_2$  and  $c_3$  - call these points H and I.
- 10) Construct the segment ID. The angle IDE is congruent to ABC and has DE as a side.



*copying the angle ABC into the segment DE*

Note that this construction actually builds two angles congruent to  $ABC$ , one on each side of  $DE$ . Also by building the rays to use for intersections rather than the angle sides already in the picture we don't have to worry about which side of an angle is bigger or shorter - the circles we build are guaranteed to hit the rays (whereas they might have missed shorter angle sides). To see this imagine in the picture above we hadn't built the ray  $DE$ . We could have intersected  $c_2$  with the existing segment  $DE$  to find  $G$  and then continued on with the construction. But once we had finished if we slid the point  $E$  so it was closer to  $D$  than  $G$  is the entire right side of the construction would vanish (as  $DE$  would fall entirely inside  $c_2$  there would be no point  $G$ ). By using rays for the intersections we never have to worry if a segment is too large or too small to use for an intersection.

The “copy an angle” construction is also what is used in the “standard” construction of a parallel line. To make a parallel line to  $BC$  through a point  $A$ , simply make the line  $AB$  and then copy the angle  $ABC$  to  $A$  using  $AB$  as a side (and on the same side of the line  $AB$ ). The copied angle will create the parallel line:



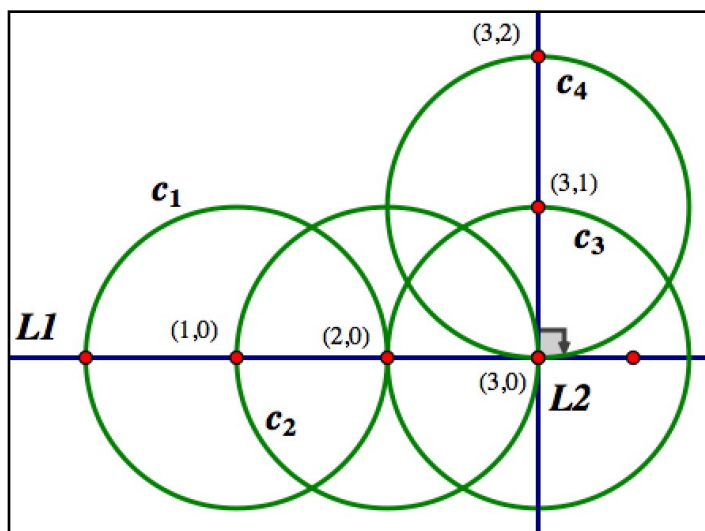
*a simple construction for parallel lines if you can copy angles (steps hidden)*

**SC construction 5:** If  $x$  and  $y$  are integers, then the point  $(x,y)$  is SC. That is the “Plot Points” command is a SC construction when you use integer coordinates.

Rather than give an exhaustive proof here is a demonstration that  $(3,2)$  is an SC point:

- 1) Construct the points  $(0,0)$  and  $(1,0)$ .
- 2) Construct the horizontal line  $L_1$  through  $(0,0)$  and  $(1,0)$  (i.e. the  $x$ -axis).
- 3) Construct the circle  $c_1$  centered at  $(1,0)$  with radial point  $(0,0)$
- 4) Construct the intersection of  $c_1$  and  $L_1$ . This will yield the point  $(2,0)$ .
- 5) Construct the circle  $c_2$  with center  $(2,0)$  and radial point  $(1,0)$ .
- 6) Construct the intersection of  $c_2$  with  $L_1$ . This will yield the point  $(3,0)$ .
- 7) Construct the vertical line  $L_2$  through  $(3,0)$  perpendicular to  $L_1$ .
- 8) Construct the circle  $c_3$  through  $(3,0)$  with radial point  $(2,0)$ .

- 9) Construct the intersection of  $c_3$  with  $L_2$ . This will yield the point  $(3,1)$ .
- 10) Construct the circle  $c_4$  centered at  $(3,1)$  with radial point  $(3,0)$ .
- 11) Construct the intersection of  $c_4$  with  $L_2$ . This will yield  $(3,2)$ .



*the point  $(3,2)$  is an SC point*

Obviously this is tedious and could be shortened (for example we could skip the point  $(3,1)$  entirely by using a circle centered at  $(3,0)$  with radial point  $(1,0)$ ) but the way this construction is set up hopefully makes it easy to see that this process could be extended/modified to locate any point which is left/right and up/down whole number distances from  $(0,0)$  - that is, it could be used to construct any point with integer coordinates. The ability to use “Plot Points” can be used to shorten several SC constructions (such as the next one).

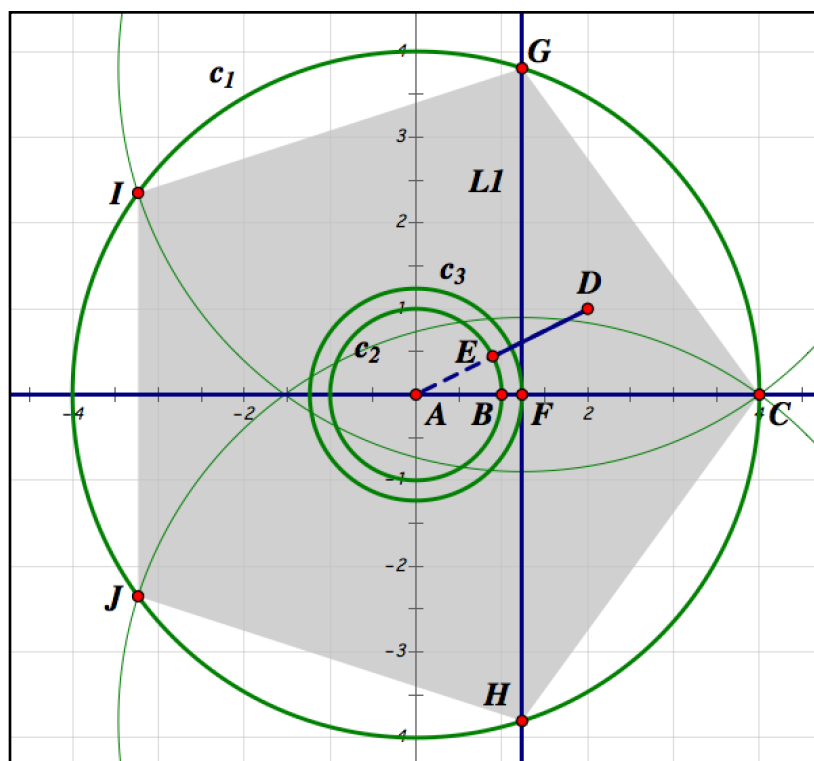
Now that we have several Sketchpad construction certified as being straightedge and compass, here is a less obvious construction that can done be using the basic ones we’ve already certified as SC:

**SC Construction 6:** The regular pentagon centered at  $(0,0)$  with a corner at  $(4,0)$  is an SC polygon.

To do this we will use the well-known fact that the central angle of a regular pentagon is  $360^\circ/5=72^\circ$  and the less well-known fact that  $\cos(72^\circ)=\frac{\sqrt{5}-1}{4}$ .

- 1) Plot the points  $A=(0,0)$ ,  $B=(1,0)$  and  $C=(4,0)$ .
- 2) Construct the circle  $c_1$  centered at  $A$  with radial point  $C$ . This will be the circle we will eventually frame (technically “inscribe”) the pentagon in.
- 3) Plot the point  $D=(2,1)$ .

- 4) Construct the segment AD. By the pythagorean theorem this segment has length  $\sqrt{(2-0)^2 + (1-0)^2} = \sqrt{4+1} = \sqrt{5}$ .
- 5) Construct the circle  $c_2$  centered at A with radial point B (the unit circle).
- 6) Construct the intersection of  $c_2$  with the segment AD - call this point E.
- 7) Construct the segment DE. This segment has length  $\sqrt{5} - 1$ .
- 8) Construct the circle  $c_3$  centered at A with radius equal to the segment DE.
- 9) Construct the line AB (the x-axis).
- 10) Construct the intersection of  $c_3$  and the line AB. Call the Quadrant I intersection point F.
- 11) Construct the line L1 through F perpendicular to the line AB.
- 12) Construct the intersection of L1 with  $c_1$ . Call these points G and H.
- 13) Construct the circle  $c_4$  centered at G with radial point C.
- 14) Construct the intersection of  $c_4$  and  $c_1$ . One point will be C; call the other I.
- 15) Construct the circle  $c_5$  centered at H with radial point C.
- 16) Construct the intersection of  $c_5$  with  $c_1$ . One point will be C; call the other J.
- 17) The polygon CGIJH is a regular pentagon.



*this perfectly regular pentagon is SC*

Why does this work? Note that the triangle AFG is a right triangle with hypotenuse 4. The segment AF is a side of AFG triangle with length  $\sqrt{5} - 1$ . For the angle GAF we can think of AF as the “adjacent side” and AG the triangle’s hypotenuse. So the cosine of GAF is “adjacent over

hypotenuse”, or  $\frac{\sqrt{5}-1}{4}$ . As  $\cos(72^\circ) = \frac{\sqrt{5}-1}{4}$  the angle GAF must measure  $72^\circ$  and so must have the right measure to be the central angle of a regular pentagon - the remaining steps just provide the remaining corners.

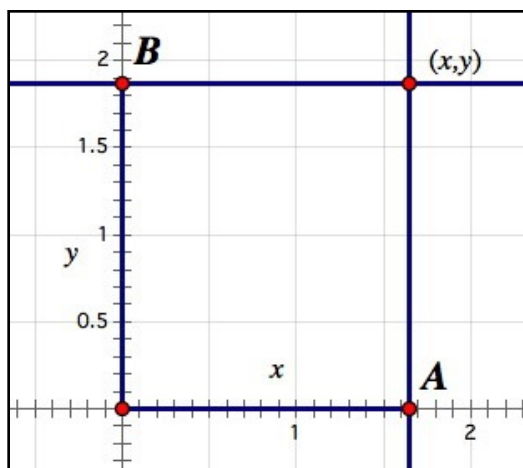
Now you may look at the regular pentagon construction and wonder how anyone knew to try such a convoluted sequence of steps. This is a common problem when learning to do straightedge and compass constructions either on paper or in Sketchpad - while someone can tell you “here is how to construct X using straightedge and compass steps” it doesn’t seem like there’s a good way to know if a construction is SC in the first place. For example is “Construct Angle Bisector” a SC construction? Is a regular nonagon (9-sided) centered at (0,0) with a corner at (1,0) a SC polygon? Or do we need to introduce some tools other than a straightedge and compass to do these? (as it turns out angle bisection is SC but the nonagon is not SC).

To determine whether or not a given construction is SC we need one more basic idea - the notion of a SC number. A real number  $x$  is called a SC number if there is an SC segment of length  $|x|$ . For example the number 1 is SC, as the segment from (0,0) to (1,0) is an SC segment (likewise by construction 5 any integer  $n$  is a SC number since the segment from (0,0) to  $(n,0)$  is SC). The number  $-\sqrt{13}$  is a SC number as the segment from (0,0) to (2,3) is SC and has length  $\sqrt{13} = |-\sqrt{13}|$  (the absolute values in the definition of an SC number mean you don’t have to worry about whether a number is positive or negative when figuring out if it is SC). SC numbers are the key to understanding to what can and can’t be done with just a straightedge and compass because of the following theorem:

Theorem: SC Points and SC Numbers

A point  $(x,y)$  is an SC point if and only if both  $x$  and  $y$  are SC numbers.

To see why this is true consider the following picture:



If the point  $(x,y)$  is SC then we can construct perpendiculars to both the  $x$ -axis and  $y$ -axis to get the points A and B - so A and B are both SC points. The SC segment from the origin to A has length  $|x|$ , so  $x$  must be a SC number. Likewise the SC segment from the origin to B has length  $|y|$ , so  $y$  is a SC number. Therefore if  $(x,y)$  is an SC point we know that  $x$  and  $y$  are SC numbers. Conversely suppose  $x$  and  $y$  are SC numbers. Then there are segments  $j$  and  $k$  where the length of  $j$  is  $|x|$  and the length of  $k$  is  $|y|$ . By constructing a circle centered at the origin with radius  $j$  and intersecting it with the  $x$ -axis we can construct the point A (if  $x > 0$  use the intersection point on the right, if  $x < 0$  take the one on the left). Likewise by constructing the circle centered at the origin with radius  $k$  and intersecting it with the  $y$  axis we can construct the point B (if  $y > 0$  pick the top intersection, if  $y < 0$  pick the bottom one). After building the perpendicular to the  $x$ -axis at A and the perpendicular to the  $y$ -axis at B their intersection will be the point  $(x,y)$  - so the point  $(x,y)$  is SC. Therefore if  $x$  and  $y$  are SC numbers the point  $(x,y)$  is SC - and this finishes the proof of the theorem.

This theorem tells us that instead of worrying about which points are SC we can instead try to find out which numbers are SC. We know every integer  $n$  is a SC number but what about other numbers? It turns out SC numbers work very well with the basic operations of arithmetic and one operation from algebra:

**Theorem: Combining SC numbers**

Suppose  $x$  and  $y$  are SC numbers. Then the following numbers are also SC:

- a)  $x+y$ ,  $x-y$ ,  $xy$ , and  $x/y$  (provided  $y$  is not 0).
- b) If  $x > 0$ , then  $\sqrt{x}$  is SC as well.

We will check these individually and at the same time use Sketchpad's custom tool feature to make these operations easier to do in Sketchpad in the future:

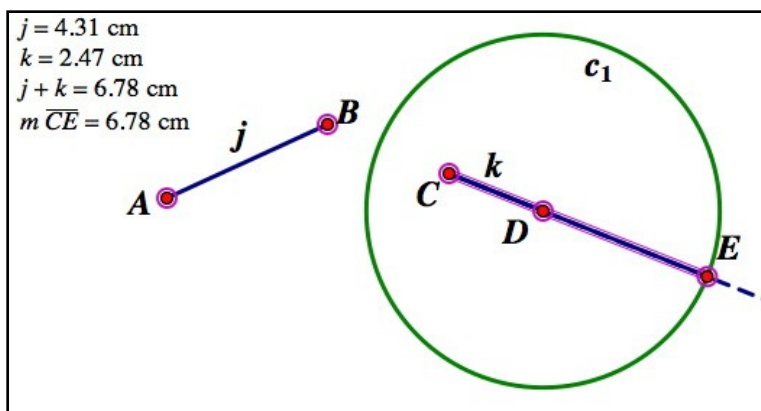
**Custom Tool:** "Add larger segment  $j$  to the smaller segment  $k$ " (i.e. if you have numbers  $x > y > 0$  find  $x+y$ )

Parents: The endpoints A and B of the larger segment  $j$  and the endpoints C and D of the smaller segment  $k$

Child: A segment CE whose length is the length of  $j$  added to the length of  $k$

- 1) Construct the ray CD.
- 2) Construct the circle c1 centered at D whose radius is the larger segment  $j$ .
- 3) Intersect c1 with the ray CD to get a unique intersection point E.
- 4) Construct the segment CE - this is the desired child.
- 5) Select A, B, C, D, E, and the segment CE in order and then create a custom tool.





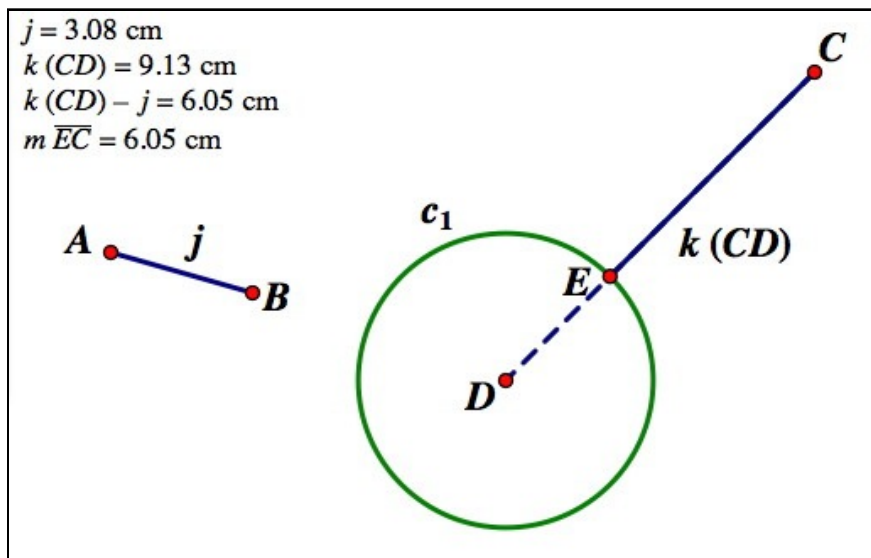
*adding two segments, with sample measurements*

**Custom Tool:** “Subtract smaller segment  $j$  from larger segment  $k$ ” (i.e. if you have  $x > y > 0$  then find  $x - y$ ).

Parents: The endpoints A and B of the smaller segment  $j$  and the endpoints C and D of the larger segment  $k$

Child: A segment inside of  $k$  with C as one endpoint and length  $|k| - |j|$ .

- 1) Construct the circle  $c_1$  centered at D with radius equal to the length of  $j$ .
- 2) Construct the intersection of  $c_1$  with  $k$  - there should be a unique intersection point E.
- 3) Construct the segment CE - this is the desired child.
- 4) Select the points A, B, C, D, E, and the segment CE in order and then create a custom tool.



*subtracting one segment from another, with sample measurements*

The next three tools require you to have a “unit segment” available - i.e. a segment of length 1 (you could use the segment from (0,0) to (1,0) or any arbitrary segment after using it with the

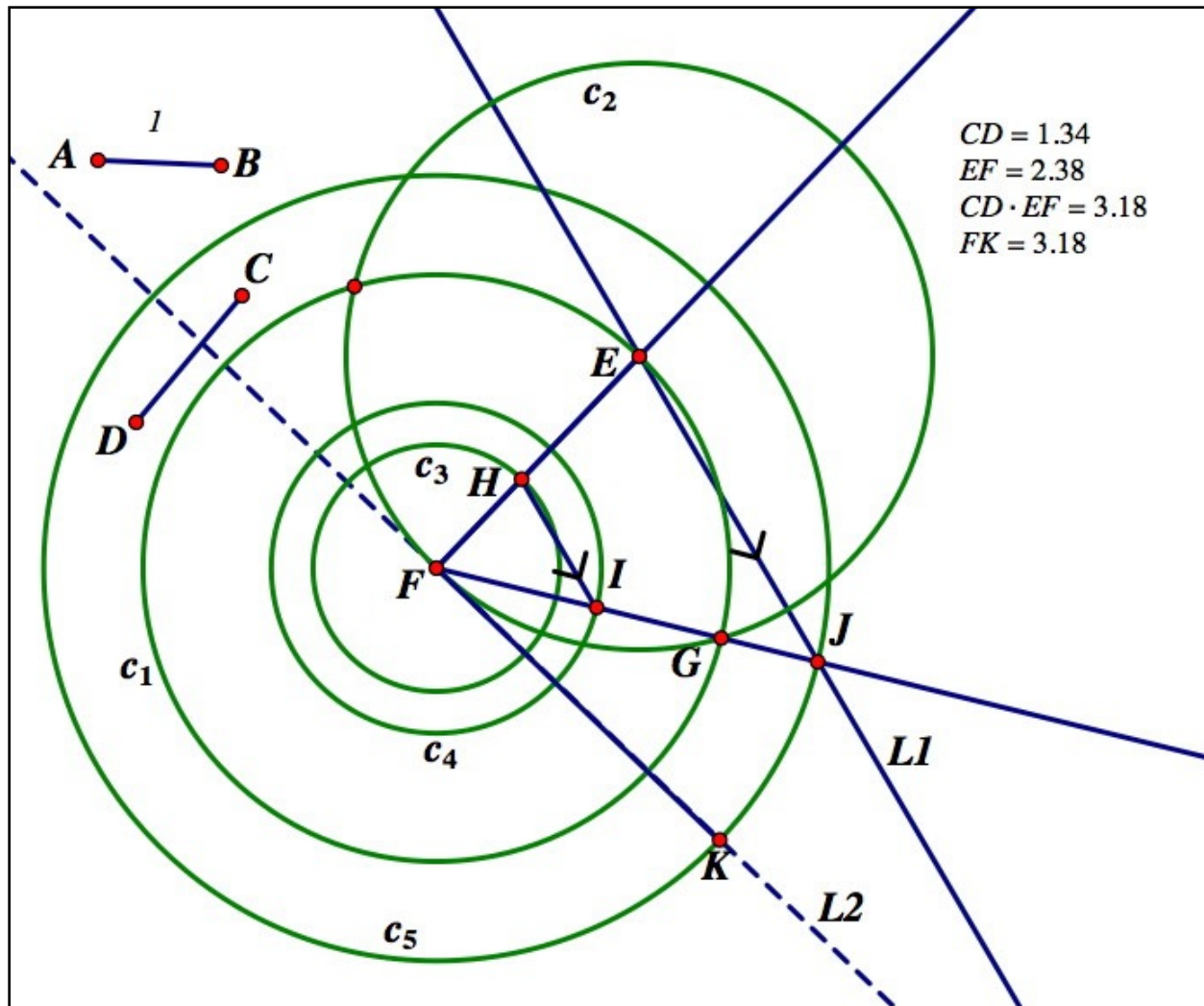
Graph menu's "Define unit distance" command). As these three constructions will all depend on having a predefined scale when you use the custom tools based on them you can no longer measure segment lengths using the "Measure Length" command - instead you will need to select the endpoints and use the "Measure Coordinate Distance" command instead.

**Custom Tool:** "Multiply segments  $j$  and  $k$ " (i.e. if  $x > 0$  and  $y > 0$  find  $xy$ )

Parents: The endpoints A and B of a unit segment, endpoints C and D of a segment  $j$ , and endpoints E and F of a segment  $k$ .

Child: A segment with F as an endpoint, is perpendicular to  $k$ , and has length equal to  $|j| |k|$

- 1) Construct circles c1 and c2 with centers E and F with  $k$  as the radius.
- 2) Construct the intersections of c1 and c2; pick one of them, call it G.
- 3) Construct the ray from F through G.
- 4) Construct the ray from F through E.
- 5) Construct the circle c3 with center F whose radius is the unit length AB.
- 6) Construct the intersection of the circle c3 with the ray FE. Call this point H
- 7) Construct the circle c4 centered at F whose radius is the segment CD.
- 8) Construct the intersection of c4 with the ray FG. Call this point I.
- 9) Construct the segment HI.
- 10) Construct the line L1 through E parallel to GH.
- 11) Construct the intersection of L1 with the ray FG. Call this point J. (the segment FJ has the right length but is not perpendicular to  $k$ )
- 12) Construct the circle centered c5 at F through J.
- 13) Construct the line L2 through F perpendicular to the segment EF.
- 14) Construct the intersections of c5 and L2. Pick one of the points and call it K.
- 15) Construct the segment FK. This segment has length  $|j| |k|$ .
- 16) Select the points A, B, C, D, E, F, K, the segment FK, and create a new tool.



*the complex construction of multiplying segments, with sample measurements*

Why does this construction work? The point G is built so that you have a point guaranteed to be off the segment EF - this makes sure that FHI and FEJ are really triangles and not just a group of overlapping segments. The segment FI has the same length as CD, and FH has length 1. Since HI and EJ are parallel the triangles FHI and FEJ are similar. Therefore the sides of FHI and FEJ are in proportion and we have  $\frac{|FH|}{|EF|} = \frac{|FI|}{|FJ|}$ , or  $|FH| |FJ| = |EF| |FI|$ . Since  $|FI|=|CD|$  and  $|FH|=1$  the equation  $|FH| |FJ| = |EF| |FI|$  becomes  $1 |FJ| = |EF| |CD|$ , or  $|FJ|=|CD| |EF|$ . The circle c5 and the point K are created just to “swivel” the new length so it is perpendicular to EF.

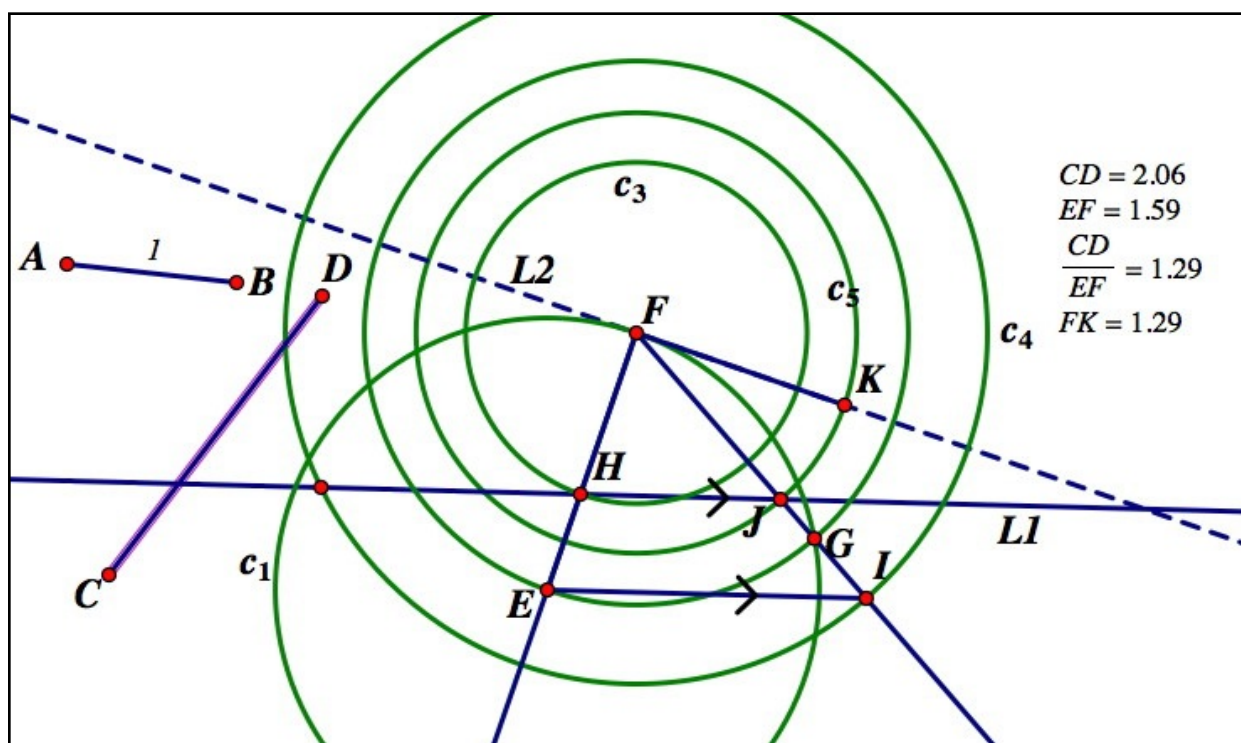
The construction for division works very similar to the one for multiplication - it also relies on a unit segment and although complicated it revolves around the construction of two similar triangles.

**Custom Tool:** “Divide segment  $j$  by segment  $k$ ” (i.e. if  $x,y > 0$  find  $x/y$  ).

Parents: Endpoints A and B of a unit segment, endpoints C and D of segment  $j$ , and endpoints E and H of segment  $k$ .

Child: A segment with F as an endpoint, is perpendicular to  $k$ , and has length equal to  $|j|/|k|$

- 1) Construct circles  $c_1$  and  $c_2$  with centers E and F with  $k$  as the radius.
- 2) Construct the intersections of  $c_1$  and  $c_2$ ; pick one of them, call it G.
- 3) Construct the ray from F through G.
- 4) Construct the ray from F to E.
- 5) Construct the circle  $c_3$  with center F whose radius is the unit length AB.
- 6) Construct the intersection point of the ray FE with  $c_3$  - call it H.
- 7) Construct the circle  $c_4$  whose center is F and whose radius is CD.
- 8) Construct the intersection of  $c_4$  with the the ray FG - call it I.
- 9) Construct the segment EI
- 10) Construct the line  $L_1$  through H parallel to EI.
- 11) Construct the intersection of the ray FG with  $L_1$  - call it J. J has the desired length.
- 12) Let  $L_2$  be the line through F perpendicular to FE.
- 13) Let  $c_5$  be the circle centered at F through J.
- 14) Construct the intersections of  $c_5$  and  $L_2$ . Pick one of them, call it K.
- 15) Construct the segment FK.



*dividing one segment by another, with sample measurements*

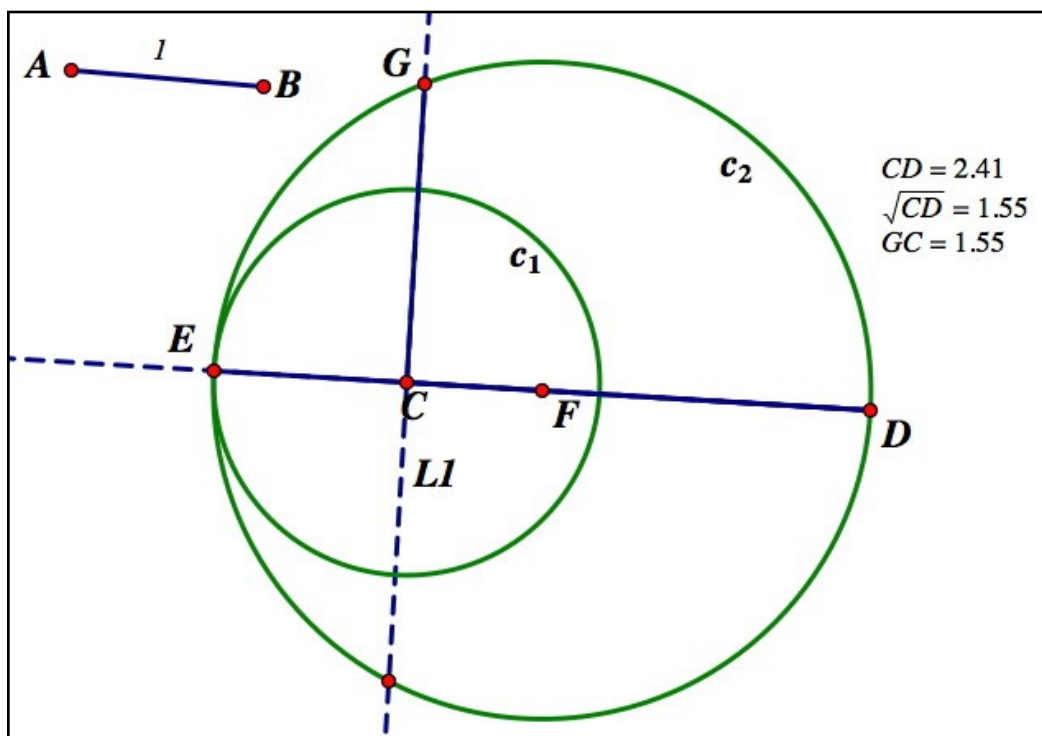
These constructions cover the four basic operations of arithmetic. The construction for taking a square root is less commonly taught at the high school level but is central to straightedge and compass constructions. Like the multiplication and division constructions it uses a unit length and so any length measurements will have to be done via the “Measure Coordinate Distance” command.

**Custom Tool:** “Square root of a segment” (i.e. if  $x > 0$ , find  $\sqrt{x}$ )

Parents: Endpoints A and B of a unit segment and the endpoints C and D of a segment  $j$ .

Child: A segment CG perpendicular to CD whose length is the square root of  $|CD|$ .

- 1) Construct the ray from D through C.
- 2) Construct a circle  $c_1$  centered at C whose radius is equal to the unit segment AB.
- 3) Construct the intersections of  $c_1$  with the ray DC. One of these will be outside the segment CD - choose it and call it E.
- 4) Construct the segment DE.
- 5) Construct the midpoint F of the segment DE.
- 6) Construct the circle  $c_2$  centered at F through D.
- 7) Construct the line  $L_1$  perpendicular to CD at C.
- 8) Construct the intersections of the circle  $L_1$  and the circle  $c_2$ . Pick one, call it G.
- 9) Construct the segment CG (this is the desired segment).
- 10) Select A, B, C, D, G, and the segment CG. Create the new tool.



*taking the square root of a segment length, with sample measurements*

These 5 custom tools prove the theorem that sum, difference, product, ratio, and (real) square root of SC numbers are all SC. We know that the integers are SC numbers, and this gives us the following theorem:

Theorem: Numbers built from the integers using arithmetic and square roots are SC

Let  $x$  be a number which is built up from the integers in a finite sequence of additions, subtractions, multiplications, divisions, and extractions of real square roots. Then  $x$  is a SC number.

As an example of how to use this theorem we can show that the point  $(3 - 4\sqrt{7}, \sqrt{1 + 2\sqrt{11}})$  can be built with straightedge and compass. We know this is true if and only if both coordinates  $3 - 4\sqrt{7}$  and  $\sqrt{1 + 2\sqrt{11}}$  are SC numbers. Starting with the first number, we know 3, 4, and 7 are SC numbers because they are integers. By taking a square root we know  $\sqrt{7}$  is SC. By multiplying 4 and  $\sqrt{7}$  we have that  $4\sqrt{7}$  is SC. By subtracting  $4\sqrt{7}$  from 3 (which is really subtracting 3 from the larger number  $4\sqrt{7}$  and making sure we think of the segment as going to the left rather than the right) we get  $3 - 4\sqrt{7}$  is SC. For the second coordinate we know that the integers 1, 2, and 11 are SC. After taking a square root we have that  $\sqrt{11}$  is SC. By multiplying 2 and  $\sqrt{11}$  we know  $2\sqrt{11}$  is SC. Adding this to 1 gives us that  $1 + 2\sqrt{11}$  is SC. And finally taking another square root gives us that  $\sqrt{1 + 2\sqrt{11}}$  is SC. So the point  $(3 - 4\sqrt{7}, \sqrt{1 + 2\sqrt{11}})$  must be SC. Even better, we could actually build this point if we wanted to by building the two numbers step-by-step using our custom tools. Then we could make a segment in the  $x$ -axis starting at the origin whose length is  $|3 - 4\sqrt{7}| = 4\sqrt{7} - 3$  pointing to the left and one in the  $y$ -axis starting at the origin and going up whose length is  $\sqrt{1 + 2\sqrt{11}}$ . By creating perpendiculars at the non-origin endpoints and intersecting them we would have constructed the point  $(3 - 4\sqrt{7}, \sqrt{1 + 2\sqrt{11}})$ .

Remember the pentagon construction? This is one way to know it was possible without anyone telling you any of the construction steps beforehand. One corner of the pentagon was at  $(4,0)$  and the next corner had an  $x$ -coordinate of  $\sqrt{5} - 1$ . Since by our theorem  $\sqrt{5} - 1$  is SC we could build the point  $(\sqrt{5} - 1, 0)$ , the circle from  $(0,0)$  to  $(4,0)$ , and then build the perpendicular to the  $x$ -axis at  $(\sqrt{5} - 1, 0)$ . This line hits the circle at 2 other points of the pentagon, and the rest follows very quickly.

Our theorem tells us that if a number can be built from the integers using  $+$ ,  $-$ ,  $\times$ ,  $/$  and  $\sqrt{\quad}$  then it must be SC. It doesn't address the question of if there is anything else that can be built with straightedge and compass. As it turns out the answer to that question is "there isn't":

Theorem: Which numbers are SC

If  $x$  is a SC number,  $x$  can be built from the integers using a finite combination of the additions, subtractions, multiplications, divisions, and real square roots. Therefore a number  $x$  is SC if and only if  $x$  can be built from the integers using a finite combination of the additions, subtractions, multiplications, divisions, and real square roots.

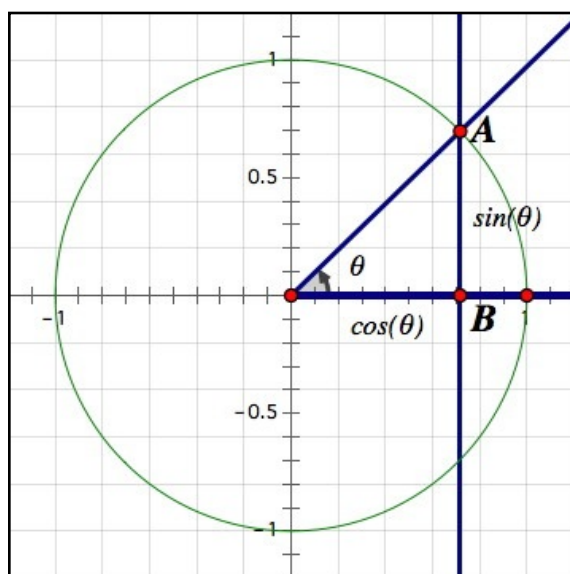
This theorem tells us that for finding SC points our custom tools are all we need - we can build every SC point using “Plot Points” to get integer lengths together with the custom tools defined in this section.

In many constructions we may not just need to work with points but also angles. We don’t need to add any new ideas to determine which angles are SC because of the following theorem:

Theorem: Which angles are SC

An angle of measure  $\theta$  is SC if and only if  $\cos(\theta)$  is a SC number if and only if  $\sin(\theta)$  is a SC number.

This theorem works because any angle can be copied into “standard position” (i.e. with the vertex at the origin and one side on the positive  $x$ -axis). That means after drawing a unit circle we can always have a picture like this:



*an angle in “standard position” with the unit circle*

If you can construct the angle  $\theta$  in this picture then by intersecting  $\theta$ ’s terminal ray with the unit circle you can construct the point A. Then by constructing the line through A parallel to the  $y$ -axis you can find the point B. By constructing the segments from the origin to B and from A to B you would have lengths  $\cos(\theta)$  and  $\sin(\theta)$  respectively. Conversely if you knew that the number  $\cos(\theta)$  was SC by making a circle centered at the origin with radius  $\cos(\theta)$  and



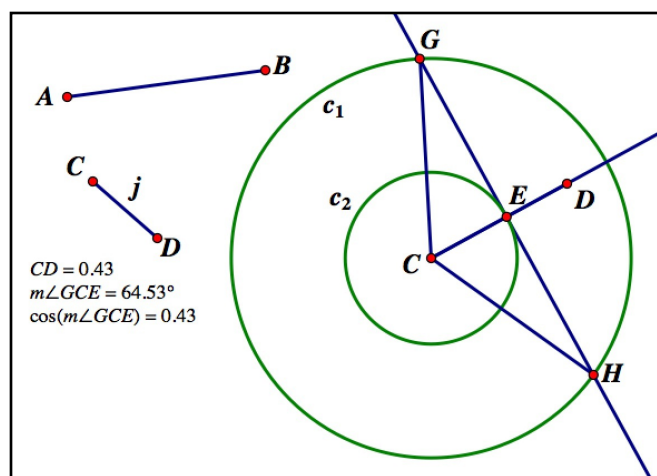
intersecting it with the  $x$ -axis you would have the point B (you would choose the right-hand point for a Quadrant I angle and the left-hand point for a Quadrant II angle). By building the line perpendicular to the  $x$ -axis at B and intersecting it with the unit circle you would have the point A - which gives you the angle  $\theta$  and the length of AB (which is  $\sin(\theta)$ ). We can automate this into a new custom tool:

**Custom Tool:** “Build Quadrant I angle from cosine” (i.e. build the angle  $\theta$  given a unit length and a segment of length  $\cos(\theta) > 0$ )

Parents: Endpoints A and B of a unit segment, a segment  $j$  of length  $0 < \cos(\theta) < 1$ , and endpoints C and D of a segment where the angle will be built.

Children: Points F and G along with segments CF and CG which form angles of measure  $\theta$  on either side of CD.

- 1) Construct the ray from C through the point D.
- 2) Construct the circle  $c_1$  centered at C with radius AB.
- 3) Construct the circle  $c_2$  centered at C with radius  $j$ .
- 4) Construct the intersection of  $c_2$  with the ray CD - call this point E.
- 5) Construct the line L1 perpendicular to EF at E.
- 6) Construct the intersections of the circle  $c_1$  and the line L1 - these are F and G.
- 7) Construct the segments CF and CG.
- 8) Select the points A, B, C, D, F, G and the segments CF and CG. Make the custom tool.



*building an angle from cosine, with sample measurements*

It's worth noting that in this tool if the initial segment AB is not a unit segment, the tool will build an angle whose cosine is the ratio  $|CD|/|AB|$ . You can also make a variation on this tool to make the angles be Quadrant II angles (i.e. measure greater than  $90^\circ$ ) simply by reversing the ray in step 1 (i.e. using the ray from D through C) and using the new ray DC in step 4. It is also



possible to make tools that build angles from sines although these need an extra step or two (as you would need to create the perpendicular to CD at C to start).

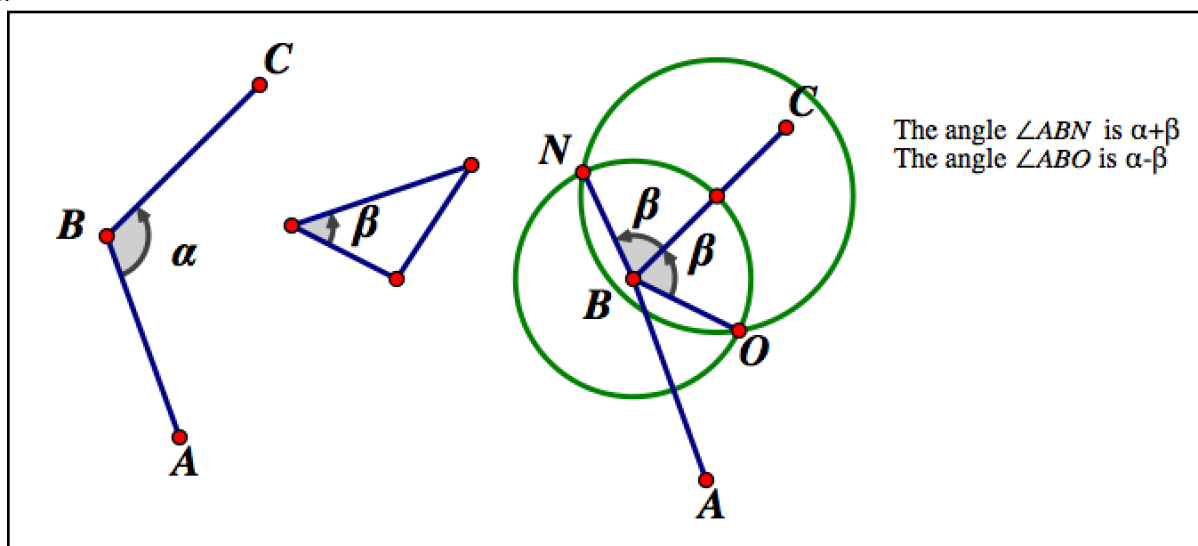
The theorem that relates whether an angle is SC to its cosine and sine gives rise to another theorem about combining SC angles:

Theorem: Combining SC angles

Let the angles  $\alpha$  and  $\beta$  be SC angles. Then:

- 1)  $\alpha + \beta$  and  $\alpha - \beta$  are SC
- 2) The angle  $\frac{\alpha}{2}$  is SC (that is, the command “Construct Angle Bisector” is SC)

The first part of this theorem follows simply from the fact that copying an angle is a SC construction - if you copy the angle  $\beta$  on to the terminal side of  $\alpha$  the resulting composite angle has measure either  $\alpha + \beta$  or  $\alpha - \beta$  depending on where you copied  $\beta$  outside of  $\alpha$  or inside of  $\alpha$



*adding and subtracting angles is SC*

The second part of the theorem follows from the trigonometric identity  $\cos\left(\frac{\alpha}{2}\right) = \pm \sqrt{\frac{1 + \cos(\alpha)}{2}}$ ; if  $\cos(\alpha)$  is SC then we can add 1 to it, divide by 2, and take a square root while remaining SC - and once  $\cos\left(\frac{\alpha}{2}\right)$  is SC, so is the angle  $\frac{\alpha}{2}$ .

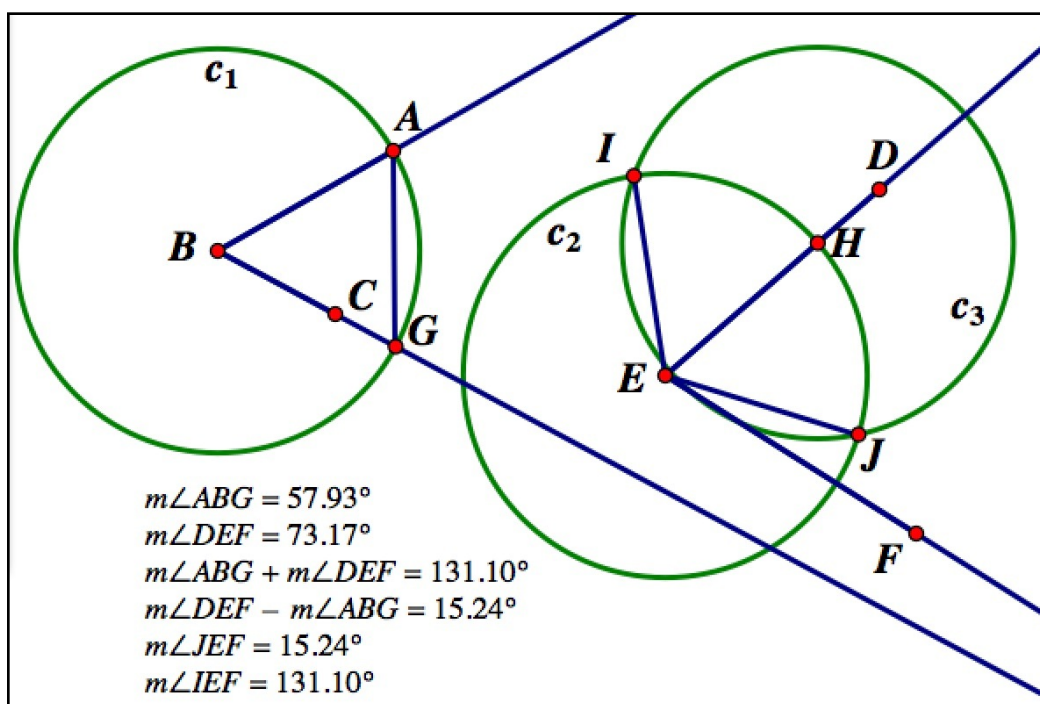
If you are going to do a lot of straightedge and compass constructions it is probably worthwhile to have a custom tool to add and subtract angles:

**Custom Tool:** “Add/subtract angle 1 from angle 2”

Parents: Points A, B, and C which define the first angle ABC and points E and D which define one side of the second angle DEF.

Children: Points I and J along with segments EI and EJ which add and subtract the first angle from the second angle using EF as a side for the new angles.

- 1) Construct the ray from E through D.
- 2) Construct the rays from B through A and C.
- 3) Construct the circle  $c_1$  centered at B through A.
- 4) Construct the segment BA
- 5) Construct the intersection of  $c_1$  with the ray BC - call this G.
- 6) Construct the segment AG.
- 7) Construct the circle  $c_2$  centered at E with radius equal to AB.
- 8) Construct the intersection of  $c_2$  with the ray ED - call this H.
- 9) Construct the circle  $c_3$  centered at H with radius equal to AG.
- 10) Construct the intersections of  $c_2$  and  $c_3$  - call them I and J
- 11) Construct the segments EI and EJ (these will define the two angles).
- 12) Select the points A, B, C, E, D, I, J, and the segments EI and EJ - create the custom tool.



*a custom tool for adding and subtracting angles, with sample measurements*

As an extra bonus this same construction can be used to copy angles onto a new segment. If ABC is an angle and DE just a segment, if you use the tool matching A, B, C, D and E you will create 2 copies of ABC on DE - one on the right of DE and one on the left. Now simply delete the unwanted copy.

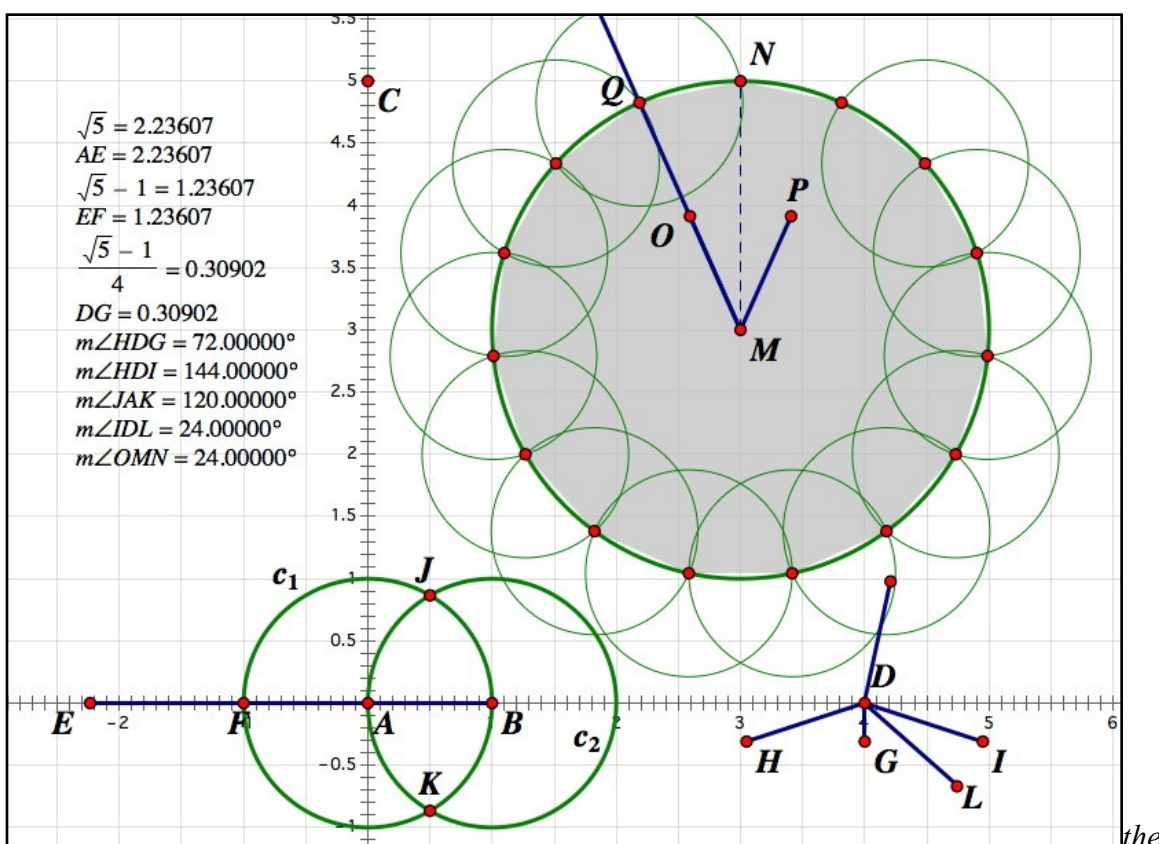
To put all of these together and use some of our custom tools here is a complex construction: a regular 15-gon is SC.

Before going into the construction steps let's break the idea down first. The central angle of a regular 15-gon is  $360^\circ/15=24^\circ$ . We know that  $\cos(72^\circ)=\frac{\sqrt{5}-1}{4}$ . We have custom tools now for taking a square root and for division - so given a segment of length 5 we can use the square root tool to quickly make the length  $\sqrt{5}$  and by subtracting a unit length we will get the length  $\sqrt{5}-1$ . By making the segment from (0,0) to (4,0) we would have a segment of length 4; using our division tool would give us a segment of length  $\frac{\sqrt{5}-1}{4}$ . Using our custom tool for building angles from cosines we would then have an angle of  $72^\circ$  (actually we have two copies of  $72^\circ$  with a common side, so we also would have the angle  $144^\circ$ ). The next step is to build an angle of  $120^\circ$ . To do this start with any segment AB. Then select A, B, and the segment AB and use the "Construct circles by center + radius" command. This will build two circles with a common radius. Intersect those two circles to create points C and D. ABC and ABD will be equilateral triangles, so the angle CAD will measure  $60^\circ+60^\circ=120^\circ$ . We now have angles of measure  $144^\circ$  and  $120^\circ$  so by using the "add/subtract" angle tool we will have an angle of  $144^\circ-120^\circ=24^\circ$ . By copying this angle to the center of any circle we can build a perfect regular 15-gon.

Converting this into Sketchpad steps:

- 1) Use "Plot Points" to create the points A=(0,0), B=(1,0), C=(0,5), and D=(4,0).
- 2) Using A and B for the unit segment and A and C for the length 5 segment, use the square root tool to build a segment of length  $\sqrt{5}$ . This will build a segment AE of the right length in the x-axis (make sure to label the endpoint E).
- 3) Use the subtraction tool to subtract the smaller segment AB from the larger segment AE. This will create a new point F, and EF will have length  $\sqrt{5}-1$ .
- 4) Using the division tool, select the points A and B as the ends of the unit segment, the points E and F as the ends of the  $\sqrt{5}-1$ , and then the points A and F (which are 4 apart). This will create a segment DG whose length is  $\frac{\sqrt{5}-1}{4}$ .
- 5) Using the "build angle from cosine tool", select A and B as the ends of the unit segment and the ends of the segment DG. This will create two back-to-back copies of the  $72^\circ$  angle that end in points H and I. The angle HDI is therefore  $144^\circ$ .
- 6) Construct the segment AB.
- 7) Construct the circles c1 and c2 with centers A and B and radius AB.
- 8) Construct the intersection of c1 and c2 to get points J and K. The angle JAK will be  $120^\circ$ .

- 9) Using the “add/subtract angles” tool, subtract the angle JAK from the angle HDI (using the segment DH). This will create two points and two segments. Let L be the point “inside” HDI. Then the angle IDL should measure  $24^\circ$ .
- 10) Plot the points  $M=(3,3)$  and  $N=(3,5)$  and construct the circle  $c_3$  centered at M through N. (M and N were chosen to make the circle  $c_3$  out of the way of the rest of the construction - we will build the 15-gon inside  $c_3$ ).
- 11) Using the “add/subtract angles” tool, copy the angle HDL onto the segment whose ends would be (3,3) and (3,5) (you don’t need the actual segment to use the tool, just the endpoints). This will create points O and P for which the angle OMN is  $24^\circ$ . (you can delete P, as we won’t be using it).
- 12) Construct the ray from M through O.
- 13) Let Q be the point of intersection of  $c_3$  with the ray MO. N and Q will be 2 corners of the regular 15-gon.
- 14) Construct the circle centered at Q through N. Intersecting this with  $c_3$  will yield the next polygon corner R. Repeating this around the circle will give you all 15 corners of the regular polygon.



*the complex construction of a regular 15-gon, with high precision measurements along the way*

The tools we have introduced in this section will let you replicate any straightedge and compass construction within Sketchpad. You might wonder what other constructions are SC and what are

some constructions that can't be done with just a straightedge and compass? Here are a few examples of some SC and non-SC constructions:

Other straightedge and compass constructions:

- A  $3^\circ$  angle
- A regular 17-gon, 257-gon, and 65,537-gon (don't try that last one at home!)
- Splitting a line segment into  $n$  equal pieces (where  $n$  is any natural number)
- A square with any given positive whole number area (or any area which is a positive SC number)
- Translating a SC object by a vector  $\langle x, y \rangle$ , where  $x$  and  $y$  are SC numbers
- Rotating any SC object through a SC angle about a SC center

Constructions that are impossible using just straightedge and compass:

- A  $1^\circ$  angle
- A regular 7-gon
- Trisecting a general angle (some angles can be trisected with a straightedge and compass and some can't - a  $60^\circ$  cannot be trisected with SC operations).
- A line segment with length  $\sqrt[3]{2}$
- A square with area  $\pi$

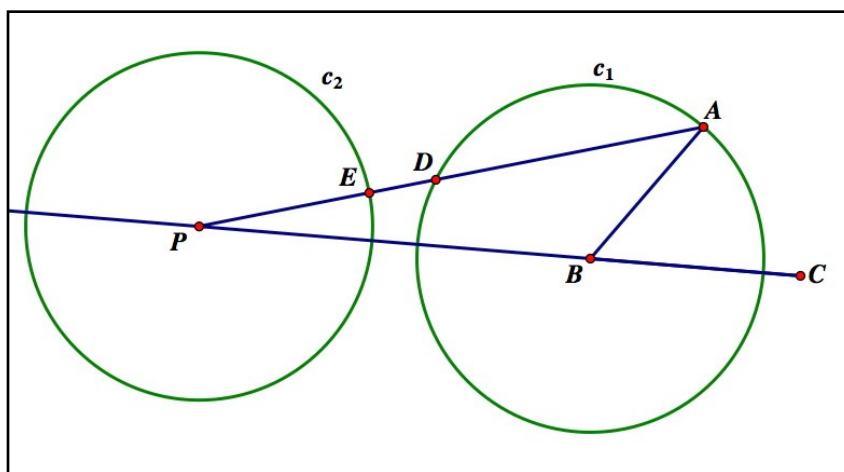
To wrap up this section let's take a look at the non-SC construction "trisection of an angle". This construction will use the idea of "verging" - that is sliding a figure around so that 2 of its points A and B match two given points in the plane C and D (this kind of idea is central to the use of a marked ruler and constructions that involve paper folding). Because we can't exactly match two pairs of points in Sketchpad this construction will be an "approximate" one - it would be perfect if we could match the points exactly, but in practice we can only get really close in Sketchpad.

**Non-SC Construction:** Trisecting a general angle

Let ABC be the given angle. Follow these steps to trisect it:

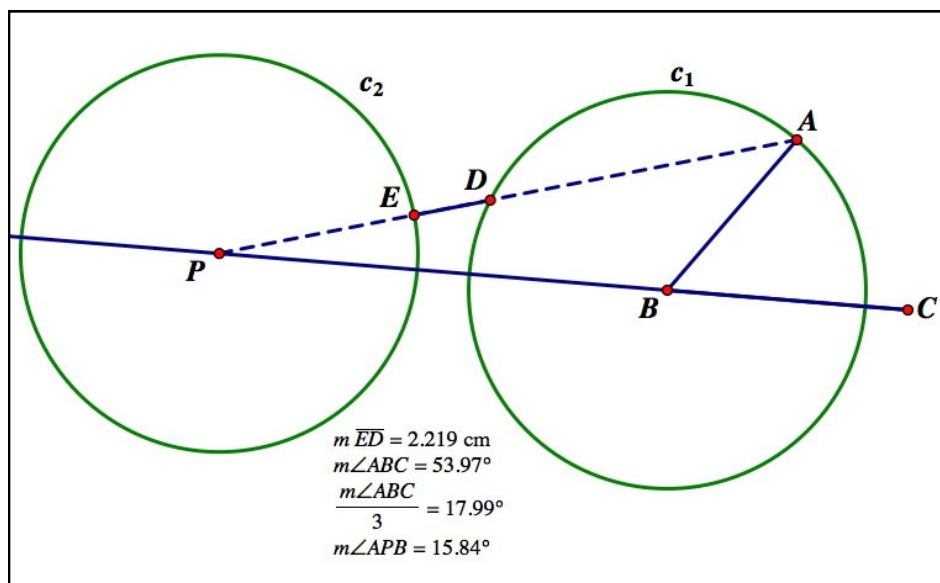
- 1) Construct the circle c1 centered at B with radius A.
- 2) Construct the ray from C through B.
- 3) Construct a point P on the ray CB. Drag it so it is well outside the circle c1 on the opposite side from C.
- 4) Construct the segment AP.
- 5) Construct the intersection of AP with c1 - call it D.
- 6) Construct a circle c2 centered at P with radius equal to AB.
- 7) Construct the intersection of the segment AP with c2 - call it E.

Your picture should now look like this:

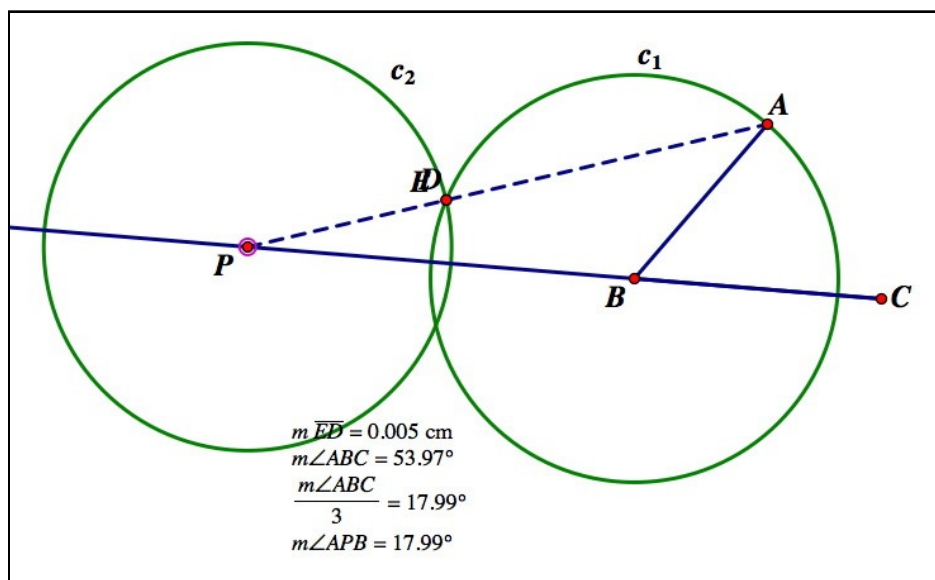


*preparation for trisecting the angle ABC*

So far everything except the sliding point P can be built with straightedge and compass. Now what we need to do is slide the point P along the ray CB so that the point E merges with the point D. This can only be done approximately and can be hard to do accurately by eyeballing it (the points D and E take up a non-trivial portion of your screen so getting them exactly over one another can be difficult). If you could merge D and E exactly then the angle APB will be exactly one-third the measure of ABC. To make the merging a bit easier and more precise we can construct the segment DE, measure its length (normally or by coordinate distance), and then slide P around to make the measured length as close to 0 as possible:



*with the measurements taken, slide P to make E merge with D*



with E and D merged as closely as possible we have (approximately) trisected the angle ABC

On paper you would do this construction with a marked ruler. Rather than construct the point P, create two marks on your ruler that match A and B. Then slide the marked ruler so that A is on its edge, the first mark is on  $c_1$ , and the second mark is on the ray CB - this second mark will be the exact location of P that will give you the trisected angle.

For more information on different types of constructions look at the excellent introductory text *Geometric Constructions* by George E. Martin (Springer, 1998) and the high-level texts *Galois Theory* by David Cox (Wiley, 2nd edition 2012) and *Contemporary Abstract Algebra* by Joseph Gallian (Cengage Learning, 8th edition, 2012).

## Section 1.7 Homework - Sketchpad and Straightedge and Compass Constructions

In the following problems do each Sketchpad problem in its own sketch. Print out the sketch for those problems and perform all measurements with the highest possible accuracy. Remember many of the constructions require a unit length, so you may need to “define unit distance” as the first main operation in a sketch and measure lengths by coordinate distance if you change the scale from the standard (if you do so you will also have to manually calculate areas using various area formulas instead of the built-in measurements). You may use the custom tools defined in this section.

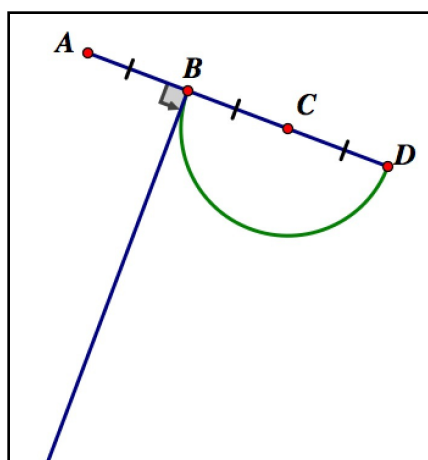
- 1) Construct a square whose area is exactly 7. Measure the area of the square for confirmation.
- 2) Construct a segment whose length is  $3/7$ . Verify this by estimating  $3/7$  in the calculator along with the length of your segment.

- 3) Construct a segment whose length is  $\sqrt{7} + \sqrt{3} - 2$ . Verify this by measuring the coordinate distance and estimating  $\sqrt{7} + \sqrt{3} - 2$  in a calculator.
- 4) Create a line segment whose length is  $\frac{2 + 3\sqrt{11}}{5 - 2\sqrt{3}}$ . Verify this by measuring the coordinate distance and estimating  $\frac{2 + 3\sqrt{11}}{5 - 2\sqrt{3}}$  in the calculator.
- 5) Construct angles of measure  $72^\circ$ ,  $45^\circ$ , and  $30^\circ$  using the SC operations. Then use additions and subtractions of angles to get whose measure is  $3^\circ$  (take measurements of your angles). If you put  $3^\circ$  at the center of a circle, what kind of regular polygon could you make with it?
- 6) Build an angle whose cosine is exactly  $2/9$  using SC operations. Verify your angle is correct by taking its measurement and using the cosine function in the calculator.
- 7) Build an angle whose sine is exactly  $1/5$  using SC operations. Verify your angle is correct by taking its measurement and using the sine function in the calculator.
- 8) Prove that the square root tool really does produce the square root (hint: in the construction the point G is a right angle - which means there are two smaller right triangles which turn out to be similar and have a common side).
- 9) Given that the cosine of  $(360/17)^\circ$  is exactly
 
$$\frac{1}{\sqrt{32}} \left( \sqrt{15 + \sqrt{17} - \sqrt{34 - 2\sqrt{17}}} + \sqrt{2 \left( 34 + 6\sqrt{17} - \sqrt{578 - 34\sqrt{17}} + \sqrt{34 - 2\sqrt{17}} + 8\sqrt{2(17 + \sqrt{17})} \right)} \right)$$
 explain why a regular 17-gon is constructible with straightedge and compass.
- 10) Explain why if a regular  $k$ -gon is SC, so is a regular  $(2k)$ -gon.
- 11) Look up in an outside source what the problems “doubling the cube” and “squaring the circle” are. Explain these in your own words and see if you can find out if these are solvable with straightedge and compass; if not, can they can be done with a marked ruler?
- 12) Look up the surprising Mohr-Mascheroni theorem and explain it in your own words.
- 13) Create a custom tool for “Create angle from sine” that is the analog of our “Create angle from cosine”.
- 14) Follow the instructions below to create a segment of approximate length  $\sqrt[3]{2}$  by “verging” (this will require a unit distance):

- 1) Let AB be a segment of length  $1/2$
- 2) Construct a circle c1 whose center is at A and whose radius is 1.
- 3) Construct a circle c2 whose center is at B and whose radius is 1.
- 4) Let C be one of the intersections of the circles c1 and c2.
- 5) Construct the ray from C through A.
- 6) Let D be the other intersection of the ray CA with c1.
- 7) Construct the ray from D through B.
- 8) Construct the ray from A through B.
- 9) Construct a point E on the ray AB (for convenience, slide it well past B).
- 10) Construct the segment CE.



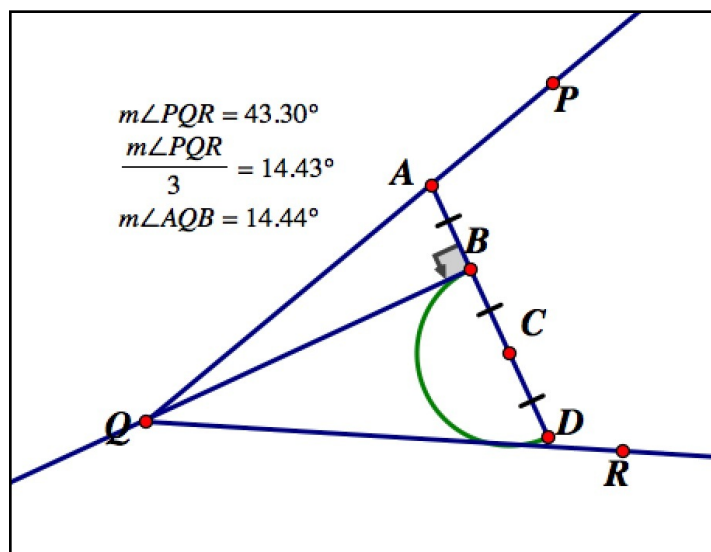
- 11) Let F be the intersection of the segment CE with the ray DB.
  - 12) Let c3 be the circle centered at E with radius 1.
  - 13) Let G be the intersection of c3 with the segment CE.
  - 14) Define the segment FG.
  - 15) Measure the coordinate distance from B to E.
  - 16) Measure the length of the segment FG.
  - 17) Slide the point E along AB until the points G and F merge. At this point the coordinate distance BE should be the cube root of 2 (verify this in the calculator).
- 15) Another tool that can be used to trisect an angle is called a “tomahawk”. A tomahawk consists of a trisected segment ABCD, a semi-circle from B to D with C as its center, and a ray from B perpendicular to ABCD:



*a “tomahawk” tool*

Give a list of Sketchpad steps (not necessarily SC ones) to create a tomahawk you can slide and rotate around. You will probably need to hide several parts of the construction and use the “Arc through 3 points” construction to get the semicircle.

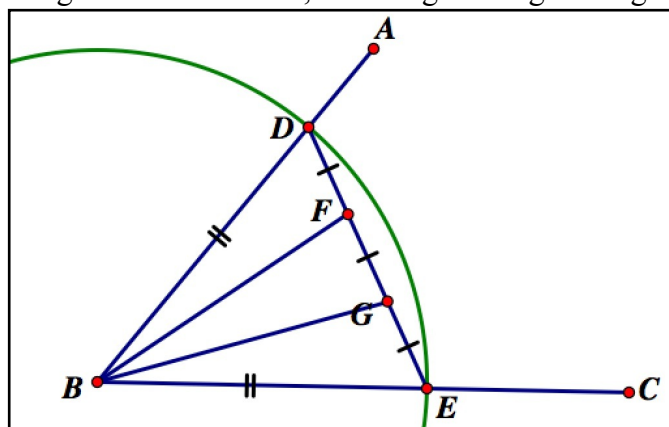
- 16) A tomahawk can be used to trisect an angle PQR as follows: Slide the tomahawk so the point A is on the ray PQ, the semi-circle is tangent to the ray QR, and the ray goes through the vertex Q. Then the angle AQB should be exactly one-third the angle PQR. Create your own angle PQR and slide the tomahawk you made in problem 15 into the described position. Then measure the appropriate angles to verify the (approximate) trisection. (Note: this is approximate only as in practice A can only approximately be moved to PQ, the ray from B can only approximately be slid to it contains Q, and the semi-circle can only be slid into the tangent position approximately - if these could be done exactly the angle would be trisected exactly)



*a tomahawk used to trisect an angle (approximately)*

17) A common proposed trisection of an angle using a straightedge and compass goes something like this:

- 1) Take the angle ABC and choose points D on AB and E on BC so that  $|AD| = |BE|$ .
- 2) Construct the segment DE.
- 3) Trisect DE, getting points F and G (this can be easily done with our division construction).
- 4) Construct the segments BF and BG, trisecting the original angle.

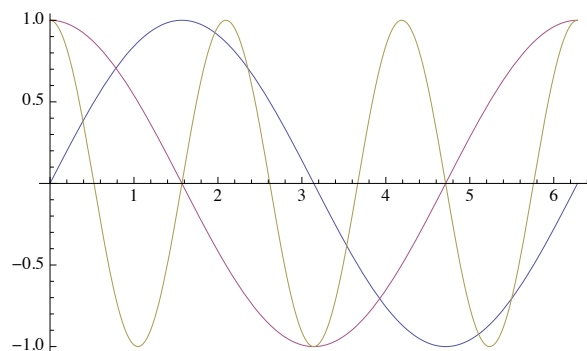


*a commonly proposed trisection of the angle ABC*

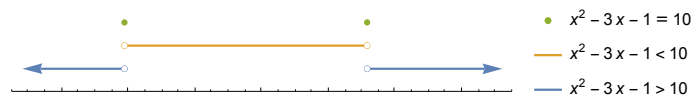
Create an example of this construction in Sketchpad and verify that it does not work. There is actually a book *A Budget of Trisections* by Underwood Dudley (Springer, 1987) which chronicles many attempts by people over the years to trisect a general angle using only a straightedge and compass even though it has been proven to be impossible.

# Chapter 2 - Mathematica Basics

```
Plot[ {Sin[x], Cos[x], Cos[3 x]}, {x, 0, 2 Pi}]
```



```
NumberLinePlot[ {x^2 - 3 x - 1 > 10, x^2 - 3 x - 1 < 10, x^2 - 3 x - 1 == 10}, {x, -4, 8}, PlotLegends -> "Expressions"]
```



```
Expand[ (2 x + 1) ^ 5]
```

$$1 + 10 x + 40 x^2 + 80 x^3 + 80 x^4 + 32 x^5$$

```
TrigReduce[ Cos[x] ^ 5]
```

$$\frac{1}{16} (10 \cos[x] + 5 \cos[3 x] + \cos[5 x])$$

```
Simplify[Tan[ ArcSec[x] ], x < -1]
```

$$-\sqrt{-1 + x^2}$$

```
FunctionDomain[ 1/(x^2 - 2 x), x]
```

$$x < 0 \mid \mid 0 < x < 2 \mid \mid x > 2$$

```
FunctionRange[ 1/(x^2 - 2 x), x, y]
```

$$y \leq -1 \mid \mid y > 0$$

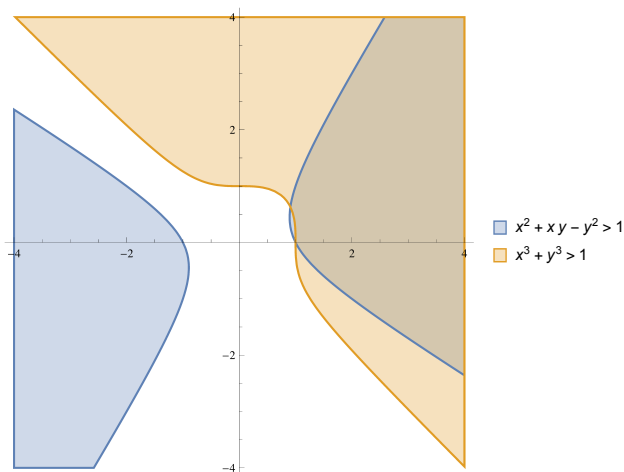
```
Solve[ x^3 - 4 x + 1 == 0, x]
```

$$\left\{ \left\{ x \rightarrow \frac{\left( \frac{1}{2} (-9 + i \sqrt{687}) \right)^{1/3}}{3^{2/3}} + \frac{4}{\left( \frac{3}{2} (-9 + i \sqrt{687}) \right)^{1/3}} \right\}, \right. \\ \left\{ x \rightarrow -\frac{(1 + i \sqrt{3}) \left( \frac{1}{2} (-9 + i \sqrt{687}) \right)^{1/3}}{2 \times 3^{2/3}} - \frac{2 (1 - i \sqrt{3})}{\left( \frac{3}{2} (-9 + i \sqrt{687}) \right)^{1/3}} \right\}, \left\{ x \rightarrow \right. \\ \left. -\frac{(1 - i \sqrt{3}) \left( \frac{1}{2} (-9 + i \sqrt{687}) \right)^{1/3}}{2 \times 3^{2/3}} - \frac{2 (1 + i \sqrt{3})}{\left( \frac{3}{2} (-9 + i \sqrt{687}) \right)^{1/3}} \right\} \left. \right\}$$

```
N[E, 1000]
```

```
2.71828182845904523536028747135266249775724709369995957\
496696762772407663035354759457138217852516642742746639\
193200305992181741359662904357290033429526059563073813\
232862794349076323382988075319525101901157383418793070\
215408914993488416750924476146066808226480016847741185\
374234544243710753907774499206955170276183860626133138\
458300075204493382656029760673711320070932870912744374\
704723069697720931014169283681902551510865746377211125\
238978442505695369677078544996996794686445490598793163\
688923009879312773617821542499922957635148220826989519\
366803318252886939849646510582093923982948879332036250\
944311730123819706841614039701983767932068328237646480\
429531180232878250981945581530175671736133206981125099\
618188159304169035159888851934580727386673858942287922\
849989208680582574927961048419844436346324496848756023\
362482704197862320900216099023530436994184914631409343\
173814364054625315209618369088870701676839642437814059\
271456354906130310720851038375051011574770417189861068\
7396965521267154688957035035
```

```
RegionPlot[ {x^2 + x y - y^2 > 1, x^3 + y^3 > 1}, {x, -4, 4}, {y, -4, 4}, Frame -> False, Axes -> True, PlotLegends -> "Expressions"]
```

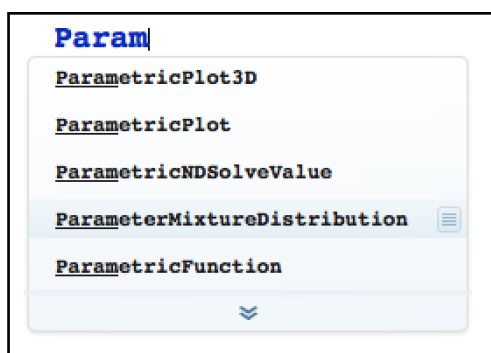


## Section 2.1 - An Introduction

Mathematica is one of the oldest and most versatile computer algebra systems available. It was introduced by Wolfram Research (<http://www.wolfram.com>) in the late 1980's. Mathematica is one of the most powerful mathematical engines available - able to work with approximate and exact quantities, plot two- and three- dimensional graphs and regions, and work with equations with equal facility. In addition to being a calculation engine Mathematica is also a full programming language, so if its ever-expanding capabilities fall short (which is rare) or you would like to create a specific command or demonstration of your own you can create your own code fairly easily. Mathematica also works with many common file formats, making it easy to import and export graphics, sound, and even HTML code (a great way to make web pages that involve mathematical notation).

There are two drawbacks to the power of Mathematica, though. The first is the sheer complexity of the program itself. Although you can (and hopefully will) be able to access many of its functions fairly quickly, back when the full instruction manual had a print version it ran over 1400 pages. This complexity can be intimidating at first, although it needn't be - in its attempt to be everything to all mathematicians Mathematica includes hundreds of different functions many of which you may never need (when was the last time you needed the Chebyshev polynomials of the second kind?). The second drawback is the cost of Mathematica. While the student version is very reasonably priced (roughly the same as a high end graphing calculator), you still need a computer to run it on (the faster the better). The cost of computer plus the software places it well above the price of simpler technological tools like calculators, although Mathematica brings far more power to the table than almost any other tool.

To help counter the complexity issue the 2012 release of Mathematica 9 included a powerful new feature - the Input Assistant. As you begin to enter a command the Input Assistant will give you a list of suggested autocompletions via a cell-phone like popup menu:

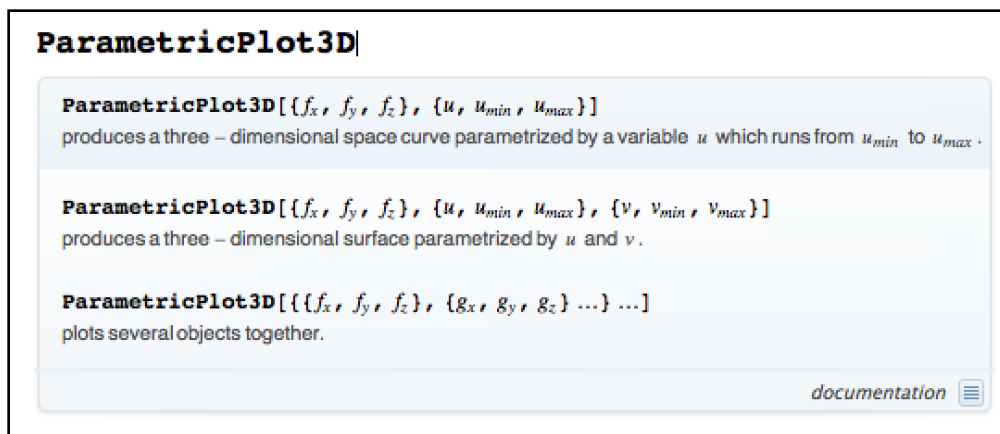


*the autocomplete feature of the Input Assistant in Mathematica 9*

The autocomplete feature makes it easier and quicker to get your commands into Mathematica. It also has the additional bonus of helping you explore Mathematica as you will mostly likely see

commands you are not familiar included in the suggestions, giving you a starting point to learn new functions and commands.

The Input Assistant goes even further though - once you have a complete command name entered, hitting the key combination Command-Shift-K (Control-Shift-K in Windows) or clicking on a popup that will appear will bring up a list of templates for that function:



*suggested templates for ParametricPlot3D*

To select one of the templates just click on it. The template you chose will replace the command name and you just need to fill in the blanks (you can use the Tab key to jump from blank to blank).

This template feature is very useful but as you first learn commands and what they do I would recommend not taking advantage of it. Learning the structure of a new command or feature “by hand” helps you learn what the command can do a bit better and can give you a better feel for how similar commands are structured. Once you have mastered a particular command and learned what you can from it then the template feature can make using it quicker and more convenient.

## Section 2.2 - Working with Mathematica Notebooks

When you first run Mathematica you will start with an empty document (called “Untitled 1” or something along those lines). These Mathematica documents are called notebooks and they are where you will interact with Mathematica and where it will place the results of your commands. Until you actually have Mathematica do anything you are only seeing part of Mathematica, however. The program itself is split into two parts - the “front end” and the “kernel”. The front end controls your interaction with Mathematica (basically it controls the display, editing, and saving of the notebook) and the kernel is the engine which stores all the values and handles the actual calculations. The kernel itself doesn’t actually start until you enter a command or calculation and have Mathematica evaluate it. If the split between front end and kernel seems odd at first glance, one of the benefits of it is that if the kernel encounters some kind of problem or freezes up you can quit just the kernel - which leaves your notebook intact in the front end where it can be saved.

To get started, in the blank notebook type in some basic computation like  $2+2$ . The text you create is placed within a bracket, called a “cell”. The cell is the basic Mathematica way of grouping together text, inputs and outputs, graphics, and other ideas. By default when you create a new cell (which you can do by clicking the mouse in between or below existing cells) the cell type is an “input cell” - meaning Mathematica will interpret the entries in the cell as commands rather than text. Having put in  $2+2$ , hit either the Enter key on the keypad or Shift+Return on the main part of the keyboard to evaluate the cell. There will be a short pause while the kernel loads (you will get that pause every time you restart Mathematica), and then you will see the result:



*The first Mathematica computation*

Not exactly the hardest calculation, but this shows you how Mathematica uses the brackets on the right edge of the notebook to group calculations together. It also illustrates how at a very basic level Mathematica uses a “question and answer” approach, a conversation between you and the computer where you pose questions and it gives the answers. You are not restricted to one input per input cell, either. Click below that cell and type in  $2^{100}$  to enter in a new input, and then hit the Return key (without the Shift key) on the main keyboard. No evaluation will be done and the cell will expand down to include a new line where you can enter another calculation, say  $1000000000^{(1/2)}$ . Evaluate this new cell (by either the keypad or Shift+Return) and you should see this:

```

In[2]:= 2 ^ 100
        1 000 000 000 ^ ( 1 / 2)

Out[2]= 1 267 650 600 228 229 401 496 703 205 376

Out[3]= 10 000  $\sqrt{10}$ 

```

*Multiple calculations at once*

There are three things worth noting about how this is displayed. First, even though we use one large input cell each output gets its own output cell (and number, which we'll discuss later) and all the pieces are grouped together by one large cell bracket. You can use this outer bracket to delete the entire set by clicking on the rightmost bracket and using the standard Cut command - this will delete both the input and output cells from the notebook (although not from the kernel's memory). Second, notice that Mathematica does a little bit of formatting on its own - long numbers are grouped into sets of three digits, for example. Third (and most important) Mathematica is able to find the square root of a billion exactly, simplifying it as far as it can go. The ability to handle quantities exactly and symbolically is at the core of Mathematica's usefulness in higher mathematics.

There may be times when you want Mathematica to do a calculation but not see the actual output (say because it is too long or you only want to use it as fodder for another calculation later on). To do this put a semi-colon at the end of the input. To see this in action evaluate the following pair of commands exactly in a single cell - `N[Pi, 30]` and `N[Pi, 10000];`

```

In[4]:= N[Pi, 30]
        N[Pi, 10 000];

Out[4]= 3.14159265358979323846264338328

```

*many digits of  $\pi$*

Based on the output it's not too hard to figure out the commands here are for - both are asking for estimates to  $\pi$ . The only difference is that one wants 30 decimal places (which fits nicely in the notebook) and the other wants 10,000 places (which would require a lot of scrolling). By putting a semi-colon at the end of the second command it is still executed but the display is suppressed, keeping the notebook cleaner and easier to read.

A nice feature of Mathematica is that it allows you to use the results of computations as fodder for new ones without needing to copy or re-type them. There are three main ways to do this - storing results in variables, using output numbers (the things like Out[7] you've already seen in the notebook), and by referring to the "previous answer".

To use a variable name all you need to do is use the = sign. For example if you want to store the value  $5^3 - 7$  for future use you can store it by calling it something, say  $t=5^3-7$ . Then any future use of the letter  $t$  will refer that number until the variable is cleared, either by evaluating the command `Clear[t]` or  $t=.$  ( $=.$  basically is an “undefine” command). So you could do a series of computations like this:

```
In[1]:= t = 5 ^ 3 - 7
Out[1]= 118

In[2]:= 3 t + 1
         t / (t + 1)
         t = .
         t / (t + 1)
Out[2]= 355

Out[3]=  $\frac{118}{119}$ 

Out[5]=  $\frac{t}{1 + t}$ 
```

*storing a value for future computations and then clearing it*

As you can see storing 118 in the name  $t$  lets it be used in a sequence of other computations. Once the the variable  $t$  has been “cleared” by  $t=.$ , however, Mathematica simply parrots  $t/(t+1)$  back to us since it no longer knows what  $t$  is. There is no practical limit to the length of a variable name. We tend to use short ones like  $t$ ,  $a$ , and  $i$  for simple things but you can use full words or compound words like `velocity`, `localmax`, or even `thisisafartoolongavariablename`. One good habit to get into is to avoid variable names that start with capital letters as built-in Mathematica commands typically start with capital letters. It’s OK to store a value in the name `max`, but `Max` is a command and if you try evaluate `Max=5` you will get an error (since you are basically trying to redefine a Mathematica command - which isn’t allowed without doing more work). It’s also probably a good idea to avoid storing values in letters like  $x$  and  $y$  that we commonly use for other things in mathematics.

You can also use the results of computations by referring to them by the “Out” number you see next to them, including the square brackets. So for example you might do something like this:



```

In[6]:= 3 ^ 10
        2 ^ 15
Out[6]= 59 049

Out[7]= 32 768

In[8]:= Out[6] - Out[7]
Out[8]= 26 281

```

*using Out to chain computations together*

In addition to output numbers you can also use a “last answer” function similar to the ones found on graphing calculators. % refers to the last answer, %% to the second to last answer, and so on. So the above arithmetic could also be done via:

```

3 ^ 10
2 ^ 15

59 049

32 768

%% - %

26 281

```

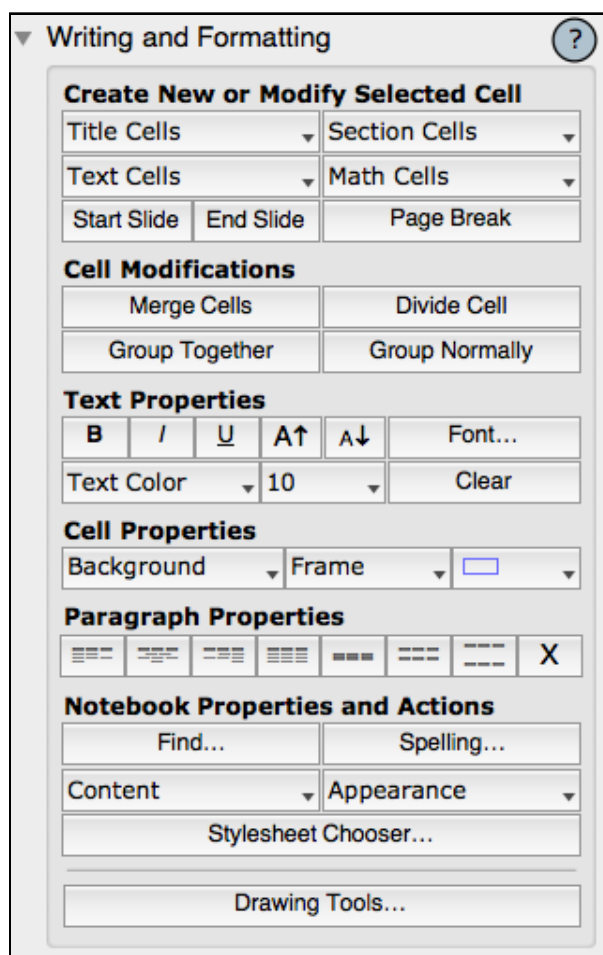
*using % and %% to reuse previous answers*

You need to be very careful when using % and Out (especially %). Both refer to answers as they were created chronologically in the current Mathematica session, not in the physical order they appear in the notebook. By using the mouse you can jump up and down a notebook and re-evaluate the cells in different orders, in which case the chronological order of the answers no longer matches the order they appear in the notebook. This makes it very easy to lose track of what things like %%% or %%% might refer to. Out numbers also refer only to the current Mathematica session - if you save the notebook, quit Mathematica (or just the kernel), and reopen the notebook all the results are still visually displayed in the same order but are no longer linked in any way - the first calculation will start over as Out[1], Out[2], etc. To be safe it's a good idea to use % and Out numbers primarily for mathematical scratch work. If you are going to save and come back to a notebook later don't jump up and down the notebook re-evaluating cells - it's best to keep everything in order (or at least make sure the cells are in the proper order before you save). That way when you reopen the notebook you can easily re-evaluate the cells in the proper order and keep any chained calculations intact. If the work you are doing goes beyond the level of scratch work it's probably best to use variable names to store important values and

to keep the cells in order rather than jumping back and forth - this will make your work much easier to follow and re-evaluate when you come back to the notebook after a break.

You can use Mathematica to go far beyond the level of scratch work documents - it actually has many of the features of a full word processor along with special formatting for mathematical symbols. The cells we have used so far have been specific types called input cells and output cells. There are also “text cells” for normal typing as well as cells meant for special styles like titles, section headers, and so on. The easiest way to create these is through the Writing Assistant Palette, which can be found opened by going to the Palette menu and selecting “Writing Assistant”. This brings up a floating palette that you can move around for convenience. The most important parts of the Writing Assistant are the top two panes.

The top pane controls the creation and formatting of cells and looks something like this:



*the writing and formatting pane of the Writing Assistant*

In terms of formatting your notebook as you would mostly likely make frequent use of the “Section cells” (which includes breaks of different kinds for main sections and subsections) and the standard Text cell (the top selection under “Text Cells”). These types of cells are pure text

cells - typically you cannot do Mathematica calculations in them (the calculations need to be in an input cell). The easiest way to mix calculations up with text is to use a Text cell to contain your text, then click with the mouse below the text cell and start the typing needed for calculation (your typing will automatically start an input cell). Once your calculation is complete, click below it with the mouse to start a new cell and then pick “Text Cell” from the menu again to create a new text cell and continue typing:

**This is a title cell**

---

**This is a Section heading cell**

I am using this text cell to type in normal text

In[1]:= **t = 901;**  
**t ^ 4**

Out[2]= 659 020 863 601

This is a new text cell where I continue to type

- **This is a subsection cell**

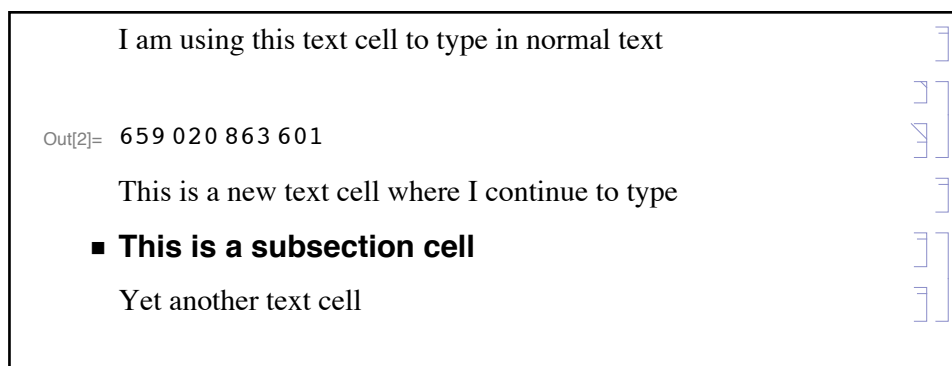
Yet another text cell

---

**Here's a new section cell**

*using the formatting available from the palette*

By default the cell brackets on the right will not be shown when printed - if you want them to be shown you have to specifically select that in the “Printing Options” submenu (which can be found under the “Print” part of the “File” menu). In practice you will also not want to show the input cell for a given calculation, just the output cell. To hide the input cell you can “close” it. To do this first select the cell bracket for the input cell you want to close. Then go to the “Cell Menu” and down to “Cell Properties”. The first option there is “Open” (checked by default) - simply click it to uncheck it and the cell will close:



*the input cell is now hidden/“closed”*

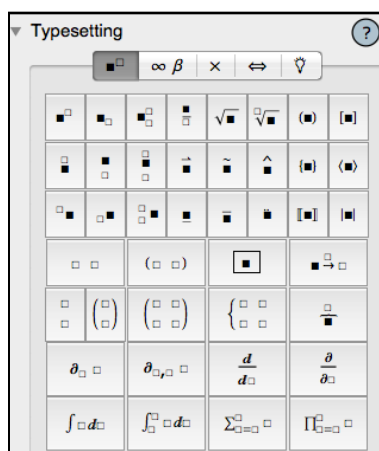
The Writing and Formatting pane also is where you will find many common format options you would use in a word processor - character font and size, line justification and spacing, and so on.

Inside a text or section cell you can also create a special sort of subcell called an “Inline Math Cell”. You can create this by using the key combination Control-9 or by selecting it from the Math Cells heading of the Writing and Formatting pane. The inline math cell will appear as an empty square with a colored background inside the other cell - just click on it and start to type:

*an inline math cell*

Notice that the single characters  $x$  and  $y$  are given the standard “math style italic” but the word velocity uses a standard style - if you want a letter to be given the “math style” you have to have it separated from other letters either by spaces or other symbols.

If all you could do with the inline math cell was type in simple formulas it wouldn’t be very useful. But the inline math cell also lets you use the second major pane from the Writing in Formatting Palette - the Typesetting pane:



*the Typesetting pane*

The buttons in the Typesetting pane create templates for all sorts of mathematical notation and constructs - powers, subscripts, roots, fractions, sums, matrices, and much more. To insert these just place the cursor where you want the template and click the appropriate button (almost all the templates also have a keyboard equivalent which you can see by hovering the mouse cursor over the template). These templates work both in inline math cells and input cells - in fact using any template starts an inline math cell of its own if you are not in one already. The basic Typesetting pane also can be toggled to alternate panes to insert special symbols (like those for pi or infinity), operators (like addition or integral symbols), and much more. By using the Typesetting pane together with the Writing and Formatting pane you can create documents as good as many other programs:

### Using the Quadratic Formula

We can solve any quadratic equation by using the so-called “Quadratic Formula”, which works as follows:

Let  $a$ ,  $b$ , and  $c$  be real numbers with  $a \neq 0$ . Then the solutions to

$$ax^2 + bx + c = 0 \text{ are } x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

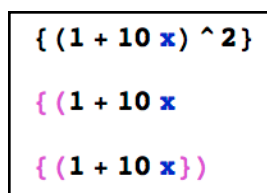
*using the Writing Assistant to create high quality text*

Before we can go on to introduce more Mathematica commands in the next section there is one more formatting issue we need to introduce - the “split” nature of how Mathematica deals with the equal sign and parentheses.

When dealing with the equal sign there are three common ways we use the symbol in mathematics and we determine which is meant using context. Computers are much worse than we are when it comes to interpreting context so Mathematica uses different versions of the equal sign to make it clear what the context is. When we say “ $x=7$ ” we usually mean we are assigning  $x$  the value 7 and they can be used interchangeably. In Mathematica, this is the standard  $=$  symbol (which we used to define variables). In mathematics we also use the equal sign for equations like  $2x + 3y = 5$ . We don’t mean that  $2x + 3y$  is synonymous with 5; we understand the equation is a statement which can be true or false depending on the values used for the variables ( $2x + 3y = 5$  is false when  $x$  is 0 and  $y$  is 1 but true when  $x$  is 1 and  $y$  is 1). When defining equations in Mathematica we use a double equal sign, so we would enter that equation as  $2x + 3y == 5$ . The third main use of the equal sign is in defining functions like  $f(x) = \sin(x)$ . In this case mathematically we are using the equal sign to define a transformation or replacement. In Mathematica we use either the standard  $=$  or  $:=$  to define functions ( $:=$  being more common to distinguish it from defining a variable). Note that these different versions of equality only need to be used in input cells - if you are typing an equation or function in a text cell (which is not going to be evaluated) you can just use the regular equal sign.

We have the same problem with parentheses - the symbols ( ) have many different uses and we have been trained to deduce which is meant from the context. For example consider the following function you might use in parametric graphing:  $f(x) = (\sin(x), (1+x)^2)$ . The parentheses here have three different meanings and again Mathematica uses a separate notation for each. On the left hand side of the equation the parentheses represent a function applied to a variable. Mathematica uses square brackets for all functions (so in Mathematica it would appear as `f[x]`). On the right hand side the outer parentheses indicate an ordered pair or set of values. Mathematica uses curly brackets { } to represent ordered sets or lists. And finally in the quantity  $(1+x)^2$  the parentheses denote arithmetic grouping, for which we use the regular ( ) symbols. So in Mathematica the equation mathematical definition  $f(x) = (\sin(x), (1+x)^2)$  is actually entered as `f[x_] := { Sin[x], (1+x)^2 }` (the underscore after the x is part of how Mathematica defines a function - which we will discuss how to define functions in a later section).

One of the things you have to be careful about when using parentheses/brackets/braces in Mathematica is to make sure that each one is “closed” - that is, each (, {, and [ has a matching ), }, or ] respectively. Mathematica automatically helps you track parenthetical groupings (as well as problems with mismatches) by coloring parentheses problems instead of using the standard black:



```
{ (1 + 10 x) ^ 2 }
{ (1 + 10 x
{ (1 + 10 x) )
```

*mismatched and unmatched brackets can cause many problems so they get their own color*

In the picture the first line has closed and matched grouping symbols so they all appear in black - everything is correct and ready to go. The middle line has unmatched brackets and the last line has mismatched brackets. Both unmatched and mismatched brackets get the same color to indicate there is a problem. In the second line if you continued to type and entered a ) to enclose the  $1 + 10x$  the matching left parenthesis would briefly flash to indicate “this is the parenthesis you just closed” - this is very useful when you start to work with complex expressions which have all sorts of nested parentheses.

## Section 2.2 Homework – Working with Mathematica Notebooks

In this and other Mathematica homework sets do your work in a Mathematica notebook and print it out. Use a title cell to put in your name and section information and then separate each problem using its own section cell. Use text cells and other formatting to make the notebook as easy to follow as possible.

- 1) Explain the difference between the front end and the kernel.
- 2) Use Mathematica to find the  $3^{60}$  both exactly and as an approximation.
- 3) Explain the difference between what you get in Mathematica when you evaluate  $\text{Cos}[2\pi/5]$  and what most calculators would give you for  $\cos(2\pi/5)$ .
- 4) What is the purpose of the semi-colon in Mathematica?
- 5) What do %, %, and %%% stand for?
- 6) In a new cell evaluate  $2+3$ . In the following cell evaluate  $\%^2$ . Then re-evaluate this second cell 4 times (you can re-evaluate the cell by clicking the cursor anywhere back up in it at hitting Enter on the keypad). Explain why the result would look wrong to someone glancing at the notebook.
- 7) In the Writing Assistant the only type of Math Cell that we talked about was an inline math cell. What other kinds are there? Give examples of each new kind of cell where each new cell contains at least 3 lines.
- 8) Use the Typesetting portion of the Writing Assistant to duplicate the following symbols and mathematical quantities in inline math cells:

$$\frac{x^2}{\sqrt{x^3+1}}$$

$$\sqrt[5]{(2y-1)^3}$$

$$\begin{pmatrix} 4 & 3 \\ 2 & 1 \end{pmatrix}$$

$$\begin{cases} x^2 & x \geq 1 \\ 1-x & x < 1 \end{cases}$$

$$x \in \mathbb{R}$$

$$z \in A \implies z \in A \cap B$$

$$\overset{\longleftrightarrow}{AB}$$

- 9) Explain the difference between the three different Mathematica notations for the standard equal sign.
- 10) Explain the difference between the  $()$ ,  $\{ \}$ , and  $[ ]$  grouping symbols.
- 11) What is the symbol  $\equiv$  used for? What is another way to achieve the same effect as  $x \equiv$ ?

## Section 2.3 - Basic Mathematica Functions

Now that we have the basics of notebooks and formatting it is time to introduce the Mathematica notation for many of the common functions. Some of these functions will only accept integer inputs and others will accept real or even complex number inputs. The complex number  $i$  can be entered simply as a capital I or as a special symbol by using the key sequence Esc-ii-Esc ( we will use Esc-...-Esc to indicate the key sequence “Escape key followed by different keys followed by the Escape key”, with the - indicating that the Escape key starts and stops the sequence and but the - is not typed itself).

Starting with common functions that deal with integers:

`Mod[a,b]`: Returns the remainder when  $a$  is divided by  $b$ . `Mod[20,6]` returns the value 2 as 6 goes into 20 3 times with remainder 2. (it turns out that `Mod` works if  $b$  is a nonzero real or complex number but in practice  $b$  is typically a positive integer).

`BaseForm[x,b]`: `BaseForm` writes the integer  $x$  in base  $b$  notation (the most common bases are 2, 8, 10, and 16 but any natural number from 2-36 will work for the base). The base you use will end up following the output in a subscript. For example to write 120 in base 2 notation use `BaseForm[120,2]` to get  $1111000_2$ .

`FactorInteger[n]`: Returns a list of the prime factors of  $n$  along with their multiplicities. `FactorInteger[20497400]` returns  $\{\{2,3\},\{5,2\},\{7,1\},\{11,4\}\}$ , indicating that  $20497400=2^3*5^2*7*11^4$ . For those who like to work over the complex numbers `FactorInteger[n,GaussianIntegers→True]` will factor over the “Gaussian Integers”, the set of complex numbers  $a + bi$  where  $a$  and  $b$  are integers. The notation  $\rightarrow$  is VERY common in Mathematica as it is used for replacements and setting options. You enter the symbol  $\rightarrow$  in Mathematica as `->`, which will automatically be converted to a nice-looking arrow as you continue to type.

`Divisors[n]`: A list of the positive divisors of  $n$ . So `Divisors[6]` returns  $\{1,2,3,6\}$ .

$n!$ : the factorial of  $n$  (recall that by definition  $0!=1$  and for integers  $n>0$   $n!=1*2*3*…*n$ ). So  $6!=1*2*3*4*5*6=720$ . The factorial can give you a result for any input (real or complex) except for negative integers using a special function (the gamma function) although in those cases you usually need to numerically estimate the result.

$n!!$ : The “even/odd” factorial. If  $n$  is odd,  $n!!$  is  $1*3*5*…*n$ . If  $n$  is even,  $n!!=2*4*6*…*n$ . Mathematica can extend  $n!!$  to take the same range of inputs as  $n!$ .

`Binomial[n,k]`: The number of ways to select  $k$  objects from a set of  $n$  without regard



to order and without replacement. To see how many 5 card poker hands can be drawn from a fresh deck of 52 cards, you would use `Binomial[52,5]` and get 2,598,960 as your answer. You can use non-integer values as well as long as they do not cause problems for the factorials in the mathematical definition  $\text{Binomial}[n,k] = \frac{n!}{(n-k)!k!}$ . The standard mathematical notation for this computation that you find in textbooks is  $\binom{n}{k}$ .

`GCD[a,b]`: Finds the greatest common divisor of a and b. So `GCD[1023,32767]` yields 31.

`LCM[a,b]`: This gives the least common multiple of a and b. `LCM[12,20]` will yield 60.

`Prime[n]`: This function returns the  $n^{\text{th}}$  prime number - `Prime[1]` yields 2, `Prime[2]` yields 3, and so on. On fast computers Prime works well even if n is in the millions-billions range.

`PrimeQ[n]`: This function returns True if n is prime and False otherwise (it ignores any negative sign, so `PrimeQ[2]` and `PrimeQ[-2]` both yield True). Mathematica functions that end in Q usually denote ones whose output is either True or False (these being perfectly good values just as 7 and -1 are).

Mathematica also has an extensive suite of functions that deal with real and complex numbers. If you give Mathematica exact inputs for these functions (i.e. ones which don't use a decimal point, so to represent 1.1 exactly you have to use 11/10) it will give you exact outputs where possible. If you do use approximate numbers by using a decimal point Mathematica will track the number of decimal places to keep automatically - for example `2.34+7.1999999` will give the result 9.54 to match the lesser precision of the input 2.34.

Here some of the more common functions that deal with real and complex numbers:

`Abs[x]`: The absolute value of x, for both real and complex numbers x. So `Abs[-4]` would return 4 and `Abs[3+4I]` yields 5. Version 11.1 of Mathematica introduced the similar command `RealAbs[x]`, which only works for real values of x. `RealAbs` works better in settings (like elementary calculus) that only involve real numbers.

`Sign[x]`: The sign of a number (-1, 0, or 1 for real numbers). For nonzero complex numbers `Sign[z]` is defined as  $z/\text{Abs}[z]$  (so `Sign[5+12I]` gives  $5/13 + 12/13 I$ ). Version 11.1 of Mathematica introduced the similar command `RealSign[x]` (which like `RealAbs[x]` only works for real values and is better suited to subjects like calculus).

`Re[x]`: The real part of x. So `Re[3-4I]` is 3.

Im[x]: The imaginary part of x. Im[3-4I] returns -4.

Conjugate[x]: The complex conjugate of x. Conjugate[5+2I] is 5-2I and Conjugate[3] is 3.

Ceiling[x]: x rounded up. Ceiling[5.7] is 6 and Ceiling[2] is 2.

Floor[x]: x rounded down. Floor[-3.4] is -4 and Floor[3] is 3.

Round[x]: x rounded to the nearest integer. Round[4.2] is 4, and Round[ 9.9] is 10. You can also use Round[x,a] to give x rounded to the nearest multiple of a. You can use this version of Round to get both approximate-number and exact-number roundings; Round[ Pi, .1] will give you 3.1 and Round[Pi, 1/10] will give you 31/10.

N[x,n]: A numerical estimate of x to n places (we used this for  $\pi$  in Section 2.2).

Sum[ *expression*, {*variable*, *start*, *finish*, *step*}]: Find the sum of *expression* as the *variable* goes from *start* to *finish* by going up by *step* each time (if *step* is not given, by default it is 1). For example to add the odd numbers from 1 to 21 you could use Sum[k, {k,1,21,2}] or Sum[2k+1, {k,0,10}]. A negative value for *step* will have the sum counting down from *start* to *finish* (in which case the *start* should be the larger number, as in Sum[  $k^2$ , {k,10,1,-1} ]). Sums of exact quantities will be returned as exact quantities (Sum[ $x^k$ , {k,2,4}] would return  $x^2 + x^3 + x^4$ ). The end of a sum can even be infinite, entered as the word Infinity. You can get the same sort of results by using the summation symbol from the Writing Assistant.

Product[*expression*, {*variable*, *start*, *finish*, dk}]: Same as Sum, except for multiplication rather than addition.

Exp[x]: The natural exponential  $e^x$  (the exponent can be real, complex or another expression like Abs[x] or -x<sup>2</sup>). The number  $e$  is represented in Mathematica by the capital letter E (or the key combination Esc-ee-Esc to get the special symbol), so Exp[x] could be replaced by E<sup>x</sup>.

Log[x]: The natural logarithm of x (NOT “log base 10 of x”).

Log[b,x]: The logarithm base b of x (i.e.  $\log_b(x)$ ). So the base 5 logarithm of 24 is Log[5,24].

Log10[x]: The base 10 log (or common log) of x. Log10[1000] would be 3.

Sqrt[x]: The square root of x, for both real and complex numbers. Sqrt[4] yields 2, Sqrt[-5+12i] gives 2+3I, and Sqrt[8] results in 2Sqrt[2]. Mathematica will simplify

whatever square roots it can. You can get a square root symbol from the Writing Assistant or by the key combination Ctrl+2.

The square root can also be represented by the power  $1/2$  - Sqrt[x] will work exactly the same as  $x^{(1/2)}$ . Other roots can be a bit trickier though. For example you would expect the cube root to be represented by  $x^{(1/3)}$  - and for non-negative values of  $x$  it is. But Mathematica assumes that the cube root (or any odd root) of a negative number is complex (remember, a “cube root of  $a$ ” is just a solution to  $x^3 - a = 0$  - and over the complex numbers there are 3 distinct solutions if  $a$  isn't 0). The same is true if you use the Writing Assistant to get a general radical symbol (or the key sequence Ctrl+2 - Ctrl+5). The reason Mathematica makes the complex choice for cube roots (in fact for all odd roots) is that it greatly simplifies Mathematica's internal algorithms for solving algebraic equations and is consistent with the definition of these roots in higher mathematics. It's definitely inconvenient for regular computations, though - a graph of  $x^{(1/3)}$  from -8 to 8 would only show values running from 0 to 8 as the other values would involve  $i$ . Prior to Mathematica 9 you had to use combinations of other commands (like Sign[x]Abs[x]^(1/3)) to get “regular” cube roots. Mathematica 9 and later greatly simplifies this with the command CubeRoot and the more general command Surd:

CubeRoot[x]: This is the standard real-number cube root. The input  $x$  must be a real number.

Surd[x,n]: The standard real-number  $n$ -th root of  $x$  (so Surd[x,3] is the same as CubeRoot[x]). The input  $x$  must be a real number, and even roots of negatives will return “Indeterminate” instead of complex numbers. The surd symbol looks like a traditional radical with a small vertical line segment hanging off the end of the radical. You can get the surd symbol via the key combination Esc-surd-Esc.

```
In[3]:= (-8.) ^ (1 / 3)
Out[3]= 1. + 1.73205 i

In[4]:=  $\sqrt[3]{-8.}$ 
Out[4]= 1. + 1.73205 i

In[5]:= CubeRoot[-8]
Out[5]= -2

In[6]:= Surd[-8, 3]
Out[6]= -2

In[7]:=  $\sqrt[3]{-8}$ 
Out[7]= -2
```

*the difference between fractional powers/radicals and CubeRoot/Surd*

Mathematica also includes the trigonometric functions and their inverses. As is common in higher mathematics Mathematica works in radian measure rather than degree measure (as radians are preferable for doing calculus). This means you will need ready access to the number  $\pi$ , which you get in Mathematica as Pi (or the key combination Esc-pi-Esc). Mathematica also includes the radian-to-degree conversion factor  $\pi/180$  as the word Degree. The trigonometric functions and their inverses are easily used with the following commands:

Sin[x]: The sine of x (ArcSin[x] is the inverse sine).

Cos[x]: The cosine of x (ArcCos[x] is the inverse cosine)

Tan[x]: The tangent of x (ArcTan[x] is the inverse tangent - ArcTan[x,y] finds the arctangent adjusted for the Quadrant of the point (x,y))

Cot[x]: The cotangent of x (ArcCot[x] is the inverse cotangent with range  $(-\pi/2, \pi/2)$ )

Sec[x]: The secant of x (ArcSec[x] is the inverse secant, with range  $[0, \pi]$ )

Csc[x]: The cosecant of x (ArcCsc[x] is the inverse cosecant)

Here are some examples of how you use the trigonometric functions in Mathematica:

<b>Tan[ArcSec[5]]</b>	<b>Sin[Pi / 4]</b>
$2\sqrt{6}$	$\frac{1}{\sqrt{2}}$
<b>Cos[Pi / 12]</b>	<b>ArcTan[-1 / Sqrt[3]]</b>
$\frac{1 + \sqrt{3}}{2\sqrt{2}}$	$-\frac{\pi}{6}$
<b>N[ArcSin[-3]]</b>	<b>Cot[60 Degree]</b>
$-1.5708 + 1.76275 i$	$\frac{1}{\sqrt{3}}$

*trigonometry functions in Mathematica*

Notice that not only does Mathematica know the exact values for common angles it knows the exact values for angles like  $\pi/12$  that normally aren't covered in a standard trigonometry class. You can also see that the inverse sine of a number outside the range  $[-1,1]$  is a complex number (Mathematica understands the trigonometric functions over the complex numbers so you can have complex inputs or outputs).

Before moving on it is worth mentioning that the list of functions given here is not even remotely complete - this is just a sampling of the ones that come up most often in arithmetic, algebra, and pre-calculus. As you use Mathematica you will probably encounter functions

you've never heard of before (usually as the results of a computation you've just done). To see what a function does you can either use the Documentation Center from the Help menu or you can get help in the notebook itself by using the `?` symbol. `?name` will tell you the definition of the exact symbol *name* (so `?Limit` would give you the definition of the `Limit` command). If you're not sure of the exact name of a function you can get a list of possibilities by using one or more `*` symbols as wildcards. Evaluating `?Euler*` would give you a list of all functions that start with Euler and `?*Plot*` would give you a list of all functions which include the word Plot somewhere in their name. Clicking on one of the functions in the list will give you the same help statement that you could get from using the `?` symbol.

```
In[33]:= D[x!, x]
```

```
Out[33]:= Gamma[1 + x] PolyGamma[0, 1 + x]
```

```
In[27]:= ? Gamma
```

Gamma[z] is the Euler gamma function  $\Gamma(z)$ .  
 Gamma[a, z] is the incomplete gamma function  $\Gamma(a, z)$ .  
 Gamma[a, z<sub>0</sub>, z<sub>1</sub>] is the generalized incomplete gamma function  $\Gamma(a, z_0) - \Gamma(a, z_1)$ . >>

```
In[28]:= ? *Gamma*
```

▼ System`

EulerGamma	LogGamma
ExpGammaDistribution	LogGammaDistribution
Gamma	PolyGamma
GammaDistribution	QGamma
GammaRegularized	QPolyGamma
InverseGammaDistribution	StieltjesGamma
InverseGammaRegularized	

*encountering a new function and using ? to find out more information*

Using the functions in this section is usually pretty straightforward but it can be tedious if you need use them to do repeated number crunching. Think of trying to evaluate  $(\text{Re}[x] + \text{Log}[x]) / (\text{Sqrt}[x] + \text{Im}[x])$  for  $x=10, 11, \dots, 20$ . That's a lot of typing (imagine having to type  $(\text{Re}[10] + \text{Log}[10]) / (\text{Sqrt}[10] + \text{Im}[10])$  and so on, each in its own cell). Fortunately there's an easy shortcut - define the formula in a new variable one time with  $x$ 's in it and then substitute the values of  $x$  into that variable. The basic notation for substituting/replacing a "form" in an expression with a particular value is "expression or expression name /. form→value" (think of the /. as standing for "with the replacement"). So if you evaluate  $x^2 /. x \rightarrow 9$  you would get 81 as the result and  $x^2 /. x \rightarrow 1+t$  would give you  $(1+t)^2$ . So to get  $(\text{Re}[x] + \text{Log}[x]) / (\text{Sqrt}[x] + \text{Im}[x])$  for  $x=10, 11, \dots, 20$  you could define `mymess=(Re[x]+Log[x])/(Sqrt[x]+Im[x])` and then just evaluate `mymess /. x→10`, then `mymess /. x→11`, and so on - which is a LOT faster than retyping the functions over and over and over again. The /. notation extends to plugging in values for more than one variable through the use of ordered lists (given by `{}`). If you want to substitute 2 for  $x$  and 10 for  $y$  in the expression  $x^3/\text{Log}[y]$ , enter in `x3/Log[y] /. {x→2, y→10}`.

The notation even extends to plugging in several different values for  $x$  to get a list of answers. If you wanted to find the squares of 2, -1, and 1 all at once you could use the command  $x^2/$ .

$\{\{x \rightarrow 2\}, \{x \rightarrow -1\}, \{x \rightarrow 1\}\}$  (pay careful attention to the extra internal braces - if you try to use  $\{x \rightarrow 2, x \rightarrow -1, x \rightarrow 1\}$  only the first rule  $x \rightarrow 2$  will be used). This “list of replacement rules” may seem a little cumbersome at first but with some practice it becomes second nature. Replacement rules even work for functions. For example if you wanted to replace  $\cos^2(x)$  with  $1 - \sin^2(x)$  you could use the rule  $\text{Cos}[x]^2 \rightarrow 1 - \text{Sin}[x]^2$ . The only downside of this is that it would only affect the exact form  $\text{Cos}[x]^2$  - it would not affect terms like  $\text{Cos}[x]^4$  even though  $\text{Cos}[x]^4$  could be written as  $(\text{Cos}[x]^2)^2$ .

When you use the  $/$  symbol to do a replacement you should be aware that the replacements are done one time only and then the result is shown. There may be times when you want the replacement to be done repeatedly until the answer no longer changes. For example suppose you have the list  $\{x, y, z\}$  and the replacement list  $\{x \rightarrow 5, y \rightarrow 10x, z \rightarrow y+3\}$ . If you evaluate  $\{x, y, z\}/\{x \rightarrow 5, y \rightarrow 10x, z \rightarrow y+3\}$  you will get the list  $\{5, 10x, y+3\}$ . This isn't really the best you can do - if  $x$  is 5 and  $y$  is  $10x$  then  $y$  really should be 50 (and likewise  $z$  really should be 53). To arrive at these values you need to use the replacements more than once. To do a repeated replacement you need to use  $//$  instead of  $/$  - this will use the replacements over and over until the answer doesn't change:

$\{x, y, z\} /. \{x \rightarrow 5, y \rightarrow 10x, z \rightarrow y+3\}$ $\{5, 10x, 3+y\}$ $\{x, y, z\} //. \{x \rightarrow 5, y \rightarrow 10x, z \rightarrow y+3\}$ $\{5, 50, 53\}$
---

*the difference between one-time replacements and repeated replacements*

## Section 2.3 Homework – Basic Mathematica Functions

- 1) Find the base 8, base 2, and base 16 representations of the number 27182831.
- 2) Find all the factors of 2,194,033.
- 3) Find all the prime factors of  $3^{40}-11$ .
- 4) Factor the prime factorization of 219,403,300. Give both the output of the appropriate command as well as how you would write the answer in standard form.
- 5) Repeat problem 4 but do the factorization over the Gaussian integers.
- 6) Explain the difference between  $n!$  and  $n!!$ . Which is larger,  $(10!)!!$  or  $(10!!)!$ ? (to determine the latter I recommend you find a way to determine which is larger without actually having the exact numbers printed in the notebook - one of them is over 11 million digits long!).
- 7) What is the millionth prime number?

- 8) Round  $5^{21}$  to the nearest multiple of 7.
- 9) Use the commands PrimeQ, FactorInteger, and Divisors to determine if the number 3,628,801 is prime. Explain how the output of each command answers the question.
- 10) What is the quotient and remainder when 406 is divided into 3,141,159? (one of our commands explicitly computes the remainder - the quotient you get by rounding a fraction down and we have a command for rounding down)
- 11) Estimate the number  $e$  to 20 decimal places.
- 12) What is the absolute value of  $-3$ ?  $2+3i$ ?  $\pi-4$ ?
- 13) Pick two 8 digit numbers  $a$  and  $b$  and compute their greatest common divisor and least common multiple. Compare the two products  $ab$  and  $\text{lcm}(a,b)\text{gcd}(a,b)$ . Repeat this 5 times and make a conjecture.
- 14) If  $z = (1 + 3i)^{10}$  find the real part of  $z$ , the imaginary part of  $z$ , and the conjugate of  $z$ .
- 15) Find the sum, difference, product, and quotient of  $10+37i$  and  $39-11i$ .
- 16) Estimate the square and cube roots of 40 to 15 decimal places.
- 17) Find the sum  $1+3+5+7+9+11+\dots+501$  without entering the sum long-hand.
- 18) Find the product  $1*5*9*13*\dots*41$  without entering the product in long-hand.
- 19) Numerically estimate  $e^4$  and the natural logarithm of 10.
- 20) How many ways can you draw a 7 card hand from a 52 card deck?
- 21) Explain the difference between  $(-10)^{1/3}$  and  $\text{CubeRoot}[-10]$ . Numerically estimate each.
- 22) Numerically estimate the real fifth root of -90. Do this twice - once using the command for surds and once using the symbol for surds.
- 23) Find the values of the 6 trigonometric functions at the angle  $11\pi/12$ .
- 24) Find the values of the 6 trigonometric functions at the angle  $215^\circ$ .
- 25) Find the inverse sine, cosine, and tangent of  $1/2$ .
- 26) Approximate the inverse sine, inverse cosine, and inverse tangent of  $1/3$ .
- 27) Approximate the inverse sine, inverse cosine, and inverse tangent of 10.
- 28) Use the expanded version of inverse tangent to find the angle whose terminal ray goes through the point  $(-2,2)$ . Is this different than the arctangent of  $2/-2$ ?
- 29) Explain the difference between the  $/.$  and  $//.$  symbols for replacements.
- 30) Use replacement notation to plug the value 5 into  $\frac{x^3}{x^2 + 1}$ .
- 31) Use replacement notation to replace  $x$  with  $t^2$  in the expression  $\frac{x^3}{x^2 + 1}$ .
- 32) Use replacement notation in a single command to plug in the values  $0, \pi/6, \pi/4, \pi/3$ , and  $\pi/2$  into  $\sin(x)$ .
- 33) Use replacement notation in a single command to plug the points  $(2,1), (3,0), (-4,1)$ , and  $(0,3)$  into the expression  $\frac{y}{x^3}$  ( $x$  representing the first coordinate and  $y$  the second).
- 34) Use repeated replacement notation to simplify the list  $\{x,y,z,t\}$  given that  $y = 3, x = t^2, z = x + 5$  and  $t = \frac{y}{y + 1}$ .

- 35) Evaluate the following replacement command (which involves a twist):  $\{x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9\} /. x^n \rightarrow n*x^{(n-1)}$ . How do you think this works? Based on the output how would you translate the rule  $x^n \rightarrow n*x^{(n-1)}$  into an English sentence?
- 36) Use the ? command to quickly look up the command Integrate.
- 37) What does the EulerPhi command do? (you will need to look under “more information” in the documentation.
- 38) In calculus the command for “take the derivative of *formula* with respect to *x*” is  $D[formula, x]$  (if you haven’t had calculus, the notation for the derivative of  $f(x)$  is  $f'(x)$ ). If you ask Mathematica to take the derivative of Abs[x] or Sign[x] you just get Abs'[x] and Sign'[x] (i.e. it symbolically does the derivative without really telling you anything). What does Mathematica say when you take the derivative of RealAbs[x] and RealSign[x]?



## Section 2.4 - Basic Graphing in Mathematica

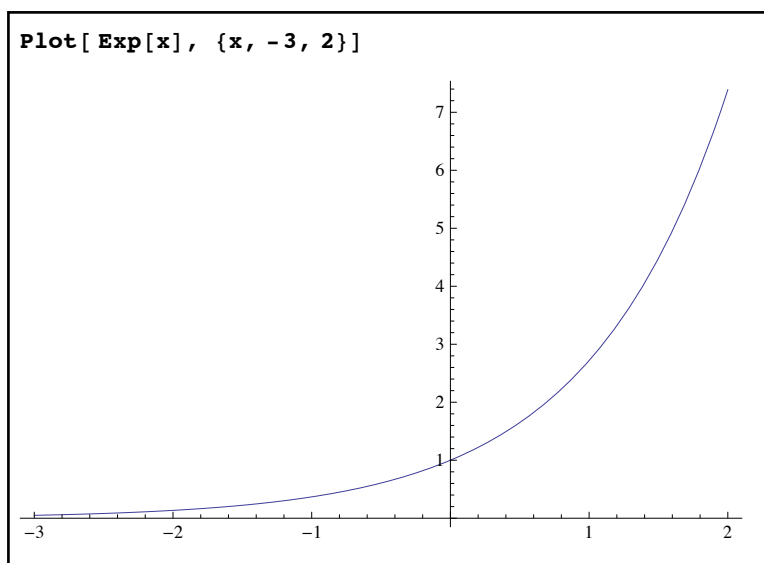
Now that we have covered lots of functions to use in Mathematica one of the first things you may need to do with them is create graphs. Mathematica has a wide selection of graphing commands, each tailored to a specific use. The three most common uses are graphing one or more functions (“graph  $x^2$  and  $x^4$  together”), graphing an equation (“graph the solutions to  $x^2 - xy + 2y^2 = 1$ ”), and graphing a 2D region “graph the area above  $y = x^2$  but below  $y = x + 1$ ”. The commands for these three cases are `Plot`, `ContourPlot`, and `RegionPlot`. Each one of these commands also has a set of options you can use to customize the graphs - we will introduce the basic options here and leave the more advanced options for a later section.

The most basic kind of graphing problem is simply to “graph this formula”. The command for this is `Plot` and it has two main forms depending on whether you want to graph a single formula or several at once:

`Plot[function, {variable, start, finish}]` creates the graph of the equation “ $y=$ *function*” as the *variable* goes from the *start* value to the *finish* value. Mathematica will automatically determine how many points to plot to create a smooth plot, what range of values to use for the vertical axis, what tick marks to use, and the scales on each axis.

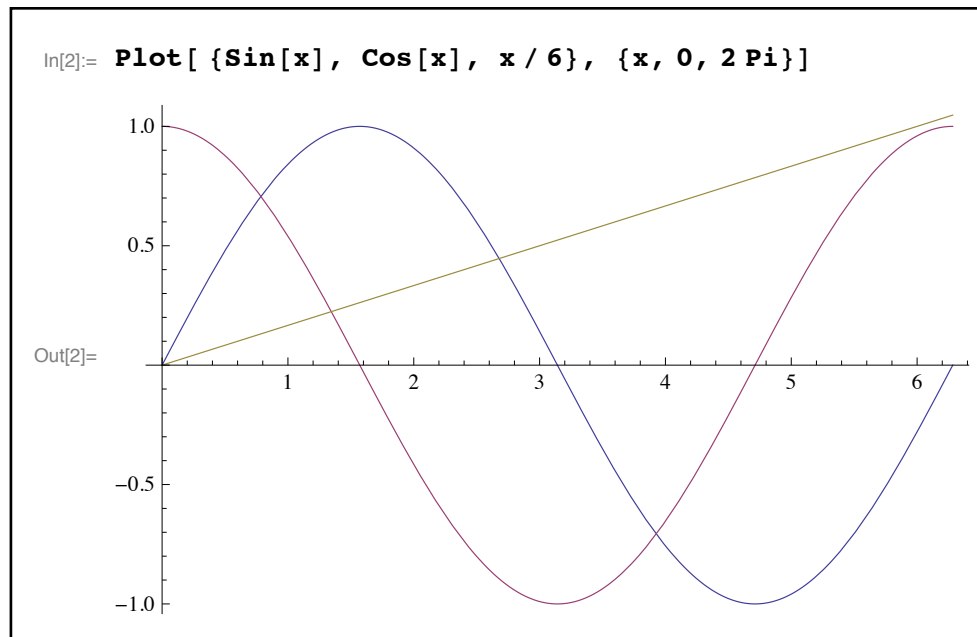
`Plot[list of functions, {variable, start, finish}]` will create a combined graph which includes each function in the list. In addition to all the other automatically determined features Mathematica will assign colors to each graph.

For example if you want to graph  $e^x$  from -3 to 2 you would evaluate `Plot[Exp[x], {x,-3,2}]`:



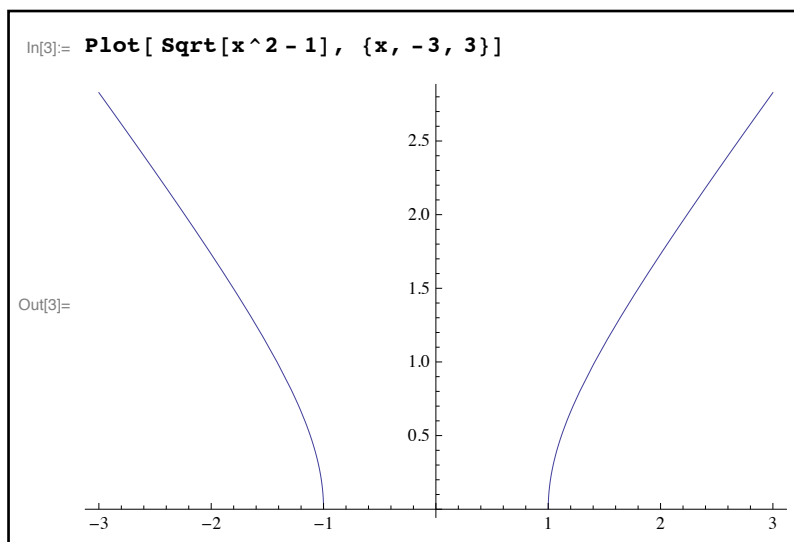
*graphing a single function using the Plot command*

If you would like to graph the functions  $\sin(x)$ ,  $\cos(x)$ , and  $x/6$  together from 0 to  $2\pi$  you would use the command `Plot[ {Sin[x], Cos[x], x/2}, {x,0,2Pi}]`:



*graphing several functions at once using Plot*

If the functions you use in `Plot` aren't defined for all x-values (as is often the case for functions with limited domain such as square roots, logarithms, etc.) Mathematica will do its best to graph the functions where they are defined and simply leave the "undefined" values empty.



*Mathematica doesn't graph in the middle where the square root is undefined*

In general Mathematica does a pretty solid job of choosing basic settings to make the graphs look good. However it's not perfect and so there will be times when you want to override Mathematica's choices. These are done by adding options to the Plot command after the curly braces that define the range of values (so the command will look like `Plot[ something, {variable, start, finish }, option1 →value1, option2 →value2,.....]`). You can see the full list of options available for Plot by evaluating the command `Options[Plot]`, but here are some of the common ones that you would use to control how a graph looks:

**AspectRatio:** The AspectRatio option controls the scale of the units used on the axes. By default Mathematica sets the scales on the axes so the graph fits into a rectangle about 60% wider than it is tall. Using the option `AspectRatio→Automatic` will force Mathematica to use the same scales on both axes (which can be important if you are looking at slopes of lines or the shapes of circular arcs, both of which can be distorted if Mathematica chooses different scales on each axis).

**PlotRange:** PlotRange controls the range of values shown in the graph. If you use the form `PlotRange→{y-min, y-max}` the range of y-values shown will go from the minimum to the maximum. You can also use `PlotRange→All` to try to see all of the y-values in the graph; this can be problematic if the graph has both huge and small values or a vertical asymptote. You can also use PlotRange as a function to see the ranges of values used in a graph, either in the form `PlotRange[ Out[number] ]` or `PlotRange[ variablename]` if you have stored your graph in a variable (say by using something like `mygraph=Plot[..... ]`).

**PlotPoints:** PlotPoints controls how many points Mathematica will use before attempting to “connect the dots” to make the graph. Mathematica usually does a pretty good job of plotting enough points but there are times you might need to increase the number used by setting `PlotPoints→number` (generally you'd try this if the graph is changing height very rapidly or if it is coming to an endpoint). You can set PlotPoints as high as 500 or 1000 without taxing Mathematica very much (although 50 or 100 will do in many cases).

**Axes:** Axes controls whether the axes are shown in the graph. By default Mathematica uses `Axes→True`; using `Axes→False` will remove the axes from the graph.

**Frame:** Frame is similar to Axes in that it controls whether to use a bounding frame for the graph (you can have both axes and a frame, neither, or just one of the two). By default Plot uses `Frame→False`; adding `Frame→True` in your options will show the frame.

**Ticks:** Ticks controls what tick marks are present on the axes (it won't do anything if Axes is set to False). The format is `Ticks→{ list of x-axis ticks, list of y-axis ticks }` (note that each tick “list” will need to be wrapped in its own set of curly braces). To

generate a standard picture of the sine curve you could use `Plot[ Sin[x], {x,0,2Pi}, Ticks→{ {0, Pi/2, Pi, 3Pi/2, 2Pi}, {-1,-1/2, 0, 1/2, 1} } ]`. If you only want to specify ticks for one axis and let the other axis ticks be chosen automatically you can use `Automatic` in place of one of the tick lists. You can also use `None` to suppress one or more sets of tick marks.

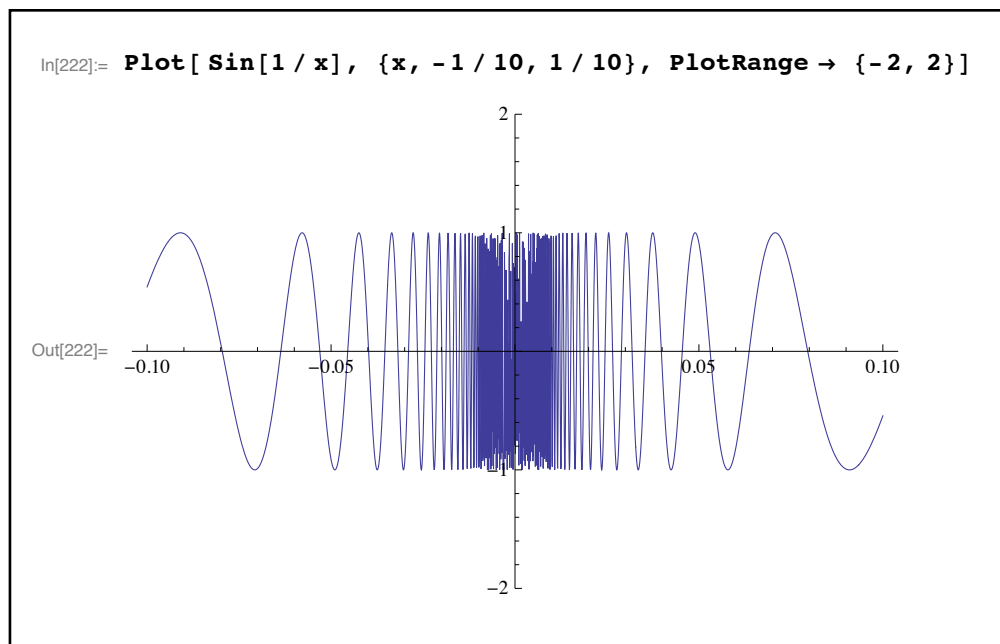
**FrameTicks:** `FrameTicks` controls what tick marks are used on bounding frames (and so does not do anything unless `Frame` is set to `True`). The format for `FrameTicks` is exactly like that of `Ticks`.

**AxesLabel:** `AxesLabel` will place labels on the axes and has the form `AxesLabel→{x-label, y-label}`. The labels can be anything (even other graphs if you want to go to extremes!) but if you want to use text as the labels you need to put the text in quotes (as in `AxesLabel→{"time", "velocity"}`). You can also have a label on just one axis by using `None` as the label for the other.

**PlotLabel:** `PlotLabel` places a label on the entire graph (centered at the top of the graph) when you use `PlotLabel→label`. As with `AxesLabel` you can use anything for the label but text is the most common (for example `PlotLabel→"this is the graph of the sine"`).

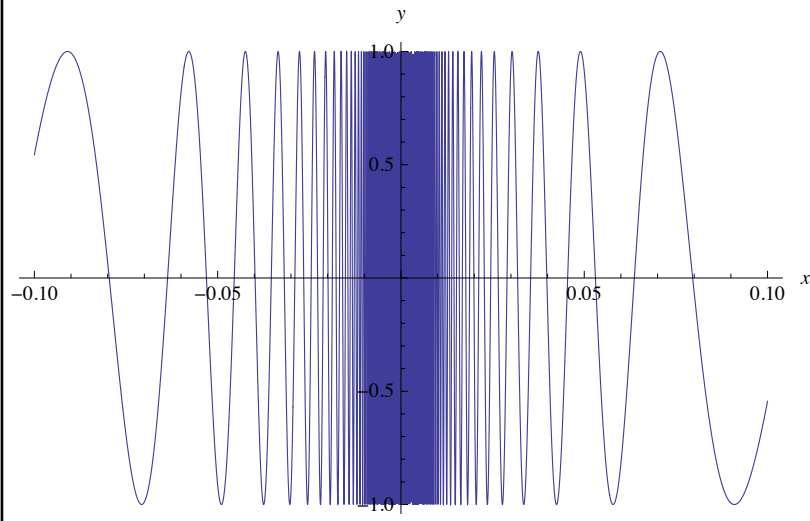
**ImageSize:** `ImageSize` controls the width of the graph in pixels when you use the format `ImageSize→pixelnumber`. This is typically used in more complex settings or when copying or exporting graphics from Mathematica to use in other programs.

Here are some examples of using these options for graphing:



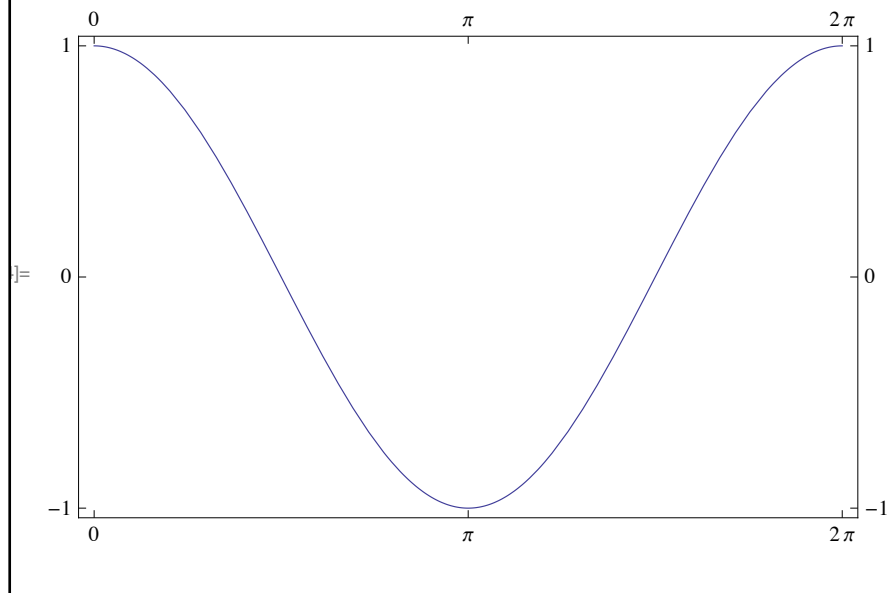
*a rapidly changing sine curve with extra vertical space from PlotRange*

```
Plot[Sin[1/x], {x, -1/10, 1/10}, PlotPoints -> 500, AxesLabel -> {x, y}]
```

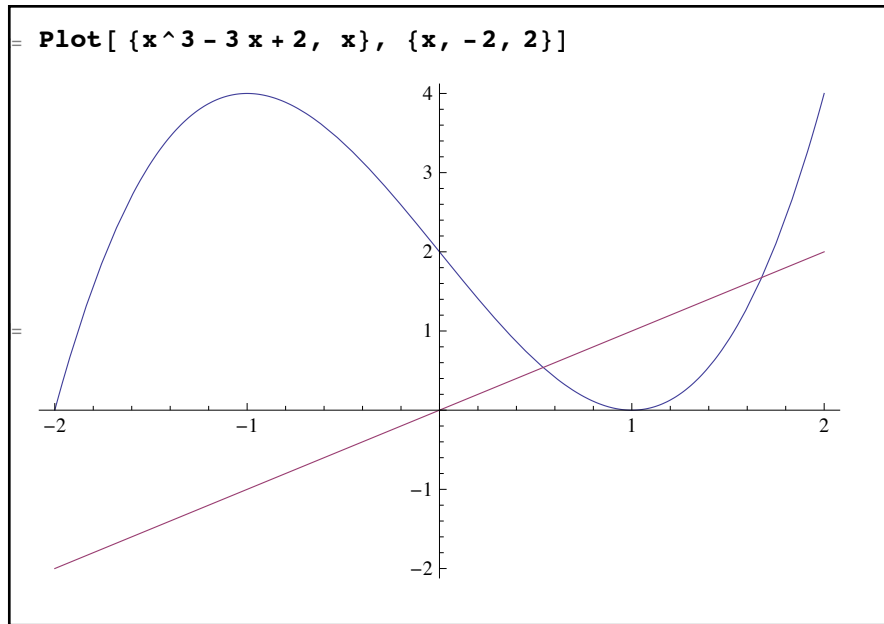


points added with `PlotPoints` to smooth the center and labels added on the axes via `AxesLabel`

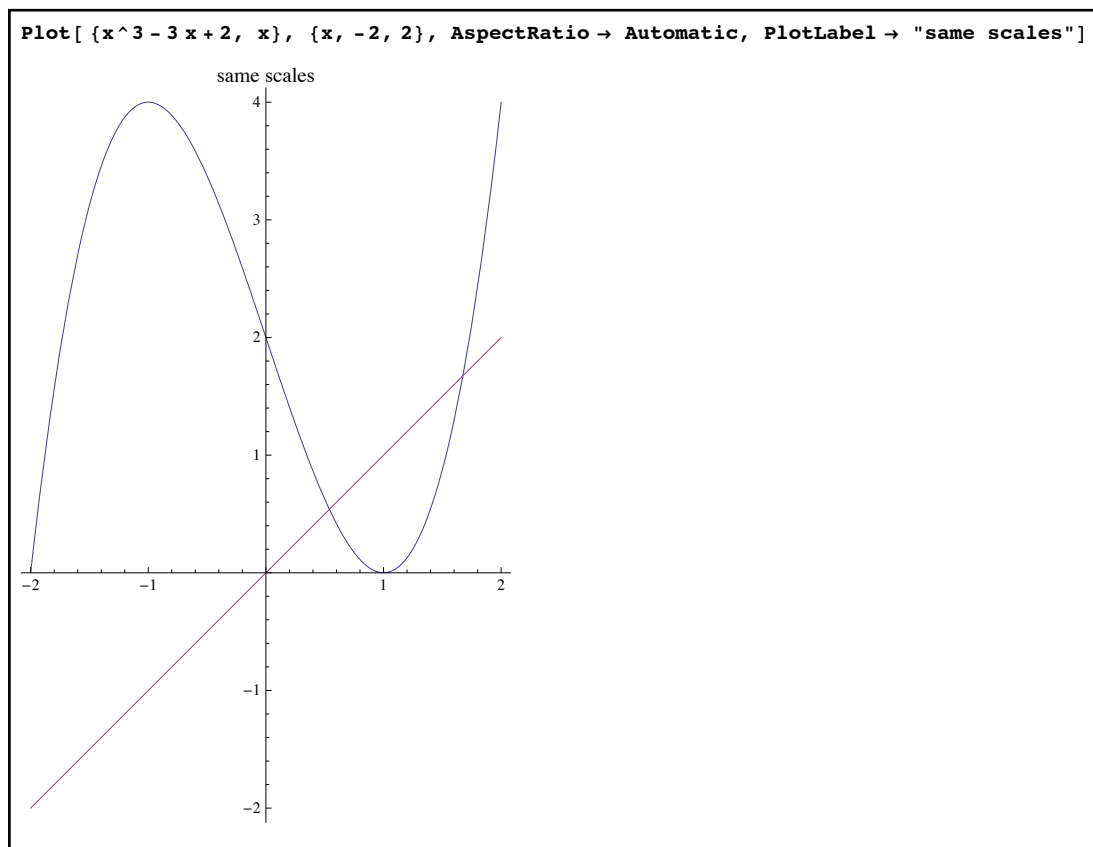
```
Plot[Cos[x], {x, 0, 2 Pi}, Frame -> True,
     Axes -> False, FrameTicks -> {{0, Pi, 2 Pi}, {-1, 0, 1}}]
```



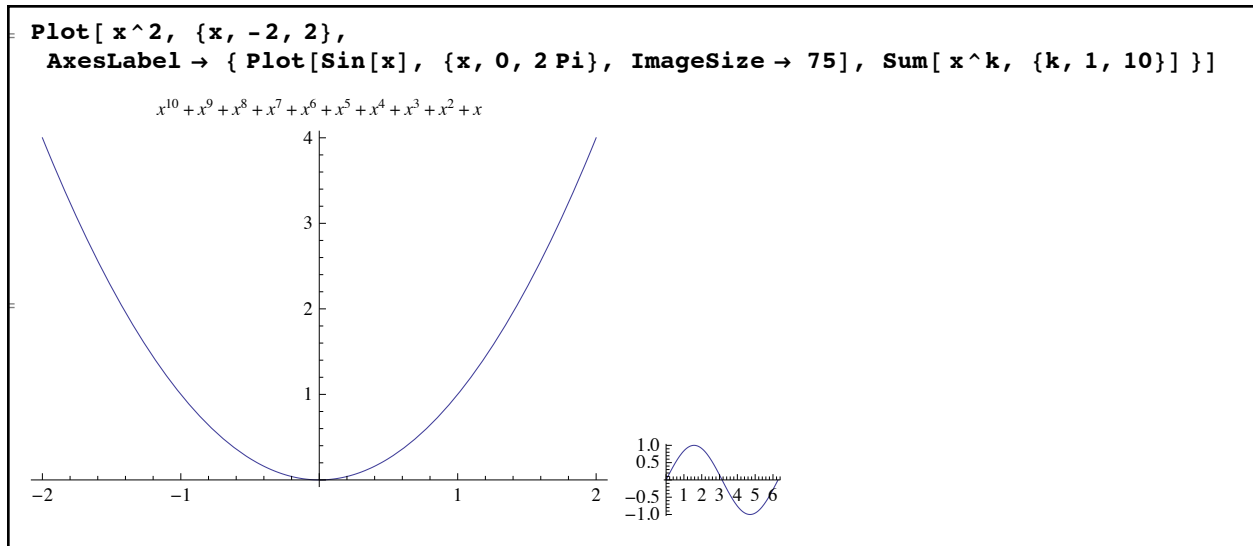
a cosine graph with axes suppressed using `Axes` and a frame with custom ticks added using `Frame` and `FrameTicks`



a graph of a cubic and line with no options - note that  $y=x$  does not have its usual  $45^\circ$  tilt

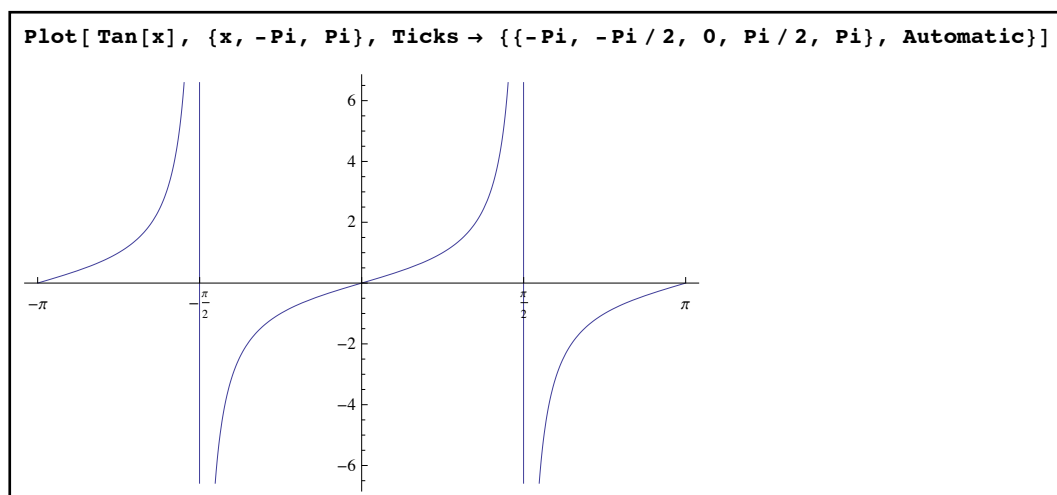


the same graph using *PlotLabel* and the same scales for both axes using *AspectRatio* -  $y=x$  now has the  $45^\circ$  tilt



*a graph whose axis labels have been set to a sum and to a small graph (whose size has been set to 75 pixels wide by ImageSize).*

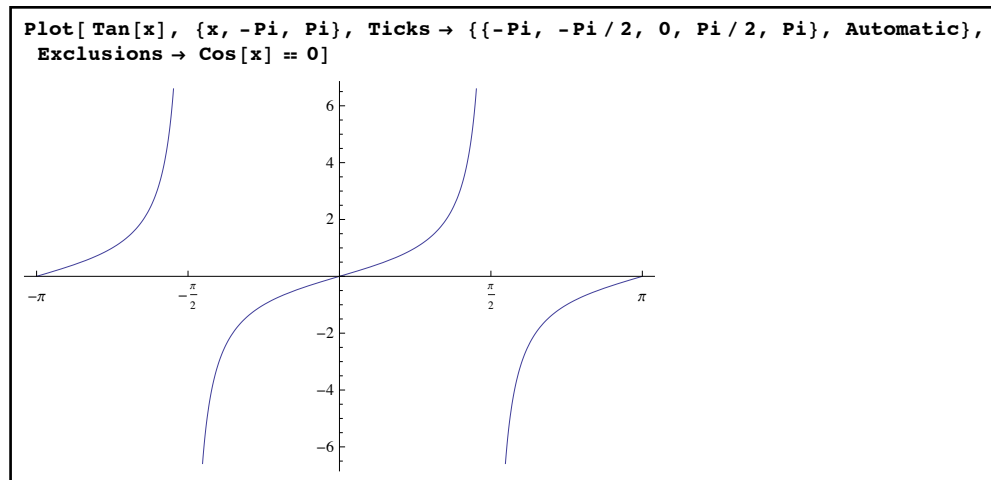
Another graphing option which can be useful for certain functions is Exclusions. Many graphs will have vertical asymptotes in your graphing range - a good example is the tangent function, which has vertical asymptotes at odd multiples of  $\pi/2$  (as  $\tan(x) = \frac{\sin(x)}{\cos(x)}$  and cosine is 0 at odd multiples of  $\pi/2$ ). If you graph  $\text{Tan}[x]$  from  $-\pi$  to  $\pi$  it will look like this:



*the graph of tangent*

Note the vertical lines that have been incorrectly placed at the asymptotes. The reason they appear is that at a base level Mathematica graphs just like a graphing calculator does - it plots points and then plays “connect the dots”. If you were to look at a value just to the left of  $\pi/2$  the value of tangent would be huge (say 1000) and a value just to the right of  $\pi/2$  would give you a deep negative number (like -2000). If you tried to connect these two values you would have a

graph so steep that for all practical purposes it would look like a vertical line - and that's exactly what you see in the graph. You can fix this problem by telling Mathematica to specifically not graph the values  $\pi/2$  and  $-\pi/2$  - that is what Exclusions is for. You can use the form either `Exclusions→{list of x-values to avoid}` or `Exclusions→equation whose solutions to avoid`. So in the tangent example we could specifically avoid the asymptotes by using `Exclusions→{-Pi/2, Pi/2}` or `Exclusions→Cos[x]==0` (remember to use the double equal sign for equations). The second version is the better one to use because you don't have to figure out the values to avoid yourself:



*the tangent graph with erroneous vertical lines removed by Exclusions*

These options together give you a good deal of control over the general appearance of the picture, but one thing they don't let you control is the appearance of the curve itself. This is one of the most important graph customizations - when you are graphing more than one function at a time controlling the curve appearance would give you an easy way to tell which curve is which. The option which controls curve appearance is `PlotStyle`, and the things like color, thickness, etc. that you set are called directives. A "simple" directive controls just one thing (like setting the color to be red and not setting anything else). A "compound" directive controls many things at one (like setting a curve to be both thick, blue, and dashed). You form a compound directive from simple ones by wrapping the command `Directive` around them - `Directive[ simple directive 1, simple directive 2, .... ]` represents a compound directive built up from several simple ones. If you have a single curve to set its directive simply use `PlotStyle→directive`; if you have several curves in your graph use the form `PlotStyle→{curve 1 directive, curve 2 directive, ....}` (if you only want to set specific directives for some of the curves you can use the directive `Automatic` for the others) .

Mathematica has a wide range of directives which allow extremely precise control of a curve's appearance. In this section we will only look at the basic ones you need to get started and leave the more advanced one to later sections:

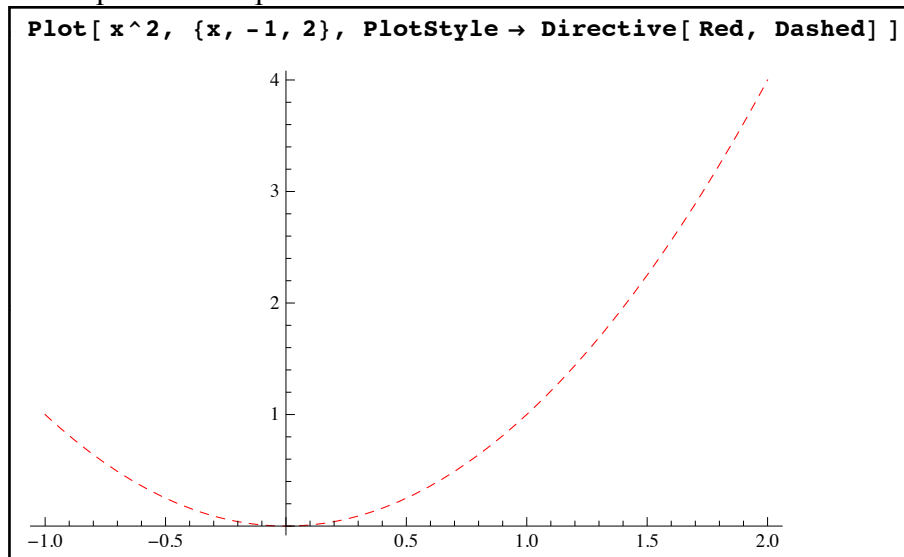


**Common colors:** You can set the color of a curve directly by using some common names, all of which start with uppercase letters (Red, Orange, Yellow, Blue, Purple, Green, etc...).

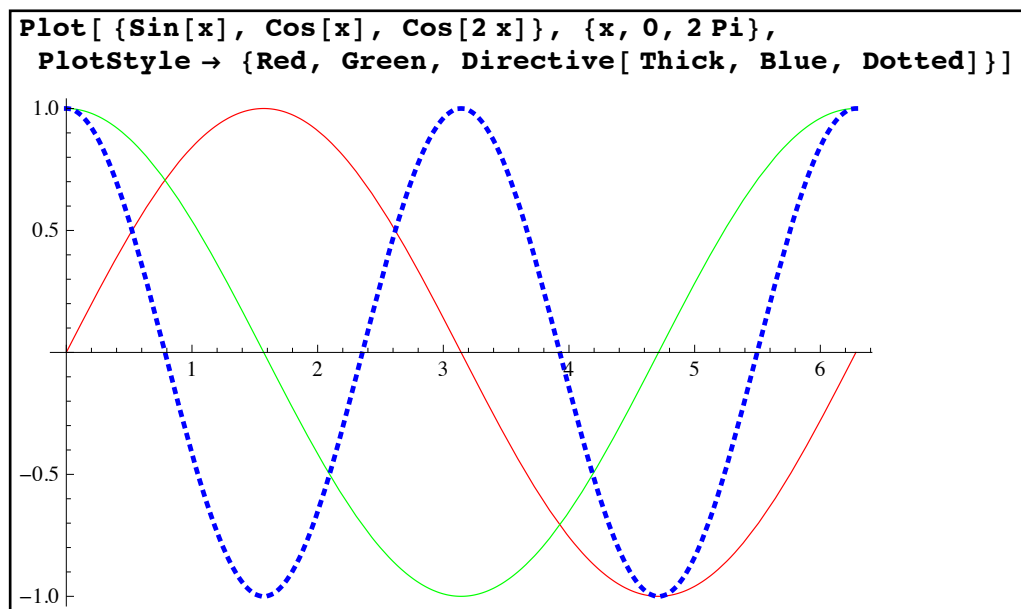
**Curve Thickness:** Thick and Thin are two common directives to quickly control the width of a curve.

**“Broken” Curves:** You can quickly control spacing within a curve by the self-explanatory directives Dashed, Dotted, and DotDashed.

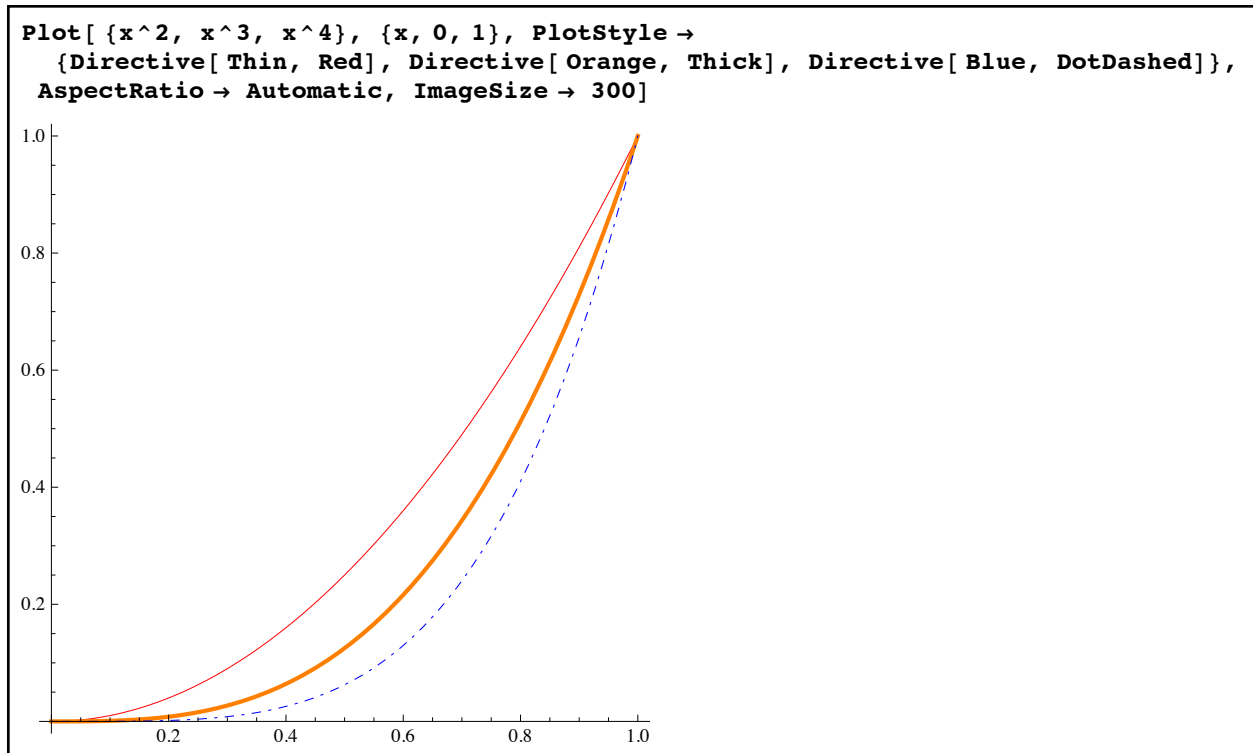
Using these directives in PlotStyle will make it easy for you to set up graphs with specific appearances and help tell them apart:



*using PlotStyle to customize the appearance of a single curve*

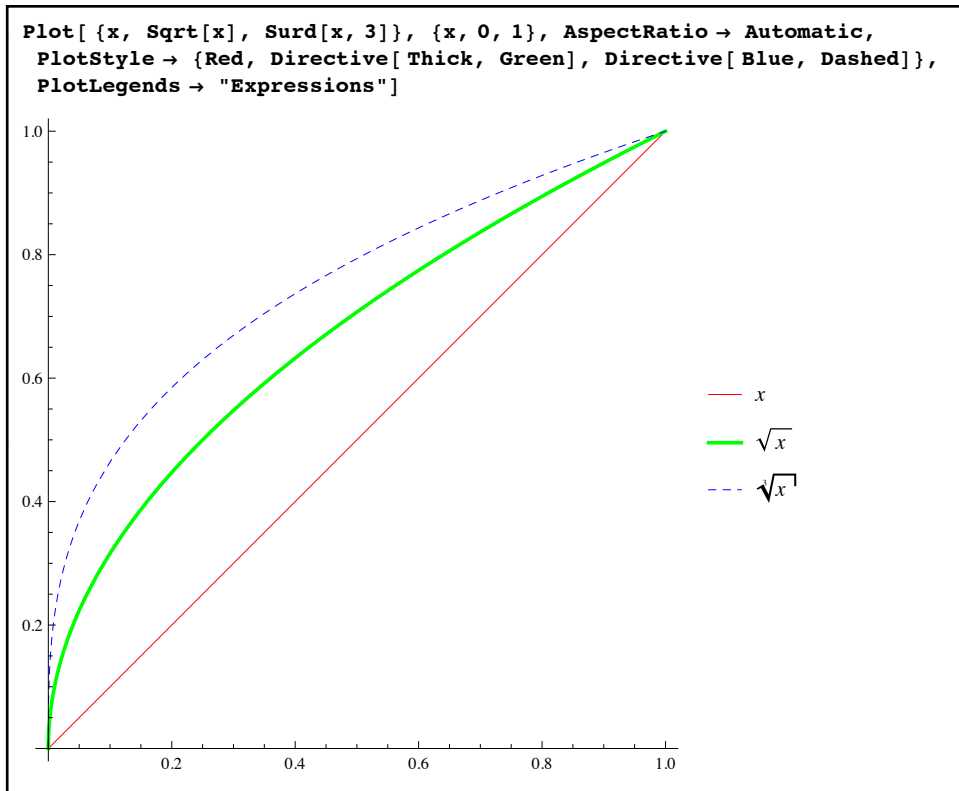


*customizing the appearance of 3 curves using PlotStyle and compound Directives*

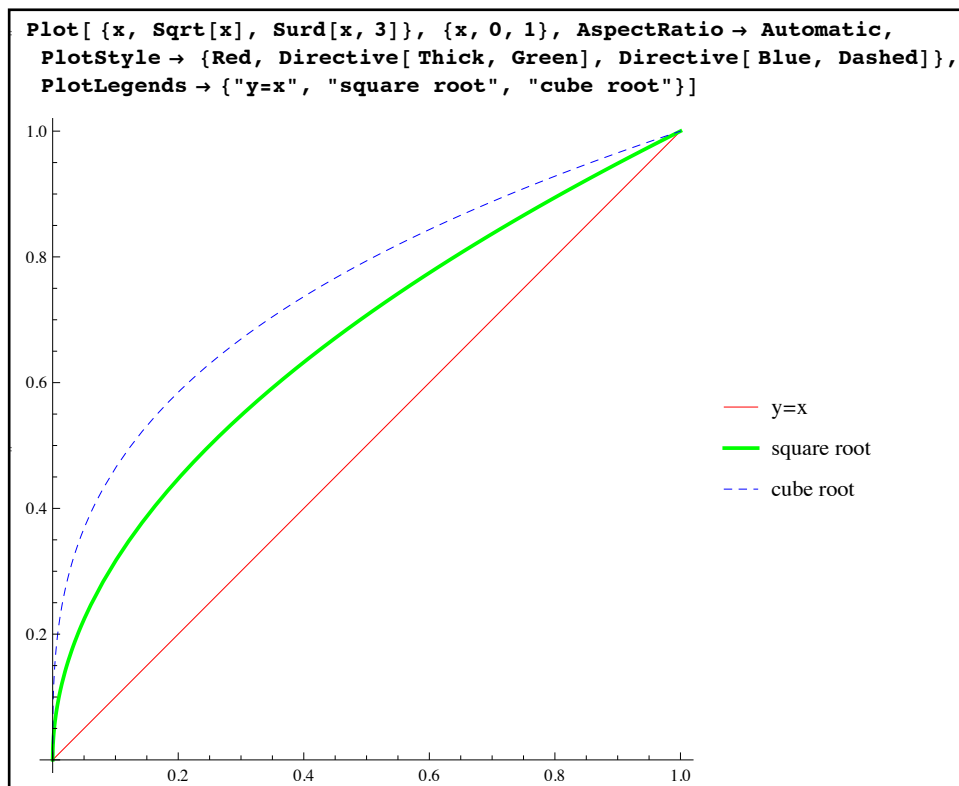


*using PlotStyle in conjunction with other options*

One of the great things about using PlotStyle is that it works well with a second option PlotLegends. The option PlotLegends will automatically create a legend for your graph that uses the same directives you set in PlotStyle (or the ones chosen by Mathematica if you didn't use PlotStyle). If you set PlotLegends→"Expressions" Mathematica will use the formulas as the legend labels; if you use PlotLegends→{label1, label2,...} you can use whatever labels you want (as with AxesLabel and PlotLabel you can use anything for the labels - but remember to include text in quotes). PlotLegends was introduced in Mathematica 9; earlier versions could accomplish something similar by loading in extra packages (which was much more cumbersome). Here are examples of both versions of PlotLegends:



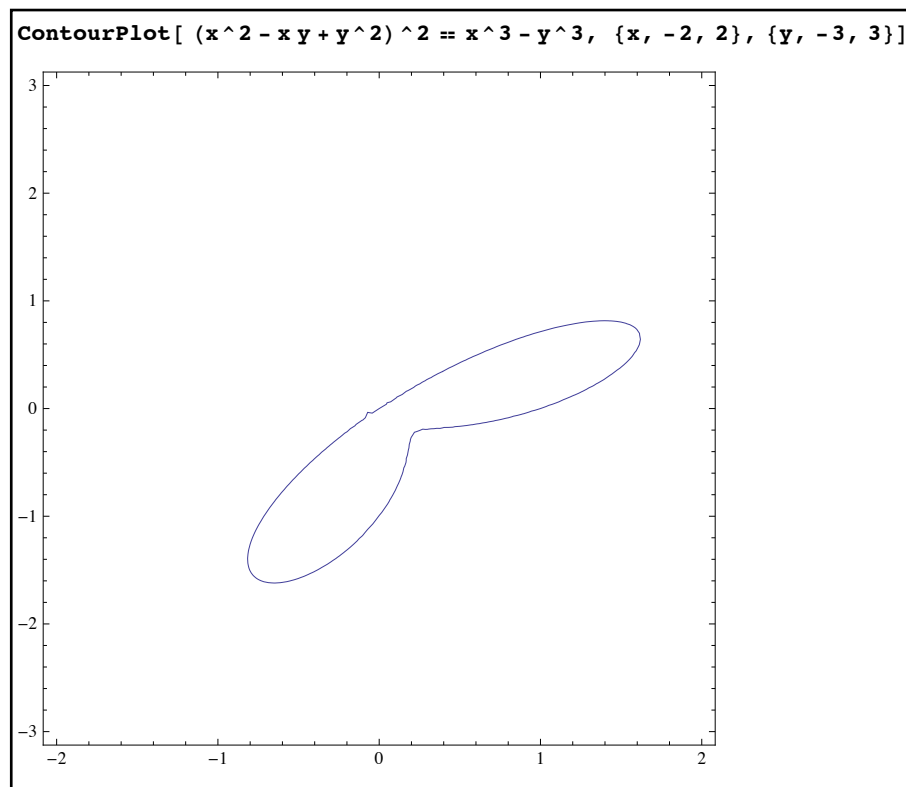
*using PlotLegends → “Expressions” to make a legend for your graph*



*using PlotLegends with custom text in the legend*

All we have done so far has been in the command `Plot`, which is centered around graphing functions. Sometimes you will need to graph an equation which is not easy to convert into a function form (or it might not even be possible) - if you need to graph  $x^3 + 3xy + y^3 = 1$  solving for either  $y$  or  $x$  is going to be painful and so `Plot` won't be of much use. The command `ContourPlot` will allow you to graph these kinds of equations - and you will be able to use many of the same options for `ContourPlot` as you did for `Plot`.

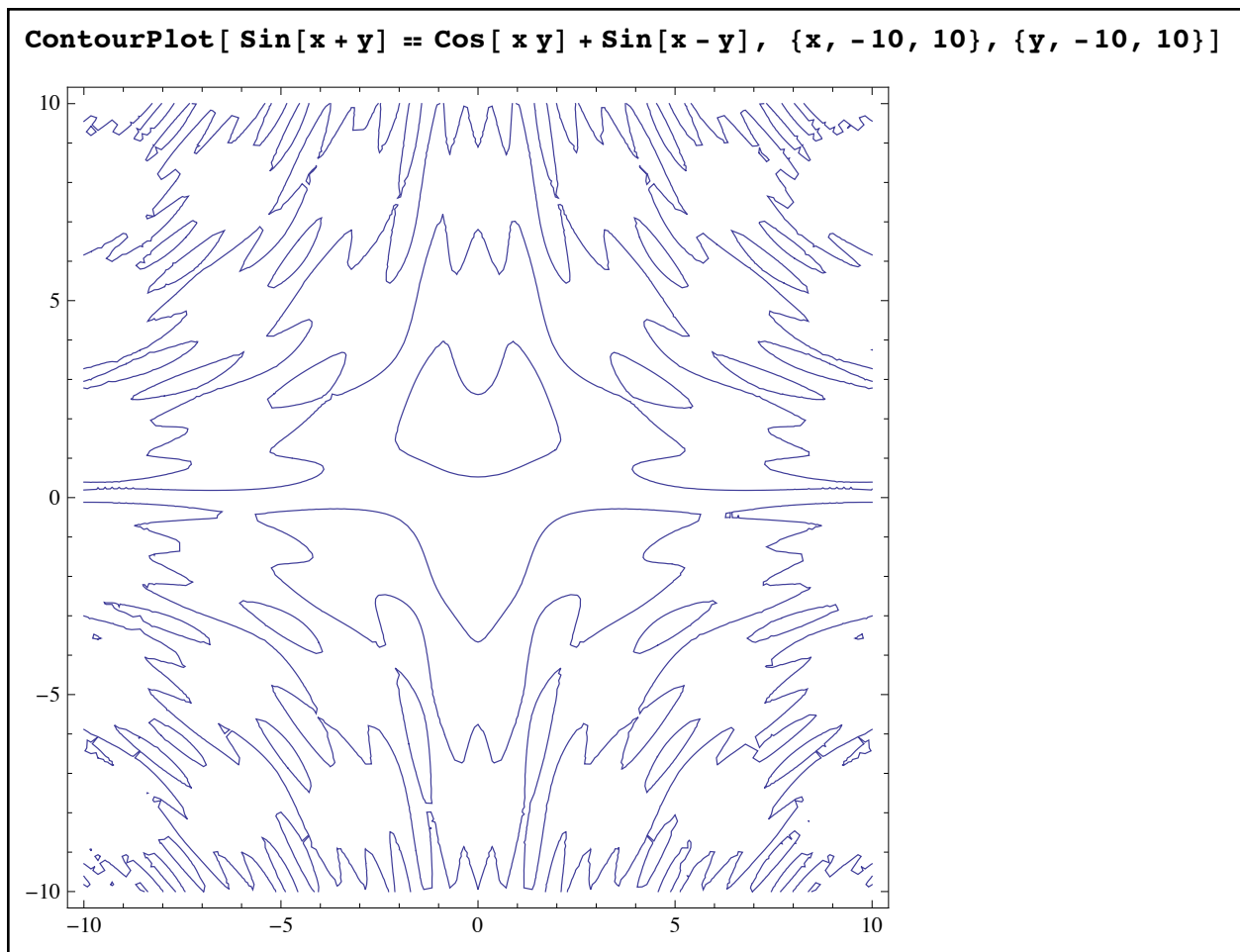
To use `ContourPlot` you will need to specify the range for both the “x” and “y” variables - when graphing one equation the basic format is `ContourPlot[ equation, {x, x-start, x-finish}, {y, y-start, y-finish}]` (any options you decide to use go after the y-range list). The equation must use the double equal sign `==` format. For example to graph the equation  $(x^2 - xy + y^2)^2 = x^3 - y^3$  as  $x$  goes from -2 to 2 and  $y$  goes from -3 to 3 use `ContourPlot[(x^2-x y+y^2)^2==x^3-y^3, {x,-2,2}, {y,-3,3}]` (don't forget the space between  $x$  and  $y$  to indicate multiplication):



*Using ContourPlot to graph an equation rather than a function*

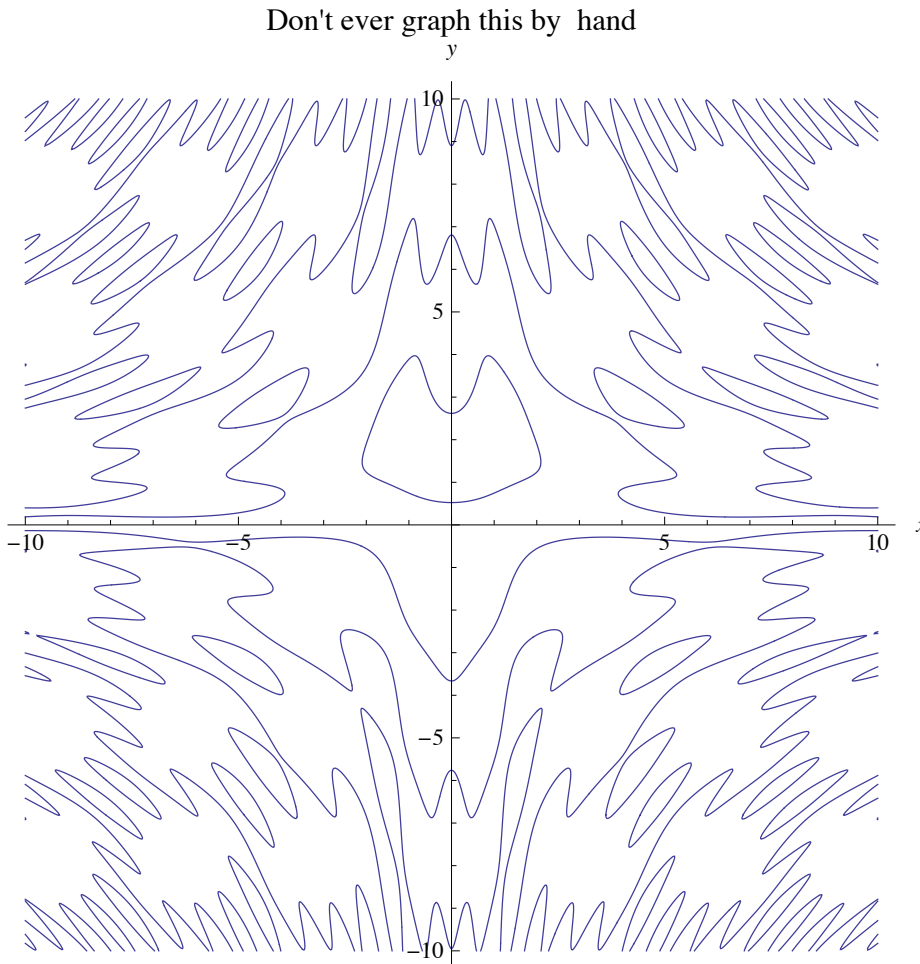
There are 3 things worth noticing about the graph right away. First is that there are no axes shown but the graph is framed. This is true by default for `ContourPlot` but can be overridden by the `Axes` and `Frame` options just as you would in the `Plot` command (you can also control tick marks the same way). Second is that the plot is square rather than rectangular by default. For `ContourPlot` the `AspectRatio` option is set to 1 by default (rather than `Automatic` or the value

1/GoldenRatio that Plot uses to make its standard rectangle) - you can set AspectRatio→Automatic if you want to use the same scales. The third thing to notice is the slight “glitch” at the coordinates (0,0). This indicates that more points need to be plotted (this often happens the graphs of equations). As with the Plot command you can remedy this with the option PlotPoints. Raising the value of PlotPoints in ContourPlot creates much more work for the computer than it does with Plot so you should be more conservative in using larger values - usually PlotPoints→50 or PlotPoints→100 should smooth out most wrinkles. You can see the difference in using various options (as well others taken from Plot) below:



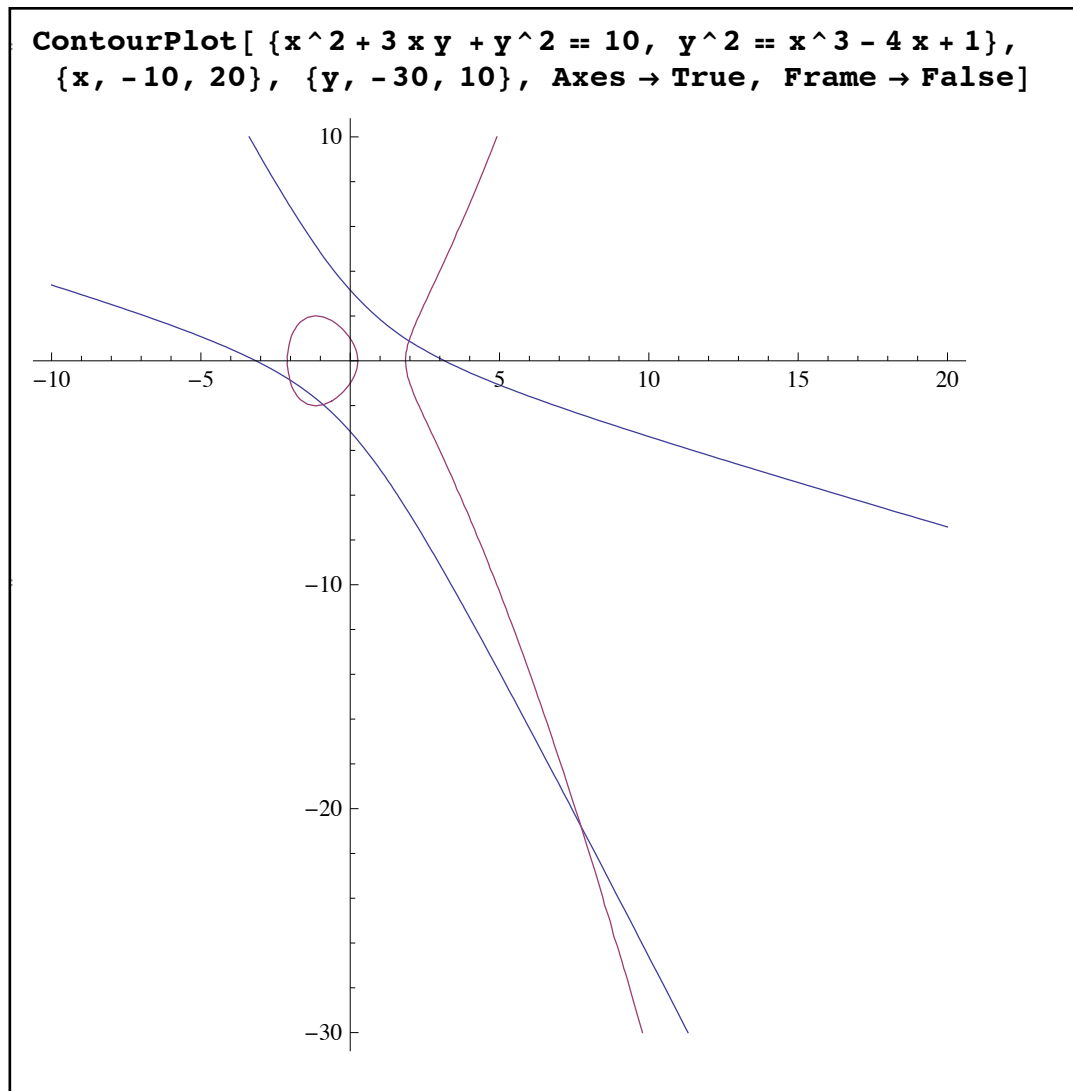
*the graph of a trigonometric equation - note the irregularities in the corners and where the graph almost forms loops*

```
ContourPlot[ Sin[x + y] == Cos[ x y] + Sin[x - y], {x, -10, 10},
{y, -10, 10}, PlotPoints -> 100, Axes -> True, Frame -> False,
PlotLabel -> "Don't ever graph this by hand", AxesLabel -> {x, y}]
```



*the same graph smoothed out by using PlotPoints→100 (which took more time) and given a more common look through the options Frame, Axes, AxesLabel, and PlotLabel*

You can also graph more than one equation at a time using ContourPlot - just replace the equation with a list of equations. This is really useful when you are looking at real solutions to systems of equations - the solutions correspond to where the curves cross. For example to graph the equations  $x^2 + 3xy + y^2 = 10$  and  $y^2 = x^3 - 3x + 1$  together try the command ContourPlot[  
 $\{x^2 + 3xy + y^2 == 10, y^2 == x^3 - 3x + 1\}, \{x, -10, 20\}, \{y, -30, 10\}, \text{Axes} \rightarrow \text{True}, \text{Frame} \rightarrow \text{False}\}$ :

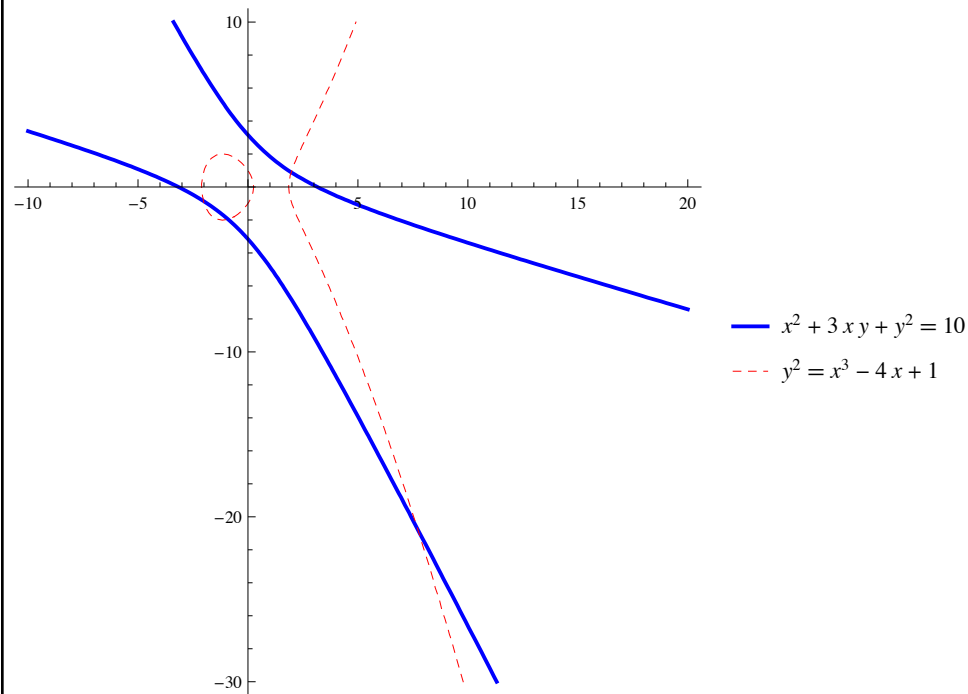


*graphing two complicated equations at the same time - it looks like there are 4 real solutions to the system as the curves cross 4 times in the given range*

In this example I had to play with the ranges for x and y to get a picture that looked like it had all the places where the curves crossed - but editing the x- and y-ranges in the command and re-evaluating it a very simple task.

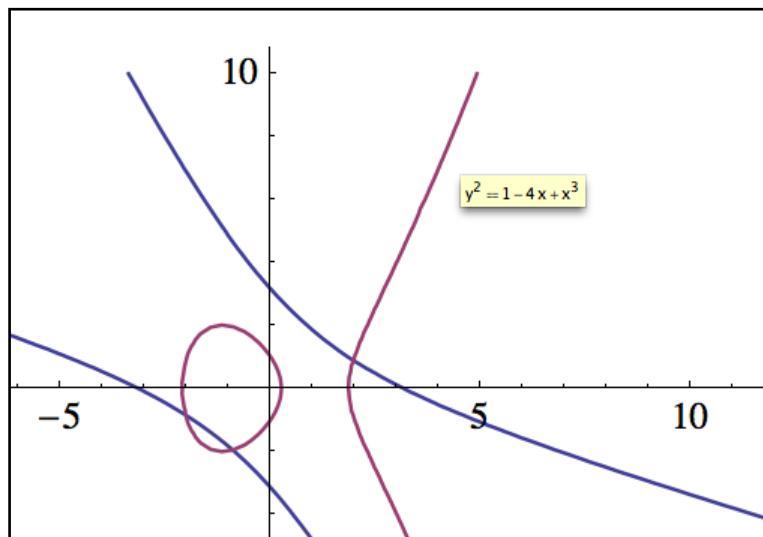
ContourPlot also inherits from Plot the notion of legends and controlling the appearance of curves. The only difference is that instead of using PlotStyle to control appearances you use ContourStyle - otherwise the formatting is exactly the same (including the use of Directive for compound directives). So in the example above I could make the first curve thick and blue and the second red and dashed by adding the option ContourStyle→{ Directive[ Blue, Thick], Directive[ Red, Dashed]} and get a legend by adding PlotLegends→"Expressions":

```
ContourPlot[{x^2 + 3 x y + y^2 == 10, y^2 == x^3 - 4 x + 1}, {x, -10, 20}, {y, -30, 10},
  Axes → True, Frame → False, ContourStyle → {Directive[Blue, Thick], Directive[Red, Dashed]},
  PlotLegends → "Expressions"]
```



*controlling the appearance of equation graphs and legends via ContourStyle and PlotLegends*

Even if you don't use PlotLegends in ContourPlot you can still tell which curve is which. If you hover the mouse pointer over the curve its equation will appear in a "floating tooltip":

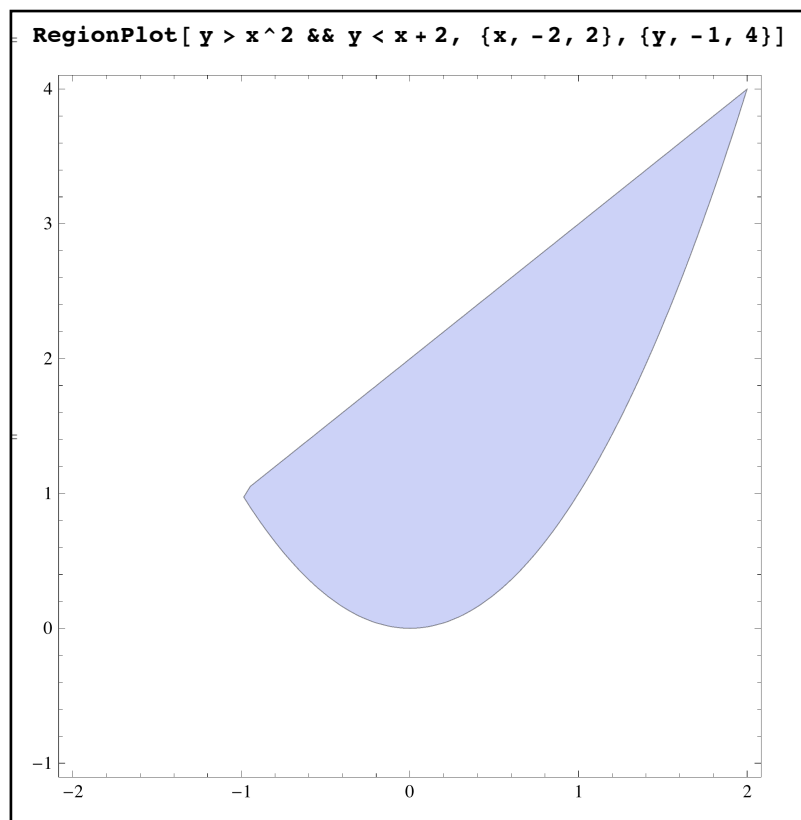


*a floating tooltip in ContourPlot (the cursor is hidden by the screenshot process)*



This floating tooltip can also be replicated in a Plot command by wrapping the command Tooltip around the list of functions to be plotted. The command `Plot[ Tooltip[ {Sin[x], Cos[x]}, {x,-10,10} ]]` would create a two-graph plot of sine and cosine where a floating tooltip would appear if you hover over one of the curves.

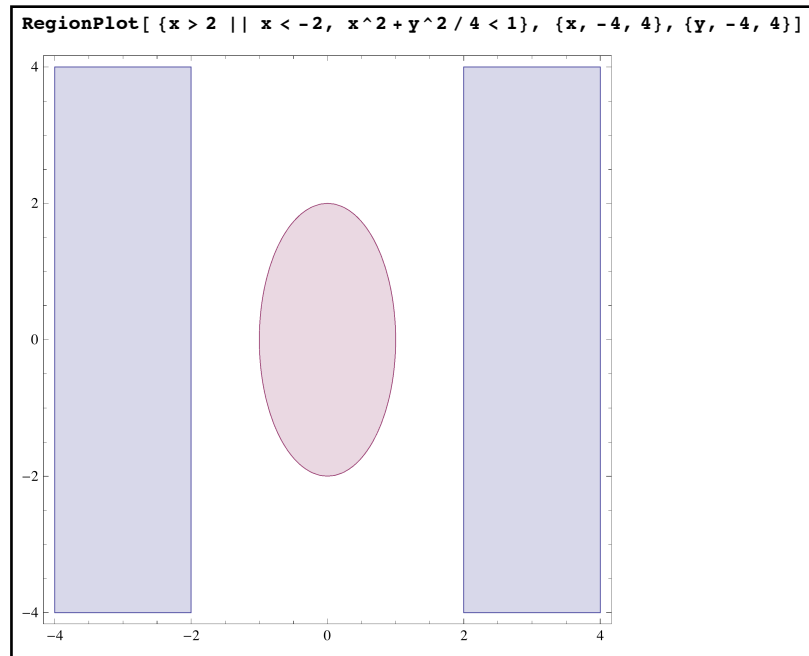
The last kind of graphing we will discuss for now is the graphing of regions. Most regions are defined by one or more inequalities; the region above the parabola  $y = x^2$  and below  $y = x + 2$  is really defined by the simultaneous inequalities  $y > x^2$  and  $y < x + 2$ . In Mathematica the word “and” is represented by the symbol `&&`, “not” is represented by `!`, and the word “or” is represented by `||` (this is the “inclusive or”, where at least 1 part must be satisfied, not exactly one). So the region above the parabola and below the line would be represented in Mathematica as  $y > x^2 \&\& y < x + 2$  (this is really more of a logical statement than an inequality). The command for graphing these kinds of regions/inequalities/logical statements is `RegionPlot` and it more or less works just like `ContourPlot` does. For a single region the format is `RegionPlot[ statement, {x, x-start,x-finish}, {y,y-start,y-finish}]`; for graphing more than one at a time the format is `RegionPlot[ list of statements, {x, x-start,x-finish}, {y,y-start,y-finish}]`:



*graphing a region with RegionPlot*

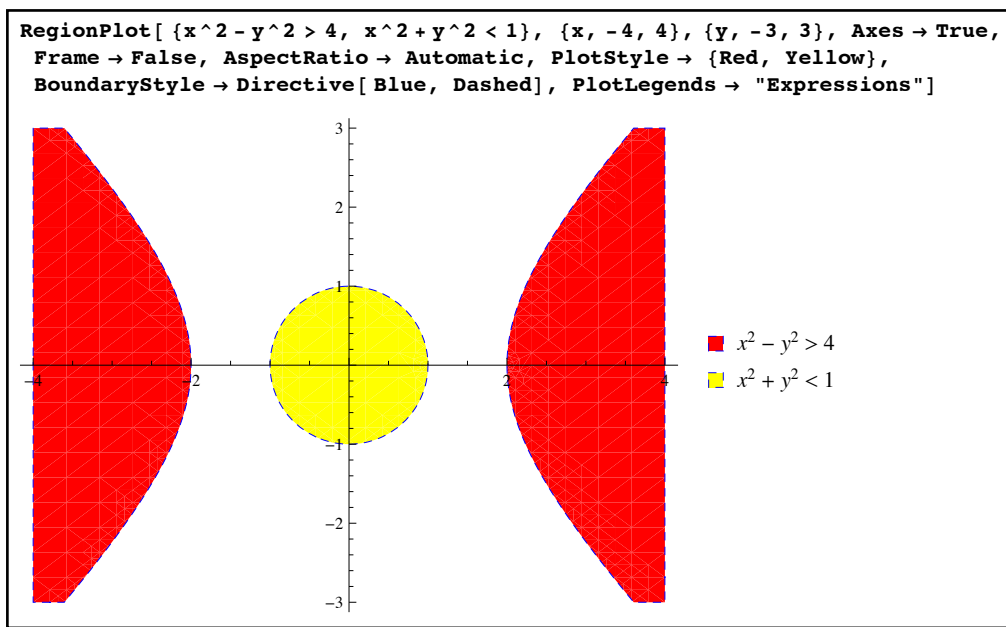
Note the notch in the left corner of the region? This is an irregularity caused by not graphing enough points and is quite common where regions have sharp corners. You can usually fix this

using the PlotPoints option (which works just as it does in ContourPlot so be cautious in how far you increase the value).



*graphing two regions (one of which has two disconnected parts to it, both blue)*

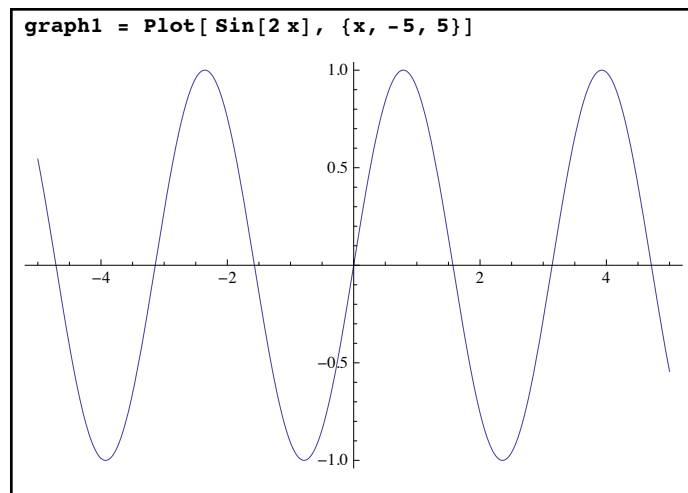
Note that like ContourPlot RegionPlot uses a square plot, frame, and no axes by default - as before you can control these with AspectRatio, Frame, and Axes. You can also control the appearance of the regions with the option PlotStyle (of the directives we have so far only the common colors apply to regions) and the appearance of the boundaries can be controlled with BoundaryStyle. You can also add a legend with PlotLegends:



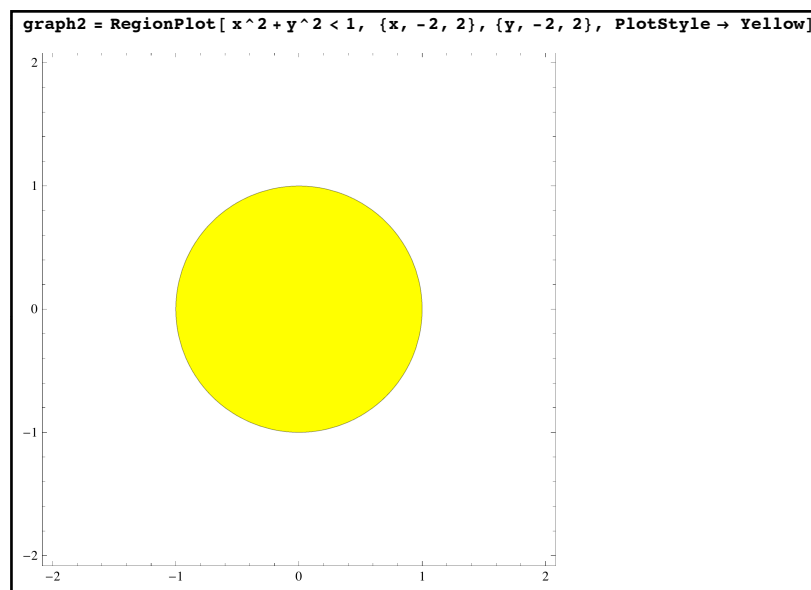
*a custom plot of two regions, complete with legend and boundaries*

You can also get floating tooltips for the regions by wrapping the Tooltip around the list of inequalities (the same way as we did for Plot).

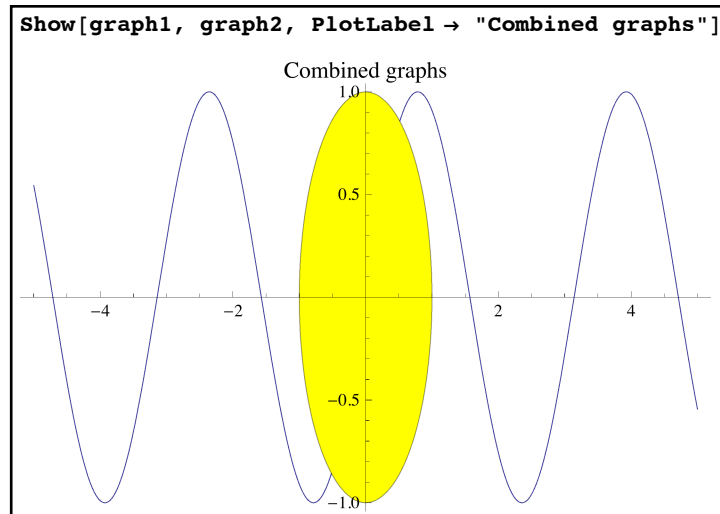
You can also combine the output of various Plot, ContourPlot, and RegionPlot commands into a single graph with the command Show. `Show[graphic1, graphic2, ..., new options]` will combine the given graphics (which are usually stored in variables like `graph1=Plot[.....]`, `graph2=RegionPlot[.....]`, etc.). The options from the first plot will be used for the combined plot unless they are reset by the “new options” at the end of the Show command (and these “new options” are limited to ones that don’t require the curves/regions to be replotted - so Axes, PlotLabel, and AspectRatio would work but not PlotStyle or PlotPoints). Consider the following graphs:



*a graph of sine using Plot, stored in the variable “graph1”*



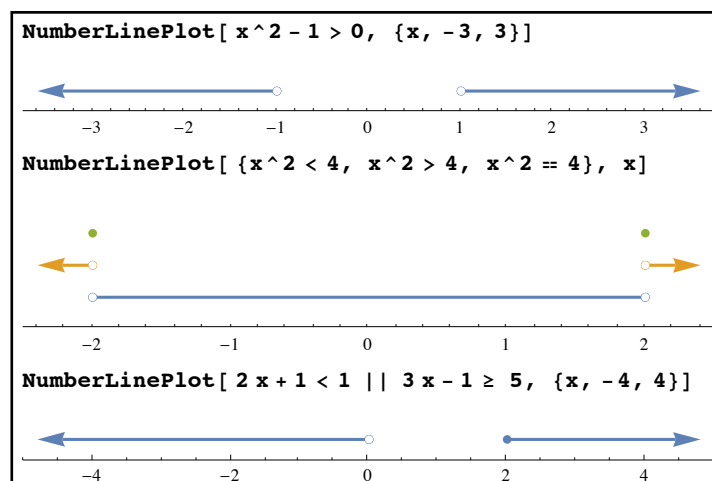
*the unit circle created by RegionPlot, stored in the variable “graph2”*



*the graphs combined using Show with a label added*

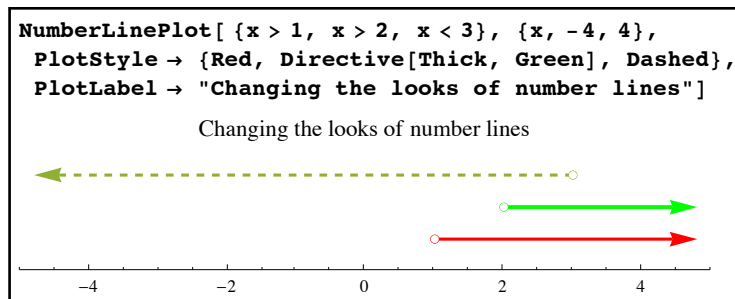
In the combined plot “graph1” came first so the values for AspectRatio, Axes, Frame, and PlotRange were inherited from “graph1” as they weren’t specifically set at the end of the Show command. In addition the unit circle region obscures the sine wave - this is because Mathematica creates the combined graphic as if each one were printed on a transparency. The sine wave in “graph1” is put down first and the region in “graph2” is laid down second/on top - so it obscures the graph underneath it where they overlap. So be careful when combining graphics with Show - the order in which the graphics are listed can make a big difference.

We wrap up the section with a new kind of graphing introduced in Mathematica 10 and which can be useful - graphing in only one dimension. What is a graph in one dimension? A number line - and these can be created by the new function NumberLinePlot. The format for NumberLinePlot is very similar to RegionPlot except there is only one variable involved. NumberLinePlot[ *logical statement or list of statements*, {*variable*, *start*, *finish*}] creates a graph (or graphs) of where the logical statements are True above a number line (you can replace the {*variable*, *start*, *finish*} with just *variable* if you want Mathematica to determine the range):

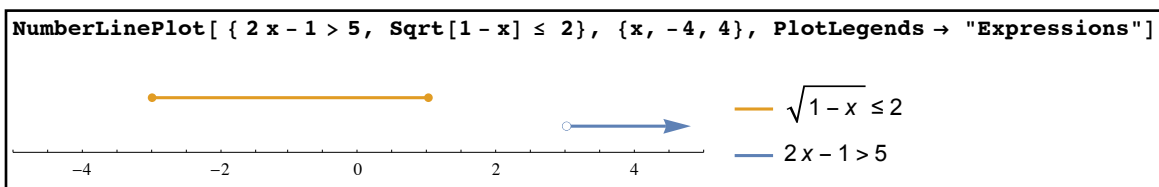


*graphing inequalities and equations in one variable with NumberLinePlot*

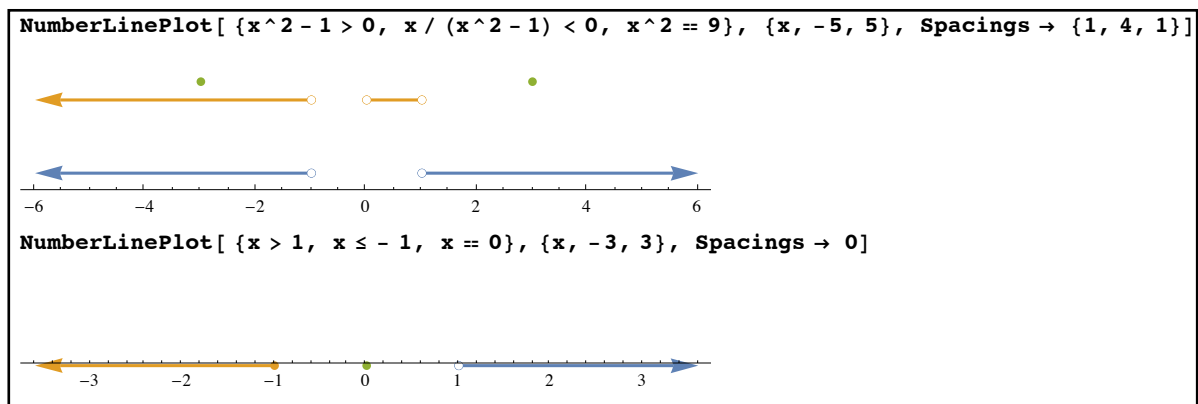
NumberLinePlot accepts the options PlotStyle, PlotLabel, and PlotLegends just as the other graphing functions do, which lets you customize the appearance of your lines:



*using options to control the appearance of the number lines*

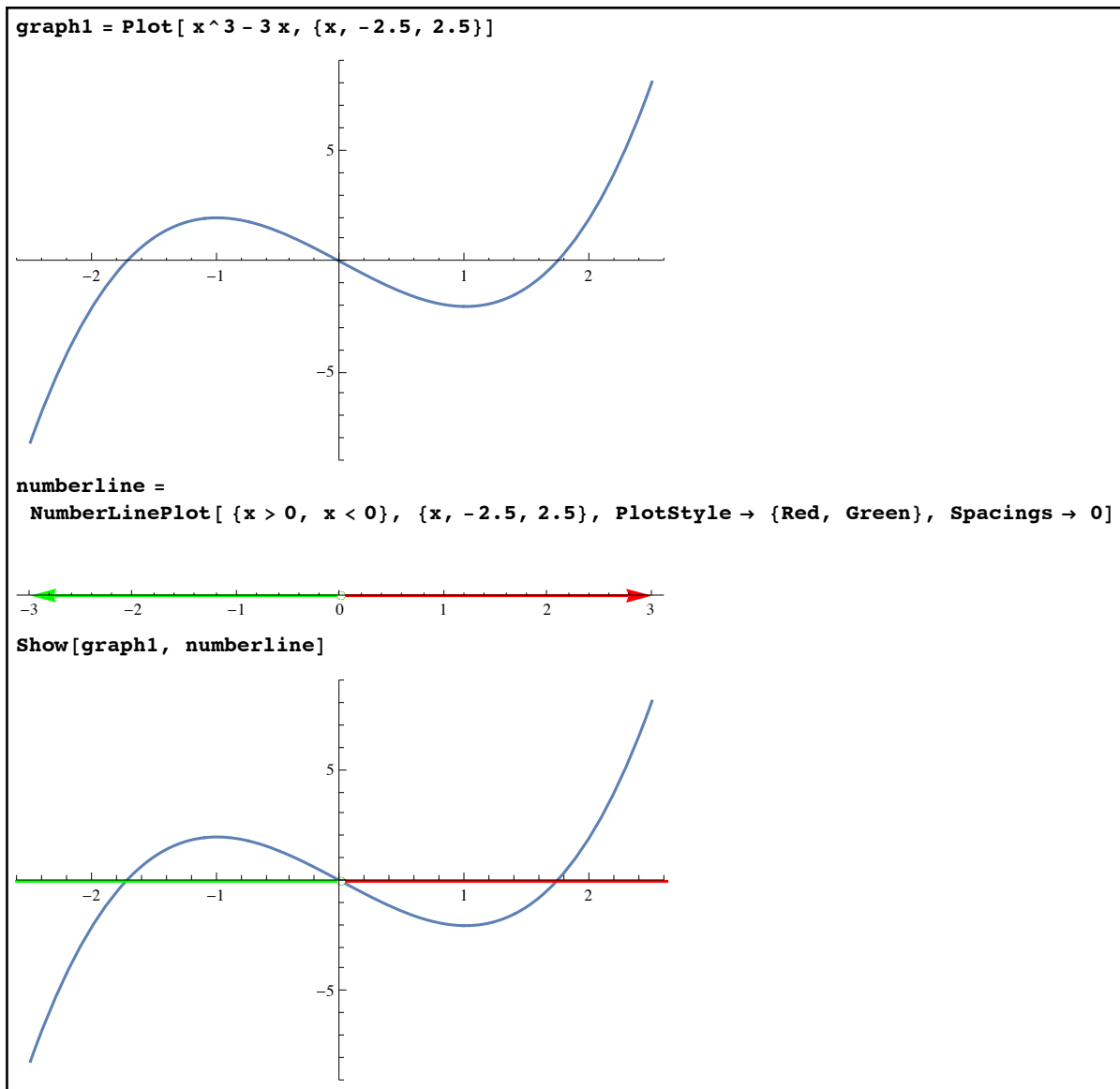


NumberLinePlot also has an option to control the heights of the lines in the graph - Spacings. Spacings→*list of numbers* controls the heights of the lines relative to the previous line (literally giving the spacing between the lines). The list {1,2,4} would mean the first line is 1 unit above the axis, the second line is 2 units above the first, and the third line is 4 units above the second. Spacings→*number* forces all of the lines to use the same spacing (this is most commonly used with Spacings→0, which forces all the lines onto the axis).



*using Spacing to control the space between lines*

Even though NumberLinePlot is really a one-dimensional graphing command the output is really a two-dimensional graphic; this has the advantage that we can use Show to combine a NumberLinePlot with other graphics. Here is an example of combining a standard graph together with a number line to indicate where the graph is bending upwards or downwards:



*combining a standard graph and number line*

In this problem it isn't hard to see which inequalities to use to illustrate the "bending up" or "bending down" ranges given that our original graph was fairly simple. Doing that in general requires you to be able to solve various inequalities (from Calculus) and we will cover how to solve inequalities in Section 2.6.

## Section 2.4 Homework – Basic Graphing in Mathematica

- 1) Graph  $\sqrt{x}$  and  $\ln(x)$  together as  $x$  goes from -1 to 10.
- 2) Plot  $\sin(x)$ ,  $\sin(2x)$ , and  $2\sin(x)$  together as  $x$  goes from  $-2\pi$  to  $2\pi$ . Include a legend on the side so you can tell which is which.
- 3) Graph  $y = x^2 - 3x + 1$  from -2 to 6. Show the  $y$  values from -10 to 20 in your graph.
- 4) Graph the lines  $y = 2x - 1$ ,  $y = x + 2$ , and  $y = -3x + 6$  on the same axes as  $x$  goes from -3 to 5. Use the same scales on the axes as well as a legend so you can tell which curve is which.
- 5) Graph  $y = \cos(x)$  from 0 to  $2\pi$ . Use tick marks every  $\pi/4$  on the  $x$ -axis and the values  $\pm 5$ ,  $\pm 1$  on the  $y$ -axis. Label the graph “the cosine function”
- 6) Repeat problem 6 but suppress the axes, and put the tick marks on a bounding frame instead.
- 7) Graph  $y = \frac{1}{x^2 - 4}$  from -5 to 5. Use Exclusions to remove any “division by zero” errors.
- 8) Repeat problem 7 but add the additional option `ExclusionsStyle → {{Red, Dashed}}`. What does this do to the graph?
- 9) Graph  $y = \sec(x)$  from  $-2\pi$  to  $2\pi$  and use Exclusions together with the `ExclusionsStyle` option from problem 8 to denote the vertical asymptotes.
- 10) Graph sine and cosine together from  $-2\pi$  to  $2\pi$  with the graph of sine blue and thick and the graph of cosine red and dashed. Include a legend with the labels “sine” and “cosine”.
- 11) Graph the real cube and fifth roots of  $x$  from -20 to 20. Label the axes and make the cube root graph black and dotted and the fifth root graph green.
- 12) Graph the equation  $xy - y^2 = 1$  as  $x$  and  $y$  go from -10 to 10.
- 13) Repeat problem 12 but remove the frame and include labelled axes.
- 14) Graph the equation  $\sin(x + y) = \cos(xy)$  as both variables go from -10 to 10. Make sure enough points are plotted to give you an accurate graph.
- 15) Graph the equations  $3x^2 + xy + y^2 = 10$  and  $x \sin(y) = 1$  together from -10 to 10 in both variables. Make the first graph red, the second blue, and include a legend. How many intersection points are there for the graphs? Store the graph in the variable `problem15` (as we will use it in the next problem)
- 16) Without creating a new `ContourPlot` restrict the viewing range of the graph from 15 to -5 to 5 in both variables, suppress the frame and show axes, and label the graph “2 equations”.
- 17) Use `ContourPlot` to graph  $x^2 + y^2 = 8$  from -3 to 3 in both directions. Copy the command into a new cell, delete the “=8” portion, and re-evaluate. What happens? Run your mouse over different parts of the graphs and see if you can determine what the tooltips indicate.
- 18) Graph the equations  $x^4 + xy + y^4 = 20$  and  $\sin(xy) + x + y = 1$  together as  $x$  and  $y$  go from -6 to 6. Make the first graph red and the second blue and thick. Suppress the frame, add labelled axes and a legend, and plot enough points to make sure the graphs are smooth.

- 19) Use RegionPlot to graph where the sine curve is greater than 0 and where it is less than 0 as  $x$  goes from 0 to  $2\pi$  and  $y$  goes from -1 to 1.
- 20) Graph the region inside the circle  $x^2 + y^2 + 2x - 4y = 25$  and above the line  $y = x + 1$ . Suppress the frame, provide labelled axes, and have the boundaries of the region graphed in black.
- 21) Graph the region defined by  $\sin(2x)\cos(3y) \geq \frac{1}{2}$ , where  $x$  and  $y$  go from 0 to  $2\pi$ . Have the region graphed in red with a boundary that is blue and dashed.
- 22) Let graph1 be the graph of the equation  $x^4 + xy + y^4 = 100$  and graph2 be the graph of the region  $(x - 2)^2 + y^2 \leq 4$  (both graphed as  $x$  and  $y$  go from -6 to 6). Explain the differences between Show[ graph1, graph2 ] and Show[ graph2, graph1 ] and why they occur.
- 23) Graph the inequality  $x^2 - 3x + 1 > 0$  on a number line. Graph a sufficient range to include all the behavior of the inequality.
- 24) Graph the inequalities  $x^2 - 4x < 0$ ,  $\cos(2x) > 0$ , and  $x^3 - 4x > 0$  together on a number line from 0 to  $2\pi$ .
- 25) Repeat problem 24 but use the spacings 0,1,5 and use a legend.
- 26) Repeat problem 24 but use the spacings -2, 4,5 and have the graphs be blue, green, and a dashed red. You will need to use the option PlotRange→All to see the results.



## Section 2.5 - Algebraic Computations and Manipulations

Mathematica can perform more or less all of the algebraic operations you learned back in your algebra and pre-calculus courses. Typically these operations fall into four categories - picking out a part of an expression or identifying a type (like finding the numerator of a fraction or saying something is an integer), performing specific operations (like bringing fractions over a common denominator), manipulating the form of an expression (like multiplying out a product), and solving equations and inequalities. The solution of equations and inequalities is such a central part of algebra that we'll dedicate an entire section to it - for the moment we'll focus on the first three kinds of algebraic operations.

The first class of operations is classifying the type of an object or picking out specific parts of it:

**IntegerQ[ *number* ]:** IntegerQ returns True if the *number* is an integer and False if it's not an integer. Mathematica will perform basic simplifications on the number before checking to see if it's an integer - so IntegerQ[5], IntegerQ[10/2], and IntegerQ[Sqrt[25]] all return True. The "Q" at the end of IntegerQ specifies a command whose result is either True or False (the T and F will always be capitalized) you can see the very long list of these functions by using the help command ?\*Q.

<b>IntegerQ[ 2031 ]</b> True <b>IntegerQ[ - 2 ]</b> True
---

*determining if numbers are integers with IntegerQ*

**EvenQ[ *number* ], OddQ[ *number* ]:** EvenQ returns True if the *number* is even and False otherwise. OddQ returns True if the *number* is odd and False otherwise.

**Element[ *object*, *set* ]:** Element returns True if the *object* is in the *set* and False otherwise. The *set* must be one of Mathematica's predefined sets - the ones you would use most often are Integers, Rationals, Reals, and Complexes. So Element[ 1/3, Integers ] is False, Element[ 5/4, Rationals ] is True, Element[ Pi, Rationals ] is False, Element[ Pi, Reals ] is True, and Element[ Pi, Complexes ] is True (as  $\pi$  is the same as the complex number  $\pi+0i$ ). Element[ *x*, Integers ] duplicates the functionality of the IntegerQ command. Element[ *list*, *set* ] returns true only if all the list elements are in the set.

```

Element[ 3 / 4, Rationals]
True

Element[ Sqrt[3], Rationals]
False

```

*checking if numbers are rational with Element*

**PolynomialQ[ *object*, *variable* ]**: PolynomialQ returns True if the *object* is a polynomial in *variable* and False otherwise. So PolynomialQ[  $x^2+1$ ,  $x$  ] is True but Polynomial[  $y^2 + y^{(1/2)}$ ,  $y$  ] is False. **PolynomialQ[ *object*, *list of variables* ]** returns True if the object is a polynomial in all the variables and False otherwise (so PolynomialQ[  $x^2+\text{Sqrt}[y]+x z-2$ , { $x,y,z$ } ] returns False as the expression is not a polynomial in  $y$ ).

```

PolynomialQ[ x^2 - 3 x + Sin[x], x]
False

PolynomialQ[ x^2 y - 10 x y + y^5 + 1, {x, y}]
True

```

**Numerator[ *fraction* ]**: Numerator gives the numerator of the *fraction* (whether it is a number or more complicated expression). Numerator will only give the expected result if the input is a single fraction; Mathematica will not combine fractions over a common denominator before applying Numerator.

**Denominator[ *fraction* ]**: Denominator gives the denominator of the *fraction*. It has the same “single fraction” requirement that Numerator does.

```

Numerator[ (x^2 + 3 x + 1) / (x + 2) ]
1 + 3 x + x^2

Denominator[ (x^2 + 3 x + 1) / (x + 2) ]
2 + x

```

*getting parts of fractions with Numerator and Denominator*

**Coefficient[ *expression*, *object* ]**: Coefficient finds the coefficient of the *object* in the *expression*. The expression does not have to be multiplied out beforehand nor does the object have to be a single variable. Coefficient[  $x^2+3x+1$ ,  $x$  ] returns 3, Coefficient[  $(x-1)^9$ ,  $x^2$  ] will return -36, Coefficient[  $(x+y+2)^2$ ,  $x y$  ] will return 2, Coefficient[  $(x+y+2)^2$ ,  $x$  ] will return  $4 + 2y$ , and Coefficient[  $\text{Sin}[x]^2+3 \text{Sin}[x] \text{Cos}[x]$ ,  $\text{Cos}[x]$  ] will return  $3\text{sin}(x)$ .

```

Coefficient[ 3 x^5 + x^3 - 4 x^4 + 2, x^4]
- 4
Coefficient[ (x^2 + y + 3 z + 1)^2, y]
2 + 2 x^2 + 6 z

```

*getting part of a polynomial with Coefficient*

**Exponent[ *expression*, *object* ]**: Exponent returns the highest exponent of *object* in the *expression*. Like Coefficient, Exponent does not require that either the expression be multiplied out beforehand or that the object be a variable. So Exponent[  $x^3 + 3x + 1, x$  ] will return 3, Exponent[  $(x^2 + 1)^{10}, x$  ] will return 20, and Exponent[  $\sin[x]^4 + \cos[x]\sin[x], \sin[x]$  ] will return 4. Exponent is used most commonly to find the degree of a polynomial but Mathematica couldn't use the word Degree for the command as that already stands for the trig conversion factor  $\pi/180$ .

```

Exponent[ (x^2 + y + 1)^300, x]
600
Exponent[ (x^2 + y + 1)^300, y]
300

```

*getting the degrees of polynomials using Exponent*

**FunctionDomain[ *function*, *variable* ]**: FunctionDomain finds the domain of *function* with the assumption that *variable* is the input variable. Typically the output is a logical statement like  $-2 < x < 2$  or  $x < 1 \mid x > 1$  (a function whose domain is all real numbers is considered to have to the domain “True”). A function which is defined everywhere will have a domain of True. FunctionDomain was introduced in Mathematica 10.

```

FunctionDomain[ Sqrt[9 - x^2] / (x^2 - 3), x]
- 3 ≤ x < -√3 || -√3 < x < √3 || √3 < x ≤ 3
FunctionDomain[ Tan[x], x]
1 x
- + - ∉ Integers
2 π

```

*finding the domain of functions*

**FunctionRange[ *function*, *input variable*, *output variable* ]**: FunctionRange finds the range of the given function, giving the answer as a logical statement in *output variable*. Like FunctionDomain a result of True means “all real numbers”. FunctionRange was introduced in Mathematica 10.

```
FunctionRange[ x^2 - 5 x + 1, x, y]

$$y \geq -\frac{21}{4}$$

FunctionRange[ x^3 / Exp[x], x, y]

$$y \leq \frac{27}{e^3}$$

```

*finding the ranges of functions*

FunctionPeriod[*function, input variable*]: FunctionPeriod tries to find the period of a function (the time to go through a full cycle). For functions where the domain is the integers or the complex numbers you can add a domain at the end as FunctionPeriod[*function, input variable, domain*] (where the domain is Integers or Complexes). FunctionPeriod was introduced in Mathematica 10.

```
FunctionPeriod[ Sin[x / 3] - Cos[x / 5], x]
30  $\pi$ 
FunctionPeriod[ x^2 - x^3, x]
0
```

*finding some periods - a result of 0 means the function is not periodic*

The second class of algebraic operations are those that perform a specific algebraic calculation, like multiplying things out or long division:

PolynomialQuotient[*polynomial1, polynomial2, variable*]: PolynomialQuotient finds the quotient when *polynomial2* is divided into *polynomial1* using *variable* as the division variable (when using polynomial division where the polynomials use more than 1 variable the choice of division variable can make a big difference).

```
PolynomialQuotient[ x^3 - 8, x^2 + x + 1, x]
-1 + x
```

*getting the quotient in polynomial division with PolynomialQuotient*

PolynomialRemainder[*polynomial1, polynomial2, variable*]: PolynomialRemainder finds the remainder when *polynomial2* is divided into *polynomial1* using *variable* as the division variable.

```
PolynomialRemainder[ x^3 - 8, x^2 + x + 1, x]
-7
```

*getting the remainder in polynomial division with PolynomialQuotient*

PolynomialQuotientRemainder[ *polynomial1*, *polynomial2*, *variable* ]:

PolynomialQuotientRemainder finds both the quotient and remainder when *polynomial1* is divided into *polynomial2* using *variable* as the division variable, returning the results as the list {quotient, remainder}.

```
PolynomialQuotientRemainder[ x^5 - 32, x - 2, x]
{16 + 8 x + 4 x^2 + 2 x^3 + x^4, 0}
```

getting a quotient and remainder (the 0 remainder indicates that  $x - 2$  is a factor of  $x^5 - 32$ )

Expand[ *expression* ]: Expand performs all multiplications and expands all the positive integer powers in *expression*. So Expand[ (x+1)(x+2) ] will give you  $2 + 3x + x^2$ , Expand[ (x-5)^2+(Sin[x]+3)^300] will give you a very long (but correct) result, and so on. As Expand works on positive powers it ignores denominators - multiplications there can be done using the related command ExpandAll.

```
Expand[ x^2 (x - 2) (x + 5)^3]
- 250 x^2 - 25 x^3 + 45 x^4 + 13 x^5 + x^6
Expand[ (x + y - 3)^3]
- 27 + 27 x - 9 x^2 + x^3 + 27 y - 18 x y + 3 x^2 y - 9 y^2 + 3 x y^2 + y^3
```

using Expand to multiply out expressions

Factor[ *expression* ]: Factor will attempt to factor *expression* as if it were a polynomial (it will not factor numbers - that is what FactorInteger is for). By default Factor will only try to factor the expression using rational numbers for coefficients. You can override this with option Extension. Extension→*number* will allow Mathematica to use the *number* when factoring, and Extension→*list of numbers* will let it use all the numbers in the list. So Factor[  $x^2+1$ , Extension→I] will result in  $(x - I)(x + I)$  and Factor[  $x^4-5x^2+6$ , Extension→{Sqrt[2], Sqrt[3]}] will let Mathematica use both square roots to get the factorization  $(x - \sqrt{2})(x + \sqrt{2})(x - \sqrt{3})(x + \sqrt{3})$ .

```
Factor[ x^8 - 1]
(- 1 + x) (1 + x) (1 + x^2) (1 + x^4)
Factor[ x^3 - 2]
- 2 + x^3
Factor[ x^3 - 2, Extension -> CubeRoot[2]]
- (2^(1/3) - x) (2^(2/3) + 2^(1/3) x + x^2)
```

factoring with the Factor command, both over the rationals and using the cube root of 2

**Cancel[ *expression* ]:** Cancel cancels common factors in the ratios of *expression*. If the expression is a sum or difference the “cancel” operation will be applied to each part individually.

$$\text{Cancel}[(x^3 - 8) / (x^2 - 4)]$$

$$\frac{4 + 2x + x^2}{2 + x}$$

*using Cancel to reduce a fraction without knowing the common factor*

**Discriminant[ *polynomial*, *variable* ]:** Discriminant computes the discriminant of *polynomial* under the assumption that *variable* is the variable and everything else is part of the coefficients. If you have never worked with discriminants they are numbers computed from the coefficients of a polynomial - for example the discriminant of the quadratic  $ax^2 + bx + c$  (using  $x$  is the variable) is  $b^2 - 4ac$ , which appears under the square root in the quadratic formula. The discriminant shows up in some advanced algebra problems, usually because of the fact that a polynomial has a repeated root/factor if and only if its discriminant is 0.

```
Discriminant[ a x^2 + b x + c, x]
b^2 - 4 a c
Discriminant[ x^5 - x^4 - 7 x^3 + x^2 + 7 x + 1, x]
2 057 216
Discriminant[ x^5 - x^4 - 7 x^3 + x^2 + 7 x - 3, x]
0
Factor[ x^5 - x^4 - 7 x^3 + x^2 + 7 x - 3]
(-3 + x) (-1 + x + x^2)^2
```

*some discriminant calculations - note the link between a zero discriminant and a repeated factor*

The third class of algebraic operations relate to converting one form of an expression to another:

**Together[ *expression* ]:** Together brings all of the terms in *expression* over a common denominator.

$$\text{Together}\left[\frac{1}{x-2} + \frac{5}{y-3} + \frac{x+1}{x^3-1}\right]$$

$$\frac{19 - 2x - 3x^2 - 13x^3 + 5x^4 - 3y - xy + x^2y + x^3y}{(-2+x)(-1+x^3)(-3+y)}$$

*using Together to combine fractions over a common denominator*

**Apart[*fraction*]**: Apart splits up the *fraction* into a sum and difference of simpler pieces if possible (simpler in this case meaning simpler denominators). If *fraction* is a rational function (polynomial over polynomial) in a single variable, the Apart command effectively does what is known in Calculus as “partial fraction decomposition”.

$$\text{Apart}[x^6 / (x^4 - 1)]$$

$$\frac{1}{4(-1+x)} + x^2 - \frac{1}{4(1+x)} + \frac{1}{2(1+x^2)}$$

*ripping a fraction into simpler pieces with Apart*

**Collect[*expression*, *variable*]**: Collect rewrites *expression* as a polynomial in *variable* by grouping like powers of *variable* together. If you try Collect[(x-2m+1)^3,x] the result will be  $x^3 + (3 - 6m)x^2 + (3 - 12m + 12m^2)x + (1 - 8m + 12m^2 - 8m^3)$  (which is written as a polynomial in x and m is thought of as a constant), and Collect[(x-2m+1)^3,m] results in the expression  $-8m^3 + (12 + 12x)m^2 + (-6 - 12x - 6x^2)m + (1 + 3x + 3x^2 + x^3)$  (in which m is treated as the variable and x is a constant).

**Collect[ (x + 3 z + 1) ^ 5, z]**

$$1 + 5x + 10x^2 + 10x^3 + 5x^4 + x^5 + (15 + 60x + 90x^2 + 60x^3 + 15x^4)z + (90 + 270x + 270x^2 + 90x^3)z^2 + (270 + 540x + 270x^2)z^3 + (405 + 405x)z^4 + 243z^5$$

*using Collect to group an expression in terms of powers of z*

**Simplify[*expression*]**: Simplify tries a range of algebraic and trigonometric transformations to find the simplest form of *expression* (Mathematica only has a limited time to try each transformation before moving on to the next).  
**Simplify[*expression*, *assumption*]** tries to find the simplest form for the *expression* using the extra information in the *assumption*. The *assumption* often takes the form of an inequality or the Element command (as in Element[x, Reals] to specify that x is a real number). You can chain one or more assumptions together using the && symbol as “and” (as in Simplify[Abs[x/y], x>0 && y<0]). You can also use the option Assumptions to list your assumptions, although it’s not really necessary in Simplify (Assumptions is an option used in many other functions).

```

Tan[ ArcSec[x] ]

$$\sqrt{1 - \frac{1}{x^2}} x$$

Simplify[%, x > 1]

$$\sqrt{-1 + x^2}$$

Simplify[ $\sqrt[4]{x^4}$ ]

$$(x^4)^{1/4}$$

Simplify[ $\sqrt[4]{x^4}$ , Assumptions → Element[x, Reals]]
Abs[x]

```

*making expressions simpler with Simplify, which sometimes requires extra assumptions*

FullSimplify[ *expression* ]: FullSimplify also tries to find the simplest form of *expression*, but uses a wider range of transformations and isn't constrained by a time limit for each one (which means FullSimplify can take a long time on complex expressions, so in general try Simplify first). FullSimplify[ *expression*, *assumption* ] uses the extra information in *assumption* to find the simplest form of *expression*.

```

Simplify[ x^3 - x /. x →  $\frac{1}{3} \left( \frac{81}{2} - \frac{3\sqrt{717}}{2} \right)^{1/3} + \frac{\left( \frac{1}{2} (27 + \sqrt{717}) \right)^{1/3}}{3^{2/3}} ]$ 

$$\frac{1}{18} \left( -3 \cdot 2^{2/3} (81 - 3\sqrt{717})^{1/3} - 3 \times 2^{2/3} \left( 3 (27 + \sqrt{717}) \right)^{1/3} + \left( (27 - \sqrt{717})^{1/3} + (27 + \sqrt{717})^{1/3} \right)^3 \right)$$

FullSimplify[ x^3 - x /. x →  $\frac{1}{3} \left( \frac{81}{2} - \frac{3\sqrt{717}}{2} \right)^{1/3} + \frac{\left( \frac{1}{2} (27 + \sqrt{717}) \right)^{1/3}}{3^{2/3}} ]$ 
3

```

*Simplify failed to help on a complicated substitution, but the extra power and allowed time in FullSimplify did the trick*

TrigExpand[ *expression* ]: TrigExpand uses trigonometric identities to break down trigonometric functions of a complicated angle (like 5x or 3x-3y) in terms of more basic ones (like x or y). The end results of TrigExpand often involve longer expressions that involve powers and products of trigonometric functions.



```

TrigExpand[ Sin[5 x]]
5 Cos[x]^4 Sin[x] - 10 Cos[x]^2 Sin[x]^3 + Sin[x]^5
TrigExpand[ Tan[x + y]]
      Cos[y] Sin[x]      Cos[x] Sin[y]
----- + -----
Cos[x] Cos[y] - Sin[x] Sin[y]  Cos[x] Cos[y] - Sin[x] Sin[y]

```

*TrigExpand breaks down complicated angles into basic ones*

**TrigReduce[ *expression* ]**: TrigReduce uses trigonometric identities to break down powers and products of trigonometric functions in *expression* into forms that involve no powers or multiplication. The end result of TrigReduce often involves trigonometric functions of more complicated angles (so TrigReduce is sort of TrigExpand in reverse).

```

TrigReduce[ Sin[x]^6]
1
-- (10 - 15 Cos[2 x] + 6 Cos[4 x] - Cos[6 x])
32
TrigReduce[ Cos[3 x] Sin[4 x]]
1
-- (Sin[x] + Sin[7 x])
2

```

*using TrigReduce to get rid of trigonometric powers and multiplications*

We end this section with four quick applications of these commands:

#### Application 1: Finding tangent lines to polynomial graphs

If  $y = p(x)$  is a polynomial then you can find the tangent line to the graph at  $x = a$  by finding the remainder when  $p(x)$  is divided by  $(x - a)^2$ ; if  $r(x)$  is the remainder, then  $y = r(x)$  will be the tangent line at that spot. As an example, here are the calculations to find the tangent lines to  $y = x^3 - 2x + 1$  at  $x = -1$  and  $x = 2$ :

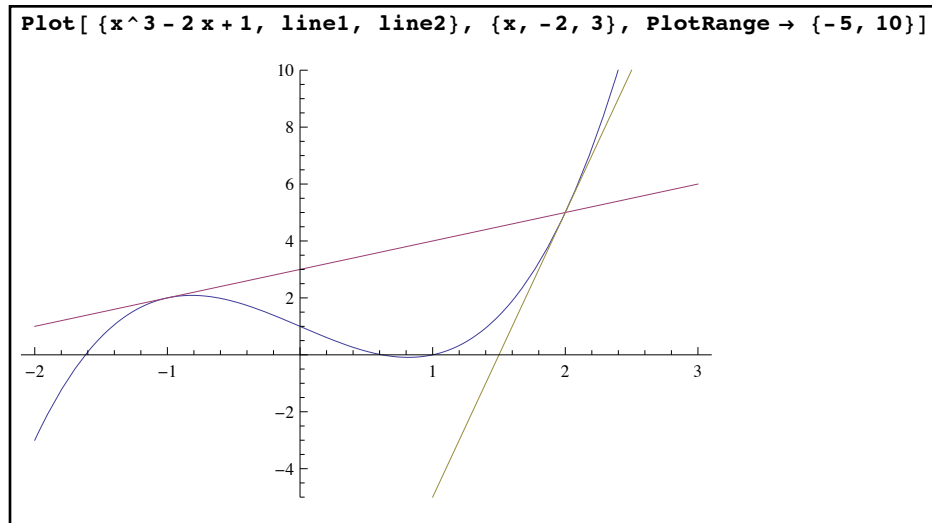
```

line1 = PolynomialRemainder[ x^3 - 2 x + 1, (x - -1)^2, x]
3 + x
line2 = PolynomialRemainder[ x^3 - 2 x + 1, (x - 2)^2, x]
-15 + 10 x

```

*finding the tangent line to a polynomial graph at two different points*

We can check this by graphing  $y = x^3 - 2x + 1$  together with the two lines and adjusting the PlotRange to get a good view of what is going on:



*the tangent lines to the graph at the x-values -1 and 2*

Application 2: Finding a relative maximum/minimum value for  $y = x^3 - 3x^2 - 9x - 6$ .

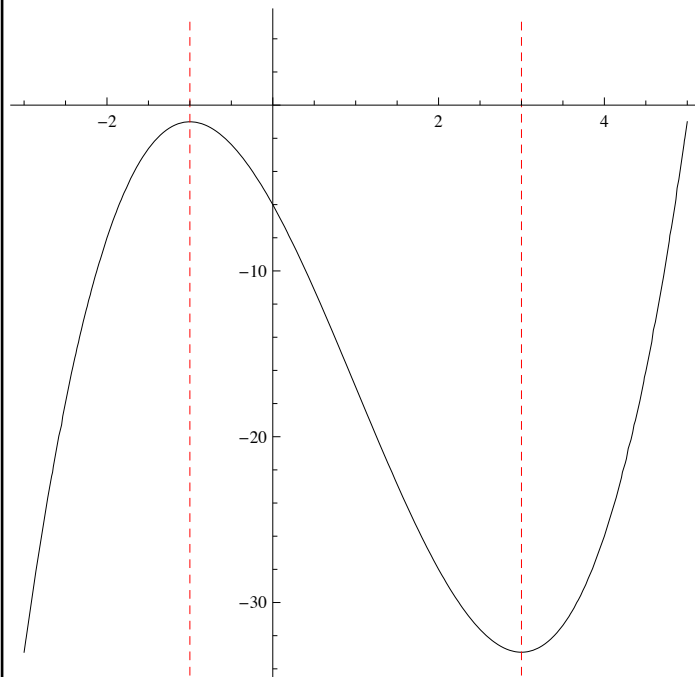
At a relative high point or low point on a graph the slope of the tangent line should be 0. From the first application we know we can find the tangent line to  $y = x^3 - 3x^2 - 9x - 6$  at any point  $x = a$  by dividing by  $(x - a)^2$  and taking the remainder. The slope of this remainder should therefore be 0, and we can get the slope as the coefficient of  $x$ . If we can see where this should be 0 we will have found where any relative high or low point will be:

```
PolynomialRemainder[ x^3 - 3 x^2 - 9 x - 6, (x - a)^2, x]
- 6 + 3 a^2 - 2 a^3 + (-9 - 6 a + 3 a^2) x
Coefficient[%, x]
- 9 - 6 a + 3 a^2
Factor[%]
3 (-3 + a) (1 + a)
x^3 - 3 x^2 - 9 x - 6 /. {{x -> -1}, {x -> 3}}
{-1, -33}
```

*finding high and low points on the graph*

From the calculation it looks like relative high and low points on the graph will be the points  $(-1, -1)$  and  $(3, -33)$ . We can verify this by graphing the curve and drawing vertical lines through the values  $x = -1$  and  $x = 3$  using ContourPlot:

```
ContourPlot[ {y == x^3 - 3 x^2 - 9 x - 6, x == 3, x == -1}, {x, -3, 5}, {y, -35, 5}, Axes -> True,
  Frame -> False, ContourStyle -> {Black, Directive[Red, Dashed], Directive[Red, Dashed]}]
```



*we've found the relative high and low points on the graph*

### Application 3: Computing a tangent slope formula

In Calculus you find a slope formula for a function  $f(x)$  (called the derivative) by finding/simplifying the “difference quotient”  $\frac{f(x+h) - f(x)}{h}$  and letting  $h$  “go to 0”. For simple functions this is often done by canceling a common factor of  $h$  in the difference quotient and then replacing  $h$  with 0 in the result. Let's do this for  $f(x) = x^{10} - 4x^9$ :

```
original = x^10 - 4 x^9
- 4 x^9 + x^10
diffquot = ( (original /. x -> x + h) - original) / h
4 x^9 - x^10 - 4 (h + x)^9 + (h + x)^10
h
Cancel[diffquot]
- 4 h^8 + h^9 - 36 h^7 x + 10 h^8 x - 144 h^6 x^2 + 45 h^7 x^2 - 336 h^5 x^3 + 120 h^6 x^3 - 504 h^4 x^4 + 210 h^5 x^4 -
504 h^3 x^5 + 252 h^4 x^5 - 336 h^2 x^6 + 210 h^3 x^6 - 144 h x^7 + 120 h^2 x^7 - 36 x^8 + 45 h x^8 + 10 x^9
% /. h -> 0
- 36 x^8 + 10 x^9
```

*finding a slope formula*

This would take you a long time to do by hand as expanding out  $(x + h)^{10}$  and  $(x + h)^9$  would take a while. The difference quotient comes up enough in practice that Mathematica 10.4 added a command for it in the form `DifferenceQuotient[ original formula in x, {x, h}]`.

#### Application 4: Finding a cosine quintuple-angle formula

A theorem from trigonometry says that if  $n$  is a natural number there is a formula for  $\cos(nx)$  that involves only powers of  $\cos(x)$ . Let's find this for  $\cos(5x)$ . We can use `TrigExpand` to break this down into basic sines and cosines:

```
TrigExpand[ Cos[5 x] ]
Cos[x]^5 - 10 Cos[x]^3 Sin[x]^2 + 5 Cos[x] Sin[x]^4
```

*expanding out  $\cos(5x)$  into basic functions*

This is a step in the right direction but still has sines in it. We can use the Pythagorean identity  $\sin^2(x) + \cos^2(x) = 1$  to rewrite the sines in terms of cosines and then multiply it all out:

```
% /. {Sin[x]^2 -> 1 - Cos[x]^2, Sin[x]^4 -> (1 - Cos[x]^2)^2}
Cos[x]^5 - 10 Cos[x]^3 (1 - Cos[x]^2) + 5 Cos[x] (1 - Cos[x]^2)^2
Expand[%]
5 Cos[x] - 20 Cos[x]^3 + 16 Cos[x]^5
Simplify[Cos[5 x] == %]
True
```

*$\cos(5x)$  as a formula involving only  $\cos(x)$*

As a check on our work the `Simplify` command reduces the equation (note the double = sign) to `True` - which means our formula always works.

## Section 2.5 Homework – Algebraic Computations and Manipulations

- 1) Find 4 Mathematica commands that end in Q other than those defined in this section and explain what they do.
- 2) Find the coefficients of  $y^2$  and  $y^5$  in  $(3y + 2)^8$ .
- 3) Find the coefficient of  $x^3$  in  $(4x^2 - 3xy + 5)^4$ .
- 4) We did not define the `CoefficientList` command in this section but look up its format and use it to find the coefficient of every term in  $(2r - 1)^6$ .
- 5) What is the degree of the polynomial  $(3x^3 + 2y + 1)^{11}$  with respect to  $x$ ? With respect to  $y$ ?

- 6) Use PolynomialQ to determine if the expression  $\frac{(x + y + 1)^3}{(y^2 + 1)^2}$  is a polynomial in  $x$ , a polynomial in  $y$ , and a polynomial in both  $x$  and  $y$ .
- 7) The list of domains for Element given in this section isn't complete - look up the additional domains in the documentation as well as the shorthand for the "element of" symbol.
- 8) Explain why Numerator[  $1/x + 1/(1+x)$  ] returns the expression  $1/x + 1/(1+x)$ . What additional command is needed to get the true numerator?
- 9) Find the numerator and denominator of the result of the addition  $\frac{x}{x-1} + \frac{2x+3}{x^2+x}$ .
- 10) Find the domain and range of the function  $\frac{\sqrt{x+4}}{x^2-1}$ .
- 11) Find the domain, range, and period of the function  $\sin(\frac{x}{2}) + 2\cos(x)$ . The range will look fairly complicated - use FullSimplify on it.
- 12) How does Mathematica report the period of the function  $x^2$ ? What does this really mean?
- 13) In this section we only used FunctionDomain and FunctionRange on functions of one variable. If  $z$  is the function  $\frac{1}{x^2 + y^2 - 1}$  find its domain and range by using the input variable list {x,y} and the output variable  $z$ .
- 14) Find the quotient and remainder when  $x^2 + 1$  is divided into  $3x^4 - 2x + 1$ .
- 15) Find the quotient and remainder when  $(x - a)^2$  is divided into  $x^4 - 2x^2 + x + 1$ .
- 16) Find the quotients and remainders when  $2x + y - 2$  is divided into  $x^3 + 2xy^2 + 3y - 1$  - once using  $x$  as the division variable and again when  $y$  is the division variable.
- 17) Multiply out the expression  $(3x + 1)^3 + (2x - 1)(x^2 + x + 1)$ .
- 18) Multiply out the expression  $(x + y + 1)^3 - (x + y)^3$ .
- 19) Factor  $x^{20} - 1$  over the rationals.
- 20) Factor  $x^6 + 3x^4 - 6x^3 + 3x^2 + 18x + 10$  over the rationals. Factor it using the complex number  $i$ . Factor  $x^6 + 3x^4 - 6x^3 + 3x^2 + 18x + 10$  again using the number  $\sqrt[3]{3}$ . Factor it using both  $i$  and  $\sqrt[3]{3}$ .
- 21) Write the fraction  $\frac{1}{\frac{x^4 - x}{x^5}}$  as a sum or difference of simpler fractions.
- 22) Write the fraction  $\frac{1}{x^2 - x - 6}$  as a sum or difference of simpler terms.
- 23) Cancel common factors in  $\frac{x^6 - 1}{x^8 - 1}$ .
- 24) Determine if the polynomials  $x^3 - 5x^2 + 3x + 9$  and  $x^3 + x^2 - 3x + 1$  have repeated roots.
- 25) Write the expression  $(x + 2y + 3)^4$  as a polynomial in  $x$ . Then as a polynomial in  $y$ .
- 26) Simplify the expression  $\sqrt{x^2 + 2x + 1}$  using the assumption that  $x$  is real. Then again using the assumption that  $x < 1$ .

- 27) Simplify the expression  $\sin(\sec^{-1}(x))$  given that  $x > 1$  and again given that  $x < -1$ .
- 28) Write  $\cos(3x)\cos(5x)$  as an expression that does not involve multiplication of trigonometric functions.
- 29) Write  $\sin^5(x)$  as a sum or difference of other trigonometric functions.
- 30) Write  $\sin(4x)$  as a combination of trigonometric functions involving only the base angle  $x$ .
- 31) Write  $\cos(6x)$  in terms of  $\cos(x)$ . (see example 4)
- 32) Verify that both Simplify and FullSimplify don't help with the expression  $\sin(3 \tan^{-1}(x))$  directly but TrigExpand does.
- 33) Use the method of example 1 to find the tangent lines to  $y = x^3 - 2x^2$  at  $x = -1$  and  $x = 1$ . Graph these lines together with the curve.
- 34) Use the method of example 3 to find the tangent slope formula for  $y = x^2 + \frac{1}{x^2}$ .

## Section 2.6 - Solving Equations and Inequalities

Some of the core applications of algebra to other areas lie in the ability to solve equations (either a single equation or a system of equations) and inequalities. Mathematica has several commands meant to solve equations and inequalities, each tailored to specific situations. Before we go through these in some detail it's worthwhile to frame some questions you might want to ask about an equation or inequality that might guide your choice about which command to use:

- 1) Do I need specific solutions or is it good enough to have a result which describes the solution in a simpler form than what I started with? For example if you want to solve  $2x - 1 = 5$  you most likely want the specific solution  $x = 3$ . However if you want to solve the inequality  $5 < 2x - 1 < 7$  you probably are thinking of the solution as  $3 < x < 4$ , which doesn't give you a specific value for  $x$  but does describe the solution in a much simpler fashion than the original equality. A description of the solution is more common when working with inequalities or equations with an infinite number of solutions (like  $\sin(x) = \frac{1}{2}$ ).
- 2) What kind of answers am I looking for? In algebra complex solutions are fine but for graphing and geometry problems usually only real numbers make sense. There are also some problems where only integer answers fit the requirements of the problem (if  $x$  is the number of people hired fractions and decimals wouldn't make sense).
- 3) Do I need exact answers or are approximate solutions good enough? If your equations use approximate numbers (i.e with decimal points) Mathematica's solutions will be approximate as well. So there is a big difference between the equations  $.2x + .5y = 7$  and  $2x + 5y = 70$  - using the former will always result in an approximate solution but the latter can be used in exact solutions.
- 4) Are my equations and inequalities algebraic or transcendental? Algebraic equations and inequalities use only the operations of arithmetic and the extraction of roots; transcendental ones are by definition "not algebraic" and can involve other kinds of functions like logarithms, sines, and so on. So  $x^7 + 3x^5 - 5x + 12x^{3/4} = 0$  is an algebraic equation (the fractional power represents a root) but  $x^2 = \sin(3x) + 2$  is a transcendental equation. Transcendental equations are typically much harder to solve exactly than algebraic ones (if not impossible); in all but the simplest cases you will probably need to find approximate solutions if transcendental equations are involved. The notion of algebraic and transcendental can be relative in some cases - the equation  $\sin^2(x) - 3\sin(x) = 1$  is transcendental if you are solving for  $x$  (because the  $x$  is inside a sine) but algebraic if you are solving for  $\sin(x)$  (as the equation is really a quadratic if you think of  $\sin(x)$  as the variable).

With these in mind, let's take a look at the main commands for solving equations and inequalities: Solve, NSolve, FindRoot, and Reduce:

Based on the command names the most obvious command to use when solving equations and inequalities is Solve. Solve is meant for solving equations only and not inequalities, however. It is geared towards finding all of the exact solutions where possible and typically works well on algebraic equations (in some cases it can provide a partial solution set to transcendental equations). The basic format for the Solve command for a single equation is `Solve[ equation, variable ]` and for systems of equations is `Solve[ list of equations, list of variables ]`. Here are a few examples of solving some basic equations:

```
Solve[ 3 x - 1 == 0, x]
{{x -> 1/3}}
Solve[ x^2 - 5 x + 3 == 0, x]
{{x -> 1/2 (5 - Sqrt[13])}, {x -> 1/2 (5 + Sqrt[13])}}
Solve[ x^2 + 4 x + 9 == 0, x]
{{x -> -2 - i Sqrt[5]}, {x -> -2 + i Sqrt[5]}}
Solve[ a x^2 + b x + c == 0, x]
{{x -> (-b - Sqrt[b^2 - 4 a c])/(2 a)}, {x -> (-b + Sqrt[b^2 - 4 a c])/(2 a)}}
```

*solving equations in one variable*

It's important to note the format of the solutions - they aren't given as "x equals this number, etc" but rather as a list of replacement rules. This is incredibly useful as one of the things you often do with the solutions to an equation is take them and plug them into something else. If you just want a list of the answers you can use the command `x /. Solve[ equation, x]` (or replace x with whatever variable you are using). You can see that your answers can be real or complex and even involve other unknowns - the last Solve command above basically generates the quadratic formula.

Here are a few examples of using Solve on systems of equations with more than one variable:



```

Solve[ {x + y == 1, x^2 + y^2 == 4}, {x, y}]
{{x -> 1/2 (1 - Sqrt[7]), y -> 1/2 (1 + Sqrt[7])}, {x -> 1/2 (1 + Sqrt[7]), y -> 1/2 (1 - Sqrt[7])}}
Solve[ {x + y == 4, x^2 + y^2 == 4}, {x, y}]
{{x -> 2 - I Sqrt[2], y -> 2 + I Sqrt[2]}, {x -> 2 + I Sqrt[2], y -> 2 - I Sqrt[2]}}
Solve[ {x + 2 y + z == 3, x + y + z == 2, 3 x - 4 y + 9 z == 21}, {x, y, z}]
{{x -> -8/3, y -> 1, z -> 11/3}}
Solve[ {x + 2 y + z == 3, x + y + z == 2, 3 x - 4 y + 9 z == 21}, {z, y, x}]
{{z -> 11/3, y -> 1, x -> -8/3}}
Solve[ {x + y + z == 3, x - 6 y + z == 1}, {x, y, z}]
Solve::svars : Equations may not give solutions for all "solve" variables. >>
{{y -> 2/7, z -> 19/7 - x}}
Solve[ {x + y + z == 3, x - 6 y + z == 1}, {z, x, y}]
Solve::svars : Equations may not give solutions for all "solve" variables. >>
{{x -> 19/7 - z, y -> 2/7}}
Solve[ {x + y + z == 2, x - 2 y - 3 z == 1, 2 x - y - 2 z == 0}, {x, y, z}]
{}

```

*equations in more than one variable*

Again note that the solutions are always given as a list of replacement rules and by default you get both real and complex solutions. In the last three examples you don't get specific numbers. Two of the systems have infinitely many solutions (that's what the warning about not giving solutions for all "solve" variables is about) - when that happens Solve will try to give you a solution in which variables later in your "variable list" are solved for in terms of earlier ones - so the order in which you list them can make a difference (in one solution above you get  $z$  in terms of  $x$  and in another you get  $x$  in terms of  $z$ ). The empty braces  $\{\}$  in the last result indicate there aren't any solutions, which can happen (if you add the first two equations you will get  $2x - y - 2z = 3$  which is inconsistent with the third equation).

Sometimes Solve can simply fail to give you results or give you answers that look strange:

```

Solve[ x^3 == Sin[x], x]

Solve::nsmet : This system cannot be solved with the methods available to Solve. >>

Solve[x^3 == Sin[x], x]

Solve[ 3 x^5 - 15 x + 5 == 0, x]

{{x -> Root[5 - 15 #1 + 3 #1^5 &, 1]},
 {x -> Root[5 - 15 #1 + 3 #1^5 &, 2]}, {x -> Root[5 - 15 #1 + 3 #1^5 &, 3]},
 {x -> Root[5 - 15 #1 + 3 #1^5 &, 4]}, {x -> Root[5 - 15 #1 + 3 #1^5 &, 5]}}

```

*strange results from Solve*

In the first case Solve is simply unable to solve the equation (note that it even misses the easy solution  $x=0$ ). Remember, Solve is geared towards solving algebraic equations - the right-hand side here involves a sine and that makes the equation transcendental. So Solve is simply a poor choice of command to use for that equation.

The second equation is another matter entirely though - there are solutions (and exactly 5 of them as you would expect from the Fundamental Theorem of Algebra) but they are given in the form of a “Root” object. Root objects are what Mathematica uses to express solutions to algebraic equations at times when it can’t give an explicit answer for them but can still represent and work with them. As you progress in your mathematical education one of the facts you will encounter is that when you have equations of degree 1, 2, 3, and 4 you can always write down the solutions in terms of the coefficients of the equation, the basic operations of arithmetic, and the use of square roots, cube roots, and fourth roots. For example here is the solution to a degree 3 equation:

```

Solve[ x^3 - 5 x + 1 == 0, x]

{{x ->  $\frac{\left(\frac{1}{2}(-9 + i\sqrt{1419})\right)^{1/3}}{3^{2/3}} + \frac{5}{\left(\frac{3}{2}(-9 + i\sqrt{1419})\right)^{1/3}}},$ 
 {x ->  $-\frac{(1 + i\sqrt{3})\left(\frac{1}{2}(-9 + i\sqrt{1419})\right)^{1/3}}{2 \times 3^{2/3}} - \frac{5(1 - i\sqrt{3})}{2^{2/3}\left(3(-9 + i\sqrt{1419})\right)^{1/3}},$ 
 {x ->  $-\frac{(1 - i\sqrt{3})\left(\frac{1}{2}(-9 + i\sqrt{1419})\right)^{1/3}}{2 \times 3^{2/3}} - \frac{5(1 + i\sqrt{3})}{2^{2/3}\left(3(-9 + i\sqrt{1419})\right)^{1/3}}}}$ 

```

*the exact solutions to a cubic equation*

This is fairly involved but only uses the operations of arithmetic and roots (remember  $i$  is the square root of -1). Once your equations reach degree 5 or higher (or the solution of a system that involves degree 5 and higher equations in intermediate solution steps) you can’t always guarantee this will be the case, which means you can’t represent the solutions as explicit

numbers in the usual sense. Mathematica uses Root to work with the solutions in these cases. Root[ *expression* &, *k* ] represents the *k*<sup>th</sup> solution to *expression*=0, where *expression* uses a “blank” like #1 to represent an unnamed variable. So Root[ 5-15 #1 + 3 #1<sup>5</sup> &, 2] means the second solution to 3(something)<sup>5</sup>-15(something)+5=0. In one sense this doesn’t help; it’s essentially saying that among the solutions to  $3x^5 - 15x + 5 = 0$  are 5 numbers, all of which satisfy  $3(\text{something})^5 - 15(\text{something}) + 5 = 0$  (it’s kind of like saying the solutions are the solutions). But it does give Mathematica something to work with - it can numerically estimate these kinds of answers, plug them into other expressions and simplify, and so on:

```
Solve[ 3 x^5 - 15 x + 5 == 0, x]

{{x -> Root[5 - 15 #1 + 3 #1^5 &, 1]},
 {x -> Root[5 - 15 #1 + 3 #1^5 &, 2]}, {x -> Root[5 - 15 #1 + 3 #1^5 &, 3]},
 {x -> Root[5 - 15 #1 + 3 #1^5 &, 4]}, {x -> Root[5 - 15 #1 + 3 #1^5 &, 5]}}
N[%, 10]

{{x -> -1.569122792}, {x -> 0.3341667189}, {x -> 1.396819852},
 {x -> -0.080931889 - 1.506323234 i}, {x -> -0.080931889 + 1.506323234 i}}
solution1 = Root[5 - 15 #1 + 3 #1^5 &, 1];
Simplify[ 3 x^5 - 15 x /. x -> solution1]
-5
FullSimplify[ solution1 + solution1^2]
Root[-85 + 255 #1 - 105 #1^2 - 90 #1^3 + 9 #1^5 &, 2]
```

*working with Root objects generated by Solve*

In this case we were able to estimate all the solutions to 10 decimal places and verify that if *x* is the first solution to  $3x^5 - 15x + 5 = 0$  then  $3x^5 - 15x = -5$ . We also find out that if *x* is the first solution to  $3x^5 - 15x + 5 = 0$  then  $x + x^2$  must be the second solution to the equation  $9y^5 - 90y^3 - 105y^2 + 255y - 85 = 0$ . If by chance a Root object corresponds to the root of a polynomial of degree 4 or less you can convert it to a “normal” form using roots via the command ToRadicals:

```
ToRadicals[Root[1 - 5 #1 - 14 #1^2 - 4 #1^3 &, 1] ]
```

$$-\frac{7}{6} - \frac{1}{12} (1 - i\sqrt{3}) \left( \frac{1}{2} (-317 + 3i\sqrt{6303}) \right)^{1/3} - \frac{17 (1 + i\sqrt{3})}{3 \times 2^{2/3} (-317 + 3i\sqrt{6303})^{1/3}}$$

*converting a root expression to a more standard notation using ToRadicals*

Although Solve works equally well with complex numbers and real numbers there will be times that you only want specific types of solutions (like real solutions only). You can have Solve only return solutions of particular types by using the formats Solve[ *equation*, *variable*, *domain* ] and Solve[ *list of equations*, *list of variables*, *domain* ], where *domain* is one of

Mathematica's official pre-defined sets. By far the most common domain you would use is Reals although there are also types defined by Complexes (which is the default) and Integers:

```
Solve[ {x^2 + y^2 == 4, x == y^2 + 1}, {x, y}]
```

$$\left\{ \left\{ x \rightarrow \frac{1}{2}(-1 + \sqrt{21}), y \rightarrow -\sqrt{\frac{1}{2}(-3 + \sqrt{21})} \right\}, \left\{ x \rightarrow \frac{1}{2}(-1 + \sqrt{21}), y \rightarrow \sqrt{\frac{1}{2}(-3 + \sqrt{21})} \right\}, \right. \\ \left. \left\{ x \rightarrow \frac{1}{2}(-1 - \sqrt{21}), y \rightarrow -i\sqrt{\frac{1}{2}(3 + \sqrt{21})} \right\}, \left\{ x \rightarrow \frac{1}{2}(-1 - \sqrt{21}), y \rightarrow i\sqrt{\frac{1}{2}(3 + \sqrt{21})} \right\} \right\}$$

```
Solve[ {x^2 + y^2 == 4, x == y^2 + 1}, {x, y}, Reals]
```

$$\left\{ \left\{ x \rightarrow 1 + \frac{1}{2}(-3 + \sqrt{21}), y \rightarrow -\sqrt{\frac{1}{2}(-3 + \sqrt{21})} \right\}, \left\{ x \rightarrow 1 + \frac{1}{2}(-3 + \sqrt{21}), y \rightarrow \sqrt{\frac{1}{2}(-3 + \sqrt{21})} \right\} \right\}$$

```
Solve[ x^2 + x y + y^2 == 27, {x, y}, Integers]
```

$$\{ \{x \rightarrow -6, y \rightarrow 3\}, \{x \rightarrow -3, y \rightarrow -3\}, \\ \{x \rightarrow -3, y \rightarrow 6\}, \{x \rightarrow 3, y \rightarrow -6\}, \{x \rightarrow 3, y \rightarrow 3\}, \{x \rightarrow 6, y \rightarrow -3\} \}$$

*solving equations over the “domains” Reals and Integers*

Mathematica can even work with Root objects to determine if they are real numbers or not:

```
Solve[ 3 x^5 - 15 x + 5 == 0, x, Reals]
```

$$\{ \{x \rightarrow \text{Root}[5 - 15 \#1 + 3 \#1^5 \&, 1]\}, \\ \{x \rightarrow \text{Root}[5 - 15 \#1 + 3 \#1^5 \&, 2]\}, \{x \rightarrow \text{Root}[5 - 15 \#1 + 3 \#1^5 \&, 3]\} \}$$

*only 3 of the 5 solutions to the equation are real, even if we can't write them exactly*

One of the most common places you will need to use the domain Reals is when the solution to equations corresponds to the intersection of curves. You can graph the equations using ContourPlot and add the intersection points using an option Epilog which lays additional graphics over the plot. The format you will want to use for most situations looks like this:

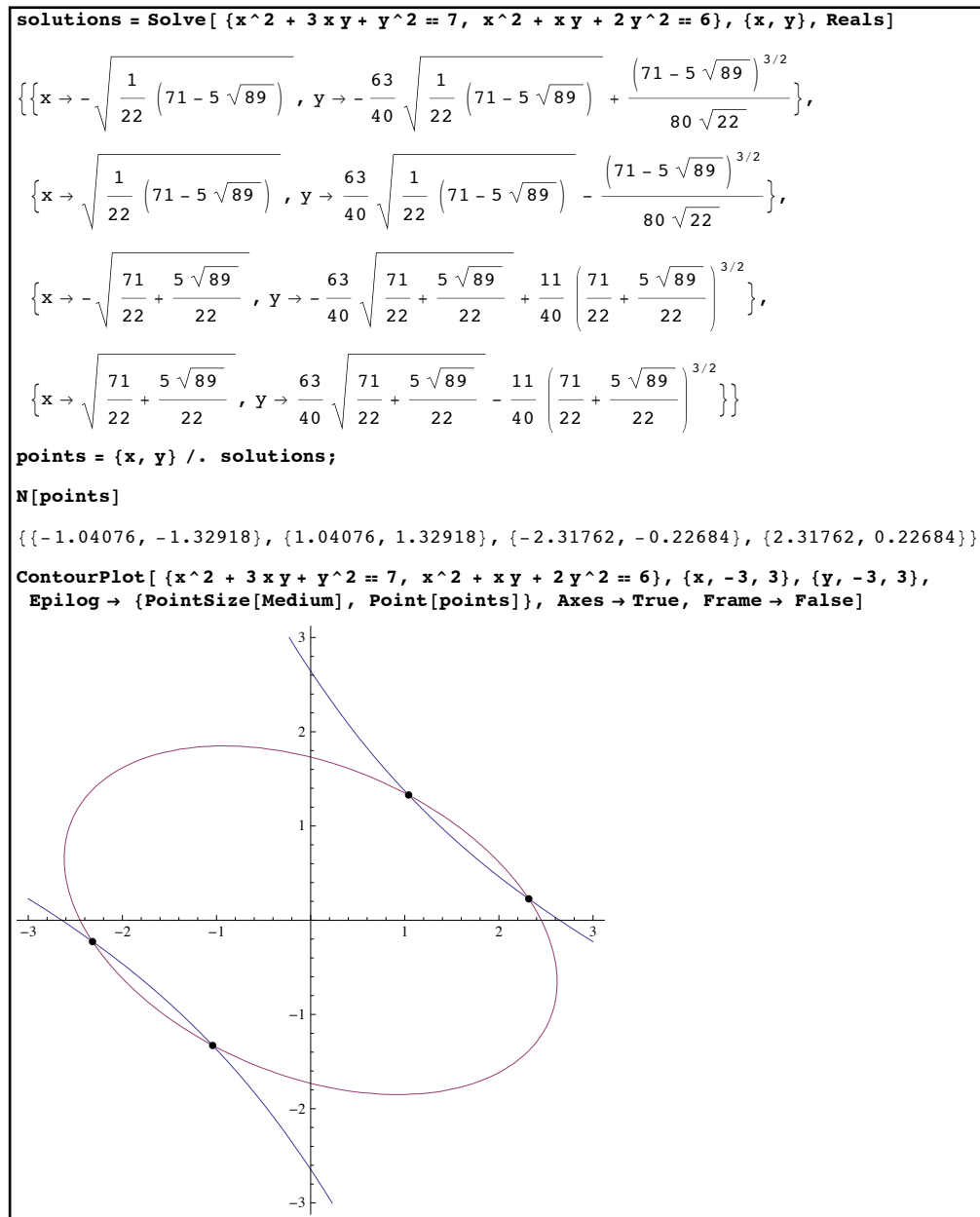
**Epilog**→{PointSize[Medium], Point[ *list of point coordinates* ]}

You can leave out the PointSize[Medium] if you like - it will simply render the points very small (which would make sense if you had dozens of intersections in a tiny space, but typically the intersections are spread out so it makes sense to use larger points). A simple step-by-step procedure to use for graphing the equations and their solutions looks something like this:

- 1) Use Solve to find the solutions over the real domain (you can store this in a variable like “solutions” or use % in the next step).
- 2) Store the point coordinates in a variable like “points” using a command like points={x,y} /. solutions from Solve (this takes advantage of the replacement rule format Solve uses).

- 3) Look at the sizes of the coordinates in “points” to determine what range of  $x$ - and  $y$ -values you will need to use for ContourPlot (you may need to use N to estimate the coordinates).
- 4) Use the command ContourPlot[ {equations}, x-range-list, y-range-list, Epilog→{PointSize[Medium], Point[ points ] } ]; add any other options you want to ContourPlot (like Axes or PlotLabel) that you want.

For example here is how you might graph the intersections of the equations  $x^2 + 3xy + y^2 = 7$  and  $x^2 + xy + 2y^2 = 6$ :



using Solve, Epilog, and ContourPlot to see the intersections of curves both exactly and graphically

In many of the places you use Solve you might not care about having the exact solutions to your equations - if for example all you are doing is graphing the intersections of curves you don't really need the exact coordinates of the points, just good estimates. The command NSolve is formatted just like the Solve command but gives numerical approximations instead of exact values:

```
NSolve[ 3 x^5 - 15 x + 5 == 0, x]

{{x -> -1.56912}, {x -> -0.0809319 - 1.50632 i},
 {x -> -0.0809319 + 1.50632 i}, {x -> 0.334167}, {x -> 1.39682}}

NSolve[ { y == x^2 + x, x^2 + y^2 == 20}, {x, y}]

{{x -> -2.49102, y -> 3.71414}, {x -> -0.55625 - 2.1674 i, y -> -4.94446 + 0.243832 i},
 {x -> -0.55625 + 2.1674 i, y -> -4.94446 - 0.243832 i}, {x -> 1.60351, y -> 4.17477}}

NSolve[ { y == x^2 + x, x^2 + y^2 == 20}, {x, y}, Reals]

{{x -> -2.49102, y -> 3.71414}, {x -> 1.60351, y -> 4.17477}}
```

*using NSolve to estimate solutions to equations, over both the Complexes and Reals domains.*

NSolve is typically faster and less memory intensive than Solve is (you might not notice the extra speed and efficiency in small examples but it would be more obvious if you were working with a hundred large equations at once). Like Solve it is meant for algebraic equations, although you won't have to worry about Root objects appearing in the solutions as you do in Solve. One potential problem that NSolve can encounter that Solve doesn't has to do with roundoff error. If the coefficients in your equations are either very large or very small the numerical procedures that NSolve uses can give estimates which aren't right. For example make the polynomial you get by multiplying the factors  $x - k^2$  together where  $k$  runs from 1 to 25. The roots of this polynomial should be the numbers 1, 4, 9, 16, 25, ... 625. Several of the coefficients for the polynomial are very large - and when you try to set this polynomial equal to zero and solve the answers aren't quite right:

```
badpoly = Expand[ Product[ x - k^2, {k, 1, 25} ] ];
N[ Coefficient[ badpoly, x^2 ] ]

-1.79972 x 10^50

NSolve[ badpoly == 0, x]

{{x -> 1.}, {x -> 4.}, {x -> 9.}, {x -> 16.}, {x -> 25.}, {x -> 36.}, {x -> 49.},
 {x -> 64.}, {x -> 81.}, {x -> 100.}, {x -> 121.}, {x -> 144.}, {x -> 169.}, {x -> 196.},
 {x -> 225.002}, {x -> 255.999}, {x -> 289.001}, {x -> 324.002}, {x -> 361.004},
 {x -> 400.}, {x -> 440.998}, {x -> 484.001}, {x -> 529.}, {x -> 576.}, {x -> 625.}}
```

*slight errors introduced by NSolve when the coefficients get large*

While many of the solutions are correct some of them (like 225.002) are off by small but noticeable amounts. This is caused by the numerical procedures NSolve is using not keeping enough digits of accuracy, causing roundoff error. To get around this you can use the option

WorkingPrecision, which sets the number of digits of accuracy those procedures keep (so WorkingPrecision→25 tries to keep 25 places in the computations, although some decimal places may be lost as the calculations proceed). Setting WorkingPrecision to 15 clears the problem up in this case:

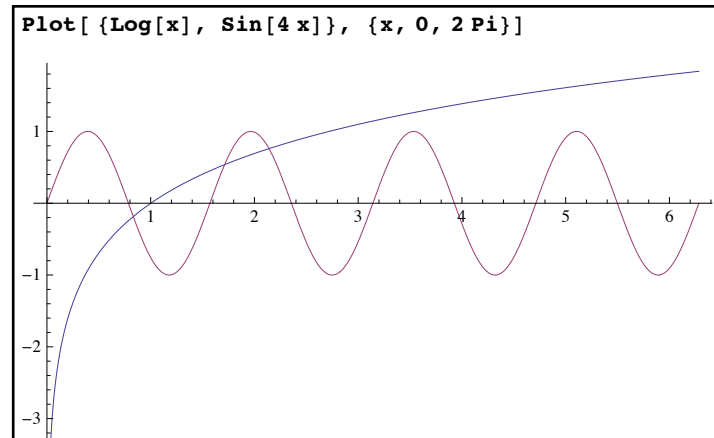
```
NSolve[badpoly == 0, x, WorkingPrecision -> 15]
{{x -> 1.000000000000000}, {x -> 4.000000000000000},
 {x -> 9.000000000000000}, {x -> 16.000000000000000},
 {x -> 25.000000000000000}, {x -> 36.000000000000000}, {x -> 49.000000000000000},
 {x -> 64.000000000000000}, {x -> 81.000000000000000}, {x -> 100.000000000000000},
 {x -> 121.000000000000000}, {x -> 144.000000000000000}, {x -> 169.000000000000000},
 {x -> 196.000000000000000}, {x -> 225.000000000000000}, {x -> 256.000000000000000},
 {x -> 289.000000000000000}, {x -> 324.000000000000000}, {x -> 361.000000000000000},
 {x -> 400.000000000000000}, {x -> 441.000000000000000}, {x -> 484.000000000000000},
 {x -> 529.000000000000000}, {x -> 576.000000000000000}, {x -> 625.000000000000000}}
```

*additional accuracy provided by WorkingPrecision*

You should try to set WorkingPrecision if you suspect the answers may not be accurate - we could detect this here in advance as we know the answers must be all whole numbers. This issue can also show up in working with equations where the coefficients are very small (which happens often in chemistry problems where concentrations are typically very small numbers).

Both Solve and NSolve are meant to solve algebraic equations - what if you have transcendental equations? These equations are often impossible to solve exactly and NSolve will often return an error as well. The command FindRoot fills in this gap left by Solve and NSolve. FindRoot tries to estimate a single solution to 1 or more equations - but it needs a starting value (sometimes called a “seed value”) for the variable(s) to look near. For a single equation the format for FindRoot is FindRoot[ *equation*, {*variable*, *seed value*} ] - this will try to find a solution to the *equation* near the *seed value*. Figuring out where to tell FindRoot where to look for solutions will take extra work on your part (typically by graphing the equations) but for difficult equations this is often the best you can do.

As an example suppose you wanted to estimate the real solutions to  $\ln(x) = \sin(4x)$ . Both Solve and NSolve will simply fail on this equation because it is transcendental. The solutions to the equation correspond to the places where the graphs of  $y = \ln(x)$  and  $y = \sin(4x)$  cross. As  $\ln(x)$  is only defined when  $x > 0$  and sine is trapped between -1 and 1 we can graph these over a reasonable range of  $x$ -values to get an idea of where the solutions might be:



*using graphs to estimate solutions to equations*

Looking at this picture it seems like there are 3 different intersection points (and therefore 3 solutions to the equation) - roughly at  $x=0.8$ ,  $1.7$ , and  $2.1$ . We can use each of these as the seed value for a FindRoot command:

```
FindRoot[ Log[x] == Sin[4 x], {x, 0.8}]
{x → 0.831726}

FindRoot[ Log[x] == Sin[4 x], {x, 1.7}]
{x → 1.71286}

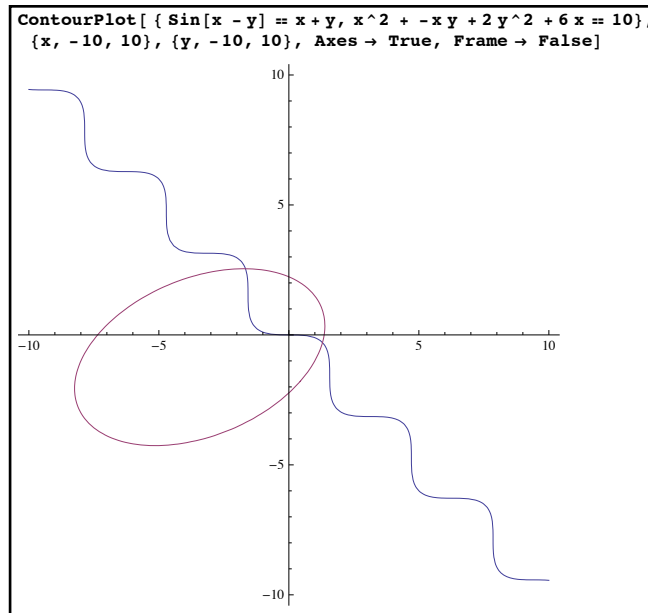
FindRoot[ Log[x] == Sin[4 x], {x, 2.1}, WorkingPrecision → 20]
{x → 2.1400474702286703963}
```

*using FindRoot to estimate solutions to a difficult equation (and increasing accuracy with WorkingPrecision)*

An alternate approach would be to graph the difference of the two sides ( $\ln(x) - \sin(4x)$  in this case) and using PlotRange→{-1,1} to focus on the area near the x-axis; when the difference graph crosses the x-axis you have found a seed value for FindRoot. This works well when the values of the graphs vary widely and it can be hard to see where the two graphs cross.

FindRoot will work on systems of equations as well - you just have to give a seed value for each variable as a list (FindRoot[ *list of equations*, {{*variable 1*, *seed 1*}, {*variable 2*, *seed 2*}, ...} ]. To find the seed values for two equations in two unknowns you will most likely to use ContourPlot to graph the equations and estimate the coordinates from the intersection points. For example suppose you have the system  $x^2 - xy + 2y^2 + 6x = 10$ ,  $\sin(x - y) = x + y$ . Using ContourPlot to graph these over a reasonable range (which can take some trial and error) yields a graph like this:





*graphing two equations in two unknowns to estimate where they cross*

Based on the graphs it looks like one solution occurs near the point (1.5, -0.5) and another near (-2, 3). Using each of these pairs to give seed values for FindRoot looks like this:

```
FindRoot[ { Sin[x - y] == x + y, x^2 + -x y + 2 y^2 + 6 x == 10}, {{x, 1.5}, {y, -.5}}]
{x -> 1.29469, y -> -0.294865}

FindRoot[ { Sin[x - y] == x + y, x^2 + -x y + 2 y^2 + 6 x == 10}, {{x, -2}, {y, 3}}]
{x -> -1.6682, y -> 2.54725}
```

*estimating solutions to a system of equations*

The three commands Solve, NSolve, and FindRoot are all about finding specific solutions to equations. None of them are suited to solving inequalities or equations that may have an infinite number of solutions (as often occurs with trigonometric equations). The solutions to these kinds of equations and inequalities are typically handled by the command Reduce, which has a different input and output structure than the other commands.

Reduce[ *logical statement, variable or variable list*] tries to break down the values for which the *logical statement* (which as mentioned earlier can use the symbols &&, !, and || for “and”, “not”, and “or”) is True into its simplest component parts. Like Solve and NSolve you can add a domain at the end if you want to restrict the solutions to sets like real numbers or integers:

```

Reduce[ x^3 - 3 x^2 + 2 x == 0, x]

x == 0 || x == 1 || x == 2

Reduce[ Sin[x] == 1/2, x]

C[1] ∈ Integers &&  $\left( x = \frac{\pi}{6} + 2\pi C[1] \mid \mid x = \frac{5\pi}{6} + 2\pi C[1] \right)$ 

Reduce[ y ≥ x^2 && y ≤ x + 2, {x, y}]

(x == -1 && y == 1) || (-1 < x < 2 && x^2 ≤ y ≤ 2 + x) || (x == 2 && y == 4)

Reduce[ y ≥ x^2 && y ≤ x + 2, {y, x}]

(y == 0 && x == 0) || (0 < y ≤ 1 && -√y ≤ x ≤ √y) ||

(1 < y < 4 && -2 + y ≤ x ≤ √y) || (y == 4 && x == 2)

Reduce[ x^2 + x y + y^2 ≤ 3 && x > 0, {x, y}, Integers]

(x == 1 && y == -2) || (x == 1 && y == -1) ||

(x == 1 && y == 0) || (x == 1 && y == 1) || (x == 2 && y == -1)

```

*using Reduce on equations and inequalities*

Note how different the output from Reduce is than the other commands - the answers are given not as a list of replacement rules but rather as another more specific logical statement. As in the other commands the order of the “solve variables” can make a big difference - variables towards the end of the list are solved for in terms of variables that appear earlier in the list. The infinite number of solutions to the trigonometric equation are all accounted for in a single statement by using integer multiples of  $2\pi$ . And the region bounded by  $y \geq x^2$  and  $y \leq x + 2$  is broken down into its endpoints along with an identification of which curve is on top and bottom of the region over the range  $-1 < x < 2$  (the second version of the command with the variable order reversed breaks the region up into two main parts, each of which has one boundary curve on the right and one on the left - this kind of information is very useful in Calculus).

If the results from Reduce boil down to a list of points (as is the case in the first and last examples above) you can easily convert the answer into the “replacement rules” format used by Solve, NSolve, and Reduce by using the command ToRules. ToRules takes the logical description of a single point and converts it into a set of replacement rules. ToRules will “thread” across an entire logical statement to convert each component point, but will need a set of exterior braces to create a true list of replacement rules (so you usually use it in the form {ToRules[%]} or something similar):

```

answer1 = Reduce[ x^2 + x y + y^2 ≤ 15 && x > 0 && y > 0, {x, y}, Integers]

(x == 1 && y == 1) || (x == 1 && y == 2) || (x == 1 && y == 3) ||

(x == 2 && y == 1) || (x == 2 && y == 2) || (x == 3 && y == 1)

answer2 = {ToRules[answer1]}

{{x → 1, y → 1}, {x → 1, y → 2}, {x → 1, y → 3}, {x → 2, y → 1}, {x → 2, y → 2}, {x → 3, y → 1}}

```

*using ToRules to convert Reduce output to a form similar to the one used by Solve*

One thing to watch out for with Reduce is that when working with systems it may not write the answer in the easiest possible form for a given purpose. For example when working with a system in  $x$  and  $y$  it might give a result like  $x == 2 \ \&\& \ y == x+1$  - this is “simple” from the point of Reduce (all the variables have been solved for) but most people would think the solution  $x == 2 \ \&\& \ y == 3$  would be simpler still. When this happens in Reduce just remember to use the notation `//.` to do a “repeated replacement” instead of `/.` :

```
Reduce[ x^2 + y^2 + z^2 == 25 && x + 2 y + z == 1 && x - y - z == 2, {x, y, z}]


$$\left( x = \frac{3}{14} (7 - \sqrt{35}) \mid \mid x = \frac{3}{14} (7 + \sqrt{35}) \right) \ \&\& \ y = 3 - 2 x \ \&\& \ z = -5 + 3 x$$


solutions = {ToRules[%]}


$$\left\{ \left\{ x \rightarrow \frac{3}{14} (7 - \sqrt{35}), y \rightarrow 3 - 2 x, z \rightarrow -5 + 3 x \right\}, \left\{ x \rightarrow \frac{3}{14} (7 + \sqrt{35}), y \rightarrow 3 - 2 x, z \rightarrow -5 + 3 x \right\} \right\}$$


{x, y, z} /. solutions


$$\left\{ \left\{ \frac{3}{14} (7 - \sqrt{35}), 3 - 2 x, -5 + 3 x \right\}, \left\{ \frac{3}{14} (7 + \sqrt{35}), 3 - 2 x, -5 + 3 x \right\} \right\}$$


{x, y, z} //. solutions


$$\left\{ \left\{ \frac{3}{14} (7 - \sqrt{35}), 3 - \frac{3}{7} (7 - \sqrt{35}), -5 + \frac{9}{14} (7 - \sqrt{35}) \right\}, \right.$$


$$\left. \left\{ \frac{3}{14} (7 + \sqrt{35}), 3 - \frac{3}{7} (7 + \sqrt{35}), -5 + \frac{9}{14} (7 + \sqrt{35}) \right\} \right\}$$

```

*in some cases the repeated replacement `//.` can work better with Reduce output than a single replacement given by `/.`*

Reduce can be a very powerful tool for solving all kinds of equations and inequalities as it allows you to pick out solutions to an equation or inequality that have a particular property by adding that property to the logical condition in Reduce. A simple example is finding a Quadrant II or III angle whose sine is  $1/3$ :

```
Reduce[ Sin[x] == 1 / 3 && Pi / 2 < x < 3 Pi / 2, x]


$$x == \pi - \text{ArcSin}\left[\frac{1}{3}\right]$$

```

*a Quadrant II angle whose sine is  $1/3$*

Or a solution to one polynomial equation which makes another quantity positive (or negative):

```

Reduce[ 3 x^2 - 6 x + 1 == 0 && 6 x - 6 > 0, x]

x ==  $\frac{1}{3} (3 + \sqrt{6})$ 

Reduce[ 3 x^2 - 6 x + 1 == 0 && 6 x - 6 < 0, x]

x ==  $\frac{1}{3} (3 - \sqrt{6})$ 

```

*picking out solutions to an equation with particular properties*

It is worth noting that although we commonly think of Reduce as the command to use with inequalities there are cases where Solve/NSolve will work with inequalities as well. If you have a mixed list of both equations and inequalities, Solve (and NSolve) will still work provided that the solutions are just isolated numbers (such as  $x = 1.2$ ) as opposed to a full range of values (like  $1 < x < 3$ ). For example consider the solutions to the equations  $x^2 + xy - y^2 = 10$  and  $x^2 + y^2 = 20$ . These equations together have four solutions, one isolated point in each quadrant in the plane. For a particular application you may only want the Quadrant I solution (where  $x > 0$  and  $y > 0$ ). By adding the basic inequalities to the list of equations you can get just the solutions in Quadrant I:

```

NSolve[{x^2 + x y - y^2 == 10, x^2 + y^2 == 20}, {x, y}, Reals]
{{x -> 4.24264, y -> -1.41421}, {x -> -4.24264, y -> 1.41421},
 {x -> -3.16228, y -> -3.16228}, {x -> 3.16228, y -> 3.16228}}

NSolve[{x^2 + x y - y^2 == 10, x^2 + y^2 == 20, x > 0, y > 0},
 {x, y}, Reals]
{{x -> 3.16228, y -> 3.16228}}

```

*getting all and just the Quadrant I solutions to a system of equations*

This sort of situation arises frequently when using Mathematica to solve equations from scientific applications - when the variables represent quantities like concentrations or speed they are not allowed to be negative. You do need to be careful when using Solve or NSolve in this way. For example if you remove the equation  $x^2 + y^2 = 20$  from the example above Mathematica will give you a warning and a misleading answer as with only one equation the true solution includes a full range of values for  $x$  and  $y$ . So only use Solve and NSolve in this way when you are fairly certain the solutions will only be isolated numbers.

We end this section with using Reduce to assist in a maximization problem: What is the largest value of  $3x + 4y$ , where  $x$  and  $y$  are non-negative integers for which  $3x + 5y \leq 20$  and  $y + 2x \geq 5$ ? First we use Reduce to locate the points  $(x, y)$  which satisfy all the requirements:

```

Reduce[ x ≥ 0 && y ≥ 0 && 3 x + 5 y ≤ 20 && y + 2 x ≥ 5, {x, y}, Integers]

(x == 1 && y == 3) || (x == 2 && y == 1) || (x == 2 && y == 2) ||
(x == 3 && y == 0) || (x == 3 && y == 1) || (x == 3 && y == 2) || (x == 4 && y == 0) ||
(x == 4 && y == 1) || (x == 5 && y == 0) || (x == 5 && y == 1) || (x == 6 && y == 0)

rules = {ToRules[%]}

{{x → 1, y → 3}, {x → 2, y → 1}, {x → 2, y → 2}, {x → 3, y → 0}, {x → 3, y → 1}, {x → 3, y → 2},
{x → 4, y → 0}, {x → 4, y → 1}, {x → 5, y → 0}, {x → 5, y → 1}, {x → 6, y → 0}}

```

*all the integer points in the required region*

Now that we have the points we need to identify which of those points gives the maximum value for  $3x + 4y$ . As the number of points is fairly small we can just look at each point, pair it with its value for  $3x + 4y$ , and just pick the largest:

```

{{x, y}, 3 x + 4 y} /. rules

{{{1, 3}, 15}, {{2, 1}, 10}, {{2, 2}, 14}, {{3, 0}, 9}, {{3, 1}, 13},
{{3, 2}, 17}, {{4, 0}, 12}, {{4, 1}, 16}, {{5, 0}, 15}, {{5, 1}, 19}, {{6, 0}, 18}}

```

*pairing points and their values together using a replacement*

Looking at the list it looks like the maximum value is 19 and it happens only at the point (5,1). If the list of possibilities was a lot longer (like say 1000 points) then visually going through and finding the maximum value would be pretty tedious. It's possible to have Mathematica do that for us, but that will require some additional commands that we will introduce in the section on list operations.

## Section 2.6 Homework – Solving Equations and Inequalities

- 1) Compare and contrast Solve, NSolve, FindRoot, and Reduce.
- 2) Solve the equation  $x^2 - 3x - 7 = 0$  for  $x$ .
- 3) Solve the system of equations  $2x + 3y = 1$ ,  $5x - 7y = 10$ .
- 4) Solve the equation  $x^3 + mx + n = 0$  for  $x$ .
- 5) Solve the system of equations  $x + 3y + 2z = 1$ ,  $x - y + 3z = 5$ ,  $3x + y + z = 1$ .
- 6) Solve the system of equations  $x + 3y + 2z = 1$ ,  $x - y + 3z = 5$ ,  $3x + 5y + 7z = 7$ .
- 7) Solve the system of equations  $x + 3y + 2z = 1$ ,  $x - y + 3z = 5$ ,  $3x + 5y + 7z = 11$ .
- 8) Solve the system of equations  $x + y = 2$ ,  $x^2 + x + y^2 - 2y = 10$ . Plot the corresponding curves together with the points of intersection.
- 9) Find the real solutions to  $x^2 + x + y^2 = 1$ ,  $y = x^2$ . Graph the corresponding curves together with the points of intersection.
- 10) Evaluate the commands `N[ Solve[ {3a^2-m==0, 2m-2a^3-1==0},{a,m}] ]` and `NSolve[ {3a^2-m==0, 2m-2a^3-1==0},{a,m}]` and then compare the results. Are they what you expected? Try to figure out why the `N[ Solve[ ... ]` command gives a different result (hint: look at the solutions without the `N` and then think about what the `N` would do to that).

- 11) Solve the equation  $x^2 + xy + y^2 = 100$  over the reals and over the integers.
- 12) Numerically estimate the real solutions to  $x^6 + x^5 - 10x^4 + 3x - 1 = 0$  using NSolve.  
Use NSolve and WorkingPrecision to get 20 decimal places of accuracy.
- 13) Numerically estimate the real solutions to the system  $x^2 + xy + y^2 = 100$ ,  $x^3 + y^3 = 2$ .  
If you allowed both real and complex solutions how many answers would there be?
- 14) Estimate the real solutions to  $x + \sqrt{x} = \ln(x) + 4$ .
- 15) Estimate the real solutions to  $x^2 - 4x + 1 = \sin(x)$ .
- 16) Estimate the real solutions to the system  $\cos(xy - x + y) = \frac{2}{3}$ ,  $x^4 + y^4 = 1$ . Graph the curves together with their points of intersection.
- 17) Use Reduce to find all solutions to  $\sin(2x) = \frac{1}{3}$  from 0 to  $2\pi$ .
- 18) Use Reduce to find all solutions to  $\sin(x) + \cos(x) = 1$ .
- 19) Use Reduce to describe the region which is inside the circle  $x^2 + y^2 = 9$  and above  $y = x + 1$  (by “describe the region” we mean identify which curve is on top and which curve is on bottom over various ranges of  $x$ -values).
- 20) Use Reduce to find all of the points inside the region  $x^2 + xy + y^2 \leq 10$  with integer coordinates. Graph the curve  $x^2 + xy + y^2 = 10$  together with these points.
- 21) Look up the command SolveAlways and describe what it does. Evaluate the commands `Solve[ x(m^2-4)+(m^3-8)==0,x]` and `SolveAlways[ x(m^2-4)+(m^3-8)==0,x]` and explain why the differences occur.

## Section 2.7 - Defining Functions

There may be times in Mathematica when you need to use the same formula over and over again - you may need to graph it, plug ten different values into it, and solve an equation that involves it. While you can certainly use copy-and-paste to make this easier the best thing to do would be to define your own function for the formula and then use it with standard function notation. This is easy to do in Mathematica although the usual “ $f(x) =$ ” notation from mathematics splits into two versions - the usual  $=$  symbol and the notation  $:=$ . If you wanted to define the function  $f(x) = e^x \cos(x)$ , you could use the notation `f[x_] = Exp[x] Cos[x]` or `f[x_] := Exp[x] Cos[x]`. Both definitions use square brackets (which is standard Mathematica notation for all functions and commands) and an underscore after the variable in the “name” of the function (the `_` defines what Mathematica calls a “blank”, which indicates what is used there goes into all occurrences on the right hand side of the equal sign). For  $e^x \cos(x)$  (and all other functions whose outputs are numbers) there is no real difference between the  $:=$  and  $=$  notations:

```
f[x_] = Exp[x] Cos[x];
g[x_] := Exp[x] Cos[x];
f[3]

e3 Cos[3]
g[3]

e3 Cos[3]
f[Log[x]]

x Cos[Log[x]]
g[Log[x]]

x Cos[Log[x]]
```

*for normal functions there is little difference between  $=$  and  $:=$*

The difference between  $=$  and  $:=$  becomes apparent when you realize your function definitions can go beyond just “functions of numbers” to include Mathematica commands like `Plot`, `Solve`, `Reduce`, and so on. When you define a function using  $=$  then any Mathematica commands that are part of the definition are evaluated right when you define the function and then never again (the  $=$  notation is sometimes called “immediate evaluation”). When you use  $:=$  in your function any Mathematica commands are not evaluated when you define the function - instead they are applied every time you use the function (“delayed evaluation”). A good example to see the difference is a function which involves factoring. Let `f[mess_] = Factor[mess^2-mess]` and let `g[mess_] := Factor[mess^2-mess]`. After defining these functions evaluate them both at  $x^2 - 1$ :

```

f[mess_] = Factor[mess^2 - mess]
(- 1 + mess) mess
g[mess_] := Factor[mess^2 - mess]
f[x^2 - 1]
(- 2 + x^2) (- 1 + x^2)
g[x^2 - 1]
(- 1 + x) (1 + x) (- 2 + x^2)

```

*the same idea implemented with = and :=*

Even Mathematica's output for the original definitions are different. The function `f` shows the output  $(mess-1)mess$  as the `Factor` command is immediately applied to  $mess^2-mess$ . The function `g` doesn't have any output at all as the `Factor` command is not evaluated - this is common when using the `:=` notation. `f[x^2-1]` only gives you two factors as it is replacing  $mess$  with  $x^2 - 1$  in  $mess(mess-1)$  - the `Factor` command is never used again. `g[x^2-1]` breaks down into 3 factors as the `Factor` command is being used rather than some previously calculated formula.

Whenever you define a function you can always undefine it through the use of the `Clear` command (which we previously used for "undefining" variables) - so `Clear[f,g]` would remove both definitions above and let us reuse the names for new functions.

There is a third notation for functions which you can use in many cases - "pure" function notation. The idea is to define a function without giving it a name at all. The notation used for pure functions is the same we saw in the `Root` objects earlier - you use `#1` to represent a variable (or `#1` and `#2` for two variables, etc.) and a single `&` to represent "end of the formula". So the pure function representation of  $f(x) = x^3$  would be `#1^3 &`:

```

(#1^3 &) [10]
1000
(#1^3 &) [t]
t^3
(#1 / #2 &) [x, 5]
x
5
(Sqrt [#1^2 + #2^2 + #3^2] &) [x, y, 10]
√100 + x^2 + y^2

```

*pure function notation*

For simple functions and applications pure function notation is probably more cumbersome than using either the `=` or `:=` notations. There are two places where it is very useful - applying your



function to every element of a list (or more complicated objects like statistical distributions or region objects) and to choose only those elements of a list which have a certain property. We'll formally discuss this in the section on lists but here is an example of each in action (using the commands Map and Select):

```
Map[#1 - Sin[#1] &, {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}]
{1 - Sin[1], 2 - Sin[2], 3 - Sin[3], 4 - Sin[4], 5 - Sin[5],
 6 - Sin[6], 7 - Sin[7], 8 - Sin[8], 9 - Sin[9], 10 - Sin[10]}
Select[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, IntegerQ[#1 / 3] &]
{3, 6, 9}
```

using *pure function notation*: computing 10 values at once with Map and only choosing multiples of 3 with Select

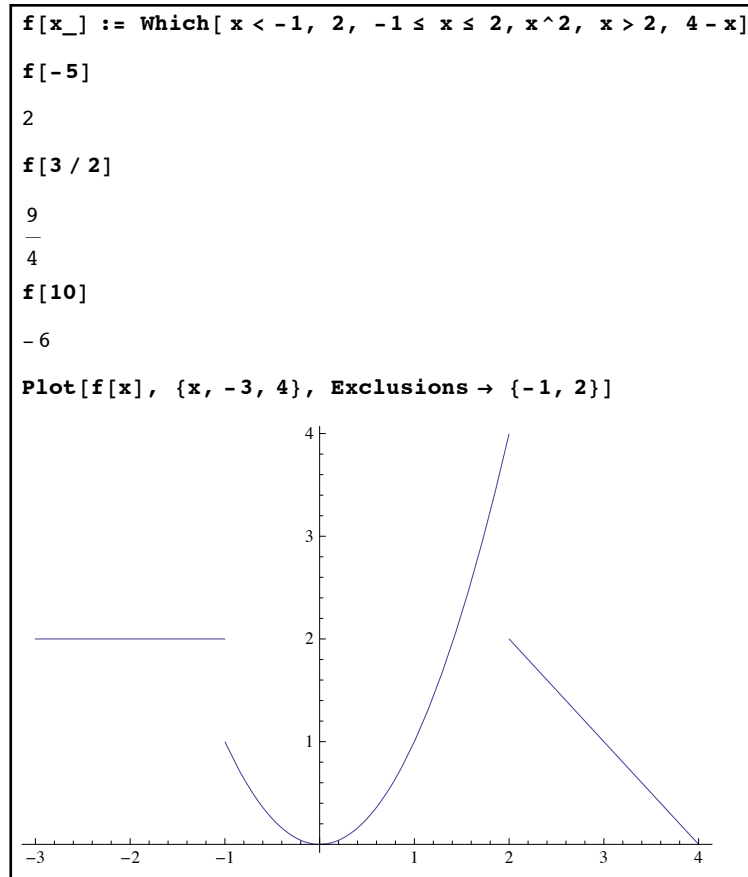
One of the more common kinds of functions you may need to set up in Mathematica are “conditional” functions - ones whose definition comes in pieces and the piece being used depends on some property of the input (the absolute value function can be thought of as a piecewise function as it behaves one way on the negatives and another way on the non-negatives). The three main commands for conditional functions are If, Which, and Piecewise:

If[ *condition*, *true-value*, *false-value*]: For any given input If will evaluate the *condition* (which often involves inequalities or functions that give True or False such as IntegerQ); if the *condition* is True then *true-value* is returned and if the *condition* is False you get the *false-value*. There may be some cases where the given condition may not resolve as True or False (think of trying to plug the complex number  $2+3i$  into  $x > 2$ ); if you want to allow for that option you can use the format If[ *condition*, *true-value*, *false-value*, *unresolvable-value*]. You will want to use the := notation for a function which uses If as you want the If command to be evaluated every time.

```
f[x_] := If[x > 5, x^2, 2 x, "undefined"]
f[10]
100
f[3]
6
f[1 + 2 I]
Greater::nord : Invalid comparison with 1 + 2 i attempted. >>
undefined
f[x^2]
undefined
```

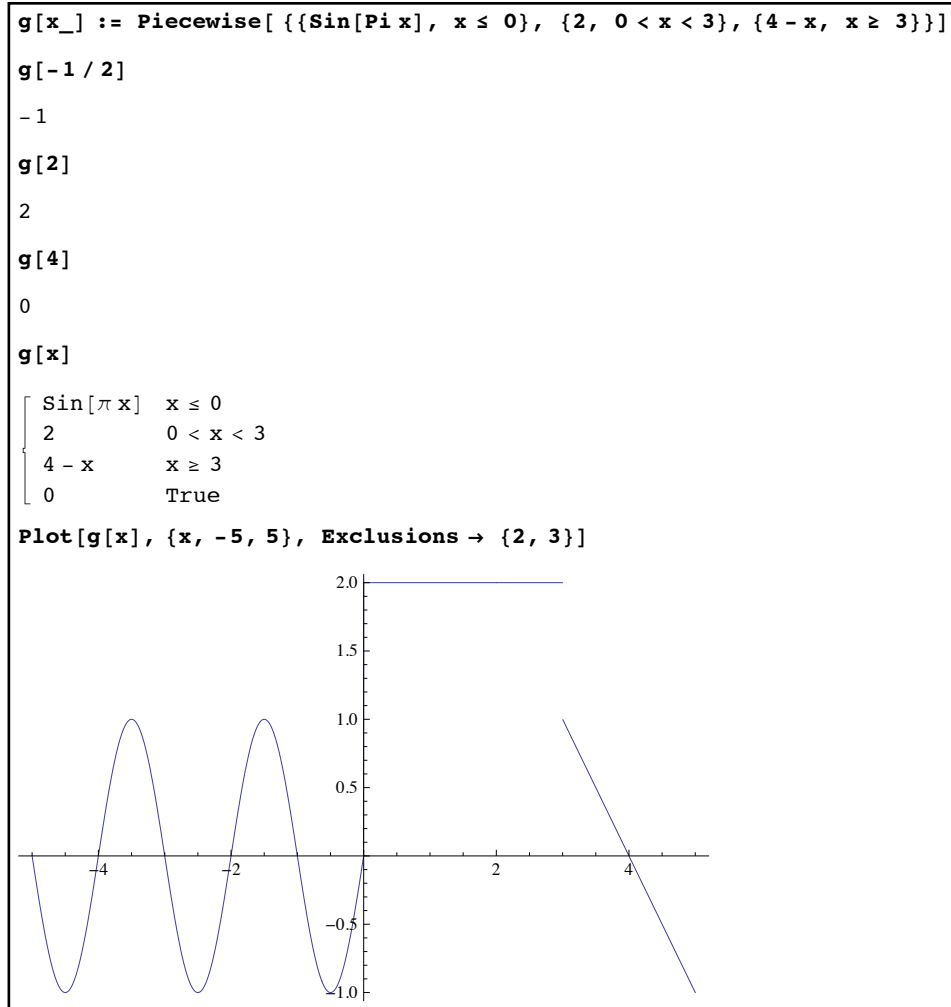
using If to define a conditional function which includes a case for “unresolvable”

**Which**[ *condition1*, *value1*, *condition2*, *value2*,...]: For any given input Which will return the first value whose *condition* is True. As Which starts at the beginning of the sequence there is no need to make sure the conditions are mutually exclusive - once Which finds a condition which is True it returns a value and stops. Which cannot handle an “unresolvable” comparison like If can, but you can have a default value by making the last condition the word True (if the Which statement gets that far down the sequence it would automatically give the value after True as the answer).



using Which to define a piecewise function - Exclusions is used to remove the “jumps” where the cases join

**Piecewise**[{{*value1*, *condition1*}, {*value2*, *condition2*},...]: Although the input is structured differently Piecewise works in the same fashion as Which - the first *condition* which is True gets its value returned. By default if none of the conditions hold for an input the value 0 is returned by default; you can replace 0 with whatever value you want by using the format Piecewise[{{*value1*, *condition1*}, {*value2*, *condition2*},...},*default-value*]. In Piecewise typically all of the conditions will be inequalities and Piecewise will use the standard mathematical notation for piecewise-defined functions.



*using Piecewise to evaluate and graph a piecewise-defined function*

Given the similarities between Which and Piecewise which should you use? Generally people tend to think of both If and Which as “programming” commands and Piecewise is thought of as a “mathematical” function. So if the cases of your function are based on numerical values try using Piecewise first; if the cases involve more complicated ideas like EvenQ or IntegerQ try Which or If.

Another type of function which you can use in Mathematica is a “recursive” function. A recursive function is one which needs its earlier values to compute later ones. For example you could define  $f(0) = 5$  and  $f(n) = f(n - 1) + 2$  for the values  $n=1,2,3,\dots$  (recursive functions often involve integer or natural number inputs so  $n$  is a common choice of variable name). What is  $f(4)$ ? Well,  $f(4) = f(3) + 2$ . Of course, this begs the question of what  $f(3)$  is.

$f(3) = f(2) + 2$ .  $f(2) = f(1) + 2$ , and  $f(1) = f(0) + 2$ . But we know the value of  $f(0) = 5$ , so  $f(1) = 5 + 2 = 7$ ,  $f(2) = 7 + 2 = 9$ ,  $f(3) = 9 + 2 = 11$ , and  $f(4) = 11 + 2 = 13$ . So to figure out  $f(4)$  we needed to compute  $f(1)$ ,  $f(2)$  and  $f(3)$ . This is what “recursion” is all about. You have a starting value (or maybe values) and a rule which determines later values in terms of

earlier ones. You can define these functions in Mathematica exactly the way we did above. We could code the function  $f$  as  $f[0]=5$  and  $f[n_]:=f[n-1]+2$ :

```
f[0] = 5;
f[n_] := f[n - 1] + 2
f[1]
7
{f[2], f[3], f[4]}
{9, 11, 13}
f[200]
405
```

*a recursively defined function*

The semi-colon after  $f[0]=5$  prevents Mathematica from just parroting the value 5 back at you when the function is defined. You will need to use the  $:=$  notation for recursive functions as the rule  $f(n) = f(n - 1) + 2$  needs to be used every time the function is used (so delayed evaluation is necessary).

When defining recursive functions there's no need to restrict yourself to functions which rely on just one previous value for their recursion. You can have functions that rely on the previous two values, three values, etc. For example you could define a function  $g[n]$  by  $g[0]=3$ ,  $g[1]=4$ , and  $g[n_]:=2g[n-1]+4g[n-2]$ . In general you will need to provide as many consecutive "starter" values as the "length" of the recursion ( $g[n]$  relies on the previous 2 values, so you need 2 consecutive starting values;  $g[n_]:=g[n-1]+g[n-4]$  would require 4 consecutive starter values).

One potential issue with recursive function is the danger of getting caught in an infinite loop. In the previous example if we had not given the definition of  $f$  at 0 then  $f(0)$  would require  $f(-1)$ ,  $f(-1)$  would require  $f(-2)$ , and so on forever. To prevent this Mathematica has a built-in restriction on how many steps a recursive function is allowed before it hits a "known value" - 1024 steps. So if we had tried to evaluate  $f[2000]$  we would have gotten an error of "Recursion depth of 1024 exceeded.". Earlier versions of Mathematica had a lower limit of 256 steps; if you need to check the limit you can find it by evaluating the command `$RecursionLimit`.

Another issue with recursive functions as we have them set up is they can be very time intensive. If we use the definition  $f[0]=5$  and  $f[n_]:=f[n-1]+2$  for a function and then ask for  $f[20]$ , the intermediate values need to be recalculated over and over ( $f[10]$  is needed for  $f[11]$ ,  $f[12]$ ,  $f[13]$ , and so on). You can speed this up by having Mathematica store the values it has previously calculated so it doesn't need to find them over and over again. A simple alteration to the function definition will do this -  $f[n_]:=f[n]=f[n-1]+2$  (the intermediate  $f[n]=$  stores the values as they are computed). Although this can take more computer memory the savings in speed typically more than make up for it. In addition this will let you find "large" values like

f[4000] - simply compute some intermediate values like f[1000], f[2000], and f[3000] along the way. As these values are being stored that means you are only using recursive jumps of 1000 steps (which are under the 1024 step limit) and won't give an error.

## Section 2.7 Homework – Defining Functions

After each homework problem you should use `Clear[function name]` to remove the definitions of any functions. This will prevent conflicts and errors between different homework problems that use the same function name.

- 1) Explain the difference between `=` and `:=` when defining functions.
- 2) Explain what it meant by a “pure function” and pure function notation.
- 3) Define the function  $f(x) = x^2 \sin\left(\frac{\pi x}{4}\right)$  and use it to compute the values of  $f(1)$ ,  $f(3)$ ,  $f(5)$ , ...  $f(11)$ .
- 4) Use pure function notation to repeat problem 3, using `Map` and the list  $\{1,3,5,7,9,11\}$ .
- 5) Define the function  $g(x, y) = \frac{x}{y^2}$ . Use this to find  $g(2,3)$ ,  $g(5,1)$ ,  $g(0,5)$ , and  $g(5,0)$ .
- 6) Define a function `PolarToCartesian` which takes two numbers  $r$  and  $\theta$  (which represent polar coordinates) and returns a list of the corresponding Cartesian coordinates  $(x,y)$ . (for those who have not worked with polar coordinates the polar point  $(r,\theta)$  corresponds to the Cartesian point  $(r \cos(\theta), r \sin(\theta))$ ).
- 7) Give `If`, `Which`, and `Piecewise` versions of a function which is equal to  $x^2$  if  $x > 2$  and  $2x + 3$  if  $x \leq 2$ . Graph any of these functions from -1 to 4.
- 8) Give `Which` and `Piecewise` versions of a function which is equal to  $\sin(2x)$  if  $x > \pi$ ,  $\cos(x)$  if  $-\pi < x \leq \pi$ , and 1 if  $x \leq -\pi$ . Graph this function from -6 to 6.
- 9) Use `If` to define a function which is equal to  $x^3$  if  $x \geq 0$ ,  $x^2$  if  $x < 0$ , and the word Indeterminate if  $x$  can't be used in either inequality.
- 10) Define a function `MyPlot` which takes a quantity *formula* and graphs it from 0 to  $2\pi$ . What happens if you try to use the “`=`” version of function definition for `MyPlot`, and why?
- 11) Define a function  $g(x, y)$  which is equal to 1 if  $x$  and  $y$  have the same sign and -1 otherwise.
- 12) Define a function  $f(n)$  by  $f(1) = 3$  and  $f(n) = 5f(n - 1) - 3$ . What is  $f(5)$ ?  $f(10)$ ? Numerically estimate  $f(1500)$ .
- 13) Define a function  $g(n)$  by  $g(1) = 1$  and  $g(n) = 3g(n - 2) - 11$ . What is  $g(9)$ ? Explain what happens when you try to evaluate  $g(10)$ .
- 14) Define a function  $h(n)$  by  $h(1) = 1$ ,  $h(2) = 2$ , and  $h(n) = 2h(n - 1) - 3h(n - 2)$ . Find  $h(11)$ ,  $h(12)$  and  $h(30)$ .
- 15) Look up the command `RSolve`. What does it do? Apply it to the functions in problems 12 and 14.

# Chapter 3 - Additional Pre-Calculus Mathematica Topics

```
In[126]:= sequence = {p, q, r, !r, q || !r, p && (q || !r)};
          headings = Map[TraditionalForm, sequence]
```

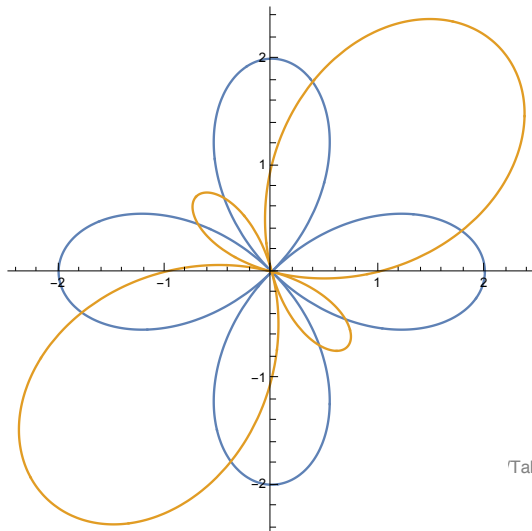
```
Out[127]:= {p, q, r, ¬r, q ∨ ¬r, p ∧ (q ∨ ¬r)}
```

```
In[128]:= TableForm[ BooleanTable[ sequence, {p, q, r}], TableHeadings → {None, headings} ]
```

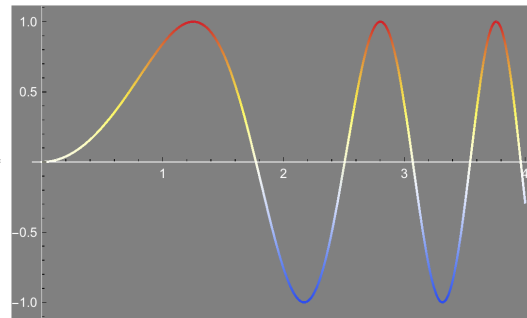
```
Out[128]//TableForm=
```

$p$	$q$	$r$	$\neg r$	$q \vee \neg r$	$p \wedge (q \vee \neg r)$
True	True	True	False	True	True
True	True	False	True	True	True
True	False	True	False	False	False
True	False	False	True	True	True
False	True	True	False	True	False
False	True	False	True	True	False
False	False	True	False	False	False
False	False	False	True	True	False

```
PolarPlot[ {2 Cos[2 t], 1 + 2 Sin[2 t]}, {t, 0, 2 Pi}]
```



```
Plot[ Sin[x^2], {x, 0, 4}, ColorFunction → "TemperatureMap",
      Background → Gray, AxesStyle → White]
```



```
TableForm[Table[a b, {a, 1, 5}, {b, 1, 5}],
           TableHeadings → {Range[5], Range[5]}]
```

```
TableForm=
```

	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25

```
Maximize[ {x^2 Cos[3 x] - x Cos[x], 0 ≤ x ≤ 2 Pi}, x]
```

```
{-2 π + 4 π^2, {x → 2 π}}
```

## Section 3.1 - An Introduction

In Chapter 2 we established some of the basics of Mathematica - the basic formatting of notebooks, elementary functions, graphing, and many of the operations and computations of algebra and trigonometry. Mathematica's capabilities go well beyond these pre-calculus basics into calculus and beyond. We will cover some of these more advanced uses of Mathematica in Chapter 4, but before we get there there are still many smaller topics that don't require calculus where Mathematica can be very useful. So this chapter is devoted to a selection of smaller topics, which include:

- Working with lists
- Basic descriptive statistics and curve fitting
- Interactive computations using Manipulate
- Advanced directives for working with graphs
- Importing and exporting from Mathematica
- Additional graphing commands
- Optimization
- Working with logic

## Section 3.2 - The Basics of Lists

We have already seen many examples of lists in Mathematica. They are used in the output of commands like `FactorInteger` and `Solve` and appear in the formatting the inputs of commands like `Plot` and `Piecewise`. We haven't focused on the lists themselves though - either how to generate them (other than typing them directly using the `{ }` notation), manipulate them, or use them in their own right.

There are many ways to have Mathematica create certain lists for you - the two you will encounter most often are `Table` and `Range`, and if you enjoy exploring ideas probability you may also use `RandomChoice` from time to time:

`Table[formula, {variable, start, finish}]`: This creates the a list using *formula* as *variable* goes from *start* to finish by 1's. The objects produced by *formula* don't have to be numbers - they can be algebraic objects, graphs, or even lists themselves. If you don't want the *variable* to go up by 1 each time you can use the more detailed version `Table[formula, {variable, start, finish, step}]`. This will let the value for *variable* go up by *step* each time (if *step* is negative the value of the *variable* will go down, in which case *start* should be larger than *finish*).

Here are several examples of using `Table` to create different lists:

A simple list going from 10 to 20:

**Table[*k*, {*k*, 10, 20}]**

{10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}

Another simple list going from 30 down to 10 by 2's:

**Table[*k*, {*k*, 30, 10, -2}]**

{30, 28, 26, 24, 22, 20, 18, 16, 14, 12, 10}

The 50th through 60th primes:

**Table[Prime[*t*], {*t*, 50, 60}]**

{229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281}

Powers of *x*:

**Table[*x*^*k*, {*k*, -7, 7}]**

$\left\{ \frac{1}{x^7}, \frac{1}{x^6}, \frac{1}{x^5}, \frac{1}{x^4}, \frac{1}{x^3}, \frac{1}{x^2}, \frac{1}{x}, 1, x, x^2, x^3, x^4, x^5, x^6, x^7 \right\}$

A list of points on the graph of  $y = x^2$ :

**Table[{*x*, *x*^2}, {*x*, -5, 5}]**

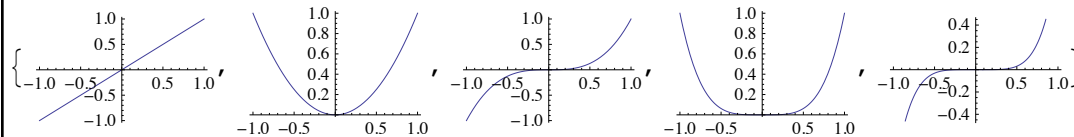
{{-5, 25}, {-4, 16}, {-3, 9}, {-2, 4}, {-1, 1}, {0, 0}, {1, 1}, {2, 4}, {3, 9}, {4, 16}, {5, 25}}

A list of replacement rules for powers of sine:

**Table[Sin[*x*]^*j* → (1 - Cos[*x*]^2)^(*j*/2), {*j*, 2, 10, 2}]**

$\{\text{Sin}[x]^2 \rightarrow 1 - \text{Cos}[x]^2, \text{Sin}[x]^4 \rightarrow (1 - \text{Cos}[x]^2)^2,$   
 $\text{Sin}[x]^6 \rightarrow (1 - \text{Cos}[x]^2)^3, \text{Sin}[x]^8 \rightarrow (1 - \text{Cos}[x]^2)^4, \text{Sin}[x]^{10} \rightarrow (1 - \text{Cos}[x]^2)^5\}$

**Table[Plot[*x*^*i*, {*x*, -1, 1}, ImageSize → 100], {*i*, 1, 5}]**



*using Table to create different kinds of lists*

**Range[*number*]**: Range applied to a positive *number* will create the list of natural numbers from 1 to *number* (if *number* isn't whole the list will stop at the last whole number before *number*). **Range[*start*, *finish*]** will create the simple list {*start*, *start* + 1, *start* + 2, ...} up to *finish*. **Range[*start*, *finish*, *step*]** will create the same kind of list going up by *step* each time instead of 1. All of these lists could be generated by using Table but Range is quicker and easier to use.



```

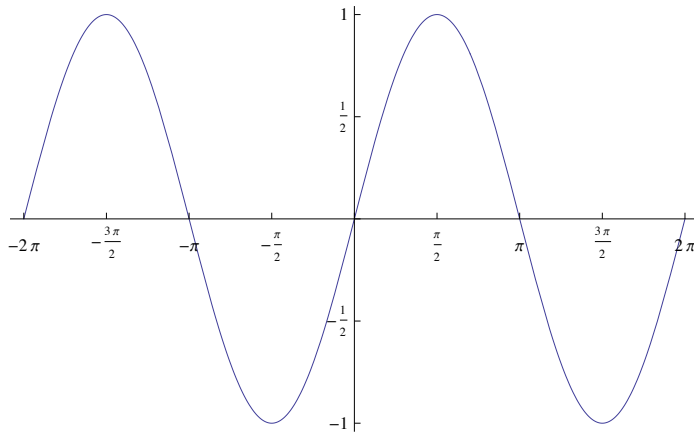
Range[10]
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

Range[3.5, 10]
{3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5}

Range[-4 Pi, Pi, Pi]
{-4 Pi, -3 Pi, -2 Pi, -Pi, 0, Pi}

Plot[Sin[x], {x, -2 Pi, 2 Pi}, Ticks -> {Range[-2 Pi, 2 Pi, Pi/2], Range[-1, 1, 1/2]}]

```



*using Range to build simple lists - including a nice way to get ticks marks for a graph without a lot of typing*

**RandomChoice[ list, length]:** RandomChoice builds a new list of the given *length*, where each element of the new list has been randomly chosen from the given *list*. So RandomChoice[ {"Heads", "Tails"}, 10] would simulate 10 flips of a fair coin and RandomChoice[ {1,2,3,4,5,6}, 20] would simulate 20 rolls of a fair die. If you don't want the choices to be equally likely you can weight the choices using the format RandomChoice[ {weight 1, weight 2,...} -> {choice 1, choice 2,...}, length] where the weights represent how likely each choice is (the weights will automatically be normalized to add up to 1). So if you wanted to simulate 30 flips of a coin which was weighted to come up heads 70% of the time and tails 30% of the time you could use RandomChoice[ {.7, .3} -> {"Heads", "Tails"}, 30] or RandomChoice[ {7,3} -> {"Heads", "Tails"}]. RandomChoice[list] with no length would just return a single random list element instead of a list.

```

RandomChoice[{"Heads", "Tails"}, 10]
{Heads, Tails, Heads, Heads, Heads, Heads, Tails, Heads, Heads, Heads}
RandomChoice[{"Heads", "Tails"}, 10]
{Heads, Heads, Heads, Tails, Heads, Heads, Heads, Tails, Heads, Heads}
RandomChoice[ {.9, .1} → {"Heads", "Tails"}, 20]
{Heads, Heads, Tails, Tails, Heads, Heads, Heads, Heads, Heads, Heads,
Heads, Heads, Heads, Heads, Heads, Heads, Heads, Heads, Heads, Heads}
RandomChoice[{1, 2, 3, 4, 5, 6}, 20]
{3, 3, 4, 1, 2, 6, 3, 2, 6, 5, 2, 4, 6, 6, 5, 6, 5, 2, 3, 3}

```

*simulating experiments with RandomChoice*

Once you have a list one of the more common tasks to do will be to pick out individual elements or a range of elements. The easiest way to do this is to use “part” notation. Given a list *list*, *list*[[*k*]] represents the *k*<sup>th</sup> element in *list*. If *k* is negative then the element you pick is counted back from the end of the list instead of from the start. To get a range of elements use the notation *list*[[*k* ;; *m*]] to get a new list starting with the *k*<sup>th</sup> element of the list and ending with the *m*<sup>th</sup> element (*k* and *m* can be positive or negative). *list*[[All]] will give you all the elements of *list*; this may seem silly but the option of using All (instead of say *list*[[1;;-1]]) will be useful when we start working with matrices:

```

mylist = Range[0, 60, 3]
{0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60}
mylist[[2]]
3
mylist[[-2]]
57
mylist[[3 ;; 8]]
{6, 9, 12, 15, 18, 21}
mylist[[6 ;; -5]]
{15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48}
mylist[[All]]
{0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60}

```

*getting elements and ranges in list using the part notation*

The part notation actually works on a much wider range of objects than lists. If *plot* is a graph then typically *plot*[[1]] are the colors and tiny segments that make up the picture and *plot*[[2]] represents the options being used in *plot*. ( $x^2 + 3xy + z^6$ )[[3]] is  $z^6$ , ( $x^2 + 3xy + z^6$ )[[1]] is  $x^2$ , and so on. This goes beyond what we’ll need to use part notation for but if you are

interested in seeing how this works check the documentation for the command FullForm, which shows you how different expressions are viewed by Mathematica's internal programming.

In addition to getting parts of lists Mathematica has commands to perform many basic list operations:

**Length[*list*]:** Length tells you how long a list is. Length[{1,2,3,4}] would be 4 and Length[{1,{2,3},4}] would be 3 (the middle list counts as a single object). Length can be used on a list that is only computed and not stored - Length[ Divisors[ 10^6] ] would tell you how long the list Divisors[10^6] is (that is, how many positive factors a million has) without explicitly storing or showing you the factors.

**Join[ *list1*, *list2*, ...]:** Join directly connects the different lists in the order which they appear. No action is taken to sort the elements or remove duplications.

**Union[*list1*, *list2*, ...]:** Union creates the “union” of the lists - a single list of all of the elements from the combined lists sorted and with duplicates removed. Union[*list*] would sort and remove duplicates from a single list.

**DeleteDuplicates[*list*]:** DeleteDuplicates removes the duplicates from *list* without performing any sorting.

**Sort[*list*]:** Sort sorts the elements of *list* using the standard notion of less than or greater than. You can Sort the *list* using non-standard ideas of less than and greater than if you use the form Sort[*list*, *sorting function of two variables*]. This is one of the places where the “pure function notation” from section 2.7 can be quite useful. If *list* is a list of points, then Sort[*list*] would sort the points by their first coordinate (the standard notion of “less than/greater than” for points). Sort[*list*, (#1[[2]] < #2[[2]])&] would sort the points of *list* by their second coordinate instead (in pure function notation #1 represents a first variable and #2 a second variable).

**Intersection[*list1*, *list2*, ...]:** Intersection finds the “intersection” of all of the lists - a single list of elements common to all of the lists, sorted and with duplicates removed. So Intersection[ {1,2,3},{2,3,4},{2,3,9,10}] would be {2,3} and Intersection[ {1,2,3}, {4,5,6}] would be {}. Intersection[ Divisors[1000], Divisors[364] ] would give you a list of the common divisors of 1000 and 364.

**Delete[*list*, *n*]:** Delete creates a new list by deleting the element at position *n* from list (*n* can be positive or negative). If you use the form Delete[*list*, { {*position 1*}, {*position 2*}, ...}] the new list can have multiple elements deleted. If *list* is a sorted list of numbers then Delete[ *list*, Table[ {*i*}, {*i*, 1, *k*} ] ] would make a new list by deleting the first (and therefore lowest) *k* elements from *list* (this comes up a lot in practice - dropping low homework grades, for example).

MemberQ[*list*, *element*]: MemberQ returns True if the *element* is in the list and False if not. So MemberQ[ {1,2,3,4}, 2] is True and MemberQ[ {1,2,3,4},5] is False.

MemberQ does not look deeper into lists of lists so MemberQ[ {1,{2,3},4},2] would be False (MemberQ only sees {2,3} as a single object distinct from the 2 inside).

Count[*list*, *element*]: Count tells you how many times the *element* appears in the list. Like MemberQ the Count command does not look deeper into sublists.

Tally[*list*]: Tally provides a count for how many times each element occurs in the *list*. So Tally[ {"Heads", "Heads", "Tails", "Tails", "Tails"}] would return {{ "Heads", 2}, {"Tails", 3}}. The order in the list of counts will be the order in which the elements in the *list* occur (so they may not be sorted in the way you might expect - Tally[ {3,3,2,2,2,2}] would give {{3,2}, {2,4}}). You can "fix" this by using the Sort command (so Sort[ Tally[ {3,3,2,2,2,2}] ] would return {{2,4},{3,2}}). A more advanced use of Tally involves altering the idea of what it means for two elements to be "the same" for counting. Tally[ *list*, *function of two variables that gives True or False*] would do the tally and count two elements as "the same" if the *function* gives True for those two elements. For example Tally[*list*, Floor[ #1/ 10]== Floor[#2/10] &] would provide a tally of the elements of *list* but consider two elements the same if they round down to the same multiple of 10 (so 71 and 73 would count as "equal" for the tally).

Partition[*list*,*n*]: Partition splits up the *list* into sublists of length *n* (if *n* does not evenly go into the *list* length the partial sublist left over at the end will be dropped). So Partition[ {1,2,3,4,5,6},3] would return {{1,2,3},{4,5,6}} and Partition[ {1,2,3,4,5,6}, 4] would return {{1,2,3,4}} (dropping the last 2 elements).

Flatten[*list*]: Flatten removes all the list structure inside *list*, leaving one long list. So Flatten[ {{1,2,3},{4,5},{3,6}} ] would return {1,2,3,4,5,3,6}. Flatten[*list*, *n*] only removes *n* levels of the list structure - So Flatten[ { {{1,2,3},{4,5,6}},{{7,8,9},{10,11,12}}, {3}}, 1] returns {{1,2,3},{4,5,6},{7,8,9},{10,11,12},3} (having removed 1 level of list structure from each part of the original list).

Split[*list*]: Split takes the list and breaks it into a list of "runs" where each run consists of identical elements. So Split[{1,1,2,3,3,3,1}] would return {{1,1},{2},{3,3,3},{1}}. As in Tally you can use a different notion of "identical" using the form Split[ *list*, *function of two variables that gives True or False*]. So if you wanted to consider the elements 0-4 the "same", 5-9 the "same", and so on you could use Split[*list*, Floor[ #1/5]== Floor[#2/5]&]. So Split[ {1,2,3,6,7,11,12,13}, Floor[ #1/5]== Floor[#2/5]&] would return {{1,2,3},{6,7},{11,12,13}}.

**Select**[ *list*, *function of 1 variable which is True or False*]: **Select** creates a new list consisting of all the elements of *list* for which the *function* is True. So **Select**[ *list*, **IntegerQ**] would create a new list consisting of those *list* elements which are integers. **Select**[ *list*, **Mod**[*#1*,3]==1 &] would return a new list of just those numbers whose remainder is 1 when you divide by 3. **Select**[ *list*, **70**≤*#1*<**80** &] would make a new list of numbers whose values are at least 70 but less than 80 (the traditional “C” range). **Select**[ *list*, **PolynomialQ**[*#1*,*x*] && **Exponent**[*#1*,*x*]≤**2** &] would create a new list all of whose elements were polynomials in *x* whose degree is 2 or less.

**GatherBy**[*list*, *function of 1 variable*]: **GatherBy** splits up the *list* into sublists where the elements of each sublist give the same value in the *function*. So **GatherBy**[ **Range**[10], **Mod**[*#1*,3]&] would split up the list {1,2,...,10} into sublists each of which have the same remainder when you divide by 3 ({1,4,7,10},{2,5,8},{3,6,9}).

**Map**[ *function*, *list*]: **Map** creates a new list by applying the *function* to each element of the *list*. For many simple built-in functions (like **Sin** or **Abs**) you can do this just by using the list as the input for the function (as in **Abs**[{-1,0,3}]) but **Map** lets you do this for functions you define and for pure functions. So a quick way to make a list of points for the graph of  $y = x^3 - x$  from -10 to 10 would be **Map**[ {*#1*,*#1*<sup>3</sup>-*#1*} &, **Range**[-10,10] ].

Let’s take a look at some applications of these functions:

#### Example 1: Finding common divisors

Let’s find the common divisors of 3072 and 2814. The complete list of divisors for each number are **Divisors**[3072] and **Divisors**[2814]. **Intersection** finds the common elements of two lists, so the answer would be **Intersection**[ **Divisors**[3072], **Divisors**[2814] ]:

<b>Intersection</b> [ <b>Divisors</b> [ 3072 ], <b>Divisors</b> [2814] ] {1, 2, 3, 6}
--

*computing common divisors much more quickly than by hand*

We could easily generalize this to make a **CommonDivisors** function - simply use **CommonDivisors**[*a*\_, *b*\_]:= **Intersection**[ **Divisors**[*a*], **Divisors**[*b*] ].

#### Example 2: Finding the largest prime divisor

What is the largest prime divisor of the number  $3^{100}-5$ ? This number is far too large to do the computation on paper. We can take advantage of the command **FactorInteger** to do this for us. We could just evaluate **FactorInteger**[ $3^{100}-5$ ] and visually inspect the result - but it’s

conceivable for large numbers this could be many pages of output to look through. For a positive integer the result of FactorInteger looks like  $\{\{prime\ 1, exponent\ 1\}, \{prime\ 2, exponent\ 2\}, \dots\}$  (if the number was negative the first sublist would be  $\{-1, 1\}$ ). So what we want is the first number in the last sublist. We can pull this number out by evaluating `FactorInteger[3^100-5][[-1,1]]` - this has the advantage of not needing us to see the full factorization:

```
FactorInteger[3^100 - 5][[-1, 1]]
99 748 323 925 835 341

FactorInteger[3^100 - 5]
{{2, 2}, {13 159, 1}, {4 666 979, 1},
 {1 793 751 961, 1}, {11 725 699 259, 1}, {99 748 323 925 835 341, 1}}
```

*getting the largest prime divisor of a large number*

One advantage of using the part notation is that it makes it easy to write a function to do this in general - `LargestPrimeDivisor[n_]:=FactorInteger[n][[-1,1]]`.

### Example 3: Longest run when flipping a coin

A common problem in basic probability is to find the longest run of heads in a sequence of flips. We can easily simulate this in Mathematica for much longer sequences than you would want to track by hand (say 1000 flips). We can set this up using the list commands like this:

- 1) Let the variable `sequencelength=1000`;
- 2) Define the variable `flipsequence=RandomChoice[{"Heads", "Tails"}, sequencelength];` (you definitely want the semi-colon at the end so you don't have to see the sequence).
- 3) Define the variable `runs=Split[ flipsequence];` (again the semi-colon is a good idea). This will separate the sequence out into runs.
- 4) Define the variable `runlengths=Map[ Length, runs];` This will put the Length command on each of the sublists in "runs", turning each sublist into the number of flips in the run.
- 5) Evaluate the command `tally=Sort[ Tally[runlengths] ]` to see how many runs of each length you get. If you really only want the maximum length you could just evaluate `Sort[ Tally[runlengths] ][[-1,1]]` instead. The Sort is necessary in `Sort[ Tally[runlengths] ][[-1,1]]` to make sure that the longest run appears last in the tally list (remember the order in Tally is the order in which the elements appear in the list).

```

sequencelength = 1000;
flipsequence = RandomChoice[{"Heads", "Tails"}, sequencelength];
runs = Split[flipsequence];
runlengths = Map[Length, runs];
Sort[Tally[runlengths]]

{{1, 249}, {2, 130}, {3, 55}, {4, 42}, {5, 13}, {6, 6}, {7, 2}, {8, 3}, {9, 1}, {10, 1}}

```

*the longest run was 10 in a row, and it only happened once in 1000 flips*

You could of course shorten this a bit - for a single example you don't need to really store 1000 in "sequencelength", you could just use the 1000 value in RandomChoice. But if you are to repeatedly experiment doing it this way would make it easier to change the 1000 flips to 2000 or 5000; just change the value in sequencelength and reevaluate the cells.

In this example the longest run was 10. If you reran the commands you might get unluckier (say 9) or luckier (maybe 12). Although it looks messy it is possible to have Mathematica rerun the experiment for you 50 times by using Table and nesting the commands inside each other instead of giving each its own line:

```

sequencelength = 1000;
Table[ Sort[ Tally[ Map[Length,
    Split[ RandomChoice[{"Heads", "Tails"}, sequencelength] ] ] ] ][[-1, 1]], {k, 1, 50}]
{8, 10, 11, 11, 7, 9, 9, 9, 10, 11, 10, 13, 8, 10, 10, 13, 20, 8, 12, 9, 9, 10, 12, 12, 9, 7,
 8, 10, 11, 10, 11, 9, 9, 12, 11, 12, 10, 10, 9, 11, 9, 11, 10, 11, 11, 10, 9, 12, 11, 12}

```

*repeating the experiment 50 times*

The command is pretty messy and has a lot of square brackets. When building a complex command like this yourself it's easiest to start with the inside command (that you calculate first) and then go back and "wrap" each successive command around it (remembering the square brackets at the beginning and end). So you would enter the RandomChoice command first, then wrap Split around it, then the "Map[ Length, ]" around that, and so on. It takes practice to do something that complicated quickly but it's a lot faster than flipping a coin 50,000 times yourself.

#### Example 4: Counting grades

Suppose you had a list of 1000 grades (out of a possible 100). Anything 90 and up gets an A, 80 to 89 a B, 70 to 79 a C, 65 to 69 a D, and anything 64 and under gets an F. How could you quickly count how many of each grade there were?

There are many different ways you could go about this, several of which will need you to define a function to assign grades to a given score. As the grades are given by several ranges

Which is a natural command to use: `Grader[n_]:= Which[ n<65, "F", n<70, "D", n<80, "C", n<90, "B", True, "A"]` (this takes advantage of the way Which works - if you don't like the overlapping cases you could certainly have the "D" criterion be  $65 \leq n < 70$ , etc.).

Once you have the Grader function defined you could use GatherBy to group the scores together into distinct clumps and then measure the Length of each clump:

```
randomgrades = RandomChoice[ Range[40, 100], 1000];
Grader[n_] := Which[ n < 65, "F", n < 70, "D", n < 80, "C", n < 90, "B", True, "A"]
gradeclumps = GatherBy[ randomgrades, Grader];
Map[ {Grader[ #1[[1]] ], Length[ #1]} &, gradeclumps]
{{B, 175}, {F, 414}, {A, 171}, {C, 161}, {D, 79}}
```

*using GatherBy to count grades*

The function inside the Map command serves two purposes - it counts the length of each grade clump (that's the second element of the list) but also labels each clump (as `Grader[ #1[[1]] ]` is the letter grade of the first score in the clump). Just using the Length command would have given us the list {175,414,171,161,79} - but we wouldn't have known which number was for which grade.

A simpler method for this particular problem would be to apply the Grader function to every element of the grade list and then Tally it:

```
lettergrades = Map[ Grader, randomgrades];
Tally[ lettergrades]
{{B, 175}, {F, 414}, {A, 171}, {C, 161}, {D, 79}}
```

*a simpler method counting grades*

If the second method is so much simpler why bother with the GatherBy approach? The Tally works well for the specific problem of counting grades from a long list. But in practice a teacher might have 30 grades in their grade book along with the last and first names of the students and need to both assign the letter grades and then give a list of the A students, B students, and so on. It's not hard to modify the GatherBy approach to do this but much harder to do it with the Tally approach. To see how this might work we will need to generate an example of a grade book without doing a whole lot of typing. Evaluate the following command exactly as you see it below - don't worry about the particulars (this will require an internet connection and you will see something about WordData initializing). It will create a list of 3-element lists, the first 2 elements of which are random words drawn from a dictionary (some of the dictionary selections will likely be compound phrases but it's the best we can do):



```

gradebook = Table[
  Join[ RandomSample[WordData[All], 2], RandomChoice[ Range[40, 100], 1 ] , {k, 1, 30}]
  {{genus Matthiola, Cuban spinach, 52}, {genus Polanisia, Momotus, 48},
   {jackscrew, acoustician, 60}, {play therapy, doorjamb, 89},
   {continental quilt, garbage dump, 69}, {audiometric, Tractarian, 48},
   {anisogamete, Dioscorea bulbifera, 93}, {East Indian rosewood, bedaubed, 46},
   {counselling, geophagy, 41}, {alder buckthorn, calling, 44},
   {geodesical, short temper, 59}, {integrate, Waikiki, 70},
   {Edward James Muggeridge, eggs Benedict, 83}, {sonar, fete day, 72},
   {monogenesis, partial breach, 57}, {anemometric, essentiality, 55},
   {gas constant, order Lobata, 78}, {copalite, Marlowe, 86},
   {Megachile, chamber music, 79}, {stater, maple-like, 77},
   {George Gershwin, family Asparagaceae, 41}, {social phobia, buy into, 40},
   {Greek architecture, trumpet, 81}, {clink, atomic number 52, 86},
   {transversus abdominis, throw-in, 43}, {acquitted, mandibular fossa, 84},
   {acne rosacea, activity, 55}, {SCSI, vena metatarsus, 46},
   {honour, financial organisation, 94}, {William Green, near beer, 74}}

```

*generating a random grade book*

We can (loosely) interpret this as a set of entries {first name, last name, grade} from a grade book. We can group the entries by grade in two steps - first we can use Map to create a new copy of the grade book with the letter grades in place, and then group that by the last element:

```

lettergrades2 = Map[ {#1[[1]], #1[[2]], Grader[#1[[3]] ] } &, gradebook]

{{genus Matthiola, Cuban spinach, F}, {genus Polanisia, Momotus, F},
 {jackscrew, acoustician, F}, {play therapy, doorjamb, B},
 {continental quilt, garbage dump, D}, {audiometric, Tractarian, F},
 {anisogamete, Dioscorea bulbifera, A}, {East Indian rosewood, bedaubed, F},
 {counselling, geophagy, F}, {alder buckthorn, calling, F}, {geodesical, short temper, F},
 {integrate, Waikiki, C}, {Edward James Muggeridge, eggs Benedict, B},
 {sonar, fete day, C}, {monogenesis, partial breach, F}, {anemometric, essentiality, F},
 {gas constant, order Lobata, C}, {copalite, Marlowe, B}, {Megachile, chamber music, C},
 {stater, maple-like, C}, {George Gershwin, family Asparagaceae, F},
 {social phobia, buy into, F}, {Greek architecture, trumpet, B},
 {clink, atomic number 52, B}, {transversus abdominis, throw-in, F},
 {acquitted, mandibular fossa, B}, {acne rosacea, activity, F}, {SCSI, vena metatarsus, F},
 {honour, financial organisation, A}, {William Green, near beer, C}}

GatherBy[ lettergrades2, #1[[3]] &]

{{{genus Matthiola, Cuban spinach, F},
 {genus Polanisia, Momotus, F}, {jackscrew, acoustician, F},
 {audiometric, Tractarian, F}, {East Indian rosewood, bedaubed, F},
 {counselling, geophagy, F}, {alder buckthorn, calling, F},
 {geodesical, short temper, F}, {monogenesis, partial breach, F},
 {anemometric, essentiality, F}, {George Gershwin, family Asparagaceae, F},
 {social phobia, buy into, F}, {transversus abdominis, throw-in, F},
 {acne rosacea, activity, F}, {SCSI, vena metatarsus, F}},
 {{play therapy, doorjamb, B}, {Edward James Muggeridge, eggs Benedict, B},
 {copalite, Marlowe, B}, {Greek architecture, trumpet, B}, {clink, atomic number 52, B},
 {acquitted, mandibular fossa, B}}, {{continental quilt, garbage dump, D}},
 {{anisogamete, Dioscorea bulbifera, A}, {honour, financial organisation, A}},
 {{integrate, Waikiki, C}, {sonar, fete day, C}, {gas constant, order Lobata, C},
 {Megachile, chamber music, C}, {stater, maple-like, C}, {William Green, near beer, C}}}

```

*grouping together the very oddly-named class by grades*

### Example 5: Making a math problem with nice numbers in the solution

Suppose you were an algebra teacher and you wanted to give your students a quadratic to solve on an exam - but you wanted the answers to be rational (so no square roots would be needed). To make it difficult for them to factor you decide the equation should have the form  $2x^2 + 12x + k = 0$ , where  $k$  is a positive whole number. What values of  $k$  would give the equation rational solutions?

The brute force way to do this would be to try a bunch of values for  $k$ , try to solve the corresponding quadratic, and check if both roots are rational. This would take far too long to do by hand but we can easily automate it in Mathematica. We can generate a Table of lists of the form  $\{k, \{\text{solutions to } 2x^2 + 12x + k = 0\}\}$  and then use Select to only pick out the parts where the solution list is made up of rational numbers. Then we could take those answers and form the quadratics from their first part:

```
bigtable = Table[ {k, x /. Solve[ 2 x^2 + 12 x + k == 0, x] }, {k, 1, 1000}];
niceones = Select[bigtable, Element[#1[[2]], Rationals] &]
{ {10, {-5, -1}}, {16, {-4, -2}}, {18, {-3, -3}} }
Map[ 2 x^2 + 12 x + #1[[1]] == 0 &, niceones]
{ 10 + 12 x + 2 x^2 == 0, 16 + 12 x + 2 x^2 == 0, 18 + 12 x + 2 x^2 == 0 }
```

*surprisingly only 3 quadratics out of 1000 have nice answers*

A second approach would be to use the fact that a quadratic has rational roots if and only if its discriminant is a perfect square (in this case a perfect square integer):

```
niceones2 = Select[Range[1000], IntegerQ[ Sqrt[ Discriminant[ 2 x^2 + 12 x + #1, x] ] ] &]
{ 10, 16, 18 }
Map[ 2 x^2 + 12 x + #1 == 0 &, niceones2]
{ 10 + 12 x + 2 x^2 == 0, 16 + 12 x + 2 x^2 == 0, 18 + 12 x + 2 x^2 == 0 }
```

*a higher-tech way of approaching the same problem*

Most of these examples have used simple lists. The grade book example used something a little more complicated - a list of lists, all of which had 3 elements. This is an example of something called a matrix. Matrices have many uses (a large part of the subject linear algebra is based on them, but even simple things like addition and multiplication tables are essentially matrices). Matrices can be created in three main ways in Mathematica.

The first way is by hand using the  $\{\}$  notation for lists - you just need to be careful to make sure each sublist has the same length. So the short lists  $\{\{1,2\},\{3,4\},\{7,8\},\{10,11\}\}$  and  $\{\{1,2,3,4\},\{5,6,7,8\}\}$  are both matrices that are small enough to enter manually. If these look

like lists of points or data values that's no coincidence - data sets are often represented by matrices.

A second way is to enter the matrix manually in a grid. If you go the **Insert** menu one of the options will be the submenu "Table/Matrix" from which you can pick "New". This will bring up a popup dialog where you can enter the numbers of rows and columns of the matrix as well as a few options for how the matrix will look. On hitting "OK" at the insertion point you will get a matrix of the size you wanted filled with empty squares for the entries. Click on a square and you can enter its value (be careful not click between the squares). You can also use the tab key to move from square to square. When a particular entry in the matrix is selected going to the **Insert** and "Table/Matrix" menus will allow you to add a new row or column as well. This approach is a great way to enter statistical data as a matrix. If each data point is just an x value and y value and you need to enter in 30 points you can just create a 2-by-30 matrix and enter your numbers. If you need to add 5 more data points at a later time you can use the "add new row" command from the menus to enter in the new data point by point.

The third way to create a matrix is to use a 2-dimensional version of the Table command. `Table[formula, {variable 1, start 1, finish 1}, {variable 2, start 2, finish 2}]` creates a matrix using *formula*, where *variable 1* is constant across any "row"/sublist and *variable 2* is constant across any column/sublist part (you can also add a step size to each of the variable lists if you need to).

```
Table[ i + 100 j, {i, 1, 10}, {j, 1, 5}]
{{101, 201, 301, 401, 501}, {102, 202, 302, 402, 502},
 {103, 203, 303, 403, 503}, {104, 204, 304, 404, 504},
 {105, 205, 305, 405, 505}, {106, 206, 306, 406, 506}, {107, 207, 307, 407, 507},
 {108, 208, 308, 408, 508}, {109, 209, 309, 409, 509}, {110, 210, 310, 410, 510}}
```

*making a matrix from a formula with Table*

In this case the first sublist corresponds to  $i=1$ , the second to  $i=2$ , and so on.  $j=1$  corresponds to the first element in each sublist,  $j=2$  corresponds to the second element in each sublist, and so on. The order of the variables can make a big difference:

```
Table[ i + 100 j, {j, 1, 5}, {i, 1, 10}]
{{101, 102, 103, 104, 105, 106, 107, 108, 109, 110},
 {201, 202, 203, 204, 205, 206, 207, 208, 209, 210},
 {301, 302, 303, 304, 305, 306, 307, 308, 309, 310},
 {401, 402, 403, 404, 405, 406, 407, 408, 409, 410},
 {501, 502, 503, 504, 505, 506, 507, 508, 509, 510}}
```

*the variable order makes a big difference in Table*

Even the sizes are different here - the first matrix is "10 by 5" and the second is "5 by 10".

We can more easily visualize these matrices by using `MatrixForm` and `TableForm`. `MatrixForm[matrix]` makes a printed copy of *matrix* with the customary bracketing:

```
In[39]:= MatrixForm[ {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {x, y, z}} ]
Out[39]//MatrixForm=
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ x & y & z \end{pmatrix}$$

*using MatrixForm to visually represent an array in “standard” form*

Note the change to the “output number” - it is now followed by `MatrixForm`. This indicates that while the output appears to be something new (a printed version of the matrix), for all intents and purposes the output is still the plain matrix (so you can use it in functions like `Join` or `Tally`). The same goes for `TableForm[matrix]`, which just prints the array as a grid without the bounding brackets:

```
In[50]:= TableForm[ {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {x, y, z}} ]
Out[50]//TableForm=
```

1	2	3
4	5	6
7	8	9
x	y	z

*using TableForm to represent a matrix as a grid*

`MatrixForm` and `TableForm` have an option called `TableHeadings` which can be used to place labels on the rows and columns - the form is `TableHeadings→{list of row headers, list of column headers}` (if you don’t want any headers on just rows or columns you can use `None` in place of that particular header list. This can be very useful when your matrix carries some meaning to it. Here is how you might use `TableHeadings` to represent grades:

```
In[52]:= TableForm[ {{89, 97, 70, 82}, {70, 73, 74, 77}, {50, 70, 90, 100}},
TableHeadings → {{ "Student 1", "Student 2", "Student 3"},
{"Exam 1", "Exam 2", "Exam 3", "Final Exam"}} ]
Out[52]//TableForm=
```

	Exam 1	Exam 2	Exam 3	Final Exam
Student 1	89	97	70	82
Student 2	70	73	74	77
Student 3	50	70	90	100

*using TableHeaders to make grades readable*

Or you could use `TableHeadings` to make a table of points for the formula  $y=x^2$ :

```
In[53]:= TableForm[ Table[ {x, x^2}, {x, -4, 4}], TableHeadings -> {None, {x, y}}]
```

```
Out[53]//TableForm=
```

x	y
-4	16
-3	9
-2	4
-1	1
0	0
1	1
2	4
3	9
4	16

*points on the graph of a parabola in the usual setup*

Or even a brief multiplication table:

```
In[54]:= TableForm[ Table[ i j, {i, 1, 7}, {j, 1, 7}], TableHeadings -> {Range[7], Range[7]}]
```

```
Out[54]//TableForm=
```

	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	2	4	6	8	10	12	14
3	3	6	9	12	15	18	21
4	4	8	12	16	20	24	28
5	5	10	15	20	25	30	35
6	6	12	18	24	30	36	42
7	7	14	21	28	35	42	49

*the “seven times table”*

The part notation applies to tables and matrices just as it does to lists. If *matrix* is a matrix then *matrix*[[*k*,*m*]] represents the element of the matrix in row *k* and column *m*. *matrix*[[*k*]] represents row *k* of the matrix (a matrix is just a list of lists after all). You can also pull out a given column of the matrix as *matrix*[[All,*m*]] (in other words take elements from all rows in column *m*) - when used this way the columns are represented as normal lists (and not as a matrix).

We can use matrix notation to handle a much larger version of a problem from Section 2.7: What is the largest value of  $\frac{x}{2} + \frac{y}{3}$  where *x* and *y* are non-negative integers for which  $150 \leq x + y \leq 400$ ,  $x + \frac{y}{2} \leq 750$ , and  $\frac{x}{3} + y \leq 700$ ? The way we handled this in a smaller problem back in Section 2.7 involved 3 simple steps. First we used Reduce to find all of the points (*x*,*y*) which fit the constraints (which was a fairly short list). Then we had Mathematica compute the values of  $\frac{x}{2} + \frac{y}{3}$  for each one of those points. Finally we just visually picked out the largest value from that list. Technically we could do that for this problem too - except there are 69,276 valid points (*x*,*y*) which satisfy all the constraints. We can use matrix notation to avoid having to check all those points ourselves using the following steps:

- 1) Find all of the points  $(x,y)$  which satisfy the constraints using ToRules and Reduce, storing them as replacement rules in the variable “sols”. You definitely want to use a semi-colon for this as you don’t want to see the output.
- 2) Have Mathematica compute the value for  $\frac{x}{2} + \frac{y}{3}$  for each point, creating a matrix “values” whose rows will have the format  $\{x, y, \frac{x}{2} + \frac{y}{3}\}$ . Again use a semi-colon to suppress the output.
- 3) Instead of trying to pick the largest value  $\frac{x}{2} + \frac{y}{3}$  at the end of each row ourselves use the Max command on the third column of the matrix “values” (as values[[All,3]]).
- 4) Use Select to pick out all the rows of “values” whose 3rd coordinate is equal to the maximum (in problems like this the maximum may occur at more than one point - using Select will get us all of the points where the maximum occurs).

```
sols = {ToRules[ Reduce[
  x ≥ 0 && y ≥ 0 && 150 ≤ x + y ≤ 400 && x + y / 2 ≤ 750 && x / 3 + y ≤ 700, {x, y}, Integers]]];
values = {x, y, x / 2 + y / 3} /. sols;
max = Max[values[[All, 3]] ]
200
Select[values, #1[[3]] == 200 &]
{{400, 0, 200}}
```

*finding the maximum value for a problem with 69,276 possible solutions*

So the maximum value of  $\frac{x}{2} + \frac{y}{3}$  is 200 and it occurs when  $x$  is 400 and  $y$  is 0. Using matrix notation to isolate the third column for Max and having Select pull out the right row is a lot faster than checking all 69,276 possibilities on our own!

In Section 3.8 we will look at how to solve optimization problems like this directly - as a preview here is how to have Mathematica solve the same problem for us:

```
Maximize[ {x / 2 + y / 3, x ≥ 0 && y ≥ 0 && 150 ≤ x + y ≤ 400 && x + y / 2 ≤ 750 && x / 3 + y ≤ 700},
{x, y}, Integers]
{200, {x → 400, y → 0}}
```

*a look ahead at the Maximize command*

## Section 3.2 Homework – The Basics of Lists

- 1) Use Table to make a list of the values of  $k^3$  as  $k$  goes from 3 to 20.
- 2) Use Table to make a list of points of the form  $\{x, x^2 - x\}$  as  $x$  goes from -4 to 4 by halves. Use TableForm and TableHeadings to make your list appear as a table of points for graphing.
- 3) Repeat problem 3, except instead of using Table to generate the list of points use Map, a pure function, and Range instead.
- 4) Use Table to create the list  $\{1000, 997, 994, 991, \dots, 1\}$ .
- 5) Graph  $y = 3 \sin(2x - \frac{\pi}{2}) + 1$  from  $-2\pi$  to  $2\pi$ , using Range to generate tick marks every  $\pi/4$  on the  $x$ -axis and at integer values on the  $y$ -axis.
- 6) How many divisors does 39,243,024,823,000,000 have? Use “part” notation to find its 3rd largest divisor.
- 7) Find the common divisors of 304,292,146 and 919,500,832. (hint: think Intersection)
- 8) The list of angles you typically first learn in trigonometry are the multiples of  $30^\circ$  and  $45^\circ$  that go from  $0^\circ$  to  $360^\circ$ . Create this list of angles using Union and Range and store it in the variable “anglelist” for problem 9.
- 9) Use Map and the variable anglelist from problem 8 to create a matrix whose rows have the form  $\{\text{angle}, \sin(\text{angle}), \cos(\text{angle}), \tan(\text{angle})\}$  for each angle in anglelist (don’t forget to use Degree). Then use TableForm and TableHeadings to convert this matrix into a table whose rows are not labelled but whose columns are labelled with “angle”, “sine”, “cosine”, and “tangent”.
- 10) Create a list “list1” whose elements are the lists  $\{k, k^2 - k, k^3 - 9k\}$  as  $k$  goes from -5 to 5. Create two new lists from this - one which is sorted on the second part of each list and another which is sorted on the third part of each list. Use TableForm on each list to make it easy to verify your sort is correct.
- 11) Use Select to create a list of the integers from 1 to 1000 whose GCD with 1000 is 1.
- 12) Generalize problem 11 to a function  $U[n]$  whose output is a list of those integers from 1 to  $n$  whose GCD with  $n$  is 1.
- 13) Create a list of 100 fair coin flips. How many times do you get “Heads” and “Tails”? What was the longest run of either “Heads” or “Tails”?
- 14) Repeat problem 14 for a coin which comes up “Heads” 20% of the time and “Tails” 80% of the time.
- 15) Use Table and RandomChoice to create a list of 100 pairs of the form  $\{\text{coin flip}, \text{die roll}\}$ , where the coin flip is either “Heads” or “Tails” and the die roll is 1,2,3,4,5, or 6 (remember RandomChoice[set] chooses a single random element from the set). Store this list in the variable “randomset” for problems 16-18.
- 16) Use Select to get just those elements of randomset whose first element is “Tails”. Use Select to get just those elements of randomset whose first element is “Heads”.
- 17) Use GatherBy and the pure function for “first part of #1” to recreate the results of problem 16.
- 18) Use Select to find those elements of randomset whose die roll is 1, 2, or 5.

- 19) Redo example 3 but only find the length of the longest run of “Heads” instead of just the longest run. (Hint: Insert a new step between step 3 and 4 that involves Select and MemberQ).
- 20) Use Table and TableHeadings to make the “mod 10 multiplication table” (that is a table whose rows and columns correspond to integers  $a$  and  $b$  and whose entries are the remainder when  $ab$  is divided by 10).
- 21) Generalize problem 20 to a function ModularMultiplication[n] to make a “mod  $n$  multiplication table”.
- 22) Create a function DropLowest4[ *list* ] which takes a *list* of grades (all numbers), sorts them, and then drops the lowest 4 values (returning the smaller list as the result). Test your function on a randomly generated list of 10 grades, each of which is from 60-100.
- 23) Look up the Tuples command and describe what it does in your own words. Use it to find a list of all possible rolls of 5 6-sided dice (store this in the variable “dicerolls”).
- 24) Use Map and a pure function to take the 5 rolls in each part of “dicerolls” and add them up (think of the pure function “first part + second part + third part + fourth part + fifth part”). Use Tally and Sort to see which sum or sums are most common.



## Section 3.3 - Basic Statistics

In addition to working with algebra and trigonometry Mathematica includes a wide range of tools for working with statistics and probability (many more than we can discuss here as we are not assuming you have had a full statistics course). Rather than try to give an extended list of all the tools available we will focus on the two main parts of statistics that don't require a lot of preliminary discussion and definitions - descriptive statistics and regression. Descriptive statistics are those calculations used to describe a data set (to keep things simple we will stick to one-variable data for descriptive statistics, like a set of 30 exam grades or a set of 20 temperature measurements). Descriptive statistics are oriented towards asking questions like "what is the center of the data measurements?" or "how widely is the data spread out?". Regression on the other hand tries to find formulas that fit the data very well and that can be used to make predictions. Regression works on 2 variable data (of the form  $(x,y)$ ), 3 variable data (of the form  $(x,y,z)$ ), and so on. The basic idea is that you think that one of the variables depends on all of the others ( $y$  depends on  $x$ , or  $z$  depends on  $x$  and  $y$ ) and you want to find the best formula you can for that dependence.

Before you can work with either descriptive statistics or regression you need to get your data into Mathematica. This is done through the use of lists (for 1 variable data like exam scores) or lists of lists/matrices (for data which is paired measurements like (time of day, temperature) or (time of day, humidity, temperature)). For example suppose you gave a quiz to 6 students and their scores were 9,5,7,5,6,8 (out of 10). You could represent this 1 variable data in Mathematica by using `quizdata={9,5,7,5,6,8}`. If you had taken temperature measurements every hour and at 1 p.m. it was 89, at 2 p.m. it was 92, at 3 p.m. it was 98, and at 4 p.m. it was 96, you could represent these paired measurements as `tempdata= { {1,89}, {2, 92}, {3,98}, {4,96} }` (you could also enter this directly as a matrix with 4 rows and 2 columns). The nice thing about using lists and matrices for data is that you can use commands we already know like `Length` and `Join` on the data. It is also possible to import data from a Microsoft Excel file into Mathematica but we'll leave a discussion of how to do that for later in the chapter.

Once your data is in Mathematica if you have 1 variable data (like `quizdata` above, or 30 exam grades, etc.) you can easily get common descriptive statistics by using the following commands:

`Mean[data]`: `Mean` gives the average of the data set. If your data consists of exact numbers instead of approximate numbers the average will be an exact number as well.

`TrimmedMean[data, amount]`: `TrimmedMean` gives you the mean of the data after the percentage *amount* has been dropped from both the top and bottom of the sorted data. So `TrimmedMean[data, .12]` would give the mean of the data after dropping both the

lowest and highest 12% of the data. `TrimmedMean[data, {bottomamount, topamount}]` allows you to drop different percentages from the high and low ends.

`Median[data]`: Median gives you the median of the data set. The median is the middle value of the data when it has been sorted (if the data set has an even number of values the median is the average of the middle two numbers after sorting). One way of thinking of the median is it is the 50% mark in the sorted data.

`Commonest[data]`: Commonest give you the mode of the data (most commonly appearing number in the data) given in a list. If there is a tie for the most common number all the tied numbers are reported (which is why the result is always given in a list).

`Quartiles[data]`: Quartiles gives you a list {first quartile, median, third quartile}. The first quartile is the 25% mark in the data and the third quartile is the 75% mark in the data (after sorting).

`Quantile[data, fraction]`: Quantile generalizes the idea of quartiles and median by returning the value which is *fraction* of the way through the sorted data. So the first quartile can be obtained as `Quantile[data, 1/4]`, the median as `Quantile[data, 1/2]`, and the third quartile as `Quantile[data, 3/4]`. If you wanted the 93% mark in the data, you could use `Quantile[data, 93/100]`.

`Max[data]` and `Min[data]`: The largest and smallest data values, respectively.

`StandardDeviation[data]`: StandardDeviation returns the “sample standard deviation” of the data. Without going into the technical definition of a standard deviation it is a numerical measure of how the data is spread out around the mean - widely spread data sets have a large standard deviation and tightly clustered data sets have small standard deviations.

`ListPlot[data]`: This creates a graph of the 1 variable *data*, where the first data value is assigned the x-coordinate 1, the second gets x-coordinate 2, and so on. So `ListPlot[{3,4,1,2}]` would create a plot of the points (1,3), (2,4), (3,1), and (4,2). `ListPlot[data, Joined → True]` joins the points in the plot, creating what is usually called a line graph of the data. ListPlot also works directly on 2 variable data.

On the next page and a half are examples of using these descriptive statistics on 2 randomly generated sets of numbers. The first data set has an average of 5.2 and is fairly spread out around that value, whereas the second data set has a similar average but is much more tightly packed around it. This leads to the first data set having a larger standard deviation than the smaller one does. Also note that ListPlot accepts many of the same options as Plot, including AspectRatio and PlotRange:

```

data1 = Sort[ Table[ Random[Integer, {1, 10}], {i, 1, 15}]]
{1, 2, 2, 3, 3, 3, 5, 6, 6, 6, 6, 8, 8, 9, 10}

data2 = Sort[ Table[ Random[Integer, {4, 7}], {i, 1, 15}]]
{4, 4, 4, 4, 4, 5, 5, 5, 5, 6, 6, 6, 7, 7, 7}

N[Mean[data1]]
N[Mean[data2]]
5.2
5.26667

N[TrimmedMean[data1, .20]]
N[TrimmedMean[data1, {.2, 0}]]
5.11111
6.08333

Median[data1]
Median[data2]
6
5

Commonest[data1]
Commonest[data2]
{6}
{4}

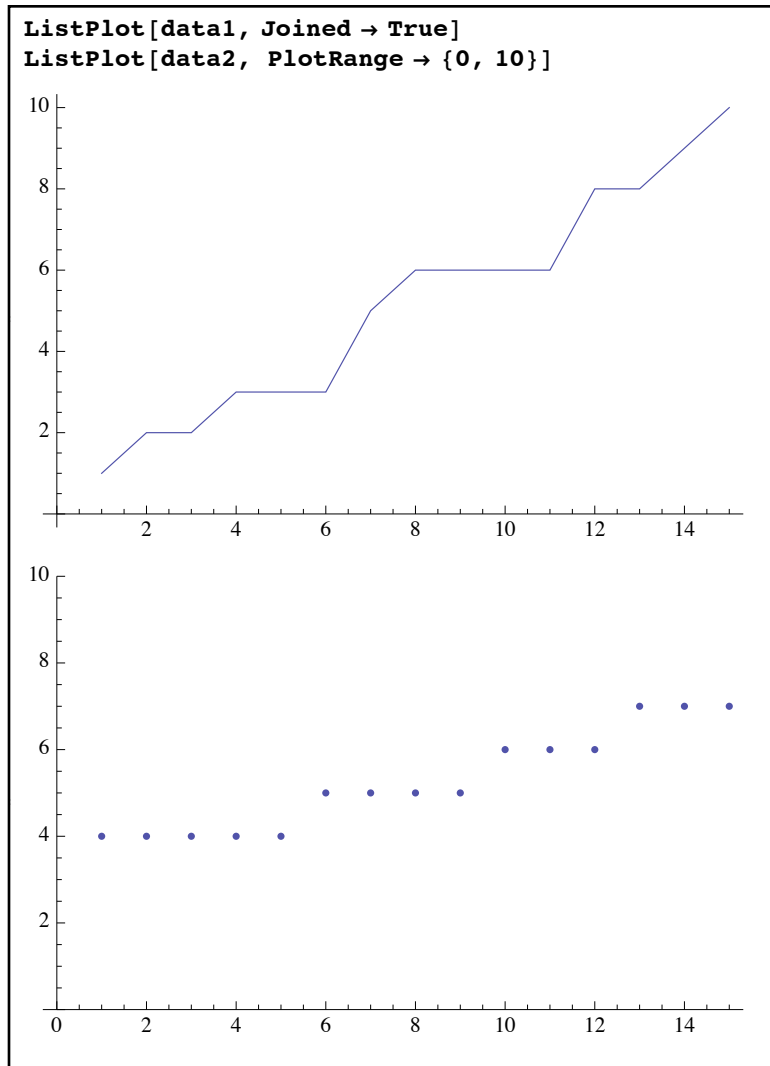
Max[data1]
Min[data2]
10
4

Quartiles[data1]
Quartiles[data2]
 $\left\{3, 6, \frac{15}{2}\right\}$ 
{4, 5, 6}

Quantile[data1, 35 / 100]
Quantile[data2, 35 / 100]
3
5

N[StandardDeviation[data1]]
N[StandardDeviation[data2]]
2.7826
1.16292

```



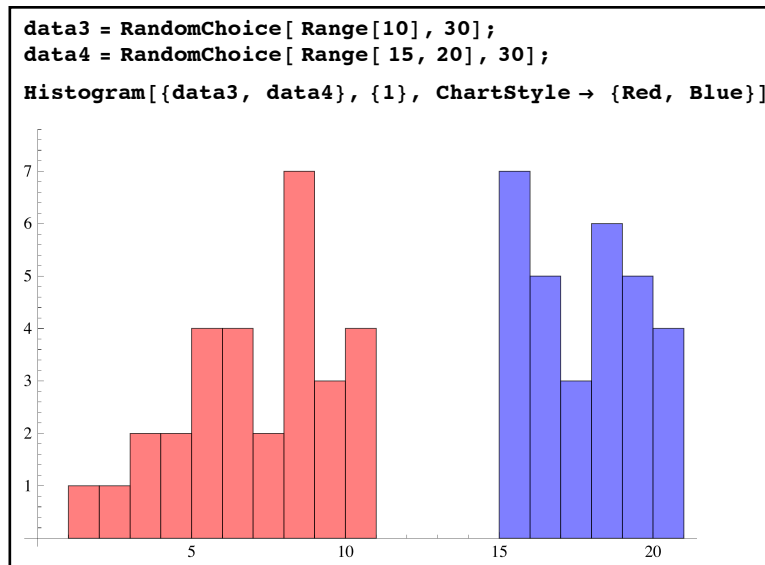
*Descriptive statistics and plotting 1 variable data*

In some applications your data with consist of the same numbers over and over again - imagine a survey question where the responses can only be the numbers 1-5. If you had several hundred responses it would be pretty inefficient to have to type those numbers over and over again - it would much easier to describe the data in the format “150 1’s, 200 2’s, ....”.

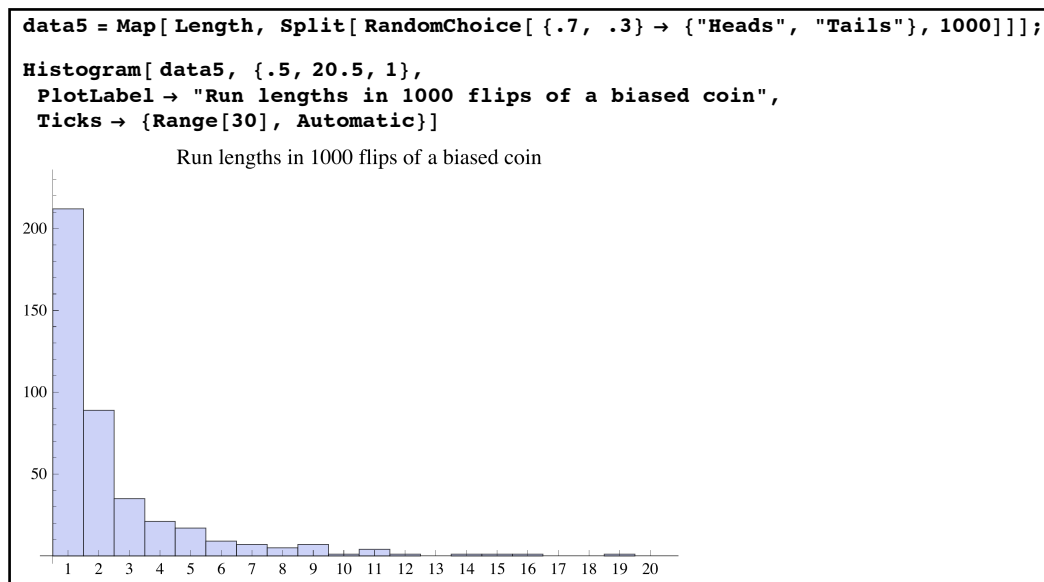
Mathematica has a command for that - `WeightedData`. `WeightedData[ list of values, list of weights ]` represents a data set where each value from the first list is given the appropriate weight from the second list. So if your survey responses consists of “150 1’s, 200 2’s, and 30 3’s”, you could enter the data as `data=WeightedData[ {1,2,3}, {150,200,30} ]`. Once created the `WeightedData` object can be used with `Mean`, `Median`, and other statistical functions (not including `ListPlot`).

Another common visualization for one-dimensional data (and one that works with `WeightedData`) is the histogram, which counts the number of data points in various ranges and represents the counts as the height of rectangles. Histograms are created in Mathematica by the

surprisingly-named Histogram command. The format for the command is `Histogram[data list, bin specification]`, where *bin specification* is a way of telling Mathematica how you want the ranges for each rectangle to be set up. The specification is usually either a number  $n$  (the number of rectangles), a 1-number list  $\{w\}$  (the width of each rectangle), or a range similar to the kind you use for Plot and Table ( $\{start, end, step\}$  where the step size is the width for each rectangle). You can graph more than one histogram at a time by replacing the data list with a list of data lists as well (although you should use the option `ChartStyle` instead of `PlotStyle` if you want to set colors):



*histograms for two data sets - the rectangles have width 1 but no mandates for start and end points*



*a histogram representing run lengths for a biased coin - note that the specification  $\{.5, 20.5, 1\}$  forces the rectangles to be centered over whole numbers*

All of the descriptive statistics above apply only to 1 variable data - basically lists of single numbers. Just as useful is the notion of regression - finding the formula which is the closest possible fit to a data set with more than one variable (the most common in elementary statistics is working with 2 variable data, which you can visualize as points in the plane - so regression becomes finding the best fitting curve to the points). There are two main types of regression - linear and nonlinear.

In linear regression you have a particular set of functions you would like to use to fit your data and you are only allowed to add, subtract, and scale the functions. For example, a linear regression using the terms  $1$ ,  $x^2$ , and  $x^3$  to a set of data might result in a “best fit” of  $4.3 + 4.7x^2 - 5.1x^3$  (only using those 3 terms, addition, subtraction, and constant multiples). If you do a linear regression using  $1$ ,  $x^2$ ,  $x^3$ , and  $\sin(x)$ , the “best fit” curve might be  $3.7 + 3.8x^2 - 4.2x^3 + 2.0 \sin(x)$ . You’ll notice that phrase “best fit” is in quotes - what you find in linear regression is the best fit using your particular functions and no others. You would expect the second fit to be better than the first because it uses all the available terms from the first and  $\sin(x)$  as well. In general you will get better fits as you add more terms to your regression - but you pay for that by having a more complicated formula. Statisticians have developed a measure for how good a linear regression is which factors in how good the fit is and penalizes it for complexity introduced by adding in extra terms. This measure is called the “Adjusted  $R^2$ ” value of the fit and is a number which is typically from 0 to 1, with 0 (or lower) being a horrible fit and 1 being a perfect fit. So you can do two different linear regressions and the one with the higher adjusted  $R^2$  value is in general the one you should use.

The command for linear regression to a set of points *data* is `LinearModelFit`, which uses the format `LinearModelFit[ data, termslist, variable]` (this command is set up for sets of points in the plane - if you replace *variable* with *variablelist* you can work with data with 3 or more coordinates). So if you have a set of data called *mydata* and you would like to find the best fitting line to the data (i.e. using only the terms  $1$  and  $x$ ), you would use the command like `myline= LinearModelFit[mydata,{1,x},x]`. If you wanted to find the best fitting parabola (i.e. using only the terms  $1$ ,  $x$ , and  $x^2$ ), you would use something like `LinearModelFit[ mydata, {1,x,x^2}, x]`.

In both the examples of the notation below you’ll notice that we didn’t just use `LinearModelFit`, but instead stored the fit itself inside a name like *myline* or *myparabola*. This is fairly standard practice because of the way Mathematica returns the best fit:

```

mydata = Table[{k, Sin[k]}, {k, 0, 3, .2}]

{{0., 0.}, {0.2, 0.198669}, {0.4, 0.389418}, {0.6, 0.564642}, {0.8, 0.717356}, {1., 0.841471},
{1.2, 0.932039}, {1.4, 0.98545}, {1.6, 0.999574}, {1.8, 0.973848}, {2., 0.909297},
{2.2, 0.808496}, {2.4, 0.675463}, {2.6, 0.515501}, {2.8, 0.334988}, {3., 0.14112}}

bestline = LinearModelFit[mydata, {1, x}, x]

FittedModel[0.542702 + <<20>> x]

bestquadratic = LinearModelFit[mydata, {1, x, x^2}, x]

FittedModel[-0.0384337 + <<19>> x - <<20>> x^2]

```

*LinearModelFit on a set of points from the sine curve*

The results from the command don't look like anything we've seen before. FittedModel represents a function that you can plug numbers directly into, say as bestline[2.1] or bestquadratic[4.5]. To see the best fit in more standard notation put the name you've stored it in followed by [x] (bestline[x], for example - you can use this form in a Plot command too). You can even get the adjusted R<sup>2</sup> value by putting "AdjustedRSquared" (quotes included) inside the brackets:

```

bestline[x]
0.542702 + 0.0543376 x

bestquadratic[x]
-0.0384337 + 1.29963 x - 0.415097 x^2

bestline["AdjustedRSquared"]
-0.0448773

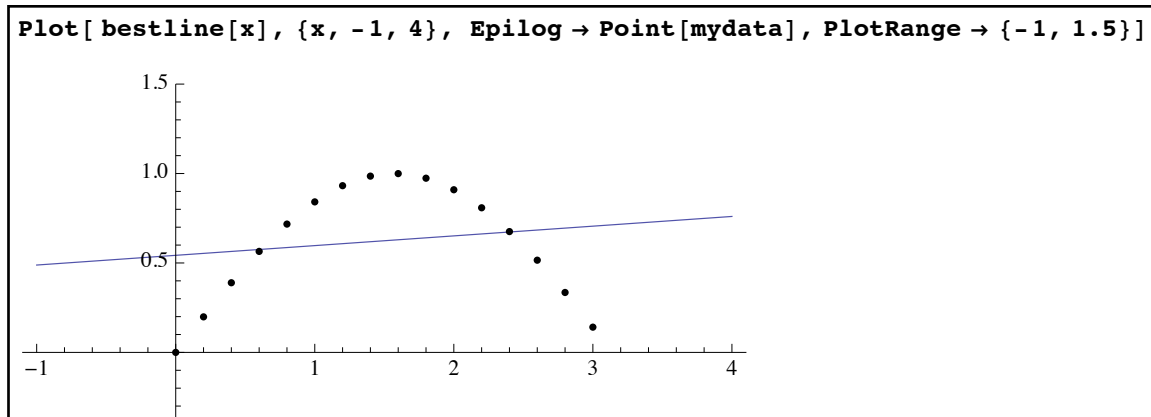
bestquadratic["AdjustedRSquared"]
0.99609

bestquadratic[4]
-1.48147

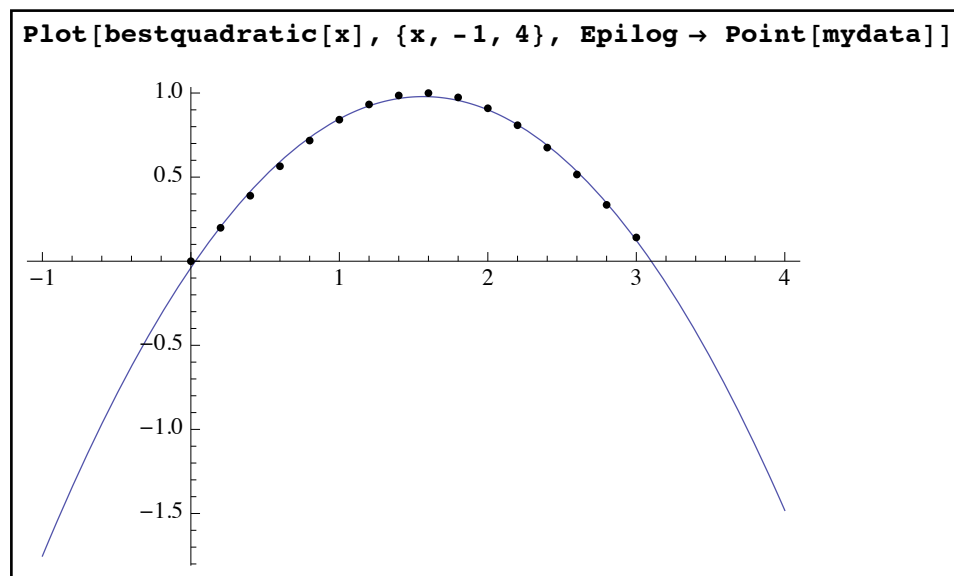
```

*seeing and judging linear regressions*

So here we have the formulas for the best fitting line and parabola to the data. We can see by the adjusted R<sup>2</sup> value that the line is a very bad fit but the parabola is a pretty good fit. We can also say that the parabolic fit predicts that when x is 4, y should be -1.48147. To see the fits and the data at the same time remember you can use the Plot command (remember to use the [x] after the name of your fit) together with the option Epilog→Point[data]:



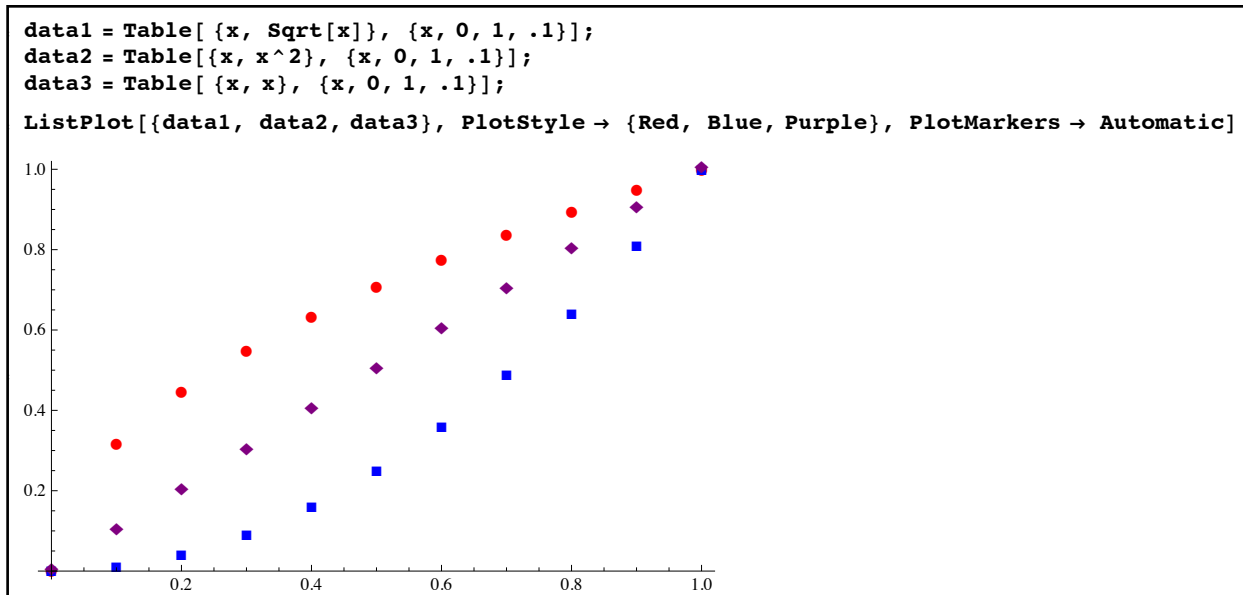
*a bad linear regression - not even close to the data*



*a good linear regression - the parabola fits the data pretty well*

If you don't like using Epilog to overlay the data on your graph you can also use ListPlot. If *dataset* is a list of coordinate pairs then ListPlot[*dataset*] is a graph of the data points which you can then use with Show to combine with other plots and graphics. If you have a list of different data sets, ListPlot[*list of data sets*] will graph all the data sets together and give each data set a unique color. You can use PlotStyle to change the color and PlotMarkers->Automatic to give the points from each set different shapes:





*graphing data using ListPlot along with PlotStyle and PlotMarkers*

Common linear regressions include “linear regression” (best fitting line, using the terms 1 and  $x$ ), “quadratic regression” (best fitting parabola, using the terms 1,  $x$  and  $x^2$ ), and “cubic/degree 3 regression” (using 1,  $x$ ,  $x^2$ , and  $x^3$ ).

Linear regressions are useful but at the same time they are restricted by the fact that the terms you decide you use can only be added, subtracted, or scaled. There are many formulas which cannot be broken down into basic terms using just those three operations. For example a common formula in many applications is exponential - that is, you would like to fit the formula  $ae^{bx}$  to your data where  $a$  and  $b$  are real numbers to be found. You can’t use linear regression to do this (at least not directly) since the unknown  $a$  is multiplied against the exponential and the unknown  $b$  is inside the exponent. Likewise linear regression would be unable to find a good fitting formula of the type  $\frac{1}{ax+b}$  where  $a$  and  $b$  are unknown numbers. This is where nonlinear regression comes in. The function `NonlinearModelFit` applies advanced statistical methods to fit a given formula to your data. Your formula will contain a set of “to be found” unknowns (usually called parameters) as well as the underling variable  $x$ . The command you would use to do nonlinear regression then becomes `NonlinearModelFit[ data, formula, parameterlist, x]` (you can replace the  $x$  with whatever variable you are using). So to fit the formula  $ae^{bx}$  to a set `data`, you would use the command `NonlinearModelFit[ data, a*Exp[b x], {a,b}, x]`. The result is another `FittedModel` object which you can use the same way as the ones from linear regression. For example, here is a set of data created from the tangent function, together with 2 different nonlinear fits:

```

mydata = Table[ N[ {k, Tan[k]} ], {k, 0, 3 / 2, 1 / 8}]

{{0., 0.}, {0.125, 0.125655}, {0.25, 0.255342}, {0.375, 0.393627},
 {0.5, 0.546302}, {0.625, 0.721484}, {0.75, 0.931596}, {0.875, 1.19742},
 {1., 1.55741}, {1.125, 2.09257}, {1.25, 3.00957}, {1.375, 5.04192}, {1.5, 14.1014}}

exponentialfit = NonlinearModelFit[mydata, a Exp[b x], {a, b}, x]

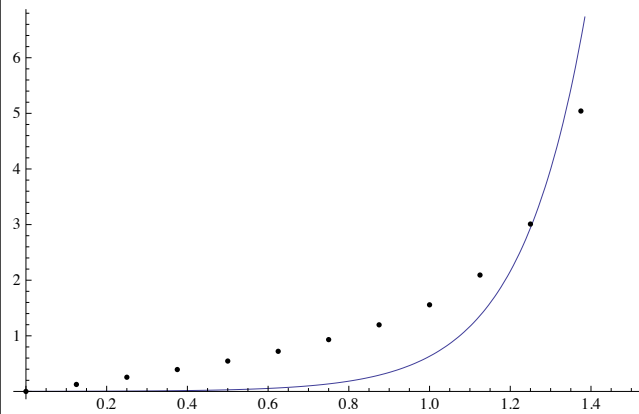
FittedModel[ 0.00134866 e6.14897 x ]

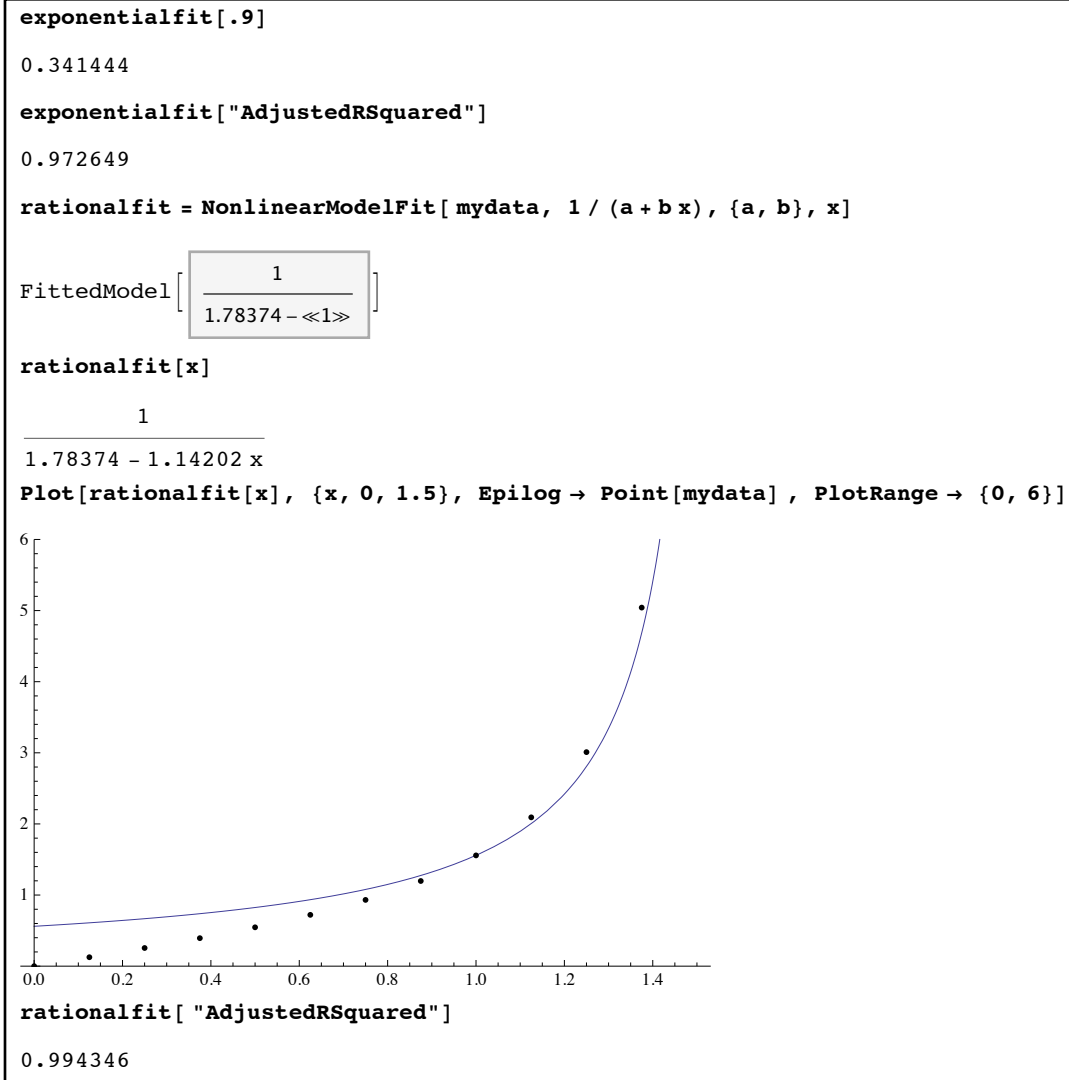
exponentialfit[x]

0.00134866 e6.14897 x

Plot[exponentialfit[x], {x, 0, 1.5}, Epilog -> Point[mydata]]

```





*two different nonlinear fits - while both fits are good the second is better with its higher Adjusted  $R^2$  value*

NonlinearModelFit will allow you to fit curves of almost any form to your data but it does have its pitfalls. The numerical estimation process tends to break down somewhat when there are too many parameters to find, when the number of data points is small, or when there's not a unique solution for the parameters. For example fitting curves of the form  $a \sin(x + b)$  to data may not work very well because more than one value for  $a$  and  $b$  will give the same fit (for example  $a = 1.2$  &  $b = 0$ ,  $a = 1.2$  &  $b = 2\pi$ , and  $a = -1.2$  &  $b = \pi$  all essentially give the same formula). If you have a rough idea of what the values should be you can give them to NonlinearModelFit in a manner similar to FindRoot - instead of  $\{a, b\}$  you could give starting values such as  $\{\{a, 1.0\}, \{b, 6.2\}\}$ . If giving parameter estimates does not help try looking up the option Method in the documentation (as in Method  $\rightarrow$  "FindMinimum" or Method  $\rightarrow$  "QuasiNewton"). The Method option requires a good bit of mathematics to explain in detail - it

essentially switches the numerical method used in estimating the parameters from one complicated technique to another complicated technique.

In this section we have barely scratched the surface of what Mathematica can do in probability and statistics. If you have had probability and statistics courses here are some additional Mathematica commands and options to look up which will help you get started on more advanced uses of Mathematica:

PieChart

BarChart

RandomReal (used for making pseudorandom real numbers from a variety of distributions)

PDF and CDF (probability density functions and cumulative distribution functions)

NormalDistribution, StudentTDistribution, GammaDistribution, ExponentialDistribution (a variety of continuous distributions)

BinomialDistribution, NegativeBinomialDistribution, GeometricDistribution,

PoissonDistribution, DiscreteUniformDistribution (a variety of discrete distributions)

“ParameterConfidenceIntervals” (obtain confidence intervals for parameters in

LinearModelFit)

ConfidenceLevel (set the confidence level for a confidence interval)

WeatherData (a command for downloading and installing weather data from stations all over the world)

FinancialData (a command for downloading and installing data from the stock markets)

### Section 3.3 Homework – Basic Statistics

- 1) Find the mean, quartiles, standard deviation, and mode for the data set  $\{1,1,1,2,2,3,3,3,4,4,4,4,5\}$ . Make a histogram for this data set.
- 2) Find the mean, median, quartiles, and 65th percentile for the data set  $\{55,56,65,70,72,80,81,82,83,87,87,89,90,90,92,98,100\}$ . Make a histogram of this data using 10 rectangles. Make other histograms using rectangles that are 1 unit, 5 units, and 10 units wide.
- 3) Suppose you give a survey and one question's responses are 1 (200 times), 2 (350 times), 3 (100 times), 4 (250 times), and 5 (75 times). Find the mean, median, standard deviation, and an appropriate histogram for these responses.
- 4) Enter the set of points  $\{(0,1), (1,3), (2,3), (3,5), (4,6), (5,9), (6,7)\}$  as the list “mydata”. Use ListPlot to graph this data both as a set of points and as a second plot where the points are connected by lines.
- 5) Find the best fitting line, parabola, and cubic for the data set “mydata” in problem 4. Give the formulas for these regressions, plot them against the data, and use their adjusted  $R^2$  value to determine which is the best fit. In addition for each model find the value predicted when  $x$  is 3.5 and 4.5.
- 6) Enter the points  $\{(-5,0.2), (-4, 0.3), (-3,0.4), (-2,0.5), (-1,0.7), (0,1), (1,1.4), (2,2), (3,2.7), (4,3.8), (5,5.4)\}$  into Mathematica as “data6”. Find the best fitting line, parabola, and

cubic for this data. Plot each one against the points and use the adjusted  $R^2$  value to determine which fit is best. Use each model to forecast the value of  $y$  when  $x$  is 6.

- 7) Find the best fitting curve to “data6” from the previous problem which has the form  $ae^{bx}$ , where  $a$  and  $b$  are parameters to be found. Plot this fit against the points and find its adjusted  $R^2$  value. Does this model appear to be better or worse than the “linear model” fits from problem 5?
- 8) Let “data8” be the set of points  $\{ (0,1), (.5, 1.09), (1,1.18), (1.5, 1.26), (2, 1.33), (2.5,1.39), (3,1.45), (3.5, 1.49), (4, 1.52), (4.5, 1.55), (5,1.58), (5.5, 1.59), (6, 1.61), (6.5, 1.62), (7, 1.63), (7.5, 1.64), (8, 1.64) \}$ . Find the best fit of the form  $\frac{1}{a + be^{kx}}$  where  $a$ ,  $b$ , and  $k$  are parameters. Plot this fit against the data and use it to estimate the value of  $y$  when  $x$  is 6.2. (the model  $\frac{1}{a + be^{kx}}$  is a “logistic” function which is often used to approximate the growth of animal populations over time).
- 9) Create a function LongestRun[ $n$ ] which returns the length of the longest run of heads or tails in  $n$  flips of a fair coin (this will use a combination of RandomChoice, Map, Split, Length, and Max). (you will use LongestRun in questions 10-12)
- 10) Use Table and your LongestRun function to create a data set consisting of 1000 values, each of which is the longest run of 100 coin flips. What is the average longest run? Make a histogram for this data.
- 11) Repeat problem 10 except use 1000 coin flips instead of 100. Then for 10,000 coin flips.
- 12) This problem will take the average run length ideas from problems 9-11 and generalize them. Define “rundata” as Table[ { $n$ , N[Mean[ Table[ LongestRun[ $n$ ], { $j$ ,1,1000} ] ] }], { $n$ ,1000,20000,1000} ] (this will take several minutes or so depending on your computer speed). This will create a list of data of the form (# of coin flips, average longest run length). Find the best fits to this data of the form  $a + bx$ ,  $a + b\sqrt{x}$ , and  $a + b \ln(x)$ . Plot the 3 functions against the data (include a legend) and use the adjusted  $R^2$  value to see which is the best fit.
- 13) In this problem we will look at some real-world weather data and try to fit an appropriate curve to it. It will take a few steps to get the data into a form that you can easily work with - so follow these steps:
  - a) Define “rawdata” by rawdata=WeatherData[ “KDUA”, “Temperature”, {{2005,1,1},{2014,7,31}}, “Day”], “DateNonMetricValue”][“Path”]; (this will download the daily temperature in Celsius from a weather station at Eaker Field in Durant, Oklahoma in the form { {year, month, day}, celsius temperature} - you can verify this by looking at rawdata[[1;;10]]).
  - b) Define “data” by data=Map[ {DateDifference[ {2005,1,1}, #1[[1]], “Year” ][[1]], #1[[2]]}&, rawdata]; (this will convert the data to the form {decimal date, fahrenheit temperature} where the date is the number of years since 1/1/2005 - you can verify this by looking at data[[1;;10]]).
  - c) Use ListPlot to graph the data using the option PlotStyle→PointSize[Tiny]. The points should appear to be roughly “sinusoidal” - that is a reasonable model might have the form  $a \sin(bx + c) + d$ . In this model  $|a|$  is the amplitude, the period

is  $\frac{2\pi}{|b|}$ ,  $d$  is the  $y$ -value on which the wave is centered, and  $c$  represents the shift of the wave to the left or right.

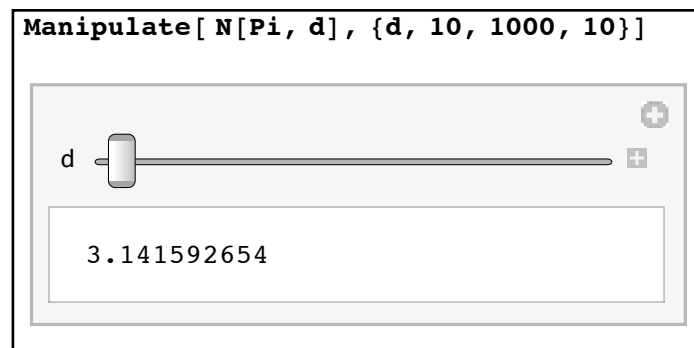
- d) In  $a \sin(bx + c) + d$  different values of the 4 parameters can give the same curve (think of the absolute values that relate  $a$  and  $b$  to the curve), so NonlinearModelFit will likely have trouble unless we give it some starting values for  $a$  and  $b$ . Using the graph from (c) estimate the values of  $a$  and  $b$ .
- e) Use the estimated values of  $a$  and  $b$  from part (d) in NonlinearModelFit to create a model “sine”. Graph the model and the data together as  $x$  goes from 0 to 6.

The WeatherData command in problem 13 can be used to pull a wide range of weather values (temperature, wind speed, humidity, etc.) from weather stations all over the country (“KDUK” happens to be the name of the station at Eaker Field - look at the documentation for WeatherData to see how to locate other weather stations). There are several “Data” commands in Mathematica that allow you to pull curated data sets off the internet and into Mathematica. Others include FinancialData, CountryData, ChemicalData, LanguageData, AstronomicalData, UniversityData, CityData, and so on (use ?\*Data to find these and others). You can see the types of information each can give you by evaluating `dataname[“Properties”]`.

## Section 3.4 - Basic Interactive Manipulations

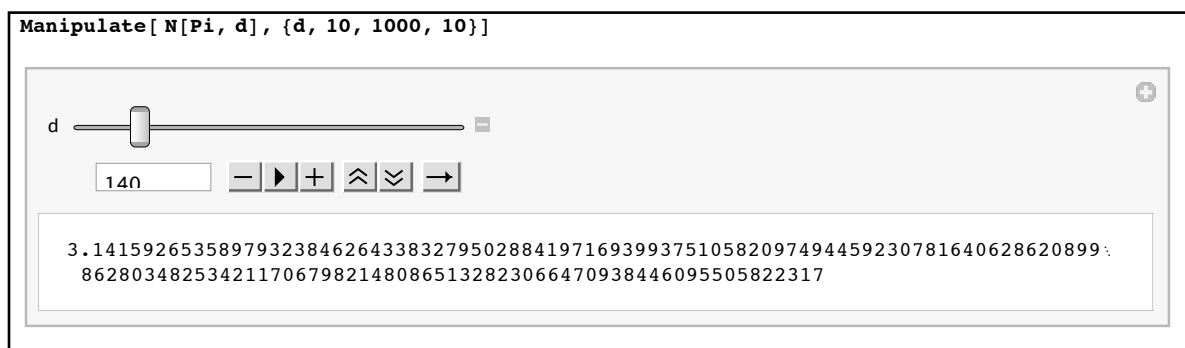
Versions 6 and later of Mathematica have a very useful capability - the ability to create your own interactive manipulations of mathematical concepts. These manipulations can greatly extend your ability to investigate and display different mathematical ideas in the same way that Geometer's Sketchpad allows you to demonstrate ideas in geometry in ways that can have a much greater impact than static pictures.

The key command for creating these interactive manipulations is `Manipulate`. The basic idea behind `Manipulate` is to have a Mathematica command or chain of commands that have one or more parameters in them instead of specific inputs, then vary the parameters as you want. The most basic form for `Manipulate` is `Manipulate[expression, {parameter, start, finish}]` or `Manipulate[expression, {parameter, start, finish, step}]`. For example, evaluate the command `Manipulate[N[Pi, d], {d, 10, 1000, 10}]`:



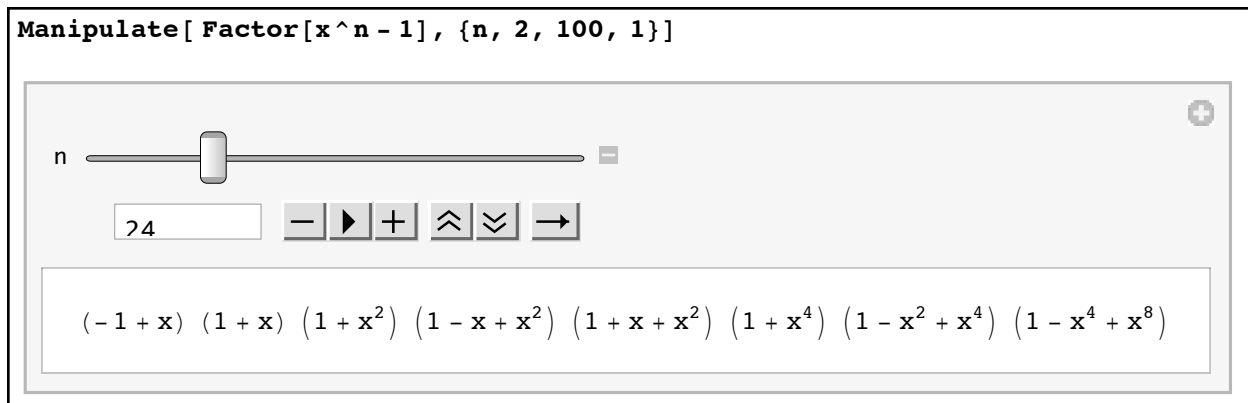
*A very basic manipulation window*

This manipulation estimates the value of  $\pi$  to  $d$  places, where  $d$  is a parameter which starts at 10 and goes up to 1000 in steps of 10. In this case the value of  $d$  is controlled by a labelled slider (which is the default control for a parameter which represents a number which varies over a range). If you click the small “+” symbol at the end of the slider you will see the value of  $d$ , which will change as you move the slider around:



*140 digits of  $\pi$*

This isn't a very deep manipulation, but here is a similar one from algebra:  
 Manipulate[ Factor[  $x^n - 1$  ], {n, 2, 100, 1}] (with the slider moved and expanded):



*A manipulation for factoring  $x^n - 1$*

In this case the step value of 1 is necessary to make sure the exponent is always a whole number (otherwise just like when graphing a curve the slider will sample values every so often, and these intermediate values may not always be whole numbers). Like the manipulation for the digits of  $\pi$  at first glance this may not seem so important. But ask yourself (or imagine asking a class of algebra students) the following questions:

What is always a factor of  $x^n - 1$ ? When does  $x^n - 1$  only have two factors?

While these questions are easy to pose actually working enough examples to make a guess as to the answers would be very tedious on paper. Using the manipulation takes all the tedium out and makes the questions easier to answer (the answer to the first question is " $x - 1$ " and the answer to the second is "when  $n$  is prime").

The controls we can use aren't limited to just sliders. The most common include sliders (which are the default when a parameter runs over a range of values), input fields (where you can type whatever you want in a blank field), selections from a range of buttons (pushing a different button changes the parameter), and popup menus (where the different values of the parameter appear in the popup menu). You can have more than one parameter you can vary in any given manipulation, have the parameter start with any value you want, and label the control for each parameter as well. The expanded version of the Manipulate command will look like

Manipulate[ *expression*, *parameterlist1*, *parameterlist2*, ....]

where each *parameterlist* will control a single parameter and its control type, starting value, and label. To get each common control type, use the following formats:



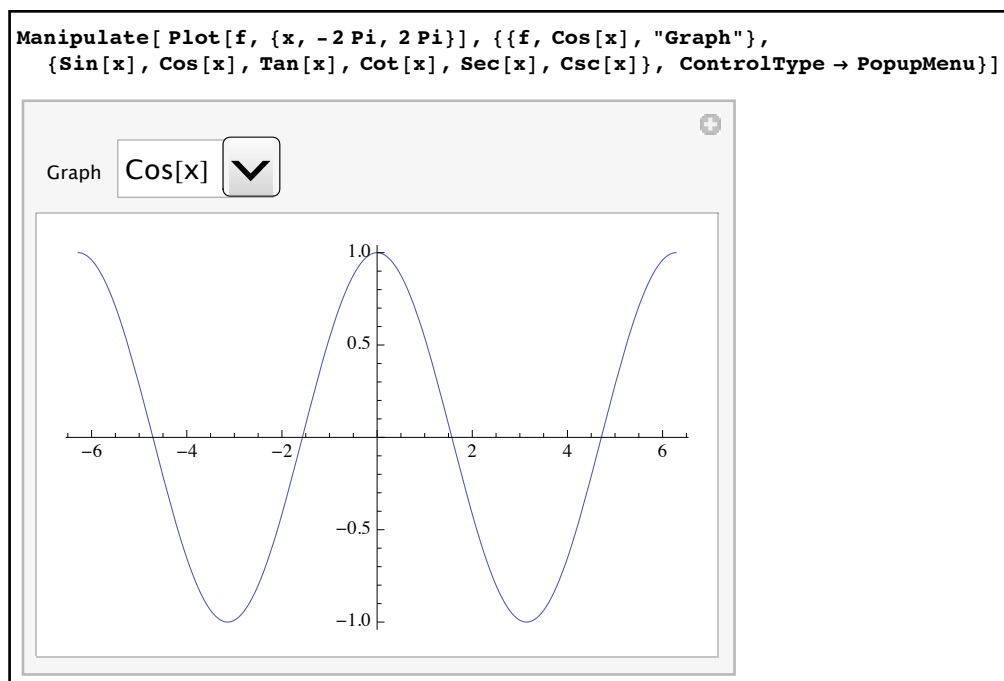
Input fields: Use either  $\{parameter\ name, start\ value\}$  or  $\{\{parameter\ name, start\ value, "label"\}\}$  (the *label* must be in quotes if it consists of text).

Sliders: Use either  $\{parameter\ name, range\ start, range\ end\}$  or  $\{parameter\ name, range\ start, range\ end, step\}$  for a basic slider. To label your slider or to have the initial value not be *range start*, use  $\{\{parameter\ name, initial\ value, "label"\}, range\ start, range\ end\}$  or  $\{\{parameter\ name, initial\ value, "label"\}, range\ start, range\ end, step\}$ .

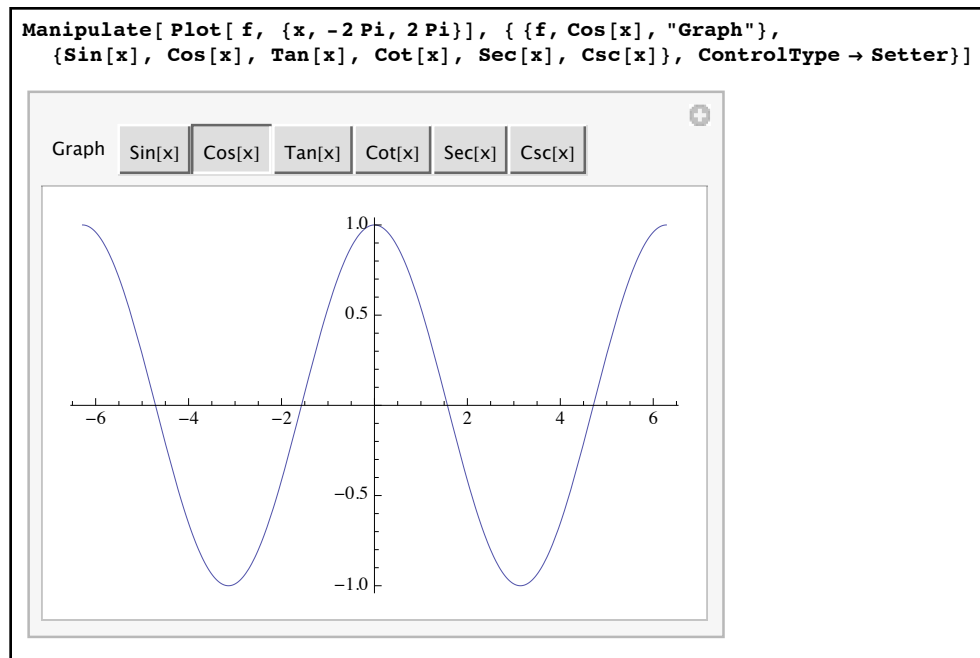
Buttons: Use either  $\{parameter\ name, \{value\ 1, value\ 2, \dots\}, ControlType \rightarrow Setter\}$  for a basic row of buttons or  $\{\{parameter\ name, initial\ value, "label"\}, \{value\ 1, value\ 2, \dots\}, ControlType \rightarrow Setter\}$  to label the control or start with a button other than the first one selected to begin.

Popup menus: Use either  $\{parameter\ name, \{value\ 1, value\ 2, \dots\}, ControlType \rightarrow PopupMenu\}$  for a basic popup menu or  $\{\{parameter\ name, initial\ value, "label"\}, \{value\ 1, value\ 2, \dots\}, ControlType \rightarrow PopupMenu\}$  to label the control or start with a menu selection other than the first one selected to begin.

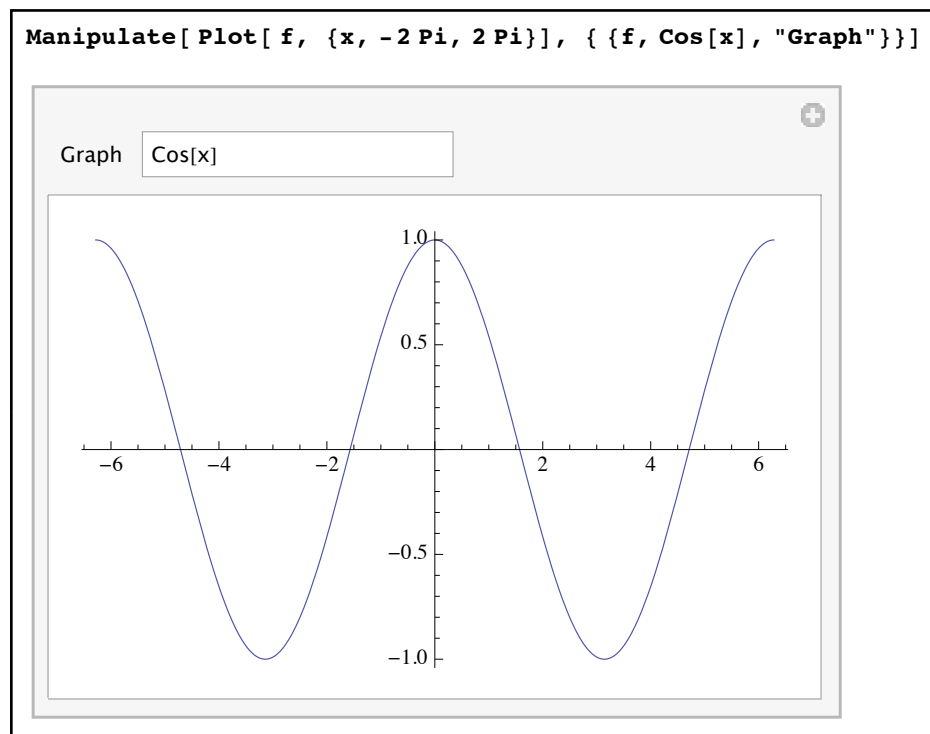
To see each of these control types in action imagine that you want to be able to see the graphs of the 6 basic trigonometric functions from  $-2\pi$  to  $2\pi$ . In each case the command you are going to use is `Plot[f, {x, -2Pi, 2Pi}]`, where *f* is the trigonometric function you want to see. For whatever reason you want to list the functions in the order sine, cosine, tangent, cotangent, secant, and cosecant but start with cosine as the shown graph. Here are 3 different manipulations that can make this happen (a slider control wouldn't work well for this):



*A manipulation with a popup menu*



*A manipulation using buttons*

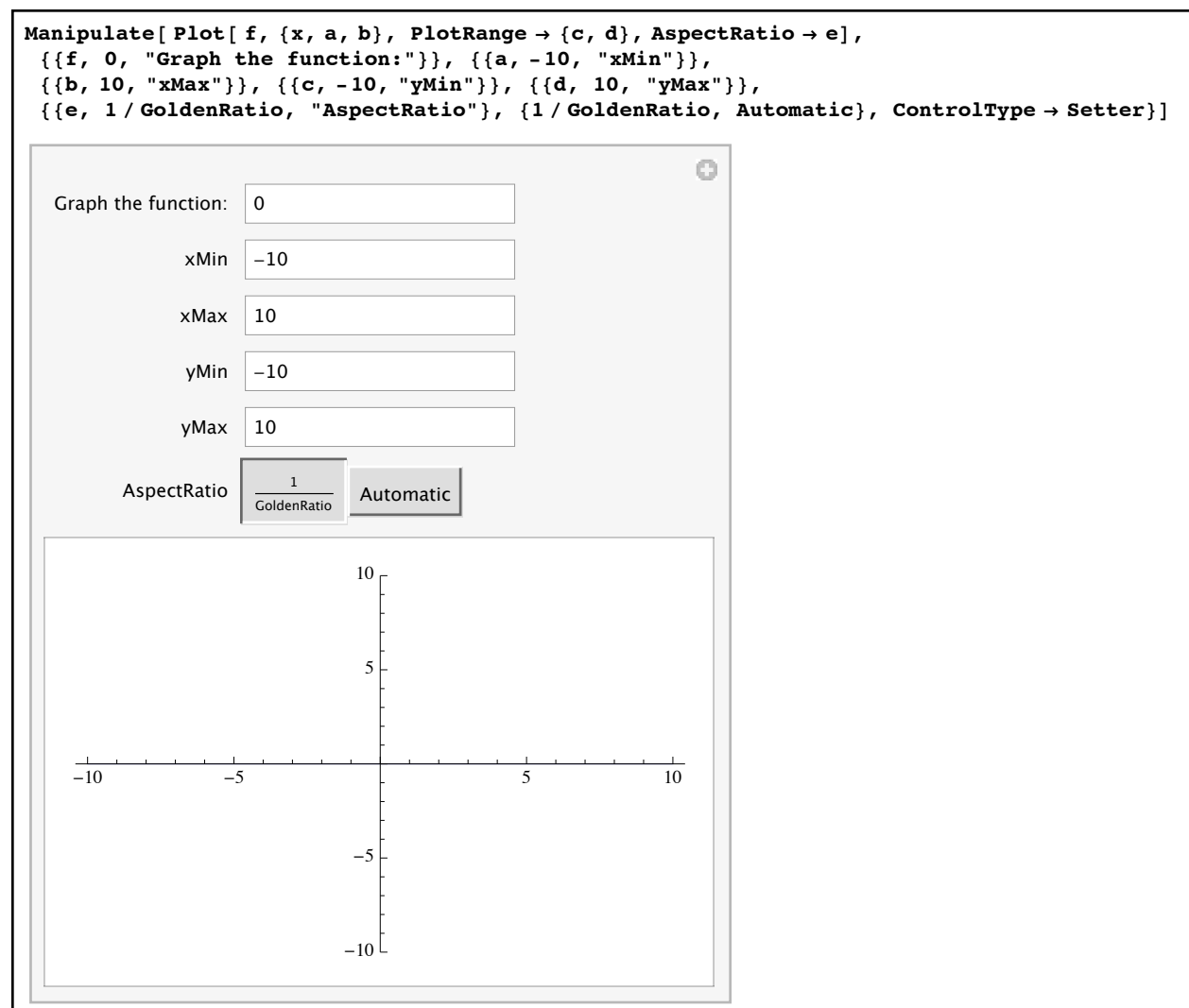


*A manipulation using an input field*

The input field manipulation will allow you to graph not just the 6 trigonometric functions but also functions like  $x^2$ . This is either good or bad depending on what you are trying to do - if you need to focus on just the trigonometric functions the button or popup approaches might be better

as they give you a specific set of choices. If you want the flexibility to graph functions like  $\sin^2(x)$  then the input field approach is probably the better one.

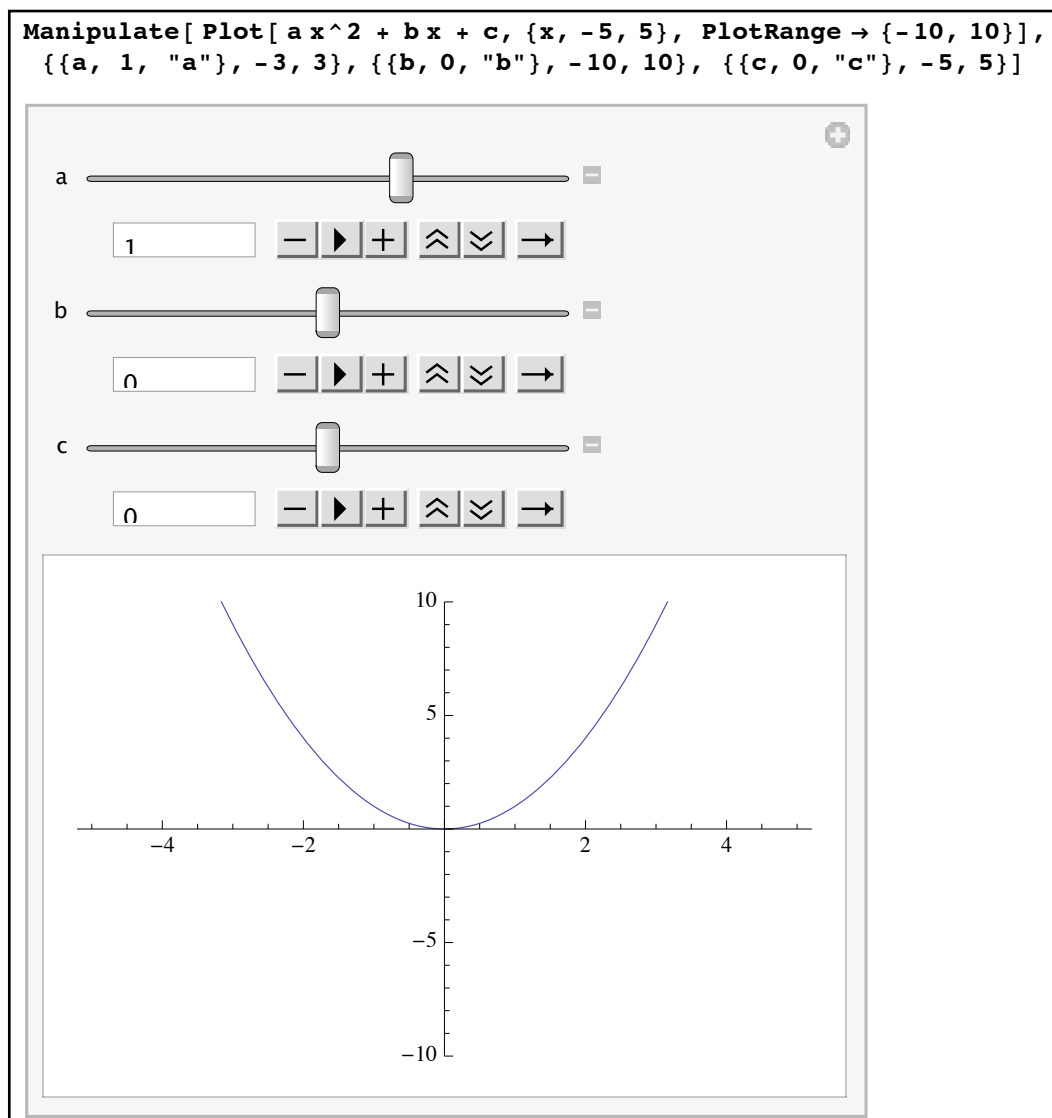
Each of these manipulations used just one parameter. You are not limited to just one parameter per Manipulate - you can have as many as you want. Suppose you want to mimic a graphing calculator. The inputs you need are the function to be graphed, the ranges to use for both the x values and y values in the display (by default most calculators go from -10 to 10 in both variables), and whether you want the graph to use the same scales on the axes or not (by default most calculators use a smaller scale for the y-axis than the x-axis just as Mathematica does). There are several different ways in which you can set this up. Using a different parameter for each and putting in the appropriate labels you would get something that looks like this:



*A graphing calculator manipulation*

Using the initial function 0 hides the graph in the x-axis so it looks like the graph is blank to begin with (just like on a calculator).

Another example of using a simple manipulation to explore a mathematical concept is understanding the graph of a quadratic. A quadratic function is one of the form  $f(x) = ax^2 + bx + c$  where  $a$ ,  $b$ , and  $c$  are real numbers. A common question when first learning about quadratics is “What effect does changing  $a$ ,  $b$ , and  $c$  have on the graph of  $f(x) = ax^2 + bx + c$ ?”. We can set this up for investigation using a Manipulate command where  $a$ ,  $b$ , and  $c$  can take on different values. Slider controls would work well for all three parameters (since they can vary over a range). The most basic quadratic is  $f(x) = x^2$ , which uses  $a=1$ ,  $b=0$ , and  $c=0$  (this gives us a good starting value for each parameter). Putting these together and picking a decent range of values for  $a$ ,  $b$ , and  $c$  might lead to the following manipulation:



*A manipulation for examining graphs of quadratics*

For this manipulation having a fixed PlotRange setting is important - it makes the changes in the graph smoother (because the scale on the y-axis won't be changing as we move the sliders around). In general to make a nice-looking manipulation involving graphs you either want to use the same range of x- and y-values or have them set by controls in the manipulation itself.

This all just barely scratches the surface of what you can do with interactive manipulations in Mathematica. There are additional types of controls (radio buttons, check boxes, and even movable points as in Geometer's Sketchpad) as well as finer ways to display them (using labels for buttons and popup menus, placing the controls on different sides of the manipulation, etc.). You can find thousands of examples of manipulations (called demonstrations) at the Wolfram Demonstrations Project ([demonstrations.wolfram.com](http://demonstrations.wolfram.com)). If you don't have Mathematica on a given computer you can see these demonstrations by downloading the free Wolfram CDF Player from [www.wolfram.com/cdf-player](http://www.wolfram.com/cdf-player). The CDF Player functions like a PDF reader for Mathematica - you can see Mathematica documents and demonstrations (although you cannot use demonstrations with input fields), but you can't edit them. Demonstrations are also limited in the kinds of controls they can use - for example input fields can only be used for numerical input.

### Section 3.4 - Basic Interactive Manipulations

- 1) Explain the difference between an input field, setter, popup menu, and slider control.
- 2) Create a manipulation which applies TrigReduce to  $\sin^n(x)$ , where  $n$  is an integer from 1 to 20. Use the information in the manipulation to make a guess as to what the angles will be in the result.
- 3) Create a manipulation which plays the sound  $\sin(2\pi \times 220t) + \sin(2\pi \times kt)$  for 4 seconds, where  $k$  is controlled by an input field with an initial value of 230. Use this manipulation to see what happens as you try values of  $k$  closer and closer to 220 (recall that the Play command is used to generate sounds).
- 4) Create a manipulation which graphs the curve  $y = a \sin(bx + c) + d$  from  $-2\pi$  to  $2\pi$  in terms of  $x$  and  $-5$  to  $5$  in terms of  $y$ , where  $a$ ,  $b$ ,  $c$ , and  $d$  are controlled by sliders with ranges of  $-3$  to  $3$  and initial values of  $a = 1$ ,  $b = 1$ ,  $c = 0$ , and  $d = 0$  respectively. Use these sliders to determine what roles  $a$ ,  $b$ ,  $c$ , and  $d$  play in the graph.
- 5) Create a manipulation which displays the graphs of the following formulas as  $x$  and  $y$  go from  $-10$  to  $10$ :  $\frac{1}{x}$ ,  $x$ ,  $\sqrt{x}$ ,  $\sqrt[3]{x}$ ,  $x^2$ ,  $x^3$ . Have the function selected from a row of buttons.
- 6) Repeat problem 5 but have the functions selected from a popup menu.
- 7) Create a manipulation for the command `Plot[ Sin[x], {x,0,2Pi}, PlotStyle→Hue[k] ]`, where  $k$  is controlled by a slider from  $0$  to  $1$ . What do you think Hue controls based on this manipulation?
- 8) Create a manipulation which will display the standard " $n$  times" tables as  $n$  goes from  $2$  to  $50$  (i.e. the "5 times table", the "6 times table", etc.). Use whatever control you think is appropriate.

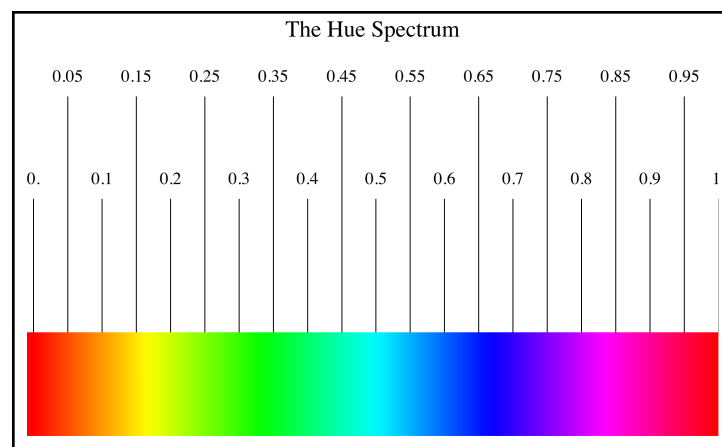
- 9) Repeat problem 8 except use the “mod  $n$  multiplication” tables as  $n$  goes from 2 to 50 (modular multiplication tables were mentioned in the homework in Section 3.2). As you vary  $n$  sometimes 0's will appear in the middle of the table and sometimes the only 0's will be in the first row and column. What property of  $n$  controls this?
- 10) Evaluate the following command and describe the result:  
`Manipulate[ Rotate[ Plot[ Sin[x], {x,0,2Pi}], k Degree], {k,0,360,1} ]`
- 11) Go to the Wolfram demonstrations site and find 3 demonstrations from your subject area that you find interesting. List the titles of these demonstrations.
- 12) Dr. Moretti has a number of downloadable Mathematica notebooks that use Manipulate to illustrate various mathematical ideas. Go to his website (<http://homepages.se.edu/cmoredti/>) and find 3 notebooks you find interesting there (they are grouped by mathematical subject). List their titles.

## Section 3.5 - Advanced Graph Control

In Chapter 2 we took a look at several directives that could be used to control the appearance of curves and regions - both common colors and other modifiers like `Thin` and `Dashed`. Each one gave us ways to distinguish graphs but were still limited in extent - we had to rely on Mathematica's limited set of color names, predefined notions of thickness, and so on. Each one of those directives is just the tip of the iceberg - each can be generalized to give you far more control over the appearance of curves and regions.

Mathematica has many different ways to control color and shading, for example:

`Hue[h]`: Hue represents color along a spectrum, where  $h$  ( $h$  ranging from 0 to 1) controls which color from the spectrum you want to use. The color on this spectrum varies from red, yellow, green, blue, pink, and then back to red (red being both `Hue[0]` and `Hue[1]`).



*the Hue spectrum*

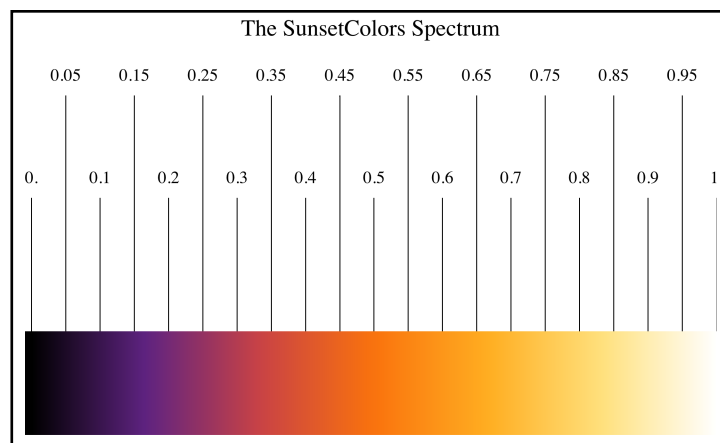
`RGBColor[r,g,b]`: `RGBColor[r,g,b]` (where  $r$ ,  $b$ , and  $g$  range from 0 to 1) allows you to give the precise RGB specification of any color. The RGB specification is used in many high-end graphics programs like Adobe Photoshop (although the values given for such programs range from 0-255 and so have to be scaled by dividing each number by 255). The numbers  $r$ ,  $g$ , and  $b$  represent the strength of the red, green, and blue used in the display - 0 is off and 1 is full strength. So `RGBColor[1,0,0]` is pure red, `RGBColor[0,0,1]` is blue, `RGBColor[1,0,1]` is purple, `RGBColor[1,1,1]` is white, and so on. For most purposes it is easier to use the "common colors" specifications. If you need to use a precise color (like vermilion for example) you can find the RGB specifications for it on the internet and recreate it using `RGBColor`.

`CMYKColor[c,m,y,k]`: `CMYKColor` (where the values of  $c$ ,  $y$ ,  $m$ , and  $k$  go from 0 to 1) represents colors in the CMYK color system (the CMYK system is used in high-end

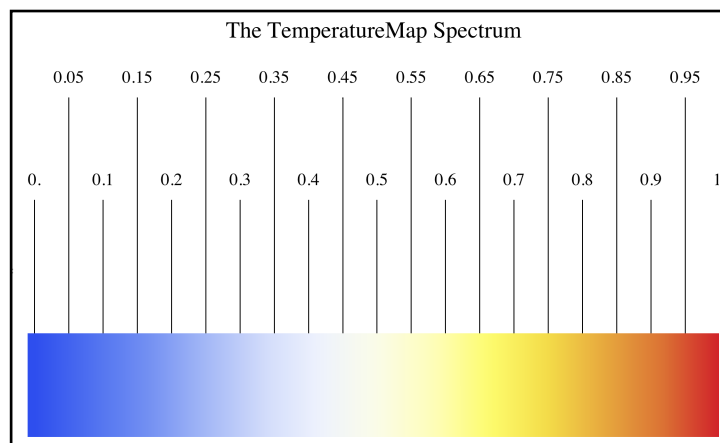
print applications - the kind used for sending magazines off to large-scale printing).  $c$  is the strength of cyan,  $y$  the strength of yellow,  $m$  the strength of magenta, and  $k$  the strength of black.

`ColorData[gradient][h]`: `ColorData` represents a function which selects colors from a pre-defined list of color gradients (the gradient names are strings like “SunsetColors” or “TemperatureMap”, quote symbols included). Each gradient is a spectrum of colors and the value of  $h$  (from 0 to 1) represents where on the spectrum the color you want is. So you might use a plotting option like `PlotStyle`→

`ColorData[“SunsetColors”][.3]`. You can get a complete list of gradient names using the command `ColorData[“Gradients”]`.



*the “SunsetColors” gradient*



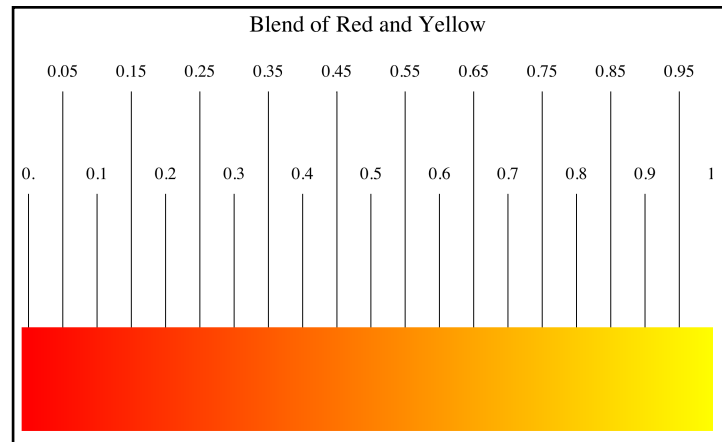
*the “TemperatureMap” gradient*

`Lighter[color]`: `Lighter` gives you a lighter version of *color* (regardless of how the color was named - so you could have `Lighter[Yellow]` or `Lighter[RGBColor[.3,.4,.5]]`. `Lighter` will typically give the new color using `RGBColor`.



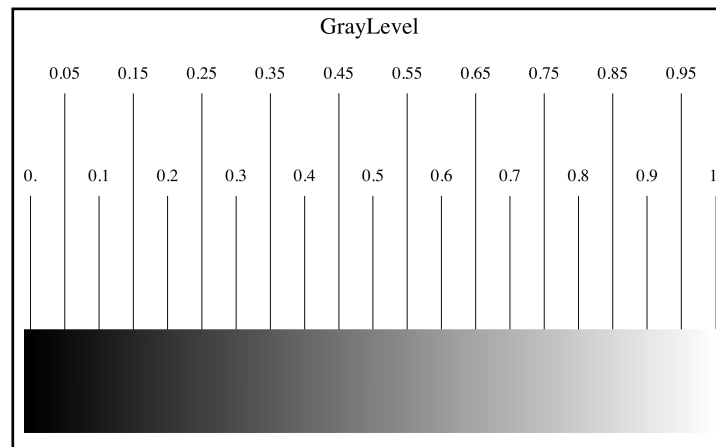
`Darker[color]`: More or less the same idea as `Lighter`.

`Blend`: `Blend[ {color1, color2}, x]` represents a color which is “*x*<sup>th</sup>” of the way from *color1* to *color2*. So `Blend[ {Red, Yellow}, 1/4]` would be a color which is 1/4 of the way from Red to Yellow (which is a reddish orange). `Blend` also can be used to combine more than 2 colors - to see how this is done check the Mathematica documentation.



*Blend[{Red, Yellow}, *k*] as *k* goes from 0 to 1*

`Graylevel[g]`: `Graylevel` (with *g* varying from 0 to 1) represents shades of gray with 0 being black and 1 being white.



*Graylevel[*g*] as *g* goes from 0 to 1*

`Opacity[o]`: `Opacity` (with *o* ranging from 0 to 1) represents the transparency of a curve. 0 represents complete transparency, and 1 represents complete solidity. When graphing multiple objects with `Opacity` remember that objects defined first will be on the “bottom” of the graph stack - so the graphing order can make a big difference if some of the objects are semi-transparent.

Dashing[ {*solidamount*, *spaceamount*}]: Draw a dashed curve with segments and spaces of the appropriate width, where *solidamount* and *spaceamount* range from 0 to 1 and represent the fraction of the overall plot width. Dashing[ {.1, .04}] represents a dashed curve where the solid parts are .1 of the width of the plot and the gaps are .04 of the plot width.

Thickness[*t*]: Thickness[*t*] (*t* varying from 0 to 1) represents the thickness of the curve as a fraction of the horizontal width of the plot. Very rarely would you use a value for *t* larger than say .05 but the option is there.

PointSize[*s*]: PointSize (with *s* varying from 0 to 1) controls how large points will be drawn. *s* represents the diameter of the dots used for each point, given in terms of the width of the entire plot (so as in Thickness you will typically use small values in PointSize). PointSize can be used as a PlotStyle directive in ListPlot. It can also be used in Epilog for the placing points in a graph. You do this by setting Epilog to a list in which a PointSize or color directives affect everything that comes after it. So Epilog→{PointSize[.03], Red, Point[data1], Blue, PointSize[.01], Point[data2]} would overlay a set of red points and blue points on a graph with the blue points one third the size of the red points.

Using these directives in graphing options like PlotStyle, RegionStyle, BoundaryStyle, and ChartStyle will give you very precise control over the appearance of your graphs. They are also used in some other “Style” options:

AxesStyle: AxesStyle→*colordirective* uses the *colordirective* as the color for the axes and tick marks.

FrameStyle: FrameStyle→*colordirective* uses the *colordirective* as the color for the frame and tick marks. You will still need to include the option Frame→True.

Background: Background→*colordirective* uses the *colordirective* for the background color.

BaseStyle: BaseStyle→*colordirective* uses the *colordirective* as the default color for the axes, tick marks, labels, and curve. BaseStyle controls all those things at once, and individual components can be overridden by AxesStyle, etc.

## Section 3.5 - Advanced Plot Control

- 1) Graph  $y = x^2$  from -2 to 3 in light gray.
- 2) Graph  $y = \cos(x)$  from 0 to  $2\pi$ , making the graph thicker than the standard Thick option.

- 3) Graph  $y = x^2$ ,  $y = (x - 3)^2$ , and  $y = (x - 2)^2 - 2$  on the same set of axes, distinguishing curves by thickness (use a legend as well).
- 4) Repeat problem 3 but make the curves orange, yellow, and green using Hue (this may require some experimentation).
- 5) Repeat problem 3 but make the curves orange, yellow, and green using RGBColor (this may require some experimentation).
- 6) Repeat problem 3 but make the curves crimson, forest green, and powder blue using RGBColor (you will need to look up the RGB values for these on the internet - don't forget to scale the numbers to the 0-1 range by dividing by 255).
- 7) Repeat problem 3, making each curve Red with a different level of opacity over a black background.
- 8) Repeat problem 3, coloring each curve using the values .1, .5, and .7 in the "CandyColors" gradient.
- 9) Repeat problem 3, coloring each curve using the values .1, .5, and .7 in the "Alpine" gradient.
- 10) Repeat problem 3, coloring each curve using the values .1, .5, and .7 in the "LakeColors" gradient.
- 11) Repeat problem 3, coloring the curves red, yellow, and blue with green axes and a white background.
- 12) An option for plot we have not discussed is ColorFunction, which colors the graph based on the coordinates of its points. It takes the form of ColorFunction→(*pure function of 2 variables in parentheses*) or ColorFunction→*gradient name* (by default values for directives like Hue are automatically scaled to the right range). Evaluate Plot[ Sin[x], {x, 0, 2Pi}, ColorFunction→(Hue[#1]&) ] and Plot[ Sin[x], {x, 0, 2Pi}, ColorFunction→(Hue[#2]&) ] and describe how they work.
- 13) Evaluate the command Plot[  $x^2$ , {x, -3, 4}, ColorFunction→"TemperatureMap"] and describe how it works.
- 14) Many of these directives can be used in places other than plotting commands - try evaluating Style[ N[Pi, 20], RGBColor[0, 1, 0] ] and Style[ Expand[ (x+1)^6 ], Opacity[.2] ].

## Section 3.6 - Importing and Exporting from Mathematica

If you are creating Mathematica graphs to use in other applications then in addition to using the techniques of the previous sections to make your graphics you will also need to get them from Mathematica to the other programs. The easiest way to do this is to simply select the graph you have made, use the standard “copy” shortcut (ctrl-C in Windows and command-C in Mac OS X), switch to the other program, and then use the “paste” shortcut (ctrl-V and command-V respectively) to insert the picture in the other application. This works at a basic level but has one major restriction - the pasted image is a low resolution bitmap image which may be unsuitable for many applications and may not look good when printed.

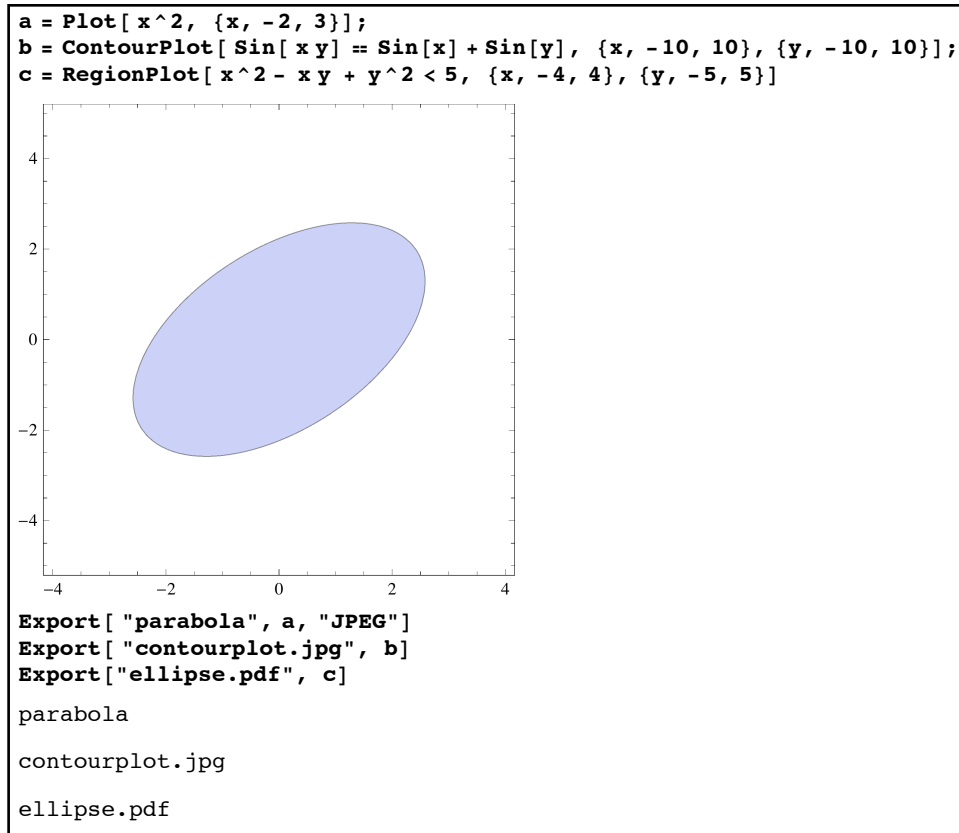
To counter this Mathematica has the ability to export graphics in many different formats. For saving simple bitmap files (as well as a range of other graphics formats) this can be done through the use of the “Save Selection As...” menu option. To use this, simply select the graphic, and then under the **Edit** menu you will be able to select “Save Selection As...”. This will force another pop-out menu which will allow you to select the file type. A standard “Save” dialog box will then let you save the graphics file where you want to. Mathematica can save graphs and other outputs to a wider range of formats through the use of the Export command.

The Export command saves a particular Mathematica output as a new file. The standard save directory is usually the “My Documents” folder under Windows and the Home directory in OS X. To see what the default save directory is simply evaluate the command `Directory[]` (with nothing in the square brackets) to see where files will be saved.

The basic structure of the Export command is `Export[“filename”, objectname, “filetype”]`, where *objectname* is the name of the graphic to be exported (if not given an individual name it can be referred to by an `Out[]` number), *filetype* is a name for the type of file you would like to end up with, and both *filename* and *filetype* must be in quotes (for those wishing to shorten the command a bit the proper file extension can be used at the end of the *filename* and then the “filetype” and its preceding comma may be omitted). A partial list of the image file types is:

BMP	Windows bitmap
AI	Adobe Illustrator
EPS	Encapsulated Postscript
EPSTIFF	Encapsulated Postscript with TIFF preview
GIF	GIF file
JPEG	JPEG file (with moderate compression)
PDF	Adobe Portable Document Format
PICT	Macintosh PICT format (bitmap)
TIFF	TIFF file
PNG	PNG file

You can get a full list of every possible export type (not just for image files) by using the command `$ExportFormats`. The filetype you use will depend very much on what you want to use the image for - most word processors will be able to use JPEG files and the more advanced ones can use PDF or PNG files. Almost all the images used in this book were saved as PDF files and then imported into a desktop publishing program. Here is an example of setting up three different graphics and exporting them in various types:



*Exporting files from Mathematica*

In many cases you will want to specify the resolution of the exported file, especially if you are using the common JPEG and TIFF formats. The standard image resolution for the Export is only 72 dpi (“dots per inch”) which is not enough for high quality printing (if you’ve worked with printing digital photos at all you probably know that most good prints use 300-600 dpi). You can increase the resolution of the exported files using the option `ImageResolution→d`, where *d* is the number of dots per inch. This option will not work for resolution-independent formats like PDF and EPS. Increasing the resolution of the exports will also increase their file size; as a rule of thumb doubling the resolution roughly quadruples the file size.

Although we have only currently discussed the Export command in terms of graphics Mathematica can use the Export command to create non-graphics files as well. One example of this is the creation of sound files. By using the file type AIFF Mathematica can export a sound (such as the result of the Play command, which creates a sound in much the same way that Plot

creates a graph) as an AIFF sound file (the raw sound file type used on CD's). This file can then be converted by other programs such as Windows Media Player or Quicktime Pro into various other sound formats. Another example of exporting a non-graphic file is exporting a table in Mathematica as an Excel file by using the file type "XLS" (as in `Export["mydata.xls", Out[12]]` or something similar).

There may come a time when you wish to export not just an individual graphic but an entire notebook, calculations and all. You may want to post an entire notebook online to show some complex calculation or to take advantage of Mathematica's typesetting capabilities. To do this go to the **File** menu and choose "Save as Special". One of the options on the pop-out menu will be "HTML" (HTML files are the most common type of file for posting pages on the web). A save dialog will come up, and you will have to enter a name. What this does is creates a folder with the entered name. Inside this folder will be a file called `index.html` and lots of supporting files and folders. The `index.html` file is the web page which represents the notebook. When opened by a web browser the `index.html` file looks exactly like the original notebook, including all complex notation and graphics. The web page is not interactive, however - people cannot evaluate cells, resize graphics, or use the controls of a Manipulate command. Mathematical expressions and graphs are part of the page as GIF graphic files instead of text. It is possible to save notebooks in the more advanced XML format, but this may require special browsers and/or fonts to be installed.

The mirror image of the Export is the Import command. `Import["filename"]` will bring the file *filename* into Mathematica. Like Export, Import will look for files in the default directory given by the `Directory[]` command and it will automatically try to deduce the file format from the extension at the end of the file name. So `graphic1=Import["beach.jpg"]` would look for the file `beach.jpg` in the default directory, copy it into Mathematica, and store it in the variable `graphic1` (Mathematica has many functions for working with graphic files although they are beyond the scope of this course). One of the most common file types to import are Excel files (as the spreadsheet format used by Excel is a common way to enter and store large data sets). Excel files store each table in a different sheet and the tables can be extracted at the time of import using the part notation for lists. So `dataset1=Import["datafile.xls"][[1]]` would store the first sheet in `datafile` as `dataset1`, `dataset2=Import["datafile.xls"][[2]]` would read in the second sheet and so on. You could just do a single import as `fulldata=Import["datafile.xls"]` and then let `dataset1=fulldata[[1]]`, and so on as well.

## Section 3.6 Homework - Importing and Exporting from Mathematica

- 1) Export the graphic for problem 6 in Section 3.3 as a PDF file.
- 2) Define a graphic "graph" to be the result of the command `Plot3D[x^2-y^2, {x,-1,1},{y,-1,1}]` (this is a 3-dimensional graph). Export this graphic to the JPEG `saddle1.jpg` and then to a second JPEG `saddle2.jpg` which uses 300dpi. Bring these files into your favorite word processor and compare the results.

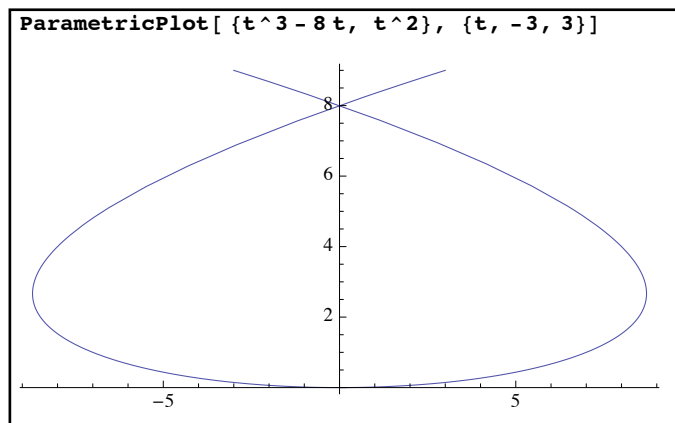
- 3) Take the data and fits from problem 11 in section 3.3. Export this as a JPEG file saved at 600dpi.
- 4) Create a 5 second “play” of the sound  $\sin(2\pi * 440t)$ . Export this as “sound.aiff”, and see if you can play it on your computer. (remember sounds are generated by the Play command).
- 5) Make a table of the numbers  $\{k, \text{Prime}[k]\}$  as  $k$  goes from 1 to 500. Export this as an Excel file and check in it Excel or similar program.
- 6) Save your 3.5 homework notebook as html and view it in your browser of choice.

## Section 3.7 - Additional Graphing Commands

In Chapter 2 we introduced several graphing commands to handle specific situations - Plot, ContourPlot, and RegionPlot. All three commands handle variations of the same basic scenario - graphs of Cartesian quantities (be they functions, equations, or inequalities) that use standard scaling on the two axes. Mathematica has several additional graphing commands to address other situations - non-Cartesian graphing (parametric and polar) and the use of logarithmic scales (commonly used applications in science and engineering).

In parametric graphing rather than thinking of  $x$  and  $y$  as being directly related we consider them to be functions of another variable  $t$  - so each value for  $t$  corresponds to a point  $(x(t), y(t))$  and the combination of all those points makes a curve (often you think of the curve as being sketched out by a point moving over time, which is why  $t$  is commonly used as the underlying variable).

Suppose that a graph is defined parametrically in terms of a pair of functions ( $x$ -function,  $y$ -function) where both functions are in terms of a third variable  $t$ . To create the parametric graph as  $t$  ranges from  $a$  to  $b$  you would use the command `ParametricPlot[ { $x$ -function,  $y$ -function}, { $t$ ,  $a$ ,  $b$ }]`. If you have several different parametric curves given by  $(x$ -function1,  $y$ -function1),  $(x$ -function2,  $y$ -function2), and so on, you can graph them all at once using `ParametricPlot[ { { $x$ -function1,  $y$ -function1}, { $x$ -function2,  $y$ -function2}, ... }, { $t$ ,  $a$ ,  $b$ }]` (notice how this parallels the format of the Plot command). So if you wanted to graph  $x = t^3 - 8t$ ,  $y = t^2$  as  $t$  goes from -3 to 3 you would use `ParametricPlot[ { $t^3 - 8t$ ,  $t^2$ }, { $t$ , -3, 3}]`:

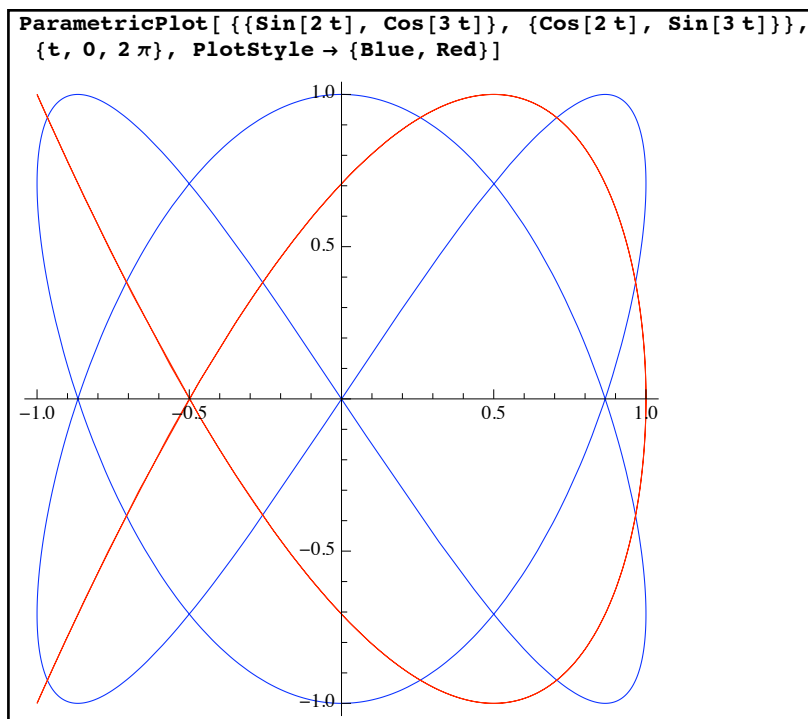


*a parametric graph*

Notice that as with the Plot command the scales on the axes are not to scale by default - this requires `AspectRatio→Automatic`. ParametricPlot will accept the same options and directives that Plot does - PlotStyle, PlotPoints, AxesLabel, PlotLegends, and so on. To graph both the parametric curves  $x = \sin(2t)$ ,  $y = \cos(3t)$  and  $x = \cos(2t)$ ,  $y = \sin(3t)$  you would use:

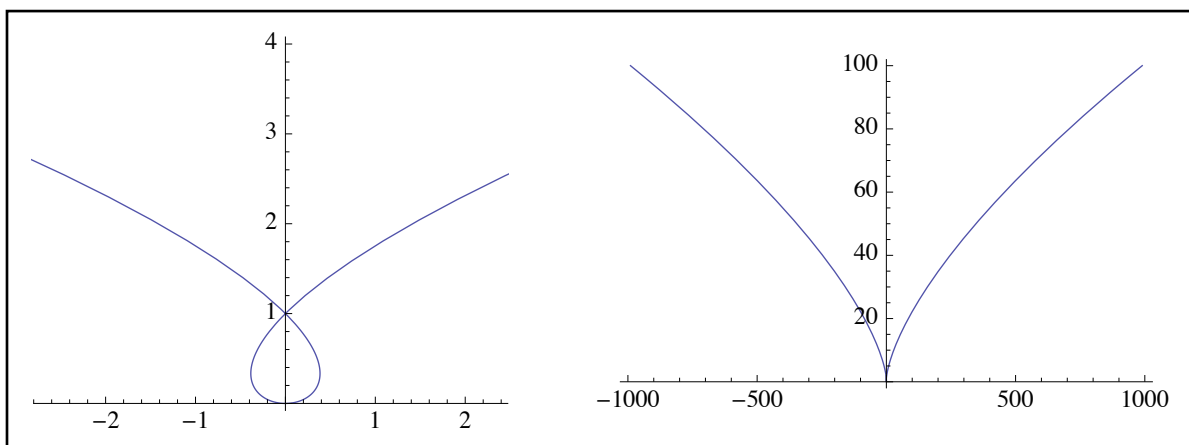


as  $t$  goes from 0 to  $2\pi$ , you could use `ParametricPlot[ {{Sin[2t], Cos[3t]}, {Cos[2t], Sin[3t]} }, {t,0,2Pi}, PlotStyle→{Blue, Red}]`



*two parametric graphs identified via PlotStyle*

Unlike `ContourPlot` the `ParametricPlot` command does not by default enable mouseover descriptions of the curves (although these can be added via `Tooltip` just as you can do in `Plot`). You will also need to take some care to choose a proper range of values for the variable  $t$ . For example if you graph  $x = t^3$ ,  $y = t^2$  from -2 to 2 and from -10 to 10 the pictures look quite different; the  $x$ -values get so large in the second graph that they crush out some of the detail at the origin:

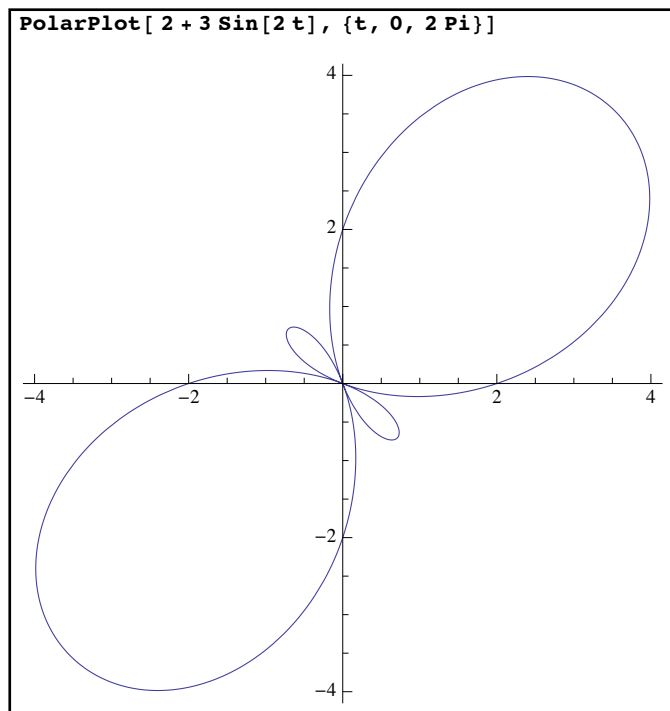


*the same parametric equations, two different ranges and appearances*

In the right graph the `AspectRatio` was set to `1/GoldenRatio` (which is the default for `Plot`) - otherwise the graphic would be very long and flat.

Another common form of graphing involves polar coordinates. In polar coordinates the coordinate pair  $(r, \theta)$  denotes the point  $r$  units out from the origin along a ray inclined at an angle of  $\theta$  with the positive  $x$ -axis (a negative value for  $r$  indicates you go backwards along the ray rather than forwards). So the Cartesian point  $(3,0)$  can be represented in polar coordinates as  $(3,0^\circ)$  and the point  $(0,-4)$  can be represented  $(4,270^\circ)$  (Mathematica will use radians rather than degrees for angular measure but hopefully this makes the idea clear). The same point can have many different polar coordinates -  $(0,-4)$  is both  $(4,270^\circ)$ ,  $(4, -90^\circ)$ , and  $(-4, 90^\circ)$ . For this reason it is fairly common for polar graphs of the form  $r = f(\theta)$  to have closed loops you don't see in the graphs of cartesian functions - you can go through a point once using one set of coordinates and loop back through it again using a different set of coordinates.

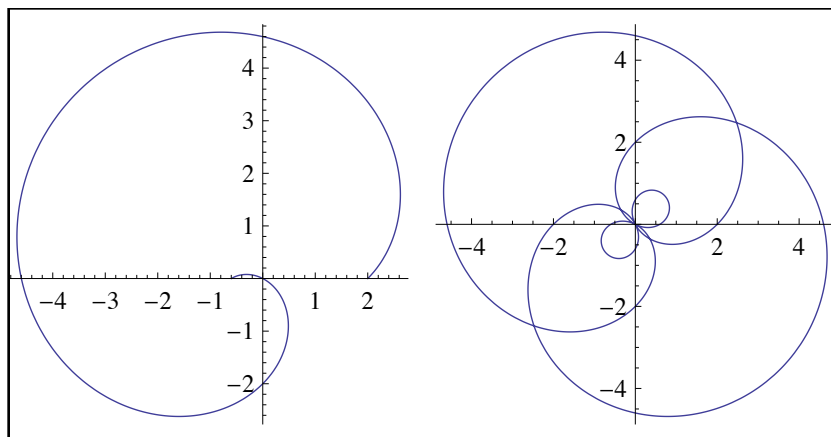
For graphing in polar coordinates the basic command is `PolarPlot` and it uses the basic format `PolarPlot[formula, {variable, start, finish}]`. For example to graph the polar equation  $r = 2 + 3 \sin(2\theta)$  from 0 to  $2\pi$  evaluate the command `PolarPlot[2+3Sin[2t], {t,0,2Pi}]` (while you can get  $\theta$  from a palette or as `Esc-theta-Esc` it's often easier to just use  $t$  in the command):



*the polar graph of  $r=2+3\sin(2\theta)$*

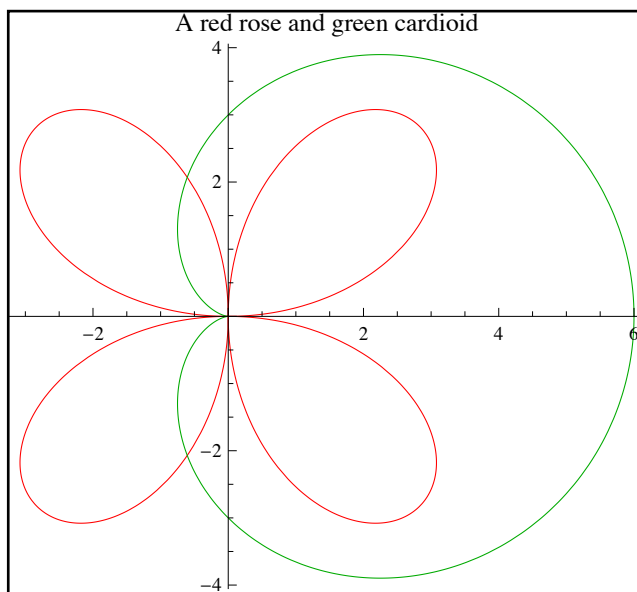
Just as in any graphing technique some care must be taken when choosing the range of angles to be graphed. As a rule of thumb you need to graph through a range which is multiple of  $2\pi$  and also a multiple of a period of all the functions involved. For example suppose you wanted to

graph the polar function  $r = 2 + 3 \sin(\frac{2\theta}{3})$ . As the formula has a period of  $3\pi$  you would choose a range of values for  $\theta$  which is a common multiple of  $2\pi$  and  $3\pi$ , namely  $6\pi$ . Graphing over a shorter range is likely to give you an incomplete graph - for example, here is  $r = 2 + 3 \sin(\frac{2\theta}{3})$  graphed first from 0 to  $2\pi$  and then from 0 to  $6\pi$ :



*graphing a polar curve over a longer range can make a big difference*

Just as in Plot and ParametricPlot you can graph several curves at once in PolarPlot (by using a list of formulas) and use many of the same styling options such as PlotStyle, PlotLabel, and so on. For example suppose you wanted to graph the “4-leaved rose”  $r = 4 \sin(2\theta)$  on the same axes as the “cardioid”  $r = 3 + 3 \cos(\theta)$ . By evaluating `PolarPlot[ {4Sin[2t], 3+Cos[t]}, {t, 0,2Pi}, PlotStyle→{Red, Darker[Green]}, PlotLabel→“A red rose and green cardioid”]` you would get the following plot:



*Graphing and styling multiple polar curves*

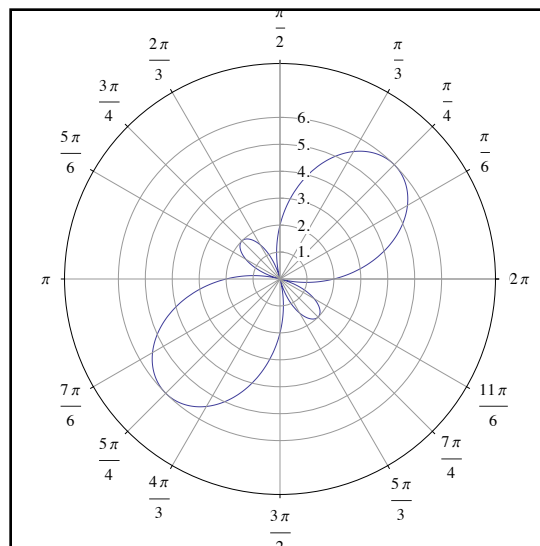
While PolarPlot accepts many of the options from Plot (PlotStyle, PlotPoints, PlotLegends, and so on) there are some options which are specific to PolarPlot which give you a better context into which to view polar graphs:

**PolarAxes:** The option `PolarAxes→True` removes the usual Cartesian axes and replaces them with a single axis on which to measure distance (the positive y-axis) and an enclosing circle with which to measure polar angles. By default the angular marks on the circle are in radian measure and go up by  $\pi/12$ .

**PolarTicks:** `PolarTicks→{anglelist, radii}` allows you to specify specific tick marks on the radial axis and specific angles on the enclosing circle. Both sets of tick marks must be lists (so a very sparse example is `PolarTicks→{ {0, Pi/2, Pi, 3Pi/2}, {1,2} }`) and `PolarAxes→True` must be specified for there to be any effect. If you want to get tick marks in degrees you need to multiply each value by the constant `Degree`. The `Range` command is very useful here - you could get a quick list of tick marks every 10 degrees using `Degree*Range[0,359,10]` to get the list of angles.

**PolarGridLines:** `PolarGridLines→{anglelist, radii}` will add gray rays that correspond to the angles given in *anglelist* and gray circles that correspond to the radii given in *radii*.

To see these options in action try graphing  $r = 2 + 4 \sin(2\theta)$  from 0 to  $2\pi$ . First define the quantity `polarticks={Union[ Range[ 0, 2Pi, Pi/6], Range[0, 2Pi, Pi/4] ], Range[6]}` (the common ticks values in radian measure are either multiples of  $\pi/6$  or  $\pi/4$  so the Union will combine both lists while removing the duplicated). You could then use the command `PolarPlot[ 2+4Sin[2t], {t,0,2Pi}, PolarAxes→True, PolarTicks→polarticks, PolarGridLines→polarticks]` to get:

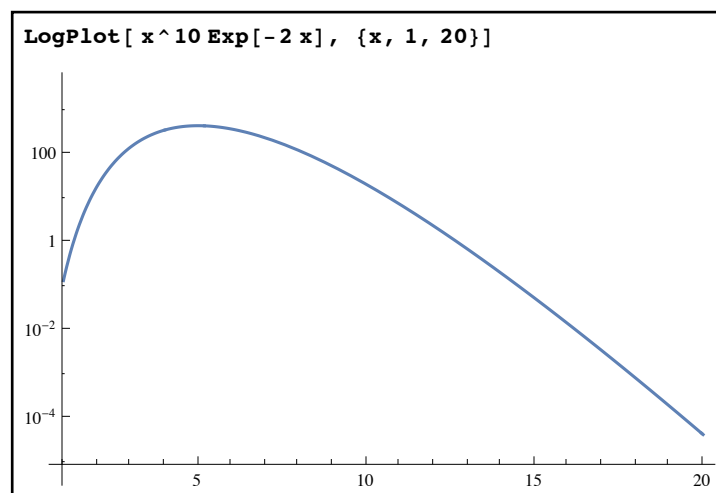


*a polar graph with a full set of polar axes and ticks*

The use of Union and Range to get a nice set of angular ticks marks is fairly common trick (one that was alluded to in an earlier homework problem) - basically in the definition of polarticks we get the multiples of  $\pi/4$  and  $\pi/6$  from 0 to  $2\pi$  and then combine them (which saves a lot of typing).

In addition to polar and parametric graphic Mathematica includes some twists on Cartesian graphing that are sometimes used in working with scientific data. These twists involve plotting the common logarithm of one or more variables instead of the variables themselves (like plotting  $(x, \log(y))$  instead of  $(x,y)$ ). This is typically done to reveal particular mathematical relationships in data. The Mathematica commands for these are LogPlot, LogLinearPlot, and LogLogPlot:

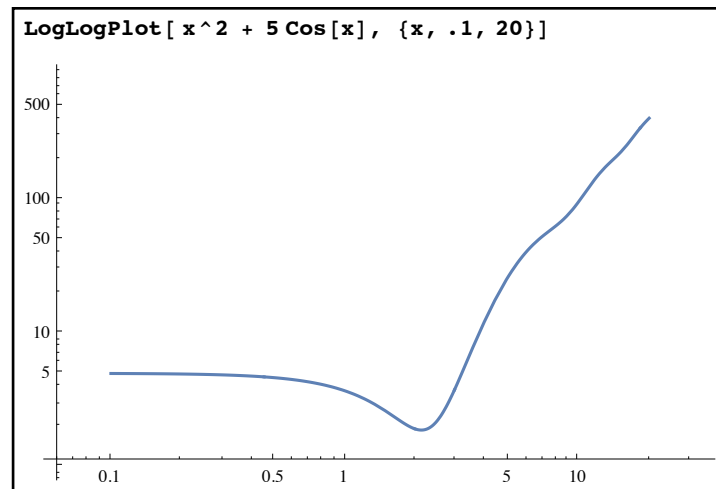
`LogPlot[function, {variable, start, finish}]`: LogPlot creates the “log(y) vs. x” graph for *function* as the *variable* goes from *start* to *finish*. So `LogPlot[ x^10 Exp[-2x], {x,1,20}]` essentially is the graph of  $y = \log(x^{10}e^{-2x})$  - except that the y-axis ticks are the original y values with logarithmic spaces:



*a graph of log(y) vs. x using LogPlot*

LogPlot is typically used when you are trying to see an exponential relationship between y and x - if  $y = a10^{bx}$  (where a and b are constants) then  $\log(y) = \log(a) + bx$  - so the graph of the function (or the plots of points if you are looking at data) would appear linear in LogPlot.

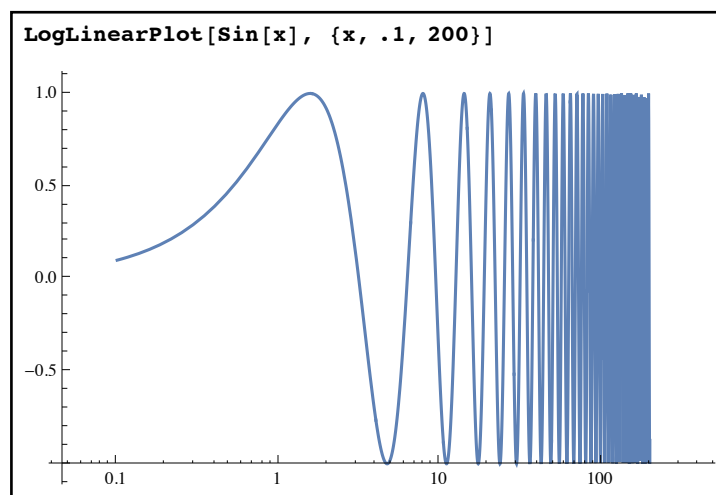
`LogLogPlot[function, {variable, start, finish}]`: LogLogPlot creates the “log(y) vs. log(x)” graph for *function* as the *variable* goes from *start* to *finish*. The value for *start* must be positive as you can only take the logarithm of positive numbers when you graph (the logarithms of negatives would be complex). So `LogLogPlot[ x^2 + 5 Cos[x], {x,.1,20}]` would create the “log-log” graph of  $y = x^2 + 5 \cos(x)$  from .1 to 20.



a “log-log” plot - note the logarithmic spacing on both axes

LogLogPlot is often used when you suspect that your function resembles a power of  $x$  over a given range - if  $y = cx^n$  then  $\log(y) = \log(c) + n \log(x)$  (and so the log-log graph would be a straight line of slope  $n$ ). Had the graph above continued to the right it would flatten out to become indistinguishable from a line of slope 2 (you could see that more easily using `AspectRatio→Automatic`) - the slope of 2 would indicate the formula was approximately that of  $x^2$  (as in the long run the cosine term becomes very small compared to the  $x^2$  term).

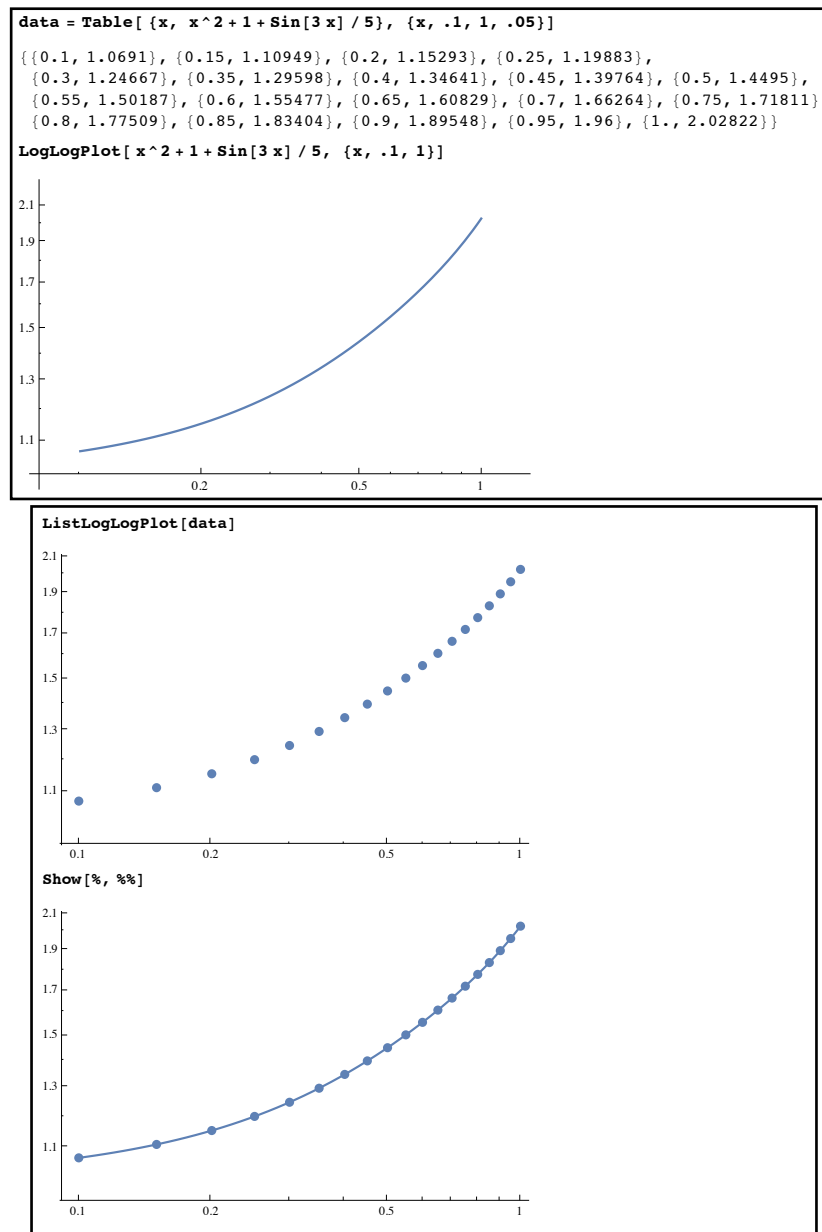
`LogLinearPlot[function, {variable, start, finish}]`: LogLinearPlot creates the “y vs.  $\log(x)$ ” graph for *function* as the *variable* goes from *start* to *finish*. As in LogLogPlot the *start* value must be positive as the graph will include the logarithm of this number:



a “log-linear” plot of  $\sin(x)$

LogLinearPlot is typically used when you expect  $x$  to be an exponential in  $y$  (which is the reverse of LogPlot) - if  $x = a10^{by}$  then  $\log(x) = \log(a) + by$ , or  $y = \frac{\log(x) - \log(a)}{b}$  - whose graph is a line.

All 3 log-plotting commands will accept most of the options from Plot. One issue is the use of Epilog to graph data alongside the curve (which is one of the big uses of these commands). Epilog won't properly graph the data points as they don't fit with the logarithmic scaling. To get around this there are logarithmic variations of ListPlot - ListLogPlot, ListLogLogPlot, and ListLogLinearPlot. To view the logarithmic data and curve together simply do both plots individually and then combine them with show:



*getting a “log-log” plot of data and a function together on the same axes*

### Section 3.7 Homework - Additional Graphing Commands

- 1) Sketch the parametric curve  $x = 2 \cos(t)$ ,  $y = 4 \sin(t)$ .
- 2) Sketch the parametric curve  $x = \sinh(t)$ ,  $y = \cosh(t)$  as  $t$  goes from -5 to 5 (the hyperbolic sine and cosine are represented by Sinh and Cosh).
- 3) Sketch the parametric curve  $x = 2 \cos(t) + 2.25 \cos(\frac{5t}{3})$ ,  $y = 2 \sin(t) - 2.25 \sin(\frac{5t}{3})$ .
- 4) Sketch the parametric curves  $x = 2 + \cos(t)$ ,  $y = \sin(t)$  and  $x = 3 - 2 \cos(2t)$ ,  $y = 1 + \sin(t)$  on the same axes. Make the first curve thick and blue, the second curve red and dashed, and include a legend.
- 5) Sketch the polar curve  $r = \sin(2\theta)$ .
- 6) Sketch the polar curves  $r = 2 + 3 \sin(\theta)$ ,  $r = 2 \cos(2\theta)$  on the same axes. Make the first curve red and the second curve dark green.
- 7) Sketch the polar curve  $r = \sin(\frac{\theta}{3}) + \cos(\frac{\theta}{4})$ . Make sure you graph a sufficient range of values to get the full graph.
- 8) If  $r = f(\theta)$  one way to determine what range of values for  $\theta$  use in graphing is to take advantage of FunctionPeriod via `FunctionPeriod[{f(θ), sin(θ)}, θ]` (including the sine makes sure that the fact the polar angle repeats every  $2\pi$  is taken into account). Use this trick to verify your graphing range for problem 7 and to graph  $r = \sin(\frac{2\theta}{3}) + \cos(\frac{2\theta}{5})$ .
- 9) Sketch the polar curve  $r = 3 \sin(3\theta) + 2$  using polar axes with angular tick marks every  $\pi/6$  and grid lines on the integers.
- 10) Create a LogPlot of the function  $e^{x/3} + x^2$  from 0 to 20. If you were presented with the graph and not the function how would you have seen that in the long run the function was essentially an exponential?
- 11) Create a LogLogPlot of the function  $x^3 + 50 \sin(x)$  from 1 to 30. If you were presented with the graph and not the function how would you have seen that in the long run the function was essentially a power of  $x$ ?
- 12) Create a LogLogPlot of the functions  $\sqrt{x}$ ,  $x^2$  and  $x^3$  with the option `AspectRatio→Automatic`. If you didn't have the formulas how could you estimate them from the graph?
- 13) Create the dataset `data13=Table[ {k, Sinh[k]}, {k,1,10,.4}]`. Graph this data using `ListLogPlot`, `ListLogLinearPlot`, and `ListLogLogPlot` and use the graphs to determine whether you think the Sinh function is essentially a power of  $x$ , an exponential function, or a logarithmic function.
- 14) Create the dataset `data14=Table[ {k, Log[1+1/k]}, {k,1,10,.4}]`. Graph this data using `ListLogPlot`, `ListLogLinearPlot`, and `ListLogLogPlot` and use the graphs to determine whether you think  $\ln(1 + \frac{1}{x})$  is essentially a power of  $x$ , an exponential function, or a logarithmic function.



- 15) Look up the Plot option `ScalingFunctions` and describe how it can be used to replicate the graph types of `LogPlot`, `LogLogPlot`, and `LogLinearPlot` using just the basic `Plot` command.

## Section 3.8 - Optimization

One of the most important applications in early calculus is optimization - finding the highest or lowest value for a function. Mathematica has several commands that allow you to maximize or minimize the value of a function without showing you the underlying calculations, letting you explore how these problems can be set up and what the solutions look like without having to master the underlying theory. These commands also work with problems that only allow integer solutions (which take the problems outside of calculus and into the branch of mathematics known as operations research).

The two basic commands for optimization are Maximize and Minimize. Maximize[*function*, *variable*] will try to find both the largest value for the *function* and a value for the *variable* at which it occurs (Minimize will do the same thing except it finds the lowest value). The results for both commands take the form of a list {*max or min*, *location of max or min as a replacement rule*}:

```
Minimize[ x^2 - 4 x + 1, x]
{-3, {x -> 2}}
```

```
Maximize[ Sin[x] + 2 Cos[x], x]
{2 Cos[2 ArcTan[2 - Sqrt[5]]] - Sin[2 ArcTan[2 - Sqrt[5]]], {x -> -2 ArcTan[2 - Sqrt[5]]}}
```

```
FullSimplify[%]
{Sqrt[5], {x -> -2 ArcTan[2 - Sqrt[5]]}}
```

```
Minimize[ x^4 - x^3 + 3 x^2 + 2 x + 1, x]
{1 + 2 Root[2 + 6 #1 - 3 #1^2 + 4 #1^3 &, 1] + 3 Root[2 + 6 #1 - 3 #1^2 + 4 #1^3 &, 1]^2 -
  Root[2 + 6 #1 - 3 #1^2 + 4 #1^3 &, 1]^3 + Root[2 + 6 #1 - 3 #1^2 + 4 #1^3 &, 1]^4,
  {x -> Root[2 + 6 #1 - 3 #1^2 + 4 #1^3 &, 1]}}
```

```
N[%]
{0.703298, {x -> -0.279651}}
```

```
Maximize[ x^3 - 3 x, x]
Maximize::natt : The maximum is not attained at any point satisfying the given constraints. >>
{Infinity, {x -> Infinity}}
```

*basic optimization problems in one variable*

Maximize and Minimize will work very well in problems involving algebraic expressions; they will work on some problems involving transcendental functions (like the one involving  $\sin(x) + 2 \cos(x)$  above) but not all of them. The results may be given in terms of Root objects but you can estimate these using the N command if need be. Maximize and Minimize will also

recognize problems where a maximum or minimum might not exist - in the last example above the values of  $x^3 - x$  get arbitrarily large as the values of  $x$  get big.

Maximize and Minimize also work on functions of more than one variable - in this case the format is simply `Maximize[function, list of variables]` (and Minimize works the same way):

```
Minimize[ x^2 + x y + y^2 + 5 x - y, {x, y}]
{- 31/3, {x -> - 11/3, y -> 7/3}}
```

```
Maximize[ (x + 2 y) / (x^2 + y^2 + 1), {x, y}]
{sqrt(5)/2, {x -> 1/sqrt(5), y -> 2/sqrt(5)}}
```

```
Maximize[ Cos[x - 3 y], {x, y}]
Maximize[Cos[x - 3 y], {x, y}]
```

```
Minimize[ x^2 + 3 x y + y^2, {x, y}]
Minimize::natt : The minimum is not attained at any point satisfying the given constraints. >>
{-infinity, {x -> Indeterminate, y -> Indeterminate}}
```

*basic optimization problems in more than one variable*

Multivariate optimization is typically much harder than optimization over a single variable - Mathematica fails to maximize the cosine even though we know the maximum value for cosine is 1. It is also much more common with multivariate functions for the maximum or minimum to fail to exist (as is the case in the last example).

In all of the basic optimization problems so far there have been no restriction on the variables - the values of  $x$  and  $y$  could be anything. Many optimization problems include restrictions on the allowed variables - they may need to be non-negative or maybe they represent the coordinates of a point drawn from a particular region like a square or circle. Maximize and Minimize allow you to impose constraints on the variable by using the form `Maximize[ {function, logical statement}, variable or list of variables]` (the same form works for Minimize). In this format the maximum or minimum will only come from those values of the variables for which the *logical statement* is True (if no such values exist you will get negative infinity from Maximize and infinity from Minimize). Typically the constraints are equations and inequalities linked by “and”. If the inequalities are not strict (i.e. use “less than or equal to” or “greater than or equal to”) the maximum or minimum will sometimes occur on the boundary (where they are “equal”). If the inequalities are strict (“less than” or “greater than”) the maximum and minimum will often not exist - in this case Maximize and Minimize may return a value but let you know the value really belongs on the excluded boundary:

```

Maximize[{x^3 - 12 x + 1, -3 ≤ x ≤ 1}, x]

{17, {x → -2}}

Maximize[{x^2, -1 ≤ x < 3}, x]

Maximize::wksol :
Warning: There is no maximum in the region in which the objective function is defined and the constraints
are satisfied; Mathematica will return a result on the boundary. >>

{9, {x → 3}}

Maximize[{x^2 + 3 x y - y^2, x^2 + y^2 ≤ 1}, {x, y}]

{
   $\frac{\sqrt{13}}{2}$ , {x → Root[9 - 52 #1^2 + 52 #1^4 &, 1], y → Root[9 - 52 #1^2 + 52 #1^4 &, 2]}
}

Minimize[{x + 3 y, x ≥ 0 && y ≥ 0 && x^2 + x y + y^2 + 3 x ≤ 1}, {x, y}]

{0, {x → 0, y → 0}}

Maximize[{x^3 + x y + y^2, x^2 + 4 y^2 == 4}, {x, y}]

{-Root[-11 830 787 - 49 644 634 #1 - 74 391 709 #1^2 -
  48 455 174 #1^3 - 11 042 659 #1^4 + 756 000 #1^5 + 186 624 #1^6 &, 1],
 {x → Root[16 - 20 #1^2 + 48 #1^3 - 139 #1^4 - 12 #1^5 + 36 #1^6 &, 4],
  y → Root[1 - 24 #1 + 139 #1^2 + 72 #1^3 - 283 #1^4 - 48 #1^5 + 144 #1^6 &, 4]}}

N[%]

{8.09037, {x → 1.99185, y → 0.090197}}

```

*constrained max/min problems, including a case where the maximum happens on an excluded boundary*

In some applications you might not need the exact value for a maximum or minimum - perhaps a numerical estimate will suffice. In these cases you can use the functions `NMaximize` and `NMinimize` - these use the same format as `Maximize` and `Minimize` but are often faster on more complex problems. If you need to increase the accuracy of the result you can use the option `WorkingPrecision` to get more decimal places:

```

NMaximize[{x^2 + 10 x y - 3 y^2, x^2 < y < 4 - x}, {x, y}]

{22.678, {x → 1.56155, y → 2.43845}}

NMinimize[{x^2 + 10 x y - 3 y^2, x^2 < y < 4 - x}, {x, y}, WorkingPrecision → 15]

{-290.678047194273, {x → -2.56155291879897, y → 6.56155322843190}}

```

*numerical estimates for maximum and minimum values*

Another tweak in some optimization problems is to require integer values for the variables. This kind of restriction often comes from the nature of the variables in a problem - if  $t$  is the number of airplane flights you can schedule then it wouldn't make sense for  $t$  to take on a fractional value. If all the variables represent integers the easiest way to do this is to add the domain `Integers` to the command at the end (in the same fashion that we do for `Solve` and `Reduce`). You can also enforce this restriction using the `Element` command to specify that all of

the variables are integers (Maximize and Minimize won't work if only some variables are integer and some are real - they all need to be the same type):

```
Maximize[{x + 3 y, x ≥ 0 && y ≥ 0 && x + y ≤ 10}, {x, y}, Integers]
{30, {x → 0, y → 10}}

Minimize[{x^2 + 3 y, x ≥ 0 && y ≥ 0 && 2 x + 3 y ≥ 100 && Element[{x, y}, Integers]}, {x, y}]
{100, {x → 1, y → 33}}
```

*some basic max/min problems over the integers*

Problems with integer constraints can be very difficult and time-consuming even for a computer - Mathematica will be able to do integer optimization as long as all of the constraints are linear and in many other cases where the set of solutions to check is not overly large. As the problems get larger and larger the time it takes to find the solution will likely increase rapidly so there are some practical limits on how big of a problem you can do. This isn't a Mathematica problem but inherent to the problems themselves - problems involving hundreds or thousands of variables can take years to solve even for supercomputers.

All of the examples we've seen so far have been "pure" max-min problems - they involve finding the maximum or minimum of a function we've already been given. We end this section with a few examples of "applied" max-min problems - ones where you have an application and need to derive the function to be maximized or minimized as well as any constraints.

Example 1: The height of a thrown ball

A ball is thrown upward from a height of 5 feet with a velocity of 24 feet per second. What is the maximum height of the ball and when does it reach this maximum?

The formula for the height of a thrown object (ignoring air resistance) is  $h = h_0 + v_0 t - \frac{1}{2} g t^2$ , where  $t$  is the time and  $g$  is the local gravitational constant. Using seconds and feet for the units the value for  $g$  is 32 feet per second squared. So we are looking to maximize  $h = 5 + 24t - 16t^2$  where  $t$  is the time in seconds. As the ball is thrown this formula only makes sense when  $t \geq 0$  and so we have a constrained maximization problem:

```
Maximize[{5 + 24 t - 16 t^2, t ≥ 0}, t]
{14, {t → 3/4}}
```

*finding the greatest height of a thrown ball*

So the maximum height is 14 feet and it occurs 3/4 of a second after being thrown.

Example 2: Building the largest possible garden

You have 300 feet of fencing to enclose a rectangular garden; you can use a long existing wall as one side. What is the largest garden you can build (in terms of area) and what are its dimensions?

If  $x$  is the length of the garden out from the existing wall and  $y$  is the width of the garden along the existing wall then the goal is to maximize the area  $xy$ . The total fence used is  $2x + y$  and as  $x$  and  $y$  represent lengths they cannot be negative. As you only have 300 feet of fence to use we have the constraint  $2x + y \leq 300$ :

```
Maximize[{x y, x ≥ 0 && y ≥ 0 && 2 x + y ≤ 300}, {x, y}]
{11 250, {x → 75, y → 150}}
```

*the largest area garden*

So the largest garden has an area of 11,250 square feet and is 75 feet out from the existing wall and 150 feet along it.

Example 3: A more general garden

You have  $f$  feet of fencing to enclose a rectangular garden; you can use a long existing wall as one side. What is the largest garden you can build (in terms of area) and what are its dimensions?

The only difference between this problem and the previous one is we don't have a specific value for the amount of fencing we have to use. So the previous constraint  $2x + y \leq 300$  becomes  $2x + y \leq f$ :

```
Maximize[{x y, x ≥ 0 && y ≥ 0 && 2 x + y ≤ f}, {x, y}]
{ { { 0      f == 0
      { f^2/8  f > 0
      -∞      True
    , {x → { 0      f == 0
              { f/4  f > 0
              Indeterminate True
    , y → { 0      f == 0
            { f/2  f > 0
            Indeterminate True
  } } }
Simplify[%, f > 0]
{ { f^2/8, {x → f/4, y → f/2} }
```

*the maximum area of a more general garden*

In this case the result is originally given as a piecewise function - we didn't specify that the length  $f$  had to be positive and so we get different cases depending on whether  $f$  is 0, positive, or negative. We can pick out the case we want by simplifying the answer using the assumption that  $f > 0$  (or we could have included that in the constraint statement) - the largest garden will occur

when you use a quarter of the fencing to build out from the wall on each side and the remaining half of the fencing as the remaining side.

#### Example 4: Building an open-topped box

You need to build a box (with no top) from a 30" by 50" piece of cardboard. The easiest way to do this is to remove squares of equal sizes from the four corners and then fold up the remaining flaps. What is the largest box (in terms of volume) you can make this way and what are its dimensions?

Let  $x$  be the side length of the squares you will be removing from each corner. Then the depth of the box will be  $x$ , the width will be  $30 - 2x$ , and the length will be  $50 - 2x$  (you lose  $x$  inches on either side of both the width and length when you remove the squares). So the volume of the box is  $\text{depth} \times \text{width} \times \text{length} = x(30 - 2x)(50 - 2x)$ . The side length  $x$  will need to be at least 0 but no larger than 15 (the smaller edge of the original cardboard is 30" and you are removing  $x$  from each side):

```

Maximize[{ x (30 - 2 x) (50 - 2 x), 0 ≤ x ≤ 15}, x]

{ 20/3 (8 - √19) (-25 + 5/3 (8 - √19)) (-15 + 5/3 (8 - √19)), {x → 5/3 (8 - √19)} }

Simplify[%]

{ 1000/27 (28 + 19 √19), {x → -5/3 (-8 + √19)} }

{x, 30 - 2 x, 50 - 2 x} /. %[[2]]

{-5/3 (-8 + √19), 30 + 10/3 (-8 + √19), 50 + 10/3 (-8 + √19)}

Simplify[%]

{-5/3 (-8 + √19), 10/3 (1 + √19), 10/3 (7 + √19)}

N[%]

{6.0685, 17.863, 37.863}

```

*the largest volume box you can build from a 30" by 50" sheet of cardboard*

You can generalize this problem to an arbitrary piece of cardboard as well; if the dimensions of the original sheet are  $a$ " by  $b$ " (with  $a < b$ ) then the volume would be given by  $x(a - 2x)(b - 2x)$  and you would use the constraints  $0 \leq x \leq \frac{a}{2}$  and  $0 \leq a \leq b$ .

#### Example 5: Designing the cheapest can

You need to design a can with a volume of 100 cubic inches. The metal that makes up the side of the can costs 2 cents per square inch. The metal in the can's top and bottom needs to be a bit thicker and costs 4 cents per square inch. It also costs 1/4 cent per inch to weld the top and bottom to the side to form the can. What is the cheapest can you can build with these requirements and what are its (approximate) dimensions?

The can is a cylinder of radius  $r$  and height  $h$ . The top and bottom of the can will be circles of radius  $r$  so the total area of the top and bottom will be  $2\pi r^2$ . The side of the can is a rectangle of height  $h$  and width  $2\pi r$  (think of slitting a can open and then laying its side flat - the side would be a rectangle which is as long as the circumference of the top circle). The weld length is the total circumference of the top and bottom - as these are circles of radius  $r$  the total weld length is  $4\pi r$ . So the total costs of the can is (cost of top and bottom)+(cost of side)+(welding cost), or  $4(2\pi r^2) + 2(2\pi r h) + \frac{1}{4}(4\pi r) = 8\pi r^2 + 4\pi r h + \pi r$ . We need the volume of the can to be 100 cubic inches - as the volume of a cylinder is  $\pi r^2 h$  this becomes the condition  $\pi r^2 h = 100$ . The values of  $r$  and  $h$  need to be positive or the volume will be zero, and this gives us the following Minimize command:

```
Minimize[{ 8 Pi r^2 + 4 Pi r h + Pi r, r > 0 && h > 0 && Pi r^2 h == 100}, {r, h}]
{Root[-276480000 Pi - 1600 Pi^2 - 57600 Pi #1 + Pi #1^2 + 32 #1^3 &, 1],
 {r -> Root[-400 + Root[-276480000 Pi - 1600 Pi^2 - 57600 Pi #1 + Pi #1^2 + 32 #1^3 &, 1] #1 - Pi #1^2 - 8 Pi #1^3 &,
 2], h -> 100 / (Pi Root[-400 + Root[-276480000 Pi - 1600 Pi^2 - 57600 Pi #1 + Pi #1^2 + 32 #1^3 &, 1] #1 -
  Pi #1^2 - 8 Pi #1^3 &, 2]^2)}}
N[%]
{306.77, {r -> 1.97586, h -> 8.15342}}
```

*building the cheapest can with a volume of 100 cubic inches*

So the cheapest possible can will cost roughly \$3.07 and be about 8.16" tall and about 1.98" in radius.

#### Example 5: Formulating the cheapest mixture

Your company has been given an order for 1000 pounds of animal feed. The customer specifications are that the feed has to be at least 4% fat, 15% fiber, and 15% protein. You have 5 ingredients that you can use: wheat midds, rice feed, cotton meal, soy beans, and alfalfa. Each of these has a fat percentage, fiber percentage, protein percentage, and cost given in the following table:



Ingredient	Fat %	Fiber %	Protein %	Cost per pound
Wheat mids	3	5	15	0.05
Rice feed	6	30	5	0.02
Cotton meal	3	10	40	0.10
Soy beans	2	35	10	0.05
Alfalfa	4	25	20	0.08

How should you mix the feed order to be as cheap as possible?

Let  $w$ ,  $r$ ,  $c$ ,  $s$ , and  $a$  be the number of pounds you use for each type of ingredient. The order is for 1000 pounds so  $w + r + c + s + a = 1000$ . The final mixture needs to have at least 40 pounds of fat, 150 pounds of fiber, and 150 pounds of protein. The fat in the mixture is  $.03w + .06r + .03c + .02s + .04a$ , so  $.03w + .06r + .03c + .02s + .04a \geq 40$ . The fiber requirement will be  $.05w + .3r + .1c + .35s + .25a \geq 150$  and the protein requirement will be  $.15w + .05r + .4c + .1s + .2a \geq 150$ . The total cost of the mixture is  $.05w + .02r + .1c + .05s + .08a$ . Using the decimal numbers in our Minimize command will give us approximate numbers in the result - the easiest way around this would be to multiply all the constraints by 100 to remove the decimals. But in an actual business application the estimates would probably be fine so we can leave them in:

```
Minimize[ {.05 w + .02 r + .1 c + .05 s + .08 a, w ≥ 0 && r ≥ 0 && c ≥ 0 && s ≥ 0 && a ≥ 0 &&
.03 w + .06 r + .03 c + .02 s + .04 a ≥ 40 && .05 w + .3 r + .1 c + .35 s + .25 a ≥ 150 &&
.15 w + .05 r + .4 c + .1 s + .2 a ≥ 150 && w + r + c + s + a == 1000}, {w, r, c, s, a}]
{42.8571, {w → 0., r → 714.286, c → 285.714, s → 0., a → 0.}}
```

*finding the cheapest feed mixture*

So the cheapest feed mixture would cost about \$42.86 per 1000 pounds and consist of about 714.3 pounds of rice feed and 285.7 pounds of cotton meal.

#### Example 6: Work scheduling

You are running a factory and need to schedule your workforce. Due to union rules each worker has to work “5 on, 2 off” (that is has to work 5 straight days followed by 2 days off in the long run). From past experience you need to have at least 80 people work on Sunday, 110 on Monday, 100 on Tuesday, 105 on Wednesday, 112 on Thursday, 120 on Friday, and 90 on Saturday. How should you schedule people each day to minimize the total number of workers (and therefore your payroll costs)?

Let  $a$  be the number of people who start on Sunday,  $b$  the number of people who start on Monday,  $c$  the the number of people who start on Tuesday, and so on (through  $g$  for people who start their “5 on” on Saturday). Then your goal is to minimize  $a + b + c + d + e + f + g$ . As you need at least 80 people on Sunday  $a + d + e + f + g \geq 80$  (the people who start their week on Monday and Tuesday are off on Sunday so there are no  $b$  or  $c$  terms included). The Monday constraint becomes  $b + e + f + g + a \geq 110$ , the Tuesday constraint becomes  $c + f + g + a + b \geq 100$ , and so on. Each of the variables must be non-negative and has to be an integer (you can’t hire a quarter person) - so we will need to use the Integers domain:

```
Minimize[ {a+b+c+d+e+f+g,
  a ≥ 0 && b ≥ 0 && c ≥ 0 && d ≥ 0 && e ≥ 0 && f ≥ 0 && g ≥ 0 && a+d+e+f+g ≥ 80 &&
  b+e+f+g+a ≥ 110 && c+f+g+a+b ≥ 100 && d+g+a+b+c ≥ 105 && e+a+b+c+d ≥ 112 &&
  f+b+c+d+e ≥ 120 && g+c+d+e+f ≥ 90}, {a, b, c, d, e, f, g}, Integers]
{144, {a → 7, b → 47, c → 17, d → 17, e → 24, f → 15, g → 17}}
```

*figuring out the minimum number of workers to staff a factory*

So the minimum number of people you need to hire is 144; 7 start on Sunday, 47 on Monday, 17 on Tuesday, 17 on Wednesday, 24 on Thursday, 15 on Friday, and 17 on Saturday. This gives you 3 extra people working on Tuesday but it is the best you can do with the “5 on 2 off” constraint.

There are many ways to vary this problem - you could change the number of hours needed each day or allow people to work overtime (in which case you would introduce extra variables to represent overtime and give those variables a coefficient larger than 1 in the formula to be minimized to represent their greater cost) or add part-time workers to the problem (which would be cheaper in the function to be minimized but not contribute as effectively - maybe they cost 60% of a full time worker, can only work 6 hours a day in a “4 on 3 off” fashion, and only count for half as much as a full time worker).

## Section 3.8 Homework - Optimization

- 1) Find the maximum and value of  $x^3 - 2x^2 - 3x + 1$  as  $x$  goes from -2 to 3. Use Plot and Epilog to graph the function along with its maximum and minimum.
- 2) Find the maximum value of  $x^2 - 2x$  as  $x$  goes from -1 to 3. Graph the function on this interval and compare the graph to the maximum you found.
- 3) Find the maximum value of  $x^2 e^{-x}$  when  $x \geq 0$ .
- 4) What is the difference between finding the maximum value of  $x^2 - 4x$  on the interval  $0 \leq x \leq 5$  and the interval  $0 < x < 5$ ?
- 5) Find the maximum and minimum value of  $x + 2y$  where  $(x^2 + y^2)^2 = x^2 - y^2$  (give the answer exactly and as an approximation). Use ContourPlot and Epilog to find the graph of  $(x^2 + y^2)^2 = x^2 - y^2$  along the maximum point (in blue) and the minimum point (in red).

- 6) Find the maximum and minimum value of  $x^2 - 3xy$  where  $0 \leq x + y \leq 10$  and  $x^2 + y^2 \leq 49$  (give the answer exactly and as an approximation). Use RegionPlot and Epilog to find the graphs of  $0 \leq x + y \leq 10$  and  $x^2 + y^2 \leq 49$  along the maximum point (in blue) and the minimum point (in red).
- 7) Find the maximum and minimum values of the function  $\frac{1}{x^2 + x + y^2 + 1}$ .
- 8) Find the maximum and minimum values of  $xy - z$  on the sphere  $x^2 + y^2 + z^2 = 4$ .
- 9) Find the largest area rectangle which can be inscribed in the ellipse  $\frac{x^2}{4} + y^2 = 1$ .
- 10) You are going to sell 2 different kinds of plates - fancy and plain. The fancy plates cost 12 dollars to make and sell for 30 while the plain plates cost 5 dollars to make and sell for 18. Because of time constraints you can make no more than 300 plates per week and only have 2500 dollars per week to spend on materials and labor. How many of each plate should you make to maximize your profit?
- 11) An oil company owns two refineries. Refinery A can produce 20 barrels of gas and 25 barrels of fuel oil per day and costs \$300 a day to operate. Refinery B can produce 40 barrels of gas and 20 barrels of fuel oil per day but costs \$500 a day to operate. You need to produce at least 1000 barrels of gas and 800 barrels of fuel oil. How many days should you run each refinery to minimize costs?
- 12) You are standing 20 feet from the water on a beach and throw a frisbee. The frisbee lands 100 feet down the beach and 30 feet into the water. If you can walk at 3 feet per second and swim at 1 foot per second how should you travel to minimize the time it will take to reach the frisbee? Give your answer as an approximation. (suggestion: draw a picture and let  $x$  be how far down the beach you will cross into the water - your path will then be the hypotenuses of two right triangles, one where you walk and one where you swim)
- 13) Repeat example 6 without the constraint that the number of employees be integers. Verify that the integer solution to example 6 is not a rounding of the real number solution. This is important in practice as integer-based problems are much harder to solve than real number-based problems. If you could always round the real number solutions to get the integer solutions you would just work with real number-based problems instead - but sadly this is not the case.
- 14) Repeat example 6 with the following addition: you may hire part-time workers in addition to full-time workers. Part-time workers must work the same "5 on 2 off" pattern, perform half the work of a full-time worker, and cost 75% as much to hire. How should you assign full-time and part-time workers to minimize your costs?
- 15) Repeat problem 14 but assume the part-time workers only cost 60% of the full-time workers. How should you assign full-time and part-time workers to minimize your costs? (common sense would indicate this would increase the use of part-time workers)
- 16) Repeat problem 14 but assume the part-time workers only cost 49% of the full-time workers. How should you assign full-time and part-time workers to minimize your costs? (common sense would indicate this would shift the workforce to entirely part-time workers)

- 17) Repeat example 5 assuming the cost of wheat mids is only .02 per pound.
- 18) In example 5 no wheat mids were used in the optimal solution; in problem 16 some wheat mids. So it stands to reason there is a price for wheat mids somewhere between .02 and .05 per pound where they go from being cheap enough to use to being too expensive to use. Find this value to the nearest .001 by replacing the cost of wheat kids with a variable  $k$  and then wrapping a Manipulate around the Minimize command, where in the manipulation the value of  $k$  is on a slider going from 0 to .05 in steps of .001.
- 19) Set up a Manipulate command to generalize example 5 to any costs for the 5 ingredients. The best way to do this is to replace each of the cost values .05, .02, .1, .05, and .08 with its own variable and have each of those variables controlled by an input field (such as {{wheatcost, .05, "Wheat mid cost:"}}).

## Section 3.9 - Logic in Mathematica

We have already seen some basic elements of logic within Mathematica - functions that end in Q (like IntegerQ) have the boolean outputs True and False, the interpretation of equations and inequalities as logical statements in ContourPlot, RegionPlot, and Simplify, and in the formatting of conditional functions like If and Which. Mathematica has commands which will let you go beyond these basic ideas and explore more complex logical ideas and their applications. We will take a look at these commands in three groups - logical connectives, logical computations, and quantified statements.

We've already seen the three most basic logical connectives when working with RegionPlot and Reduce - “and” (represented by &&), “or” (the inclusive disjunction, represented by ||), and “not” (represented by !). Each of these representations is actually shorthand for a Mathematica command which can be represented using either standard Mathematica notation or by its usual logical symbol:

`And[statement1, statement2,...]`: And represents the statement which is True when all of its component statements are true and False otherwise (that is when any of the component statements are False). We have used && as the “and” connector previously but you get can the traditional symbol  $\wedge$  using the key combination Esc-and-Esc (i.e. the Escape key followed by and followed by the Escape key).

```
And[ True, False, True, True]
False
True  $\wedge$  False  $\wedge$  True  $\wedge$  True
False
 $x^2 == 4 \wedge x > 0 /. \{x \rightarrow 1\}, \{x \rightarrow 2\}$ 
{False, True}
```

*the general And statement along with its standard notation*

`Or[statement1, statement2,...]`: Or represents the “inclusive disjunction” statement which is True when at least one of its component statements is True and False otherwise (i.e. when they are all False). We have used the || notation for Or in the past but you can also gets its standard notation  $\vee$  using the key combination Esc-or-Esc.

```

Or[ False, True, True, False]

True

False  $\vee$  True  $\vee$  True  $\vee$  False

True

 $x^2 = 4 \vee x > 0$  /. {{x  $\rightarrow$  1}, {x  $\rightarrow$  2}, {x  $\rightarrow$  -3}}

{True, True, False}

```

*the Or statement along with its standard notation*

Not[statement]: Not represents logical negation - it returns True when *statement* is False and False when *statement* is True. We have used ! for not in the past but you can get its standard notation  $\neg$  using the key combination Esc-not-Esc.

```

Not[True]

False

! True

False

 $\neg$  True

False

 $\neg (-2 \leq x \leq 2)$  /. {{x  $\rightarrow$  1}, {x  $\rightarrow$  3}}

{False, True}

```

*negation using Not and its standard notation*

In addition to the three basic connectives Mathematica adds a few other commonly used connectives using Implies, Equivalent, Xor, and BooleanCountingFunction:

Implies[statement1, statement2]: Implies represents the basic “if-then” conditional, the assertion that *statement1*’s truth guarantees *statement2*’s truth. If you don’t want to use the Implies command notation you can get the standard conditional symbol  $\Rightarrow$  using Esc- $\Rightarrow$ -Esc.

```

Implies[ True, False]
False
Implies[ False, False]
True
True  $\Rightarrow$  False
False
(p  $\wedge$  q)  $\Rightarrow$  p  $\vee$  q
p & q  $\Rightarrow$  p || q
Simplify[%]
True

```

*If-Then conditionals in Mathematica using Implies and standard notation - and checking that “p and q” is True always implies “p or q” is True*

**Equivalent[statement1, statement2,...]**: Equivalent is Mathematica’s version of the biconditional statement - it is True when all the component statements have the same value and False otherwise. You can get the standard biconditional symbol  $\Leftrightarrow$  using Esc-equiv-Esc (the sequence Esc-<=>-Esc is a similar symbol but doesn’t function as a Mathematica command).

```

Equivalent[ True, True, False, True]
False
Equivalent[ False, False, False]
True
True  $\Leftrightarrow$  False
False
Simplify[ Equivalent[p  $\Rightarrow$  q,  $\neg$  p  $\vee$  q]]
True

```

*“if and only if” in Mathematica, including checking that an If-Then statement is always the same as a particular combination of Not and Or*

**Xor[statement1,statement2,...]**: Xor is the “exclusive or” statement. If you have just two statements **Xor[statement1, statement2]** is True when the statements have the opposite value and False when they have the same value. If you have more than two statements Xor will return True when an odd number of the statements are True and False when an even number are True. You can get the Xor symbol as Esc-xor-Esc.

```

Xor[True, False]
True
Xor[True, True, False, False]
False
Xor[ True, True, False, True]
True
True  $\vee$  False  $\vee$  True
False

```

*the “exclusive or” along with its standard notation*

**BooleanCountingFunction:** BooleanCountingFunction represents a function whose outputs are True and False depending on how many of its inputs are True. It has two different forms for the applications “at most this many are True” and “exactly this many are True”. BooleanCountingFunction[*truenumber*, *numberofvariables*] represents a function which is True when up to *truenumber* of its *numberofvariables* are True. So BooleanCountingFunction[2,5] represents a function with 5 inputs and is True when up to 2 of them are True and False if more than 2 are True. BooleanCountingFunction[{*truenumber*}, *numberofvariables*] represents a function which is True when exactly *truenumber* of its *numberofvariables* are True. So BooleanCountingFunction[{2},5] would represent a function with 5 inputs and is True when exactly 2 of them are True and False if 0, 1, 3, 4, or 5 of them are True. While you can use BooleanCountingFunction directly (as in BooleanCountingFunction[{1},3][p,q,r]) it is often cumbersome to do so due to the extra brackets. In many cases it might be easier to define a new function using a format like f:=BooleanCountingFunction[{1},3] and then just using the new name (so f[True, False, False] would be True). BooleanCountingFunction[{1}, *n*] is what many people would incorrectly guess Xor to represent with more variables - the statement which is True when exactly 1 of the inputs is True.

```

BooleanCountingFunction[2, 5][True, False, False, False, False]
True
BooleanCountingFunction[{2}, 5][True, False, False, False, False]
False

```

*BooleanCountingFunction in its “raw” form. The first statement is the “up to 2 True” version and the second is the “exactly 2 are True” version*



```

UpToTwoOfThree := BooleanCountingFunction[ 2, 3]
UpToTwoOfThree[True, False, False]

True

TwoOfThree := BooleanCountingFunction[{2}, 3]
TwoOfThree[False, True, True]

True

TwoOfThree[True, False, False]

False

```

*using function notation and descriptive names with BooleanCountingFunction*

**TraditionalForm[expression]:** TraditionalForm rewrites *expression* using standard mathematical notation - so it will convert expressions that use &&, ||, etc. into the forms you can get using the keystroke combinations. It won't insert extra parentheses as it assumes you know the logical order of operations. TraditionalForm works on a wide range of non-logic expressions as well (it will write polynomials in standard “high to low power” form, etc.). TraditionalForm acts as a “wrapper” expression just as TableForm and MatrixForm do.

```

In[57]:= TraditionalForm[ Implies[ p || q, Equivalent[ p && q, Xor[p, q] ] ] ]
Out[57]/TraditionalForm=

$$p \vee q \Rightarrow p \wedge q \Leftrightarrow p \underline{\vee} q$$


In[56]:= TraditionalForm[ Sin[ 1 + x + x^2 ] / x^3 == Sqrt[ Log[ 3 x] Exp[x] ] ]
Out[56]/TraditionalForm=

$$\frac{\sin(x^2 + x + 1)}{x^3} = \sqrt{e^x \log(3x)}$$


```

*using TraditionalForm to get standard mathematical expressions*

In addition to the basic logical connectives Mathematica has many commands for routine logical computations:

**LogicalExpand[expression]:** LogicalExpand will use any of the distributive laws (like DeMorgan's Laws) to “multiply out” a logical expression into several simpler pieces. LogicalExpand will work with “hidden” logical connectives like the “and” implicit in double inequalities.

```

In[63]:= LogicalExpand[ p ∧ (q || ¬ r) ]
Out[63]= (p && q) || (p && ! r)

In[64]:= TraditionalForm[%]
Out[64]//TraditionalForm=

$$(p \wedge q) \vee (p \wedge \neg r)$$


In[69]:= LogicalExpand[ ! (y > 1 || -1 ≤ x < 2) ]
Out[69]= (-1 > x && y ≤ 1) || (x ≥ 2 && y ≤ 1)

In[70]:= TraditionalForm[%]
Out[70]//TraditionalForm=

$$(-1 > x \wedge y \leq 1) \vee (x \geq 2 \wedge y \leq 1)$$


```

*breaking down complicated expressions using LogicalExpand*

**BooleanConvert[expression]:** BooleanConvert takes the *expression* and converts it to “disjunctive normal form” - a form defined by a string of cases joined by Or statements. BooleanConvert[expression, “form”] (where *form* is drawn from several pre-defined names) can convert a statement into a variety of lesser-used forms - you can find a list of forms in the documentation for BooleanConvert.

```

BooleanConvert[Implies[p && q, r]]
! p || ! q || r

BooleanConvert[ BooleanCountingFunction[ {2}, 3][x, y, z] ]
(x && y && ! z) || (x && ! y && z) || (! x && y && z)

BooleanConvert[ BooleanCountingFunction[ 2, 4][x, y, z, t] ]
(! t && ! x) || (! t && ! y) || (! t && ! z) || (! x && ! y) || (! x && ! z) || (! y && ! z)

```

*BooleanConvert transforms a logical statement into a list of “or” cases*

**TautologyQ[expression]:** TautologyQ returns True if *expression* is True for all True/False values of the underlying variables and False otherwise. The expression must be a boolean expression - it can’t involve equalities or inequalities.

TautologyQ[ Equivalent[ expression1, expression2]] would return True if the two expressions are logically equivalent and False otherwise.

```

TautologyQ[ p || ! p]
True

TautologyQ[ p && ! q]
False

TautologyQ[ Equivalent[ ! (p ∨ q), ¬ p ∧ ¬ q]]
True

```

*using TautologyQ to see if statements are always True and if two statements are logically equivalent*

**SatisfiableQ[*expression*]**: SatisfiableQ returns True if the *expression* is True for at least once for some combination of underlying variables and False if *expression* is always False. So SatisfiableQ is essentially the claim “the expression is not a contradiction”. Like TautologyQ this function will only work on boolean expressions.

<b>SatisfiableQ[ <math>p \ \&amp;\&amp; \ (q \    \ r)</math> ]</b> True <b>SatisfiableQ[ <math>p \ \wedge \ \neg \ p</math> ]</b> False
---

using SatisfiableQ to determine if statements can be True at least some of the time

**FindInstance[*expression*, *variable* or *variable list*, *domain*]**: FindInstance tries to find an example (drawn from the *domain*) where *expression* is True. Common choices for the domains are Complexes, Reals, Integers, and Booleans.

FindInstance[*expression*, *variable* or *variable list*, *domain*, *number*] tries to find *number* examples instead of just one. FindInstance works best on logical statements that involve algebraic expressions.

<b>FindInstance[ <math>x + y \geq 3 \wedge x \geq 0 \wedge y \leq 0 \wedge 3x + y \leq 20</math>, {<i>x</i>, <i>y</i>}, Integers]</b> {{ <i>x</i> → 4, <i>y</i> → -1}} <b>FindInstance[ <math>x + y \geq 3 \wedge x \geq 0 \wedge y \leq 0 \wedge 3x + y \leq 20</math>, {<i>x</i>, <i>y</i>}, Integers, 4]</b> {{ <i>x</i> → 7, <i>y</i> → -4}, { <i>x</i> → 6, <i>y</i> → -2}, { <i>x</i> → 3, <i>y</i> → 0}, { <i>x</i> → 6, <i>y</i> → -1}} <b>FindInstance[ <math>p \ \&amp;\&amp; \ \text{Implies}[q, \neg r]</math>, {<i>p</i>, <i>q</i>, <i>r</i>}, Booleans, 4]</b> {{ <i>p</i> → True, <i>q</i> → True, <i>r</i> → False}, { <i>p</i> → True, <i>q</i> → False, <i>r</i> → True}, { <i>p</i> → True, <i>q</i> → False, <i>r</i> → False}} <b>FindInstance[ <math>x^2 / y^2 = 4</math>, {<i>x</i>, <i>y</i>}, Integers, 3]</b> {{ <i>x</i> → -76, <i>y</i> → 38}, { <i>x</i> → -2, <i>y</i> → -1}, { <i>x</i> → 6, <i>y</i> → -3}} <b>FindInstance[ <math>x^2 / y^2 = 2</math>, {<i>x</i>, <i>y</i>}, Integers]</b> {}
---

using FindInstance to find examples of where things are True. The last equation has no integer solutions (as the square root of 2 is irrational)

**BooleanTable[ *expression* or *list of expressions*, *variable* or *list of variables* ]**:

BooleanTable makes a table of the True/False values of the boolean *expression*/ *list of expressions* for all the True/False values of the underlying *variable* or *list of variables*. BooleanTable by itself is not very useful but when combined with TableForm and TableHeadings it can easily create complicated “truth tables” for an expression or list of expressions. You do this by defining the list of expressions to start with the basic variables involved, then the intermediate

components of your final statement, and finishing up with the final statement. You can then use the list both in BooleanTable and as the list of column headings for TableHeadings.

```
In[124]:= BooleanTable[ p && q, {p, q}]
Out[124]= {True, False, False, False}

In[125]:= BooleanTable[ p && (q || ! r), {p, q, r}]
Out[125]= {True, True, False, True, False, False, False, False}

In[126]:= sequence = {p, q, r, ! r, q || ! r, p && (q || ! r)};
headings = Map[TraditionalForm, sequence]
Out[127]= {p, q, r, ¬ r, q ∨ ¬ r, p ∧ (q ∨ ¬ r)}

In[128]:= TableForm[ BooleanTable[ sequence, {p, q, r}], TableHeadings → {None, headings} ]
Out[128]//TableForm=
```

$p$	$q$	$r$	$\neg r$	$q \vee \neg r$	$p \wedge (q \vee \neg r)$
True	True	True	False	True	True
True	True	False	True	True	True
True	False	True	False	False	False
True	False	False	True	True	True
False	True	True	False	True	False
False	True	False	True	True	False
False	False	True	False	False	False
False	False	False	True	True	False

*BooleanTable and its use in creating a full nicely-formatted truth table*

Version 5 of Mathematica added the capability to work with “quantified” logical statements - the mathematical notions of “for all” and “there exists”. These are handled by the commands ForAll and Exists. These commands work in conjunction with FullSimply, Reduce, FindInstance, and a new command Resolve (which tries to eliminate any quantified statements):

**ForAll:** ForAll represents the assertion that something is always True and it has a few different formatting possibilities depending on how you want to use it. ForAll[ *variable*, *expression*] and ForAll[ *variable list*, *expression*] are the most basic - they would be True when the *expression* is True for all values of the variable or variables. In many applications you may need to restrict the variables somehow (such as requiring that they be positive) - you achieve this by using ForAll[ *variable/variable list*, *condition*, *expression*]. These can also be entered by using the key combination Esc-fa-Esc to get the standard ForAll symbol and then putting the variable/variable list and conditions in a subscript (which you can get via the key combination Ctrl+\_ ). A common condition to use is to restrict the variables to a particular domain - you can do this either using Element or the “element of” symbol  $\in$  (via the key combination) Esc-elem-Esc.

```

ForAll[x, Element[x, Reals], x^2 > 9]

 $\forall_{x, x \in \text{Reals}} x^2 > 9$ 

FullSimplify[%]

False

ForAll[x, x > 2, x^2 > 4]

 $\forall_{x, x > 2} x^2 > 4$ 

FullSimplify[%]

True

ForAll[ {x, y}, x^2 + y^2 ≤ 1 , Abs[x] ≤ 1 ∧ Abs[y] ≤ 1]

 $\forall_{\{x, y\}, x^2 + y^2 \leq 1} (\text{Abs}[x] \leq 1 \ \&\& \ \text{Abs}[y] \leq 1)$ 

FullSimplify[%]

True

```

*quantified statements using ForAll*

The last statement reduces to True as the unit circle is a subset of the square whose opposite corners are (1,1) and (-1,-1).

These quantified statements all reduced to True or False when simplified - in many cases the truth or falsehood might depend on additional factors (like the value of another variable) in which case FullSimplify would process the statement as far as it could and then return it.

**Exists:** Exists represents the assertion that something is True at least once (i.e. it is not always False). The formatting for Exists parallels that of ForAll: *Exists[ variable or variable list, expression]* claims that the *expression* is True at least once for some value of the variable(s) and *Exists[variable or variable list, condition, expression]* does the same but restricts the values of the variable or variables to where *condition* is True. The “exists” symbol is created by the key sequence Esc-ex-Esc and as in ForAll the variables and conditions need to be put in a subscript. If Exists returns True then you should be able to use FindInstance to get an example of where the statement is true.

```

Exists[ {x, y, z}, {x, y, z} ∈ Integers ∧ x > 0 ∧ y > 0 ∧ z > 0, x^2 + y^2 == z^2]

 $\exists_{\{x, y, z\}, (x|y|z) \in \text{Integers} \ \&\& \ x > 0 \ \&\& \ y > 0 \ \&\& \ z > 0} x^2 + y^2 == z^2$ 

FullSimplify[%]

True

FindInstance[ (x^2 + y^2 == z^2) ∧ x > 0 ∧ y > 0 ∧ z > 0, {x, y, z}, Integers]

{{x → 8, y → 6, z → 10}}

Exists[ x, x ∈ Reals, x^2 + 2 x - 1 == 0]

 $\exists_{x, x \in \text{Reals}} -1 + 2x + x^2 == 0$ 

```

```

FullSimplify[%]
True
Exists[ x, x ∈ Reals, x^2 + 2 x + 5 == 0]
 $\exists_{x, x \in \text{Reals}} 5 + 2x + x^2 == 0$ 
FullSimplify[%]
False
Exists[ {x, y}, Element[ {x, y}, Reals], x^2 + y^2 == 9 && x + y == 2]
 $\exists_{\{x, y\}, (x|y) \in \text{Reals}} (x^2 + y^2 == 9 \ \&\& \ x + y == 2)$ 
FullSimplify[%]
True

```

*using the Exists statement to see if there are Pythagorean triples and if there are real solutions to an equation or system of equations*

All of these statements happened to quickly reduce to either True or False - FullSimplify was able to determine if a quantity existed or a general law was obeyed without needing to get additional information. The Resolve command is meant to deal with more general situations:

Resolve: Resolve works to remove ForAll and Exists statements from a logical expression, either by reducing it to True or False or converting it to a simpler condition in another variable that does not involve quantified statements. Resolve[*expression*] tries to determine if the *expression* is simply True or False. Resolve[*expression*, *variable or list of variables*] tries to determine what condition on the *variable* or *list of variables* will make the expression True. Resolve[*expression*, *variable or list of variables*, *domain*] tries to determine what condition on the *variable* or *list of variables* will make the expression True under the assumption that the values for *variable* or *list of variables* must be from the *domain* (common domains are Reals, Complexes, Integers, and Booleans). Resolve works best when working with algebraic equations and inequalities.

Here are a few examples of how Resolve can be used in a number of situations:

Example 1: When does a quadratic have a real root?

The general quadratic equation  $ax^2 + bx + c = 0$  (where  $a$ ,  $b$ , and  $c$  are real and  $a$  is not 0) does not always have real roots. What condition on  $a$ ,  $b$ , and  $c$  will make sure there are real roots?

```

In[40]:= Resolve[ Exists[ x, a x^2 + b x + c == 0] && a != 0, {a, b, c}, Reals]

Out[40]=  $\left( a < 0 \ \&\& \ c \geq \frac{b^2}{4a} \right) \mid \mid \left( a > 0 \ \&\& \ c \leq \frac{b^2}{4a} \right)$ 

In[41]:= TraditionalForm[FullSimplify[%]]
Out[41]//TraditionalForm=

$$a \neq 0 \bigwedge 4ac \leq b^2$$


```

*the condition under which a quadratic will have a real root*

Example 2: When does a cubic have all real roots?

The simplified cubic equation  $x^3 + bx + c = 0$  ( $b$  and  $c$  real) either has 3 real roots or 1 real root and 2 non-real roots. What condition on  $b$  and  $c$  will yield 3 real roots?

As we know the cubic has either 1 or 3 real roots all we need to check is that it has two real roots  $x$  and  $y$  which are different:

```

In[43]:= Resolve[ Exists[ {x, y}, Element[{x, y}, Reals],
    x^3 + b x + c == 0 && y^3 + b y + c == 0 && x != y], {a, b, c}, Reals]

Out[43]=  $b < 0 \ \&\& \ -\frac{2\sqrt{-b^3}}{3\sqrt{3}} \leq c \leq \frac{2\sqrt{-b^3}}{3\sqrt{3}}$ 

In[44]:= TraditionalForm[%]
Out[44]//TraditionalForm=

$$b < 0 \bigwedge -\frac{2\sqrt{-b^3}}{3\sqrt{3}} \leq c \leq \frac{2\sqrt{-b^3}}{3\sqrt{3}}$$


```

*the condition under which  $x^3 + bx + c = 0$  has three real roots*

Example 3: Is a region bounded?

A region  $R$  is said to be bounded if it lies inside a disk centered at the origin of some finite radius. Is the region  $x^2 + 3xy + 4y^2 \leq 10$  bounded?

The region being bounded is equivalent to there being a value  $m$  for which  $(x, y)$  being a solution to  $x^2 + 3xy + 4y^2 \leq 10$  implies that it is also a solution to  $x^2 + y^2 \leq m^2$ :

```

Resolve[ Exists[m, m ∈ Reals,
    ForAll[ {x, y}, {x, y} ∈ Reals, x^2 + 3 x y + 4 y^2 ≤ 10 => x^2 + y^2 ≤ m^2 ] ] ]
True

```

*the region is bounded as a value for  $m$  exists*

We can do even better in this case and see what values for the disc radius  $m$  will enclose the entire region - we simply take out the “Exists” statement and Resolve for the variable  $m$  over the reals:

```
Resolve[ForAll[{x, y}, {x, y} ∈ Reals,
  Implies[x^2 + 3 x y + 4 y^2 ≤ 10, x^2 + y^2 ≤ m^2]] && m > 0, m, Reals]
m ≥ Root[400 - 200 #1^2 + 7 #1^4 &, 4]
N[%]
m ≥ 5.13883
```

*the entire region would be encased in a circle of radius of about 5.14 centered at the origin*

We can even find what range of  $x$  and  $y$  values are used in the region - the region includes an  $x$ -value if there is a corresponding  $y$ -value which makes the region inequality True; likewise a  $y$ -value will be in the region if there is a corresponding  $x$ -value which makes the region inequality True:

```
Resolve[Exists[y, y ∈ Reals, x^2 + 3 x y + 4 y^2 ≤ 10], x, Reals]
-4 √(10/7) ≤ x ≤ 4 √(10/7)
N[%]
-4.78091 ≤ x ≤ 4.78091
Resolve[Exists[x, x ∈ Reals, x^2 + 3 x y + 4 y^2 ≤ 10], y, Reals]
-2 √(10/7) ≤ y ≤ 2 √(10/7)
N[%]
-2.39046 ≤ y ≤ 2.39046
```

*x- and y-ranges for the region*

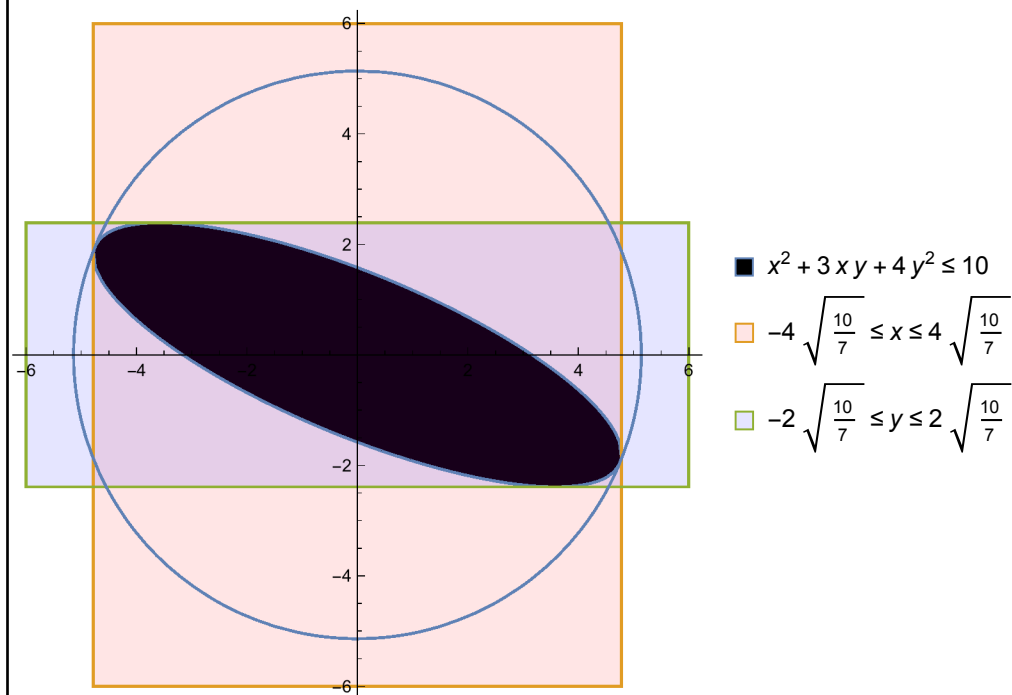
We can put this all together graphically by using RegionPlot to graph the full region in black and the  $x$ - and  $y$ -ranges in mostly-transparent colors and then using Show to combine the RegionPlot with the ContourPlot of the bounding circle:



```

regiongraphs = RegionPlot[ { x^2 + 3 x y + 4 y^2 ≤ 10,
  -4 √(10/7) ≤ x ≤ 4 √(10/7), -2 √(10/7) ≤ y ≤ 2 √(10/7} , {x, -6, 6}, {y, -6, 6},
  PlotStyle → {Black, Directive[ Red, Opacity[.1]], Directive[Blue, Opacity[.1]]},
  PlotLegends → "Expressions"];
boundingdisk = ContourPlot[ x^2 + y^2 == 5.13883^2, {x, -6, 6}, {y, -6, 6}];
Show[ regiongraphs, boundingdisk,
  Axes → True, Frame → False, AspectRatio → Automatic]

```



*the graph of the region, its x- and y-ranges, and bounding circle*

#### Example 4: Finding right triangle side lengths

Is it possible to find a right triangle with rational side lengths whose hypotenuse has a length of  $3/2$ ?

At first glance the Exists command doesn't seem to work well with the requirement that the side lengths be rational - Rationals isn't one of the domains we listed for Resolve. We can get around this by treating a single rational number as the ratio of two integers and using the Integers domain instead. We can then check if those ratios work in the Pythagorean Theorem:

```

Exists[{x, y, a, b}, {x, y, a, b} ∈ Integers,
  x > 0 && y > 0 && a > 0 && b > 0 && (x / y) ^ 2 + (a / b) ^ 2 == 9 / 4]


$$\exists_{\{x,y,a,b\}, (x|y|a|b) \in \text{Integers}} \left( x > 0 \ \&\& \ y > 0 \ \&\& \ a > 0 \ \&\& \ b > 0 \ \&\& \ \frac{x^2}{y^2} + \frac{a^2}{b^2} == \frac{9}{4} \right)$$


Resolve[%]

True

FindInstance[ x > 0 && y > 0 && a > 0 && b > 0 && (x / y) ^ 2 + (a / b) ^ 2 == 9 / 4,
  {x, y, a, b}, Integers]

{{x → 9, y → 10, a → 6, b → 5}}

```

*a right triangle with sides 9/10 and 6/5 will have a hypotenuse of length 3/2*

If you change the hypotenuse length 3/2 to other rational numbers (which would change the square 9/4 to other numbers) Mathematica may be unable to verify if solutions exist - the Resolve command would then just parrot the same Exists statement back at you.

#### Example 5: Arbitrarily large real numbers

A common pair of questions when first working with quantified statements is “Is there a largest real number?” and “Are there arbitrarily large real numbers?”. The first asks is there a single number  $x$  which is larger than every number  $y$ ; the second asks if for every number  $y$  there is a larger number  $x$ . We can easily set these up in Resolve:

```

Resolve[ Exists[x, ForAll[y, x ≥ y] ], Reals]

False

Resolve[ ForAll[y, Exists[x, x ≥ y] ], Reals]

True

```

*there is no single largest number but given any number there's always a larger one*

These statements and others like them are used to show that when combining “for all” and “there exist” statements the order can make a big difference.

## Section 3.8 Homework - Logic in Mathematica

- 1) In your own words explain the difference between the Or and Xor commands.
- 2) Explain the difference between the commands `BooleanCountingFunction[k,n]` and `BooleanCountingFunction[{k},n]`.
- 3) Use `TautologyQ` to show that the logical statement  $(p \wedge q) \Rightarrow r$  is not always True. Use `FindInstance` to find values of  $p, q$ , and  $r$  for which the statement is False. (for the latter negate the statement and use the domain `Booleans`).

- 4) Use TautologyQ to show the logical statement  $(p \wedge q) \Rightarrow (p \vee q)$  is always True.
- 5) Repeat problems 3 and 4 but instead of TautologyQ and FindInstance use BooleanTable and TableForm to create truth tables instead.
- 6) Use FullSimplify and Equivalent to determine if the statements  $\neg(\neg p \wedge q)$  and  $p \vee \neg q$  are logically equivalent (use FullSimplify[ Equivalent[... - if you get True as the result they are equivalent.]). Use BooleanTable and TableForm to check your answers.
- 7) Determine if the logical statements  $p \Rightarrow (q \vee r)$  and  $(p \wedge \neg q) \Rightarrow r$  are logically equivalent by any means.
- 8) Use BooleanTable and TableForm to create a truth table for  $p \wedge (\neg q \Rightarrow (r \Leftrightarrow s))$ , making sure all the required subexpressions are included in the table.
- 9) Use BooleanTable and TableForm to create a truth table for the expressions BooleanCountingFunction[2,4][x,y,z,t] and BooleanCountingFunction[{2},4][x,y,z,t].
- 10) Determine if the Xor function is associative or not (that is if Xor[a,Xor[b,c]] and Xor[Xor[a,b],c] are logically equivalent).
- 11) Use LogicalExpand to distribute the “not” in  $\neg(a \vee b \vee c)$ . Use TraditionalForm to put the answer in standard form.
- 12) Use BooleanConvert to convert BooleanCountingFunction[{3},5][a,b,c,d,e] into a string of cases where the statement is True. Use TraditionalForm to put the answer in standard form.
- 13) One of the drawbacks of TraditionalForm is that it does not stop computations in expressions it is applied to (to see this try looking at TraditionalForm[2^27^5] and TraditionalForm[Tan[Pi/3]]). One way around this is to use the command HoldForm, which stops all evaluations in what it is applied to. Another way is to use the command Inactive, which “turns off” functions it is applied to. Evaluate the commands TraditionalForm[ HoldForm[ 2^2 7^5 ] ], TraditionalForm[ HoldForm[Tan[Pi/3]]==Tan[Pi/3]], and TraditionalForm[ Inactive[Tan][Pi/3]==Tan[Pi/3] ] to see how these work.
- 14) Use FindInstance to find 5 examples of real numbers  $x$  for which  $x^2 + x + 1 < 4$ .
- 15) Use FindInstance to find 3 real points on the curve  $y^2 = x^3 + 2x + 1$ . Try to find 3 integer points on the curve.
- 16) Use FindInstance to find an example of a rectangle with area 100 and perimeter of 60.
- 17) Use Resolve to find when a quadratic equation  $ax^2 + bx + c = 0$  has exactly 1 real root.
- 18) Use Resolve to show that if  $(x,y)$  is a solution to  $x^2 + y^2 \leq 1$  then it is also a solution to  $xy \leq 1$  (that is, the first region is a subset of the second one). Verify this using RegionPlot.
- 19) Use the method of example 3 to show the region  $x^4 + xy + y^4 \leq 20$  is bounded. Estimate the smallest radius  $m$  for which the region is inside the disk of radius  $m$  centered at the origin.
- 20) Use the method of example 4 to estimate the  $x$ - and  $y$ - ranges of points inside the region  $x^4 + xy + y^4 \leq 20$ .

- 21) Find the slopes  $m$  of lines through  $(2,2)$  which are tangent to the curve  $y = x^2$ . To do this we can use the fact that any line through  $(2,2)$  can cross the parabola at most twice. So the tangent slope  $m$  would have the property that there is a point  $(x,y)$  which satisfies  $y = x^2$  and  $y = m(x - 2) + 2$  and if  $(x_1, y_1)$  and  $(x_2, y_2)$  are both solutions to the two equations then  $(x_1, y_1) = (x_2, y_2)$ . (Note: This approach would work in general for conic sections but not for more complicated curves). Graph the lines and parabola together to verify your answer.

```
CreateHand[5]
{{4, Clubs}, {10, Clubs}, {Ace, Hearts}, {10, Diamonds}, {Jack, Hearts}}

FlushQ[%]
False

CreateHand[3]
{{Queen, Diamonds}, {9, Diamonds}, {Ace, Diamonds}}

FlushQ[%]
True
```

## Section 4.1 - An Introduction

Despite the fact that there are thousands of pre-defined commands in Mathematica it is inevitable that you will want to do some calculation or create some sort of graphic that is simply not covered by the standard commands in the program. No matter how large any computer program is (Mathematica or otherwise) this will always happen. One of the strengths of Mathematica is that you can write programs in it to extend its functionality and fill in the gaps that almost everyone who uses it regularly will find from time to time. In this chapter we will talk about some of the elements that can be used to write these new Mathematica programs. We will not assume anything more than routine computer knowledge in our discussion although those students with some knowledge of programming will no doubt see much that is familiar, particularly if they have used any of the C-based languages.

A program in Mathematica works like any other command – that is it has a name and uses square brackets to enclose its “inputs”. So if I want to define a function to plot any trigonometric function in  $x$  from 0 to  $2\pi$ , the command to call that program might look something like `TrigPlot[ Sin[x], x]` when I use it. You do not have to follow the Mathematica convention of giving functions names that start with capital letters but it is generally recommended that you do so to make it more consistent with the Mathematica commands that people are familiar with.

So what is a Mathematica program? At its most basic level a Mathematica program is a set of stored commands which are linked together and turn a particular type or group of inputs into something else. This is no different than the other Mathematica commands we have discussed before. If you think about it, what does the command `Plot[ mess, {var , start, finish }]` do? It takes two inputs (the first a function and the second a list of 3 objects, starting with a variable and ending with two numbers) and outputs a picture. This is the essence of a Mathematica program - a set of inputs which then give rise to an output. This idea (input associated to output) should sound very familiar to mathematics students. It is the idea behind what we normally call “functions”. In fact all Mathematica programs are functions.

This is something of an oversimplification of course but in fact most Mathematica programs are simply very complex functions. This means that technically we already know how to write simple Mathematica programs using the techniques of Chapter 2. Recall that in Mathematica a function like  $f(x)=x^2$  is represented by `f[x_]:= x^2`, where the underscore lets Mathematica know we are talking about an “input variable” and the `:=` stands for “evaluate every time from scratch as” (i.e. delayed evaluation). For such simple functions we could omit the `:` sign and just write `f[x]=x^2`. But as most programs that we write won’t just execute a formula but rather use other Mathematica commands we will almost always need to use the `:=` notation for functions/programs to ensure the commands are evaluated in full every time. In addition we also know how to “undefine” or remove a program from the computer’s memory. In earlier chapters we did this through the use of `=.` to clear a definition. It is more common for programs to use the command `Clear`. `Clear[program name]` will remove the program with the given name

and all of its values from memory. It is very common to clear the function before defining it just to make sure there are no conflicts, as follows:

```
Clear[f]
f[x_] := x^2
```

*“clearing” a function name before defining it in order to prevent conflicts*

Clear is a very important command to use before defining a program for one very practical reason – very rarely do programmers get a program perfect on the first attempt. Usually there is a bit of trial and error involved, if only to get rid of typographical errors. By using Clear in the same cell right before the program definition each time you evaluate the program definition in the trial-and-error process your program will start from scratch, avoiding all sorts of conflicts. There is a similar command called Exit. Exit[ ] (with empty brackets) terminates the Mathematica kernel, removing all definitions. Some people who do a series of programs or problems in one notebook will use Exit at the end of each “section” to prevent conflicts, but since Exit wipes all definitions in many cases it is overkill.

There is another recommendation that will aid you greatly in program writing, particularly if you are new to it: never just sit down and write the program on the fly. It is usually helpful to do some scratch work first to see what you will need and help you logically order the various bits and pieces of the program. This scratch work usually involves three categories: “things the program will need to run/inputs”, “things I will have to compute or refer to in the program”, and “the main steps in the program”. It is pretty common to have to go back and add things to the “things the program needs” and “things I need to compute or refer to” as you write down the list of the main steps. For example in a program to compute the angle between two lines, in the formula for the angle (which would be a “step”), you will suddenly realize you need the slopes of the lines (a “thing I need to compute”). And in computing the slopes you will most likely need as “inputs” the equations of the lines (or at least the points they run through). Once you have all this scratch work down on paper it will be a lot easier to sit down and write the program in fairly simple steps that are logically ordered, avoiding code that kind of jumps back and forth to reflect “stream of consciousness” thinking (this is sometimes called “spaghetti code”, a bane of computer science instructors everywhere as it can be very hard to follow). The few examples of programs in this section won’t require this full-blown approach because they are fairly short but in later sections thinking things out ahead of time will make writing your programs a bit easier (and in some cases will take less time).

Before getting down to a few simple programs there are two new commands which will be of use: Print and Return. Print is fairly straight forward – it prints whatever is inside the brackets. If x has been defined to be 3, Print[x^2] will print the number 9 onscreen. Note that it will only print the number 9 as a visual display, not give you the value 9 – you won’t be able to use the 9 in future computations (for example, following the Print command with %+5 will not give you 14, but rather “5+Null”). If you want to print normal text enclose it in quotes the same

way as you would for the Plot options AxesLabel and PlotLabel. If you have to print more than item per line simply separate the different items by commas, as in the following examples:

```
Print["The square of 3 is ", 3^2]
Print[ x^2 (x - 1) ,
      " when multiplied out is ",
      Expand[ x^2 (x - 1) ] ]
Print[ y^3]
Print[% + 4]

The square of 3 is 9

(-1 + x) x^2 when multiplied out is -x^2 + x^3
y^3
4 + Null
```

*examples of the Print command*

Notice that if you follow text by another object in a Print statement (such as a number) you should put an extra space at the end of the text in the quotes for everything to line up correctly. The same sort of rule applies if text is following a previous part of the Print statement - you can see an example in the string “ when multiplied out ” in the examples above.

The simplest use of the command `Return[item]` is to tell Mathematica that *item* is the final result of the program/computation and to cease work, with *item* returned as the result. If Mathematica sees `Return` without anything inside the brackets it will simply end evaluating your program without returning anything as the answer. For simple one-line programs `Return` is rarely needed but as we begin to write more complex multi-part programs in the sections that follow it will be necessary to tell Mathematica which of the values we are computing is supposed to be the result of the program.

Here are a few examples of simple one-line programs along with some explanation of how they were written. Each program is posed as a math problem together with a list of inputs.

#### Example 1: Finding the largest prime divisor

Input: An integer  $n$

Output: The largest prime number which divides  $n$ .

Discussion: To find the largest prime factor we need the prime factorization of  $n$ , which is computed using the `FactorInteger` command. `FactorInteger` returns the factorization as  $\{\{prime1, exponent1\}, \{prime2, exponent2\} \dots\}$ , where the primes are given in ascending order. So we need the first element of the last sublist. So we could define the program in a single shot as `LargestPrimeDivisor[n_Integer]:=FactorInteger[n][[-1,1]]`



```

LargestPrimeDivisor[n_] :=
  FactorInteger[n][[-1, 1]]

LargestPrimeDivisor[10]
5

LargestPrimeDivisor[2^60 - 7^6]
351 931

LargestPrimeDivisor[8.9]
FactorInteger::exact :
  Argument 8.9 in FactorInteger[8.9] is not an exact number. >>
Part::partd :
  Part specification FactorInteger[8.9][[-1, 1]] is longer
  than depth of object. >>
FactorInteger[8.9][[-1, 1]]

```

*LargestPrimeDivisor definition and some outputs*

Note that the program breaks down when we try to use it on the value 8.9. This makes sense - 8.9 isn't an integer so prime factorization doesn't apply. This kind of problem (where the input needs to be a certain type for computations to make sense) happens a lot in mathematical programming. While you can handle this by wrapping the main line in an If command (say as If[IntegerQ[n], FactorInteger[n][[-1,1]], Indeterminate] ) there's a slicker way to tell the program "only work this way if the input has this type" - object "heads". The head of an object in Mathematica is its general type. Common heads are Integer, Real, Symbol, Rational, Equal, Graphics, and Graphics3D. You can require that inputs to have a certain head by including the required head after the underscore in the function definition (so *n\_Integer* or *p\_Graphics*). Adding this to our LargestPrimeDivisor command looks like this:

```

Clear[LargestPrimeDivisor]
LargestPrimeDivisor[n_Integer] :=
  FactorInteger[n][[-1, 1]]

LargestPrimeDivisor[10]
5

LargestPrimeDivisor[8.9]
LargestPrimeDivisor[8.9]

```

*a version of LargestPrimeDivisor that only works on integer inputs*

This time when we try to find the largest prime divisor of 8.9 Mathematica simply parrots the command back at us - it knows that LargestPrimeDivisor only works on integers.

Example 2: A “fixed” of the inverse cotangent

Input: A real number  $x$ .

Output: The quadrant I or II angle (in  $(0, \pi)$ ) whose cotangent is  $x$ .

Discussion: For real numbers  $x$  recall that the Mathematica definition of ArcCot[ $x$ ] is essentially the angle in  $(-\pi/2, \pi/2)$  whose cotangent is  $x$  (this makes some calculus easier but violates some trigonometric identities). In ArcCot[ $x$ ] positive values for  $x$  give angles in Quadrant I, which agree with the standard definition. For negative values of  $x$  the returned angle in ArcCot[ $x$ ] is in Quadrant IV (in the interval  $(-\pi/2, 0)$ ) – these must be fixed to the corresponding Quadrant II value which can be done by adding  $\pi$  radians to the ArcCot value. So we could use FixedArcCot[ $x$ ]:= ArcCot[ $x$ ] + If[ $x < 0$ , Pi, 0]:

```
FixedArcCot[x_] := ArcCot[x] + If[x < 0, Pi, 0]

FixedArcCot[Sqrt[3]]

 $\frac{\pi}{6}$ 

FixedArcCot[-Sqrt[3]]

 $\frac{5\pi}{6}$ 

ArcCot[-Sqrt[3]]

 $-\frac{\pi}{6}$ 
```

*FixedArcCot output*

Unfortunately we cannot use the head Real on the variable  $x$  here as Real refers to an approximate real number. So using  $x\_Real$  would render our function unable to process exact numbers like  $\pi$  or Sqrt[3].

Example 3: Slope between 2 points

Inputs: Two points, given as { $a, b$ } and { $c, d$ }.

Output: The slope of the line between the 2 points.

Discussion: The only tricky parts here are how the points should be entered and a check for an indeterminate slope. This last part can be done either by a If or a Which command. Slope[ { $a, b$ }, { $c, d$ }] := If[  $a == c$ , Indeterminate,  $(d-b)/(c-a)$ ]

```

Slope[{a_, b_}, {c_, d_}] := If[a == c, Indeterminate, (d - b) / (c - a)]

Slope[ {5, 4}, {7, -5}]


$$-\frac{9}{2}$$


Slope[ {2, 1}, {2, 5}]

Indeterminate

Slope[ {2, x}, {5, y}]


$$\frac{1}{3} (-x + y)$$


```

*the Slope function*

The value Indeterminate is used by Mathematica to represent 0/0 and related expressions. Note that the structure of the inputs here is very specific - Slope will only work if the inputs have the precise form 2-element list 1 and 2-element list 2. Any other structure for the inputs and Mathematica would just parrot the entry back at us.

Example 4: Slope between 2 points, version 2

New Discussion: In this version we will assume the inputs are simply 2 lists and then pick the coordinates out through the [[]] notation. The mechanics of the computation are the same, only how we are treating the inputs is different:

```

Clear[Slope]
Slope[point1_, point2_] := If[point1[[1]] == point2[[1]], Indeterminate, (point2[[2]] - point1[[2]]) / (point2[[1]] - point1[[1]])]

```

```

Clear[Slope]
Slope[point1_, point2_] := If[point1[[1]] == point2[[1]],
  Indeterminate, (point2[[2]] - point1[[2]]) / (point2[[1]] - point1[[1]])]

Slope[ {5, 4}, {7, -5}]


$$-\frac{9}{2}$$


Slope[ {2, 1}, {2, 5}]

Indeterminate

```

*a new version of the Slope function*

This second version of the Slope command will give results identical to the first if the same  $\{a,b\}, \{c,d\}$ -style inputs are entered. Why would you use the second version if the first is shorter? Notice that in the second version we were able to use simple descriptive input names, which can make it easier to follow the logical flow of a program. In addition what would happen in the two versions if we evaluated `Slope[{1,2,3},{4,5,6}]`? The original version would balk as the structure of the inputs is not an exact match for the function definition. The second version would return the value 1 since it does not assume that the points have just 2 coordinates – it just uses the first two. It would also be easier to use heads in the second version of the program (`point1_List` and `point2_List`) than in the first - in the original version we can't use the head `Real` on the 4 inputs as it would preclude integer or symbolic coordinates.

Example 5: Counting the real roots of a quadratic

Inputs: a quadratic “quad” and a variable x

Outputs: The number of real roots of the quadratic (which would be 0, 1, or 2).

Discussion: The number of real roots of a quadratic is determined by whether its discriminant (the “ $b^2-4ac$ ” of the quadratic formula) is positive, negative, or 0. The discriminant can be determined using the `Coefficient` command to pick out the values for “a” and “b” (this is why we need the variable x as an input - without it the `Coefficient` command won't work). The constant term “c” can be found by substituting in 0 for the variable x. Because there are 3 cases we will need to use the `Which` command rather than `If` and we can make the entry much easier by using copy-and-paste to copy the messy formula for the discriminant rather than typing it from scratch 3 times.

```
CountQuadRealRoots[ quad_, x_]:= Which[ Coefficient[ quad,x]^2-4 Coefficient[ quad,
x^2]*(quad /. x→0)>0, 2, Coefficient[ quad,x]^2-4 Coefficient[ quad, x^2]*(quad /. x→0)==0,1,
Coefficient[ quad,x]^2-4 Coefficient[ quad, x^2]*(quad /. x→0)<0,0]
```

```
CountQuadRealRoots[quad_, x_] :=
Which[Coefficient[quad, x]^2 - 4 Coefficient[quad, x^2] * (quad /. x → 0) > 0,
2, Coefficient[quad, x]^2 - 4 Coefficient[quad, x^2] * (quad /. x → 0) == 0,
1, Coefficient[quad, x]^2 - 4 Coefficient[quad, x^2] * (quad /. x → 0) < 0, 0]

CountQuadRealRoots[ x^2 - 2 x - 1, x]

2

CountQuadRealRoots[ x^2 + 1, x]

0

CountQuadRealRoots[ x^2 - 6 x + 9, x]

1
```

*counting the real roots of a quadratic*

You probably noticed that `CountQuadRealRoots` is quite messy because we had to compute the discriminant from scratch each time we needed to test with it. In the next section we will learn about “local variables” which make computations like this much easier. In addition there’s a subtle problem with `CountQuadRealRoots` - it may return nonsense or even wrong answers. For example if you use it on  $x^3-x$  you will get 1 as the answer despite the 3 obvious roots. The problem of course is that  $x^3-x$  is not a quadratic - and our program assumes that “quad” is quadratic. But it is not easy to test for “quadraticity” in a one-line program – it could be done with a `Which` command but it would have to be a bit involved. We will work with “input checking” in the next sections.

## Section 4.1 Homework - An Introduction

- 1) Explain the difference between the commands `Print[“The square of 3 is”,3^2, “and the square of (x+1)^2 is”,Expand[(x+1)^2]]` and `Print[“The square of 3 is ”,3^2, “ and the square of (x+1)^2 is ”,Expand[(x+1)^2]]`.
- 2) In Mathematica, look up and explain the difference between `Indeterminate`, `Infinity`, and `-Infinity`.
- 3) Explain in your own words why most programs can be considered mathematical functions.
- 4) What is the purpose of the `Return` command?

Program 1: Write a program `SmallestPrimeDivisor[n]` which returns the smallest prime divisor of the integer `n`.

Program 2: Write a program called `SquareFreeQ[n]` which returns `True` if `n` is not divisible by a squared integer larger than 1 and `False` if it is. (Math hint: An integer is squarefree if in its prime factorization all of the exponents are 1).

Program 3: Write a program `MissingSide[side1, angle, side2]` which finds the length of the missing side of a triangle which has two sides of the given length and the given angle where they join. (Math hint: Law of Cosines)

Program 4: Write a program `PointOfDivision[pt1,pt2,r]` which finds a point `Q` for which is “`r`” of the way from `pt1` to `pt2` along the line between them. (so `t=1/2` is the midpoint, `t=2/3` corresponds to a point  $\frac{2}{3}$  of the way from `pt1` to `pt2`, etc.).

Program 5: Write a program `PolyMult[poly1, poly2, var]` which multiplies the two polynomials in the variable `var` and returns the remainder when the product is divided by `var^2+1`.

Program 6: Write a program `CollinearQ[pt1,pt2,pt3]` which returns `True` if the given points are collinear and `false` otherwise. (Note: If you wish to use the `Slope` program you may do so, but if you do include it in the cell with `CollinearQ` so both may be evaluated at once).

## Section 4.2 - Types of Variables

In the last section all of the programs we wrote were just one line long and no intermediate steps or computations were needed. In real life most programs are fairly complicated and can rarely be done in a single step with any ease. Mathematica has two very closely related structures for linking together several lines of commands into a single program: Block and Module. Both of them use what are called “local variables”, so before we discuss exactly what Block and Module are and the subtle difference between them we need to understand the notion of what it means for a variable to be either “local” or “global”.

In Mathematica we often define numerical constants like  $t=6$  or as expressions like  $\text{mess}=x^3$ . By using the equal sign we are telling Mathematica that  $t$  is identical to 6,  $\text{mess}$  is identical to  $x^3$ , and that these values may be used or interchanged anywhere in Mathematica – if you evaluate  $t^2$  you will get 36 as the answer anywhere in the notebook (or even in another open notebook as long as you don’t quit Mathematica). These types of defined variables (which are the only ones we’ve seen so far) are called global variables since they can be called on anywhere in Mathematica.

A local variable on the other hand is one whose definition does NOT extend throughout your Mathematica session but only a limited portion of it. Usually a local variable is contained or exists within the confines of a single program - and once the program is finished running the variable is undefined, tossed out like scratch paper.

A good analogy for understanding the difference between local and global variables is a geometry homework assignment. Imagine an assignment where you have to figure out several different areas. In the first problem you get  $A=15\pi$ , in the second problem you get  $A=10$ , in the third you get  $A=2\pi$ , and so on. The symbol  $\pi$  is understood to be the same throughout the entire homework as 3.14159... and is therefore a global variable. The variable  $A$  takes on different values in different problems, though. The value for  $A$  in the first problem has absolutely no effect on the value of  $A$  in the second problem – the first value is discarded and you start from scratch in the second problem. This means  $A$  would be a local variable since its value exists only within the confines of a specific problem.

Why do we need local variables? To prevent programs (and main Mathematica calculations) from accidentally overwriting each other’s values. For example suppose you write a program to find the equation of a line. You know you are going to have to calculate the slope of the line and like most people you would call it  $m$ . So you write your program and save it. Someone else uses it, except when they are doing some homework problem they type in  $m=5$  by itself somewhere in the notebook. If all the variables are global ones then you are going to have a problem – either your program will use the incorrect value 5 all the time or it will overwrite the  $m=5$  definition and give the other person a wrong answer in their other calculation. But if in your “find the line” program the variable  $m$  is defined to be a local variable then Mathematica

will really have 2 different “m’s” – the one that the other person typed in that exists normally and a temporary and distinct copy that lives in the confines of the program. Mathematica treats the global  $m=5$  as different from the local  $m$  used in the program so there is no conflict. And when the line program is finished running it throws away the “local value” (whatever it was) and only the global  $m=5$  remains. To avoid conflicts all the “scratch work”/“intermediate quantity” variables that you use in a program should be local variables, not global ones.

Now back to the Block and Module constructs. The format for both is the same: `Block[{local variables}, body]` and `Module[{local variables}, body]`, where the local variables are separated by commas and *body* is a sequence of Mathematica statements, each of which ends in a semi-colon (it’s also best to use the Return key on the main keyboard after a semi-colon to drop down to a new line for typing). The only difference between Block and Module are the starting values for the local variables. In Module no matter what global variables have been defined all local ones (even if they have the same name as a global one) start fresh with no definition – there is no communication between the variables outside the Module and those inside (in fact, Mathematica thinks of the local variables in a Module as having an extra bit of notation at the end –  $m$  really becomes  $m\$1$  in Mathematica’s raw internal code). In Block if a local variable has the same name as an already defined global one (like the 2  $m$ ’s in the analogy) the local variable starts with the same value as the global one. Any changes to that value due to the commands in *body* stay only within the program itself and has no effect on the global variable when the program is over. To see the differences look at the following examples:

```
In[1]:= k = x ^ 3;
        Module[ {x}, x = 4; Print[k + 3]]

3 + x3

In[3]:= Block[ {x}, x = 4; Print[k + 3]]

67

In[4]:= Block[ {x}, x = 8;]
        x

Out[5]= x
```

#### Module vs. Block

Note that the Module line the  $x$  used in the definition of  $k$  is not the same as the  $x$  used inside the module (the “local”  $x$  in Module being completely shielded from the “global”  $x$ ) so the values do not interact at all (in the Module the local variable  $x$  is actually written internally as a more arcane variable like  $x\$9$ ). In the Block command the values for  $x$  used inside and outside are the same so the output for Block line can substitute in the local value  $x=4$  into  $k$ . Even so you’ll

notice that the value 8 inside the last Block command doesn't escape the Block, as you can see in the last command. That's what a "local" variable is all about – not changing values in the main Mathematica notebook.

When should you use Block and when should you use Module? The answer is that if you are careful and always assign starting values at the beginning of the program there is no difference whatsoever - so use what you want. Some people tend to use Block by default because that gives them the option to use some values from outside the program. Others tend to play it safe and use Module to always start with fresh variables and avoid conflicts. In this text we will generally use Block over Module.

Now on to our first program that uses local variables. In the previous section we wrote a program to count the number of real roots for a quadratic. The program was quite messy and would be hard to read to or explain to another person. We can make the program much easier to follow through the use of local variables. Here is a new version of CountQuadRealRoots using Block:

```
CountQuadRealRoots[quad_, x_] :=
  Block[ {a,b,c, disc, roots},
    a=Coefficient[ quad, x^2];
    b=Coefficient[ quad, x];
    c= quad /. x→0;
    disc=b^2-4a*c;
    roots = Which[ disc>0, 2, disc<0, 0, disc==0, 1];
    Return[roots];
  ];
```

Following the logic of the program we start by creating local variables a, b, c, disc, and roots. The first 3 lines of the body define a, b, and c just like they are in the quadratic formula (using the "plug in 0 for x" trick to get the constant term c). We then define the discriminant disc as the " $b^2-4ac$ " term and define roots as the number of real roots as given by whether the discriminant is positive, negative, or 0. This value "roots" is the program's answer so we "return it" by putting it inside a Return command. This step-by-step approach makes it easier to both write the program and follow its internal logic. It also makes it MUCH easier to go back and correct any mistakes. It's best to put a semi-colon after the ] which closes off the Block command so you don't have to see the whole program again.

We can make this program even easier for someone else to follow through the use of "comment lines". Comment lines are little asides you can put anywhere inside Mathematica that it completely ignores. Comments are enclosed in (\*\*) (as in (\* this is a comment line \*)) and show up as gray text. Comments can be used as little notes that explain things in your program. They are not necessary of course, but can make life much easier – particularly if you either share your work or go back to the code to change it weeks or months later (it also makes it much easier



for the person grading your work, hint, hint). You do not need to use semi-colons after a comment because it is ignored by Mathematica. So we could change CountQuadRealRoots as follows:

```
CountQuadRealRoots[quad_, x_] :=
  Block[ {a,b,c, disc, roots},
    (* a is the coefficient of x^2 *)
    a=Coefficient[ quad, x^2];
    (* b is the coefficient of x *)
    b=Coefficient[ quad, x];
    (* c is the constant term *)
    c= quad /. x->0;
    (* disc is the discriminant *)
    disc=b^2-4a*c;
    (* roots counts the number of real roots by looking at the discriminant *)
    roots = Which[ disc>0, 2, disc<0, 0, disc==0, 1];
    Return[roots];
  ];
```

**CountQuadRealRoots [  $x^2 - 2x - 1$ ,  $x$  ]**

2

**CountQuadRealRoots [  $x^2 + 1$ ,  $x$  ]**

0

**CountQuadRealRoots [  $x^2 - 6x + 9$ ,  $x$  ]**

1

*the same examples with the new CountQuadRealRoots*

This version is functionally identical to the previous one - it just is a bit easier to follow. Comments can be used for anything, not just knowing what the variables are – it is common to use them to set off and identify parts of a program ( (\* figuring out the center of the circle \*), for example). For small programs like we will be writing it may seem as if the commenting is as long as the program code itself. But for large and complicated programs commenting can be a huge help towards understanding the flow of a program's logic and where things might be going wrong.

Here are some more examples of programs which use Block (and could use Module):

Example 1: Finding the equation of a line

Inputs: 2 points in the form {a,b}, {c,d}, and variables x and y

Outputs: An equation for the line of the form y== formula or x==number

Discussion: In addition to using local variables we need to check for the case of a vertical line as well as the possibility that the two points are the same. For the main non-vertical line case we will essentially be using the point-slope form for a line based on the point (a,b).

EquationOfLine[{a\_,b\_},{c\_,d\_},x\_,y\_]:=

```
Block[ {m, rhs},
  (* m is the slope *)
  (* rhs is the right-hand side of the equation *)
  (* same point twice case *)
  If[ {a,b}=={c,d}, Print["Not well defined – same point twice"];Return[] ];
  (* vertical line case *)
  If[ a==c, Return[ x==a]];
  (* non-vertical case *)
  m=(d-b)/(c-a);
  rhs=Expand[ m(x-a)+b];
  Return[ y==rhs]
];
```

```
In[9]:= EquationOfLine[ {1, 2}, {4, 7}, x, y]
```

```
Out[9]=  $y = \frac{1}{3} + \frac{5x}{3}$ 
```

```
In[10]:= EquationOfLine[ {3, 7}, {3, 4}, x, y]
```

```
Out[10]=  $x = 3$ 
```

```
In[11]:= EquationOfLine[ {1, 2}, {1, 2}, x, y]
```

```
Not well defined - same point twice
```

*finding the equations of lines in the plane*

There are several features of the code in this program which are worth pointing out. First in the portion of the program that covers the case of the same point twice, the "true" portion of the If statement contains 2 commands – the printed warning and the command to Return nothing. You can chain commands like this inside an If or Which statement by using semi-colons between them. Second, notice that there are several Return commands here, one for each case. When you

use the program once the program executes a Return it gives the answer and stops. That's why there is no If statement for the "non-vertical case"- if you got that far in the program you would know that the line was well-defined and not vertical.

Example 2: The “naïve” derivative (i.e. without using limits)

Inputs: The expression whose derivative is to be taken (mess), the independent variable x.

Outputs: The derivative of “mess” with respect to x.

Discussion: We will use the “naïve” definition of derivative – you form the difference

quotient  $\frac{f(x+h)-f(x)}{h}$ , cancel a common factor of h, and then plug in h=0. Since

we will be introducing the “h” it will need to be a local variable even though it never takes on a value.

NaiveDerivative[mess\_, x\_] :=

Block[ {diffquot, simplified, deriv, h},

(\* diffquot is the difference quotient, simplified is the quotient after canceling, deriv is the derivative \*)

diffquot = ( (mess /. x→x+h)-mess)/h;

simplified=Cancel[FullSimplify[diffquot]];

deriv = simplified /. h→0;

Return[deriv];

];

**NaiveDerivative[ x ^ 6, x]**

6 x<sup>5</sup>

**NaiveDerivative[Sqrt[x], x]**

$\frac{1}{2\sqrt{x}}$

**NaiveDerivative[ (2 x + 1) ^ 5, x]**

2 (5 + 40 x + 120 x<sup>2</sup> + 160 x<sup>3</sup> + 80 x<sup>4</sup>)

**NaiveDerivative[ Sin[x], x]**

Indeterminate

*some basic derivatives*

The combination of Cancel and FullSimplify in that order (i.e. simplify then cancel) is to give Cancel the greatest chance of actually canceling the common factor of h (it will even work on

Sqrt[x]). The derivative of sine failed - in fact the program will fail on most exponential and trigonometric functions as taking those derivatives require special “limits” from calculus (you can’t just cancel the h). In this program just for variety we used one long comment line instead of several small ones.

Example 3: The circle through 3 points

Inputs: 3 points {a,b}, {c,d}, {e,f} and variables x and y.

Output: the standard equation  $(x-h)^2+(y-k)^2=r^2$  of the circle through the points.

Discussion: The first thing we will need to check for is that all the points are different.

We can plug each point into the standard equation, giving us 3 equations using h, k, and r. The Solve command will find the values for h, k, and r, although we’ll have to be careful and check to make sure there actually is a solution.

```
CircleThroughPoints[{a_,b_},{c_,d_},{e_,f_},x_,y_]:=
Block[{h,k,r, standard, eqn1, eqn2, eqn3, sols},
(* h,k, and r are the coordinates of the center and the radius *)
(* standard is the standard equation of a circle *)
(* eqn1, eqn2, eqn3 are the equations you get by plugging in the 3 points into the standard
equation one at a time*)
(* sols is the solution set for the three equations in h, k, and r *)
(* Check for equal points *)
If[{a,b}=={c,d}, Print["Two points are the same."]; Return[] ];
If[{a,b}=={e,f}, Print["Two points are the same."]; Return[] ];
If[{c,d}=={e,f}, Print["Two points are the same."]; Return[] ];
(* set up the equations, one for each point *)
standard = ( (x-h)^2+(y-k)^2==r^2);
eqn1 = standard /. {x->a, y->b};
eqn2 = standard /. {x->c, y->d};
eqn3 = standard /. {x->e, y->f};
(* get solutions for h, k, r*)
sols = Solve[ {eqn1, eqn2, eqn3}, {h,k,r}];
(* Check for no solution *)
If[ Length[sols]==0, Print[ "There is no circle through the points – they must be
collinear"];Return[] ];
(* Real solution with duplicates removed *)
Return[ Union[standard /. sols][[1]] ];
];
```

```
In[21]:= CircleThroughPoints[ {1, 2}, {4, 5}, {6, 8}, x, y]
```

$$\text{Out[21]} = \left( \frac{23}{2} + x \right)^2 + \left( -\frac{35}{2} + y \right)^2 == \frac{793}{2}$$

```
In[22]:= CircleThroughPoints[ {1, 2}, {4, 5}, {1, 2}, x, y]
```

Two points are the same.

```
In[23]:= CircleThroughPoints[ {1, 2}, {4, 5}, {7, 8}, x, y]
```

There is no circle through the points - they must be collinear

*finding the circles through different points*

Although this program is a bit longer the only tricky part of the program is the final Return command. standard /. sols is the standard equation with the values for h, k, and r plugged in. The Union command removes duplicate equations (you get duplicates here because in algebra values like r=-2 are just as good as r=2 even if they don't make sense geometrically) and the [[1]] says take the first (and now only) element of the list of equations.

In the next example we create our first real program that will create graphs. When generating graphs inside of a program you can (and should) assign them local variable names for easy reference. Several graphs generated in this way can be combined with a Show command. The semi-colons we use at the end of each line will prevent the graphs from being shown (this was not true in earlier versions of Mathematica, where you had to use the Plot/ParametricPlot/Plot3D option DisplayFunction→Identity to prevent the graphs from being displayed).

Example 4: Graphing y=f(-x)

Inputs: A function "mess" and a graphing interval/variable in the standard {x,a,b} notation

Output: A graph of y=mess together with the graph of the equation you get by taking y=mess and replacing x with -x. To keep the graphs distinct the second graph should be in red. The graph should be from x=a to x=b.

Discussion: We simply need to create a variable newmess to represent the second function, generate the graphs, and combine them.

```
PlotNegativeX[mess_, {x_, a_, b_}] :=
```

```
Block[{newmess, graph1, graph2},
```

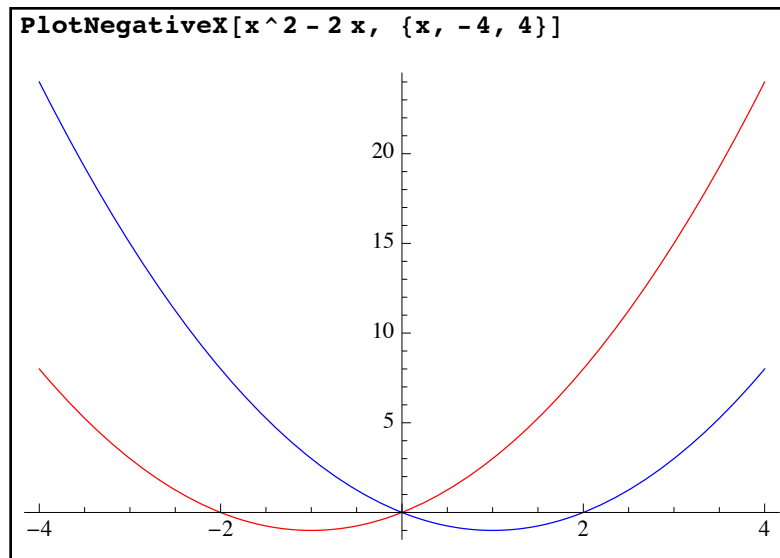
```
(*newmess is mess with x→-x, graph1 is the graph of mess, graph2 is the graph of newmess*)
```

```
newmess = mess /. x → -x;
```

```
graph1 = Plot[mess, {x, a, b}, PlotStyle → Blue];
```

```
graph2 = Plot[newmess, {x, a, b}, PlotStyle → Red];
```

```
Return[Show[graph1, graph2]];
];
```



*the output for PlotNegativeX*

If you try PlotNegativeX for several different functions you can probably see that the graph of  $y=f(-x)$  (in red) is the graph of  $y=f(x)$  flipped around the y-axis.

Example 5: The angle between two lines.

Inputs: Two lines line1 and line2 in a certain order (assumed to be the right-hand sides of the form  $y=mx+b$ ) and a variable x.

Output: The angle from line1 to line2

Discussion: The tangent from a line of slope  $m_1$  to a line of slope  $m_2$  is  $(m_2-m_1)/(1+m_1 m_2)$ . The angle itself is in Quadrant I or II, so a negative tangent will have to be adjusted from Quadrant IV to Quadrant II if an arctangent is used. The slopes can be determined by the Coefficient command and we will need to check for perpendicular lines separately.

```
AngleBetweenLines[ line1_, line2_, x_]:=
```

```
Block[ {m1, m2, tangent},
(* m1 and m2 are slopes of line1 and line2, tangent is the tangent of the angle *)
(* Getting slopes *)
m1=Coefficient[ line1, x];
m2=Coefficient[ line2, x];
(* Perpendicular Case *)
If[ m1*m2==-1, Return[ Pi/2 ] ];
(* Computing tangent *)
tangent = (m2-m1)/(1+m1*m2);
(* Returning angle with quadrant fix *)
```

```
Return[ ArcTan[tangent]+If[ tangent<0, Pi, 0] ];
];
```

$$\begin{aligned} &\mathbf{AngleBetweenLines}[3, x+2, x] \\ &\frac{\pi}{4} \\ &\mathbf{AngleBetweenLines}[x+2, 3, x] \\ &\frac{3\pi}{4} \\ &\mathbf{AngleBetweenLines}[3x+1, 1/3x-2, x] \\ &\pi - \text{ArcTan}\left[\frac{4}{3}\right] \end{aligned}$$

*finding the angle between 2 lines*

Example 6: The multiplication table for U(n)

Inputs: a positive integer n

Output: The “mod n” multiplication table for the set of numbers from 1 to n that are relatively prime to n.

Discussion: The set U(n) is the integers in the range 1 to n that have no common factor with n (which can be determined via Select and either GCD or CoprimeQ). Under multiplication modulo n (which is given by the Mod command) this set is closed and so has a “multiplication table” which can be created via TableForm/TableHeadings.

```
MultiplicationTableforU[n_Integer]:=
Block[ {set, size,multtable,i,j},
(* set will be those integers from 1 to n whose GCD with n is 1 *)
(* size will be how many elements are in set *)
(* multtable will be the raw multiplication table modulo n *)
(* i and j will be the variables used in Table command for multtable*)
(* Check: make sure n is at least 1 *)
If[ n<1, Print[ “The input must be at least 1.”]; Return[ ] ];
(* create the set U(n) and find its size *)
set = Select[ Range[n], GCD[ #1,n]==1 & ];
size = Length[set];
(* create the raw multiplication table *)
multtable=Table[ Mod[ set[[i]] * set[[j]], n ], {i,1,size},{j,1,size}];
(* return a nice table *)
Return[ TableForm[ multtable, TableHeadings-> {set, set} ] ];
```

];

In[39]:= <b>MultiplicationTableforU[15]</b>								
Out[39]/TableForm=								
	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

*the multiplication table for  $U(15)$*

Example 7: Determining the largest prime power which divides a given number.

Input: A non-zero integer  $n$  and a prime  $p$ .

Output: The largest non-negative power of  $p$  which divides  $n$ .

Discussion: In addition to checking the non-zero and prime conditions, we should return

1 immediately if  $p$  does not divide  $n$ . If  $p$  does divide  $n$ , then we can use

FactorInteger to get the prime factorization, Select with MemberQ to pick out the one part that corresponds to the prime, and then form the appropriate power.

```
PrimePowerExactlyDivides[n_Integer,p_Integer]:=
```

```
Block[ {factors,primeandpower},
```

```
(* factors is the prime factorization *)
```

```
(* primeandpower will be the part of the factorization which uses p *)
```

```
If[ n==0, Print["The first argument must be non-zero."];Return[ ];
```

```
If[ !PrimeQ[p], Print[ "The second argument must be prime."]; Return[ ];
```

```
If[!IntegerQ[n/p], Return[1]]; factors=FactorInteger[n];
```

```
(* the [[1]] will make sure remove the extra set of braces from Select *)
```

```
primeandpower=Select[factors, MemberQ[#1,p]&][[1]];

```

```
Return[ p^primeandpower[[2]] ];
```



```

In[99]:= PrimePowerExactlyDivides[ 100, 5]

Out[99]= 25

In[100]:= PrimePowerExactlyDivides[3144, 2]

Out[100]= 8

In[101]:= PrimePowerExactlyDivides[ 0, 5]

The first argument must be non-zero.

```

*finding the highest power of a prime which goes into a number*

Hopefully these examples have helped you understand how the Block/Module command and local variables can help you string individual Mathematica commands into easy-to-follow commented programs.

## Section 4.2 Homework - Types of Variables

- 1) Explain in your own words the difference between a global and local variable. Give an example other than the "area A in a geometry homework" given in the text.
- 2) Explain in your own words the difference between Block and Module.
- 3) What is a comment line, and how do you create one in Mathematica? What are comment lines often used for?
- 4) What happens when a Mathematica program hits a line with If[ something, Return[ answer] ];

Program 1: Write a program FindParabola[ point1, point2, point3, x, y] which finds the equation of the parabola  $y = ax^2 + bx + c$ . Have your program check to make sure that the 3 points all have distinct  $x$ -coordinates. Do not use any of the fitting commands in Mathematica - just use pure algebra.

Program 2: Write a program SphereThrough4Points[pt1,pt2,pt3,pt4,x,y,z] which will find the equation of the unique sphere through the points if they are not coplanar and an error if they are. (Math hint: Use the previous program to help with the warning. Otherwise, the standard equation of a sphere with center (a,b,c) and radius  $r$  is  $(x-a)^2+(y-b)^2+(z-c)^2=r^2$ .)

Program 3: Write a program PlotReplacedX[ mess, {x,a,b},c] which creates a combined graph whose elements are the graph of  $y=\text{mess}$  in black and the the graph of  $y=\text{mess1}$  in red, where mess1 is mess with  $x$  replaced with  $x-c$ . The range of  $x$ -values should be the interval  $[a,b]$ .

Program 4: Write a program PlotAbsolute[ mess,{x,a,b}], as in Program 3 except that mess1 is the absolute value of mess and should be plotted as a dashed red graph.

Program 5: Write a program `InversePlot[ mess, {x,a,b},{y,c,d}]` which plots the graph of  $y=\text{mess}$  in black, the graph of the inverse function for mess in red, and the line  $y=x$  as a dashed line. The graphs should be over the x-interval  $[a,b]$  and y-interval  $[c,d]$ . (Math hint: The inverse is found by setting up the mathematical equation  $y=\text{mess}$  and interchanging  $x$  and  $y$  - the resulting equation is best graphed by `ContourPlot`).

## Section 4.3 - Conditionals and Loops

In many of the programs we have looked at so far we have used various If and Which statements. Both of these are called conditional statements and rely on logic (things being either true or false) in order to determine what action to perform. So far the logical statements we have used have been rudimentary – simple numerical statements like  $x > 0$ . However Mathematica has a large number of logical operators and functions to work with a much broader range of input types. Some of these were discussed in prior sections but are repeated here for completeness together with some additional functions which are commonly used to test if various quantities have a particular form (remember, any built-in function ending in Q evaluates to either True or False):

**&&** : As previously mentioned, && is the "and" operator – so *condition1* && *condition2* is true only when both conditions are true and false otherwise (this can also be done as `And[ condition1, condition2]`, which extends to many variables as well).

**||** : || stands for the logical “or” operator, so *condition1* || *condition2* is true is when at least one of the conditions is true and false if both are false (this can also be done as `Or[ condition1, condition2]`, which also extends to many variables as well).

**Xor**: `Xor[ condition1, condition2 ]` is the exclusive or – it is true when exactly one of the conditions is true and false otherwise (this extends to many variables too although in terms of an odd number being true).

**!** : ! is the negation symbol – it reverses the value of a logical statement (so ! True becomes False).

**!=** : != stands for “not equal to”. Objects that have the same value but represented differently in Mathematica (like  $3/2$  and 1.5) are considered “equal” (so  $1.5 != 3/2$  would be False).

**!=** : != stands for not identical to. The difference between != and == is that == also takes into account the representation of the objects, not just the values. So  $1.5 == 3/2$  is true, even though  $1.5 != 3/2$  is False. == can also be represented as `UnsameQ`.

**MemberQ**[ *list*, *element* ]: The assertion that *element* is in *list*. MemberQ also works on expressions - `MemberQ[  $x^2+2y$ , z ]` will return False as z does not show up in  $x^2+2y$ .

**FreeQ**[ *list*, *element* ]: The assertion that *element* is not in *list*. This is equivalent to !MemberQ. Like the MemberQ command FreeQ works on expressions.

**ListQ**[*object*]: The assertion that *object* is a list.

MatrixQ[*object*]: The assertion that *object* is a matrix (all sublists have the same length).

EvenQ[*number*]: The assertion that *number* is an even integer.

OddQ[ *number*]: The assertion that *number* is an odd integer.

NumericQ[*object*]: The assertion that *object* is a number of any type. There is a command NumberQ which treats symbolic quantities as  $\pi$  as “non-numeric”.

IntegerQ[*object*]: The assertion that *object* is an integer.

PrimeQ[*object*]: The assertion that *object* is a prime number.

Element[ *object*, *domain*]: The assertion that *object* is the set *domain*. Common domains are Complexes, Integers, Primes, Rationals, Reals. Element represents a generalization of Integer and PrimeQ

SquareFreeQ[*object*]: The assertion that *object* is an integer and not divisible by the square of a prime number.

CoprimeQ[ *object1*, *object2*]: The assertion that the the integers *object1* and *object2* are relatively prime (have no common factor other than 1). CoprimeQ[ *object1*, *object2*, *object3*,...] is the assertion that all of the integers are pairwise relatively prime.

PolynomialQ[*object*, *var*]: The assertion that *object* is a polynomial in the single variable *var*. This can be extended to multivariate polynomials by using a list of variables at the end instead of just one.

TautologyQ[ *logicalexpression*, *variablelist*]: TautologyQ[ *logicalexpression*, *variablelist*] is the claim that *logicalexpression* (composed of variables from *variablelist*) is True for all values of the variables.

SatisfiableQ[ *logicalexpression*, *variablelist*]: SatisfiableQ[ *logicalexpression*, *variablelist*] is the assertion that *logicalexpression* (composed of variables from *variablelist*) is True for at least one value of the variables.

With these extra logical functions in place we can write programs which have fairly complex cases in them. For our first example let's return to the program CountQuadRealRoots from earlier in this chapter. In our first attempt at this program we were unable to test whether the input was actually a quadratic but with the new set of logical functions we can improve the function as follows:

```

CountQuadRealRoots[quad_, x_] :=
  Block[ {a,b,c, disc, roots},
    (* a,b,c are the values from the quadratic formula *)
    (* disc is the discriminant of the quadratic *)
    (* roots is the number of real roots, determined by the discriminant *)
    (* check for quadratic input *)
    If[ !( PolynomialQ[quad,x] && Exponent[ quad, x]==2), Print["The input was not a
quadratic in ", x]; Return[ ]; ];
    a=Coefficient[ quad, x^2];
    b=Coefficient[ quad, x];
    c= quad /. x -> 0;
    disc=b^2-4a*c;
    roots = Which[ disc>0, 2, disc<0, 0, disc==0, 1];
    Return[roots];
  ];

```

```

In[5]:= CountQuadRealRoots [ y ^ 2 + 3 y - 1 , y ]

Out[5]= 2

In[6]:= CountQuadRealRoots [ x ^ 3 + 1 , x ]

The input was not a quadratic in x

In[7]:= CountQuadRealRoots [ Sin [x] ^ 2 - Sin [x] + 2 , x ]

The input was not a quadratic in x

```

*our program enhanced with checks on the inputs*

The added If statement has as its test “it is not the case that both quad is a polynomial in x and its degree is 2” – if that is true, then it returns a printed warning and quits.

These extra logical functions can be used in more than just case-checking functions like If and Which – they can also be used in a type of programming construct called a loop. In many applications you need to repeat some step over and over again – either a fixed number of times or until some condition is met (an example from Calculus of this would be taking a sixth derivative, which is a normal derivative applied 6 times). The repeating action is called a loop in programming and Mathematica contains loop commands for both the “fixed number of repetitions” loop type and “until a condition is met” loop type – Do and While.

`Do[ body, {j,start,finish}]`: Repeat *body* (which can be many statements linked by semi-colons) for each value of *j* from *start* to *finish* going up by 1. As in the Table

command you can go up by a different step size by using  $\{j, start, finish, step\}$ .  $j$  is called the iterator or an iterated variable.

While[*condition*, *body*]: Repeat *body* for as *condition* remains True.

Example 1: A slow version of PrimeQ

Input: A natural number  $n$

Output: True if  $n$  is prime, False otherwise

Discussion: We will check to see if  $n$  is prime by seeing if any of the numbers from 2 to the square root of  $n$  are factors of  $n$ . To help with this we will use a local variable “answer” which will start out as True but will switch to False if we find a factor. We also need to worry about the possibility that  $n$  is negative (technically negative numbers can be prime). But a number is prime if and only if its negative is - so we can use absolute values for that case.

```
SlowPrimeQ[n_Integer]:=
  Block[ {answer, k},
    (* answer is the final result, k is the iterator *)
    answer = True;
    (* 1,0,-1 are not prime *)
    If[ n==1 || n==0 || n==-1, Return[False]];
    (* Check for factors from 2 up to Sqrt[n] *)
    Do[ If[ IntegerQ[n/k], answer=False], {k,2,Sqrt[Abs[n]]}];
    Return[answer];
  ];
```

<b>SlowPrimeQ[17]</b>  True  <b>SlowPrimeQ[20]</b>  False
---

*output from SlowPrimeQ*

This is a “slow” program because in many cases it does a lot of unnecessary work. If we ask SlowPrimeQ to check the number 1,000,000,000,000 it will find that it is not prime on the very first step (2 goes into it evenly) but will keep on checking to see if 3 is a factor, 4 is a factor, all the way through a check to see if 1,000,000 is a factor. We will revisit this program as an example of using While later in the section.

The next example is to see if a number is what could be called a “minimal palindrome”. A number is a palindrome in a given base  $b$  if it is the same forwards and backwards in that base

(for example, 121 is a palindrome in base 10). A number  $n$  is a “minimal palindrome” if it is not a palindrome for any bases  $b$  from 2 to  $n-1$  in which it has at least 3 digits. (No matter what  $n$  is, in base  $n-1$  its digits are 11 which is a palindrome). The Mathematica command to create a list of the digits of  $n$  in base  $b$  is `IntegerDigits[n,b]`.

#### Example 2: Checking for a Minimal Palindrome

Input: A natural number  $n$  at least 2.

Output: True or False depending on whether  $n$  is a minimal palindrome.

Discussion: We can use a Do loop to check individual numbers  $b$  from 2 to  $n-1$ . We will need to create a line in the body of the Do loop to reverse the digits of the number  $n$  base  $b$ . We will also use an “answer” variable which starts as True but it changed to False if  $n$  is found to be a palindrome in a base which it is at least 3 digits long.

```
MinimalPalindromeQ[n_Integer]:=
  Block[{digitlist, reversed, answer,b},
    (* answer is the result, b a base/iterator, digitlist the normal list of digits, reversed the
    backwards list of digits, k another iterator *)
    (* Check for valid input *)
    If[n<2, Print["The input must be at least 2. "]; Return[] ];
    answer=True;
    Do[
      digitlist=IntegerDigits[ n,b];
      reversed=Table[ digitlist[[k]], {k, Length[digitlist],1,-1}];
      If[ digitlist==reversed && Length[digitlist]>2, answer=False];
      , {b,2,n-1}];
    Return[answer];
  ];
```

<pre>MinimalPalindromeQ[ 17] False  MinimalPalindromeQ[8128] True</pre>
---

*MinimalPalindromeQ*

The main trick is reversing the numbers in digitlist, which is done by using a Table command which counts down from how long digitlist is and ends at 1 (there is actually a command Reverse which would do this for us but the Table approach is good practice).

While correct this output is not as useful as it could be. It doesn't tell us why a given number is not a minimal palindrome. We can create an “extended” version of the program by just adding a line or two to make things more clear. We could add a second list “palindromes”

to represent the bases/digits in which the number  $n$  is a palindrome. “palindromes” could start out as the empty list {}, and as part of our If command we could add the bases/digits to “palindromes” by using the command AppendTo (AppendTo[ *list*, *element* ] adds *element* to the end of *list*).

Example 3: An “extended” minimal palindrome checker

Input: A natural number  $n$  at least 2.

Output: A list of bases in which  $n$  is a palindrome at least 3 digits long, together with True (if it is a minimal palindrome) or False (if not).

```
ExtendedMinimalPalindrome[n_Integer]:=
  Block[{digitlist, reversed, answer,b,palindromes},
    (* answer is the result, b a base/iterator, digitlist the normal list of digits, reversed the
    backwards list of digits, k another iterator, palindrome the list of bases/digits in which n is
    a palindrome *)
    (* Check for valid input *)
    If[n<2, Print["The input must be at least 2."]; Return[] ];
    answer=True;
    palindromes={};
    Do[
      digitlist=IntegerDigits[ n,b];
      reversed=Table[ digitlist[[k]], {k, Length[digitlist],1,-1}];
      If[ digitlist==reversed && Length[digitlist]>2, answer=False;
        AppendTo[ palindromes, {digitlist,b} ] ];
      , {b,2,n-1}];
    Return[ {palindromes,answer}];
  ];
```

```
ExtendedMinimalPalindrome[170 001]

{{{1, 160, 1}, 340}, {{1, 25, 1}, 400}}, False}

ExtendedMinimalPalindrome[8128]

{{}, True}
```

*more information on palindromes*

Here we can see 170,001 is not a minimal palindrome because it fails the “palindrome test” in the bases 340 and 400.

Do loops all have a fixed number of iterations in them by definition. The While loop structure is more flexible – it repeats as long as some condition is met. Of course if nothing is ever changed in a While loop it would continue forever - so there are some short commands for



changing numerical values in the middle of computations (called increment commands). If  $j$  is the variable you are tracking, we have:

- $++j$ : Add one to the value of  $j$  before using it.
- $j++$ : Add one to the value of  $j$  after using it.
- $--j$ : Subtract 1 from the value of  $j$  before using it.
- $j--$ : subtract 1 from the value of  $j$  after using it.

You can see the difference between  $++j$  and  $j++$  ( $--j$  and  $j--$  work the same way) in the following evaluations:

```
In[24]:= j = 7
          Print[j ++]
          Print[j]

Out[24]=  7

          7

          8

In[27]:= k = 20
          Print[++k]
          Print[k]

Out[27]= 20

          21

          21
```

*++ and --*

With  $j++$  the value is used in Print (so 7 appears twice), and upped to 8 at the end of  $\text{Print}[j++]$ . With  $++k$ , the value is increased from 20 to 21 before being used by Print.

As an example of using a While loop consider the following:

Example 4: Fractional part of a positive number

Input: A positive real number  $x$

Output: The fractional part of  $x$ .

Discussion: To get the fractional part of a positive number you simply have to keep subtracting 1's from it until the result is less than 1. So we will copy the value for  $x$  into a new variable  $y$  and then drop  $y$  by 1's until it is less than 1.

```

MyFractionalPart[x_]:=
  Block[{y},
    (* Number Check *)
    If[ x<0, Print[ x, " is not a positive real number."]; Return[] ];
    y=x;
    While[ y>=1, y--];
    Return[y];
  ];

```

```

In[7]:= MyFractionalPart [32.4]
Out[7]= 0.4

In[8]:= MyFractionalPart [4]
Out[8]= 0

In[9]:= MyFractionalPart [-2]
-2 is not a positive real number.

```

*fractional parts of positive numbers*

As another example of using a While loop we can improve the efficiency of our old “SlowPrimeQ”. The original version of SlowPrimeQ[n] checked to see if every integer from 2 to the square root of n was a factor. However, if we find a single example of a number which is a proper factor we don’t need to check all the rest of the potential factors - one proper factor is enough to prevent it from being prime. We can encode the “check for factors until you don’t need to” idea in a While loop rather than a Do loop.

Example 5: A better version of SlowPrimeQ

Input: A natural number n

Output: True if n is prime and False otherwise

Discussion: We will check factors k from 2 to Sqrt[n], but use a While loop to stop if we find a proper factor. In addition to prevent the loop from having to reevaluate Sqrt[Abs[n]] each time we go through the loop we will give that quantity its own name before we hit the While loop.

```

SlowPrimeQ[n_Integer]:=
  Block[{k, answer},
    (* k is the potential factor being checked, answer is the answer, n1 is the stopping value *)
    (* 0,1,-1 not prime *)
    If[ n== -1 || n==0 || n==1, Return[False] ];
    answer=True;

```

```

n1=N[ Sqrt[Abs[n]] ];
k=2;
While[ k≤n1 && answer, If[ IntegerQ[n/k], answer=False]; k++];
Return[answer];
];

```

To check the efficiency of the new version of SlowPrimeQ against the previous “OldSlowPrimeQ” function, we can use the Timing command. Timing[ *command* ] gives a list of the form { *time taken*, *result* } where the time is given in seconds. Comparing the current version of SlowPrimeQ with the older version on  $n=10,000,000,000,000$  yields

```

Timing[SlowPrimeQ[10 000 000 000 000]]
Timing[OldSlowPrimeQ[10 000 000 000 000]]

{0.000163, False}

{6.90098, False}

```

*fast and not-so-fast prime checking*

In this case the newer version is over 42,000 times faster than the older version. This is an extreme case of course (since  $10,000,000,000,000$  has a factor of 2) but in general the newer version will be no slower and in most cases significantly faster.

For our last example of using a While loop we will create our own polynomial division algorithm. The way polynomial division works is that you divide the high power of the denominator into the high power of the numerator (if possible). You then take that new term, multiply it by the denominator, and subtract. This is repeated until you cannot divide the high powers any further. The sum of the terms from each step is the quotient and the leftover is the remainder.

Example 5: Homemade polynomial division

Inputs: the original polynomial poly1, the polynomial poly2 you are dividing by, and the underlying variable x.

Outputs: A list of the form {quotient, remainder}

```

MyPolyDivision[ poly1_, poly2_, x_]:=
Block[ {q,ratio,newpoly,highterm,newpolyhighterm },
(* q is the variable which will be the quotient *)
(* newpoly is the polynomial after various subtractions *)
(* the highterm variables are the leading terms of the polynomials *)
(* ratio is the ratio of the leading terms *)
(* Check for nonpolynomials and the case where no division is needed *)
If[ !(PolynomialQ[poly1,x] && PolynomialQ[poly2,x]), Print["Nonpolynomial inputs

```

```

    encountered"];Return[] ];
(* Case where no division is needed *)
If[ Exponent[poly2,x]>Exponent[poly1,x], Return[ {0,poly2} ] ];
(* Initializing newpoly, q, and highterm for main loop *)
newpoly=poly1;
q=0;
highterm= x^Exponent[poly2,x]*Coefficient[poly2,x^Exponent[poly2,x]];
(* Main loop *)
While[ Exponent[ poly2,x]<=Exponent[newpoly,x],
    newpolyhighterm=x^Exponent[newpoly,x]*Coefficient[newpoly,
    x^Exponent[newpoly,x] ];
    ratio=Cancel[ newpolyhighterm/highterm];
    q=q+ratio;
    newpoly=Expand[newpoly- ratio*poly2];
];
Return[ {q,newpoly} ]
];

```

```

In[77]:= MyPolyDivision[ x^5, x^2 + x + 1, x]

Out[77]= { 1 - x^2 + x^3, - 1 - x}

In[78]:= MyPolyDivision[ x^4 - 1, x - 1, x]

Out[78]= { 1 + x + x^2 + x^3, 0}

In[79]:= MyPolyDivision[ x, x^3 - 1, x]

Out[79]= { 0, - 1 + x^3}

In[80]:= MyPolyDivision[ Sin[y], y + 1, y]

Nonpolynomial inputs encountered

```

*quotients and remainders via MyPolyDivision*

As a final note about loop constructs, it is worth being aware that there are ways to exit a loop early (i.e. before the final value in a Do loop and when the condition of a While loop is still met). A Return[*item*] command in a loop will exit the *loop* rather than the entire program, with *item* being assigned as the answer to the loop (for example if you evaluate z=Do[ If[ i==4, Return[10]], {i,1,20}], z will be given the value 10 and the loop broken without i ever reaching the values 5 or higher). The command Break[ ] (with no entry in the brackets) will force a loop to exit early without returning any particular value.

Closely related to loop constructs (such as Do and While) are iterated constructs. While the commands Do and While apply a sequence of commands over and over, iterated constructs are more narrowly defined - they apply a single function over and over to the same input. Examples of iterated constructs would be the idea “start with the number 3 and double it 7 times” and “start with the number 3 and keep doubling it while it’s under 1,000,000”. You can do these sorts of things with Do and While loops but there are dedicated commands for it - Nest and NestWhile:

`Nest[function, input, n]`: Apply the *function* to the *input* *n* times, returning the final result.

`NestWhile[function, input, condition]`: Apply the *function* to the *input* repeatedly until the *condition* becomes False, returning the final value.

Both Nest and NestWhile are commonly used with pure functions. We can implement both “start with the number 3 and double it 7 times” and “start with the number 3 and keep doubling it while it’s under 1,000,000” easily using Nest and NestWhile:

```
Nest [ 2 #1 &, 3, 7]
384

NestWhile [ 2 #1 &, 3, #1 < 1 000 000 &]
1 572 864
```

*repeated doublings with Nest and NestWhile*

Note that the final answer is over 1,000,000 as the last doubling takes you from under a million to over it.

If you need to see the intermediate steps in an iterated construct you can get those as well with the related commands NestList and NestListWhile:

`NestList[function, input, n]`: Apply the *function* to the *input* *n* times, returning a list of all the steps (`{a, f[a], f[f[a]], ..., }`)

`NestListWhile[function, input, condition]`: Apply the *function* to the *input* repeatedly until the *condition* becomes False, returning a list of all the steps.

```

NestList[ 2 #1 &, 3, 7]
{3, 6, 12, 24, 48, 96, 192, 384}

NestWhileList[ 2 #1 &, 3, #1 < 1 000 000 &]
{3, 6, 12, 24, 48, 96, 192, 384, 768, 1536, 3072, 6144,
 12 288, 24 576, 49 152, 98 304, 196 608, 393 216, 786 432, 1 572 864}

```

*repeated doublings with the intermediate steps shown*

Note that in the NestWhile and NestWhileList examples above we used #1 in two separate functions. The variables in pure functions are always considered to be local so there is no conflict between using #1 in the “function to be nested” and in the “test to see if you should keep going”.

Example 6: The odd part of an integer

Inputs: A non-zero integer n

Output: The largest odd factor of n.

Discussion: The easiest way to get the largest odd factor of an integer is to divide out all of the factors of 2; we can accomplish this with NestWhile and IntegerQ.

```

OddPart[n_Integer]:=
Block[ {answer},
  (* check for 0 *)
  If[ n==0, Print[ “0 does not have an odd part.”]; Return[] ];
  (* divide by 2 for as long as you can *)
  answer=NestWhile[ #1/2 &, n, IntegerQ[#1/2]& ];
  Return[answer];
];

```

```

OddPart[-342]
-171

OddPart[1024]
1

OddPart[2^1032 × 3]
3

```

*finding the odd parts of integers*

If course we could have done this by manipulating the output from FactorInteger (simply removing any part of the FactorInteger output that starts with a 2 and then converting the result

back into an integer). But this way is actually better (and a little simpler). FactorInteger can take a long time on large numbers (think about how long it can take to factor a 500 digit number into primes) but dividing by 2 is fairly quick even on huge numbers. So using NestWhile to find the odd part can be much faster than using a program based on FactorInteger.

The various forms of the Nest command are also useful when dealing with recursively defined sequences of numbers such as Newton's method from calculus. We will take a look at both of these applications in future sections.

### Section 4.3 Homework - Conditionals and Loops

- 1) Define a short Mathematica command DivisibleBy3Q.
- 2) Define a short Mathematica command RealQ which determines whether a 20 digit approximation of a number is real (Hint: show the imaginary part of the estimate is 0).
- 3) Explain the difference between While and Do.
- 4) Explain the difference between ++j and j++.
- 5) Suppose you had to increment a variable j by 1/2 instead of 1. Can you figure out a way to do this?
- 6) Explain why in the second version of SlowPrimeQ the definition of the variable n1 speeds the program up. Compare versions of this program which use n1 with those that don't.
- 7) Look up the For command, and compare it to While.
- 8) Explain the danger in the innocent looking command `i=1; While[ i>0, i++]`.

Program 1: Create a new version of MyFractionalPart[] which works for any real number input, not just positive numbers.

Program 2: It is well-known that if  $a$  is a positive integer and  $d$  is a rational number, then the average of  $d$  and  $a/d$  is a rational number which is generally closer to the square root of  $a$  than  $d$  is (if  $d$  is fairly close to the square root of  $a$ , the average is usually much closer). Write a program RationalApproximateToRoot[a,d,n], which starts with  $d$  as an approximation and computes  $n$  successive averages (using each average as the new "d" in the next step), returning the last average. Base your program on a Do loop rather than Nest.

Program 3: Repeat program 2 to create a program RationalApproximateToRoot2 but base your program on a Nest command.

Program 4: Write a program RationalApproximateToRoot3[a,d,tolerance] which computes the successive averages described for Program 2 until the difference between them is no bigger in size than tolerance (tolerance should be positive of course). Base your program on a While loop rather than NestWhile.

Program 5: Repeat program 4 to create a program RationalApproximateToRoot4 but base your program on a NestWhile command rather than a While loop.

Program 6: Write a program `PartialCantor[n]` (where  $n$  is a natural number) which creates a list of intervals of the form  $\{ \{x_1, y_1\}, \{x_2, y_2\}, \dots \}$  as follows:

- 1) Start with the interval  $[0, 1]$  in a single element list (as  $\{\{0, 1\}\}$ ).
- 2) Remove the middle third of the interval to get a new list of smaller intervals (so  $\{ \{0, 1/3\}, \{2/3, 1\} \}$ ).
- 3) Go back to step 2, removing the middle third of each piece. Do this so the “middle” thirds” are removed a total of  $n$  times.

The output for `PartialCantor[1]` should be  $\{ \{0, 1/3\}, \{2/3, 1\} \}$ . The output for `PartialCantor[2]` would be  $\{ \{0, 1/9\}, \{2/9, 1/3\}, \{2/3, 7/9\}, \{8/9, 1\} \}$ , and so on. It is probably worth doing a few steps of this process on paper first so you get a good idea of what the mathematical process is before trying to program it. There are many ways to set up `PartialCantor` but one of the easiest involves defining a separate program `RemoveMiddleThird[interval]` and then using it with `Nest`, `Map`, and `Flatten`.

Program 7: Define a program `ChaosIterate[r, y, n]` which returns a list of  $n$  numbers starting with  $y$  and whose next element is the previous element run through the function  $r \cdot x \cdot (1 - x)$ .  $r$  must be between 0 and 4, and  $y$  between 0 and 1.

Program 8: Create a program `ChaosIterate2[r, y, n]` which runs as in program 7, except the list returned is obtained by running  $y$  through the function  $n+50$  times and then dropping the first 50 elements of the list.



## Section 4.4 - Formatting and Presentation

Most of the programs we have done so have been designed to create a single output - a graph, a number, and so on. As your programs get more complex you may want to set things up so that the output are several related things - maybe a graph and some related calculations or two graphs side-by-side. You may also need to control the appearance of portions of your output - instead of Mathematica's standard output  $1 + 3x + x^2$  you may want to see a result in the more traditional form  $x^2 + 3x + 1$  (and maybe you want that in red for some reason). Mathematica has a number of commands designed to give you fine control about how things are grouped as well as how individual elements should appear.

First let's take a look at the commands for grouping several objects together:

**Row[*list*]**: **Row[*list*]** presents the objects of *list* together as a single row. There is no spacing included between the objects (so **Row[ {x,y} ]** appears as *xy* rather than *x y*). You can add spacing by including a blank space at the beginning or end of strings (such as in " the slope is ") or by including extra copies of " " as elements of *list*. **Row[*list*, *object*]** uses *object* as a separator between the parts of the row. Row will accept the options **Frame**→**True** and **Background**→*color* to create a background and frame for your row.

```
Row[ {"The slope of the line is", 4} ]
```

```
The slope of the line is4
```

```
Row[ {"The slope of the line is ", 4} ]
```

```
The slope of the line is 4
```

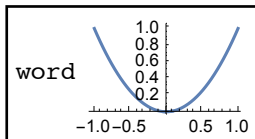
```
Row[ {1, 2, 3, 4, 5}, "X" ]
```

```
1 X 2 X 3 X 4 X 5
```

```
Row[ {"The slope of the line is ", 4}, Frame → True, Background → LightBlue]
```

```
The slope of the line is 4
```

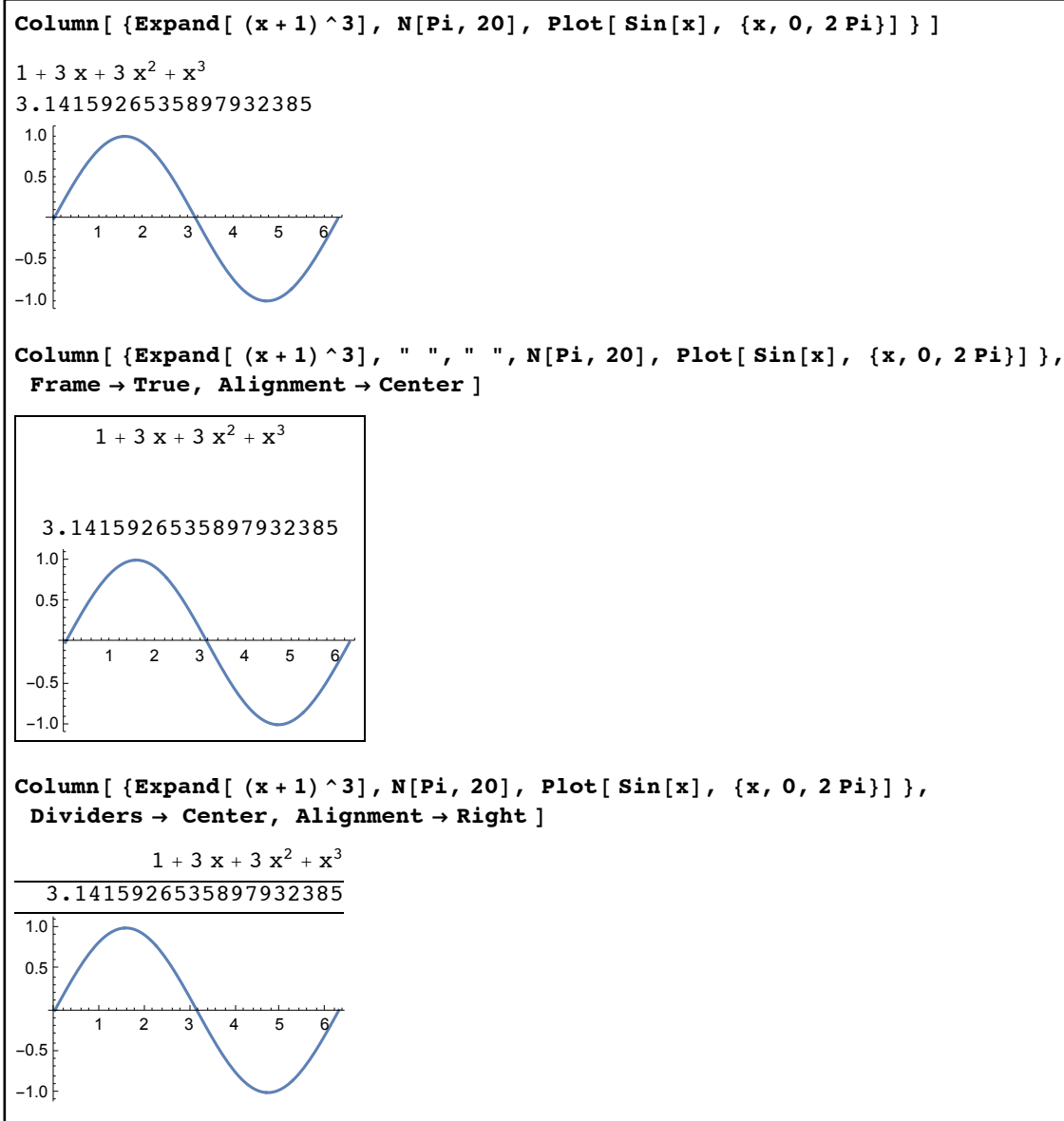
```
Row[ {"word", Plot[ x^2, {x, -1, 1} ]}, Frame → True]
```



*some simple Row commands*

`Column[list]`: `Column[list]` creates a column from the elements of *list*. In addition to the `Frame` and `Background` options there is also an option `Alignment` to control how the parts of the column will be lined up; it can be set to either `Left`, `Center`, or `Right`.

Another common option is `Dividers`, which is used to draw separators. `Dividers→All` will draw both a frame and separators, `Dividers→True` will draw a frame, and `Dividers→Center` will only draw interior dividers (i.e. with no frame).



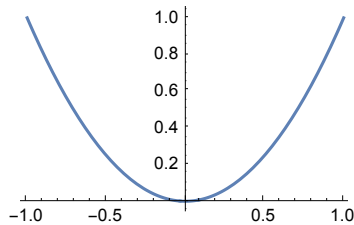
*using column to stack several items*

Note that you can have a column whose elements are given by `Row` as well as a row whose elements are given by `Column`:

```
Column[ {
  Row[ {"the sum of 9 and 5 is ", 9+5}],
  Row[ {"the slope of ", y == 3 x - 1, " is ", Coefficient[ 3 x - 1, x]}],
  Plot[ x^2, {x, -1, 1}]
}]
```

the sum of 9 and 5 is 14

the slope of  $y = -1 + 3x$  is 3



*mixing columns and rows*

`Multicolumn[ list, number ]`: `Multicolumn[ list, number ]` presents the *list* broken up into the appropriate *number* of columns. `Multicolumn[list]` will present the *list* in as “square” a form as possible. `Multicolumn[list, {number, Automatic}]` will present the list using the appropriate *number* or rows, and `Multicolumn[ list, {rownumber, columnnumber}]` will use the appropriate numbers of rows and columns (truncating the list as needed). In all of the versions the elements of *list* will start at the upper left and then proceed down the columns.

```
mylist = Range[30];

Row[ {Multicolumn[mylist], Multicolumn[mylist, 4],
      Multicolumn[ mylist, {7, Automatic}]}, "  "]

      1  9  17 25
      2 10 18 26   1  8  15 22 29
1  6  11 16 21 26   3 11 19 27   2  9  16 23 30
2  7  12 17 22 27   4 12 20 28   3 10 17 24
3  8  13 18 23 28   5 13 21 29   4 11 18 25
4  9  14 19 24 29   6 14 22 30   5 12 19 26
5 10 15 20 25 30   7 15 23       6 13 20 27
      8 16 24       7 14 21 28

Multicolumn[mylist, {3, 5}]

1  4  7 10 13
2  5  8 11 14
3  6  9 12 15
```

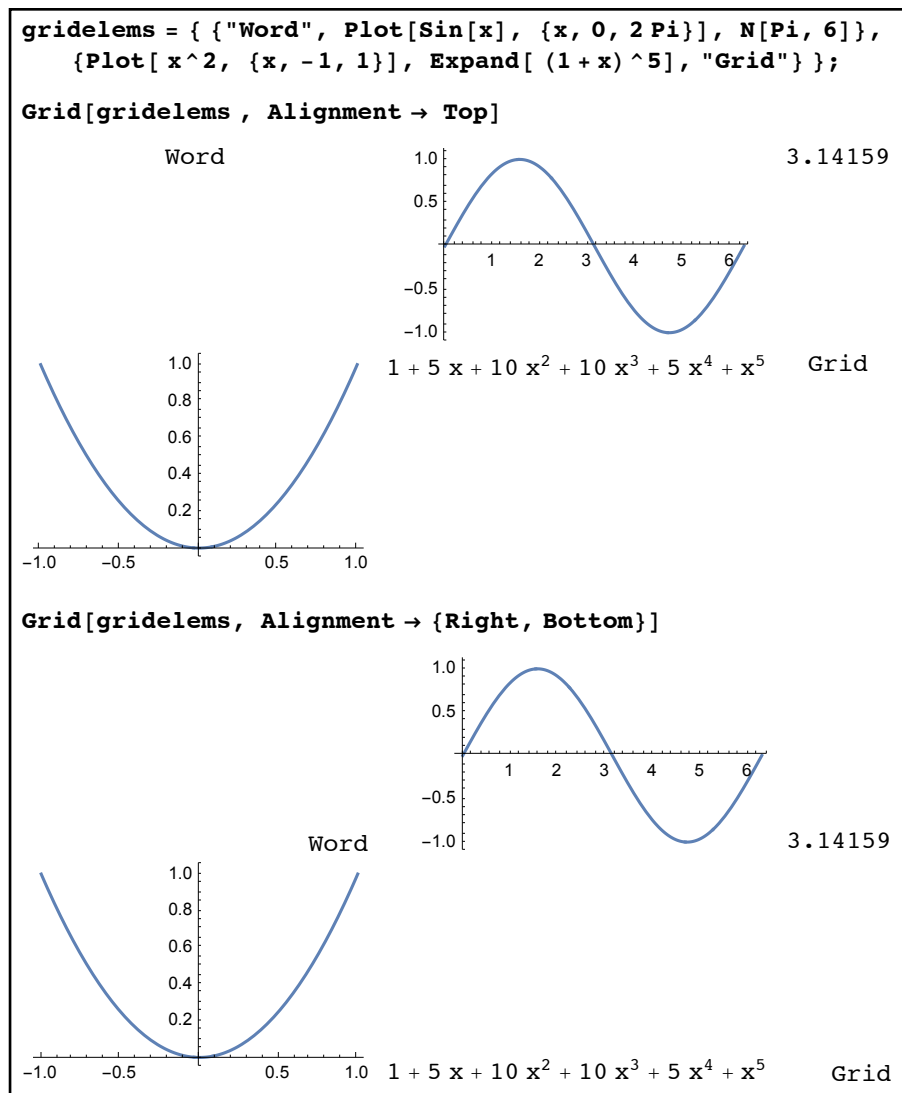
*arranging the numbers from 1-30 using Multicolumn*

`Grid[ list of lists ]`: `Grid` creates an array of the *list of lists*, where the first sublist represents the first row, the second sublist the second row, and so on. This may seem similar to `TableForm` but it is more flexible about the elements, display options, and grouping.

In addition each grid element is allotted the same amount of space (so alignment can be fairly important).

As some of the Grid options have several different forms we discuss them individually:

**Alignment:** Alignment controls how the grid elements are aligned with each other both horizontally (Top, Center, Bottom) and vertically (Left, Center, Right). Alignment can be used either with just a horizontal specification (Alignment→Top), a vertical specification (Alignment→Right) or a list of both (Alignment→{Right, Bottom}).



*alignments in Grid*

**Frame:** The Frame option for Grid controls whether some or all of the grid elements will have a frame around them. Frame→True will draw a single frame around the entire

grid. `Frame→All` will draw a frame around every grid element. `Frame→{All, None}` will draw column dividers and `Frame→{None, All}` will draw row dividers. To frame specific elements use `Frame→{None, None, {list of positions}}`, where elements of *list of positions* have the form `{row, column}→True`.

```
grid2 = Table[ i + j, {i, 0, 6}, {j, 0, 7}];
Row[{Grid[grid2, Frame → True], "    ", Grid[grid2, Frame → All]]]

Grid[grid2, Frame → {All, None}], "    ", Grid[grid2, Frame → {None, All}] ]

Grid[grid2, Frame → {None, None, {{2, 3} → True, {4, 4} → True, {6, 1} → True}}]
```

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13

*various framing options for Grid*

**Background:** Background sets the background color for the grid either whole or in part. `Background→color` sets the background for the entire grid. `Background→{list of colors, None}` sets the column colors individually and `Background→{None, list of colors}` sets each row color individually. In either version using an extra `{}` around the *list of colors* will cause the colors to alternate rather than being set individually.

```

grid2 = Table[ i + j, {i, 0, 6}, {j, 0, 7}];

Row[{Grid[grid2, Background → LightBlue], " ",
      Grid[grid2, Background → {{Red, Yellow, White, LightBlue}, None}]]]

0 1 2 3 4 5 6 7    0 1 2 3 4 5 6 7
1 2 3 4 5 6 7 8    1 2 3 4 5 6 7 8
2 3 4 5 6 7 8 9    2 3 4 5 6 7 8 9
3 4 5 6 7 8 9 10   3 4 5 6 7 8 9 10
4 5 6 7 8 9 10 11  4 5 6 7 8 9 10 11
5 6 7 8 9 10 11 12  5 6 7 8 9 10 11 12
6 7 8 9 10 11 12 13  6 7 8 9 10 11 12 13

Row[{Grid[grid2, Background → {None, {Red, Yellow, White, Orange, LightBlue}}],
      " ", Grid[grid2, Background → {{LightRed, LightBlue}}, None}]]]

0 1 2 3 4 5 6 7    0 1 2 3 4 5 6 7
1 2 3 4 5 6 7 8    1 2 3 4 5 6 7 8
2 3 4 5 6 7 8 9    2 3 4 5 6 7 8 9
3 4 5 6 7 8 9 10   3 4 5 6 7 8 9 10
4 5 6 7 8 9 10 11  4 5 6 7 8 9 10 11
5 6 7 8 9 10 11 12  5 6 7 8 9 10 11 12
6 7 8 9 10 11 12 13  6 7 8 9 10 11 12 13

```

*background options for Grid*

You can also set the background for individual items in the grid. While you can do this in a similar manner to setting frames there is a much easier way through the `Item` command. `Item[object, property]` assigns the *property* to the given item. So by using `Item[4, Background→Orange]` instead of 4 as an element in a `Grid` you would see the 4 with an orange background. This is a handy trick for highlighting grid elements that have a certain property. For example we can use this to easily create a table of the numbers 1-200 and highlight the primes:

- 1) First create the numbers 1-200 via `Range`.
- 2) Use `Map` and a pure function to replace the primes in the list with the form `Item[ number, Background→Yellow]` (and leave nonprimes alone).
- 3) Use `Partition` to carve the 200 element list into an array (say where each row has 15 items).
- 4) Use `Grid` with `Frame→All` to display the array.

```
list1 = Range[200];
list2 = Map[If[PrimeQ[#1], Item[#1, Background → Yellow], #1] &, list1];
list3 = Partition[list2, 15];
Grid[list3, Frame → All]
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100	101	102	103	104	105
106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
121	122	123	124	125	126	127	128	129	130	131	132	133	134	135
136	137	138	139	140	141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160	161	162	163	164	165
166	167	168	169	170	171	172	173	174	175	176	177	178	179	180
181	182	183	184	185	186	187	188	189	190	191	192	193	194	195

*a quick representation of the primes from 1-195 using Item to set individual backgrounds*

Closely related to Row, Column, and Grid are the commands GraphicsRow, GraphicsColumn, and GraphicsGrid. These functions are meant to link graphics (like those created by Plot and similar commands) into rows, grids, and arrays. The main advantage of the “Graphics” versions is that they allow the row/column/grid to be resized as a whole rather than element by element.

In addition to the “grouping” commands like Row and Grid there are also commands that control the appearance of individual expressions, whether they are intended to be part of a row, column, or grid or just standalone expressions:

**TraditionalForm:** TraditionalForm[*expression*] writes the *expression* in standard mathematical (rather than Mathematica) form. So polynomials will be written from high power to low, square brackets will be replaced with parentheses, equations will use = rather than ==, and so on.

```

In[172]:= TraditionalForm[ Expand[ (2 x + 1) ^ 5 ] ]
Out[172]/TraditionalForm=

$$32 x^5 + 80 x^4 + 80 x^3 + 40 x^2 + 10 x + 1$$


In[173]:= TraditionalForm[ Surd[ Sin[x] ^ 5 + Sqrt[x], 3 ] ]
Out[173]/TraditionalForm=

$$\sqrt[3]{\sqrt{x} + \sin^5(x)}$$


In[176]:= TraditionalForm[ x ^ 2 + y ^ 2 == x y + Log[ x / y ] ]
Out[176]/TraditionalForm=

$$x^2 + y^2 = x y + \log\left(\frac{x}{y}\right)$$


```

*using TraditionalForm to write things in “normal” notation*

**NumberForm:** NumberForm controls the expression of a number. The most common way to use NumberForm is on an integer when you want to use commas in the usual way; use NumberForm[ *number*, DigitBlock→3].

```

In[181]:= NumberForm[ 2 ^ 135, DigitBlock → 3 ]
Out[181]/NumberForm=
43,556,142,965,880,123,323,311,949,751,266,331,066,368

```

**HoldForm:** HoldForm[*expression*] returns the *expression* without doing any of the computations implicit in *expression*. This is useful for showing individual parts of a computation (often embedded in a Row).

```

Row[ {"The slope from (1,3) to (5,7) is m=",
      HoldForm[ (7 - 3) / (5 - 1) ], "=", (7 - 3) / (5 - 1) } ]

```

$$\text{The slope from (1,3) to (5,7) is } m = \frac{7-3}{5-1} = 1$$

*suppressing the arithmetic in a slope computation with HoldForm*

**Text:** Text[ *expression* ] converts any strings (the text between “ ”) into actual text. The main difference for our purposes is the font used. The *expression* can be a Row, Column, or Grid and the Text will work on all of the appropriate parts.

```

Row[ {"The slope from (1,3) to (5,7) is m=",
      HoldForm[ (7 - 3) / (5 - 1) ], "=", (7 - 3) / (5 - 1) } ]

Text[%]

```

$$\text{The slope from (1,3) to (5,7) is } m = \frac{7-3}{5-1} = 1$$

*using text to convert strings to true text*



Style: `Style[expression, options]` applies various styling options to the expression. The most common options are colors (as in `Style[x^3, Red]`), “FontFamily” (as in `Style[“word”, “FontFamily” → “Arial”]`) and “FontSize” (as in `“FontSize” → 48`).

```
In[190]:= Style[ 2^10, Blue]
Out[190]= 1024

In[193]:= TraditionalForm[Style[ Expand[(x + 1)^3], Red, "FontSize" → 34]]
Out[193]/TraditionalForm=
```

$$x^3 + 3x^2 + 3x + 1$$

```
In[196]:= Style[ "A large word", "FontFamily" → "Times", "FontSize" → 48]
Out[196]=
```

A large word

*custom styles with the Style command*

Inactive: `Inactive[function]` creates an “inert” form of *function* which cannot process any values. In some ways this is similar to `HoldForm` except inactive functions can be used with `Map` and other Mathematica commands. Inactive functions are grayed out in regular Mathematica expressions but not when `TraditionalForm` is wrapped around them. Wrapping the command `Activate` around an expression will make inactive functions come to life.

```
Inactive[Cos][0]
Cos[0]

TraditionalForm[%]
TraditionalForm=
cos(0)

Map[ Inactive[ Sin], {0, Pi / 6, Pi / 4, Pi / 3}]
{Sin[0], Sin[Pi/6], Sin[Pi/4], Sin[Pi/3]}

TraditionalForm[%]
TraditionalForm=
{sin(0), sin(Pi/6), sin(Pi/4), sin(Pi/3)}

Activate[%]
{0, 1/2, 1/sqrt(2), sqrt(3)/2}
```

*using Inactive to create “unevaluated” expressions*

Now let's take a look at applying these ideas to some programs:

Example 1: Finding the slope between two points

Inputs: 2 points {a,b} and {c,d}

Outputs: A step-by-step computation of the “delta x” and “delta y” needed for the slope as well as the slope itself.

Discussion: We can use Column to stack the 3 individual computations, with each one contained in a row. HoldForm will let us display the computation's parts and Text wrappers will make the output look nice.

```
MySlope[ {a_,b_},{c_,d_}]:=
Block[ {deltax, deltay, slope,output},
(* deltax, deltay, and slope refer to the mathematical quantities *)
(* output will be the formatted answer *)
deltax=c-a;
deltay=d-b;
slope=deltay/deltax;
output=Column[ {
  Row[ {"The change in x is given by  $\Delta x =$ ", HoldForm[c-a], " $=$ ",deltax}],
  Row[ {"The change in y is given by  $\Delta y =$ ", HoldForm[d-b], " $=$ ",deltay}],
  Row[ {"The slope is given by  $m = \frac{\Delta y}{\Delta x} =$ ",HoldForm[(d-b)/(c-a)], " $=$ ",slope}]
}];
Return[Text[output] ];
];
```

**MySlope[{1, 3}, {5, -11}]**

The change in x is given by  $\Delta x = 5 - 1 = 4$

The change in y is given by  $\Delta y = -11 - 3 = -14$

The slope is given by  $m = \frac{\Delta y}{\Delta x} = \frac{-11-3}{5-1} = -\frac{7}{2}$

*a program which shows the parts of a slope computation*

One nice thing about having a program like this is you can wrap a Manipulate command around it to make it interactive:

```
Manipulate[ MySlope[ point1, point2],
  {{point1, {1, 3}, "Point 1 (in the form {x1,y1):"}},
  {{point2, {5, 11}, "Point 2 (in the form {x2,y2):"}},
  ]
```

Point 1 (in the form {x<sub>1</sub>,y<sub>1</sub>): {1, 3}

Point 2 (in the form {x<sub>2</sub>,y<sub>2</sub>): {5, 11}

The change in x is given by  $\Delta x = 5 - 1 = 4$   
 The change in y is given by  $\Delta y = 11 - 3 = 8$   
 The slope is given by  $m = \frac{\Delta y}{\Delta x} = \frac{11-3}{5-1} = 2$

*the slope calculation made interactive*

This would be a great thing to have in a basic algebra course - you could have the students choose their own points (as many times as they want) and see the slope calculation done step-by-step. One drawback to having the freely-chosen input is that you will get an error if the first and second point have the same x-coordinate. You will fix this issue (for both this example and the next) in the homework.

Example 2: Finding the equation of a line and graphing it.

Inputs: 2 points (assumed to have different x-coordinates)

Output: The slope computation for the points, the equation of the line (in point-slope and slope-intercept forms), and the graph of the line with the points shown.

Discussion: We will need to find the slope, the 2 forms for the line, and the graph.

HoldForm and TraditionalForm can be used to store the computations and display them nicely. We can use Min and Max to find the range on which we should graph (say plus 2 units on either side).

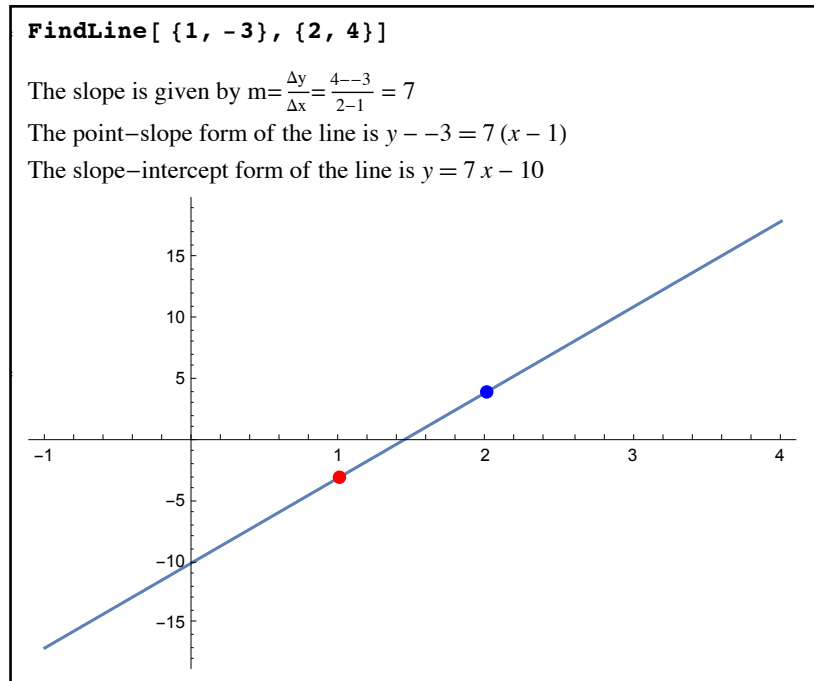
FindLine[ {a\_,b\_},{c\_,d\_}]:=

```
Block[ {m, pointslope, slopeintercept, graph, minx,maxx},
  (* slope is the slope of the line *)
  (* pointslope and slopeintercept are the forms for the line *)
  (* minx and maxx are the limits for the graph *)
  (* graph is the graph of the line *)
  m=(d-b)/(c-a);
  pointslope=HoldForm[ y-b]==m HoldForm[(x-a)];
  slopeintercept= y== Expand[ m(x-a)+b];
  minx=Min[{a,c}]-2;
  maxx=Max[{a,c}]+2;
```

```

graph=Plot[ Expand[ m(x-a)+b], {x,minx,maxx}, Epilog→{PointSize[Large], Red,
Point[ {a,b}], Blue, Point[ {c,d}], ImageSize→400}];
Return[ Text[
  Column[ {
    Row[{"The slope is given by  $m = \frac{\Delta y}{\Delta x} =$ ", HoldForm[(d-b)/(c-a)], " = ", m}],
    Row[{"The point-slope form of the line is ", TraditionalForm[pointslope]}],
    Row[{"The slope-intercept form of the line is ",
      TraditionalForm[slopeintercept]}],
    graph
  }]
]
];

```



One curious thing you may have noticed is that in defining the slope-intercept form of the line rather than one large HoldForm we used two, with the slope  $m$  being outside the second HoldForm. The reason for this is essentially that  $m$  is a computed value and HoldForm stops all computation (if you leave the  $m$  inside the HoldForm it will be printed as just “ $m$ ” in the output). There are other ways around this but they are fairly technical (one way involves putting the Block inside another command called With, with the slope computed inside the With but before the Block).

Example 3: The Sieve of Eratosthenes

Input: An integer  $n$  (at least 2) and a list of non-negative integers

Output: A (roughly) square array of the integers from 2 to  $n$ , with proper multiples of the integers from the list marked with a light red background.

Discussion: As we are looking for a roughly square array we should Multicolumn for the final formatting. The background shading can be done using Table to create a list of Item commands, where the Background is set according to either White or LightRed based on divisibility. It is good programming technique to make sure that the list only contains natural numbers.

```
SieveOfE[ n_Integer, div_List]:=
Block[ {k,shadednumbers},
(* k is an iterator *)
(* shadednumbers is the list of Items that will go into Multicolumn *)
(* error check for n *)
If[ n<2, Print[ "The given integer must be 2 or greater."]; Return[] ];
(* error check for the list div *)
If[ MemberQ[ Map[ (IntegerQ[#1]&& #1>0)&, div ], False ], Print["The second
input must be a list of natural numbers"]; Return[] ];
(* start the program *)
shadednumbers=Table[
Item[k, Background→If[ MemberQ[ Map[ IntegerQ[k/#1] && k>#1 &, div],
True ], LightRed, White] ],
{k,2,n} ];
Return[ Multicolumn[ shadednumbers, Frame→All ] ];
];
```

**SieveOfE[ 100, {2, 3, 4.5}]**

The second input must be a list of natural numbers

**SieveOfE[100, {2, 3}]**

2	12	22	32	42	52	62	72	82	92
3	13	23	33	43	53	63	73	83	93
4	14	24	34	44	54	64	74	84	94
5	15	25	35	45	55	65	75	85	95
6	16	26	36	46	56	66	76	86	96
7	17	27	37	47	57	67	77	87	97
8	18	28	38	48	58	68	78	88	98
9	19	29	39	49	59	69	79	89	99
10	20	30	40	50	60	70	80	90	100
11	21	31	41	51	61	71	81	91	

*the Sieve of Eratosthenes*

There are two parts of the SieveOfE program that are a bit clumsy - the error check for the list div and the criterion “are any of the numbers from div proper factors of k” that is used in the If statement. Both are based on the same idea - Map a True/False function across a list, and then use MemberQ to see if the resulting list contains a False or True we are looking for. A nicer (and more useful) way to get the same result is to use the command Apply. Apply[*function*, *list*] takes *list*, remove the outermost braces, and use the resulting sequence as the inputs for *function*. This can be very useful as most functions (like GCD, And, and Or) don’t work on lists directly (or if they do they essentially “Map” to the list rather than taking all of the list elements as input for a single application). So GCD[ {2,6,8} ] would return the list {2,6,8} as the GCD is mapped to each of the three elements (and the GCD of a single number is itself). But Apply[GCD, {2,6,8}] is effectively the same as GCD[2,6,8] (which is 2). Here are several examples of Mathematica commands that work best with lists using Apply:

```
Taking GCD's:
Apply[ GCD, {5, 6, 7}]
1

Is at least one element of a list true?
Apply[ Or, {True, False, False, False, True} ]
True

Are all of the elements of a list True?
Apply[ And, {True, True, False, True, True}]
False

Make a list of lists into one large list:
Apply[ Join, { {1, 2}, {3, 5, 6}, {6, 7, 8} } ]
{1, 2, 3, 5, 6, 6, 7, 8}
```

*different applications of Apply*

In our SieveOfE program we could have replaced MemberQ[ Map[ (IntegerQ[#1]&& #1>0)&, div ], False ] with Apply[ And, Map[ (IntegerQ[#1]&& #1>0)&, div ] and replaced MemberQ[ Map[ IntegerQ[k/#1] && k>#1 &, div], True ] with Apply[ Or, Map[ IntegerQ[k/#1] && k>#1 &, div] ].

## Section 4.4 Homework - Formatting and Presentation

- 1) If problem1 is defined to be {Plot[ Sin[x], {x,0,2Pi}], Plot[ x^2, {x,-1,1}], ContourPlot[ x+ x y+1==3, {x,-10,10},{y,-10,10}] }, what is the difference between Row[problem1] and GraphicsRow[problem1]
- 2) What does the Appearance option control in Multicolumn?

- 3) What is the difference between `Plot[ x^2, {x,-2,2}, PlotLabel→ y==x^2]` and `Plot[ x^2, {x,-2,2}, PlotLabel→ Text[y==x^2] ]`?
- 4) If `f[x_, y_] := x+y^2`, how would you use `Map` and `Apply` to create the list of values for `f` for the list `{{1,3},{2,-6},{3,5}, {4,8}, {5,17}}`?

Program 1: Write a program `TrigValues` that takes a number *angle* and creates a 3-by-2 grid of the value of the six trigonometric functions at *angle*, where each value is presented along the lines of  $\sin(\frac{\pi}{6}) = \frac{1}{2}$  (i.e. the function and angle are presented on the left side of the = and the value is on the right) and the grid uses alternating colors as the background for each row.

Program 2: Write a program `MultiplyOut` that takes a list of ordered pairs `{ {a,b}, {c,d}, ... }` and forms the number  $a^b c^d \dots$  (essentially reversing the output form we see in `FactorInteger`). (Programming hint: Look up the command `Times` and use it with `Map` and `Apply`).

Program 3: Write a program `Show derivatives` whose input form `ShowDerivatives[function, {variable, start, finish}]` and whose output is a 2-by-2 array of the graphs *function*, `D[function, variable]`, `D[function, {variable,2}]`, and `D[function, {variable,3}]`, all of which are labeled (we will discuss the `D` command in the next chapter).

Program 4: Write a program `CompleteTheSquare` which takes a quadratic in a variable *x* and shows the step-by-step process of completing the square, with each step separated from the next by a dividing line. (Mathematical hint: there are two cases, a simpler one where the leading coefficient is 1 and a longer case where it isn't).

## Section 4.5 - Random Generators

There are many times in Mathematica when you want to randomly generate various objects - points on a curve, a random formula for a function, simulations of an experiment, and so on. The majority of the time you will probably want to select a random number or object “uniformly” - in other words, with the same likelihood of each allowed object being chosen. This is the simplest type of randomness to create in Mathematica and are created by a group of commands that all start with “Random” (Random itself is a legacy function that was superseded with the following commands in version 6 of Mathematica):

`RandomReal[ {minimum, maximum} ]`: `RandomReal` generates a random real number in the given range. `RandomReal[ {minimum, maximum}, n]` generates a list of *n* random real numbers in the given range.

`RandomInteger[ {minimum, maximum} ]`: `RandomInteger` generates a random integer in the given range. `RandomInteger[ {minimum, maximum}, n]` generates a list of *n* random integers in the given range.

`RandomComplex[ {zmin, zmax} ]`: `RandomComplex` a random complex number from the rectangle whose opposite corners are *zmin* and *zmax*. `RandomComplex[ {zmin, zmax}, n]` creates a list of *n* such numbers.

`RandomChoice[ list ]`: A random choice from *list*. `RandomChoice[ list, n]` makes a list of *n* random choices from *list* (repetitions allowed). `RandomChoice` was mentioned in Section 3.2 and also allows you to weight the elements of the list so they are not equally likely.

`RandomSample[ list, n]`: A list of *n* random choices from *list* with no repetitions allowed. Like `RandomChoice` `RandomSample` allows for a weighting of the elements using the same basic form `RandomSample[ weightlist→list, n]`. `RandomSample[ list ]` gives a random permutation (i.e. rearrangement) of all the elements in *list*.

Here are some examples that illustrate the use of the “random” functions in use:



```

RandomReal[{-5, 6}]
-4.61173

RandomReal[{-5, 6}, 8]
{-1.13968, -1.69343, 3.18418, 5.17749, -1.12004, 0.60983, -0.572392, -1.3045}

RandomInteger[{200, 300}]
200

RandomInteger[{1, 10 000 000}, 5]
{2 948 993, 825 993, 3 412 129, 871 040, 6 556 697}

RandomComplex[{1 + I, 3 + 2 I}]
1.44054 + 1.14573 i

RandomComplex[{1 + I, 3 + 2 I}, 6]
{1.46631 + 1.11804 i, 1.02485 + 1.32445 i, 2.3222 + 1.67749 i,
 2.33573 + 1.763 i, 1.88482 + 1.96142 i, 2.01758 + 1.72343 i}

RandomChoice[{1, 2, 3, 4, 5}]
4

RandomChoice[{1, 2, 3, 4, 5}, 4]
{5, 1, 4, 5}

RandomSample[{1, 2, 3, 4, 5}, 4]
{4, 1, 2, 3}

```

*various randomly generated objects*

The key distinction between these is between RandomChoice and RandomSample - in RandomChoice the selection from the list is “with repetition”, and in RandomSample it is “without repetition”. You can force RandomSample to allow a kind of repetition by repeating elements of the list - in RandomSample[ {1,1,1,2}, 3] you can get multiple 1’s as RandomSample treats the 3 1’s as distinct possible choices (sort of “1 #1”, “1 #2”, and “1 #3”).

Before we look at several examples of using the Random commands in programs there is one other important command that involves Mathematica’s random number generators – SeedRandom. When working with computers there is no such thing as a truly random number. The random number generator basically has a hidden internal number which it then uses to generate the random number you ask for. It then uses the next hidden internal number in its built-in pattern to generate the next random number, and so on. If the generator sees the same internal

number twice it will actually give the same “random number” output, so the numbers you generate are not truly random (technically they’re called “pseudorandom”). This can be a problem if two computers start with the same internal number – they will generate identical “random” output. To combat this there is a command called SeedRandom. When used in the form SeedRandom[ ] (no input) the internal number is reset to the time of day. SeedRandom[n] (where n is an integer) resets the internal number to the integer n. Here is an example using SeedRandom in the worst possible way – to reproduce supposedly “random” data:

```

RandomReal[ {1, 10}, 5]

{2.55598, 6.29437, 5.51704, 2.40815, 4.27487}

RandomReal[ {1, 10}, 5]

{9.5806, 7.70309, 4.07019, 5.83757, 7.1264}

SeedRandom[21]

RandomReal[ {1, 10}, 5]

{3.39091, 5.466, 5.90214, 4.54363, 3.40572}

SeedRandom[21]

RandomReal[ {1, 10}, 5]

{3.39091, 5.466, 5.90214, 4.54363, 3.40572}

```

*RandomReal looks random, but isn't always*

Using SeedRandom forces the “random” numbers generated by RandomReal to be the same in different runs. SeedRandom is never used this way in practice - if for some reason 2 computers are “stuck” generating identical sequences of random numbers you can reset them by using SeedRandom[ ]. When the Mathematica kernel is first evaluated it uses SeedRandom with the exact time of day so it’s very unlikely to different computers to generate the same string of numbers (but not impossible).

Now that we have the random generator commands down let’s take a look at some examples of programs using random numbers:

Example 1: Counting heads in a run of coin flips

Input: A natural number n.

Output: The number of times a “head” occurs in a run of n flips of a fair coin

Discussion: Other than checking the data type on n, we can do this by letting 0 represent a head and 1 a tail, generate a random sequence, and use Count.

```
CountHeadsInRun[n_Integer]:=
  Block[ {run, count},
    (* k is an iterator, run is the list of flips, 0=head 1=tail *)
    If[n<1, Print["The input must be a natural number."]; Return[]];
    run=RandomInteger[{0,1},n];
    Return[Count[run,0]];
  ];
```

**Table[CountHeadsInRun[100], {i, 1, 10}]**

{59, 53, 50, 48, 42, 47, 45, 58, 49, 44}

*a lot faster than flipping the coin yourself*

Example 2: A quadratic with distinct random integer roots

Input: Two real numbers a and b at least 2 units apart and a variable x.

Output: An expanded quadratic with distinct integer roots between a and b.

Discussion: The main trick is to make sure that the roots "m" and "n" are distinct.

```
RandomRootParabola[ {a_,b_}, x_]:=
  Block[ {m,n,poly},
    (* m and n are random roots *)
    (* poly is the returned polynomial *)
    If[ Abs[a-b]<2, Print[ "The selected range for the roots is not wide enough"];
      Return[] ];
    m=RandomInteger[{a,b}];
    n=RandomInteger[{a,b}];
    While[ m==n, n=RandomInteger[{a,b}]];
    poly=Expand[ (x-m)(x-n)];
    Return[poly];
  ];
```

**Table[ RandomRootParabola[ {-4, 7}, x], {k, 1, 10}]**

{ 5 - 6 x + x<sup>2</sup>, - 2 - x + x<sup>2</sup>, 3 - 4 x + x<sup>2</sup>, - 2 - x + x<sup>2</sup>,  
4 + 5 x + x<sup>2</sup>, 24 - 10 x + x<sup>2</sup>, 3 + 4 x + x<sup>2</sup>, - 5 x + x<sup>2</sup>, 5 - 6 x + x<sup>2</sup>, - 5 - 4 x + x<sup>2</sup>}

*randomly generated quadratics with distinct integer roots*

In this example it would have been possible to use RandomSample to generate a set of distinct roots. We could have defined a variable "rootlist", set it equal to RandomSample[ Table[ k, {k,Ceiling[a], Floor[b]}],2], and then used a line like {m,n}=rootlist. This approach works in theory (and probably in most practice), but would be very stressful for the computer if the limits

a and b were large. In this RandomSample approach the computer has to generate the list of all integers from a to b and then select from it; if a were -1,000,000 and b=1,000,000 we'd be storing a lot of unnecessary information in creating the list. The use of the While loop avoids creating or storing the large list of possibilities - which is definitely more efficient.

Example 3a: Creating a poker hand (version 1)

Inputs: An integer n in the range 1-52

Output: An hand with n cards with no repetition.

Discussion: We can represent the suits and face cards by strings of text in quotes "Ace", "Hearts", and so on. Using one list to represent 2-Ace and another for the 4 suits we can then create a card by generating two random integers from 1-13 and 1-4 and taking those positions in the lists. If the randomly generated card is one we haven't picked before (in the form {rank, suit}, checked by MemberQ) we can use AppendTo to add it to the hand.

```
CreateHand[n_Integer]:=
  Block[{hand, suits, ranks, j,k, card},
    (* j and k are used in generating a card, other variables as named *)
    If[ n<1 || n>52, Print["Invalid number of cards"];Return[]];
    ranks={2,3,4,5,6,7,8,9,10,"Jack", "Queen", "King", "Ace"};
    suits={"Hearts", "Diamonds", "Spades", "Clubs"};
    hand={};
    While[ Length[hand]<n,
      j=Random[Integer,{1,13}];
      k=Random[Integer,{1,4}];
      card={ranks[[j]], suits[[k]]};
      If[ !MemberQ[hand, card], AppendTo[hand, card]];
    ];
    Return[hand];
  ];
```

```
In[11]:= CreateHand[5]
```

```
Out[11]= {{Ace, Diamonds}, {2, Clubs}, {7, Spades}, {8, Clubs}, {Ace, Spades}}
```

```
In[12]:= CreateHand[7]
```

```
Out[12]= {{10, Hearts}, {3, Clubs}, {King, Clubs},
           {Ace, Clubs}, {7, Spades}, {King, Diamonds}, {2, Diamonds}}
```

```
In[13]:= CreateHand[53]
```

```
Invalid number of cards
```

*creating hands drawn from a 52 card deck*

### Example 3b: Creating a poker hand (version 2)

Inputs: An integer n in the range 1-52

Output: An hand with n cards with no repetition.

Discussion: This time we will use a simpler method to create the hand based on RandomSample. This will require us to build and store the entire deck ahead of time (a 52 card list instead of smaller lists ranks and suits). We can build the deck using a combination of Table and Flatten (the latter used to drop the result of Table from a 2D matrix to a 1D list).

```
CreateHand[n_Integer]:=
  Block[{suits, ranks, j,k, card,fulldeck},
    (* j and k are used in generating a card, other variables as named *)
    If[ n<1 || n>52, Print["Invalid number of cards"];Return[]];
    ranks={2,3,4,5,6,7,8,9,10,"Jack", "Queen", "King", "Ace"};
    suits={"Hearts", "Diamonds", "Spades", "Clubs"};
    fulldeck=Flatten[ Table[ {ranks[[j]], suits[[k]]}, {j,1,13},{k,1,4} ], 1 ];
    Return[ RandomSample[fulldeck, n] ];
  ];
```

This second version is simpler - so why would you ever use the first version where each card is assembled one at a time and possibly added to the hand (other than as an example of a different technique of course)? Once again it has to do with how much needs to be stored. Imagine a new type of deck with 10,000 ranks and 10,000 suits. The first version of CreateHand would need to store 2 10,000 element lists for a total of 20,000 stored elements. The second version needs to store those same 2 lists (so 20,000 elements) together with the full deck of all possible cards (which is 10,000 times 10,000 = 100,000,000 elements long). So the second version needs to generate and store 100,020,000 elements - which is a ton of additional overhead and going to be a lot harder on the resources available to the computer.

### Example 4: Is it a Flush?

Input: A hand generated by CreateHand.

Output: True if a randomly picked hand with n cards is a flush (all card have the same suit), False otherwise.

Discussion: This program will assume CreateHand has been defined (either version). To see if all cards have the same hand we can think of the hand as a matrix where the cards are the rows, the first column is the ranks, and the second column is the suits. If Union[hand[[All, 2]]] has just one element then hand would be a flush. This should be short enough to not require the use of a Block construct.

```
FlushQ[hand_]:= If[Length[Union[ hand[[All,2]] ] ]==1, True, False]
```

```

CreateHand[5]

{{4, Clubs}, {10, Clubs}, {Ace, Hearts}, {10, Diamonds}, {Jack, Hearts}}

FlushQ[%]

False

CreateHand[3]

{{Queen, Diamonds}, {9, Diamonds}, {Ace, Diamonds}}

FlushQ[%]

True

```

*checking to see if different hands are flushes*

Example 5: Counting proportions of flushes, version 1.

Input: A hand length  $n$  and a number  $k$  of hands to try.

Output: A numerical approximation of what proportion of the  $k$  hands were flushes.

Discussion: We will assume That CreateHand and FlushQ are already defined. We can simply create a list using FlushQ and count the True values.

```

FlushProportion[n_Integer, k_Integer]:=
  Block[ {numberofflushes, j, trials},
    (* trials is the sequence of FlushQ draws, numberofflushes how many flushes, j an iterator
    *)
    If[ k<1, Print["Not well defined"];Return[]];
    If[n<1||n>52, Print[ "Hand length is not well-defined"]; Return[ ] ];
    trials=Table[FlushQ[CreateHand[n]],{j,1,k}];
    numberofflushes=Count[trials, True]; Return[N[numberofflushes/k],10];
  ];

```

```

FlushProportion[5, 50 000]

0.001920000000

FlushProportion[7, 200 000]

0.00004500000000

```

*experimental probabilities of getting a flush*

The technique used above for counting a proportion of flushes in a random sequence of hands is not the best way to write the program. If you tried FlushProportion[7, 1000000000], the computer would have to create and store a sequence of Trues and Falses a billion entries long

which would be unnecessarily draining on the computer's resources. A better way to do it would not be to generate a list but rather generate the hands one at a time and up a counter every time a True is found. There would still be a billion hands but no hand would have to be stored and this would make the program more efficient.

Example 6: Counting proportions of flushes, version 2.

Input: A hand length  $n$  and a number  $k$  of hands to try.

Output: A numerical approximation of what proportion of the  $k$  hands were flushes.

Discussion: We will assume that CreateHand and FlushQ are already defined. We will use a Do loop to create hands and add to a running total of newly found flushes.

```
FlushProportion2[n_Integer, k_Integer]:=
Block[ {numberofflushes, j},
(* numberofflushes the running total, j an iterator *)
If[ k<1, Print["Not well defined"];Return[]];
If[n<1||n>52, Print[ "Hand length is not well-defined"]; Return[ ] ];
numberofflushes=0;
Do[ If[ FlushQ[CreateHand[n]], numberofflushes++];, {j,1,k}];
Return[ N[numberofflushes/k,10]];
];
```

**FlushProportion2[5, 50 000]**

0.002060000000

**FlushProportion2[7, 200 000]**

0.00004500000000

*more estimates of the probability of getting a flush*

FlushProportion is an example of using Mathematica to simulate an experiment. This is a very important thing to be able to do as in many cases to get accurate estimates of probabilities experiments need to be repeated such a large number of times that it is either too expensive or too time consuming (generating 200,000 poker hands would take forever by hand but Mathematica does it about 20 seconds).

All of our “random” functions assume that the numbers are chosen from some given range with equal probability (i.e. “uniformly”). Both RandomReal and RandomInteger can be used to generate random numbers from non-uniform distributions as well by using RandomReal[*continuousdistribution*,  $n$ ] and RandomInteger[*discretedistribution*,  $n$ ]. Some common distributions you might want to use:

Continuous Distributions:

NormalDistribution[  $m$ ,  $s$ ]: The bell-curve with mean  $m$  and standard deviation  $s$ .

StudentTDistribution[ $k$ ]: The Student T distribution with  $k$  degrees of freedom.

ExponentialDistribution[ $l$ ]: The exponential distribution with parameter  $l$ .

Discrete Distributions:

BinomialDistribution[ $n,p$ ]: A randomly generated “number of successes” from a list of  $n$  trials each of which has a chance of  $p$  to succeed.

GeometricDistribution[ $p$ ]: A randomly generated “number of trials before the first success”, where the chance of succeeding in any trial is  $p$ .

NegativeBinomial[ $n,p$ ]: A randomly generated “number of failures in repeated trials before getting  $n$  successes”, where the chance of succeeding in any trial is  $p$ .

## Homework Section 4.5 - Random Generators

Note: Some of these “programs” are really calculations.

Program 1: Write a program StandardDiceRoll[] (no inputs) which rolls 2 standard dice and returns the sum.

Program 2: Let rolldata be a list of 100,000 StandardDiceRoll outputs (for obvious reasons use a ; when you define this!). Use this list to estimate the probability of getting each result from 2-12. Make a histogram of the results as well.

Program 3: Write a program DieRollWithDoubles[] (no inputs) which rolls 2 standard dice and adds the results - except if you roll doubles you roll again and add the rolls together (ad infinitum!). So if you rolled {2,3} your total is 5, but if you roll {2,2} followed by {5,5} by {1,1} by {1,2} your total would be 19. (this is essentially Monopoly™ dice-rolling rules without the “3 doubles sends you to jail” option). (Programming hint: the possibility of doubles is best handled using a While loop in the form “while the 2 dice match reroll and add those to the running total)

Program 4: Let rolldata be a list of 100,000 DieRollWithDoubles outputs (again, use a semi-colon!). Use this list to estimate how likely each result is (because of the doubles some of the results will be larger than 12). Make a histogram of the results as well.

Program 5: Write a random generator RandomDisk[r,n] which selects n random complex number whose absolute value is r or less (this is essentially picking a random point in a solid disk of radius r centered at the origin). To make the selection uniformly random, create complex numbers within a square containing the disk and only keep the ones inside the disk.



Program 6: Write programs `BlackJackQ[hand]` and `BackJackProportion[n]` which work as the programs `FlushQ` and `FlushProportion`, except the hand is assumed to be a 2 card hand. (A blackjack is a 2 card hand with an ace and a "value 10" card [10-king]).

Program 7: Write programs `FullHouseQ[hand]` and `FullHouseProportion[n]` which function as `FlushQ` and `FlushProportion` do, where hands are assumed to have 5 cards. (A full house is 3 cards of one rank and 2 of another - 3 kings and 2 queens, for example). (Programming hint: Tally may be useful here).

Program 8: Write a program `RelativelyPrimeProportion[k,n]` ( $k$  and  $n$  integers) which measures the proportion of randomly generated integer pairs in the range  $[1,k]$  which are relatively prime (i.e. have a GCD of 1). (Fun math fact: If  $k$  were allowed to be infinity, as  $n$  gets large the proportion should approach  $6/\pi^2$ ).

Program 9: Some casinos don't use a single 52-card deck but rather several decks put together. Write a program `MultideckHand[n,c]` which takes  $n$  identical 52-card decks and creates a hand with  $c$  cards from the "multideck" (note: it will be possible to get the same card more than once in such a hand). As a programming hint it is probably easier to build the "multideck" using `Apply` and `Join` and then use `RandomSample`.

Program 10: The programs `FlushQ` and `FlushProportion2` can be easily modified work with hands created with `MultideckHand`. Based on experimental evidence for 5 card hands only, does the chance of getting a flush change as you increase the number of decks? (that is, try using a modified `FlushProportion2` on 500,000 trials with a standard deck, than 500,000 trials on a 2-deck multideck, then a 3 deck multideck, and so on). It may take about a minute for each run of `FlushProportion2` with 500,000 hands.

## Section 4.6 - Default Values and Program Options

Unlike many Mathematica functions the programs we have written so far have been very rigid in terms of their inputs – there are no options that could be set in any of them. But the Mathematica programs you write for yourself can be set up with options just like any other command – they can even use the same options as other Mathematica commands if they are set up properly.

The simplest case of an option would be when there is a single input variable which most of the time would be the same. For example, in our CreateHand command from the last section the most common number of cards in a hand is probably 5. So you might want to set up CreateHand so that unless told otherwise it would assume the number of cards was 5 – this would save you some typing if this command came up a lot. The way to create a default value for a single variable is to change how it is listed as an input in the function definition. So far we have used either `x_` or `x_Head` to define an input (as in `x_` or `x_Integer`). If we want that variable to have a default value of `y` (which does not have to be a number), we would change this to `x_:y` or `x_Head:y`. In the case of our CreateHand function we would start the definition as `CreateHand[n_Integer:5]:=` with the rest of the definition remaining the same. If we just evaluated `CreateHand[ ]`, it would assume we wanted a 5 card hand but `CreateHand[7]` would still give us a 7 card hand.

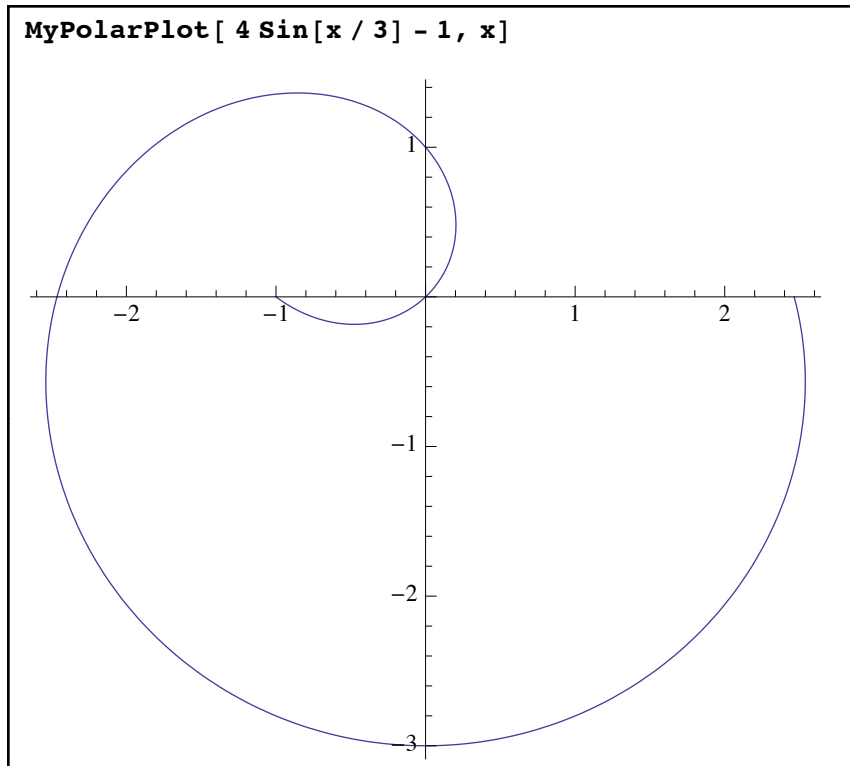
Example 1: A simple polar plotting function

Inputs: A function `mess`, a variable `x`, and an optional range `{a,b}` with default value `{0,2 $\pi$ }`

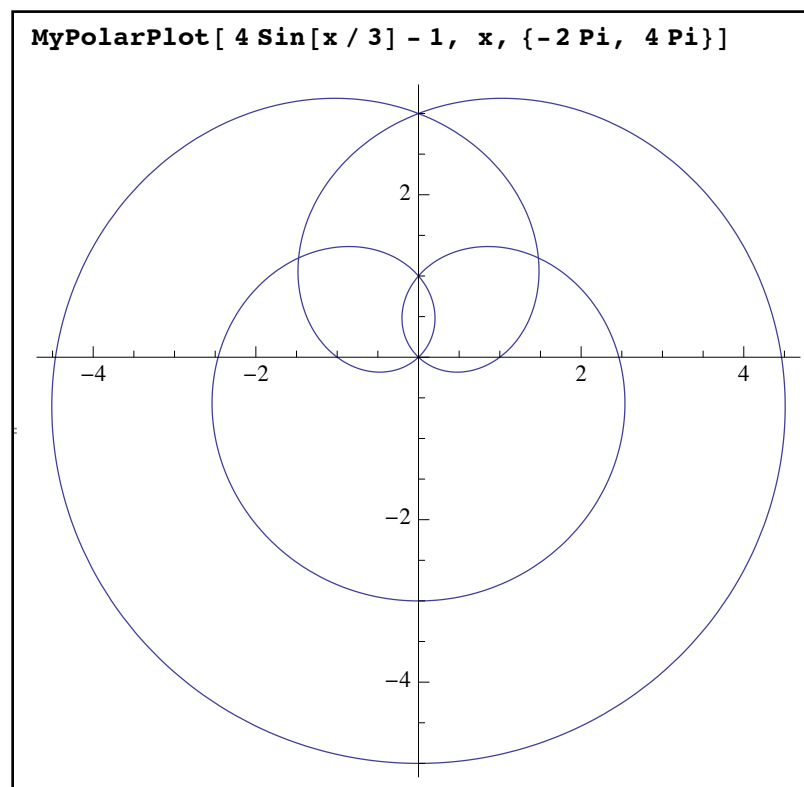
Output: The polar graph of “`mess`” over the appropriate range, with proper scales.

Discussion: Since we are building our own simple polar plotter, it would be cheating to use `PolarPlot` directly. The polar graph can be obtained via `ParametricPlot` – the parametric definition of `r=mess` where the angle is `x` is `{mess*Cos[x], mess*Sin[x]}`.

```
MyPolarPlot[ mess_, x_, range_: {0,2Pi} ]:=
  Block[ {start, finish, graph},
    (* start and finish are the limits of the graph *)
    start=range[[1]];
    finish=range[[2]];
    graph=ParametricPlot[ {mess*Cos[x], mess* Sin[x]}, {x,start, finish}, AspectRatio →
Automatic];
    Return[graph];
  ];
```



*MyPolarPlot graphing a function using a default range of 0 to  $2\pi$*



*MyPolarPlot graphing a curve over a given range*

You can use the colon notation to define a function with more than one “optional” variable but you need to be careful when you do so. You could define a function that starts with `f[x_:20, y_:40, z_:60]:=...` to set the default values of `x`, `y`, and `z` to be 20, 40, and 60. However any values that you give as inputs would be assumed to be `x` first, then `y`, and only then `z`. That is if you evaluate `f[50,70]`, it would assume that `x=50` and `y=70` and give `z` the default value of 60. It would not be possible to use the default value of `x` and then specify values for `y` and `z` (so if you want `x` to be 20, `y` to be 10, and `z` to be 20 you would have to use `f[20,10,20]` even though `x` has a default value of 20).

For this reason and others specifying default values for variables will not give you enough control for more complicated programs. In many cases you will want to set up true options for your program in the same way Plot has options like `AspectRatio`, `PlotRange`, etc. Defining options and using them in programming is a little more complicated then setting default values for input variables so we will break the process into 3 parts.

The first thing you will need to do is to define the option names for your function and what their standard values will be (for example, at some point someone decided that Plot would have an option called `Filling` whose default value was `None`). The command for defining your program’s options is `Options`. `Options` is a command which is used outside your program’s definition to set up the various option names and values. The format is simply to give the options in a list; `Options[programname] = {option1name → option1value, option2name → option2value, ... }`. For example, for a plotting program `MyPlot` you might have `Options[MyPlot] = {Origin → {0,0}, Labels → {"x", "y"}, PlotSize → 360}`. It is traditional (and safer) to include the `Options` definition for a function in the same cell as the function itself so the options get defined whenever the function does. The `Options` command can be thought of as a function which takes in names and returns the options list for that name. In fact you can use it to see the complete list of options for any built-in Mathematica command:

```
Options[Solve]
```

```
{InverseFunctions → Automatic, MakeRules → False, Method → 3,
  Mode → Generic, Sort → True, VerifySolutions → Automatic, WorkingPrecision → ∞}
```

```
Options[RowReduce]
```

```
{Method → Automatic, Modulus → 0, Tolerance → Automatic, ZeroTest → Automatic}
```

*options for some built-in commands*

Once you have defined your program’s options you need to add the possibility of options to your function’s sequence of inputs. The easiest way to do this is to add a new variable name at the end of your input list (say, “`opts`”), but instead of following it with a single underscore follow it with three underscores. Three underscores denotes a “blank sequence” in Mathematica, essentially a braceless list of 0 or more objects (a double underscore requires the sequence to have at least 1 object in it - in most cases you want to allow for no options to be specified by the

user, so the triple underscore is more common). For example if you wanted to create your own graphing function `MyPlot` you would definitely need a formula, variable, and range followed by the possibility of adding options. The definition in that case might look like

$$\text{MyPlot}[\text{formula\_}, \{x\_ , a\_ , b\_ \}, \text{opts\_}]:= \dots$$

where `opts` represents the options a user might add.

Once the options have been allowed for and their default values set you are ready to introduce them inside the main program. To do this each default variable from your `Options` command should have a regular program variable waiting to accept it. For example I previously defined `Options[MyPlot]= {Origin→{0,0}, Labels→{"x", "y"}, PlotSize→360}`. In my `MyPlot` program maybe I want the variable "center" to represent the value of `Origin`, "names" to represent the values from `Labels`, and "resolution" to take on the value of `PlotSize`. Three options, three variables to store them. The command to assign the values is a bit complex and refers to the option sequence name from the function definition (which I'll call "opts"):

$$\{\text{list of program variables}\} = \{\text{list of corresponding option names}\} /. \{\text{opts}\} /. \text{Options}[\text{function name}]$$

There are two things to be careful about. First, the *list of corresponding option names* does not include the arrow or values – just the names themselves. Second, the order of the program variables needs to match the order of the option names. For example, in `MyPlot[formula_, {x_,a_,b_}, myoptions_]:=... function` and the `Options` and variables defined above I would use:

$$\{\text{center, names, resolution}\} = \{\text{Origin, Labels, PlotSize}\} /. \{\text{myoptions}\} /. \text{Options}[\text{MyPlot}]$$

This would cause the variables `center`, `names`, and `resolution` to take on the default values from the `Options` command unless an option like `PlotSize→720` was used in the inputs. These program variables can then be used in the program like any other variable.

For an example of using options in a program we will now write a program for doing numerical Riemann sums from Calculus. The Riemann sum is an estimate for the area underneath a curve over an interval. The way this is usually done is to divide the interval up into equally long pieces, pick one point from each interval, and then form the rectangles whose bases are the sub-intervals and whose heights are given by the height of the curve over the chosen points. The Riemann sum is the sum of the areas of those rectangles. The most common choices for the points which determine the heights are the left endpoints of the subintervals and the right endpoints of the subintervals, so we will use an option `HeightPoints` as well as an option for the number of decimal places to be used.

Example 2: Numerical Riemann Sums

Inputs: A formula  $f$ , a variable/interval  $\{x, a, b\}$ , a number of intervals  $n$

Options: HeightPoints (set to either "Left" or "Right", default "Left"), and Digits (number of digits for the estimate, default 20)

Output: The numerical estimate of the Riemann sum for  $f$  on  $[a, b]$  using  $n$  subintervals and the chosen point scheme.

Discussion: If  $dx$  is the length of one of the subintervals then the left endpoint of the  $j$ -th subinterval is  $a+(j-1)*dx$  and the right endpoint is  $a+j*dx$ . So if there is a variable  $m$  set to 1 for left points and 0 for right points, we can use a single formula  $a+(j-m)*dx$  for both.

```
NRiemannSum[ f_, {x_, a_, b_}, n_Integer, opts___]:=
  Block[{dx, rsum, j, m, dig, pts},
    (* dx is the length of a subinterval, j an iterator *)
    (* m is 1 or 0 for Left or Right points from pts *)
    (* dig is the number of digits to keep, rsum the Riemann sum *)
    If[n<1, Print["Use a positive number of intervals"];Return[]];
    {pts, dig}={HeightPoints, Digits} /. {opts} /. Options[NRiemannSum];
    If[pts!="Right",m=1,m=0];
    dx=(b-a)/n;
    rsum=Sum[ (f /. x-> a+(j-m)*dx)* dx, {j,1,n}];
    Return[N[rsum, dig]];
  ];
Options[NRiemannSum]={HeightPoints->"Left", Digits->20};
```

```
NRiemannSum[ x^3, {x, 0, 2}, 30, Digits -> 30]
3.7377777777777777777777777777778

NRiemannSum[ x^3, {x, 0, 2}, 30, HeightPoints -> "Right", Digits -> 12]
4.271111111111
```

#### *Riemann sums*

In this program if the "heights" are not explicitly given as "Right", the assumption is that they should be taken as "Left" points. A more complex (and better) way to handle this would be to include an error-checking line which would print a message and stop the program if "HeightPoints" was set to something other than what you allowed for in your program (in this case "Left" or "Right").

#### Example 3: Specifying kinds of solutions

Inputs: An equation (eqn) in one variable  $x$ .

Options: SolutionType, which can be set to All (default), "Reals", or "NonReals".

Outputs: The solution to the equations which are of the appropriate type (this is similar to

using Solve with a domain, but there isn't a domain for non-reals).

Discussion: Since we will be using Solve we can assume the equation will be algebraic and have roots that are at worst complex. We can print a message if some of the roots are not explicit numbers (for example, those that involve quantities like  $\text{Sqrt}[a]$ ), and then return the solutions of a given type.

```
SolveType[eqn_, x_, opts___]:=
Block[ {fullsols, numericssols, nonnumericssols, soltype},
(* fullsols is the full solution set *)
(* numericssols are solutions which are explicit numbers *)
(* nonnumericssols are solutions which are not explicit numbers *)
(* soltype is the solution type to be shown *)
(* assign values from options and error check *)
{soltype}={SolutionType} /. {opts} /. Options[SolveType];
If[ !MemberQ[ {All, "Reals", "NonReals"}, soltype], Print[ "Invalid solution type"];
Return[ ] ];
(* make sure eqn is really an equation using the head Equal *)
If[ Head[eqn]!=Equal, Print[ "Invalid equation"]; Return[ ] ];
(* solve the equation *)
fullsols= x/. Solve[ eqn,x];
(* separate numerical and non-numerical solutions *)
numericssols=Select[ fullsols, NumericQ];
nonnumericssols=Select[ fullsols, !NumericQ[#1]&];
If[ nonnumericssols!= {}, Print[ "Non-numeric solutions ", nonnumericssols, " found." ] ];
If[ soltype==All, Return[numericssols] ];
If[ soltype=="Reals", Return[ Select[numericssols, Element[#1 , Reals]& ] ] ];
If[ soltype=="NonReals", Return[ Select[numericssols, !Element[#1, Reals]& ] ] ];
];
Options[SolveType]={SolutionType→ All};
```

**SolveType[ $x^2 + 3x + 1, x$ ]**

Invalid equation

**SolveType[ $x^3 + 3x + 1 == 0, x$ ]**

$$\left\{ -\left(\frac{2}{-1 + \sqrt{5}}\right)^{1/3} + \left(\frac{1}{2}(-1 + \sqrt{5})\right)^{1/3}, -\frac{1}{2}(1 + i\sqrt{3})\left(\frac{1}{2}(-1 + \sqrt{5})\right)^{1/3} + \frac{1 - i\sqrt{3}}{2^{2/3}(-1 + \sqrt{5})^{1/3}}, \right. \\ \left. -\frac{1}{2}(1 - i\sqrt{3})\left(\frac{1}{2}(-1 + \sqrt{5})\right)^{1/3} + \frac{1 + i\sqrt{3}}{2^{2/3}(-1 + \sqrt{5})^{1/3}} \right\}$$

**SolveType[ $x^3 + 3x + 1 == 0, x, \text{SolutionType} \rightarrow \text{"Reals"}$ ]**

$$\left\{ -\left(\frac{2}{-1 + \sqrt{5}}\right)^{1/3} + \left(\frac{1}{2}(-1 + \sqrt{5})\right)^{1/3} \right\}$$

**SolveType[ $x^3 + 3x + 1 == 0, x, \text{SolutionType} \rightarrow \text{"NonReals"}$ ]**

$$\left\{ -\frac{1}{2}(1 + i\sqrt{3})\left(\frac{1}{2}(-1 + \sqrt{5})\right)^{1/3} + \frac{1 - i\sqrt{3}}{2^{2/3}(-1 + \sqrt{5})^{1/3}}, \right. \\ \left. -\frac{1}{2}(1 - i\sqrt{3})\left(\frac{1}{2}(-1 + \sqrt{5})\right)^{1/3} + \frac{1 + i\sqrt{3}}{2^{2/3}(-1 + \sqrt{5})^{1/3}} \right\}$$

**SolveType[ $x^3 + 3x + 1 == 0, x, \text{SolutionType} \rightarrow \text{"Algebraic"}$ ]**

Invalid solution type

**SolveType[ $x^2 - a == 0, x, \text{SolutionType} \rightarrow \text{"Reals"}$ ]**

Non-numeric solutions  $\{-\sqrt{a}, \sqrt{a}\}$  found.

$\{\}$

### *solutions to equations of different types*

When checking to make sure “eqn” was a true equation it was necessary to use  $!=$  (“is not identical to”) rather than  $!=$  (“not equal to”) since the heads of objects are not numbers, and  $==$  does not return False if the objects are not directly comparable (try evaluating  $3==\text{True}$  to see another example of where  $==$  doesn’t behave as you would expect). We could have used  $===$  as well, which is the “is identical to” positive form of  $!=$ .

In many of your own programs you may want to use the option lists in standard Mathematica functions – for example, if your program generates some graphics you may want to pass options like PlotPoints or AspectRatio to the Plot command which generates the graphics. If your program uses options other than those the Plot command can accept this can be a problem. Fortunately there is a command which can pick out which options from a given set can be used in another command - FilterRules. To pick out which parts of a list of rules can be used in in



*command*, use the form `FilterRules[ listofrules, Options[ command ] ]`. The result will be a list of those options from *listofrules* that can be used in *command*. As `FilterRules` will return a list of rules rather than a sequence of rules, you will often have to wrap the command `Evaluate` around the results when you use them in commands like `Plot`.

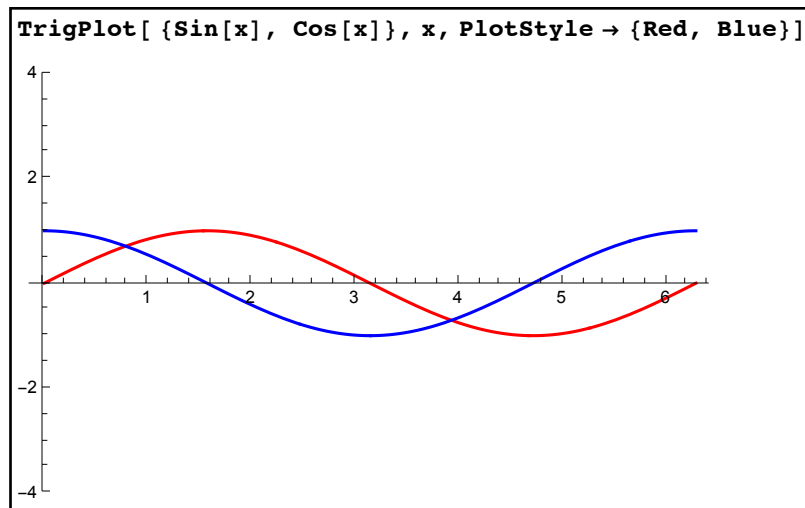
#### Example 4: TrigPlot

Inputs: A function or list of functions graphs to be plotted, and a variable *x*.

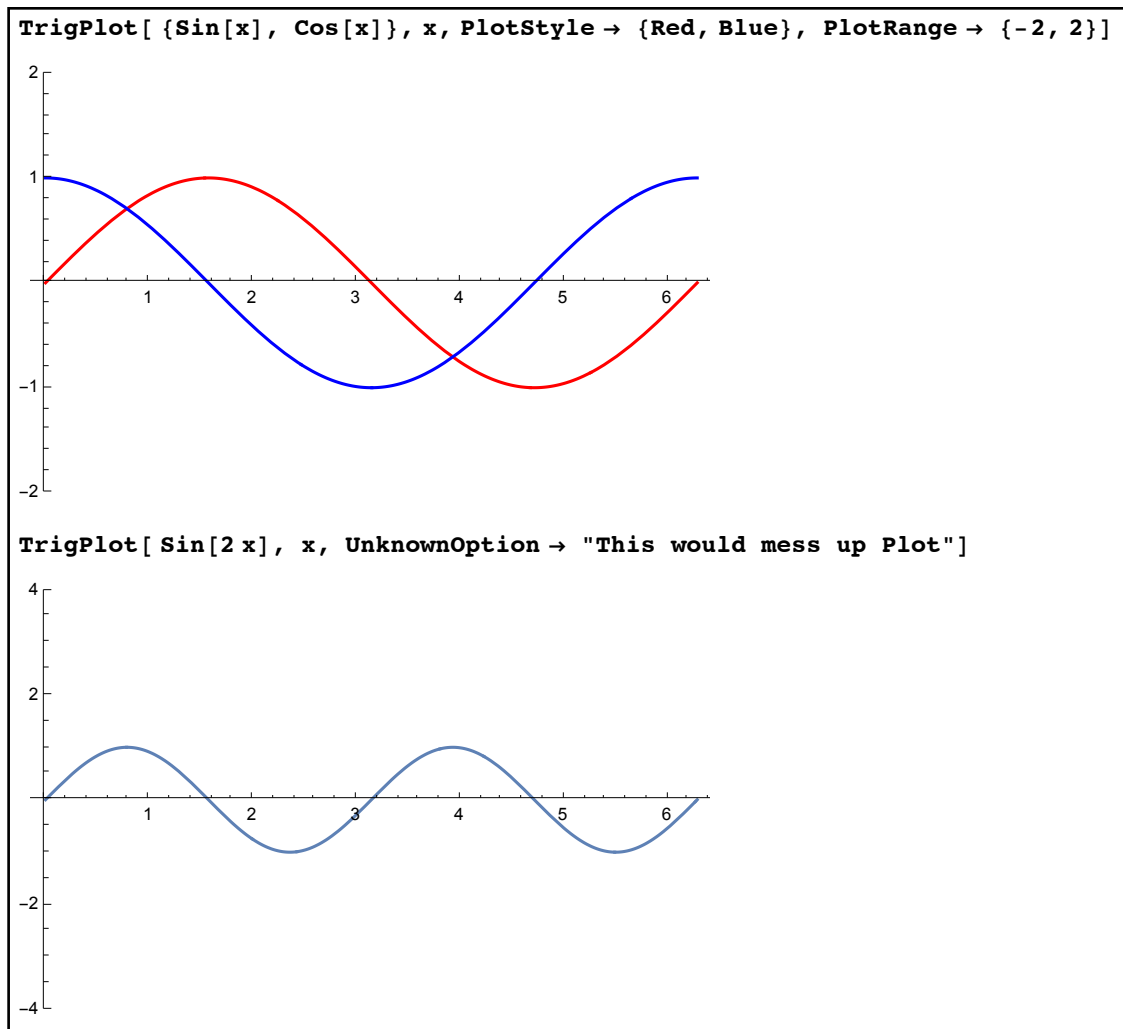
Options: Any acceptable for a `Plot` command.

Output: The graph of the functions from 0 to  $2\pi$  and  $-4$  to  $4$  with the given options.

```
TrigPlot[ functions_, x_, opts___]:=
Block[ {plotopts},
(* plotopts are those options from the list which are valid for Plot *)
plotopts=FilterRules[ {opts}, Options[Plot] ];
Return[ Plot[ functions, {x,0,2Pi}, Evaluate[plotopts], PlotRange→{-4,4} ] ];
];
```



*TrigPlot passes the PlotStyle option directly to Plot*



*passing options to Plot via FilterRules*

There are 2 subtle things to note about the way commands are used in the TrigPlot definition. First, in the FilterRules command “opts” had to be placed in list braces - FilterRules works on lists, not sequences. Second, the placement of PlotRange in the Plot command makes a big difference. By placing it after Evaluate[plotopts] it won’t be “seen” by Plot if plotopts itself contains a PlotRange option (as earlier options overwrite later ones). If the PlotRange had come before Evaluate[plotopts], then the plotted range would always be -4 to 4 regardless of what the user entered.

Before we begin the next section there is one additional topic which can be useful in practice – the notion of a “help” line for your function that you can access using the ? command. Like Options the “help line” (called a usage message) is defined outside the main program. To create a usage comment for your program use a command like `FunctionName::usage = “your description here”`. In the last example we could have defined a usage comment as `TrigPlot::usage = “TrigPlot[f,x] creates a graph of the function/list of functions f in the variable x. x goes from 0 to 2Pi, y goes from -4 to 4. TrigPlot accepts all Plot options.”`. Usage comments

are very helpful as you build a library of programs over time or if you share programs with other people. It is best practice to put the Options and any usage remarks in the same cell as your program so they are always evaluated together.

```
In[154]:= TrigPlot[functions_, x_, opts___] :=
  Block[{plotopts},
    (*plotopts are those options from the list which are valid for Plot*)
    plotopts = FilterRules[{opts}, Options[Plot]];
    Return[Plot[functions, {x, 0, 2 Pi}, Evaluate[plotopts], PlotRange -> {-4, 4}]];];
Options[TrigPlot] = {};
TrigPlot::usage =
  "TrigPlot[f,x] creates a graph of the function/list of functions f in the variable x. x
  goes from 0 to 2Pi, y goes from -4 to 4. TrigPlot accepts all Plot options.";

In[157]:= ? TrigPlot
```

TrigPlot[f,x] creates a graph of the function/list of functions f in the variable x. x goes from 0 to 2Pi, y goes from -4 to 4. TrigPlot accepts all Plot options.

*adding a blank option line and a usage statement to TrigPlot*

## Homework Section 4.6 - Default Values and Program Options

- 1) In defining a program, how would you specify a variable x to be an integer with a default value of 7?
- 2) Explain the difference between `__` and `___` in allowing for options in a program.
- 3) Explain the difference between a list and a sequence in Mathematica.
- 4) What is FilterRules used for? How is the result from FilterRules used in another command?
- 5) What is a usage comment, and how do you add one to a program?

Program 1: Modify the program TrigPlot from this section so you can modify the x-values used in the graph via an option Domain (where the default for Domain is {0,2Pi}). Include a usage comment.

Program 2: Modify the program NRiemannSum program to allow for a 3rd option "MidPoint", and add a usage comment.

## Section 4.7 - Raw Graphics

In all of the programs in which we have used graphics so far we have relied completely on graphics generated completely by Mathematica commands like Plot (or one of its variations like ParametricPlot). There are many useful programs that can be written just using those commands but there will be times when you will need to make your own “custom” graphics. Imagine trying to graphically display the previously-mentioned Riemann sum using the rectangles beneath the curve – it is easy to generate the curve itself using a Plot command but trying to create the rectangles through the use of a Plot command would be amazingly long and tedious. Fortunately Mathematica allows you to build up custom graphics from simple shapes using a command called Graphics. Graphics[*list*] represents the raw undisplayed form of *list*, where *list* is a list of individual objects (points, lines, etc.) and graphics directives (such as Hue, Thickness, RGBColor, color names, etc.). To see the graphics you simply use the Show command just as you would for redisplaying standard graphs. The more common graphics objects are:

**Text:** Text[ *string*, {*x*,*y*} ] places a string of text (encased in double quotes) in the graphic, centered at the point (*x*,*y*). You can also specify text alignment in the left-right and up-down directions (the default is to have things centered) using the form Text[ *string*, {*x*,*y*}, {*horizontal*, *vertical*}], where *horizontal* can be Left, Center, or Right and *vertical* can be Top, Center, or Bottom. It is also possible to rotate text but that is beyond the scope of what we will need.

**Point:** Point[ {*x*,*y*} ] represents a point at the given coordinates. You can also specify several points by using the form Point[ { *point1*, *point2*, ... } ], where each point is a 2 element coordinate list.

**Line:** Line[ {*point1*, *point2*, ...} ] represents the line segments joining the points in the order given. The last point is not connected to the first point, so if you want to “close a loop” you would have to start and end your list of points with the same thing.

**InfiniteLine:** InfiniteLine[ { {*x1*,*y1*}, {*x2*,*y2*} } ] represents the infinite line through the two points. InfiniteLine[ {*x1*,*y1*}, {*a*,*b*} ] represents the infinite line through (*x1*,*y1*) and then goes *a* units to the right and *b* units up (*a*=0 corresponds to a vertical line). The infinite lines will automatically extend to the edges of the given graphic.

**Rectangle:** Rectangle[ {*a*,*b*}, {*x*,*y*} ] is the solid rectangle whose opposite corners are {*a*,*b*} and {*x*,*y*}.

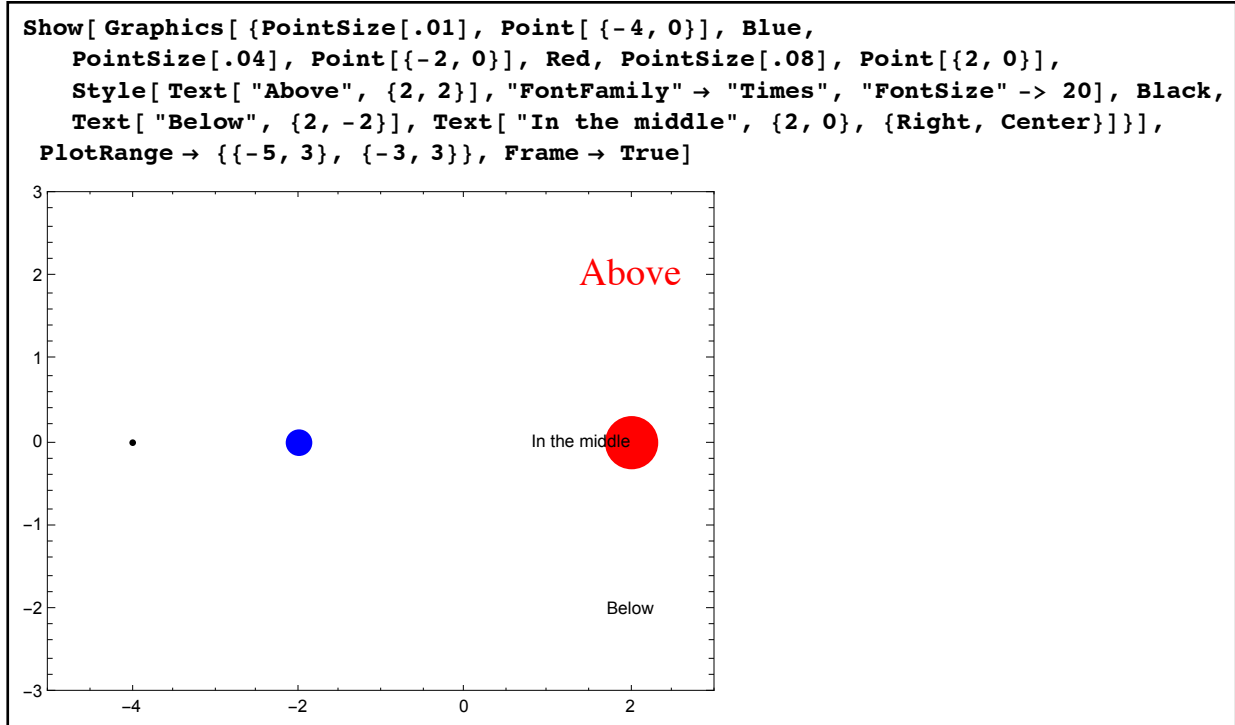
**Polygon:** Polygon[ {*point1*, *point2*, ...} ] is the solid polygon formed by connecting the points (all of which are given by coordinate pairs) in the given order. The last point in the list will be connected to the first point, so no duplication is necessary.

Circle: `Circle[ {x,y}, r]` represents a circle of radius  $r$  centered at  $\{x,y\}$ .

Disk: `Disk[ {x,y}, r]` represents a solid disk of radius  $r$  centered at  $\{x,y\}$ .

Common directives include all of those we've seen previously: `Thickness`, `Hue`, `PointSize`, `RGBColor`, `GrayLevel`, `Dashing`, various color names and generic directives like `Thick`, and a more general form of `PointSize`. `PointSize[  $p$  ]`, where  $0 < p < 1$ , represents the size of the points relative to the graphic itself (sort of a `Thickness` for points - and like `Thickness` you generally use values close to 0). In a list of raw graphics objects and directives each directive applies to everything in the list that follows it. So if you use `Blue` in your list everything after it will be blue unless you use another directive like `Black` later in the list. Here are some examples of some raw graphics:

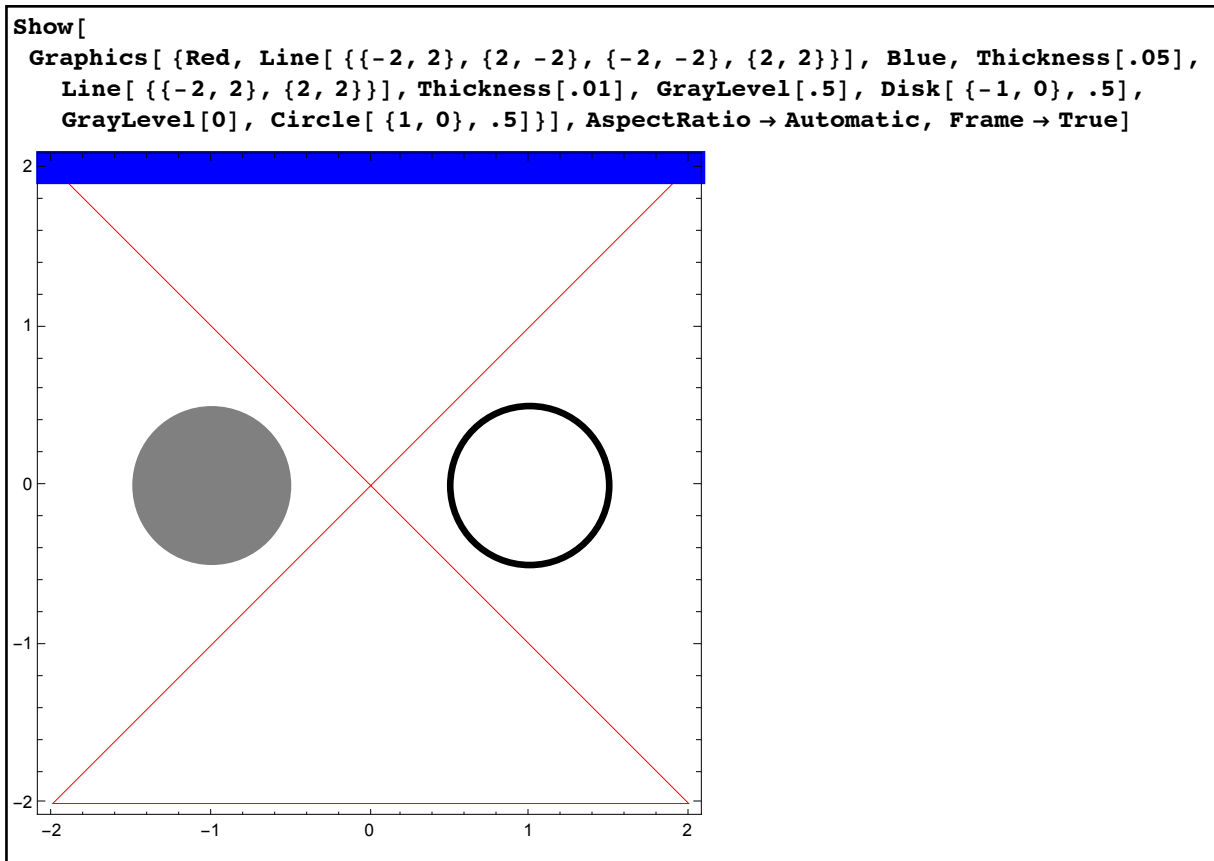
#### Example 1: Points and Text



*points and text*

The use of `PlotRange` in the `Show` command is not always necessary but is sometimes useful in avoiding cropping the overall image or adding blank space around the created graphic (which may be important if you are exporting the image for use in another program). Note that the options “`FontFamily`” and “`FontSize`” cannot be used inside `Text` - you have to wrap a `Style` command around your `Text` command to change the font and size.

#### Example 2: Lines, circles, and disks

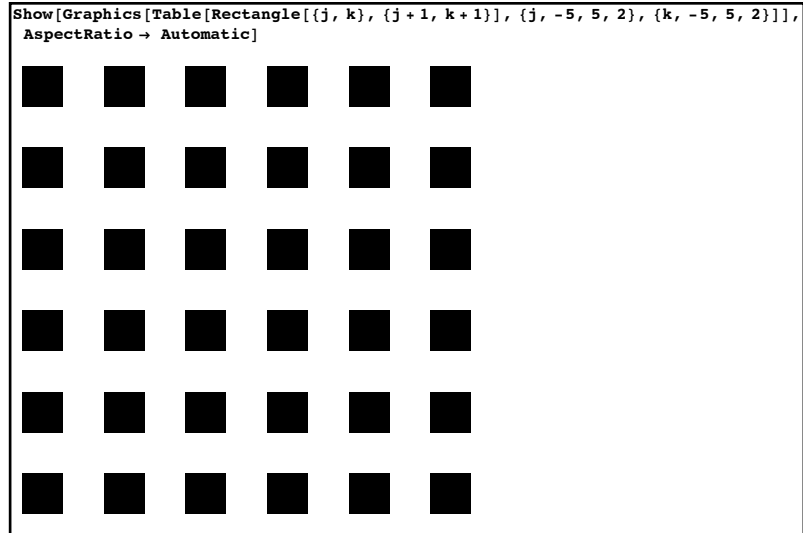


*lines, circles, and disks*

Again some care must be taken to "reset" commands which modify graphics, in this case Thickness (without setting it to .01 the circle would appear as thick as the blue line). Note that the blue line lies on top of the red lines - in general graphics which come later in the list are placed over those that came earlier. The AspectRatio option makes sure the right scales are used, preserving the circular shapes.

Example 3: A table of rectangles:

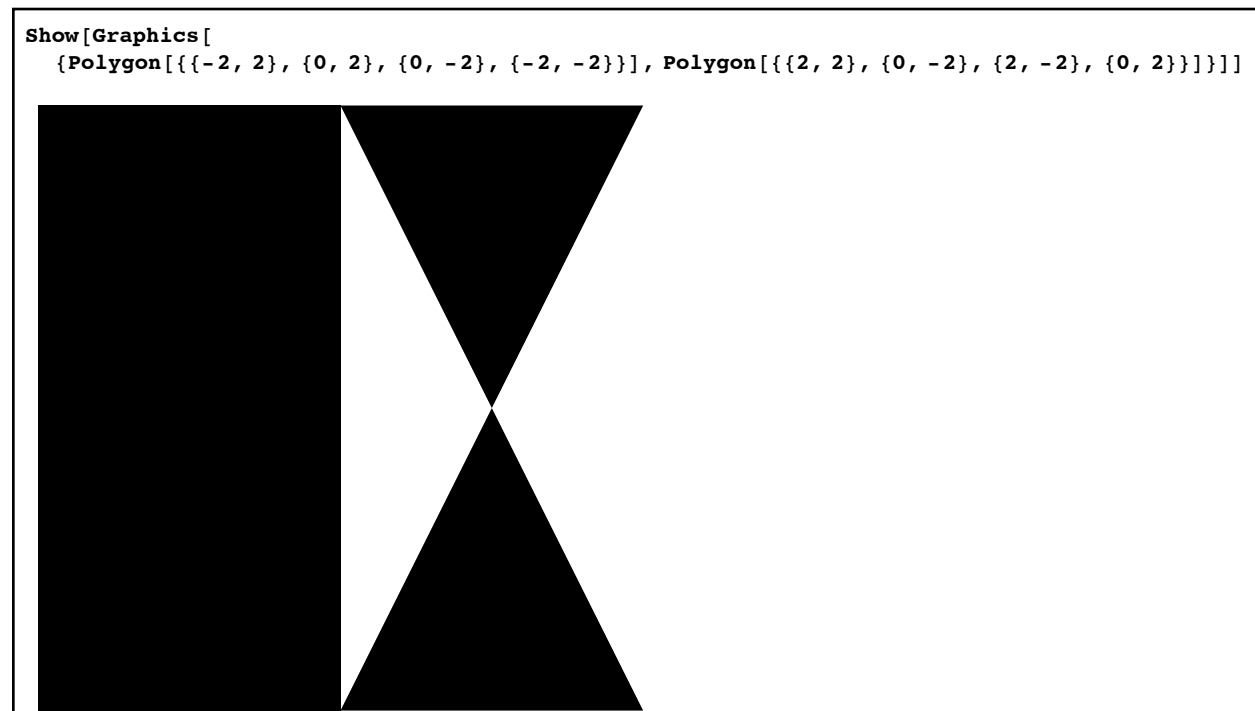
Here is an example of using Table to generate a list to use with Graphics - in this case Table[ Rectangle[ {j,k}, {j+1, k+1}], {j,-5,5,2}, {k,-5,5,2}]. Note that even though the Table is two-dimensional (i.e. a list of lists), it still works correctly with Graphics without needing to be "flattened" into a regular list of objects.



*a grid of rectangles via Table*

If you look at the command that generated these rectangles you'll notice that the Graphics command is not wrapped around a simple list but rather a matrix of Rectangle objects. In general Graphics is not restricted to a 1-dimensional list so there is no need to use a command like Flatten if you are generating your graphics objects from a Table or similar command.

#### Example 4: Order matters in Polygon



*two polygons, one convex, one concave*

As we can see in this example the results of a Polygon object are somewhat dependent on the order in which the points are placed in the list (just as we saw in Sketchpad).

Now on to some examples of using these graphics objects in different programs:

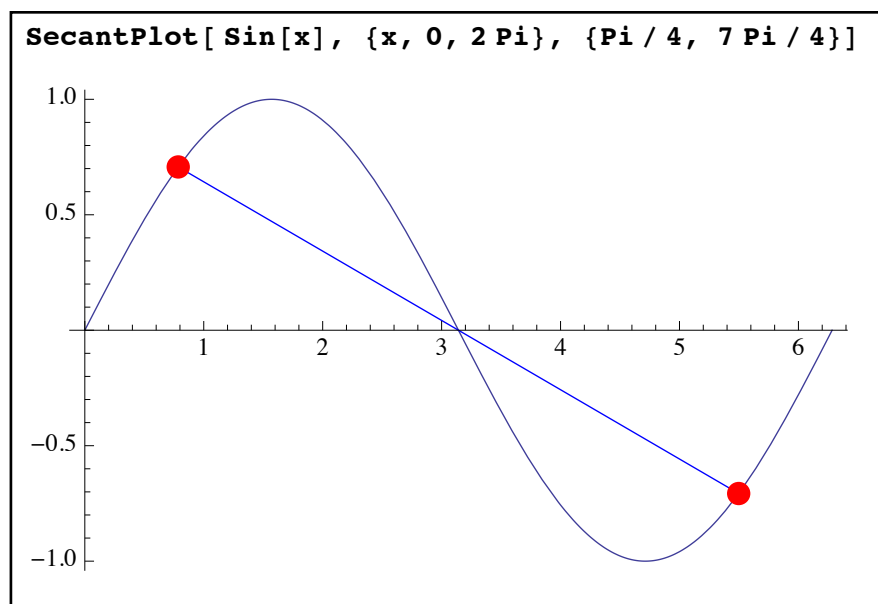
Example 5: Plotting a secant line segment:

Inputs: A function, a variable  $x$  and a range to graph over, 2  $x$ -values  $s$  and  $t$  for the secant line.

Options: Any for Plot

Output: The graph of the function together with the line segment joining the points on the curve corresponding to  $x=s$  and  $x=t$ . The points will be red and the segment blue.

```
SecantPlot[ function_, {x_, a_, b_}, {s_, t_}, opts___]:=
Block[ {maingraph, rawgraphics,ys, yt, plotopts},
(* maingraph is the graph of the function, rawgraphics the points and line *)
(* ys/yt are the y-values for s and t, plotopts the passed options *)
If[ a ≥ b, Print[ "Invalid plot range"]; Return[] ];
If[ !( a ≤ s ≤ b && a ≤ t ≤ b), Print[ "The points are not in the given range"]; Return[] ];
ys= function /. x→s;
yt=function /. x→ t;
plotopts=FilterRules[ {opts}, Options[Plot]];
rawgraphics={Blue, Line[ {{s,ys}, {t,yt}},PointSize[.03], Red, Point[ {{s,ys}, {t,yt}}]};
maingraph=Plot[ function, {x,a,b}, Evaluate[plotopts]];
Return[ Show[ maingraph,Graphics[ rawgraphics ] ] ];
];
```



*graphing secant line segments*



### Example 6: Solid regular polygons

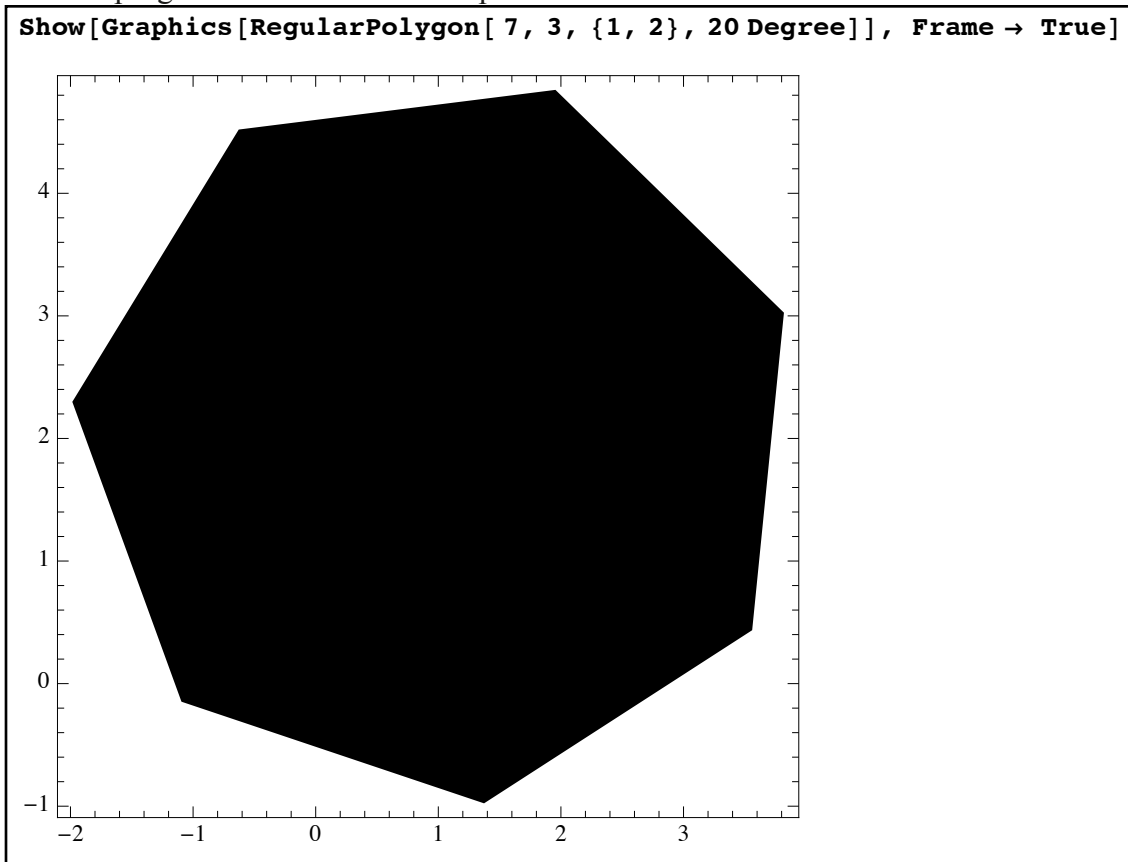
Input: A number of sides  $n$ , a radius  $r$  (the distance from the polygon center to any corner), a center  $\{a,b\}$ , and an angle which will determine the first point of the polygon (with 0 corresponding to directly to the right of the center,  $\pi/2$  directly above the center, etc.).

Options: By default we can make the angle 0

Output: The regular polygon described by the center, radius, and angle offset for a corner.

```
RegularPolygon[ n_, r_, {a_,b_}, angle_:0]:=
Block[ {k, cornerlist},
(* k is an iterator, cornerlist the corners of the polygon *)
cornerlist=Table[ r*{ Cos[ 2 Pi k/n + angle], Sin[ 2 Pi k/n + angle] }+{a,b}, {k,0,n-1}];
Return[ Polygon[cornerlist] ];
];
```

As the result of this command is a raw graphics object, to see the polygon we will need to combine our program with Show and Graphics:



*a regular heptagon of radius 3 centered at (1,2) with an offset of  $20^\circ$*

The  $\{\text{Cos}[\text{theta}], \text{Sin}[\text{theta}]\}$  format is a way of locating points on the unit circle – by dividing up  $2\pi$  into  $n$  parts we get points equally spaced around the circle and adding a fixed angle rotates

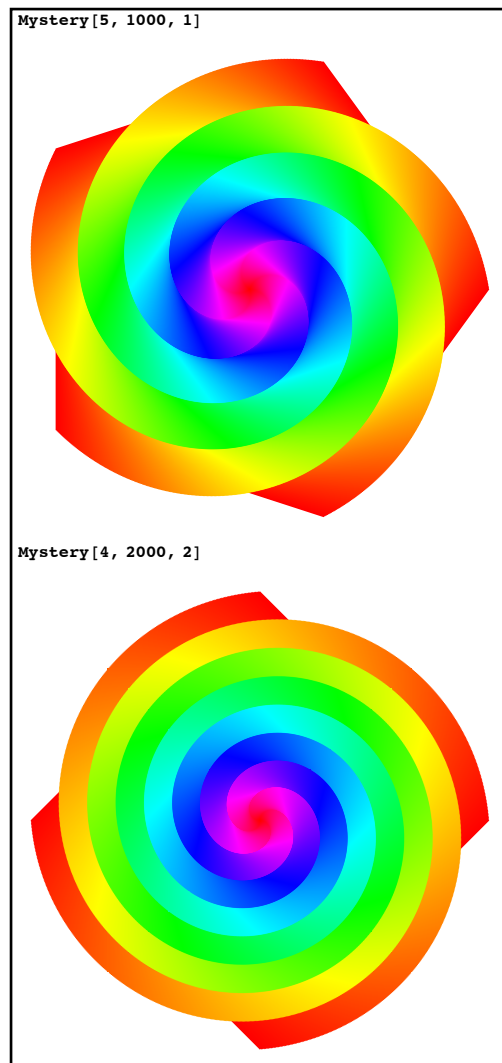
them as desired. Multiplying the coordinate list by  $r$  changes the radius and adding the center  $\{a,b\}$  performs the shift.

See if you can deduce what the following program (which assumes `RegularPolygon` has been defined) will do:

Example 7: ????

Input: Three integers  $n,k,m$

```
Mystery[n_,k_,m_]:=
  Block[{j, polytable},
    polytable=Flatten[ Table[{Hue[j/k],RegularPolygon[n,1-j/k,{0,0}, 2Pi j*m/k]},{j,0,k}]];
    Return[Show[Graphics[polytable], AspectRatio→Automatic]];
  ];
```



*output from the Mystery program*

Our next example involves creating a program which will animate the notion of the derivative as the limit of secant lines:

#### Example 8: Animating a derivative

Inputs: a function `mess`, a variable `x`, an interval `[a,b]` to graph over, a y-range `{c,d}` to use for plots, a point `e` at which to base the derivative, the number `n` of animation frames, and an optional variable `side` to determine which side the derivative should be on (defaulting to "Right")

Outputs: A sequence of animation frames showing the line at the point on the graph corresponding to  $x=e$  and a variable point coming in from the left or right depending on the value of the variable `sides`. `n` frames will be used, with the starting `x`-value for the second point on the line either being `a` or `b`.

Discussion: The program will essentially come in two very similar parts depending on whether we animate from the right ( $x=b$ ) or the left ( $x=a$ ). These parts will be embedded in 2 If statements. We will need to find the equation of the lines at each step so they can be graphed easily from  $x=a$  to  $x=b$ . In addition we will put the two points in red on the curve slightly larger than normal for emphasis. The graphs shown will need to use `AspectRatio` to make sure the scales are correct. The animation will be created via the command `ListAnimate` (and its option `AnimationRate`, which controls the number of frames per second), which creates an animation from a list of graphics.

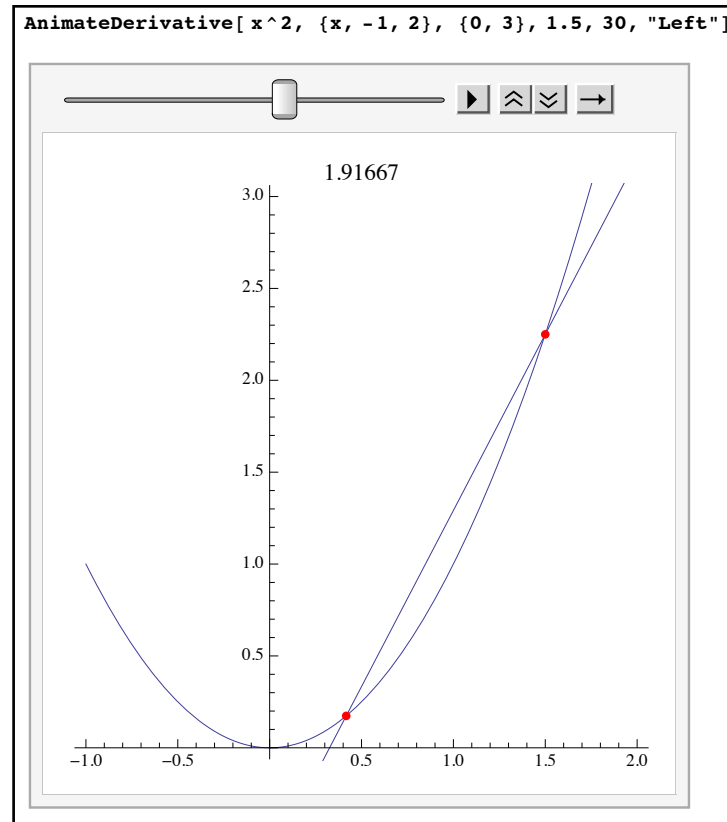
As this is a more complex program than many we have attempted, it is a good idea to add extra comment lines to track the parts of the program.

```
AnimateDerivative[mess_, {x_, a_, b_}, {c_, d_}, e_, n_, side_ : "Right"] :=
  Block[ {maingraph, slope, fullpic, lineformula, linegraph, mainpoint, newx, newpoint,
    stepsize, k, rawgraphics, graphicslist},
    (* maingraph is the graph of the function, linegraph the graph of the line, rawgraphics the
    points, fullpic a full frame of animation, graphicslist the master list for the animation *)
    (* mainpoint is the point corresponding to  $x=e$  *)
    (* stepsize will be the change in  $x$  from graphic to graphic *)
    (* newx is the  $x$ -value of the 2nd point, newpoint its coordinates *)
    (* slope will be the calculated slope, lineformula the point-slope form of the line *)
    (* k is an iterator *)
    (* Error checks *)
    If[ a>=b, Print[ "Invalid range."];Return[] ];
    If[ !(a<=e<=b), Print[ "Invalid point."]; Return[] ];
    (* Program start *)
    (* Setting up the values which will not change from one picture to the next *)
    maingraph = Plot[ mess, {x, a, b}];
```

```

(* the stepsize is found by breaking the right or left side into n equal pieces *)
stepsize = If[ side == "Right", (b - e)/n, (e - a)/n];
mainpoint = {e, mess /. x → e};
graphicslist = {};
(* The Do loop to create the plots *)
Do[
  (* creating the second x-value and point *)
  newx = If[ side == "Right", b - k *stepsize, a + k*stepsize];
  newpoint = {newx, mess /. x → newx};
  (* finding the slope, the line, and its graph *)
  slope = (newpoint[[2]] - mainpoint[[2]])/(newx - e);
  lineformula = slope*(x - e) + mainpoint[[2]];
  linegraph = Plot[ lineformula, {x, a, b}];
  (* creating the graphics for the points *)
  rawgraphics = {Red, PointSize[.015], Point[ {mainpoint, newpoint} ]};
  (* Creating a full picture for one frame via Show, and appending it to the list *)
  fullpic = Show[ maingraph, linegraph, Graphics[rawgraphics],
    AspectRatio→Automatic, PlotRange→ {c, d},
    PlotLabel→ N[slope] ];
  AppendTo[ graphicslist, fullpic];
  , {k, 0, n - 1}];
(* return the animation *)
Return[ ListAnimate[graphicslist, AnimationRate → 10]];
];

```



*slopes of secant lines to estimate a derivative*

The last program for this section is another Riemann sum program. This time instead of just returning the value of the Riemann sum we would also like to show a picture of the curve with the shaded rectangles as you have most likely seen in many calculus texts. We can reuse the RiemannSum program and add some additional lines to create the graphics.

#### Example 5: Graphical Riemann sums

Inputs: A formula  $f$ , a variable/interval  $\{x, a, b\}$ , a number of intervals  $n$ .

Options: HeightPoints (set to either "Left" or "Right"), and Digits (number of digits for estimate, default 20).

Output: The numerical estimate of the Riemann sum for  $f$  on  $[a, b]$  using  $n$  subintervals and the chosen point scheme along with a picture of the curve and rectangles.

Discussion: If  $dx$  is the length of one of the subintervals, then the left endpoint of the  $j$ -th subinterval is  $a + (j-1)dx$  and the right endpoint is  $a + jdx$ . So if there is a variable  $m$  set to 1 for left points and 0 for right points, we can use a single formula  $a + (j-m)dx$  for both.

```
NRiemannSum[f_, {x_, a_, b_}, n_Integer, opts___] :=
  Block[{dx, rsum, k, m, dig, pts, curvegraph, polys, j},
    (*dx is the length of a subinterval, j an iterator*)
    (*m is 1 or 0 for Left or Right points from pts*)
```



The Print command is necessary to get the picture as without it the semi-colon would prevent the display of the combined graph. The downside of this is that the actual result is the number and not the graph. Which is fine if you want to use the area estimates but bad if you wanted to export the picture for use in another program (the picture itself is not assigned an Out number and so can't be used with Export). If you really needed the picture however you could easily alter the program to return the actual picture as the answer (possibly with including the area estimate via PlotLabel).

We end this section with a short note about 3D graphics. Raw three dimensional graphics follow the same basic rules as two dimensional graphics in terms of being a list of objects and directives. The wrapper function is Graphics3D instead of Graphics. The graphics objects Point, Line, Polygon, and Text all work more or less the same way (just using 3 coordinates instead of 2). Rectangle is replaced by Cuboid, and Circle by Sphere (there are also commands for cones and cylinders - and even ones for common polyhedra).

## Homework Section 4.7 - Raw Graphics

- 1) Why does evaluating a Graphics command not produce anything visual?
- 2) Explain the difference between Disk and Circle.
- 3) Explain how the position of graphics directives like Hue, PointSize, etc. in a list affect the results.
- 4) Explain why the order makes a difference in a Polygon[] command, with an example.

"Program" 1: Make a stick figure with a smiley face in Mathematica. You do not have to create it all at once - it may be done in parts that are then combined with Show.

Program 2: Create a program called SlidePoint[mess, {x,a,b}, frames] which shows an n-frame animation of a red dot sliding along the graph of  $y=\text{mess}$  from a to b.

Program 3: Create a program MyPlot[mess,{x,a,b},n] which simulates the Plot command by using n line segments to create the graph of  $y=\text{mess}$ . Make sure the axes and such are included.

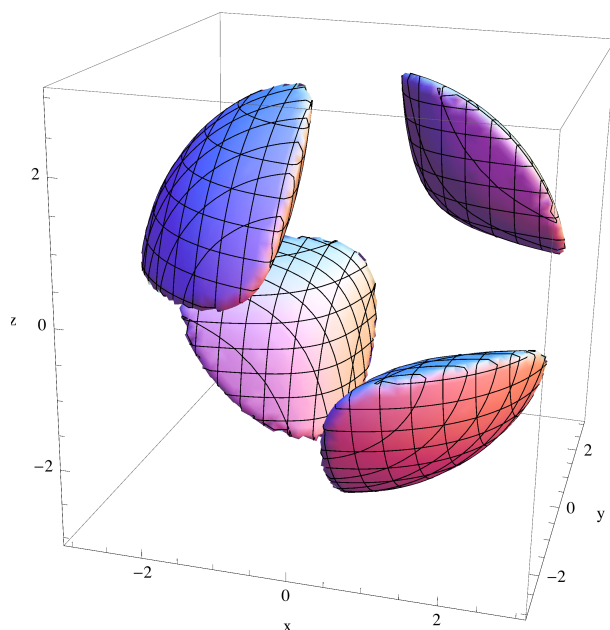
Program 4: Create a program called TangentSlopeMovie[mess,{x,a,b},frames] which works as SlidePoint did, but adds in a line segment through the red point extending 1 unit forward and 1 unit back in terms of x whose slope is essentially that of the tangent line at the red point. (Math hint: To estimate the slope, estimate from  $x=c$  [where c is the location of the red point] to  $x=c+.001$ . The use the "rise over run" notion of slope to get the ends of the line segment). Because you are viewing slopes AspectRatio is an issue.

Program 5: In Section 4.3 you wrote a program ChaosIterate2[r,y,n] which created a list of points of n points starting from a value y which using a value r from 0 to 4. Create a program

`Bifurcation[n,dx]` which creates a graphic as follows: For each value of  $r$  from 0 to 4 going up by  $dx$ , create the list `ChaosIterate[r,1/2,n]`. Turn that list into a set of points of the form  $(r, \text{values})$  so that all the numbers created from  $r$  lie over  $x=r$ . Then graph all the points from different values of  $r$ . Because the points will be very small, you may want to increase their size with a directive like `PointSize[1/n]` or something along those lines. For large values of  $n$  and small values of  $dx$ , an interesting picture should emerge.



# Chapter 5 - Mathematica in Analytic Geometry & Calculus



`D[x^10, {x, 2}]`

$90 x^8$

`D[Sin[Sqrt[x]], x, x, x, x]`

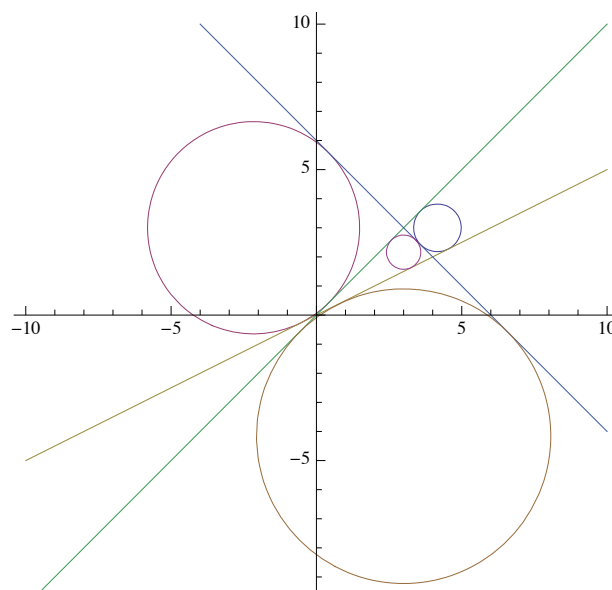
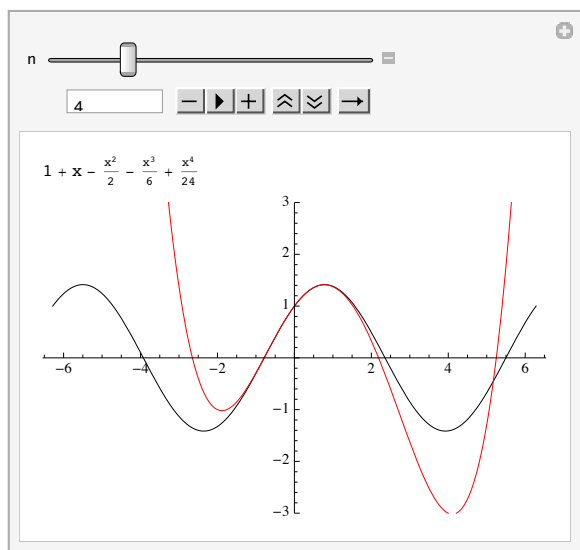
$$-\frac{15 \cos[\sqrt{x}]}{16 x^{7/2}} + \frac{3 \cos[\sqrt{x}]}{8 x^{5/2}} - \frac{15 \sin[\sqrt{x}]}{16 x^3} + \frac{\sin[\sqrt{x}]}{16 x^2}$$

`Integrate[Sqrt[1 + D[x^2, x]^2], {x, 2, 3}]`

$$\frac{1}{4} \left( -4 \sqrt{17} + 6 \sqrt{37} - \text{ArcSinh}[4] + \text{ArcSinh}[6] \right)$$

`N[%, 20]`

5.1003049961756216527



## Section 5.1 - An Introduction

In Chapters 2 and 3 we introduced the basics of Mathematica - essentially how to use it in graphing and pre-calculus mathematics. Just as pre-calculus mathematics forms the foundation for working in higher branches of mathematics such as calculus, these basic Mathematica skills will form the basis for working on problems in higher mathematics. So before we proceed into this chapter, it is probably worthwhile to quickly go back and review some of the sections in the previous chapters (particularly the ones dealing with algebra & trigonometry).

Chapter 5 will primarily deal with the lower tier of collegiate mathematics - that is, the courses that most college students are likely to take in their first two years and usually come with a “2000” level number attached (primarily analytic geometry and calculus). As not every student will have necessarily had all of these courses (or may be taking them at the same time as reading this) we will briefly introduce/refresh the ideas or formulas for different applications. Mathematica’s ability to work with complex expressions will greatly expand the type of problems you can work on using the mathematical knowledge you already have.

## Section 5.2 - Analytic Geometry

Analytic geometry is the study of geometry by applying the ideas of coordinate systems and algebra to geometric problems. The notion of slope and distance play a prominent role in many of these problems so before beginning to look at any applications we need to understand how to compute slopes and distances in Mathematica.

In analytic geometry we think of points as being more or less the same as their Cartesian coordinates. So when looking for an arbitrary point on a curve we tend to represent them as a two element list like  $\text{point} = \{x, y\}$ . We can retrieve the individual coordinates using our “part” notation for lists - to retrieve the x-coordinate we can use `point[[1]]` and for the y-coordinate `point[[2]]`. This makes computing the slope from pre-defined points `pt1` and `pt2` fairly easy (if perhaps a bit cumbersome) as  $(\text{pt2}[[2]] - \text{pt1}[[2]]) / (\text{pt2}[[1]] - \text{pt1}[[1]])$  (essentially the change in y over the change in x). If you are going to measure several different slopes repeatedly in the same Mathematica session you can speed up your work by defining your own slope function, say as `Slope[ pt1_, pt2_ ] := (pt2[[2]] - pt1[[2]]) / (pt2[[1]] - pt1[[1]])`. As with all programs/functions this would need to be reevaluated each time you open your notebook in a new Mathematica session but can remove much of the tedium of finding the slopes between many points.

```
In[6]:= Slope[pt1_, pt2_] := (pt2[[2]] - pt1[[2]]) / (pt2[[1]] - pt1[[1]])
```

The slope from (1,2) to (3,5):

```
In[7]:= Slope[{1, 2}, {3, 5}]
```

```
Out[7]=  $\frac{3}{2}$ 
```

The slope from (2,7) to (x,3):

```
In[8]:= Slope[{2, 7}, {x, 3}]
```

```
Out[8]=  $-\frac{4}{-2 + x}$ 
```

The slope from (3,2) to (3,8)

```
In[9]:= Slope[{3, 2}, {3, 8}]
```

```
Out[9]= ComplexInfinity
```

*calculating slopes in Mathematica using your own function*

Note that in the case where the two points would yield a vertical line you get “ComplexInfinity” as the result which makes good sense as the slope is undefined. In a problem where you wanted to take the second expression and find out what value for x made the line vertical, you could pull the denominator of the slope using the Denominator command, set that equal to 0, and then solve for x.

Distances can be defined in Mathematica in a few ways, the easiest of which is to use the built-in function EuclideanDistance (which has the form EuclideanDistance[ {a,b}, {c,d} ] to find the distance from (a,b) to (c,d) )

```
EuclideanDistance[{2, 1}, {10, 5}]
```

```
4  $\sqrt{5}$ 
```

```
EuclideanDistance[{-3, 1}, {x, y}]
```

```
 $\sqrt{\text{Abs}[-3 - x]^2 + \text{Abs}[1 - y]^2}$ 
```

```
Simplify[% , Element[{x, y}, Reals]]
```

```
 $\sqrt{(3 + x)^2 + (-1 + y)^2}$ 
```

*finding distances in Mathematica using EuclideanDistance*

Note that in the second example above the `EuclideanDistance` results in an expression involving absolute values (which are cleaned up in the following line). The absolute values show up because `EuclideanDistance` works not just with real coordinates but complex ones as well (so the absolute values are necessary). In analytic geometry your coordinates will always be real so you can clean up the absolute values with the combination of `Simplify` and `Element` shown above. It's worth mentioning that `EuclideanDistance` also works in higher dimensions - `EuclideanDistance[ {a,b,c,d}, {x,y,z,t} ]` would find the distance between the two points in 4 dimensions.

In the applications that follow we will assume that you have already defined the function `Slope` in your Mathematica session. This will make the problems easier to work and the logic easier to follow.

Example 1: Finding a circle through 3 points (version 1)

Find all circle(s) through the points (2,3), (4,5), and (6,1).

In the first version of this problem we will approach it using the same logic as we did using Geometer's Sketchpad. We know that a circle of radius  $r$  with center  $(h,k)$  has the equation  $(x - h)^2 + (y - k)^2 = r^2$ . If we can locate the center  $(h,k)$  then its distance to any of the three points will be the radius  $r$ . We know that the perpendicular bisector of a chord runs through the center of the circle; so if line1 is the perpendicular bisector of the chord from (1,3) to (4,5) and line2 is the perpendicular bisector of the chord from (4,5) to (6,4), then the intersection of line1 and line2 will be the center of the circle  $(h,k)$ .

To find the equations of the two bisectors we can use the fact that the equation of a line with slope  $m$  through  $(x_1, y_1)$  is  $y - y_1 = m(x - x_1)$  and the slope of a perpendicular line is found by taking the "negative" reciprocal of the old slope. The point  $(x_1, y_1)$  is the midpoint of each line segment, which we can find by averaging both the  $x$ -coordinates and  $y$ -coordinates of the endpoints. We run this through Mathematica step by step:

```

point1 = {2, 3};
point2 = {4, 5};
point3 = {6, 1};

midpoint1 = (point1 + point2) / 2
{3, 4}

line1 = (y - 4 == -1 / Slope[point1, point2] (x - 3) )
-4 + y == 3 - x

midpoint2 = (point2 + point3) / 2
{5, 3}

line2 = (y - 3 == -1 / Slope[point2, point3] (x - 5) )
-3 + y ==  $\frac{1}{2} (-5 + x)$ 

center = {x, y} /. Solve[{line1, line2}, {x, y}][[1]]
 $\left\{\frac{13}{3}, \frac{8}{3}\right\}$ 

radius = EuclideanDistance[center, point1]
 $\frac{5\sqrt{2}}{3}$ 

radius^2
 $\frac{50}{9}$ 

```

*finding the circle through 3 points*

So the only circle through (2,3), (4,5), and (6,1) is  $(x - \frac{13}{3})^2 + (y - \frac{8}{3})^2 = \frac{50}{9}$  (the square of the radius).

There are several things worth noticing about how the Mathematica computation above was set up. First, everything important is stored in a descriptive variable name like “point1” or “center”. This makes it much easier to follow what is going on in the notebook - especially if you save the notebook and come back to it later. Second, rather than compute each midpoint “by hand” in the notebook, they were computed by simply adding the two points together and then dividing by 2. This works because the points are given as lists and Mathematica adds one list to another component by component. Third, in defining the equation of each line we used an extra set of parentheses were used (line1 = (y-4 == -1/Slope[ point1,point2] (x-3) ), etc.). These extra parentheses aren’t strictly necessary (you can define something line line3= y==x and Mathematica will interpret it correctly) but are useful for helping someone read through the notebook (it’s useful for focus as “line1 is the equation....”). And finally in the definition of the

“center”, we’ve used the replacement notation  $\{x,y\} /. \text{Solve}[ \dots ]$  so we get a real point rather than a list of replacement rules - we’ve used the  $[[1]]$  at the end so that we get just one point instead of the one-answer list  $\{\{13/3, 8/3\}\}$  (remember Solve always gives its results as a list of answers, even if there is just one answer).

### Example 2: Finding a circle through 3 points (version 2)

Find all circle(s) through the points (2,3), (4,5), and (6,1).

In the first version of this problem we used the geometric logic we used for the same problem in Geometer’s Sketchpad. However, can also use the purely algebraic way to approach the same problem we programmed in the previous chapter. Given that a circle with center  $(h,k)$  and radius  $r$  has the equation  $(x - h)^2 + (y - k)^2 = r^2$ , for (2,3) to be on the circle means that when you replace  $x$  with 2 and  $y$  with 3 the equation becomes true. So  $(2 - h)^2 + (3 - k)^2 = r^2$ . We can get a similar equation corresponding to the point (4,5), and a third corresponding to (6,1). That would give us 3 equations in the unknowns  $h$ ,  $k$ , and  $r$ . Solving this system of 3 equations by hand would be ugly (doable, but ugly). Having Mathematica do the grunt work makes this a much more practical approach than it would be on paper:

```
In[61]:= point1 = {2, 3};
point2 = {4, 5};
point3 = {6, 1};

In[64]:= circle = ( (x - h) ^ 2 + (y - k) ^ 2 == r ^ 2 )

Out[64]= (- h + x) ^ 2 + (- k + y) ^ 2 == r ^ 2

In[65]:= eqn1 = circle /. {x -> 2, y -> 3}

Out[65]= (2 - h) ^ 2 + (3 - k) ^ 2 == r ^ 2

In[66]:= eqn2 = circle /. {x -> 4, y -> 5}

Out[66]= (4 - h) ^ 2 + (5 - k) ^ 2 == r ^ 2

In[67]:= eqn3 = circle /. {x -> 6, y -> 1}

Out[67]= (6 - h) ^ 2 + (1 - k) ^ 2 == r ^ 2
```

```

In[68]:= Solve[ {eqn1, eqn2, eqn3}, {h, k, r}]

Out[68]= {{r -> -\frac{5\sqrt{2}}{3}, h -> \frac{13}{3}, k -> \frac{8}{3}}, {r -> \frac{5\sqrt{2}}{3}, h -> \frac{13}{3}, k -> \frac{8}{3}}}

In[69]:= circle /. %

Out[69]= {{\left(-\frac{13}{3} + x\right)^2 + \left(-\frac{8}{3} + y\right)^2 == \frac{50}{9}}, {\left(-\frac{13}{3} + x\right)^2 + \left(-\frac{8}{3} + y\right)^2 == \frac{50}{9}}}

In[70]:= Union[%]

Out[70]= {\left(-\frac{13}{3} + x\right)^2 + \left(-\frac{8}{3} + y\right)^2 == \frac{50}{9}}

```

*a second way to find the circle*

There are two things you should see about this approach. First it is much easier to enter and follow than the geometric approach. This is because while the algebraic setup is easier (no need to find the equations of lines or the midpoints) the underlying algebra is a lot messier. We never see this messy algebra though because it's all done via the Solve command. Second as mentioned in the earlier program the Solve command actually gives two answers rather than one. This is because in the equation  $(x - h)^2 + (y - k)^2 = r^2$  a negative value for the radius is valid for the algebra (since the radius is squared the final value would be non-negative anyway) but doesn't make geometric sense for the circle. Substituting in the values for  $h$ ,  $k$ , and  $r$  into the circle formula gives the same answer twice (as the only difference is the + vs. - on the value for  $r$  - and this is lost in the squaring process), and then Union removes the duplicates.

Which of these two approaches is better? Mathematically they are equally valid. However the second version which uses an algebraic approach is probably better if you are using Mathematica. There are two reasons for this. First, the entry is a bit easier. Second, the second method sidesteps some of the problems that may crop up. For example, if you change the points it might be the case that one of the chords is horizontal, in which case the perpendicular bisector is vertical (which would mean you can't use the point slope form of the line). You can adjust for this of course by using the form for a vertical line (i.e.  $x=a$ ) but the algebraic approach is more "plug-and-chug".

Example 3: Finding circles tangent to 3 lines.

Find all circles which are simultaneously tangent to the lines  $y = x$ ,  $y = \frac{x}{2}$ , and  $y = 6 - x$ .

To find these circles we will again start with the standard form for a circle  $(x - h)^2 + (y - k)^2 = r^2$  where  $(h, k)$  is the center and  $r$  is the radius. The key idea from the geometry will be about tangent lines to a circle: A line  $L$  is tangent to a circle  $C$  if and only if the

distance from L to the center of C is equal to the radius of C. If we take the distance from each of the 3 lines to the unknown center  $(h,k)$  and set it equal to the radius  $r$  we will get three equations in the three unknowns  $h$ ,  $k$ , and  $r$  and hopefully we will get our solutions. The trick is in remembering the formula for the distance from a line to a point. If a line is given by  $Ax + By + C = 0$  and the point is  $(x_1, y_1)$ , the distance from the line to the point is

$$\frac{|Ax_0 + By_0 + C|}{\sqrt{A^2 + B^2}}.$$

Since we will need to compute three distances it is probably worthwhile to

define a function to measure the distance from a point to a line (the line in the “left-hand side only” form  $Ax + By + C$ ):

```
DistanceFromLineToPoint[ line_, point_] := Abs[ line /. {x→point[[1]], y→point[[2]]} ] /
Sqrt[ Coefficient[line,x]^2 + Coefficient[line,y]^2].
```

We'll need to be careful to put each line in  $Ax + By + C$  format (omitting the  $=0$ ) to make this work. So we can rewrite the lines mathematically as  $x-y=0$ ,  $x-2y=0$ , and  $x+y-6=0$  and proceed:

```
circle = (x - h) ^ 2 + (y - k) ^ 2 == r ^ 2
```

$$(-h + x)^2 + (-k + y)^2 = r^2$$

```
DistanceFromLineToPoint[line_, point_] := Abs[line /. {x → point[[1]], y → point[[2]]} ] /
Sqrt[Coefficient[line, x]^2 + Coefficient[line, y]^2]
```

```
eqn1 = ( DistanceFromLineToPoint[ x - y, {h, k} ] == r )
```

$$\frac{\text{Abs}[h - k]}{\sqrt{2}} = r$$

```
eqn2 = ( DistanceFromLineToPoint[ x - 2 y, {h, k} ] == r )
```

$$\frac{\text{Abs}[h - 2 k]}{\sqrt{5}} = r$$

```
eqn3 = ( DistanceFromLineToPoint[ x + y - 6, {h, k} ] == r )
```

$$\frac{\text{Abs}[-6 + h + k]}{\sqrt{2}} = r$$



```
Solve[{eqn1, eqn2, eqn3}, {h, k, r}]
```

$$\left\{ \left\{ r \rightarrow 2\sqrt{2} - \sqrt{5}, h \rightarrow 3, k \rightarrow -1 + \sqrt{10} \right\}, \left\{ r \rightarrow -\sqrt{2} + \sqrt{5}, h \rightarrow 1 + \sqrt{10}, k \rightarrow 3 \right\}, \right. \\ \left. \left\{ r \rightarrow \sqrt{2} + \sqrt{5}, h \rightarrow 1 - \sqrt{10}, k \rightarrow 3 \right\}, \left\{ r \rightarrow 2\sqrt{2} + \sqrt{5}, h \rightarrow 3, k \rightarrow -1 - \sqrt{10} \right\} \right\}$$

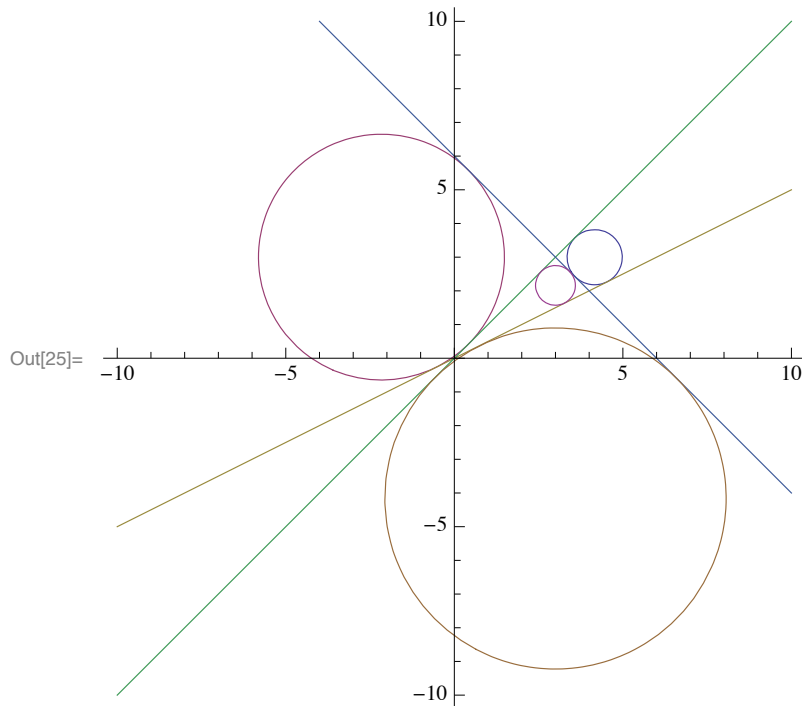
```
solutions = circle /. %
```

$$\left\{ (-3+x)^2 + (1-\sqrt{10}+y)^2 == (2\sqrt{2}-\sqrt{5})^2, (-1-\sqrt{10}+x)^2 + (-3+y)^2 == (-\sqrt{2}+\sqrt{5})^2, \right. \\ \left. (-1+\sqrt{10}+x)^2 + (-3+y)^2 == (\sqrt{2}+\sqrt{5})^2, (-3+x)^2 + (1+\sqrt{10}+y)^2 == (2\sqrt{2}+\sqrt{5})^2 \right\}$$

*finding all circles tangent to 3 lines*

In this case it appears there are 4 separate circles which are tangent to the 3 lines (none of the solutions use the complex number  $i$ ) albeit they have fairly ugly centers and radii. We can use ContourPlot to visualize the circles and lines together:

```
In[25]:= ContourPlot[Evaluate[Union[solutions, {x - y == 0, x - 2 y == 0, x + y - 6 == 0}]],  
  {x, -10, 10}, {y, -10, 10}, Axes -> True, Frame -> None]
```



*the three lines and four circles*

The three lines separate the plane into 7 regions - a middle triangular region and several unbounded regions. There is a circle tangent to all 3 lines in the middle region and for each

unbounded region which is bordered by all 3 lines (some of the unbounded regions are only bordered by 2 lines). In the ContourPlot command we used a Union command to bring the “solutions” and lines together to make one list and then used an Evaluate command to make sure that the Union was done before the ContourPlot (as mentioned previously the Evaluate command is essentially programming parentheses and is often needed when putting commands like Table or Union inside a graphing command like Plot or ContourPlot so they are executed before being graphed).

#### Example 4: Finding the equation of an ellipse

Find a polynomial equation for the set of all points  $(x,y)$  for which the sum of the distances from the point  $(x,y)$  to  $(1,3)$  and  $(4,7)$  is 10 (recall that the definition of an ellipse is the set of all points  $(x,y)$  whose sum to two distinct points is a constant).

As we already know how to find the distance between two points with EuclideanDistance at first glance it seems that this would be relatively simple:

<pre>EuclideanDistance[{1, 2}, {x, y}] + EuclideanDistance[{4, 7}, {x, y}] == 10</pre> $\sqrt{\text{Abs}[1 - x]^2 + \text{Abs}[2 - y]^2} + \sqrt{\text{Abs}[4 - x]^2 + \text{Abs}[7 - y]^2} == 10$ <pre>Simplify[%, Element[{x, y}, Reals]]</pre> $\sqrt{(-4 + x)^2 + (-7 + y)^2} + \sqrt{(-1 + x)^2 + (-2 + y)^2} == 10$
---

*an equation for the ellipse*

This equation would be perfectly good for graphing in ContourPlot - however, the question did not ask for just any equation for the ellipse, but a polynomial one. So we will need to manipulate this equation to eliminate the square roots (this is very common in analytic geometry problems as most of those problems use distance in some way and the various distance formulas all involve square roots). Unfortunately Mathematica does not have a command which will do this for us but it will allow us to proceed through the usual algebra steps fairly quickly and easily by using a special property of Mathematica's lists.

To manipulate an equation using lists first convert the equation into a simple 2 element list *{left-hand side, right-hand side}*. If you add or subtract a single object (be it a number or formula) from a list it will be added/subtracted from all of the parts of a list (which creates the effect of adding or subtracting from both sides of the equation). Likewise if you raise a list to a power all the elements of a list will be raised to that power (which in our case raises both sides of the equation to that power). We can use these types of operations to have Mathematica perform the types of steps on an equation that we could do on paper without having to slog through the potentially messy algebra steps ourselves. To eliminate the square roots in our ellipse equation we first need to put it in list form. The simplest (but not easiest!) way to do this is to just copy-

and-paste from other Mathematica output. The easiest way to do this is to take advantage of the way Mathematica actually represents an equation internally. Mathematica actually thinks of an equation like  $s==t$  as `Equal[s,t]`. So an equation has the head `Equal`, the left-hand side is the “first part”, and the right-hand side is the “second part” - which means we can immediately extract the two sides using part notation:

```
EuclideanDistance[{1, 2}, {x, y}] + EuclideanDistance[{4, 7}, {x, y}] == 10

$$\sqrt{\text{Abs}[1 - x]^2 + \text{Abs}[2 - y]^2} + \sqrt{\text{Abs}[4 - x]^2 + \text{Abs}[7 - y]^2} == 10$$

Simplify[% , Element[{x, y}, Reals]]

$$\sqrt{(-4 + x)^2 + (-7 + y)^2} + \sqrt{(-1 + x)^2 + (-2 + y)^2} == 10$$

{ %[[1]], %[[2]] }
{  $\sqrt{(-4 + x)^2 + (-7 + y)^2} + \sqrt{(-1 + x)^2 + (-2 + y)^2}$  , 10 }
```

*the equation in list form*

Algebraically a possible next step would be to square both sides of the equation. Once expanded, this will create 3 terms on the left-hand side, 2 of which will have square and square root cancellations, and a square root term which will be left over:

```
%^2
{ (  $\sqrt{(-4 + x)^2 + (-7 + y)^2} + \sqrt{(-1 + x)^2 + (-2 + y)^2}$  )^2 , 100 }
Expand[%]
{  $70 - 10x + 2x^2 + 2\sqrt{(-4 + x)^2 + (-7 + y)^2}\sqrt{(-1 + x)^2 + (-2 + y)^2} - 18y + 2y^2$  , 100 }
```

*using properties of lists to square both sides*

The next step algebraically would be to isolate the square roots on one side of the equation (by moving all the other terms to the right hand side) and then squaring again to cancel out the remaining square roots:

```
% - 70 + 10 x - 2 x^2 + 18 y - 2 y^2
{  $2\sqrt{(-4 + x)^2 + (-7 + y)^2}\sqrt{(-1 + x)^2 + (-2 + y)^2}$  ,  $30 + 10x - 2x^2 + 18y - 2y^2$  }
%^2
{  $4((-4 + x)^2 + (-7 + y)^2)((-1 + x)^2 + (-2 + y)^2)$  ,  $(30 + 10x - 2x^2 + 18y - 2y^2)^2$  }
```

*eliminating the remaining square roots*

This new “equation” no longer has square roots in it. However we do have complicated terms on both sides so it is probably a good idea to bring all the terms over to one side (essentially placing the equation in “=0” format) and then simplify to let like terms cancel or combine. Simply taking the left-hand side minus the right-hand side and using Simplify does most of the work:

```
%[[1]] - %[[2]]
```

$$4 \left( (-4 + x)^2 + (-7 + y)^2 \right) \left( (-1 + x)^2 + (-2 + y)^2 \right) - \left( 30 + 10x - 2x^2 + 18y - 2y^2 \right)^2$$

```
Simplify[%]
```

$$4 \left( 91x^2 - 10x(32 + 3y) + 25(4 - 24y + 3y^2) \right)$$

*moving all terms to one side of an equation*

This quantity is equal to 0 (since we moved all the terms to the left-hand side). So the last steps in getting a nice equation for the ellipse would be to divide out the 4, multiply everything out, and put the “=0” back in to make it an equation:

```
Expand[% / 4] == 0
```

$$100 - 320x + 91x^2 - 600y - 30xy + 75y^2 == 0$$

*the final form for the ellipse*

Using the “list” form of an equation to do algebra may seem a little cumbersome at first but the amount of time and effort saved by having Mathematica perform all the intermediate steps and algebraic gruntwork makes it well worth it in problems that involve repeated squaring or other powers.

#### Example 5: Rotation of coordinate systems

Through what angle  $\theta$  would you have to rotate the coordinate system to recognize the conic  $39y^2 + 50\sqrt{3}xy - 11x^2 - 36\sqrt{3}x + 36y - 612 = 0$  as a hyperbola, ellipse, circle, or parabola.

Before we can attack this example it is probably a good idea to quickly recap some of the mathematics behind conic sections. Any equation of the form  $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$  represents a “conic section” - that is, a hyperbola, ellipse, circle, or parabola (this includes so-called “degenerate” cases which are beyond the scope of this quick discussion). If the “xy” term is not present (i.e.  $B=0$ ) it is relatively easy to identify the conic:

- 1) If A and C have opposite signs, it is a hyperbola.

- 2) If one of A or C is 0, it is a parabola.
- 3) If A and C are equal, it is a circle.
- 4) If A and C have the same sign but are unequal, it is an ellipse.

So the trick in identifying a conic is to find a way to eliminate the “xy” term. This can be done by rotating the coordinate system counter-clockwise through an angle of  $\theta$  (this has the appearance of rotating the graph clockwise through the same angle). Such a rotation changes the original coordinates  $x$  and  $y$  into new coordinates  $x_1$  and  $y_1$  using the following substitutions:

$$\begin{aligned}x &= x_1 \cos(\theta) - y_1 \sin(\theta) \\ y &= x_1 \sin(\theta) + y_1 \cos(\theta)\end{aligned}$$

The reverse substitution (if you want to convert from the rotated coordinates  $x_1$  and  $y_1$  back to  $x$  and  $y$ ) are

$$\begin{aligned}x_1 &= x \cos(\theta) + y \sin(\theta) \\ y_1 &= -x \sin(\theta) + y \cos(\theta)\end{aligned}$$

The idea here is that you get to choose the value for  $\theta$  in order to change the equation  $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$  into an equation involving  $x_1$  and  $y_1$  where there is no  $x_1y_1$  term and we can easily tell what type of conic we have. On paper, taking our conic  $39y^2 + 50\sqrt{3}xy - 11x^2 - 36\sqrt{3}x + 36y - 612 = 0$  and using the substitutions  $x = x_1 \cos(\theta) - y_1 \sin(\theta)$  and  $y = x_1 \sin(\theta) + y_1 \cos(\theta)$  would be very messy but in Mathematica it is fairly simple:

```
conic = 39 y^2 + 50 Sqrt[3] x y - 11 x^2 - 36 Sqrt[3] x + 36 y - 612
- 612 - 36 Sqrt[3] x - 11 x^2 + 36 y + 50 Sqrt[3] x y + 39 y^2
conic /. {x -> x1 Cos[t] - y1 Sin[t], y -> x1 Sin[t] + y1 Cos[t]}
- 612 + 36 (y1 Cos[t] + x1 Sin[t]) +
  39 (y1 Cos[t] + x1 Sin[t])^2 - 36 Sqrt[3] (x1 Cos[t] - y1 Sin[t]) +
  50 Sqrt[3] (y1 Cos[t] + x1 Sin[t]) (x1 Cos[t] - y1 Sin[t]) - 11 (x1 Cos[t] - y1 Sin[t])^2
```

*substituting in the equations of rotation using  $t$  for  $\theta$*

On paper we would then multiply this expression out, combine like terms, identify the  $x_1y_1$  term, and find a value for  $t$  which will make  $x_1y_1$ 's coefficient 0. This very involved pencil-and-paper algebra and trigonometry takes only two simple commands in Mathematica:

**Coefficient[%, x1 y1]**

$$50 \sqrt{3} \cos[t]^2 + 100 \cos[t] \sin[t] - 50 \sqrt{3} \sin[t]^2$$

**Reduce[% == 0, t]**

$C[1] \in \text{Integers} \&\&$

$$\left( t == \frac{\pi}{3} + 2\pi C[1] \mid \mid t == -\frac{2\pi}{3} + 2\pi C[1] \mid \mid t == -\frac{\pi}{6} + 2\pi C[1] \mid \mid t == \frac{5\pi}{6} + 2\pi C[1] \right)$$

*finding the value for t which will eliminate the x1y1 term*

Using Reduce here instead of Solve is a better choice since the coefficient involves sine and cosine which means there will be many possible solutions (we could also have used Reduce to restrict the angle to the range  $[0, \frac{\pi}{2}]$ ). Looking at the Reduce solution the first value of  $\pi/3$  radians (or 60 degrees) seems to be an easy value which will work. Putting this value in for t in the “rotated conic” lets us identify the conic type:

**%% / . t → Pi / 3**

$$-612 + 36 \left( \frac{\sqrt{3} x_1}{2} + \frac{y_1}{2} \right) + 39 \left( \frac{\sqrt{3} x_1}{2} + \frac{y_1}{2} \right)^2 - 36 \sqrt{3} \left( \frac{x_1}{2} - \frac{\sqrt{3} y_1}{2} \right) + 50 \sqrt{3} \left( \frac{\sqrt{3} x_1}{2} + \frac{y_1}{2} \right) \left( \frac{x_1}{2} - \frac{\sqrt{3} y_1}{2} \right) - 11 \left( \frac{x_1}{2} - \frac{\sqrt{3} y_1}{2} \right)^2$$

**Expand[%]**

$$-612 + 64 x_1^2 + 72 y_1 - 36 y_1^2$$

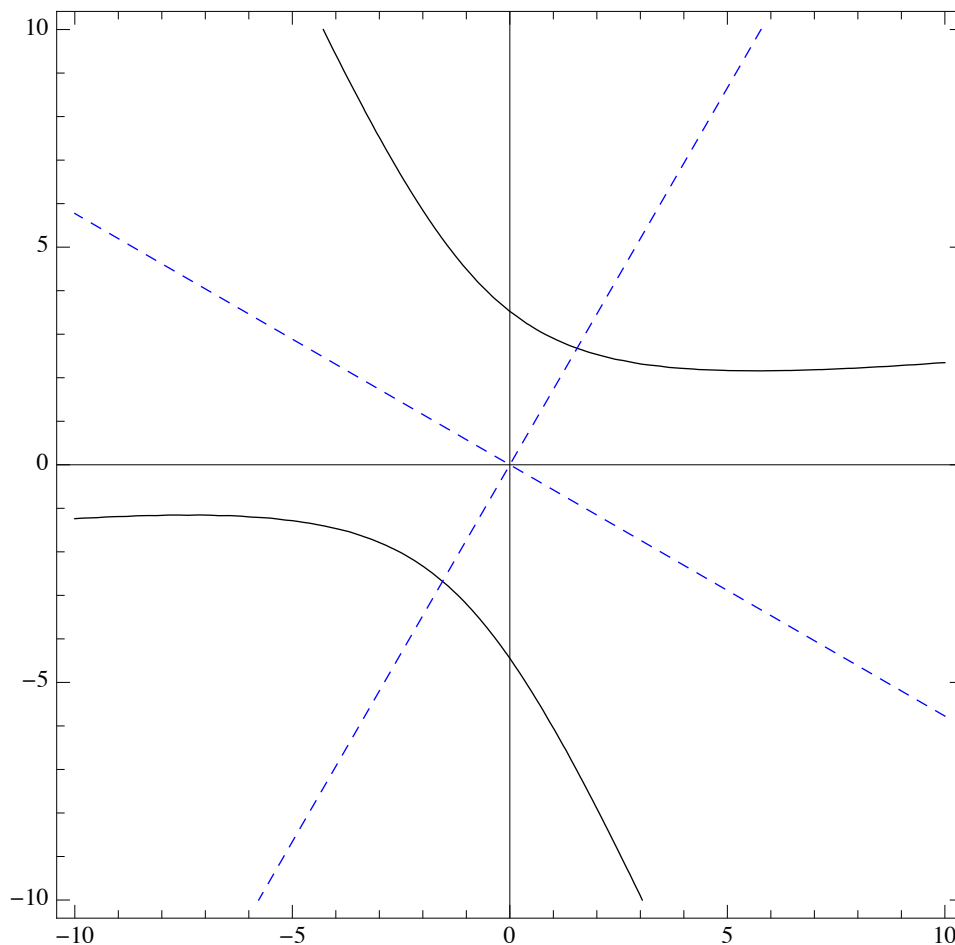
*the conic with axes rotated by 60 degrees*

A quick look at this form tells you that the coefficients of  $x_1^2$  and  $y_1^2$  have opposite signs which means the conic is a hyperbola. We can even do better and show the hyperbola with the rotated axes. In the rotated coordinate system the axes are given by  $x_1=0$  and  $y_1=0$ . Using the reverse substitutions given in the background discussion this becomes  $x_1 = 0 = x \cos(\theta) + y \sin(\theta)$  and  $y_1 = 0 = -x \sin(\theta) + y \cos(\theta)$  where  $\theta=\pi/3$ . We can use ContourPlot to graph the original conic and the rotated axes together:

**equations = {conic == 0, 0 == x Cos[t] + y Sin[t], 0 == -x Sin[t] + y Cos[t]} / . t → Pi / 3**

$$\left\{ -612 - 36 \sqrt{3} x - 11 x^2 + 36 y + 50 \sqrt{3} x y + 39 y^2 == 0, 0 == \frac{x}{2} + \frac{\sqrt{3} y}{2}, 0 == -\frac{\sqrt{3} x}{2} + \frac{y}{2} \right\}$$

**ContourPlot[Evaluate[equations], {x, -10, 10}, {y, -10, 10},  
ContourStyle → {{Black}, {Blue, Dashed}, {Blue, Dashed}},  
Axes → True, AxesOrigin → {0, 0}]**



*viewing the conic and the rotated coordinate system*

Note that if you tilt your head to match the rotated coordinate system the hyperbola opens directly to the left/right and is not tilted at all - this is the geometric interpretation of eliminating the  $xy$  term.

## Section 5.2 Homework - Analytic Geometry

- 1) Find all circles which go through  $(2,1)$ ,  $(5,6)$ , and  $(7,0)$  using the “geometric” approach.
- 2) Find all circles which go through  $(2,1)$ ,  $(5,6)$ , and  $(7,0)$  using the “algebraic” approach.
- 3) Find all circles which are simultaneously tangent to the lines  $y = 2x + 1$ ,  $y = x - 5$ , and  $x + y = 4$ . Graph these circles together with the lines.
- 4) Find all circles which are simultaneously tangent to  $y = 2x + 1$ ,  $y = 2x + 9$ , and  $y = x - 1$ . Graph these circles together with the lines.
- 5) Find a polynomial equation for the set of points  $(x, y)$  whose distance to  $(3,1)$  is the same as its distance to the line  $y = x + 1$ .

- 6) Find a polynomial equation for the set of points  $(x, y)$  for which the difference of the distances from  $(x, y)$  to  $(1, 1)$  and  $(3, 3)$  is 10. (Note: you will need to enclose the difference in absolute value to start, as you won't know which distance is the larger).
- 7) Find a polynomial equation for the set of all points  $(x, y)$  whose distance to  $(2, 3)$  is 3 times its distance to  $(4, 7)$ .
- 8) Find a polynomial equation for the set of points  $(x, y)$  such that the sum of the slopes from  $(x, y)$  to the points  $(1, 1)$  and  $(3, 4)$  is 2. Graph the equation.
- 9) Use the rotation of coordinate systems to classify the conic section
 
$$21x^2 + 58xy + 21y^2 - 42\sqrt{2}x - 58\sqrt{2}y - 158 = 0$$
- 10) Use the rotation of coordinate systems to classify the conic section
 
$$19x^2 + 6\sqrt{15}xy + 61y^2 + (8 - 32\sqrt{15})y - (32 + 8\sqrt{15})x - 176 = 0$$
- 11) Use the rotation of coordinate systems to classify your equation from problem 8 and see that it matches the graph.
- 12) Look up the command ManhattanDistance and explain what it does. Use ContourPlot to graph a “Manhattan circle” centered at  $(1, 2)$  with radius 2 and a “Manhattan ellipse” with foci  $(1, 1)$  and  $(3, 2)$  and a fixed distance of 6.
- 13) In this section we saw that changing an equation into a list was useful for performing algebraic operations. We did this manually with part notation, but there other ways to achieve the same goal. What happens when you evaluate the command  $x^2 + y^2 = x^3$  /. Equal → List ? What is the result of  $\{x+y, 1\}$  /. List → Equal ? Why do you think these work?

Program 1: Write a program to find the circle through 3 points using the “bisect two chords/geometric” method (don't assume the Slope command has been defined). Have the program return Indeterminate if no such circles exist. Check your answer to problem 1 using your program.

Program 2: Write a program to find the circle through 3 points using the “algebraic/plug in the 3 points” method. Have the program return Indeterminate if no such circles exist. Check your answer to problem 2 using your program.

Program 3: Write a program to find the angle through which a conic equation (not assumed to be in “=0” form) must be rotated to depress the  $xy$  term. As the inputs you should have both the conic and the underlying variables  $x$  and  $y$ .

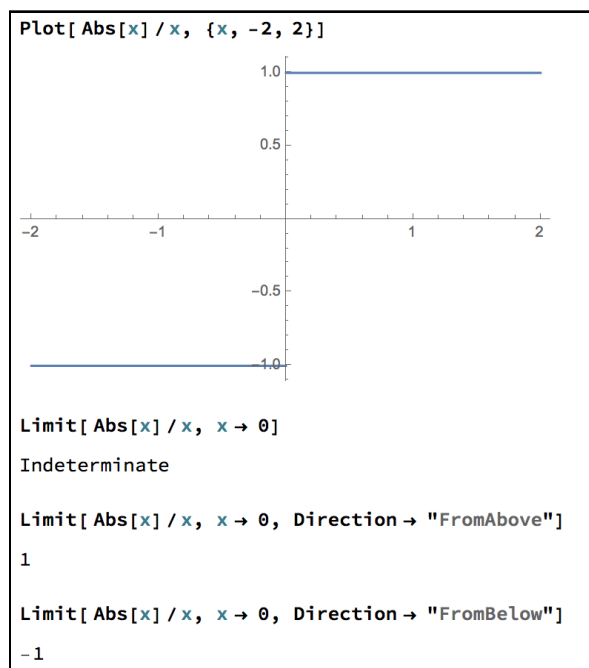
Program 4: Write a program to find the circles tangent to 3 lines. Your inputs should be the 3 lines (not assumed to be in “=0” format) and a column of the three circle equations and the combined graphs. Programming hint: The trickiest part of this will be to get the graphing ranges. The best way to get them without using more advanced commands to assume the  $x$ -range extends from 1 unit to the left of the leftmost point on all 4 circles up to 1 unit to the right of the rightmost point of all 4 circles (and something similar to get the  $y$ -range).



## Section 5.3 - The Computations of Calculus

Most of elementary calculus relies on four main concepts and calculations - limits, derivatives, integrals, and series. All of these computations can be done easily in Mathematica through the use of a few simple commands and options. In this section we will focus on the versions of these commands you would use in first-year calculus - basically the calculus involving functions of a single variable (usually  $x$ ).

As you might expect the basic command to do limits in Mathematica is `Limit`, and it was updated in Mathematica 11.2 to be easier to use for basic calculus. The basic structure of the command is `Limit[ expression, variable  $\rightarrow$  value ]`. This is the basic two-sided limit, and will return `Indeterminate` if a limit does not exist. Prior to 11.2 the same command represented a one-sided limit (by default the limit from above). One-sided limits are now handled by the option `Direction`, with the two principal values being `"FromAbove"` (quotes included, for limits from the right) and `"FromBelow"` (quotes included, for limits from the left). Here is the graph of the function  $|x|/x$  together with its various limits at 0:



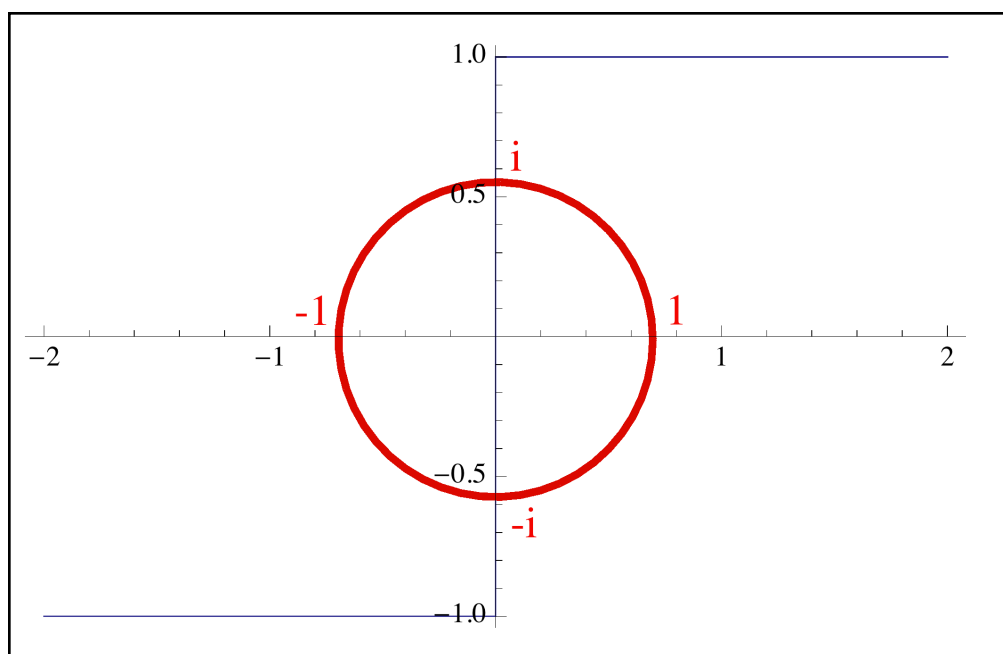
*the graph and various limits of  $|x|/x$*

Looking at the graph of  $|x|/x$  it should be clear that the full two-sided limit does not exist at 0. As you get close to 0 on the left the values are always -1; from the right, +1.

The original pre-Mathematica 11.2 version of `direction` still works, and is more useful when you move beyond basic calculus and into analysis. The more general format for the

Direction is  $\text{Direction} \rightarrow \text{number}$ , where the *number* (typically, but not necessarily, chosen to have absolute value 1) represents the direction of the limit as follows:

- 1) Imagine a small circle centered on where you are taking the limit.
- 2) On this unit circle mark the point rightmost point as 1, the leftmost point as -1, the top as  $i$ , and the bottom point as  $-i$ .
- 3) When using  $\text{Direction} \rightarrow \text{number}$ , where *number* has magnitude 1, you are taking the limit in the direction “circle center towards *number*” (that is, in the direction of the arrow from the circle center to *number*).



*the imaginary circle centered at  $x=0$  with the points labelled*

If you wanted to take the limit at 0 from the left/above you would use  $\text{Direction} \rightarrow -1$  as the arrow from the center to -1 points to the left. If you wanted the limit at 0 from the left/below, you need an arrow which points to the right; this would come from an arrow starting at the center and point at - so you would use  $\text{Direction} \rightarrow 1$ . Now it is easy to just think of this as “limit from the right,  $\text{Direction} \rightarrow -1$ , limit from the left,  $\text{Direction} \rightarrow 1$ ”. However the Limit command goes beyond basic calculus and lets you take limits using complex numbers, and the circle helps you with that too. If you wanted to take the limit of  $|x|/x$  as  $x \rightarrow 0$  using small positive imaginary numbers (like  $.1i$ ,  $.01i$ ,  $.001i$ , etc.), you need an arrow pointing down; that would be from the center to  $-i$  so you would use  $\text{Direction} \rightarrow -i$ . You can see the different results below:

```

Limit[Abs[x] / x, x → 0, Direction → 1]
- 1

Limit[Abs[x] / x, x → 0, Direction → -1]
1

Limit[Abs[x] / x, x → 0, Direction → -I]
- i

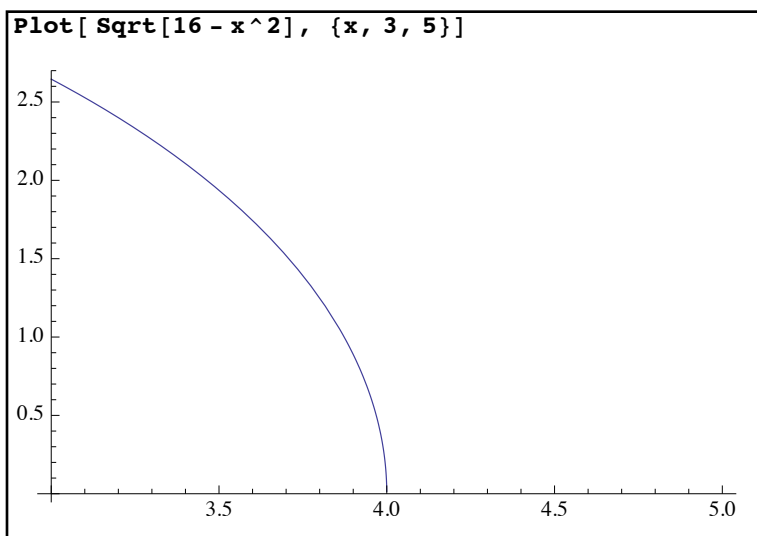
Limit[Abs[x] / x, x → 0, Direction → I]
i

Limit[Abs[x] / x, x → 0, Direction → (1 + I) / Sqrt[2]]
-  $\frac{1 - i}{\sqrt{2}}$ 

```

*limits of  $|x|/x$  as  $x \rightarrow 0$  from different directions*

If working with complex numbers in the limits seems overly difficult it is something that cannot really be avoided. For example suppose you wanted to take the two-sided limit of  $\sqrt{16 - x^2}$  as  $x$  goes to 4. A quick look at the graph indicates the two-sided limit does not exist in the Calculus 1 sense for the simple reason that the graph does not exist on the right of 4:



*no graph on the right of 4, the limit on that side doesn't exist in Calculus 1*

Try the one-sided limits in Mathematica however and something odd happens:

The limit from the left

```
In[19]:= Limit[Sqrt[16 - x^2], x → 4, Direction → 1]
```

```
Out[19]= 0
```

The limit from the right

```
In[20]:= Limit[Sqrt[16 - x^2], x → 4, Direction → -1]
```

```
Out[20]= 0
```

*according to Mathematica the limit on the right is 0 even though there's no graph!*

So according to Mathematica the two-sided limit exists and is 0. Is this wrong? From a Calculus 1 perspective, yes. But from a complex number perspective this is actually correct. When  $x > 4$  the quantity under the square root is negative and so the square root is an imaginary number. But as  $x$  gets closer to 4 on the right those imaginary numbers become smaller and smaller in size - so in fact they do go to 0. So the limit from a “complex calculus” sense is 0 even though in Calculus 1 would we say the limit does not exist. This is an issue which is not really one you can easily resolve. As it turns out calculus using complex numbers (usually called complex analysis) is very important in many applications so Mathematica really needs to be able to work in this fashion. This is just an issue you need to be aware of when taking limits.

Mathematica's Limit command can also work with limits at  $\pm\infty$  by using the numbers Infinity and -Infinity. It also knows L'Hopital's Rule, so you can work with all sorts of indeterminate forms:

```
Limit[ x^3 / Exp[x], x → Infinity]
```

```
0
```

```
Limit[ Sqrt[ n^2 - 5 n + 7] - n, n → Infinity]
```

```
5  
-  
2
```

```
Limit[ (1 + r / n)^n, n → Infinity]
```

```
er
```

*limits of indeterminate forms*

Mathematica will also show you when limits do not exist and even in some cases give you additional information:

```

Limit [1 / x^2, x → 0]

∞

Limit[ Sin[x], x → Infinity]

Interval[{-1, 1}]

```

*limits that do not exist*

Both of these limits fail to exist. In the first case Mathematica returns the usual value of “infinity”, essentially saying that the values of the function are unbounded (remember, this is from the right by default). In the second though the result “Interval” is returned. This is Mathematica’s way of telling you that as  $x$  goes to infinity, the values of  $\sin(x)$  oscillate continually through the interval  $[-1, 1]$  and never settle down to a single value.

After limits the next main computation of calculus is the derivative. The basic command for taking the derivative is `D`, in the form `D[formula, variable]`. So taking the derivative of  $x^2$  would be `D[x^2, x]`. If you’ve defined your function as `f[x]`, you can also use the standard “prime” notation for derivatives (`f'[x]`, `f''[x]`, and so on):

```

D[x^3, x]

3 x^2

D[x^6 / (Exp[x] + Sin[x]), x]


$$-\frac{x^6 (e^x + \cos[x])}{(e^x + \sin[x])^2} + \frac{6 x^5}{e^x + \sin[x]}$$


D[Tan[Sin[Sqrt[x]]], x]


$$\frac{\cos[\sqrt{x}] \sec[\sin[\sqrt{x}]]^2}{2 \sqrt{x}}$$


D[x^4 / Exp[x], x]


$$4 e^{-x} x^3 - e^{-x} x^4$$


f[x_] := x^5 - 4 x^3 + 2 x - 1

f'[x]

2 - 12 x^2 + 5 x^4

```

```

f'''[x]

- 24 + 60 x2

D[ x^2 == Exp[Sin[1 / x]], x]


$$2x == - \frac{e^{\sin\left[\frac{1}{x}\right]} \cos\left[\frac{1}{x}\right]}{x^2}$$


D[ g[x]^4 - Tan[g[x]], x]

4 g[x]3 g'[x] - Sec[g[x]]2 g'[x]

D[x!, x]

Gamma[1 + x] PolyGamma[0, 1 + x]

```

*many different derivatives*

You can see how Mathematica can take the derivatives of rather ugly functions pretty easily. It also can take the derivatives of equations that use the == sign (the result being another equation) and it can handle “symbolic” derivatives that use unknown functions like the undefined  $g(x)$  used above. It can even take the derivative of the function  $x!$  although it returns some “special functions” as the answer (looking up the definition of special functions as they come up, either in the documentation or online in Mathworld is a great way to start learning about them).

All of the examples of derivatives given above were “explicit” derivatives - ones where you have been given a function  $y=f(x)$  and been asked to find  $y'$ . Mathematica can also compute implicit derivatives - that is, the derivative of a function which has been defined by an equation which has not been solved for  $y$  (for example finding the derivative  $y'$  for a curve defined by  $x^3 - xy^2 + x^2y + y^5 = 2$  at the point (1,1). Solving these sorts of equations for  $y$  is often ugly at best and impossible at worst. To find the derivative for a function defined by an equation try the following steps:

- 1) Enter your equation in Mathematica using the == notation, but use  $y[x]$  in place of  $y$  (this tells Mathematica that  $y$  depends on  $x$ ).
- 2) Take the derivative of the equation. This will yield a new equation involving  $y'[x]$ .
- 3) Use the Solve command to solve for  $y'[x]$  (it is quicker to use a format like  $y'[x] /. \text{Solve}[\%, y'[x]][[1]]$  when you solve the equation as it will give you the formula right away).

So to try to find the derivative for  $x^3 - xy^2 + x^2y + y^5 = 2$  at the point (1,1) we would try the following:

```

equation = ( x^3 - x y[x]^2 + x^2 y[x] + y[x]^5 == 2 )

x^3 + x^2 y[x] - x y[x]^2 + y[x]^5 == 2

equation /. {x -> 1, y[x] -> 1}

True

D[equation, x]

3 x^2 + 2 x y[x] - y[x]^2 + x^2 y'[x] - 2 x y[x] y'[x] + 5 y[x]^4 y'[x] == 0

y'[x] /. Solve[%, y'[x]][[1]]

- 3 x^2 - 2 x y[x] + y[x]^2
-----
x^2 - 2 x y[x] + 5 y[x]^4

% /. {x -> 1, y[x] -> 1}

- 1

```

*finding the value of an implicit derivative*

There are a few things you worth mentioning about how this was set up. First there is a check on whether the point (1,1) is on the curve (the equation becomes “True” there). Second as part of the process of “plugging” in  $x=1$  and  $y=1$  it was necessary to use  $y[x]$  instead of just  $y$  (if you don’t the results will contain something silly like  $1[1]$ ). And last, as part of the process we found both the general formula for  $y'$  at every point on the curve, and then specifically the value at (1,1). If you had to find the values of the derivative at several points along the curve it would be a good idea to store the general formula in a variable like “firstder”, and then use that to substitute in several different values of  $x$  and  $y$ .

Mathematica can also easily compute higher derivatives. For the second derivative you could use  $D[\text{formula}, \text{variable}, \text{variable}]$ , for the third derivative  $D[\text{formula}, \text{variable}, \text{variable}, \text{variable}]$ , and so on. This can get pretty tedious past the second or third derivative so Mathematica provides a shortcut notation for higher derivatives - if  $n$  is a natural number, you can find the  $n^{\text{th}}$  derivative of  $\text{formula}$  by  $D[\text{formula}, \{\text{variable}, n\}]$ . When computing higher derivatives Mathematica will not do anything but the most basic simplifications - for example, it will write division by exponentials as negative exponents, but won’t bring together terms over a common denominator. So when finding the derivatives you may want to use commands like Simplify, FullSimplify, Together, or Cancel. Here are some examples of computing higher derivatives using both notations:

```

D[ x^10, x, x]

90 x^8

D[ x^10, {x, 2}]

90 x^8

D[ Sin[Sqrt[x]], x, x, x, x]

-  $\frac{15 \cos[\sqrt{x}]}{16 x^{7/2}}$  +  $\frac{3 \cos[\sqrt{x}]}{8 x^{5/2}}$  -  $\frac{15 \sin[\sqrt{x}]}{16 x^3}$  +  $\frac{\sin[\sqrt{x}]}{16 x^2}$ 

D[ Sin[ Sqrt[x]], {x, 4}]

-  $\frac{15 \cos[\sqrt{x}]}{16 x^{7/2}}$  +  $\frac{3 \cos[\sqrt{x}]}{8 x^{5/2}}$  -  $\frac{15 \sin[\sqrt{x}]}{16 x^3}$  +  $\frac{\sin[\sqrt{x}]}{16 x^2}$ 

D[ Exp[x], {x, 314 159}]

e^x

```

*some higher derivatives*

Integrals are the third main computation of calculus and from a computational standpoint they are the probably the hardest (a great deal of time is usually spent in second semester calculus learning how to find antiderivatives of specific forms). The basic command for finding an antiderivative (essentially an indefinite integral) is `Integrate[formula, variable]`. The only difference between the result of `Integrate` and what we think of as an indefinite integral is the `Integrate` command does not use the infamous “+C”. For the definite integral of *formula* as *variable* goes from *a* to *b* the command is `Integrate[formula, {variable, a, b}]`. These can also be done using standard mathematical notation via Mathematica’s palettes but we will be focusing on keyboard-entry techniques. Here are some basic examples of integrals done in Mathematica (both proper and improper ones):

```
Integrate[ x^3, x]
```

$$\frac{x^4}{4}$$

```
Integrate[ 1 / t, t]
```

```
Log[t]
```



```

Integrate[ x^10 Sin[x], x]
- (- 3 628 800 + 1 814 400 x^2 - 151 200 x^4 + 5040 x^6 - 90 x^8 + x^10) Cos[x] +
  10 x (362 880 - 60 480 x^2 + 3024 x^4 - 72 x^6 + x^8) Sin[x]

Integrate[x^2, {x, 1, 2}]
7
—
3

Integrate[ 1 / x^2, {x, 0, 1}]

Integrate::idiv : Integral of  $\frac{1}{x^2}$  does not converge on {0, 1}. >>

 $\int_0^1 \frac{1}{x^2} dx$ 

Integrate[1 / x^2, {x, 1, ∞}]
1

```

*several basic integrals in Mathematica*

Note that the integral of  $1/x^2$  on  $[0,1]$  is simply returned back together with a warning - this is because the integral is divergent (in older versions of Mathematica the warning would appear in a separate window).

Integration in Mathematica is often tricky in the sense that some integrals may be returned to you in unfamiliar forms or simply the same form spat back at you. Unfamiliar forms tend to show up for two reasons. First, Mathematica may use different techniques to approach an integral than you have been taught. For example, in second semester calculus you would approach  $\int \sin^7(x) dx$  by pulling off one sine, rewriting the remaining sines in terms of cosines, and then using the direct substitution  $u=\cos(x)$ . This would give you an antiderivative involving powers of cosine (up to  $\cos^7(x)$ ). Mathematica returns something different:

```

Integrate[ Cos[x]^7, x]

35 Sin[x] + 7 Sin[3 x] + 7 Sin[5 x] + Sin[7 x]
— + — + — + —
64 64 320 448

```

*a different looking antiderivative*

Mathematica is using a different technique here - it is essentially using TrigReduce on the integrand and then integrating the resultant formula (whose antiderivative is much simpler). Neither answer is wrong (in fact, you could use TrigExpand and other commands to essentially turn one into the other), they just differ in form and by a constant. If you are unsure if your

antiderivative is the same as Mathematica's the easiest way to check is to graph them both. If both antiderivatives are correct then the two graphs will be parallel (that is, one of them will be the other moved straight up or down). The second reason an antiderivative may look unfamiliar is that it may involve functions you've never heard of. One of the most common ways to get special functions is by integration - most functions don't have "closed form" antiderivatives and so special functions are created so the integrals may be processed (many of the special functions are in fact defined by integrals - which may seem like circular reasoning except you can numerically estimate the special functions using techniques like Simpson's rule). For example consider the following integrals:

```

Integrate[ Sin[x^2] , x]


$$\sqrt{\frac{\pi}{2}} \text{FresnelS}\left[\sqrt{\frac{2}{\pi}} x\right]$$


Integrate[ Exp[-x^2] , x]


$$\frac{1}{2} \sqrt{\pi} \text{Erf}[x]$$


Integrate[Sqrt[1 - 2 Sin[x] ^2] , {x, 0, 1 / 2}]


$$\text{EllipticE}\left[\frac{1}{2}, 2\right]$$


N[%]

0.456992 + 0. i

Integrate[Sin[1 / x] , x]


$$-\text{CosIntegral}\left[\frac{1}{x}\right] + x \text{Sin}\left[\frac{1}{x}\right]$$


Integrate[ Sin[Cos[x]] , x]


$$\int \text{Sin}[\text{Cos}[x]] \, dx$$


```

*integrals involving special functions*

In all but the last example the integrals yield special functions (you can find the definitions of the different functions in the Mathematica documentation). In the last case Mathematica simply can't do the integral (which happens - integration is a tricky business) and so the integral is simply returned unevaluated. It's also worth noting that in the numerical estimate of `EllipticE`

there is an imaginary part to the answer. This happens frequently when evaluating certain special functions - it essentially occurs because the answer has several terms with imaginary parts that are supposed to cancel out perfectly. But when you numerically estimate small errors can be introduced which throw off the perfect balance between the terms and leaves you with a small imaginary number (written as  $0.i$ ). When you know your integral is real and this happens, you can convert the answer to a real number by using the command `Chop` (as in `Chop[%]`). `Chop` rounds down very small numbers to 0, which in problems like this converts your “almost real” answer to real. You can also use the `NIntegrate` command to estimate integrals directly, and `NIntegrate` uses the same `WorkingPrecision` option that similar numerically-oriented commands do.

Another issue that may crop up in integration are formulas that involve a constant (for example  $\sin(bx)$  where  $b$  is a constant). In many cases how an integral is handled may depend on the value of a constant or what range it is in. For example, consider the formulas  $\frac{1}{1+ax^2}$  and  $\frac{1}{1-ax^2}$ , where  $a$  is a constant. These are really the same formula (since  $a$  is a constant that could be positive or negative), but their integrals are handled differently:

$\text{Integrate}\left[\frac{1}{1+ax^2}, x\right]$ $\frac{\text{ArcTan}\left[\sqrt{a} x\right]}{\sqrt{a}}$ $\text{Integrate}\left[\frac{1}{1-ax^2}, x\right]$ $\frac{\text{ArcTanh}\left[\sqrt{a} x\right]}{\sqrt{a}}$
--

*same basic form, different results*

Even worse for many definite integrals whether or not the integral converges may depend on the values of a constant. Mathematica often handles this via an option called `Assumptions`:

$\text{Integrate}\left[\frac{1}{a^x}, \{x, 1, \text{Infinity}\}\right]$ $\text{If}\left[\text{Re}[\text{Log}[a]] > 0, \frac{1}{a \text{Log}[a]}, \text{Integrate}[a^{-x}, \{x, 1, \infty\}, \text{Assumptions} \rightarrow \text{Re}[\text{Log}[a]] \leq 0\right]$
--

*an integral whose existence depends on what “a” is*

In this example we are told what the value is in the case that the real part of  $\ln(a)$  is positive (which for real numbers means  $a > 1$ ), and in the case where the real part of  $\ln(a)$  is negative the integral is returned unevaluated (since it does not converge). You can use Assumptions in your own Integrate commands to tell Mathematica something about the range of a variable. For example if you wanted to find  $\int_1^{\infty} \frac{1}{x^n} dx$  where  $n > 1$  (so the integral exists) you could use `Integrate[1/x^n, {x,1,Infinity}, Assumptions→ n > 1 ]`.

The fourth and final computation of calculus is that of a series. For series of numbers we can simply use the Sum and NSum commands, and if possible Mathematica will try to find whether or not the Sum converges and if so what to (a divergent sum will be returned unevaluated with a warning message). Sums that are returned without a divergence message still converge, but Mathematica cannot find the exact value for the true sum. In these cases you will need to use N or NSum to get a numerical estimate.

```
Sum[1/n^2, {n, 1, Infinity}]

$$\frac{\pi^2}{6}$$

Sum[n/(1+Exp[n]), {n, 1, Infinity}]

$$\sum_{n=1}^{\infty} \frac{n}{1+e^n}$$

N[%, 10]
0.7808004196
Sum[1/Sqrt[n], {n, 1, Infinity}]
Sum::div : Sum does not converge. >>

$$\sum_{n=1}^{\infty} \frac{1}{\sqrt{n}}$$

Sum[n^10/7^n, {n, 3, Infinity}]

$$\frac{254\,463\,896}{107\,163}$$

```

*various infinite sums in Mathematica*

The second sum in this example did not create a divergence message and so we knew it converged even though Mathematica did not have a good way to write the sum. The third sum created a “diverges” message in a separate window, so there was no need to estimate it.

If all you care about is the convergence or divergence of a series you can call on the command SumConvergence in the form `SumConvergence[general term, variable]`. This will return True if Mathematica can determine if the sum converges and False if can determine the sum diverges.

```

SumConvergence[ 1 / ( 2 + Sqrt[ 1 + n^2 ] ), n]
False

SumConvergence[ (-1)^n / (n - Sqrt[n]), n]
True

SumConvergence[ Sin[n] / Sqrt[n], n]

SumConvergence[ Sin[n] / Sqrt[n], n]

```

*a divergent sum, a convergent sum, and one that Mathematica can't figure out*

More important than basic infinite sums are power series (essentially infinitely long polynomials). Power series that you already have coefficients for can be entered by the Sum command again and Mathematica will do its best to convert the power series to explicit functions wherever possible:

```

Sum[ (-1)^n x^(3 n) / n!, {n, 1, Infinity}]

-e^-x^3 (-1 + e^x^3)

Sum[ x^n / Sqrt[n], {x, 2, Infinity}]

-1 + Zeta[-n]
-----
Sqrt[n]

Sum[ Sin[x]^n, {n, 1, Infinity}]

Sin[x]
-----
-1 + Sin[x]

Sum[ x^(3 n) / (1 + n^2), {n, 0, Infinity}]

1
-- (Hypergeometric2F1[-i, 1, 1 - i, x^3] + Hypergeometric2F1[i, 1, 1 + i, x^3])
2

```

*entering power series you have the coefficients for*

Note that when working with series it is very common to get special functions like the Zeta function or the various Hypergeometric functions. SumConvergence can also be used with a power series to see where it converges; the Reduce command can often be used to simplify the results:

```

SumConvergence[ $\frac{(x-2)^{3n}}{5n+1}, n]$ 

Abs[-2+x]3 ≤ 1 && (-2+x)3 ≠ 1

Reduce[%, x, Reals]

1 ≤ x < 3

```

*the interval of convergence for a power series*

SumConvergence will also work with the Assumptions option; this is useful when your series has some parameters in it for which you don't know a precise value but you do know a range.

You can also have Mathematica find the power series for a given function, center, and error term. For example if you want to find the formula for the power series for *formula* centered at *variable=a* up to the power *n* with an error term to represent higher powers, use Series[*formula*, {*variable*, *a*, *n* }]. So if you wanted to find a series for sin(3x)cos(x) centered at 0 up to the tenth power you would use Series[ Sin[3x]Cos[x], {x,0,10}]:

```

Series[ Sin[3 x] Cos[x], {x, 0, 10}]

3 x - 6 x3 +  $\frac{22 x^5}{5}$  -  $\frac{172 x^7}{105}$  +  $\frac{38 x^9}{105}$  + O[x]11

```

*a series for a function (including an 11<sup>th</sup> order error term)*

In this example the coefficient for  $x^{10}$  happens to be 0 and the error term is on the order of  $x^{11}$ . If we were to remove the representation of the error term we would have what is commonly known as a Taylor polynomial. You can remove the error term with the command Normal:

```

Series[ Sin[3 x] Cos[x], {x, 0, 10}]

3 x - 6 x3 +  $\frac{22 x^5}{5}$  -  $\frac{172 x^7}{105}$  +  $\frac{38 x^9}{105}$  + O[x]11

Normal[%]

3 x - 6 x3 +  $\frac{22 x^5}{5}$  -  $\frac{172 x^7}{105}$  +  $\frac{38 x^9}{105}$ 

```

*stripping an error term to get a Taylor polynomial*

Whether you want the error term or not depends on what you are doing. If you are going to be evaluating your expression at a number or graphing it then you will want to remove the error term. If you are going to be combining different series you want to keep the error terms intact as Mathematica will automatically keep track of how the errors combine (essentially keeping track of the “most significant power” for you):

```

mess1 = Series[ Tan[x], {x, 0, 12}]


$$x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^7}{315} + \frac{62x^9}{2835} + \frac{1382x^{11}}{155925} + O[x]^{13}$$


mess2 = Series[ Cos[2 x], {x, 0, 6}]


$$1 - 2x^2 + \frac{2x^4}{3} - \frac{4x^6}{45} + O[x]^7$$


mess1 + mess2


$$1 + x - 2x^2 + \frac{x^3}{3} + \frac{2x^4}{3} + \frac{2x^5}{15} - \frac{4x^6}{45} + O[x]^7$$


mess1 * mess2


$$x - \frac{5x^3}{3} + \frac{2x^5}{15} - \frac{5x^7}{63} + O[x]^8$$


Series[ Tan[x] Cos[2 x], {x, 0, 7}]


$$x - \frac{5x^3}{3} + \frac{2x^5}{15} - \frac{5x^7}{63} + O[x]^8$$


```

*arithmetic of power series with error terms*

Notice that in each case the error term  $O[x]$  tracks the significant powers and that finding the individual series and then multiplying gives the same result as multiplying the two formulas and then finding the series.

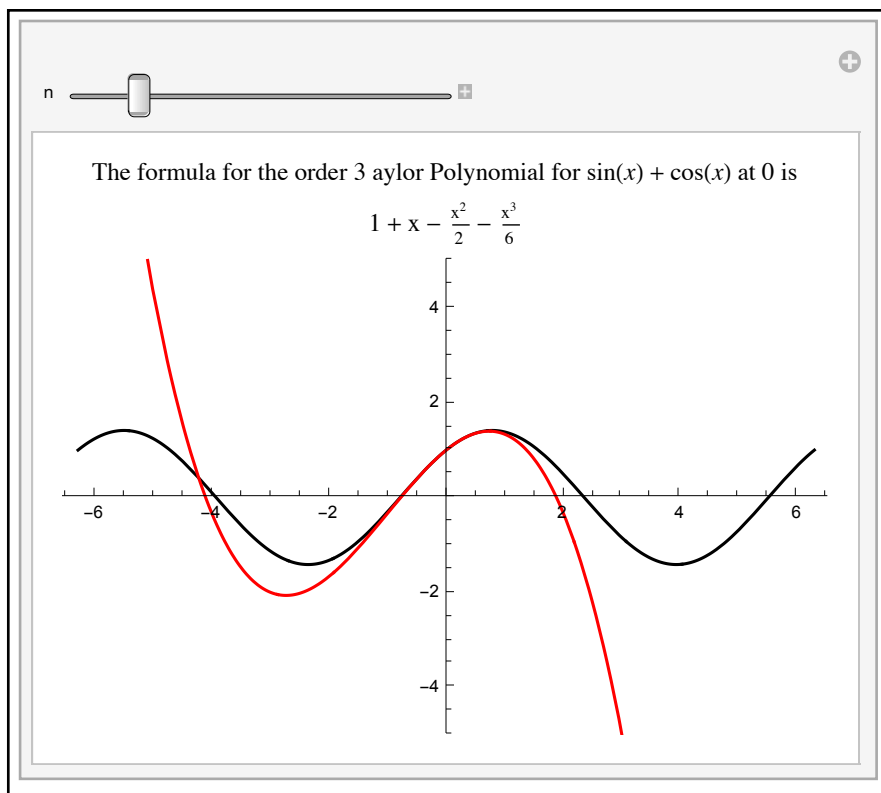
We end this section with an example of using Manipulate to illustrate the basic idea behind power series and Taylor polynomials. As you take more and more terms of a power series, you should get a polynomial which is closer and closer to the original function. We can visualize this by using Manipulate to graph a function and the series as we take more and more terms. To do this for say  $\sin(x) + \cos(x)$  from  $-2\pi$  to  $2\pi$ , try evaluating the following commands:

```

f[n_] := Normal[Series[Sin[x] + Cos[x], {x, 0, n}]]
Manipulate[Column[{
  Text[Row[ {"The formula for the order ", n,
    " Taylor Polynomial for ", TraditionalForm[ Sin[x] + Cos[x]], " at 0 is"} ]],

```

```
Text[f[n]], Plot[Evaluate[{Sin[x] + Cos[x], f[n]}], {x, -2 Pi, 2 Pi},
PlotStyle → {Black, Red}, PlotRange → {-5, 5},
ImageSize → 400]], Alignment → Center], {n, 1, 15, 1}]
```



*graphing a function and its Taylor polynomials via Manipulate*

This manipulation creates a column of the formula for the  $n^{\text{th}}$  order Taylor polynomial for  $\sin(x) + \cos(x)$  along with its graph in red from  $-2\pi$  to  $2\pi$ . As you drag the slider you will add more terms to the Taylor polynomial and the graph should bend closer and closer to the true graph of  $\sin(x) + \cos(x)$  (as you add lots of terms you may need to make the graph larger to make it easier to see).

### Section 5.3 Homework - The Computations of Calculus

- 1) Find the limit of  $\frac{1 - \cos(x)}{x^2}$  as  $x$  goes to 0.
- 2) Find the limit of  $\frac{\sin(x)}{x^3}$  as  $x$  goes to 0 from either side.
- 3) Find the limit of  $(1 + \frac{3}{t})^t$  as  $t$  goes to infinity.
- 4) Find the limit of  $x^x$  as  $x$  goes to 0 from the right.



- 5) Find the limit of  $\frac{\sqrt{16-x^2}}{\sqrt{x}-2}$  as  $x$  goes to 4 from the 4 “cardinal” complex directions 1, -1,  $i$ ,  $-i$ .
- 6) Find the limit of  $\sin(\ln(x)) + 2$  as  $x$  goes to infinity.
- 7) Find the first 4 derivatives of  $\sqrt[3]{x^2 - 3x + 5}$ .
- 8) Find the 11<sup>th</sup> derivative of  $e^{x/2} \sin(2x)$ .
- 9) Find an antiderivative of  $x^4 e^{3x}$ .
- 10) Find an antiderivative of  $\frac{1}{x e^x}$ .
- 11) Find an antiderivative of  $\frac{x^{10}}{x^6 - 1}$ .
- 12) Find an antiderivative for  $\frac{\sin(t)}{1 + \tan(t)}$ .
- 13) Find  $\int_0^6 (x^2 - 3x + 1)^4 dx$ .
- 14) Find  $\int_0^\pi \sin^4(x) \cos^3(x) dx$ .
- 15) Find  $\int_0^\infty x^5 e^{-2x} dx$ .
- 16) Find  $\int_1^\infty \frac{\ln(x)}{x} dx$ .
- 17) Find the fourth order Taylor polynomial for  $x^{1/2}$  at  $x=1$ . How does this change if you center it at  $x=4$ ?
- 18) Find the eleventh order Taylor polynomial for  $\cos(x)$  at  $x=\pi$ .
- 19) Find  $\sum_{n=1}^{\infty} \frac{1}{n^2 + 3n}$ .
- 20) Find the sum  $\sum_{n=2}^{\infty} \frac{(-1)^n}{\ln(n)}$ . Estimate it to 20 places.
- 21) If  $f(x)$  is an unknown function, find the second derivative of  $1/\ln(f(x))$ .
- 22) Find the derivative of  $x^2 + x \sin(y) + y^2 = 1$  at the point  $(1,0)$ . Use this to graph the function together with its tangent line at  $(1,0)$ .
- 23) Find  $\sum_{n=0}^{\infty} \frac{x^{2n}}{3n+1}$ . Graph this function.
- 24) If we define  $g(x) = \int_{t=0}^x f(t)^3 dt$ , find the first two derivatives of  $g(x)$ .
- 25) Estimate  $\int_0^\pi \tan(\sin(x)) dx$  to 30 places.
- 26) Estimate  $\int_0^5 e^{-x^2} dx$  to 20 places.

27) What is the interval of convergence for  $\sum_{n=1}^{\infty} \frac{(-2)^n(x-3)^{2n}}{5n-3}$ ?

28) Does the series  $\sum_{n=1}^{\infty} \frac{\sin(n)}{n^2}$  converge or diverge? If it converges estimate its sum to 10 places.

## Section 5.4 - Applications of the Derivative

Now that we have a grasp of how to perform the basic computations of calculus it is time to see how you can use Mathematica to assist you in calculus applications. Since calculus is used in many different areas in this section we will consider some of the applications of the derivative and hold off on applications of the integral until the next section. In both sections we will assume that you are already familiar with the applications from the calculus side so we can focus on how to use Mathematica for their implementation. Some of these applications can be reproduced in Mathematica using commands such as `Maximize` or those dealing with region objects (which we'll discuss in Section 5.6) - for practice we will be doing them as you would in a calculus course.

Example 1: A classic "Related Rates" problem.

Oil is leaking from a tanker, forming a circular spill. Enough oil is leaking from the tanker to increase the area of the spill at 400 square feet per minute. How fast is the radius of the spill changing when it is 1000 feet?

To solve this problem the idea is to think of both the area and radius of spill as functions that change over time (that is we can write them as functions  $A(t)$  and  $r(t)$ ). These quantities are related (hence the name of the problem type) by the geometric equation  $A(t) = \pi r(t)^2$ . By differentiating both sides of this equation we can get a new equation that relates  $A'(t)$ ,  $r'(t)$ , and  $r(t)$ . Since we know the values for  $A'(t)$  and  $r(t)$ , we can solve this for  $r'(t)$ :

```
equation = (A[t] == Pi r[t]^2)

A[t] ==  $\pi r[t]^2$ 

newequation = D[equation, t]

A'[t] ==  $2 \pi r[t] r'[t]$ 

solution = r'[t] /. Solve[newequation, r'[t]]

 $\left\{ \frac{A'[t]}{2 \pi r[t]} \right\}$ 

answer = solution /. {A'[t] -> 400, r[t] -> 1000}

 $\left\{ \frac{1}{5 \pi} \right\}$ 
```

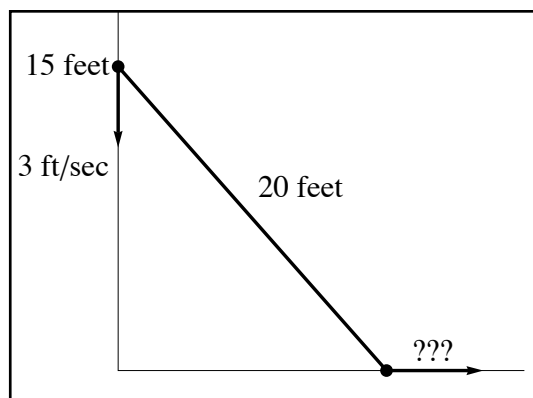
*solving a related rates problem*

So in this case the radius is increasing at a rate of  $\frac{1}{5\pi}$  feet per second.

Example 2: A related rates problem with two known rates

A 20 foot ladder is sliding down a wall it leans against. When the top of the ladder is 15 feet from the ground it is sliding down at 3 feet per second. How fast is the base of the ladder sliding outward at this moment?

To start it's a good idea to draw a picture so that we can see what is involved:



*the sliding ladder*

If we let  $x(t)$  be the position of the foot of the ladder at any time and  $y(t)$  be the position of the top of the ladder, by the Pythagorean we have that at any time  $t$   $x(t)^2 + y(t)^2 = 20^2$ . At the time in question we know  $y(t) = 15$  and  $y'(t) = -3$  feet/second. We are trying to find the value of  $y'(t)$  at this instant. As we will probably need the value of  $y(t)$  for this, it is a good idea to find it first:

```
Solve[ 15^2 + x[t]^2 == 20^2, x[t]]
{{x[t] -> -5 Sqrt[7]}, {x[t] -> 5 Sqrt[7]}}

equation = D[ x[t]^2 + y[t]^2 == 20^2, t]
2 x[t] x'[t] + 2 y[t] y'[t] == 0

generalsolution =
  x'[t] /. Solve[ equation, x'[t] ][[1]]
- (y[t] y'[t]) / x[t]

currentsolution =
  generalsolution /.
    {y[t] -> 15, y'[t] -> -3, x[t] -> 5 Sqrt[7]}
9 / Sqrt[7]
```

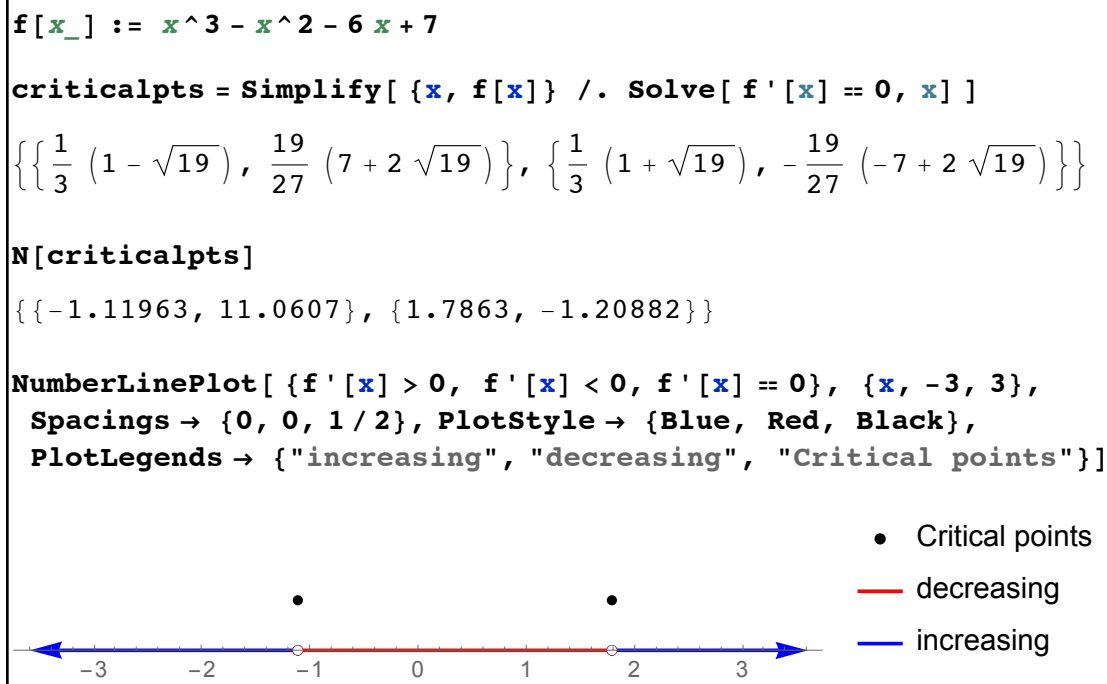
*finding the speed of the ladder's foot*

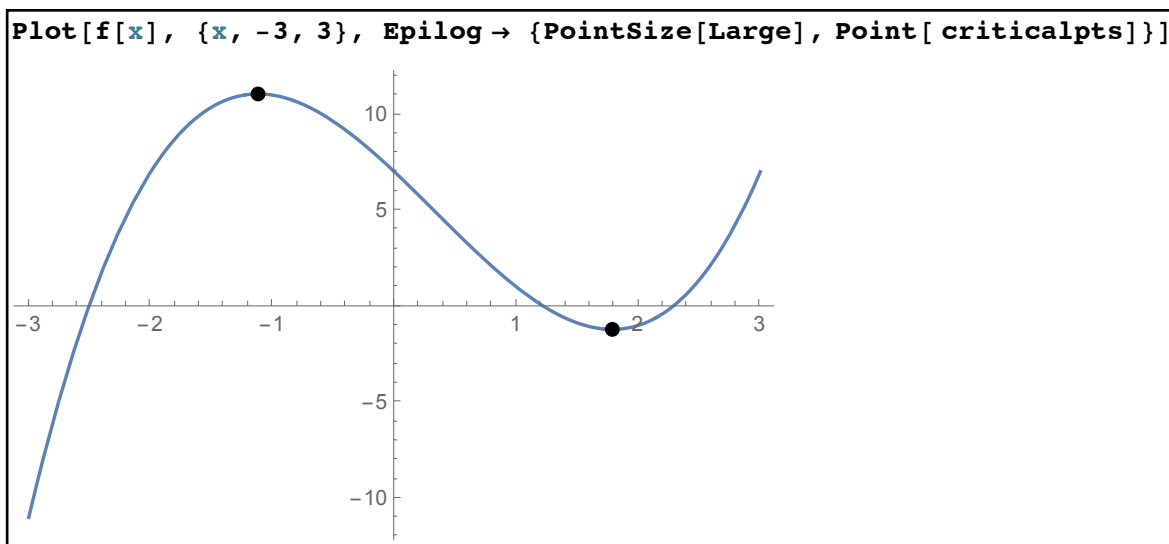
So it looks like the speed of the ladder's foot is  $\frac{9}{\sqrt{7}}$  feet per second. In this case we had to use a replacement rule for each of  $x(t)$ ,  $x'(t)$ , and  $y(t)$ , and even followed through with the right units for distance and time. You can actually have Mathematica keep the units for you all the way through the problems like this - an example of this is shown in Section 5.5.

### Example 3: The first derivative test

Use the first derivative test to find the local maxima and minima for  $f(x) = x^3 - x^2 - 6x + 7$ .

Recall from first semester calculus that on an interval a positive derivative means we are increasing, a negative derivative means decreasing, and that places where the first derivative is 0 or fails to exist (i.e. critical points) are potential maxima and minima. The derivative here will always exist so we really only need to worry about where the derivative is positive, negative, or zero. We can use the derivative to determine the critical points and then use information about where  $f(x)$  is increasing or decreasing to classify them. The easiest way to classify them is to use NumberLinePlot to show where the function is increasing and decreasing:





*finding and classifying maxima and minima*

From the Solve command it looks like we have two critical points based at  $x = \frac{1 \pm \sqrt{19}}{3}$  (using `{x,f[x]}/. Solve....` gives us the critical points in true coordinate form and the Simplify command makes Mathematica clean up the results for us). To get a feel for where the actual points are in the plane we use the N command to get estimates of the coordinates (which will be useful when the time comes to graph). The NumberLinePlot command shows us where the function is increasing and decreasing relative to the critical points. Looking at the point  $x = \frac{1 - \sqrt{19}}{3}$  it appears that the function is increasing on the left and decreasing on the right - so that point must be a local maximum. Similarly, for the second point where  $x = \frac{1 + \sqrt{19}}{3}$ , we have the function is decreasing on the left and increasing on the right. Therefore the second point must be a local minimum. Our estimates for the coordinates tells us that we need to graph from at least  $x = -2$  to  $x = 2$  to see both extrema and looking at our graph we have confirmation of the two extreme points.

#### Example 4: The second derivative test

Use the second derivative test to find the local maxima and minima for  $f(x) = x^3 - x^2 - 6x + 7$ .

The second derivative test essentially states that at a critical point on the graph a positive second derivative means the point is a local minimum and a negative second derivative means that the point is a local maximum (a second derivative of 0 tells us nothing).

```

f[x_] := x^3 - x^2 - 6 x + 7

criticalpts = Simplify[ {x, f[x]} /. Solve[ f'[x] == 0, x ] ]
 $\left\{ \left\{ \frac{1}{3} (1 - \sqrt{19}), \frac{19}{27} (7 + 2 \sqrt{19}) \right\}, \left\{ \frac{1}{3} (1 + \sqrt{19}), -\frac{19}{27} (-7 + 2 \sqrt{19}) \right\} \right\}$ 

localmaxima = Select[ criticalpts, f''[#1[[1]]] < 0 &]
 $\left\{ \left\{ \frac{1}{3} (1 - \sqrt{19}), \frac{19}{27} (7 + 2 \sqrt{19}) \right\} \right\}$ 

localminima = Select[ criticalpts, f''[#1[[1]]] > 0 &]
 $\left\{ \left\{ \frac{1}{3} (1 + \sqrt{19}), -\frac{19}{27} (-7 + 2 \sqrt{19}) \right\} \right\}$ 

```

*using the second derivative to classify maxima and minima*

As before we use a combination of Solve and Simplify to find the list critical points. We can then use Select on that list to find the places where the second derivative is positive (for minima) or negative (for maxima). It's important to use #1[[1]] and not just #1 in the Select command; the critical points have two coordinates and you need to plug only the first one into the second derivative.

Note that this is a bit easier (we don't need to piece together the sign information from NumberLinePlot) but remember there are problems where the second derivative test can fail (by having a second derivative of 0) but in which the first derivative test still works.

Example 5: The distance from a point to a parabola.

Which point on  $y=x^2+3$  is closest to  $(-6,9)$ ?

Any point on the parabola has the form  $(x, x^2+3)$  since it has the equation  $y = x^2 + 3$ . Using the formula for the distance between two points, the distance from  $(-6,9)$  to  $(x, x^2 + 3)$  is  $\sqrt{(x - -6)^2 + ((x^2 + 3) - 9)^2} = \sqrt{(x + 6)^2 + (x^2 - 6)^2}$  (if you want you can use EuclideanDistance to find this directly). We want to find the minimum value for this for all choices of  $x$ . This expression is differentiable everywhere so the only critical points will be where the derivative is 0. Since there is no restriction on  $x$  we can't use the "closed interval" technique (where you find the critical points, the endpoints of the interval, and then just make a table of values). In addition the derivative turns out to be a fraction so we must check places where the denominator of the derivative is 0:

```

f[x_] := Sqrt[(x + 6)^2 + (x^2 - 6)^2]
f'[x]

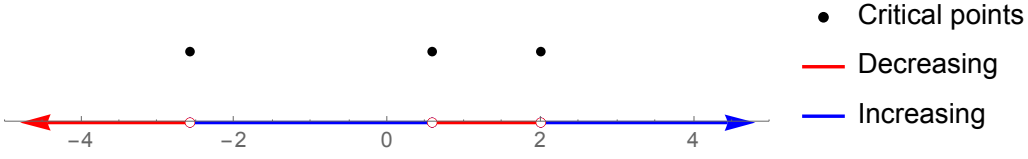
$$\frac{2(6 + x) + 4x(-6 + x^2)}{2\sqrt{(6 + x)^2 + (-6 + x^2)^2}}$$


criticalnumbers = Reduce[f'[x] == 0, x, Reals] ||
  Reduce[Denominator[f'[x]] == 0, x, Reals]
x == 2 || x ==  $\frac{1}{2}(-2 - \sqrt{10})$  || x ==  $\frac{1}{2}(-2 + \sqrt{10})$ 

N[criticalnumbers]
x == 2. || x == -2.58114 || x == 0.581139

NumberLinePlot[{f'[x] > 0, f'[x] < 0, criticalnumbers}, {x, -4, 4},
  Spacings -> {0, 0, 1},
  PlotLegends -> {"Increasing", "Decreasing", "Critical points"},
  PlotStyle -> {Blue, Red, Black}]

```



```

N[f[ $\frac{1}{2}(-2 - \sqrt{10})$ ]]
3.48242

N[f[2]]
8.24621

closestpoint = Simplify[{x, x^2 + 3} /. x ->  $\frac{1}{2}(-2 - \sqrt{10})$ ]
 $\left\{-1 - \sqrt{\frac{5}{2}}, \frac{13}{2} + \sqrt{10}\right\}$ 

```

*finding where the function is increasing or decreasing*

Now we are looking for the minimum value of the distance. Looking at where the function decreasing we can see the distance always decreases until  $x = \frac{-2 - \sqrt{10}}{2}$ . Likewise the distance is always increasing after  $x = 2$ . This means the minimum value for the distance must

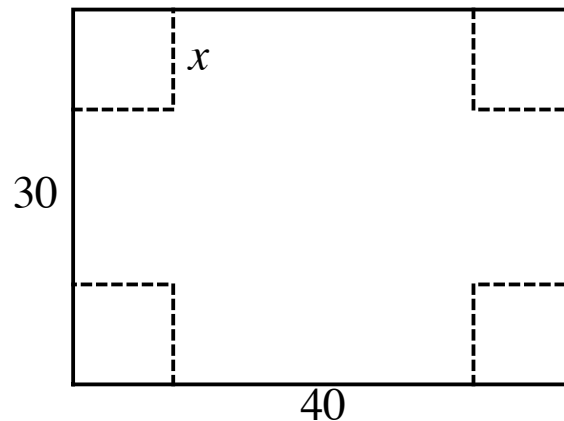


be in the closed interval  $\frac{-2 - \sqrt{10}}{2} \leq x \leq 2$ . As the function is increasing from  $\frac{-2 - \sqrt{10}}{2}$  to  $\frac{-2 + \sqrt{10}}{2}$  we know that  $x = \frac{-2 + \sqrt{10}}{2}$  cannot correspond to a minimum. So need only check the values  $\frac{-2 + \sqrt{10}}{2}$  and 2 - and it turns out that  $x = \frac{-2 + \sqrt{10}}{2}$  corresponds to the minimum, which gives us the point  $(-1 - \sqrt{\frac{5}{2}}, \frac{13}{2} + \sqrt{10})$  is closest to the parabola.

#### Example 6: The box problem

You have a square piece of cardboard that measures 30" by 40". By removing squares of equal size from each corner and folding up the resultant flaps you can make an open-topped box. What size squares should you remove to maximize the volume of the box, and what will the maximum volume/size be?

Let  $x$  be the size of the square you will remove. Then  $x$  can be no smaller than 0 and no larger than 15 (i.e. in the interval  $[0, 15]$ ). This gives the following picture of the original cardboard:



The depth of the box will be  $x$  since that is the size of the squares we are removing. The length of the box will be  $30 - 2x$  and the width will be  $40 - 2x$ . So the problem reduces to finding the maximum of  $x(30 - 2x)(40 - 2x)$  where  $x$  is in the interval  $[0, 15]$ :

```

volume = x (30 - 2 x) (40 - 2 x)

(30 - 2 x) (40 - 2 x) x

criticalnumbers = Solve[ D[volume, x] == 0, x]

{{x ->  $\frac{5}{3} (7 - \sqrt{13})$ }, {x ->  $\frac{5}{3} (7 + \sqrt{13})$ }}

N[criticalnumbers]

{{x -> 5.65741}, {x -> 17.6759}}

allvalues = {{x -> 0}, {x -> 5 / 3 (7 - Sqrt[13])}, {x -> 15}}

{{x -> 0}, {x ->  $\frac{5}{3} (7 - \sqrt{13})$ }, {x -> 15}}

Simplify[volume /. allvalues]

{0,  $\frac{1000}{27} (35 + 13 \sqrt{13})$ , 0}

dimensions = Simplify[ {30 - 2 x, 40 - 2 x, x} /. {x -> 5 / 3 (7 - Sqrt[13])}]

{ $\frac{10}{3} (2 + \sqrt{13})$ ,  $\frac{10}{3} (5 + \sqrt{13})$ ,  $-\frac{5}{3} (-7 + \sqrt{13})$ }

```

*finding the largest box*

There are only two places where the derivative of the volume is 0 and by estimating the two values we can see the only one that is in  $[0, 15]$  is  $x = \frac{5(7 - \sqrt{13})}{3}$  (this could also be done using Select and the criterion  $0 \leq x / .\#1 \leq 15 \& .$ ). Since the endpoints 0 and 15 are possible values we plug 0,  $\frac{5(7 - \sqrt{13})}{3}$ , and 15 in for  $x$  in the volume. 0 and 15 both give a volume of 0 (which makes sense if you think about it) so the maximum must occur when we cut squares of side  $x = \frac{5(7 - \sqrt{13})}{3}$ . The maximum volume is  $\frac{1000(35 + 13\sqrt{13})}{27}$ , and looking at the actual dimensions of the box we find that the box is  $\frac{10(2 + \sqrt{13})}{3} \times \frac{10(5 + \sqrt{13})}{3} \times \frac{-5(-7 + \sqrt{13})}{3}$  inches (the last number being positive).

Example 7: The generalized box problem.

Given the same procedure as in problem 6, find the largest box you can make from a piece of cardboard that is  $a$  inches by  $b$  inches, where  $a \leq b$ .

The setup for this problem is almost identical to that of problem 6, except for the unknown constants  $a$  and  $b$ . Since  $a \leq b$ , we know that if  $x$  is the size of the squares being cut,  $x$  must be at least 0 and no more than  $a/2$ . The sides of the box will be  $x$ ,  $a - 2x$ , and  $b - 2x$  respectively, giving a volume of  $x(a - 2x)(b - 2x)$ . So we need to maximize  $x(a - 2x)(b - 2x)$  on the interval  $[0, a/2]$ . We can proceed like we did in problem 6, but we soon hit a snag:

$$\text{volume} = x (a - 2x) (b - 2x)$$

$$(a - 2x) (b - 2x) x$$

$$\text{derivative} = D[\text{volume}, x]$$

$$(a - 2x) (b - 2x) - 2 (a - 2x) x - 2 (b - 2x) x$$

$$\text{Solve}[\text{derivative} == 0, x]$$

$$\left\{ \left\{ x \rightarrow \frac{1}{6} \left( a + b - \sqrt{a^2 - ab + b^2} \right) \right\}, \left\{ x \rightarrow \frac{1}{6} \left( a + b + \sqrt{a^2 - ab + b^2} \right) \right\} \right\}$$

*finding the critical values for the general box problem*

The snag is that we know that  $x$  must be in the interval  $[0, a/2]$  but since the two solutions involve the unknowns  $a$  and  $b$  it isn't clear which of them (if any) are in the right range. When we had specific numbers for  $a$  and  $b$  we could just numerically estimate the values but that is not an option here. The easiest way around this is to use `Reduce` instead of `Solve`. `Reduce` will let us add information about  $a$ ,  $b$ , and  $x$ :

$$\text{Reduce}[\text{derivative} == 0 \ \&\& \ 0 < a \leq b \ \&\& \ 0 \leq x \leq a/2, x]$$

$$b > 0 \ \&\& \ \left( \left( 0 < a < b \ \&\& \ x == \frac{a+b}{6} - \frac{1}{6} \sqrt{a^2 - ab + b^2} \right) \ || \right.$$

$$\left. \left( a == b \ \&\& \ \left( x == \frac{a+b}{6} - \frac{1}{6} \sqrt{a^2 - ab + b^2} \ || \ x == \frac{a+b}{6} + \frac{1}{6} \sqrt{a^2 - ab + b^2} \right) \right) \right)$$

*using Reduce to get more information*

By adding the information  $0 \leq a \leq b$  and  $0 \leq x \leq a/2$ , `Reduce` tells us there are two cases - one where  $a < b$  (i.e. the cardboard is not a square) and the only critical point is

$$x = \frac{a+b}{6} - \frac{\sqrt{a^2 - ab + b^2}}{6} \text{ and the square case where } a=b \text{ and both critical values still apply.}$$

We can do better, however. Clearly neither  $x=0$  nor  $x=a/2$  will give the maximum volume (why?). So we can strengthen the inequality  $0 \leq x \leq a/2$  to  $0 < x < a/2$ :

**Reduce[ derivative == 0 && 0 < a ≤ b && 0 ≤ x < a / 2, x]**

$$b > 0 \&\& 0 < a \leq b \&\& x == \frac{a+b}{6} - \frac{1}{6} \sqrt{a^2 - ab + b^2}$$

*using Reduce with a stronger constraint*

By eliminating the endpoints 0 and  $a/2$  from consideration we can see there is only one possible critical value regardless of whether the cardboard is square or not. Since there is only one critical point and the endpoints 0 and  $a/2$  do not correspond to maximum values, the maximum value must occur at  $x = \frac{a+b}{6} - \frac{\sqrt{a^2 - ab + b^2}}{6}$ . Substituting this value for  $x$  into the volume formula and trying to find the dimensions finishes up the problem:

**maxvolume = FullSimplify[ volume /. x →  $\frac{a+b}{6} - \frac{1}{6} \sqrt{a^2 - ab + b^2}$ , 0 ≤ a ≤ b]**

$$-\frac{1}{54} \left( -a - b + \sqrt{a^2 - ab + b^2} \right) \left( 2a - b + \sqrt{a^2 - ab + b^2} \right) \left( -a + 2b + \sqrt{a^2 - ab + b^2} \right)$$

**dimensions = FullSimplify[{a - 2x, b - 2x, x} /. x →  $\frac{a+b}{6} - \frac{1}{6} \sqrt{a^2 - ab + b^2}$ , 0 ≤ a ≤ b]**

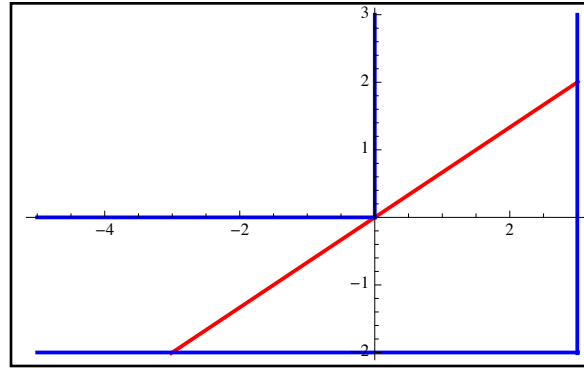
$$\left\{ \frac{1}{3} \left( 2a - b + \sqrt{a^2 - ab + b^2} \right), \frac{1}{3} \left( -a + 2b + \sqrt{a^2 - ab + b^2} \right), \frac{1}{6} \left( a + b - \sqrt{a^2 - ab + b^2} \right) \right\}$$

*FullSimplify didn't help, but it can't hurt to try*

#### Example 8: Sliding a pipe around a corner

Imagine two hallways, one of width  $a$  and another of width  $b$  ( $a \leq b$ ), which intersect in an "L" corner. What is the longest pipe (or rigid wire or anything else whose thickness is negligible compared to the hallway widths) that can be slid on the floor around the corner?

To help visualize this think of the pipe as spring-loaded so that it always extends to touch the walls as we move it around the hallway. As the pipe is on the floor we can think of it as a line segment in the plane. For any given inclination of the segment the longest pipe that can fit in the hallway would have to graze the inside corner and the two walls. If we let the origin be the inside corner of the turn and think of the width  $a$  hallway as horizontal and the width  $b$  hallway as vertical, we would have the following picture (which corresponds to  $a=2$  and  $b=3$ ):



*moving a pipe around a corner (pipe in red, walls in blue)*

As you try to swivel the pipe from its “slope 0” inclination to its near-vertical inclination the length of the pipe goes from very large, to some minimum value, and then back to very large. That minimum value is the length of the longest pipe that will make it around - it will scrape against the walls and corner just once and then make it through (anything longer will get stuck). So we wish to minimize the length of the line segment as the slope goes over all positive values, and we expect a single critical value for the slope.

To set this up note that the equation of the line is  $y = mx$  (for some slope  $m$ ) because it goes through the origin. The right end of the pipe corresponds to when  $x=b$ , and the left end to when  $y=-a$ . Since we have the equation  $y=mx$ , this is enough information to get the coordinates of the pipe ends, and therefore the distance between them (which is the length of the pipe):

```

pointonpipe = {x, m x};

rightend = pointonpipe /. x -> b
{b, b m}

leftend = pointonpipe /. Solve[{y == m x, y == -a}, {x, y}][[1]]
{ -a/m, -a }

lengthofpipe =
  FullSimplify[ EuclideanDistance[ rightend, leftend], a > 0 & b > a & m >= 0 ]

(a + b m)  $\sqrt{1 + m^2}$ 
m

```

*the formula for the length of pipe with “slope  $m$ ” in the  $a \times b$  hallway*

We can now take the critical points (in terms of  $m$ ) for this formula by taking the derivative of the length and setting it equal to 0:

```
criticalvalues = Solve[D[lengthofpipe, m] == 0, m]
```

$$\left\{ \left\{ m \rightarrow \frac{a^{1/3}}{b^{1/3}} \right\}, \left\{ m \rightarrow -\frac{(-1)^{1/3} a^{1/3}}{b^{1/3}} \right\}, \left\{ m \rightarrow \frac{(-1)^{2/3} a^{1/3}}{b^{1/3}} \right\} \right\}$$

*the critical values for the length*

At first glance it may be unclear which of the three are true critical values. Recall however that in Mathematica the cube root of a negative is counted as a nonreal complex number so the second and third values are probably complex ones (and therefore not valid choices). If the expressions for  $m$  had more than one term in them determining which roots were real would be more difficult to do since with several terms imaginary parts in various terms could cancel out (Reduce might help in this case). In our simple one-term formulas it is fairly clear that only the true/real critical value is the first one. Since there is only one critical value and the minimum exists the minimum must happen at the critical value:

```
truecritical = criticalvalues[[1]]
```

$$\left\{ m \rightarrow \frac{a^{1/3}}{b^{1/3}} \right\}$$

```
solution = Simplify[lengthofpipe /. truecritical]
```

$$\sqrt{(a^{2/3} + b^{2/3})^3}$$

*the true critical point and minimum*

So the shortest pipe which scrapes the walls (i.e. the longest one that can make it around the corner) has a length of  $\sqrt{(a^{2/3} + b^{2/3})^3} = (a^{2/3} + b^{2/3})^{3/2}$ .

Example 9: Newton's method estimates for a cube root

Give several good approximations to  $\sqrt[3]{7}$  via Newton's method.

Newton's method states that under most circumstances if  $x_{old}$  is an good approximation to  $f(x) = 0$ , then  $x_{new} = x_{old} - \frac{f(x_{old})}{f'(x_{old})}$  is a better approximation. We can think of  $\sqrt[3]{7}$  as the solution to  $f(x) = x^3 - 7$ , with  $x=2$  being a good starting approximation. The Newton's method formula then becomes

$$x_{new} = x_{old} - \frac{f(x_{old})}{f'(x_{old})} = x_{old} - \frac{x_{old}^3 - 7}{3x_{old}^2} = \frac{3x_{old}^3}{3x_{old}^2} - \frac{x_{old}^3 - 7}{3x_{old}^2} = \frac{2x_{old}^3 + 7}{3x_{old}^2}. \text{ The easiest way to}$$

process this in Mathematica is to think of the successive Newton's Method iterates as coming from a recursively defined function  $g(n)$  given by  $g(n) = \frac{2g(n-1)^3 + 7}{3g(n-1)^2}$  where  $g(1)=2$ :

```

g[n_] := g[n] = (2 g[n - 1]^3 + 7) / (3 g[n - 1]^2)
g[1] = 2
2

Table[g[n], {n, 1, 5}]

{2, 23/12, 18215/9522, 9065194654643/4738902651675, 2234872012283865246564137724025796774539/1168297130817268091225151745493343231225}

N[g[5], 20]

1.9129311827723891012

N[7^(1/3), 20]

1.9129311827723891012

```

*successive approximations to  $\sqrt[3]{7}$  via Newton's Method*

So in this case it appears that the fourth new approximation  $g(5)$  ( $g(1)=2$  being the original) is a very good approximation to  $\sqrt[3]{7}$ . Even better it is a rational number approximation (in fact, Newton's method applied to finding any root of a positive rational number from a rational initial guess will yield a rational number). We could also have approached this using NestList as NestList[(2 #1^3+7)/(3 #1^2) &, 2, 5].

## Section 5.4 Homework - Applications of the Derivative

- 1) Find the maximum and minimum values of  $x^2 + x^{-2}$  on  $[1/2, 2]$ .
- 2) Find the maximum value for  $\frac{x^3}{e^x}$  for  $x \geq 0$ .
- 3) Determine the intervals on which  $y = x^4 - 2x^2 - 1$  is increasing, decreasing, concave up, or concave down. Find and classify all the extrema and the inflection points.
- 4) Use Mathematica to show that the largest rectangle (in terms of area) than can be inscribed in a circle of radius  $r$  is a square, and find the area.
- 5) A cylinder with an open top and closed bottom is to have a volume of 54 cubic inches. What is the smallest surface area of such a cylinder?
- 6) Find the largest cylinder (in terms of volume) that can be inscribed in a sphere of radius  $r$ .
- 7) Find the point on the parabola  $y = x^2$  closest to (1,3).
- 8) Find the minimum distance from the point  $(x_0, y_0)$  to the line  $y = mx + b$ . Does this answer agree with the formula from the distance from a point to a line?
- 9) Use Newton's method to estimate  $\sqrt{10}$ .
- 10) Use Newton's method to estimate the real roots of  $3x^5 - 15x + 5 = 0$ .

- 11) Use Newton's method to estimate the complex roots of  $x^4 - x^3 + 200 = 0$ . (Hint: There are 4 roots, and Newton's method works just as well with complex numbers as it does with real ones)
- 12) A snowball is melting at the rate of 1 cubic inch per minute. How fast is the radius of the snowball shrinking when it is 5 inches?
- 13) Two cars approach an intersection, one from the north and one from the east. The car from the north is moving at 50 miles an hour, the one from the east at 60 miles an hour. If the north car is  $1/2$  mile from the intersection and the east car is 1 mile from the intersection, how fast is the distance between the cars changing?
- 14) Use the Minimize command to redo Example 5.
- 15) Another way to find the closest point on the parabola in Example 5 is to use region objects. If the parabola can be represented as `ImplicitRegion[ y==x^2+3, {x,y} ]`, look up the command `RegionNearest` and use it to solve the problem.

Optional problem: Suppose you have a 7' by 14' "L" hallway, and a heavy rectangular table that is 2' wide. If the table cannot be tilted vertically, what is the longest such table you can get around the corner? This is a generalization of the pipe problem in this section, and a similar picture can be drawn. The hard part is that the corners of the table that touch the wall are not on the line through the origin, but on a line parallel to it that is 2 units from the origin. You can find the  $y$ -intercept of this line by using the "distance from a line to a point" formula and setting this equal to 2.



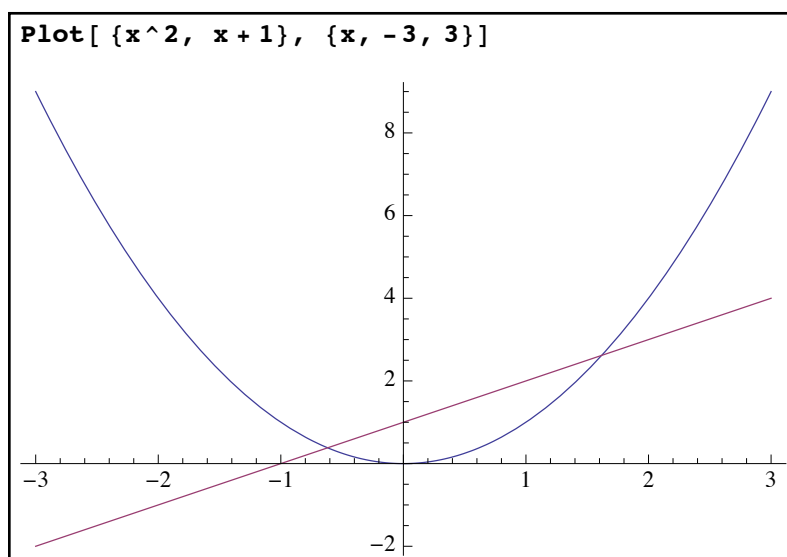
## Section 5.5 - Applications of the Integral

In addition to helping with applications that are based on the derivative Mathematica is also very useful for applications based on the integral (even more so since integration is significantly more complicated than differentiation). Having Mathematica handle the “grunt work” of the integration (exact or numerical) as well as some the work involved in setting problems up can be a huge advantage.

### Example 1: A simple area

The most basic application of the definite integral is in finding the area between two curves. If  $f(x) \geq g(x)$  on  $[a, b]$ , then the area between the curves  $y = f(x)$ ,  $y = g(x)$ ,  $x = a$ , and  $x = b$  is  $\int_a^b f(x) - g(x) dx$ . As an example let's find the area between  $y = x + 1$  and  $y = x^2$ .

Looking at the curve via the Plot command, we see:



*area between a parabola and a line*

This is a fairly simple region - it extends from an  $x$ -value around -0.5 up to about 1.5 or so, on on that range the line is on top and the parabola is on bottom. We will need the exact  $x$ -coordinates of the intersection points for the limits of the integral:

```
corners = x /. Solve[x^2 == x + 1, x]
```

$$\left\{ \frac{1}{2} \left( 1 - \sqrt{5} \right), \frac{1}{2} \left( 1 + \sqrt{5} \right) \right\}$$

*the x-coordinates for the intersection points*

The first number clearly corresponds to the left intersection and the second to the right intersection (you could always use N to estimate the coordinates if you are unsure). Since  $y=x+1$  is always on top and  $y=x^2$  is always on bottom, the area is given by  $\int_{x=(1-\sqrt{5})/2}^{x=(1+\sqrt{5})/2} (x+1) - x^2 dx$ :

```
area = Integrate[ x + 1 - x^2, {x, corners[[1]], corners[[2]] }]
```

$$\frac{5\sqrt{5}}{6}$$

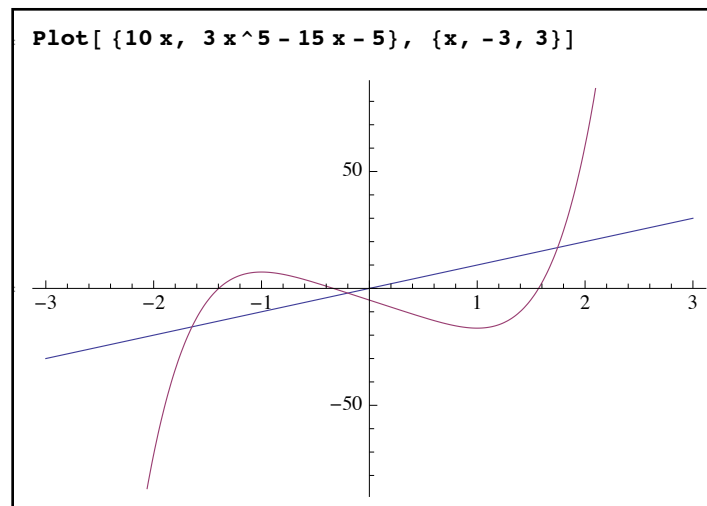
*the area between the curves*

So the area of the region between the curves is  $\frac{5\sqrt{5}}{6}$  square units.

Example 2: A more complicated area

Find the area between the curves  $y = 10x$  and  $y = 3x^5 - 15x - 5$ .

As before getting a view of the region before your proceed is always a good idea. So plotting both curves over a decent range we get:



*this area comes in 2 parts*

In this case it looks like there are 3 “corners” to the region and on the first part the quintic is on top and the line on bottom whereas on the second part the line is on top and the quintic is bottom. So we will need to use two integrals instead of just one. With more complicated curves like the quintic it is also possible that we didn’t graph a large enough region to see all the curve crossings - hopefully this will come out when we use Solve to find the corners:

```

corners = x /. Solve[ 10 x == 3 x^5 - 15 x - 5, x]

{Root[-5 - 25 #1 + 3 #1^5 &, 1], Root[-5 - 25 #1 + 3 #1^5 &, 2],
 Root[-5 - 25 #1 + 3 #1^5 &, 3], Root[-5 - 25 #1 + 3 #1^5 &, 4], Root[-5 - 25 #1 + 3 #1^5 &, 5]}

N[%]

{-1.64486, -0.200038, 1.74575, 0.0495763 - 1.70266 i, 0.0495763 + 1.70266 i}

```

*exact and approximate values for where the curves cross*

Here 3 of the values are real and two are non-real - this means the curves only cross in 3 points, and our graph shows the full region and did not mislead us. The numerical approximations also show us that the first part of the area goes from the first root to the second, and the second part of the area goes from the second root to the third. Setting up and evaluating the integrals gives us:

```

area = Integrate[ 3 x^5 - 15 x - 5 - 10 x, {x, corners[[1]], corners[[2]]}] +
Integrate[10 x - (3 x^5 - 15 x - 5), {x, corners[[2]], corners[[3]]}]

5 Root[-5 - 25 #1 + 3 #1^5 &, 1] +  $\frac{25}{2}$  Root[-5 - 25 #1 + 3 #1^5 &, 1]^2 -  $\frac{1}{2}$  Root[-5 - 25 #1 + 3 #1^5 &, 1]^6 -
10 Root[-5 - 25 #1 + 3 #1^5 &, 2] - 25 Root[-5 - 25 #1 + 3 #1^5 &, 2]^2 + Root[-5 - 25 #1 + 3 #1^5 &, 2]^6 +
5 Root[-5 - 25 #1 + 3 #1^5 &, 3] +  $\frac{25}{2}$  Root[-5 - 25 #1 + 3 #1^5 &, 3]^2 -  $\frac{1}{2}$  Root[-5 - 25 #1 + 3 #1^5 &, 3]^6

```

*an exact (if ugly) value for the area*

The Root objects give an exact form for the area which is not terribly helpful. Trying Simplify (and if that fails, N) to get a nicer answer is probably a good idea:

```

Simplify[%]

 $\frac{25}{6} \left( \text{Root}[-5 - 25 \#1 + 3 \#1^5 \&, 1] + 2 \text{Root}[-5 - 25 \#1 + 3 \#1^5 \&, 1]^2 - 2 \text{Root}[-5 - 25 \#1 + 3 \#1^5 \&, 2] - \right.$ 
 $\left. 4 \text{Root}[-5 - 25 \#1 + 3 \#1^5 \&, 2]^2 + \text{Root}[-5 - 25 \#1 + 3 \#1^5 \&, 3] + 2 \text{Root}[-5 - 25 \#1 + 3 \#1^5 \&, 3]^2 \right)$ 

N[%, 20]

49.363685358927166998

```

*Simplify fails, so getting a numerical approximation is the next best thing*

Simplify fails to break this down in any helpful way, so using N to get an approximation tells us the value for the area is about 49.363685 square units.

The presence of Root objects is fairly common as you progress to finding the area between more complicated algebraic curves. As you add more complicated curves to area problems you

may have to fall back on Reduce, NSolve, and FindRoot to estimate values for the corners of your regions.

After finding areas finding volumes of revolution is another common application of integrals.  $f(x) \geq g(x) \geq 0$ . If on the interval  $[a, b]$  then the volume of the region you get by spinning the region bounded by the two curves,  $x = a$ , and  $x = b$  around the  $x$ -axis is given by

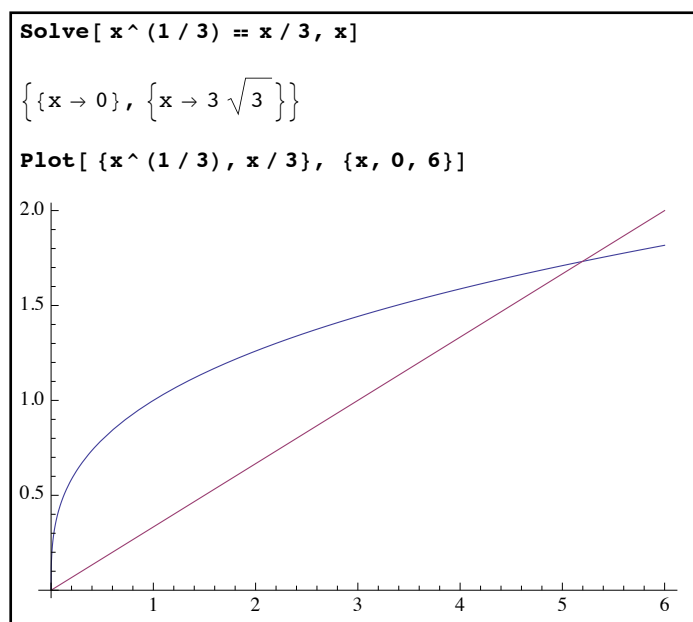
$$V = \pi \int_{x=a}^{x=b} f(x)^2 - g(x)^2 dx. \text{ If } a \geq 0 \text{ you can also find the volume of the region you get by}$$

rotating the same region around the  $y$ -axis by  $V = 2\pi \int_{x=a}^{x=b} x(f(x) - g(x)) dx$  (these are the methods of washers and cylindrical shells and can be adapted to other rotating regions about other axes).

### Example 3: Finding volumes of revolution

Find the volume of the solid you obtain by rotating the Quadrant I region bounded by  $y = \sqrt[3]{x}$  and  $y = \frac{x}{3}$  about the  $x$ -axis.

In this case we would use the method of washers where  $V = \pi \int_{x=a}^{x=b} f(x)^2 - g(x)^2 dx$ . We will need to identify where the two curves cross as well as which curve is on top (the “outer” radius really) and which curve is on bottom (the “inner” radius). Plotting the curves and using Solve we get:



*the region bounded by  $x^{1/3}$  and  $x/3$*

Since there are only two intersection points we know the region extends from  $x = 0$  to  $x = 3\sqrt{3}$  with the cube root always being on top and the line always being on bottom (which curve is on top or bottom can often be checked by Reduce). So setting  $f(x) = x^{1/3}$  and  $g(x) = \frac{x}{3}$  we get the

volume of the solid obtained by rotating around the x-axis is  $\pi \int_0^{3\sqrt{3}} (x^{1/3})^2 - (\frac{x}{3})^2 dx$ :

```
Pi Integrate[ (x^(1/3))^2 - (x/3)^2, {x, 0, 3 Sqrt[3]}]
```

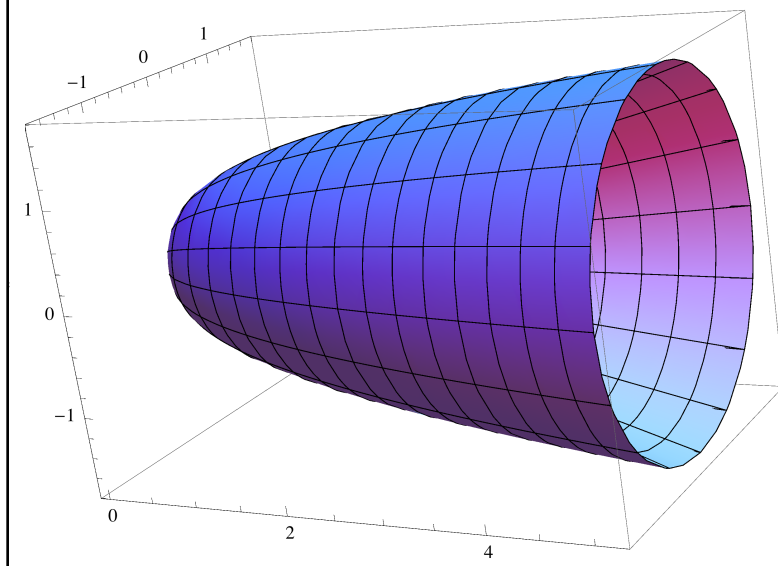
$$\frac{12 \sqrt{3} \pi}{5}$$

*the computation of the volume of revolution*

We can also see the volume through the use of a command called RevolutionPlot3D.

RevolutionPlot3D[ {f(x), g(x)}, {x,a,b}, RevolutionAxis→{d, e, f}] takes the parametric graph defined by {f(x), g(x)} from x=a to x=b and rotates it around the vector {d, e, f} (for revolution around the x-axis use {1,0,0} and for revolution around the y-axis use {0,0,1}). Since our solid is bounded by 2 curves we will need to use 2 commands and combine them with Show:

```
Show[RevolutionPlot3D[ {x, x^(1/3)}, {x, 0, 3 Sqrt[3]}, RevolutionAxis → {1, 0, 0}],  
RevolutionPlot3D[ {x, x/3}, {x, 0, 3 Sqrt[3]}, RevolutionAxis → {1, 0, 0}]]
```



*our volume of revolution*

This three-dimensional shape can be rotated within Mathematica by left-clicking and dragging across the image - this will let you see all of the sides of the solid.

If we had wanted to rotate the same region about the y-axis we would have had to use the method of cylindrical shells (as the axis of rotation is perpendicular to the independent axis and

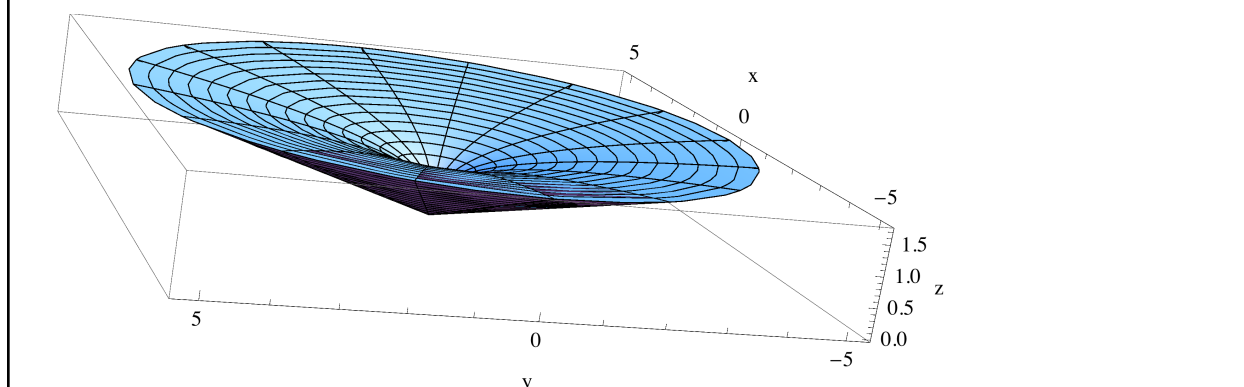
the region is to the right of the axis). As mentioned above the volume is then given by

$V = 2\pi \int_{x=a}^{x=b} x(f(x) - g(x)) dx$ , where  $f(x)$  is the top curve and  $g(x)$  is the bottom curve. For our region, we would have:

$$2 \text{ Pi Integrate} [x (x^{(1/3)} - x/3), \{x, 0, 3 \text{ Sqrt}[3]\}]$$

$$\frac{36 \sqrt{3} \pi}{7}$$

```
Show[RevolutionPlot3D[{x, x^(1/3)}, {x, 0, 3 Sqrt[3]}, RevolutionAxis -> {0, 0, 1}],
RevolutionPlot3D[{x, x/3}, {x, 0, 3 Sqrt[3]}, RevolutionAxis -> {0, 0, 1}],
AxesLabel -> {"x", "y", "z"}]
```



*the volume and solid when you rotate around the y-axis*

It's important to remember that in general if you take the same region  $R$  and rotate it around the  $x$ -axis and  $y$ -axis you will most likely get very different shapes, each with their own volume.

Example 4: Arc length

Find the arc length of  $y = x^2$  from 2 to 3.

The formula for the arc length of a curve  $y = f(x)$  from  $x = a$  to  $x = b$  is  $\int_{x=a}^{x=b} \sqrt{1 + f'(x)^2} dx$ . Putting this into Mathematica gives a simple computation:

$$\text{Integrate} [\text{Sqrt} [1 + \text{D} [x^2, x]^2], \{x, 2, 3\}]$$

$$\frac{1}{4} \left( -4 \sqrt{17} + 6 \sqrt{37} - \text{ArcSinh}[4] + \text{ArcSinh}[6] \right)$$

$$\text{N}[\%, 20]$$

$$5.1003049961756216527$$

*the arc length along a parabola from (2,4) to (3,9)*

This may seem fairly simple and straightforward - and in the sense of the general arc length formula it is. However, as you look at functions  $f(x)$  that are even slightly complicated it becomes very likely that the integral will require all sorts of special functions and may have to be numerically estimated. For example suppose we changed our function from  $x^2$  to  $x^3$ , which is not exactly a huge leap in the difficulty of the curve. Our arc length entry into Mathematica is almost exactly the same, but the result takes a huge step up in terms of complexity:

```
Integrate[ Sqrt[ 1 + D[ x^3, x ] ^2 ], {x, 2, 3} ]
```

$$3 \operatorname{Hypergeometric2F1}\left[-\frac{1}{2}, \frac{1}{4}, \frac{5}{4}, -729\right] - 2 \operatorname{Hypergeometric2F1}\left[-\frac{1}{2}, \frac{1}{4}, \frac{5}{4}, -144\right]$$

```
N[% , 20]
```

19.027752707185532760

*time to hit the Mathematica documentation to learn about Hypergeometric2F1*

So this simple calculation gives you a result in terms of hypergeometric series - a topic which is not usually covered until advanced classes in analysis or numerical methods. This is very common with arc length calculations (and with its sister calculation surface area as well) and will usually require you to find a numerical estimate for most applications.

#### Example 5: Surface Area

Find the area of the surface you get by rotating  $y = x^2$  around the  $x$ -axis on the interval  $[1,2]$ .

Going to back to most calculus texts you can find the following formula for surface area: If  $f(x) \geq 0$  on  $[a,b]$ , the area of the surface you get by rotating the curve around the  $x$ -axis is given

by  $2\pi \int_{x=a}^{x=b} f(x) \sqrt{1+f'(x)^2} dx$ . Since this is exactly the case we have here we can put it into

Mathematica and use RevolutionPlot3D to see the surface:

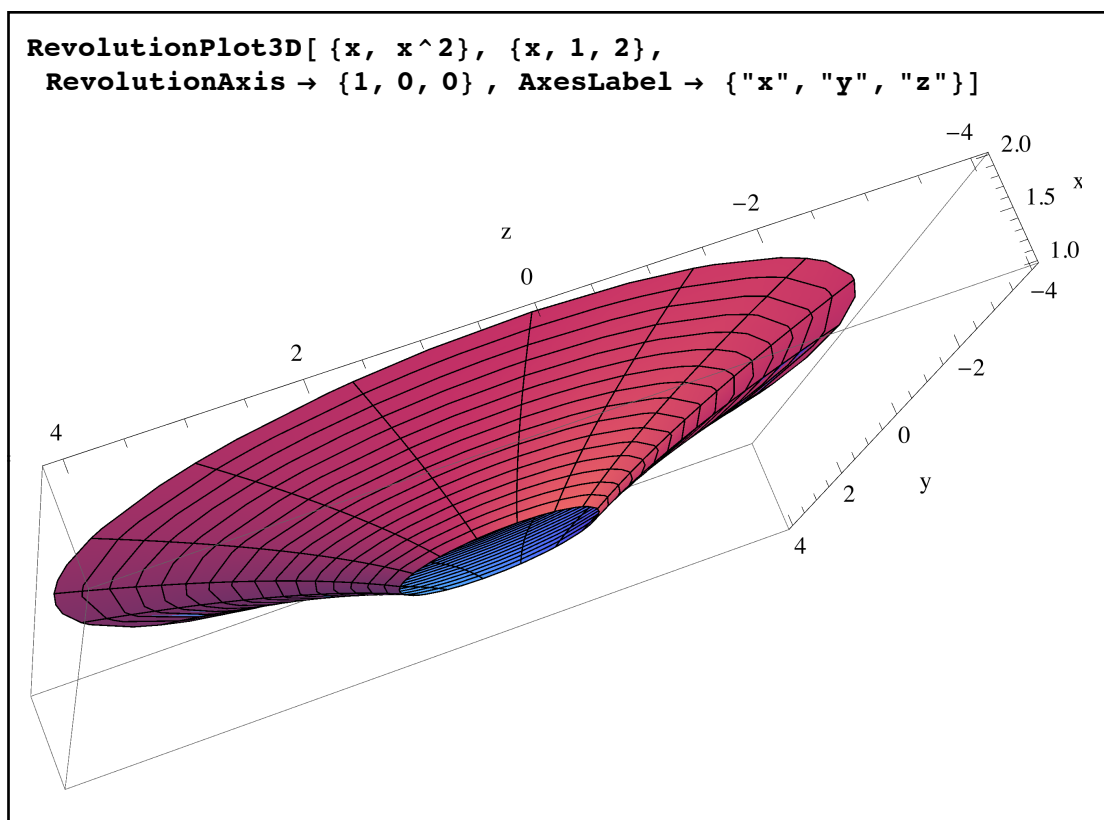
```
2 Pi Integrate[ x^2 Sqrt[ 1 + D[x^2, x ] ^2 ], {x, 1, 2} ]
```

$$\frac{1}{32} \pi \left( -18 \sqrt{5} + 132 \sqrt{17} + \operatorname{ArcSinh}[2] - \operatorname{ArcSinh}[4] \right)$$

```
N[% , 20]
```

49.416235538296800989

*the area of the surface*



*the actual surface, rotated to give a good view and not take up the whole page*

Like arc length problems surface area problems often lead to integrals which are difficult if not impossible to do - for example, if we replace  $x^2$  with  $\ln(x)$  in the example above the integral involves the HypergeometricPFQ function and would need to be estimated in most applications.

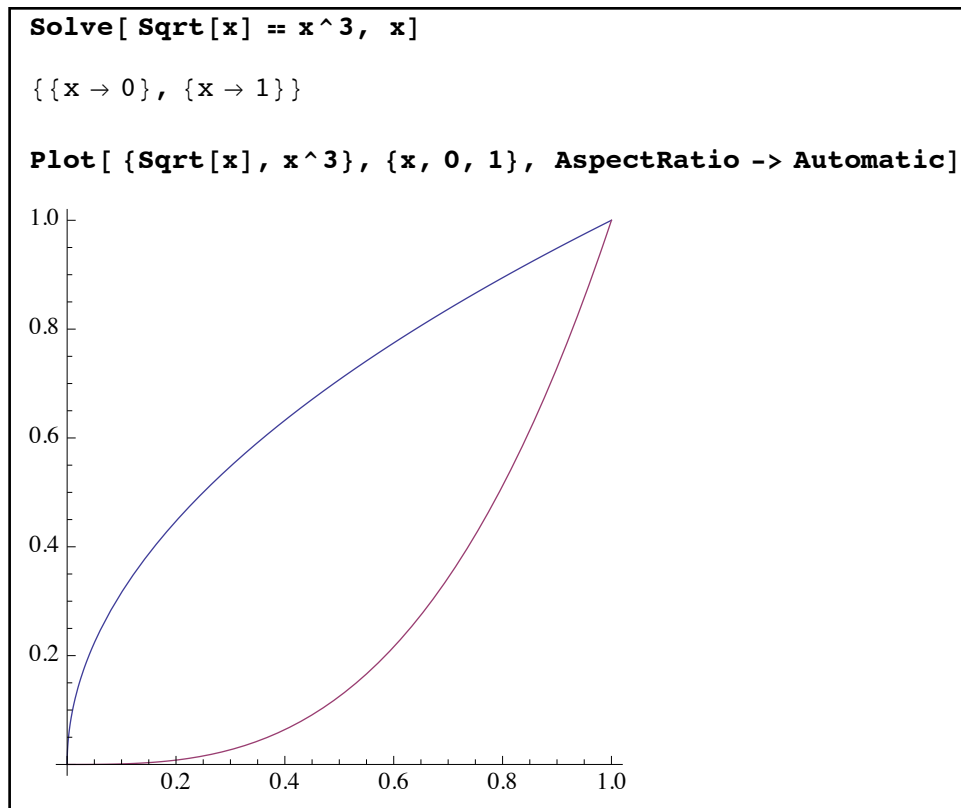
#### Example 6: Center of Mass

Find the center of mass (the balancing point, so metes called the centroid) for the Quadrant I region bounded by  $y = \sqrt{x}$  and  $y = x^3$ .

If a region is bounded on top by  $y = f(x)$ , on bottom by  $y = g(x)$ , on the left by  $x = h(y)$ , and on the right by  $x = j(y)$  where  $x$  is in  $[a, b]$  and  $y$  is in  $[c, d]$ , then the coordinates for the centers of mass of the region are  $\bar{x} = \frac{\int_{x=a}^{x=b} x(f(x) - g(x)) dx}{A}$ ,  $\bar{y} = \frac{\int_{y=c}^{y=d} y(j(y) - h(y)) dy}{A}$ , where  $A$  is the area of the region and usually given by  $A = \int_{x=a}^{x=b} f(x) - g(x) dx$ . The trick in using these formulas (other than remembering them!) is typically in finding the formulas for the bounding curves and remembering which variable to integrate with respect to.



For our region, first we need to identify the boundary curves and where the region starts and stops:



*the endpoints and plot of our region*

It appears that when looking up and down,  $y = \sqrt{x}$  is on top and  $y = x^3$  is on bottom (from the view of the  $x$ -axis). When looking left and right,  $y = x^3$  is rightmost (the “top” from the view of the  $y$ -axis) and  $y = \sqrt{x}$  is leftmost (the “bottom” from the view of the  $x$ -axis). In terms of both  $x$  and  $y$  the range of values for our region will be  $[0,1]$ . From the way the region is shaped we would probably guess the center of mass is probably a little less than halfway across the region and below the  $45^\circ$  line.

Both parts of the center of mass formula use the area of the region, so we may as well find that first:

```
area = Integrate[ Sqrt[x] - x^3, {x, 0, 1}]
```

$$\frac{5}{12}$$

*the area of the region, based on which curve is on top and which is on bottom*

To find the  $x$ -coordinate of the center of mass we use the formula  $\bar{x} = \frac{\int_{x=a}^{x=b} x(f(x) - g(x)) dx}{A}$ .

Both curves  $y = x^3$  and  $y = \sqrt{x}$  are written as functions of  $x$  and  $x$  goes from 0 to 1, so we may go right to the computation:

```
xcoordinate = Integrate[ x (Sqrt[x] - x^3) , {x, 0, 1} ] / area
12
-----
25
```

*finding the x-coordinate for the center of mass*

To find the  $y$ -coordinate, we use the formula  $\bar{y} = \frac{\int_{y=c}^{y=d} y(j(y) - h(y)) dy}{A}$ . We know that

$y = x^3$  is the rightmost curve (corresponding to  $j(y)$ ) and  $y = \sqrt{x}$  is the leftmost curve (corresponding to  $h(y)$ ). But to use the formula these curves need to be expressed as a function of  $y$ , not  $x$ . Solving both for  $x$  is a simple matter;  $y = x^3$  becomes  $x = \sqrt[3]{y}$  and  $y = \sqrt{x}$  becomes  $x = y^2$ . So entering these into Mathematica we get:

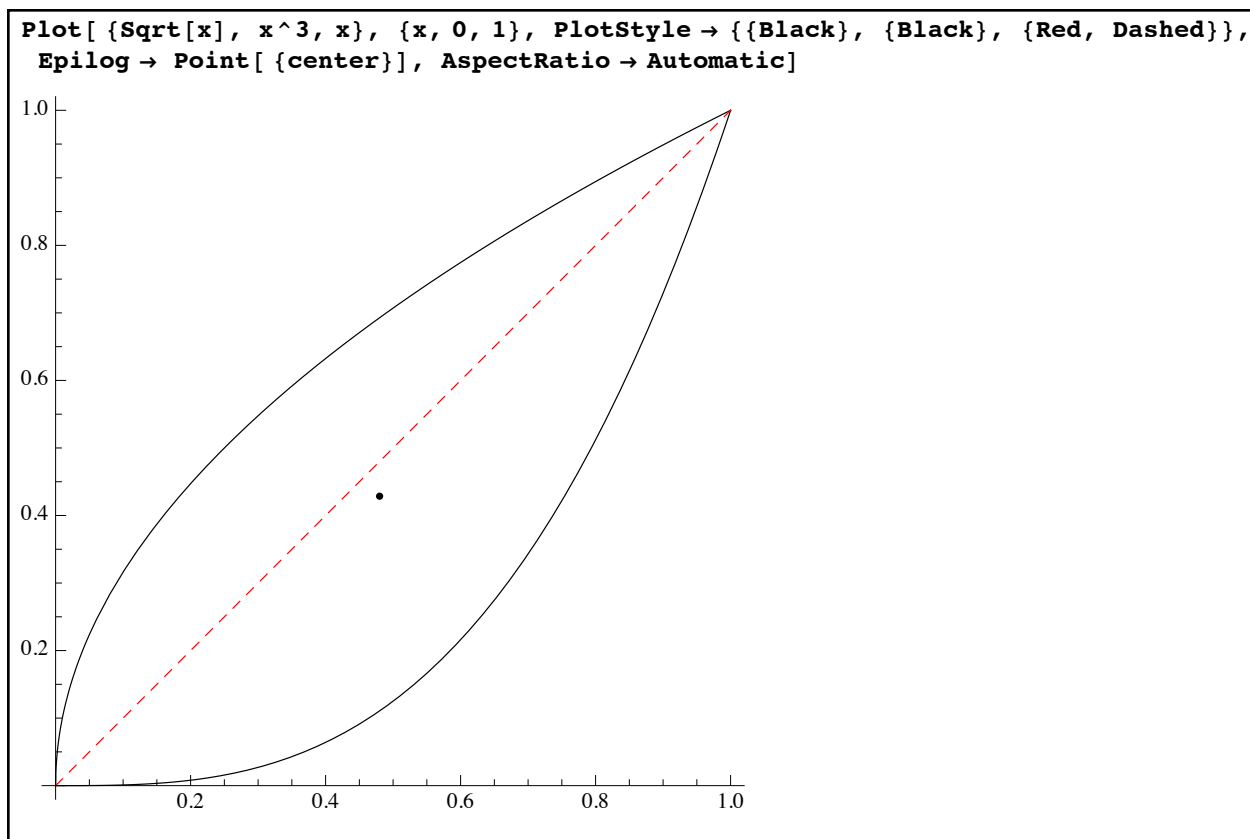
```
ycoordinate = Integrate[ y (y^(1 / 3) - y^2) , {y, 0, 1} ] / area
3
-----
7

center = {12 / 25, 3 / 7}

{ 12  3 }
{ --, -- }
{ 25  7 }
```

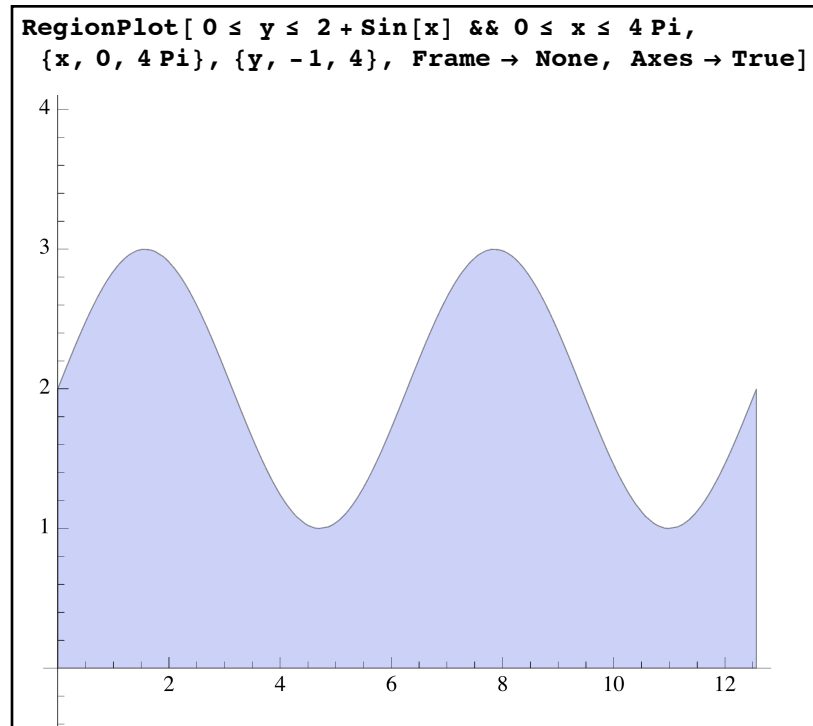
*the y-coordinate and the full center of mass*

We can check the initial guess that the center of mass would be less than halfway across the region and below the  $45^\circ$  line (i.e. below  $y=x$ ) by using Plot together with PlotStyle and Epilog:



*visualizing the center of mass*

If many of the applications of the integral we have seen so far seem rather “plug-n-chug” there is a good reason for that. When you first learn to do area, volume, and center of mass problems there tend to be two difficult parts. One hard part is in computing the integral once it has been set up - integrals can be extremely difficult from a computational perspective (which is why a great deal of time in second semester calculus is devoted to integration techniques). We are using Mathematica to do the integration though - which makes the computational part easy. The other difficult part is in setting up the integral, which usually boils down to finding which curves bound the region over different intervals (i.e. finding the “top”, “bottom”, “left”, and “right” curves for a region or the basic components of a more complicated region). The regions that we have looked at so far have been fairly simple - we haven’t had to worry too much about curves switching relative positions or the difficulties inherent in going from a curve of the form  $y = f(x)$  to one of the form  $x = g(y)$ . Plot/RegionPlot, Solve, and Reduce are very useful commands for help in sorting these problems out if they come up. We have seen simple examples of how Plot and Solve can be used to find where curves cross. Reduce is very helpful in determining which curves are on top, bottom, left, or right. For example, suppose we want to look at the region bounded above by  $y = 2 + \sin(x)$  and below by  $y = 0$  as  $x$  goes from 0 to  $4\pi$ . We can graph the region first to get an idea of what is going on; after playing with RegionPlot a bit to get the ranges right for  $x$  and  $y$  we see:



*our region with RegionPlot*

In this case it looks like there's a perfectly good top curve for the whole region ( $y = 2 + \sin(x)$ ) and bottom curve for the whole region ( $y = 0$ ). But for left and right bounding curves this region is very messy. If you are standing on the  $y$ -axis looking out over the region there are different curves which bound different parts of the region. The "bottom" of the region looks like a rectangle, but portions higher up are bounded on the left and right sides by different portions of the sine curve. Reduce can help you break this down:

```
Reduce[0 ≤ x ≤ 4 Pi && 0 ≤ y ≤ 2 + Sin[x], {x, y}]
```

```
0 ≤ x ≤ 4 π && 0 ≤ y ≤ 2 + Sin[x]
```

```
Reduce[0 ≤ x ≤ 4 Pi && 0 ≤ y ≤ 2 + Sin[x], {y, x}]
```

```
(0 ≤ y ≤ 1 && 0 ≤ x ≤ 4 π) || (1 < y ≤ 2 && 0 ≤ x ≤ π + ArcSin[2 - y]) ||
```

```
(2 < y < 3 && -ArcSin[2 - y] ≤ x ≤ π + ArcSin[2 - y]) ||
```

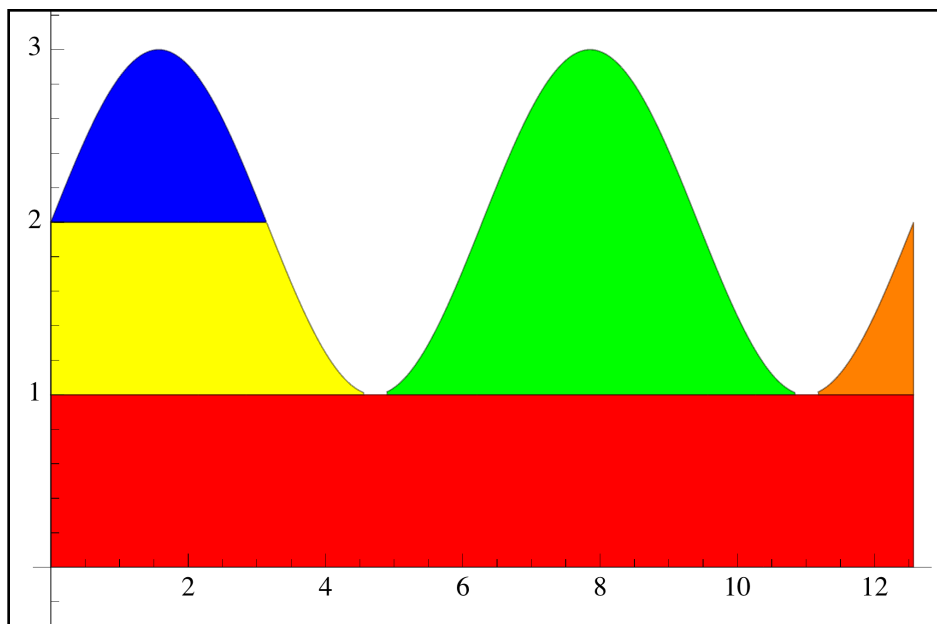
```
(1 < y < 3 && 2 π - ArcSin[2 - y] ≤ x ≤ 3 π + ArcSin[2 - y]) ||
```

```
(1 < y < 2 && 4 π - ArcSin[2 - y] ≤ x ≤ 4 π) || (y = 2 && x = 4 π) || (y = 3 && (x == π/2 || x == 5 π/2))
```

*two different orders in the Reduce command*

The first Reduce command tells us what we already knew. But in the second command the order of  $x$  and  $y$  have been switched - this tells Mathematica to think of  $y$  as being independent and  $x$  as being dependent (essentially switching from a top-bottom view to a left-right view). There are 5 main parts to this Reduce (not including the 2 isolated points at the very end). Each part

corresponds to a subregion with a well-defined “left” curve and “right curve” as well as  $y$ -values where the subregions start and stop. You can see the parts by using RegionPlot to re-graph each one in a different PlotStyle:



*the subregions of our region*

The subregions are:

- $0 \leq y \leq 1$  &  $0 \leq x \leq 4\pi$  in red
- $1 \leq y \leq 2$  &  $0 \leq x \leq \pi + \text{ArcSin}[2 - y]$  in yellow
- $2 \leq y \leq 3$  &  $-\text{ArcSin}[2 - y] \leq x \leq \pi + \text{ArcSin}[2 - y]$  in blue
- $1 \leq y \leq 3$  &  $2\pi - \text{ArcSin}[2 - y] \leq x \leq 3\pi + \text{ArcSin}[2 - y]$  in green
- $1 \leq y \leq 2$  &  $4\pi - \text{ArcSin}[2 - y] \leq x \leq 4\pi$  in orange

As ugly as this is this is what would be necessary to do in order to find the  $y$ -coordinate of the center of mass (you would replace the integral in the numerator with the sum of 5 integrals, one for each piece). Reduce won't always be able to break up regions as nicely of this of course but when it does it is a huge help.

For our last application we will look at the energy needed to lift a satellite. The work done by a variable force (one that depends only on position  $x$ ) from position  $x = a$  to  $x = b$  is given

by  $W = \int_a^b f(x) dx$ .

Example 8: Work done by a variable force

According to Newton's theory of gravity the gravitational force exerted between objects is equal to a universal constant (usually written as  $G$ ) times the product of their masses and divided by the square of the distance between their centers (this is in the case of small objects or large

spherical objects). Ignoring air resistance, how much energy is needed to lift a 3000 kilogram satellite to a height of 10,000 kilometers above the surface of the earth? How much energy would you need to impart to the satellite to make it so that it doesn't return to Earth (i.e., goes out to infinity)?

We know that the force of gravity at a height of  $x$  above the center of the earth is given by

$F(x) = \frac{G m_{\text{satellite}} m_{\text{earth}}}{r^2}$ , where  $G$  is the universal constant. The values for  $x$  would start at how

far above the center of the Earth the ground is (the Earth's radius) and would end either 10,000 kilometers =  $10^7$  meters above that height or go all the way to infinity. The mass of the satellite is 3000 kilograms, but we will still need the values for  $G$ , the mass of the earth, and the radius of the earth. Looking these up online we can see the mass of the earth is about  $5.9722 \times 10^{24}$  kilograms, the radius of the earth is about  $6.3674 \times 10^6$  meters, and the gravity constant is  $6.67 \times 10^{-11} \text{ Nm}^2/\text{kg}^2$  (a slick way to do this is to look up each quantity in a WolframAlpha cell, click on the "+" in the result window, and then select "Number data" - this will paste the numerical quantity into Mathematica for you):

```
gravconstant = 6.67 * 10^-11;
earthmass = 5.9722 * 10^24;
earthradius = 6.2674 * 10^6;

force = gravconstant * 3000 * earthmass / x^2

1.19504 × 1018
-----
      x2

Integrate[ force, {x, earthradius, earthradius + 10^7}]

1.17213 × 1011
```

*energy needed to lift a satellite*

Note that the unit for energy in the metric system is a joule (a watt is 1 joule per second). It is possible to have Mathematica automatically track units for you through the Quantity command. Quantity[ *number*, *unit* ] represents a quantity with an inherent unit to it (like "Feet", "Meters", or "Kilograms"). Although we will defer a full discussion of Quantity and units until later here is how they would be used in this problem to automatically track and process the units:

```

gravconstant = Quantity[6.67 * 10^-11, "Newtons" * "Meters"^2 / ("Kilograms"^2)];
earthmass = Quantity[5.9722 * 10^24, "Kilograms"];
earthradius = Quantity[6.2674 * 10^6, "Meters"];

force = gravconstant * Quantity[3000, "Kilograms"] * earthmass / x^2

1.19504 × 1018 m2N
      x2

Integrate[ force, {x, earthradius, earthradius + Quantity[10^7, "Meters"]} ]

1.17213 × 1011 mN

UnitSimplify[%]

1.17213 × 1011 J

```

*automatic unit tracking and conversion with Quantity and UnitSimplify*

## Section 5.5 Homework - Applications of the Integral

- 1) Find the area enclosed by the curves  $y = x^2$  and  $y = 3 - x$ .
- 2) Find the area enclosed by the curves  $y = \sin^{-1}(x)$  and  $y = \tan^{-1}(x)$  as  $x$  goes from 0 to 1.
- 3) Find the volume you get by rotating the region bounded by  $y = \sin^2(x)$  and the  $x$ -axis from  $x = 0$  to  $x = \pi$  about the  $x$ -axis
- 4) Repeat problem 3, but rotate it about the  $y$ -axis.
- 5) Find the volume of a pyramid with square base  $b$  and height  $h$ . (although not discussed the volume of a shape is given by the integral of its cross-sectional area function - to make this easy, put the pyramid's apex at the origin and have it open around the  $x$ -axis. That way its cross sections over a given value  $x$  will always be a square).
- 6) Find the volume you get by rotating the region bounded by  $y = 1$ ,  $x = 0$ , and  $y = \sin(x)$  around the  $x$ -axis.
- 7) Find the arc length of  $y = 3x - 2$  from  $x = 1$  to  $x = 7$ . Does this make sense?
- 8) Find the arc length of  $y = x^4$  from  $x = 0$  to  $x = 1$ , both exactly and as a numerical estimate. Find the surface area you would get if you rotated that curve around the  $x$ -axis.
- 9) Repeat problem 8 with the curve  $y = \sin(x)$  from  $x = 0$  to  $x = 2\pi$ .
- 10) A certain spring requires a 600 Newton force to stretch it 1/10 of a meter. How much work is done in stretching from 20 cm to 30 cm, measured from rest position? (Recall that Hooke's law for a spring says that for reasonable stretches  $x$ , the force exerted by a spring is given by  $F = kx$  for some constant  $k$ ).
- 11) How much work would it take to lift the earth from the surface of the sun to its present orbit? (find the orbital radius of the earth in meters online).
- 12) Find the center of mass for the Quadrant I region bounded by  $y = x$  and  $y = x^4$ .
- 13) Find the center of mass for the Quadrant I region bounded by  $y = x^4$  and  $y = x^5$ . Graph the region and the center of mass. Notice anything unusual?

- 14) Use Reduce to break down the region bounded by  $y = x^2$  and  $y = 5x - 6$  into subregions with nice top, bottom, left, and right boundaries. Create a composite graph as we did for the region involving  $y = 2 + \sin(x)$ .



## Section 5.6 - Geometric Computation and Regions

Mathematica 10 introduced a new type of object and related calculations - region objects. Regions represent portions of one-, two-, three-, and even higher dimensional space and both have properties of their own and can hook into other Mathematica commands like Solve and Reduce.

The regions themselves fall into 3 categories - specific pre-defined ones, ones that are defined implicitly by equations and inequalities, and “derived” regions - ones that are defined by combinations of other regions. Regions in two dimensions can be graphed using RegionPlot, either in the form RegionPlot[*region*] or RegionPlot[ {*region1*, *region2*, ...} ]. When using RegionPlot it is common to have to increase the number of points used in the graphs (via PlotPoints) and sometimes necessary to use PlotRange→All to see the full region. Regions in 3 dimensions can be viewed with the similar command RegionPlot3D.

To start off let's take a look at some simple pre-defined regions. Some of these we have seen before and many of the can be used with the Epilog option in graphs:

Interval[{*a*,*b*}]: Interval[{*a*,*b*}] represents the one-dimensional region from *a* to *b*.

Circle[{*h*,*k*}, *r*]: This represents the circle centered at the point (*h*,*k*) with radius *r*.

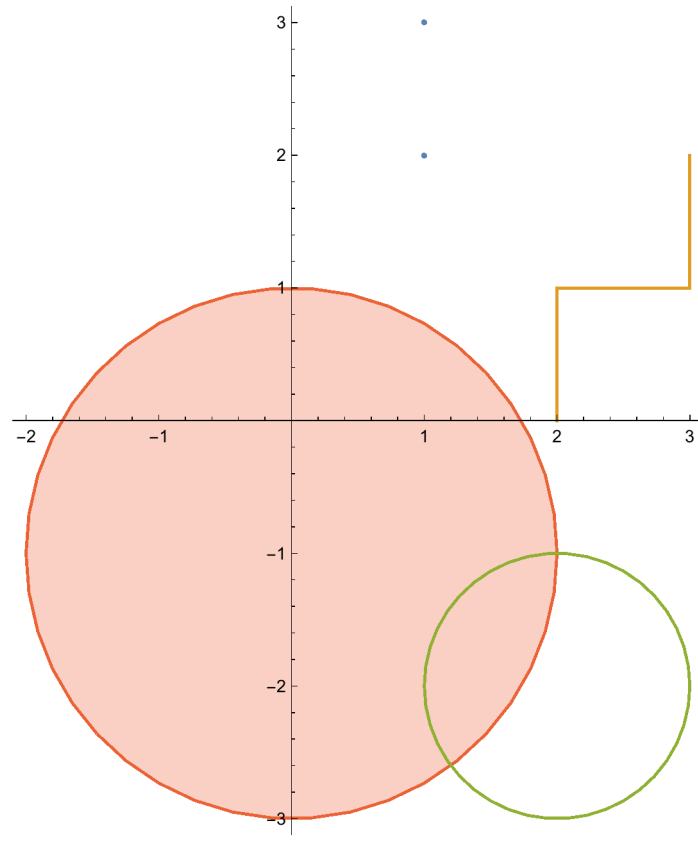
Disk[{*h*,*k*}, *r*]: This represents the solid disk centered at the point (*h*,*k*) with radius *r*.

Point[*coordinates*]: Point represents a point in any number of dimensions (so Point[{1,2}] is a point in the plane and Point[{1,2,3}] is a point in space. Point[*list of coordinate tuples*] represents several points (so Point[{ {1,2,3}, {2,2,2} }] is a region of 2 points in space).

Line[*list of coordinate tuples*]: Line represents a sequence of line segments starting at the first point in the list and going through each successive point. Note that this is not a full line in the usual sense. If the first point in the list is the same as the last point in the list the region will be a closed loop. The segments can be 2, 3, or more dimensions as given by the length of the *tuples*.

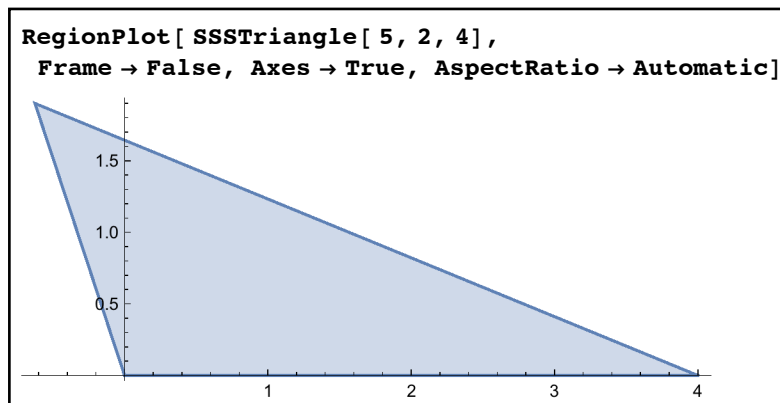
InfiniteLine[ {*point1*, *point2* } ]: InfiniteLine is the full line through the two points. InfiniteLine represents what we commonly think of as a geometric line.

```
RegionPlot[ {Point[ {{1, 2}, {1, 3}}],
  Line[ {{2, 0}, {2, 1}, {3, 1}, {3, 2}}], Circle[ {2, -2}, 1], Disk[ {0, -1}, 2]],
PlotRange → All, AspectRatio → Automatic, Frame → False, Axes → True]
```



*several geometric objects shown by RegionPlot*

`SSSTriangle[ $a, b, c$ ]`: `SSSTriangle` represents a triangle whose points are A (the origin), B (on the positive  $x$ -axis), and C (in the upper half-plane) with corresponding side lengths  $a$ ,  $b$ , and  $c$ .



*a triangle with side lengths 5, 2, and 4*

SASTriangle[*a,angle,b*]: SASTriangle defines a triangle ABC where A is at the origin, B is on the positive *x*-axis, C is in the upper half plane, and the angle ACB has a measure *angle*.

ASATriangle[*angle1,b,angle2*]: ASATriangle defines a triangle ABC where A is at the origin, B is on the positive *x*-axis, C is in the upper half plane, the angle CAB has measure *angle1*, and the angle ACB has measure *angle2*.

AASTriangle[ *angle1, angle2,c*]: AASTriangle defines a triangle ABC where A is at the origin, B is on the positive *x*-axis, C is in the upper half plane, the angle CAB has measure *angle1*, and the angle CBA has measure *angle2*.

Triangle[{*point1, point2, point3*}]: This represents the triangle whose corners are the given points. The points can be in any number of dimensions provided they all match (so Triangle[ { {1,2,3,4},{5,6,7,8},{1,3,5,7} } ] represents a triangle in 4-dimensional space).

Rectangle[{*point1, point2*}]: This is rectangle in two dimensions whose opposite corners are the given points.

Polygon[*list of points*]: This represents the filled-in polygon whose vertices are given by the list of points. The polygon can be in any number of dimensions provided the points in the list all have the same number of coordinates.

Sphere[*point,r*]: This is a spherical shell centered at the *point* with radius *r*. If the point is two-dimensional this is the same as a circle; in general the sphere will be in as many dimensions as the *point* which gives its center.

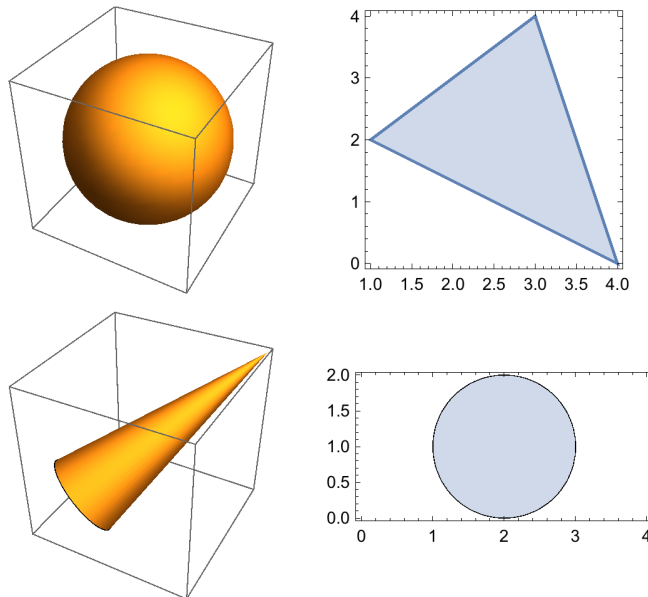
Circumsphere[*list of (n+1) points in n-dimensions*]: This gives the *n*-dimensional spherical shell through the given points. So 3 two-dimensional points define a traditional circle, 4 three-dimensional points define a traditional sphere, etc.

Ball[*point, r*]: The solid ball centered at *point* with radius *r*. If the point is 2-dimensional this is the same as Disk[*radius, r*]. The difference between Ball and Sphere is that the ball is solid and the sphere is hollow.

Cylinder[{*point1, point2*}, *r*]: This is the solid cylinder whose axis extends from *point1* to *point2* and whose radius is *r*. In 2 dimensions this would be a rectangle.

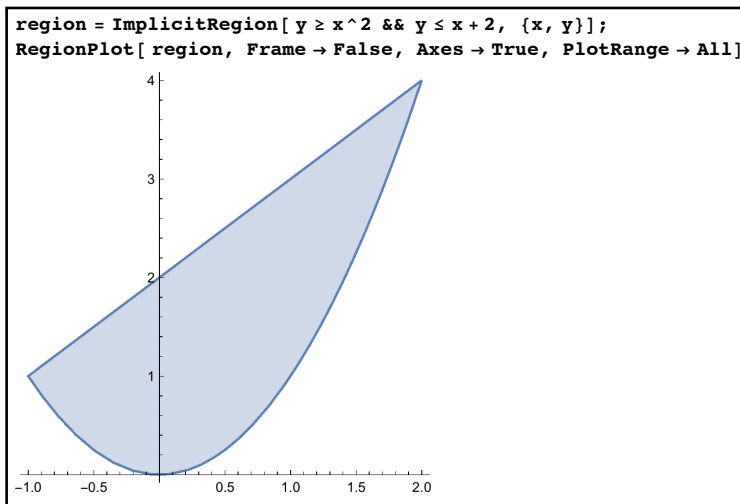
Cone[{*point1, point2*},*r*]: The solid cone whose axis goes from *point1* to *point2* with radius *r*. The tip of the cone is *point2*. In two dimensions this is a triangle.

```
GraphicsGrid[{{RegionPlot3D[ Sphere[{1, 1, 1}, 2]],
  RegionPlot[ Triangle[{{1, 2}, {3, 4}, {4, 0}}]]},
{RegionPlot3D[ Cone[{{0, 0, 0}, {3, 3, 3}}, 1]],
  RegionPlot[ Ball[{2, 1}, 1], AspectRatio -> Automatic]}}
```



*several different regions shown via GraphicsGrid, RegionPlot, and RegionPlot3D*

One of the most important and general kinds of regions are those defined by a system of equations and inequalities. These are defined by the `ImplicitRegion` command in the form `ImplicitRegion[ logical statement, list of variables ]` which represent those points for which the *logical statement* is True. So if we wanted those points which are on or above  $y = x^2$  but on or below  $y = x + 2$  we could use `ImplicitRegion[  $y \geq x^2 \ \&\& \ y \leq x + 2$ , {x,y} ]` and the ball of radius 2 centered at (1,3,4) could be `ImplicitRegion[ EuclideanDistance[ {x,y,z}, {1,3,4} ]  $\leq 2$ , {x,y,z} ]`.

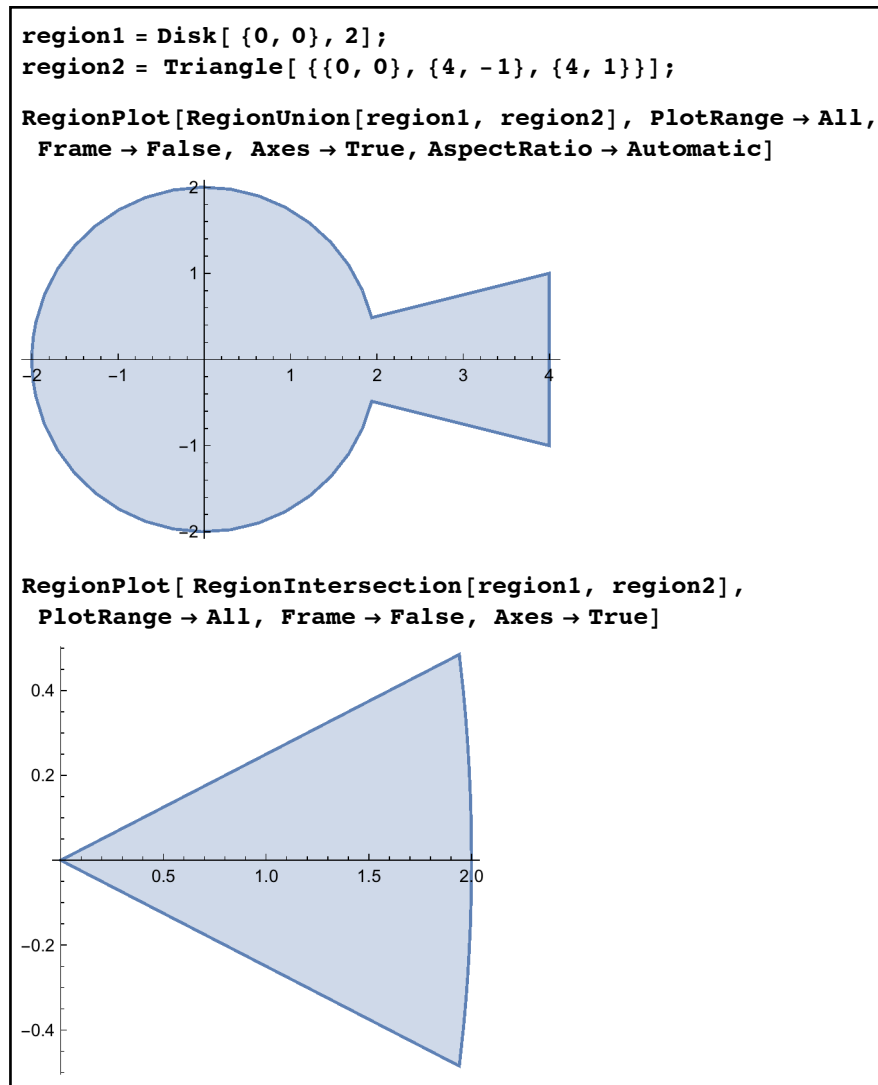


*defining a two-dimensional region implicitly*

Once you have some basic regions defined you can also define new regions in terms of these earlier ones:

`RegionUnion[region1, region2...]`: `RegionUnion` represents all of the regions put together.

`RegionIntersection[region1, region2...]`: `RegionIntersection` represents those points which are common to all of the regions.



*the union and intersection of a disk and triangle*

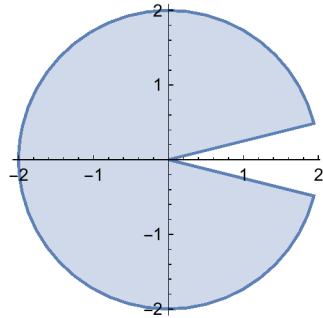
`RegionDifference[region1, region2]`: `RegionDifference` defines the region which includes all of the points of *region1* which are not part of *region2*. As `RegionPlot` by default always includes the boundaries of a region, the boundaries may included when they are not technically part of the resulting set.

```

region1 = Disk[{0, 0}, 2];
region2 = Triangle[{{0, 0}, {4, -1}, {4, 1}}];

RegionPlot[
  RegionDifference[region1, region2], PlotRange -> All,
  Frame -> False, Axes -> True, AspectRatio -> Automatic]

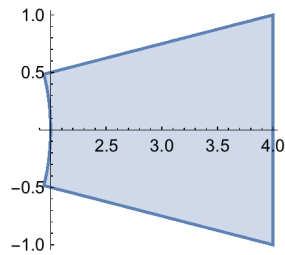
```



```

RegionPlot[RegionDifference[region2, region1],
  PlotRange -> All, Frame -> False, Axes -> True]

```



*graphing the difference of two regions - note that the order makes a big difference*

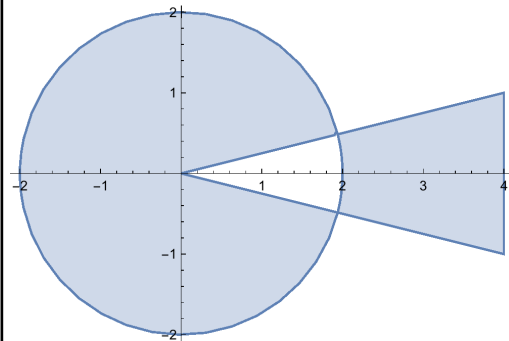
`RegionSymmetricDifference[region1, region2]`: The region of points which are in exactly 1 of the two regions.

```

region1 = Disk[{0, 0}, 2];
region2 = Triangle[{{0, 0}, {4, -1}, {4, 1}}];

RegionPlot[
  RegionSymmetricDifference[region1, region2], PlotRange -> All,
  Frame -> False, Axes -> True, AspectRatio -> Automatic, PlotPoints -> 100]

```



*the symmetric difference of two regions*

`RegionBoundary[region]`: `RegionBoundary` defines the “boundary”, or “edge” of *region*. When graphing a boundary you may need to increase the number of `PlotPoints` to get it to show correctly. Typically the boundary of a region is 1 dimension lower than that of the original region.

Now that we can create all sorts of regions there are many useful computations Mathematica can do with them. Many of these require would require calculus to do on paper (and can provide a check on some of the answers in the previous section).

`ArcLength[region]`: The length of the one-dimensional *region*.

`Area[region]`: The area of the two-dimensional *region*.

`Volume[region]`: The volume of the three-dimensional *region*.

`RegionMeasure[region]`: The *n*-dimensional measure of *region*, where *region* is itself *n*-dimensional. `RegionMeasure` generalizes `ArcLength`, `Area`, and `Volume`.

`RegionDimension[region]`: The dimension of *region*.

`RegionEmbeddingDimension[region]`: The dimension of the space that *region* is part of. So for example if *region1* was a disk in three-dimensional space (say given by `ImplicitRegion[x^2+y^2 ≤ 1 && z==1, {x,y,z}]`) then `RegionDimension[region1]` would be 2 (as a disk is 2D) but `RegionEmbeddingDimension[region1]` would be 3 (as the disk consists of points in 3D).

`RegionBounds[region]`: `RegionBounds` tries to give the full `PlotRange` for the *region*. If the *region* is not bounded (say it is a parabola) then  $\infty$  and  $-\infty$  may be part of the lists given by `RegionBounds`.

`RegionCentroid[region]`: The “center point”/“center of mass” of *region*. For a two-dimensional region this would be the point at which the region would balance if supported from below.

`RegionDistance[region, point]`: `RegionDistance` is the minimum distance from all of the points in *region* to the given *point*. `RegionDistance[region]` creates a standalone function which measures the distance from *region* to any point  $\{x,y\}$ . If you want to use such a function it is best to store it in a name using `=` rather than `:=` so the distance function is computed just once rather than from scratch each time.

`RegionNearest[region, point]`: `RegionNearest` computes the point of the *region* which is nearest the given *point*. `RegionNearest[region]` creates a standalone function which measures the distance from *region* to any point  $(x,y)$ . As in `RegionDistance` if you

want to use such a function it is best to store it in a name using `=` rather than `:=` so the distance function is computed just once rather than from scratch each time.

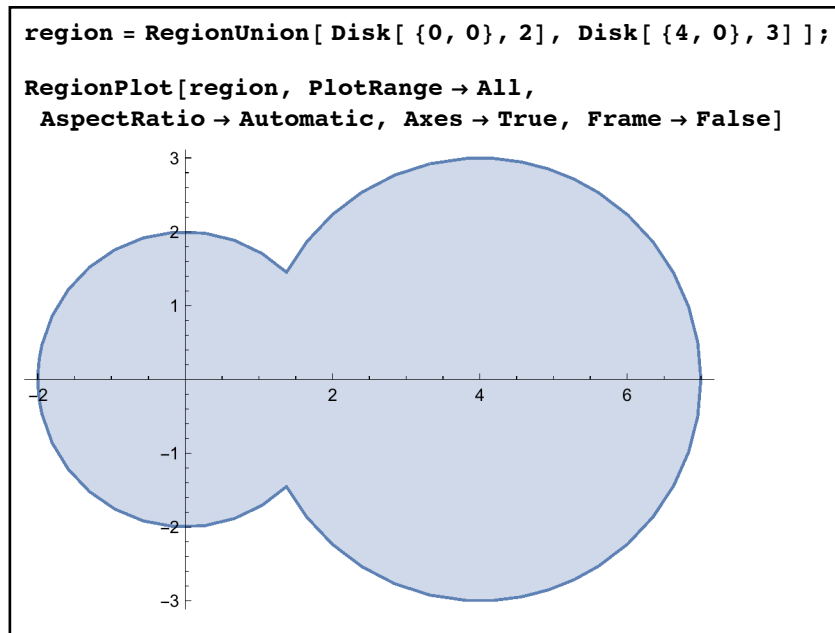
`RegionMember[region, point]`: `RegionMember` returns `True` if *point* is part of *region* and `False` otherwise. `RegionMember[region]` creates a standalone function which returns `True` or `False` depending on if a point  $\{x,y\}$  is in the region. If you want to use such a function it is best to define it using `=` rather than `:=` so the distance function is computed just once rather than from scratch each time.

### Example 1: The union of two disks

Let  $R$  be the union of the disk centered at  $(0,0)$  of radius 2 and the disk centered at  $(4,0)$  of radius 3. Find the following:

- The graph of  $R$
- The area of  $R$
- The length of  $R$ 's boundary
- The centroid of  $R$ .
- The minimum distance from  $(2,3)$  to  $R$
- The point of  $R$  that is closest to  $(2,3)$
- The graph of all points 1 unit away from  $R$

We can easily define the region using `RegionUnion` and then graph it using `RegionPlot`:



*defining the region and looking at its graph*



We can find the area and centroid of the region directly; to find the length of the boundary we will need to define a second region using RegionBoundary:

```

Area[region]

$$\frac{1}{2} \left( 3 \sqrt{15} + 22 \pi - 18 \operatorname{ArcCos}\left[\frac{7}{8}\right] + 8 \operatorname{ArcSin}\left[\frac{11}{16}\right] \right)$$


N[%]
38.8509

RegionCentroid[region]

$$\left\{ \frac{9 \left( 7 \sqrt{15} + 64 \pi - 64 \operatorname{ArcCos}\left[\frac{7}{8}\right] \right)}{8 \left( 3 \sqrt{15} + 22 \pi - 18 \operatorname{ArcCos}\left[\frac{7}{8}\right] + 8 \operatorname{ArcSin}\left[\frac{11}{16}\right] \right)}, 0 \right\}$$

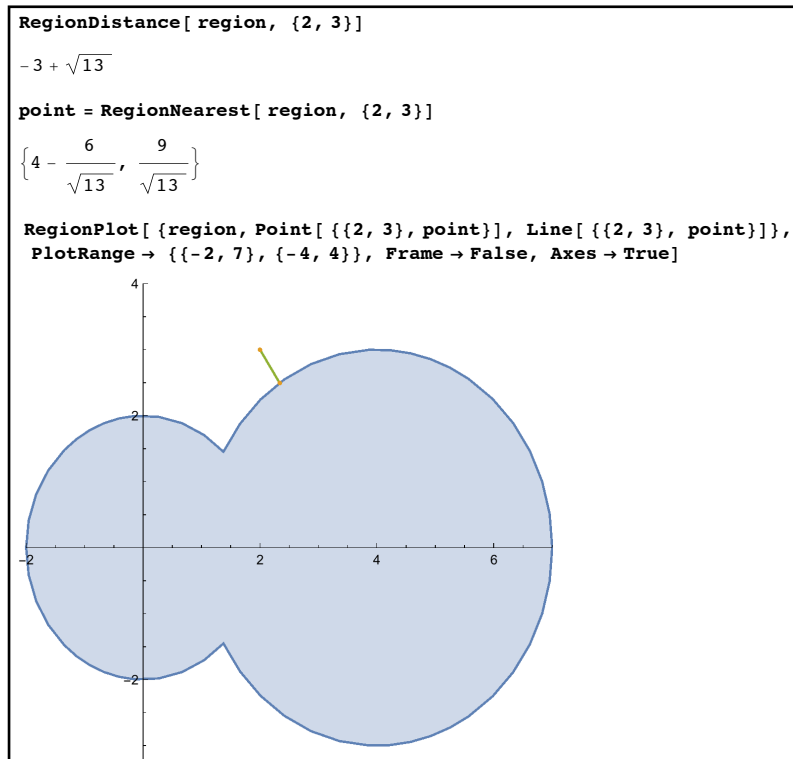

N[%]
{2.83531, 0.}

region2 = RegionBoundary[region];
ArcLength[region2]
4 \pi

```

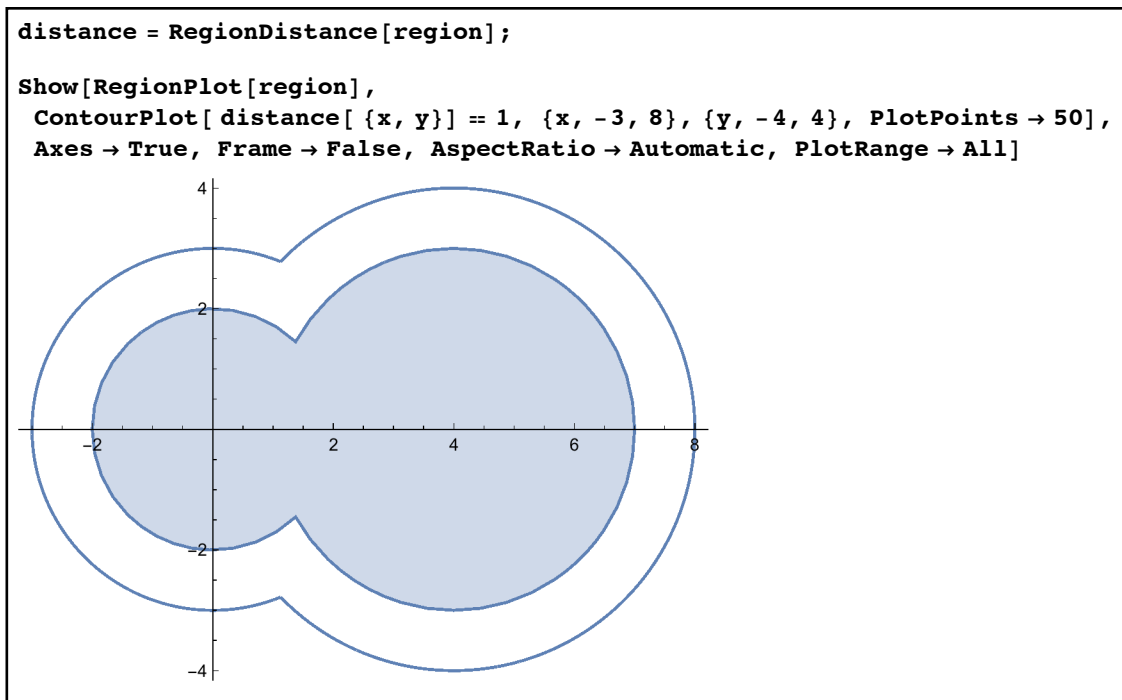
*some basic properties of the region*

To find the distance from (2,3) to the region we can use RegionDistance directly; likewise we can use RegionNearest to find which region point is closest to (2,3):



*the point in the region closest to (2,3)*

To graph all points which are 1 unit away from the region we will need to define a general distance function with `RegionDistance` before creating a graph with `ContourPlot`:



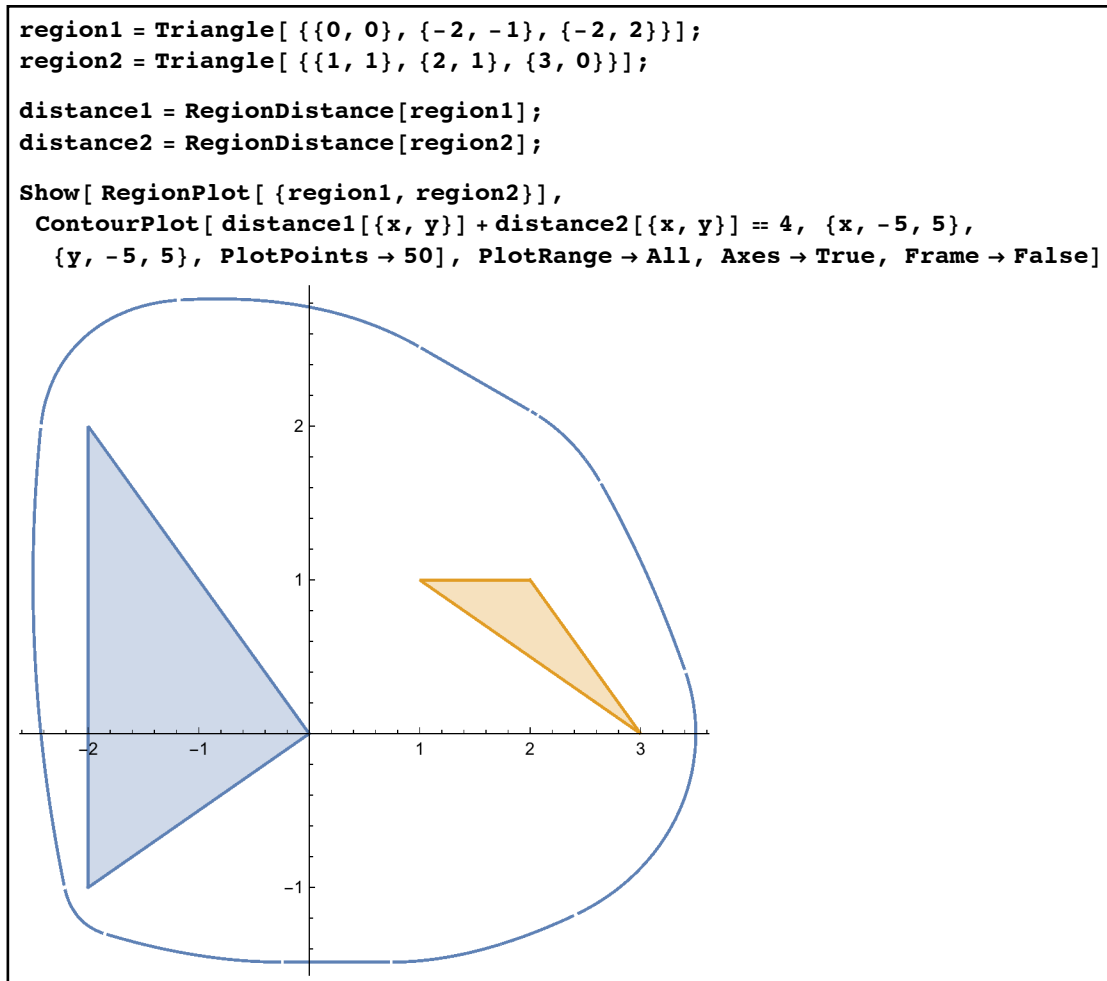
*all points 1 unit away from the region*

One important note about how the graphics are combined in this example: when combining region graphs with other graphs (like the one produced by `ContourPlot`) it is better to have the `RegionPlot` first in the `Show`. When a `RegionPlot` is done after other graphs in a `Show` command the region will be often be filled with mesh lines (a side effect of how Mathematica handles transparency).

#### Example 2: An “ellipse” with triangular foci

We know the standard definition of an ellipse is all points  $(x, y)$  such that the sum of the distances from  $(x, y)$  to two fixed points (the foci) is a constant. If  $T_1$  is the triangle whose corners are  $(0,0)$ ,  $(-2,-1)$ , and  $(-2,2)$  and  $T_2$  is the triangle whose corners are  $(1,1)$ ,  $(2,1)$ ,  $(3,0)$ , what are those points  $(x, y)$  for which the sum of the distances from  $(x, y)$  to  $T_1$  and  $T_2$  is 4?

This would be very hard to do by hand but we can set this up fairly easily in Mathematica. All we need to do is define  $T_1$  and  $T_2$  as regions, create the distance functions for each, and then set up the “sum is 4” in `ContourPlot`:



*an ellipse with triangular foci*

The small breaks in the graph are difficult to get rid of without setting PlotPoints incredibly high - a consequence of the two distance functions being complex piecewise functions.

In addition to having their own properties regions are useful in other commands such as Solve and Maximize in problems where the points under consideration must come from known regions. To require the points come from a region you simply use the “is an element of” symbol, which you can create using the key combination Esc-elem-Esc.

### Example 3: A constrained Solve command

Find all solutions to the system  $y = x^3$ ,  $y = 4x$  that lie in the unit disk.

The only difference in the Solve command is that we want the solutions to be within one unit of the origin - that is in the region `Disk[{0,0}, 1]`:

```

Solve[ {y == x^3, y == 4 x}, {x, y} ∈ Disk[{0, 0}, 1] ]
{{x → 0, y → 0}}

Solve[ {y == x^3, y == 4 x}, {x, y}]
{{x → -2, y → -8}, {x → 0, y → 0}, {x → 2, y → 8}}

```

*solutions in the unit disk as opposed to those in the entire plane*

In this case there is only 1 solution in the unit disk (as opposed to 3 if there are no restrictions at all).

Example 4: A constrained optimization problem

What is the greatest value of  $x^3 - y - 2z^2$  on the sphere of radius 2 centered at (4,2,0)?

We could do this without using regions but this would require us to find the equation of the sphere (which is not difficult but is more work). Instead we can just require the points be in `Sphere[{4,2,0},2]`:

```

Maximize[ x^3 - y - 2 z^2, {x, y, z} ∈ Sphere[{4, 2, 0}, 2] ]
{
  -Root[228 728 832 - 73 927 040 #1 +
    24 354 016 #1^2 + 219 456 #1^3 - 312 984 #1^4 + 154 548 #1^5 + 729 #1^6 &, 1],
  {x → Root[16 - 8 #1 + #1^2 + 108 #1^4 - 72 #1^5 + 9 #1^6 &, 2],
   y → Root[16 - 8 #1 + #1^2 + 108 #1^4 - 72 #1^5 + 9 #1^6 &, 2]^3 + Root[228 728 832 - 73 927 040 #1 +
    24 354 016 #1^2 + 219 456 #1^3 - 312 984 #1^4 + 154 548 #1^5 + 729 #1^6 &, 1], z → 0}}
N[%]
{214.009, {x → 5.99991, y → 1.98148, z → 0.}}

```

*maximizing a function on a sphere*

Example 5: Reduce in a region

Find all solutions to  $\sin(x) = \cos(x - 2y)$ , where both  $x$  and  $y$  run from 0 to  $2\pi$ .

For  $x$  and  $y$  to be between 0 and  $2\pi$  is the same as saying the point  $(x, y)$  must be in the rectangle whose opposite corners are (0,0) and  $(2\pi, 2\pi)$ . We can implement this directly in `Reduce` using the `Rectangle` region:

$$\begin{aligned}
 &\text{Reduce}[\text{Sin}[\mathbf{x}] == \text{Cos}[\mathbf{x} - 2 \mathbf{y}], \{\mathbf{x}, \mathbf{y}\} \in \text{Rectangle}[\{0, 0\}, \{2 \text{Pi}, 2 \text{Pi}\}]] \\
 &\left(0 \leq \mathbf{x} \leq 2 \pi \ \&\& \ \left(\mathbf{y} == \frac{5 \pi}{4} \ || \ \mathbf{y} == \frac{\pi}{4}\right)\right) || \\
 &\left(\frac{\pi}{4} \leq \mathbf{x} \leq 2 \pi \ \&\& \ \mathbf{y} == \frac{1}{4} (-\pi + 4 \mathbf{x})\right) || \left(0 \leq \mathbf{x} \leq \frac{\pi}{4} \ \&\& \ \mathbf{y} == \frac{1}{4} (7 \pi + 4 \mathbf{x})\right) || \\
 &\left(\frac{5 \pi}{4} \leq \mathbf{x} \leq 2 \pi \ \&\& \ \mathbf{y} == \frac{1}{4} (-5 \pi + 4 \mathbf{x})\right) || \left(0 \leq \mathbf{x} \leq \frac{5 \pi}{4} \ \&\& \ \mathbf{y} == \frac{1}{4} (3 \pi + 4 \mathbf{x})\right)
 \end{aligned}$$

*solving a constrained equation with Reduce*

In this example we could have added the constraints on  $x$  and  $y$  by adding some inequalities to the logical statement but region objects give us another approach to the problem.

## Section 5.6 Homework - Geometric Computation and Region Objects

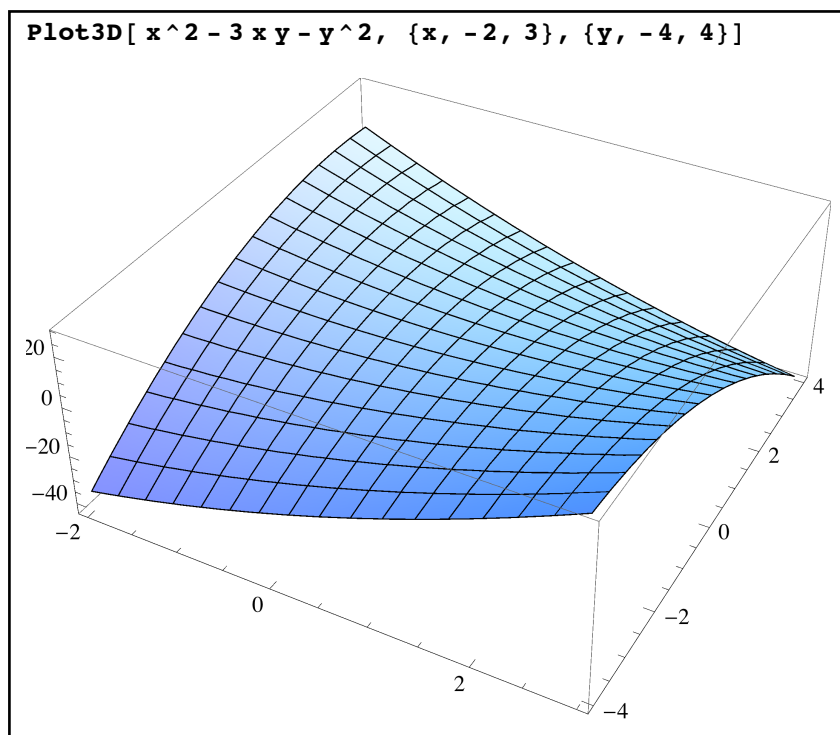
- 1) If  $R$  is the region which the triangle whose corners are (2,3), (4,1), and (9,7), find the area of  $R$ , the centroid of  $R$ , and the perimeter of  $R$ .
- 2) Let  $R$  be the region bounded between the curves  $y = x^3$  and  $y = x^4$ . Graph  $R$  and find both its area and its centroid. What point in  $R$  is closest to (0,1)?
- 3) Let  $R$  be the intersection of spheres of radius 2 centered at (0,0,0) and (1,0,0). What is the dimension and measure of  $R$ ?
- 4) Let  $R$  be the union of a solid ball of radius 3 centered at (0,0,0) and the solid ball of radius 4 centered at (2,0,0). Graph  $R$ , making sure to plot enough points for a good picture. Find the volume of  $R$  and the maximum value of  $x^2 + 4yz$  on  $R$ . What are the bounds on  $R$ ?
- 5) Use region objects to get the arc length of the curve  $y = x^4$  as  $x$  goes from 0 to 2 (both exactly and as a numerical estimate).
- 6) If  $R$  is the region above the  $x$ -axis and below  $y = 4 + \cos(2x)$  as  $x$  goes from 0 to  $2\pi$ , what is the area and centroid of  $R$ ? Explain why you could do this without region objects but in a much more painful way.
- 7) If  $R$  is the region above the  $x$ -axis and below  $y = \frac{1}{x^2}$ ,  $x \geq 1$ , what are the bounds and area of  $R$ ?
- 8) Create a function that computes the distance from a point  $(x, y)$  to the region defined by  $\frac{x^2}{4} + \frac{y^2}{9} = 1$ . Use the function to compute the distance from the region to the points (1,0), (4,5), and (11,1).
- 9) Create a chain of commands as follows:
  - a) The first command defines the region  $R$  which is a disk of radius 7 centered at the origin with a triangle from (0,4) to (5,-3) to (-2,1) removed.
  - b) The second command graphs  $R$  from -10 to 10 in both directions without a frame but with axes.
  - c) The third command is simply  $\text{pt}=\{2,1\}$ ;
  - d) The fourth command finds the point  $\text{pt2}$  in  $R$  nearest  $\text{pt}$ .

- e) The fifth command combines the graph from b) with large points representing  $pt1$  and  $pt2$  and the line between them.
- 10) Copy the command sequence from problem 9 into a new cell. Delete the definition of  $pt1$  and put the commands from 9d and 9e inside a Manipulate whose only control is  $pt1$  which is set by an input field.

## Section 5.7 - Into the Third Dimension

So far we have restricted our discussion to what is involved in first-year calculus courses - essentially calculus in the plane. The ideas of calculus extend beyond the plane into space (and even beyond space to higher dimensions). In this section we will learn how to use Mathematica to graph in three dimensions (and leave the computations to the section that follows).

The most basic kind of three-dimensional graphing is that of a surface of the form  $z = f(x, y)$  (the 3-D analog of  $y = f(x)$ ). The basic command for graphing such a surface is `Plot3D`. The format for `Plot3D` is `Plot3D[formula, {x,a,b}, {y,c,d}]`, which graphs the expression  $z = \text{formula}$  over the rectangle as  $x$  goes from  $a$  to  $b$  and as  $y$  goes from  $c$  to  $d$ . For example to graph  $z = x^2 - 3xy - y^2$  as  $x$  goes from -2 to 3 and  $y$  goes from -4 to 4 we would use `Plot3D[x^2 - 3 x y - y^2, {x,-2,3}, {y,-4,4}]`:



*the graph of a surface*

There are three features of this presentation of the surface that are worth mentioning. Despite the fact that the range of  $x$  values is smaller than the range of  $y$  values the rectangle the surface is graphed over is presented as a square (so the scale for  $y$  is smaller than the scale for  $x$ ). This type of automatic scaling is one of the default settings for `Plot3D`. The second feature is that you can view this surface from many different vantages (which we'll think of as `ViewPoints` later). To rotate the surface in 3 dimensions simply left-click on the image and drag around in different directions. This will show you the surface from whatever vantage you want, although

in most cases the apparent size of the surface will alter as well (as you rotate the surface essentially you get closer to or farther from different features so they scale). Third, as you are graphing over a rectangle you will usually see “corners” on the graph. These corners don’t really belong to the surface itself (many completely “round” surfaces will have corners when graphed with Plot3D) but are an artifact of the fact that you are graphing over a rectangular region.

Unlike Plot the Plot3D command cannot be used to graph several surfaces at once. You will need to create each plot independently and then combine them using Show. Like Plot Plot3D has many options for fine-tuning the appearance of a graph. Some of the basic ones are:

**PlotPoints:** PlotPoints→ $n$  uses  $n$  points in each direction to graph the surface, so just as in Plot you can use the option to get a finer graph. As you are graphing over a two-dimensional region increasing the PlotPoints requires a lot more work for the computer (since it has to subdivide both  $x$  and  $y$  for graphing). While using a value like PlotPoints→1000 may not be too much for Plot it can be very intensive for Plot3D.

**Mesh:** Used as either Mesh→True (the default) or Mesh→False, this option determines whether to show the “grid” on the surface or not. The mesh aids perspective but sometimes it can make it difficult to distinguish certain features on the graph.

**Boxed:** The settings Boxed→True (the default) or Boxed→False tells Plot3D whether to use a bounding box around the graph. When set to False axes will still be present on the plot edges.

**Axes:** This works the same way as Boxed just for the axes.

**BoxRatios:** BoxRatios→ $\{a,b,c\}$  sets the relative lengths of the  $x$ ,  $y$ , and  $z$  directions on the graph (the base setting is  $\{1,1,4\}$ , which is why the region you graph over is shown as a square by default and the height of the graph is 40% of the width). Setting BoxRatios→Automatic sets the scales for the 3 axes to be the same (just as how AspectRatio→Automatic works in Plot). The Automatic setting can create problems for graphs that rise and fall a lot - a surface with a wide  $z$  range and a small range for  $x$  and  $y$  will appear almost tubular.

**PlotRange:** PlotRange→ $\{zmin, zmax\}$  only shows  $z$ -values in that range (just as PlotRange does for Plot). If the graph of the surface leaves that range however it will be “clipped” against that face of the box rather than not shown at all (so the surface will appear to be flattened against the edge of box). You can set the PlotRange for all 3 directions by PlotRange→ $\{\{xmin,xmax\}, \{ymin,ymax\}, \{zmin,zmax\}\}$ .

**ClippingStyle:** Setting ClippingStyle→None will prevent Mathematica from “flattening” the surface against a face of the bounding box and simply not show the surface



there (which is more true to how the surface should look but the normal “clipping” lets the user know the surface goes beyond the box at a glance).

**PlotLabel:** This works for Plot3D just like it does for Plot.

**AxesLabel:** This works for Plot3D, but you need to include 3 labels instead of just two (AxesLabel→{“x”, “y”, “z”}).

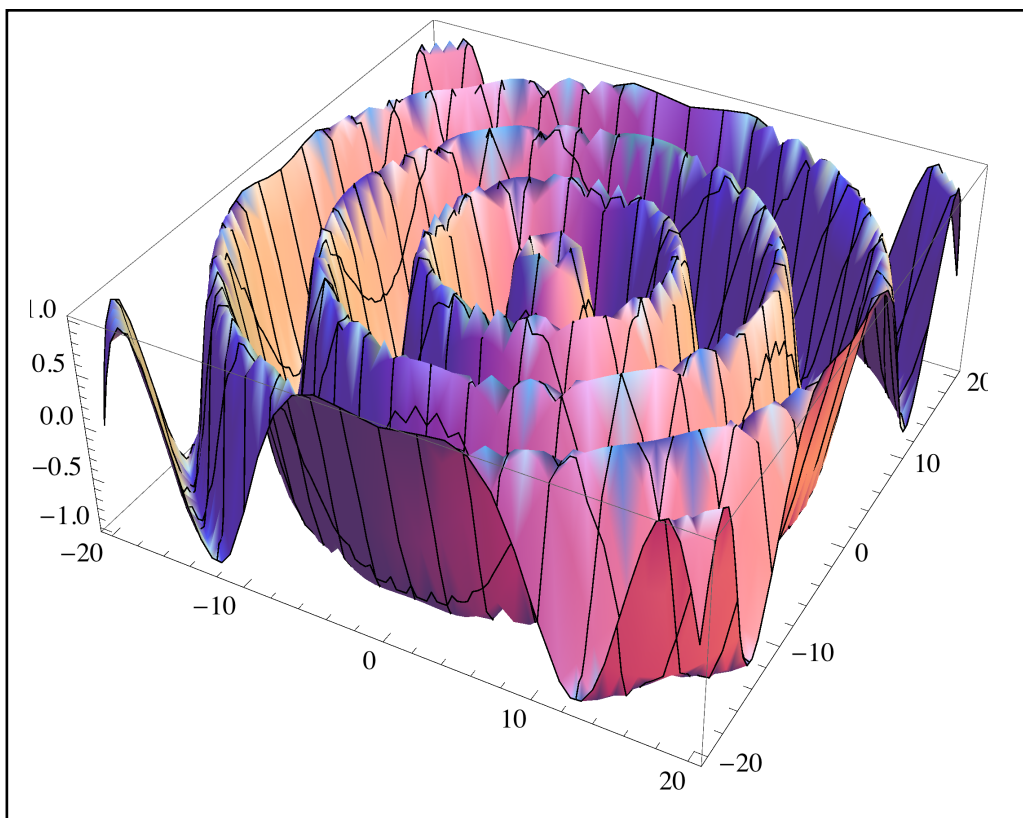
**Ticks:** This works just like it does for Plot - except you have to give a list of 3 lists rather than just two ({*xticklist*, *yticklist*, *zticklist*}).

**ViewVertical:** ViewVertical→{*a,b,c*} (not all 0) tells Mathematica to rotate the picture so that the vector/direction {*a,b,c*} is “up”. By default ViewVertical is set to {0,0,1} so the *z*-axis is up.

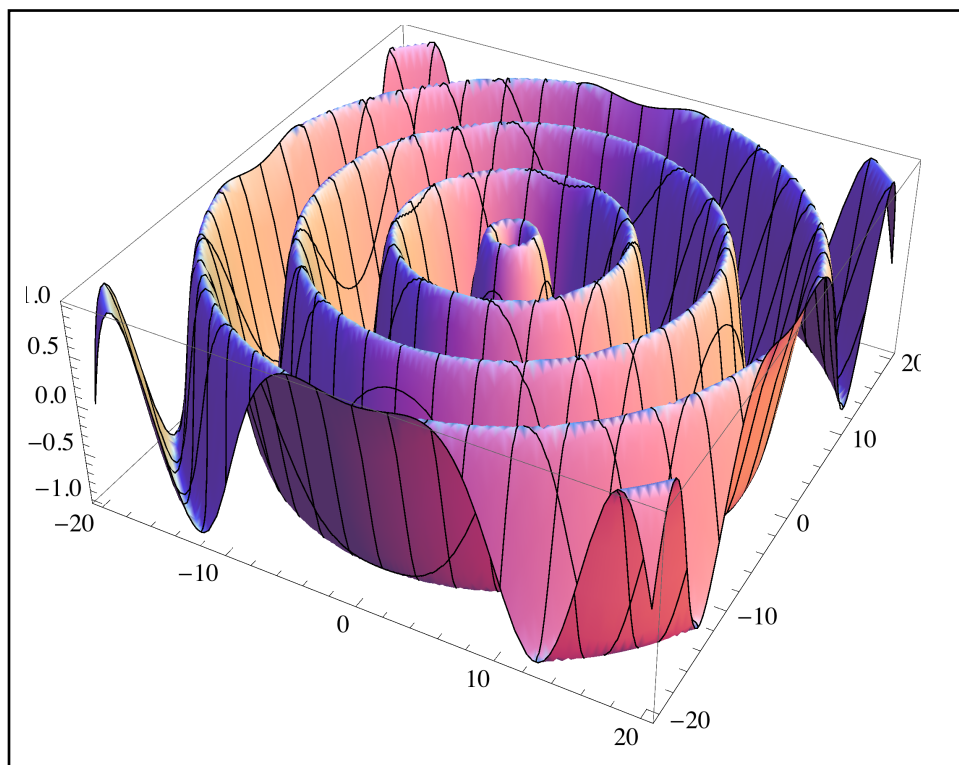
**ViewPoint:** ViewPoint→{*a,b,c*} (not all 0) tells Mathematica to arrange the view so you are essentially looking at the surface from the end of the vector {*a,b,c*}, at least up to scaling. This essentially lets you set manually and exactly what you could achieve approximately by rotating the surface via dragging. The apparent size of the surface may change with different ViewPoints as parts of the surface may extend more in different directions (and so viewing from those directions may require the picture to be scaled).

**SphericalRegion:** SphericalRegion→True encases the surface in an invisible sphere. You use this to eliminate the scaling that may occur as you look at the surface from different ViewPoints (since you are always the same distance from the sphere the scale never changes). This makes the surface appear smaller (as it is scaled to fit inside the sphere) but the uniform scale it provides is very useful in animations.

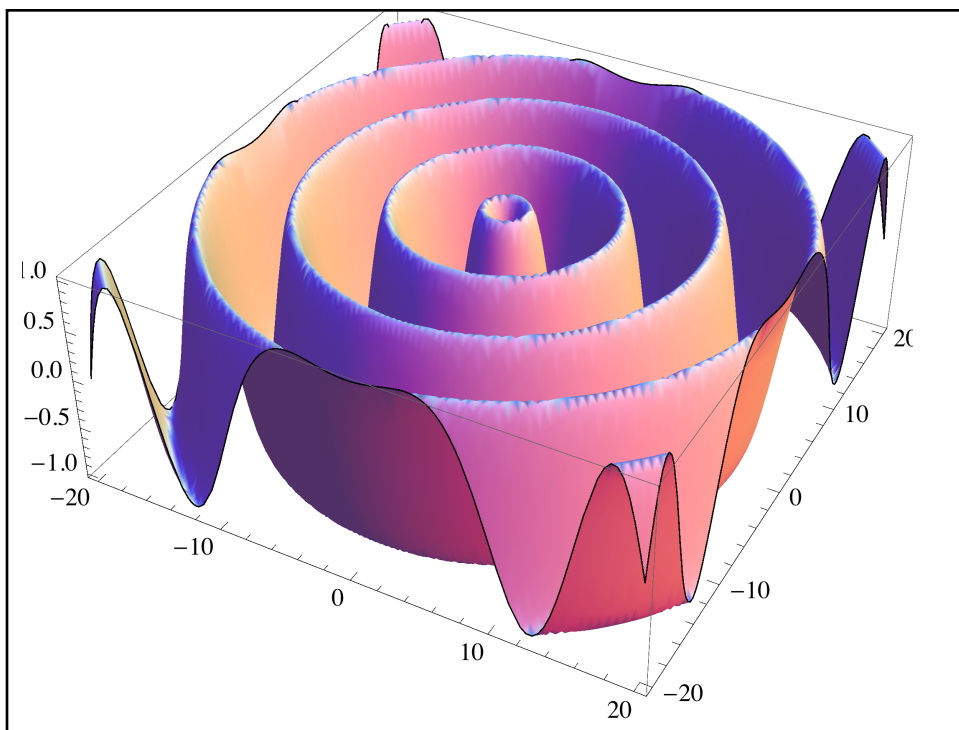
Below is the graph of the surface  $z = \sin(\sqrt{x^2 + y^2})$  as *x* and *y* go from -20 to 20 with various option sets:



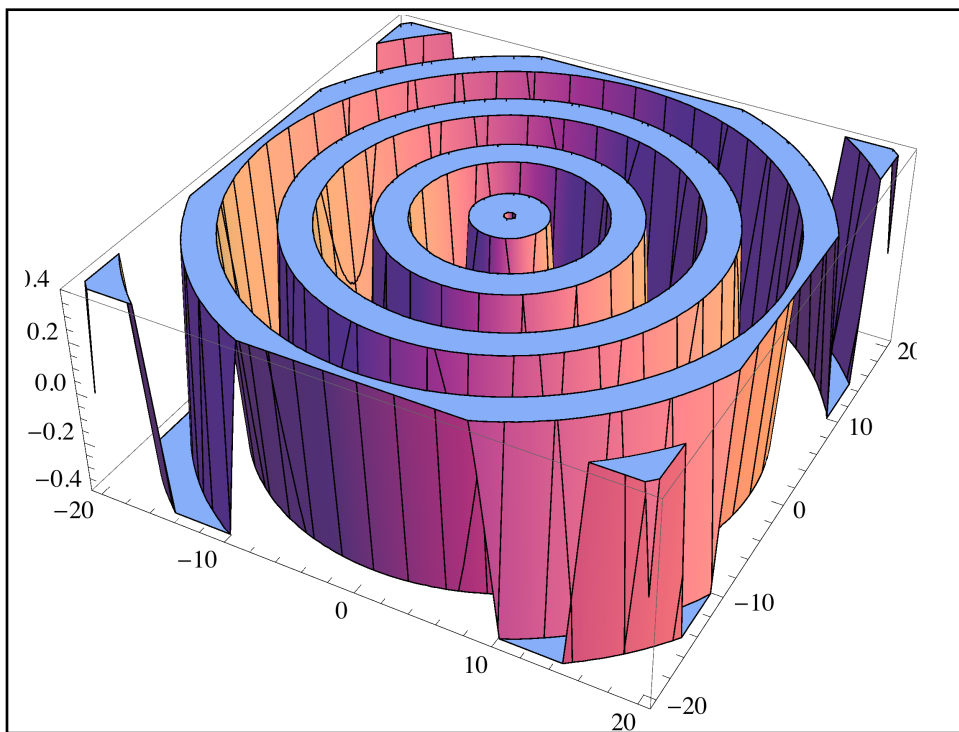
*default options*



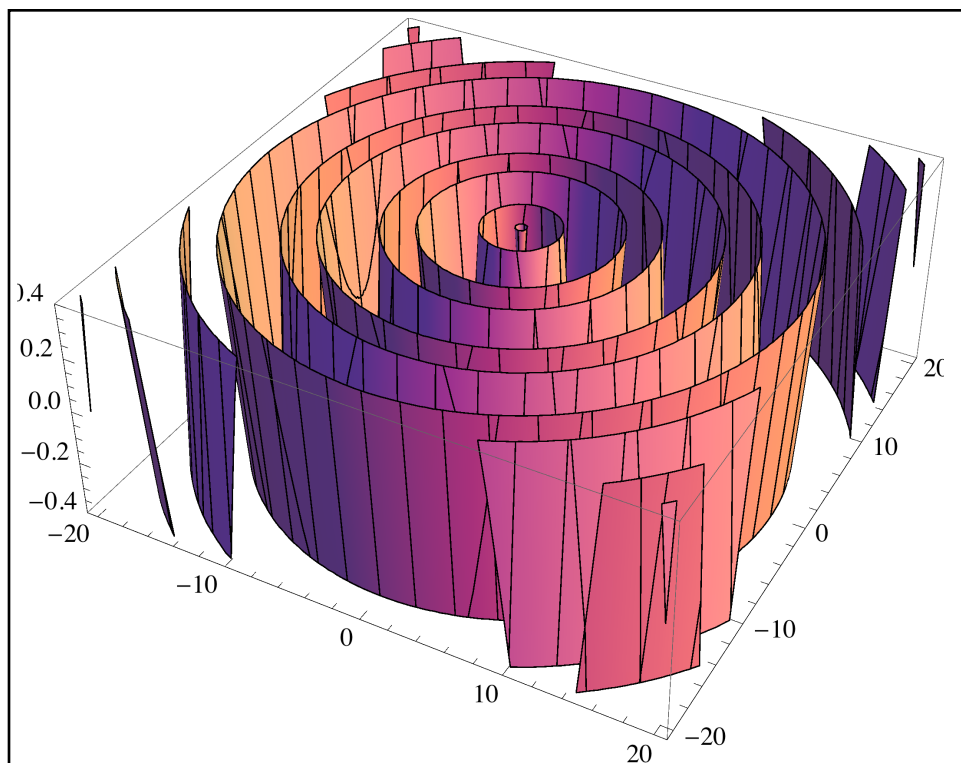
*PlotPoints -> 50*



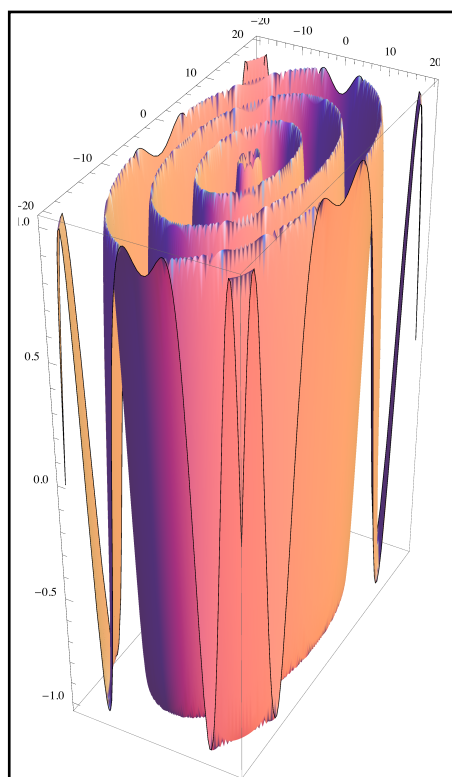
*PlotPoints*→50, *Mesh*→None



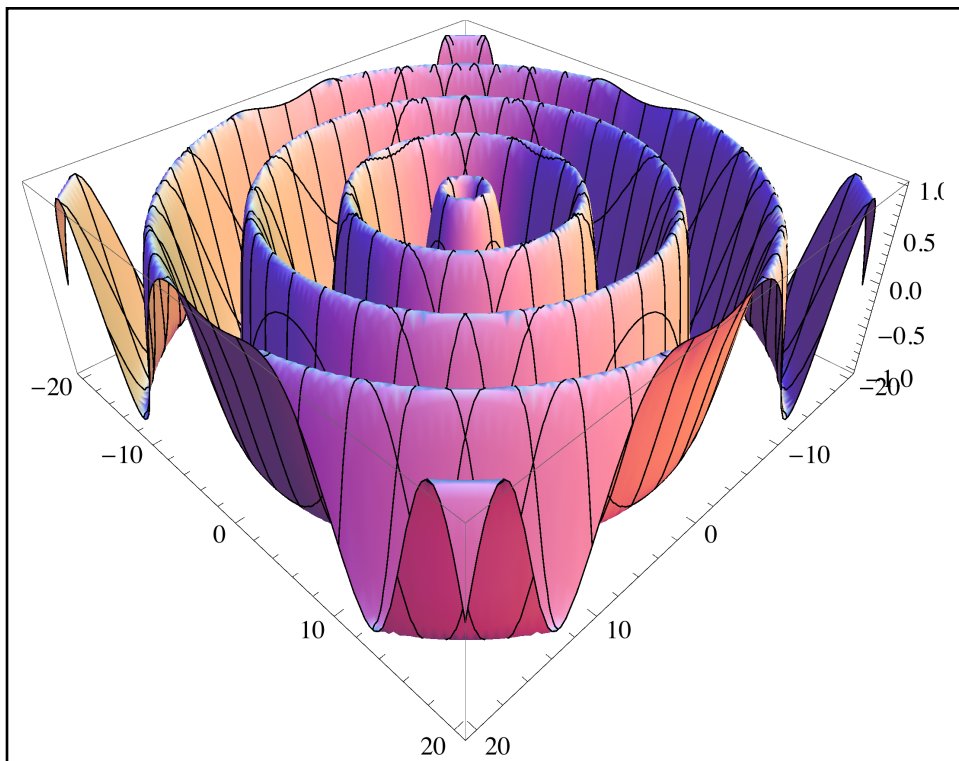
*PlotPoints*→50, *PlotRange*→{-0.4, 0.4}



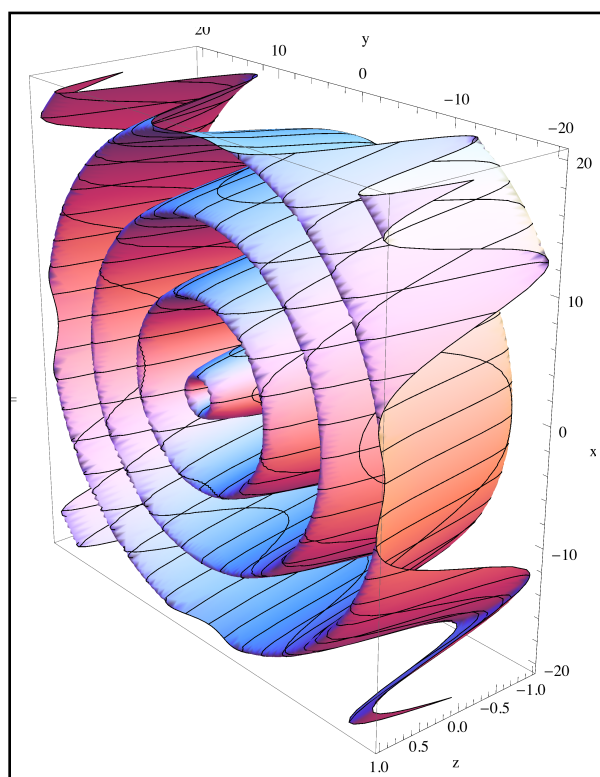
*PlotPoints*→50, *PlotRange*→{-0.4,0.4}, *ClippingStyle*→None



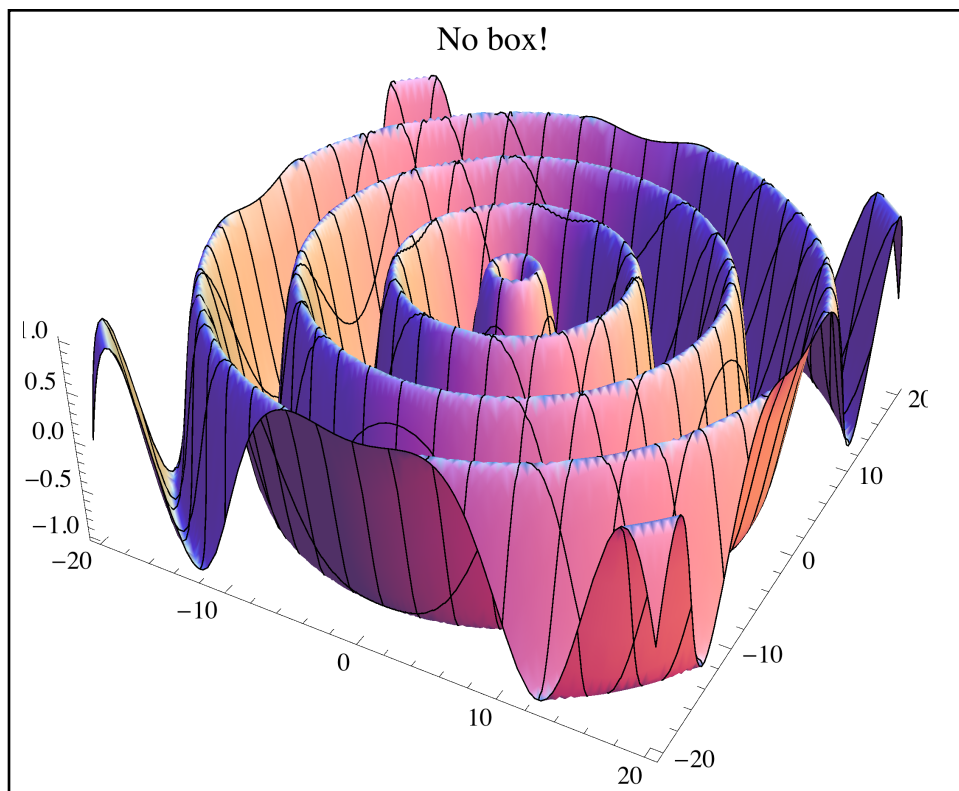
*PlotPoints*→50, *BoxRatios*→{1,2,3}, *ClippingStyle*→None



*PlotPoints*→50, *ViewPoint*→{1,1,1}



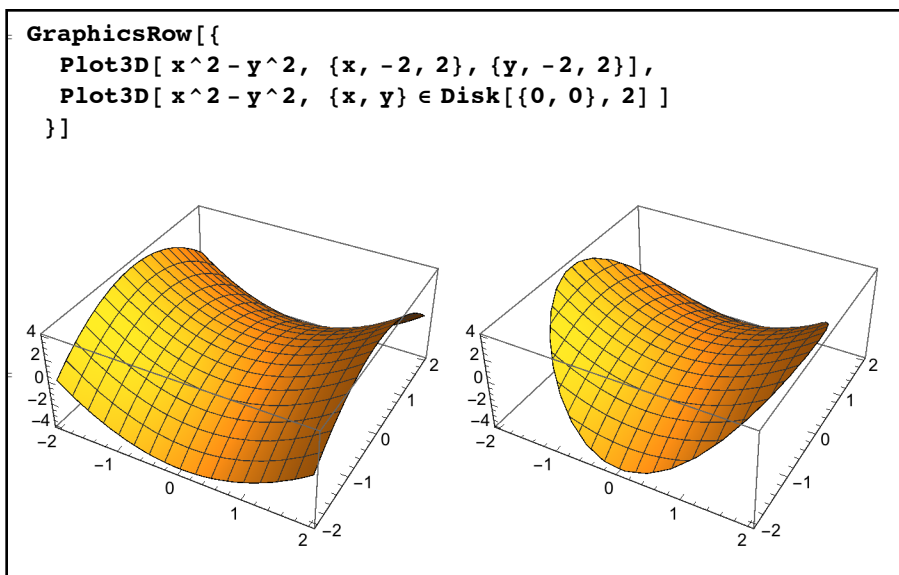
*PlotPoints*→50, *ViewVertical*→{1,0,0}, *AxesLabel*→{"x", "y", "z"}



*PlotPoints*→50, *Boxed*→None, *PlotLabel*→"No box"

The Show command plays the same role for Plot3D as it does for Plot. It can be used to combine graphics and to change options that don't require regraphing (such as ViewPoint but not PlotPoints).

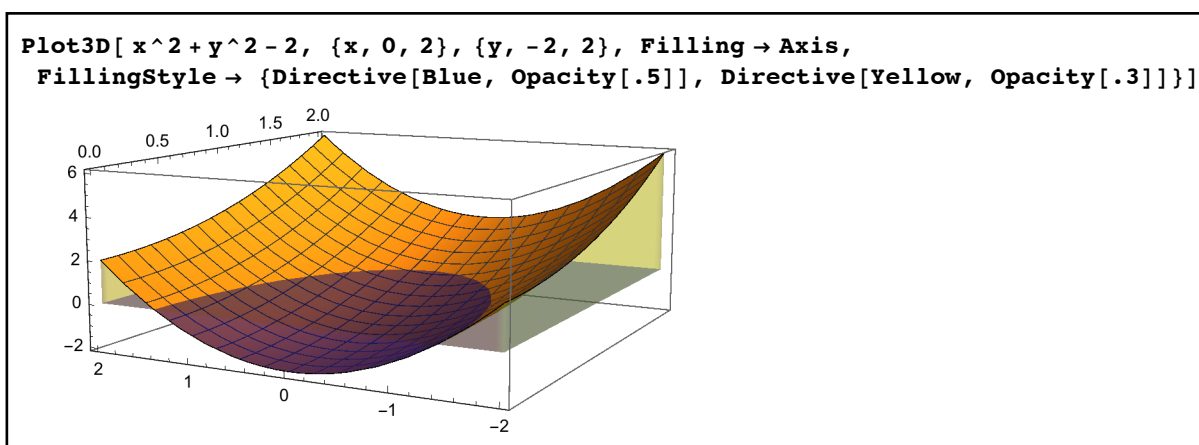
In some cases you will not want to graph a surface over a rectangular range of  $x$ - and  $y$ -values but over some other set (say a circular region or over a region bounded by various curves in the plane). To do this we can take advantage of region objects. `Plot[formula, variable list ∈ region]` will graph the formula only over the given region (provided the *region* is two dimensional - this won't work if the *region* is a curve). Here are the graphs of  $z = x^2 + y^2$  graphed over the full  $x$ - and  $y$ -range from -2 to 2 and just over a disk of radius 2 centered at the origin:



*graphing a surface over a non-rectangular region*

Previously this kind of “graphing over a region” was done using an option called `RegionFunction` but the addition of region objects in Mathematica 10 makes this significantly easier.

In some cases you won’t just want to graph the surface but also the region down or up to the  $xy$ -plane (where  $z=0$ ). This is accomplished by the option `Filling→Axis`, which shades the region between the surface and the plane (by default a semi-transparent gray). The option `FillingStyle→{color directive 1, color directive 2}` shades the region below the  $xy$ -plane according to the first directive and the region above the  $xy$ -plane according to the second directive. If you use pure colors for the directives they will be solid by default (and so obscure the surface) so it is common to use `Directive` together with something to control transparency (such as `Opacity[.3]`):

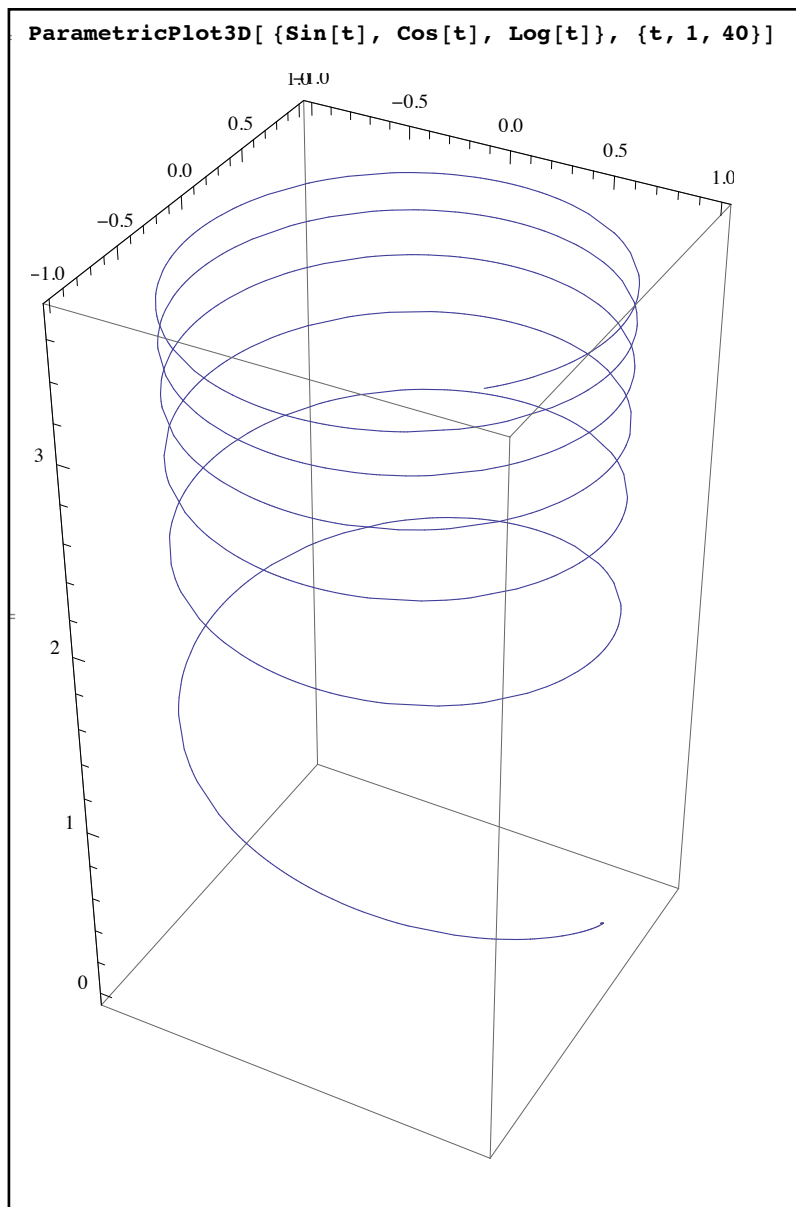


*shading regions above and below a surface with `Filling` and `FillingStyle`*



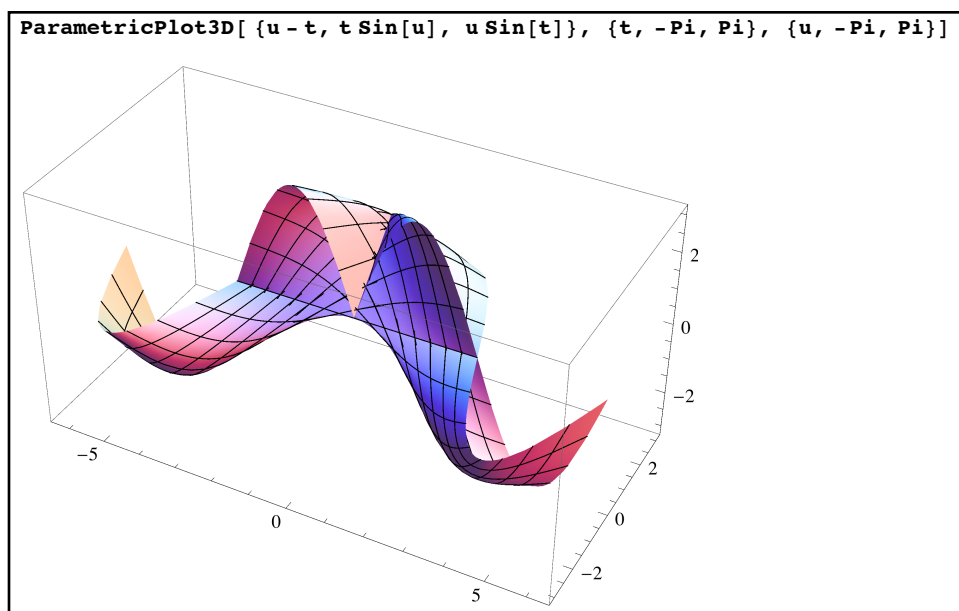
As these pictures can get pretty involved many people only use Filling when the surface is entirely above or below the surface - in which case you can get away with FillingStyle set to a single directive.

In addition to standard Cartesian graphing Mathematica will also plot surfaces and curves parametrically through ParametricPlot3D. To plot a 3-D curve parametrically use ParametricPlot[  $\{x_t, y_t, z_t\}, \{t, a, b\}$  ], where the curve is defined with  $x, y$ , and  $z$  in terms of  $t$  (and  $t$  goes from  $a$  to  $b$ ). For surfaces use the form ParametricPlot[  $\{x_{t,u}, y_{t,u}, z_{t,u}\}, \{t, a, b\}, \{u, c, d\}$  ], where the  $x, y$ , and  $z$  are defined in terms of  $t$  and  $u$ , and  $t$  and  $u$  vary over given ranges. For example:



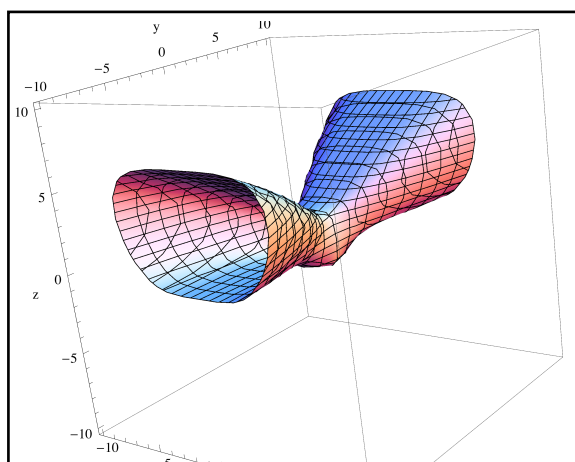
*a parametric helix*





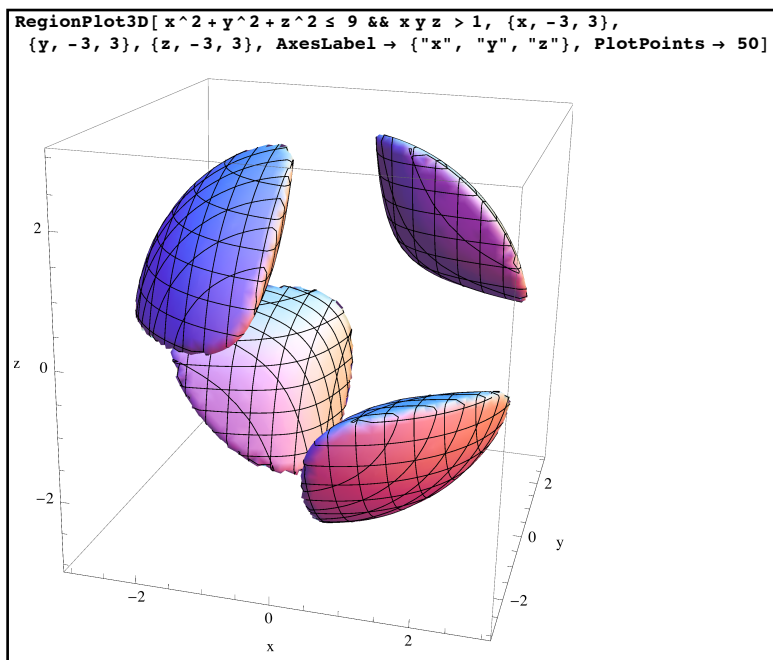
*a parametric surface*

Mathematica also has 3-D tools analogous to ContourPlot (to graph the solutions to equations) and RegionPlot (for graphing regions defined by inequalities) - ContourPlot3D and RegionPlot3D. To graph an equation of the form  $lhs=rhs$  over a given range of  $x$ ,  $y$ , and  $z$  simply use ContourPlot3D[  $lhs==rhs$ , { $x,a,b$ }, { $y,c,d$ }, { $z,e,f$ }]. ContourPlot3D works by subdividing the three-dimensional region into smaller and smaller grids to determine if a surface passes through a point; this is very resource-intensive, so ContourPlot3D can take a lot of time and memory, especially when working with equations involving trigonometric functions. ContourPlot3D uses many of the same appearance options as Plot3D (including PlotLabel, AxesLabel, etc.) which can be useful in making the results easier to read. For example, if we wanted to graph the surface given by  $x^4 - 3y^2z - y^2 + z^4 = 10$  over a reasonable region we might use the command ContourPlot3D[  $x^4-3 y^2 z-y^2 + z^4==10$ , { $x,-10,10$ }, { $y,-10,10$ }, { $z,-10,10$ }, AxesLabel→{"x","y","z"}]:



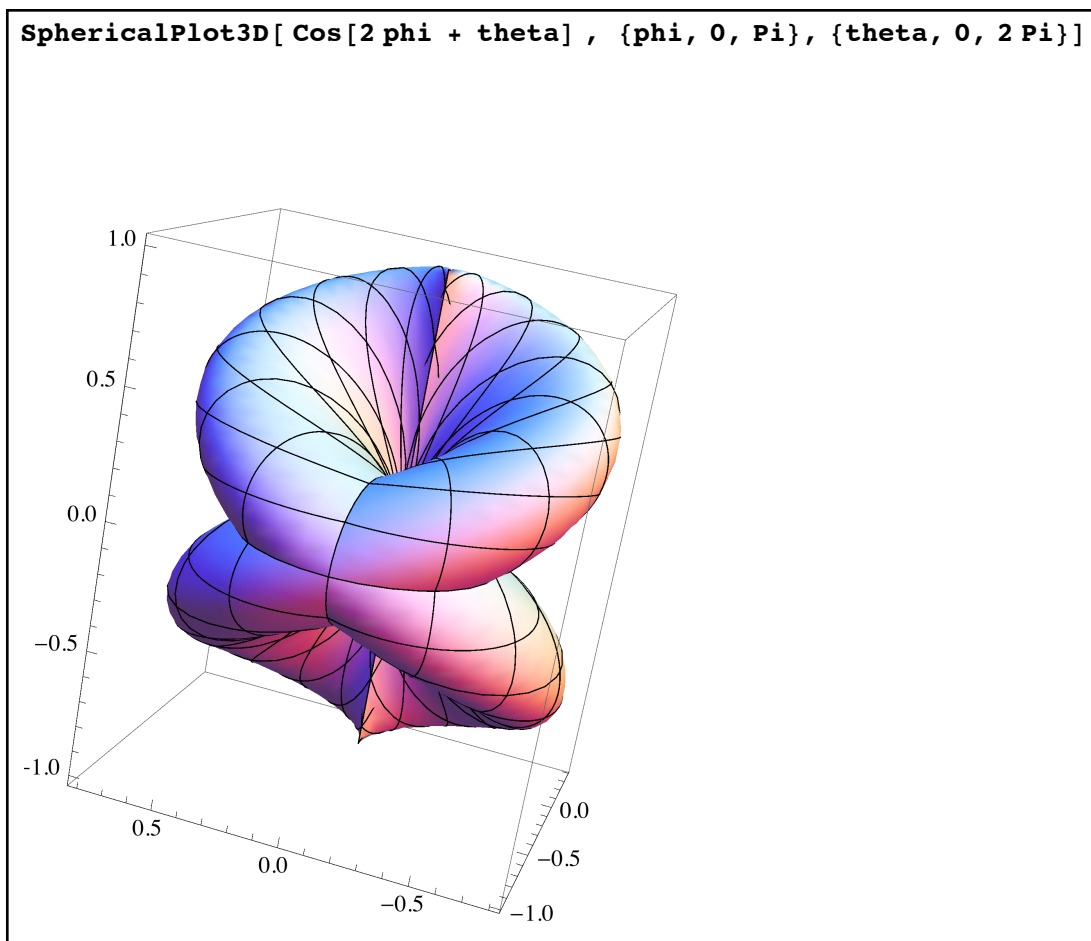
*the graph of an equation in three dimensions via ContourPlot3D*

RegionPlot3D works much the same way except you use a logical statement (usually a set of inequalities chained with && and ||) in place of an equation. Just like RegionPlot it is fairly common to get a jagged ill-defined edge on certain boundaries of the region - you can fix this by increasing the value of PlotPoints. For example, suppose we wanted to graph those points which are which satisfy both the inequalities  $x^2 + y^2 + z^2 \leq 9$  and  $xyz > 1$ , we could use the command `RegionPlot3D[ x^2+y^2+z^2<=9 && x y z>1, {x,-3,3},{y,-3,3},{z,-3,3}, AxesLabel->{"x","y","z"}, PlotPoints->50]`:



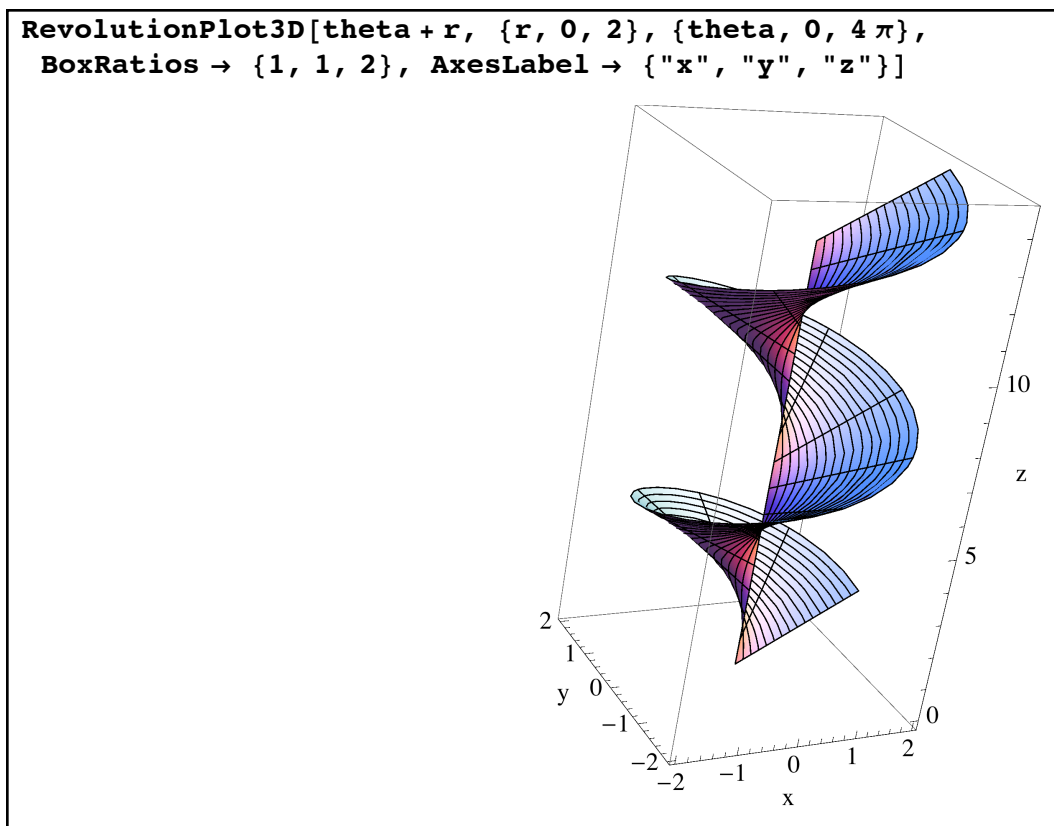
*graphing a region satisfying inequalities via RegionPlot3D*

Mathematica can also graph in spherical and cylindrical coordinates. Recall that in spherical coordinates every point is determined by a triple  $(\rho, \theta, \phi)$  where  $\rho$  is the distance from the origin,  $\theta$  is the polar angle in the plane, and  $\phi$  is the angle down from the z-axis. Many equations in spherical coordinates have the form  $\rho = f(\theta, \phi)$ . To graph this type of equation we can use SphericalPlot3D. This command has the form `SphericalPlot3D[formula, {phi,a,b}, {theta,c,d}]` and uses many of the same options as Plot3D. The order of the angles in the parameter list makes a big difference - make sure the parameter list for  $\phi$  comes first and the parameter list for  $\theta$  comes second. So if we want to graph the equation  $\rho = \cos(2\phi + \theta)$  over the usual ranges for  $\phi$  and  $\theta$  we would use `SphericalPlot3D[ Cos[2phi+theta], {phi,0,Pi}, {theta, 0,2Pi}]`:



*graphing an equation in spherical coordinates*

Recall that cylindrical coordinates is a hybrid of Cartesian and polar coordinates; points are identified as  $P = (r, \theta, z)$  where  $z$  is the distance of the point above the  $x$ - $y$  plane and it lies over the polar point  $(r, \theta)$  in the plane. Graphing in cylindrical coordinates is now handled by `RevolutionPlot3D`. If your surface has the form  $z = f(r, \theta)$ , to graph it over a given range of radii  $r$  and angles  $\theta$  use the command `RevolutionPlot3D[formula, {r,a,b}, {theta,c,d}]`. For example, suppose you wanted to graph the simple cylindrical surface  $z = r + \theta$  where the radius goes from 0 to 2 and the angle goes from 0 to  $4\pi$ , you would use `RevolutionPlot3D[ r+theta, {r,0,2}, {theta,0,4Pi}]`. `RevolutionPlot3D` uses many of the same options as `Plot3D`:



*graphing a helical surface in cylindrical coordinates using RevolutionPlot3D*

To sum up Mathematica should be able to handle most of the three dimensional graphing needs you would encounter in third semester calculus. It can handle Cartesian graphing of surfaces (through Plot3D, ParametricPlot3D, ContourPlot3D, and RegionPlot3D), the graphing of curves in space (via ParametricPlot3D), and graphing in alternative coordinate systems (using SphericalPlot3D and RevolutionPlot3D).

## Section 5.7 Homework - Into the Third Dimension

- 1) Graph the function  $z = \sin(\sqrt{2x^2 + y^2})$  where  $x$  and  $y$  both range from  $-2\pi$  to  $2\pi$ .
- 2) Graph the surface  $z = x^2 - y^2$  over the square which ranges from  $-4$  to  $4$  in both directions. When making your graph, remove the surface's mesh, the bounding box, and the axes. Make sure to use enough points to get a smooth picture.
- 3) Graph the surface  $z = \sin(x + 3y) + 2$  from  $-10$  to  $10$  in both directions. Show the  $z$ -range from  $-1$  to  $4$  and shade the region below the surface a translucent red (the surface should lie entirely above the  $xy$ -plane)
- 4) Graph the unit sphere  $x^2 + y^2 + z^2 = 1$  in 3 different ways - one which uses Plot3D to graph the top and bottom of the sphere (merged via Show), one that uses ContourPlot3D, and one that uses SphericalPlot3D.

- 5) How does `PlotRange`→{a,b} affect a `Plot3D` command? To illustrate this, plot  $z = x - y^2$  from -2 to 2 in both directions. Plot the surface normally, and then with the restricted `PlotRange` {-1,1}.
- 6) What effect does the option `ImagePadding` have in `Plot3D`?
- 7) What effect does the `Lighting`→None option have in `Plot3D`? The option `BaseStyle`→Blue?
- 8) Graph the parametric curve given by  $x = \sin(t)$ ,  $y = \cos(t)$ ,  $z = \sin(3t)$  using a good range for  $t$  and enough points to make the curve smooth.
- 9) Graph the parametric surface given by  $x = u$ ,  $y = u - v$ ,  $z = \frac{u^2}{v^2 + 1}$ , where both  $u$  and  $v$  range from -4 to 4. Plot enough points to make the surface smooth, and show the surface from 3 different vantage points.
- 10) Use `ParametricPlot3D` to graph  $z = x^2 - y^2$  over a circle of radius 3 centered at the origin.
- 11) Repeat problem 10, but graph the surface of a circle of radius 1 centered at (1,-1) (this will require you to shift the terms in the replacement of  $x$  and  $y$ ).
- 12) Plot the cylindrical equation  $z = \cos(\theta)\sin(r)$ , where both variables go from 0 to  $2\pi$ .
- 13) Plot the spherical equation  $\rho = \sin(\theta)\cos(2\phi)$ .
- 14) Graph the solutions to  $xyz = 1$  where  $x$ ,  $y$ , and  $z$  all range from -3 to 3.
- 15) Graph the region  $\sin(y + xz) > 0$ , where all variables range from  $-2\pi$  to  $2\pi$  (note - you will need to up the `PlotPoints`, which will be processor intensive).
- 16) Evaluate the following command: `Animate[Plot3D[Sin[Sqrt[x^2+y^2]+k]/(Sqrt[x^2+y^2]+k), {x,-15,15},{y,-15,15}, PlotPoints→40, PlotRange→{-1,1}], {k,0,2Pi, Pi/24}]`. What does this generate? How does the `Animate` command work? What happens when the output is “paused”?

## Section 5.8 - Calculus Computations for Three or More Dimensions

Now that we can view graphs in three dimensions it is time to add the ability to perform the related calculus computations in more than one variable (limits, derivatives, integrals, and series) as well. For the most part these computations are done by the same commands as their univariate counterparts - Limit, D, Integrate, and Series.

Recall that Mathematica 11.2 enhanced the functionality of the Limit command to make it easier to do limits; these enhancements include the ability to do different kinds of multivariate limits as well. Recall that for a function of 2 variables  $x$  and  $y$ , the limit  $\lim_{(x,y) \rightarrow (a,b)} f(x,y) = L$  if the values of  $f(x,y)$  go to  $L$  along every possible path in the plane that goes to the point  $(a,b)$  but does not include  $(a,b)$ . In a realistic sense it is harder for these limits to exist than it is for their univariate/"Calculus 1" counterparts as there are many more paths you can take to get to a point in the plane than there are that you can take to get to a point in the real number line - paths can get to  $(a,b)$  from all sorts of directions, by spiraling in, etc - not just from the left or right. As the values of  $f(x,y)$  have to go to  $L$  on *all* the paths, this is asking a lot more for limits in multiple variables than in just one variable. Mathematica 11.2 enhances the Limit command to include both the "full" multivariate limit  $\lim_{(x,y) \rightarrow (a,b)} f(x,y) = L$  as well as "nested" limits where you take the limit first on one variable and then on the other:

Limit[*formula*, { $x,y$ }  $\rightarrow$  { $a,b$ } ]: This is the "full" multivariate limit, taken along all possible paths to get to  $(a,b)$ . If this limit does not exist Mathematica will return the "value" Indeterminate.

Limit[*formula*, { $x \rightarrow a$ ,  $y \rightarrow b$ } ]: This limit computes a "nested" limit in a particular order -  $\lim_{x \rightarrow a} (\lim_{y \rightarrow b} f(x,y))$ . Note that the limit on  $y$  is done *first* (typically giving a function in  $x$ ), and then the limit on  $x$  is done *second*.

Limit[*formula*, { $y \rightarrow b$ ,  $x \rightarrow a$ } ]: This limit computes a "nested" limit in a particular order -  $\lim_{y \rightarrow b} (\lim_{x \rightarrow a} f(x,y))$ . Note that the limit on  $x$  is done *first* (typically giving a function in  $y$ ), and then the limit on  $y$  is done *second*.

If the first/"full" limit exists, then the second and third limits must agree. But it is possible that the "full" limit fails to exist and both nested limits do exist (and can be different). One of the simplest examples of such a limit is to take a look at the behavior of  $f(x,y) = \frac{x^2 - y^2}{x^2 + y^2}$  as  $(x,y) \rightarrow (0,0)$ . Using Mathematica to take the full and nested limits gives us:

```

Limit[ (x^2 - y^2) / (x^2 + y^2), {x, y} → {0, 0}]
Indeterminate

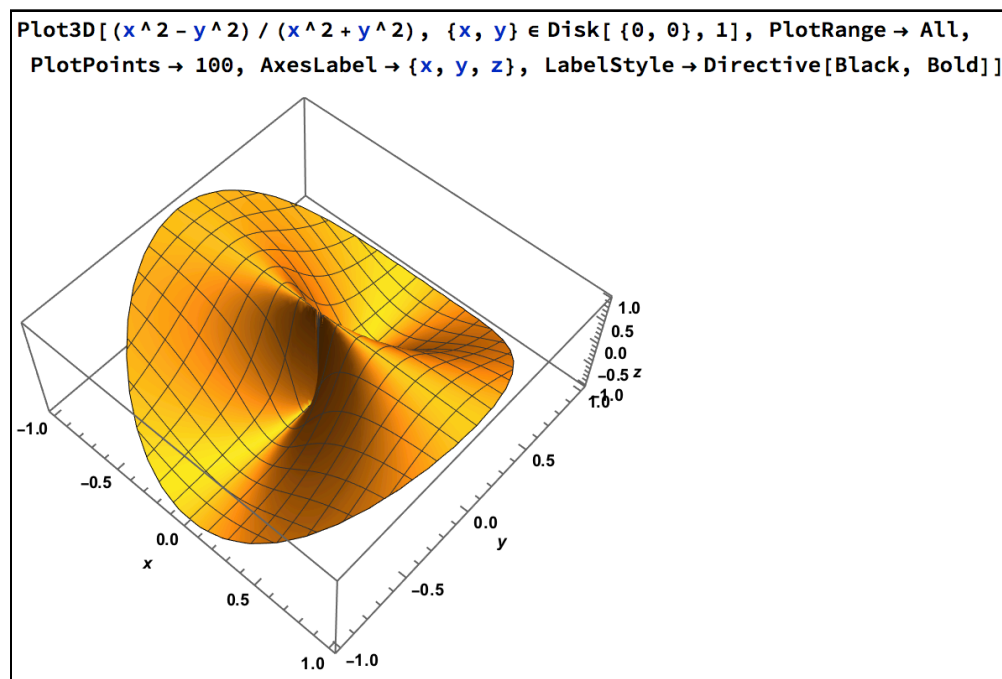
Limit[ (x^2 - y^2) / (x^2 + y^2), {x → 0, y → 0}]
1

Limit[ (x^2 - y^2) / (x^2 + y^2), {y → 0, x → 0}]
-1

```

*different types of multivariate limits*

In this case the “full” multivariate limit fails to exist, and we can see why - the two nested limits (“y first” and then “x first”) do not agree. We can gain additional insight by looking at the graph of the surface around the origin:



*the surface  $f(x, y) = \frac{x^2 - y^2}{x^2 + y^2}$*

In the picture if you let  $x$  go to zero first, you will be looking at paths at the “bottom” of the surface; letting  $y$  go to zero second you will be following the “valley” in the surface, which leads to a value of  $-1$ . But if you let  $y$  go to zero first you will be taking paths along the “bridge” at the top of the surface; then allowing  $x$  to go to  $0$  will follow the bridge to the high value  $+1$ . As different paths give you different results, the full limit does not exist.

These limit ideas (and the Mathematica notation) extend to functions of more variables as well. If you have 3 independent variables  $x$ ,  $y$ , and  $z$ , then `Limit[formula, {x, y, z} → {a, b, c}]` represents the full multivariate limit. `Limit[formula, {x → a, y → b, z → c}]` represents the nested

limit where the limit on  $z$  is done first, then the limit on  $y$ , then the limit on  $x$ . In this case there are 6 possible nested limits.

After multivariate limits, the next common multivariate concept is that of derivatives. For a function  $f$  of many variables  $x, y, \dots$  you have not one derivative but various partial derivatives. For simple partial derivatives we already have the required technique - the `D` command. `D[mess, x]` is the partial derivative of `mess` with respect to  $x$ , `D[mess, y]` is the partial derivative with respect to  $y$ , and so on. The only thing which is missing is the technique for "mixed" partial derivatives (where for example you might take the derivative first with respect to  $x$ , and then with respect to  $y$ ). For mixed partials the command form is simply `D[mess, var1, var2, var3, ...]` for as many of the variables you need - with the understanding that you take the derivatives with respect to the variables at the end of the command first and work your way backward from there. So `D[mess, x, y]` is equivalent to `D[D[mess, y], x]` as you take the derivative with respect to  $y$  first and then  $x$  (recall that mixed partial derivatives don't have to be the same if the order of the derivatives is changed, although they commonly are). The mathematical subscript notation for the same derivative would be  $\text{mess}_{yx}$ , which is the opposite of the way it appears in the `D` command (the `D` command is compatible with the differential operator notation for derivative, however). Here are some examples of higher partial derivatives:

```

mess = x^2 y + 3 y x + Sin[x y] + Cos[x] / y^2;

D[mess, x]

3 y + 2 x y + y Cos[x y] -  $\frac{\text{Sin}[x]}{y^2}$ 

D[mess, y]

3 x + x^2 -  $\frac{2 \text{Cos}[x]}{y^3}$  + x Cos[x y]

D[mess, x, y]

3 + 2 x + Cos[x y] +  $\frac{2 \text{Sin}[x]}{y^3}$  - x y Sin[x y]

D[mess, x, y, x, x, y]

- 6 y Cos[x y] + x^2 y^3 Cos[x y] +  $\frac{6 \text{Sin}[x]}{y^4}$  + 6 x y^2 Sin[x y]

```

*partial derivatives in Mathematica*



As an example of how to use partial derivatives let's find and graph the tangent plane to  $z = x^2 + y^2$  at the point  $(1, -1, 2)$ . The normal vector  $N$  for the tangent plane for  $z = f(x, y)$  is  $\{f_x, f_y, -1\}$ , and the equation of the tangent plane is  $N \cdot (\{x, y, z\} - \{1, -1, 2\}) = 0$ :

```
mess = x^2 + y^2;
generalnormal = {D[mess, x], D[mess, y], -1}

{2 x, 2 y, -1}

ournormal = generalnormal /. {x -> 1, y -> -1}

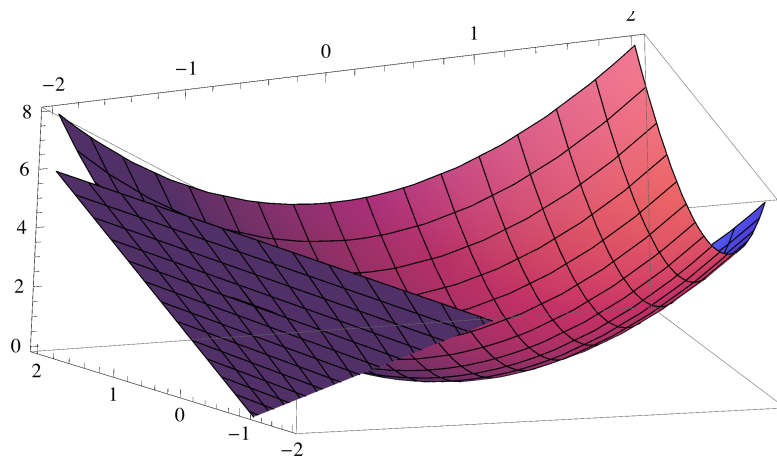
{2, -2, -1}

planeequation = z /. Solve[ournormal.({x, y, z} - {1, -1, 2}) == 0, z][[1]]

2 (-1 + x - y)
```

*finding the formula of the tangent plane*

```
Show[Plot3D[mess, {x, -2, 2}, {y, -2, 2}], Plot3D[planeequation, {x, -2, 2}, {y, -2, 2}]]
```



*the graph and its tangent plane*

We can also do optimization problems for functions of the form  $z = f(x, y)$  following the usual multivariate calculus process rather than the built-in Maximize and Minimize commands. The critical points are where  $f_x = 0$  and  $f_y = 0$ . They are then classified by looking at the quantity  $\Delta = f_{xx}f_{yy} - f_{xy}^2$ ; if  $\Delta > 0$  and  $f_{xx} < 0$  we are at a relative max, and if  $\Delta > 0$  and  $f_{xx} > 0$  we are at a relative min. If  $\Delta < 0$  we are at a saddle point, and if  $\Delta = 0$  no conclusion can be reached. For  $z = 12xy - 4x^2y - 3xy^2$  we have:

```

z = 12 x y - 4 x^2 y - 3 x y^2;
criticalpoints = Solve[ {D[z, x] == 0, D[z, y] == 0}, {x, y}, Reals]

{{x -> 0, y -> 0}, {x -> 0, y -> 4}, {x -> 1, y -> 4/3}, {x -> 3, y -> 0}}

zx = D[z, x]; zy = D[z, y];
zxx = D[z, x, x]; zyy = D[z, y, y]; zxy = D[z, x, y];
delta = zxx * zyy - zxy^2;

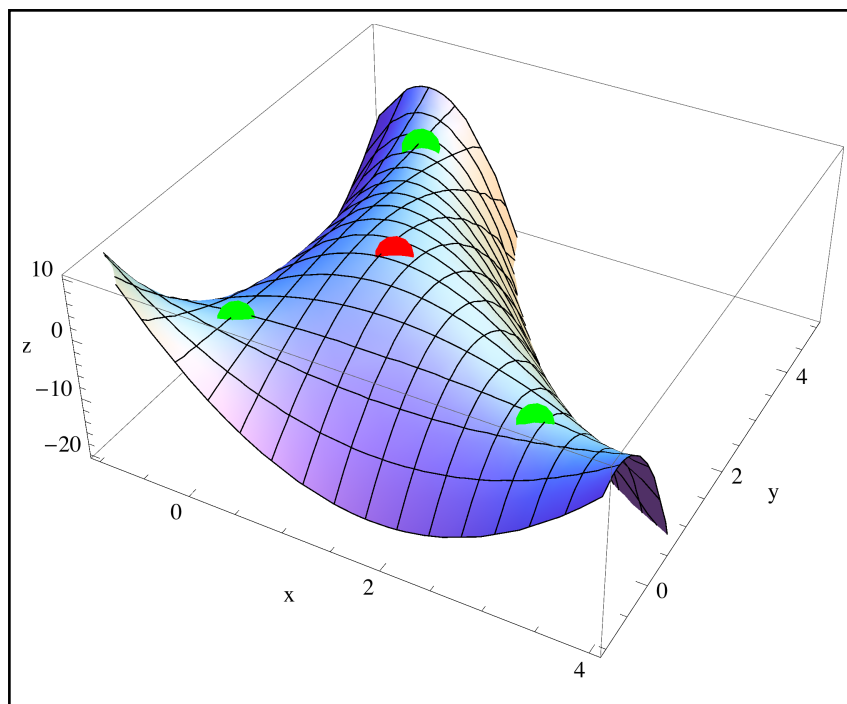
points = { {{x, y, z}}, delta, zxx} /. criticalpoints;
TableForm[points, TableHeadings -> {None, {"Point", "Delta", "z-xx"}}]

```

Point	Delta	z-xx
0 0 0	-144	0
0 4 0	-144	-32
1 4/3 16/3	48	-32/3
3 0 0	-144	0

*information about the critical points on the surface*

Looking at the table we see that the critical points (0,0,0), (0,4,0), and (3,0,0) all have a negative value for  $\Delta$  and are therefore saddle points. The critical point (1, 4/3, 16/3) has a positive value of  $\Delta$  and a negative second derivative - so it corresponds to a local maximum. Using Plot3D we can view the surface to see the saddle and maximum point - for emphasis I've placed a green dot on the saddle points and a red dot at the local maximum:



*the surface and its critical points*

Multiple integration works in a fashion similar to that of mixed derivatives. The multivariate integral  $\int_{x=a}^{x=b} \int_{y=c}^{y=d} f(x,y) dy dx$  is represented by `Integrate[f[x,y], {x,a,b}, {y,c,d}]`

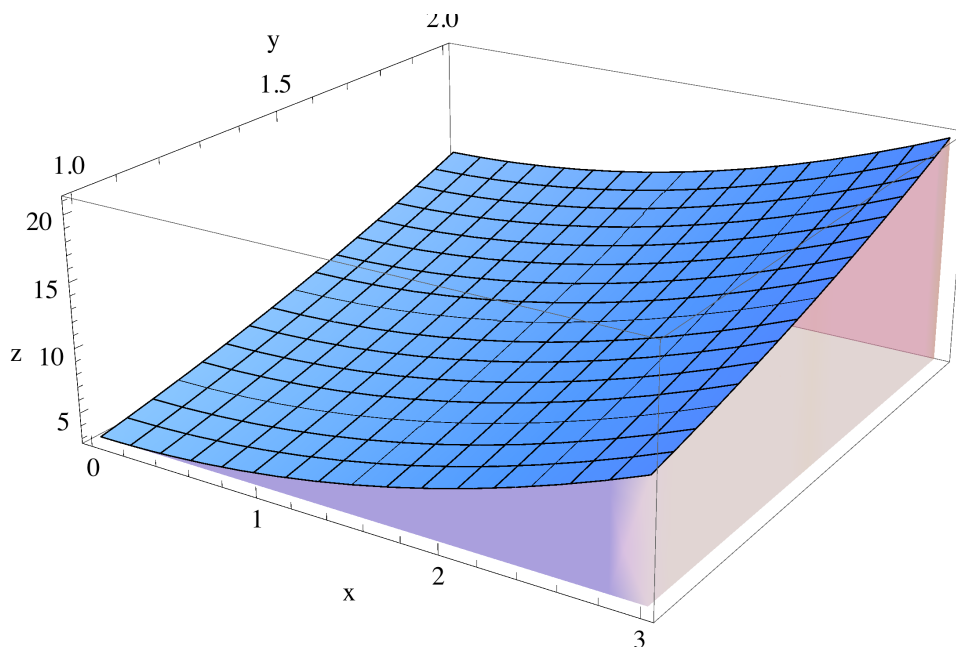
(note that the order of the variable ranges is the same as the order of the integrals), and this extends to integrals involving more than 2 variables. Because the variable ranges in the command have an order to them lists that appear later in the command can involve variables that appear earlier in the command (so `Integrate[f[x,y], {x,0,1},{y,0,1-x}]` represents the integral  $\int_{x=0}^{x=1} \int_{y=0}^{y=1-x} f(x,y) dy dx$ ).

As an example suppose you wanted to find the net volume under the surface  $z = x^2 + 3y^2$  as  $x$  goes from 0 to 3 and as  $y$  goes from 1 to 2. Mathematically this can be represented by  $\int_{x=0}^{x=3} \int_{y=1}^{y=2} x^2 + 3y^2 dy dx$ , which in Mathematica is `Integrate[x^2+3y^2, {x,0,3},{y,1,2}]`:

```
Integrate[x^2 + 3 y^2, {x, 0, 3}, {y, 1, 2}]
```

```
30
```

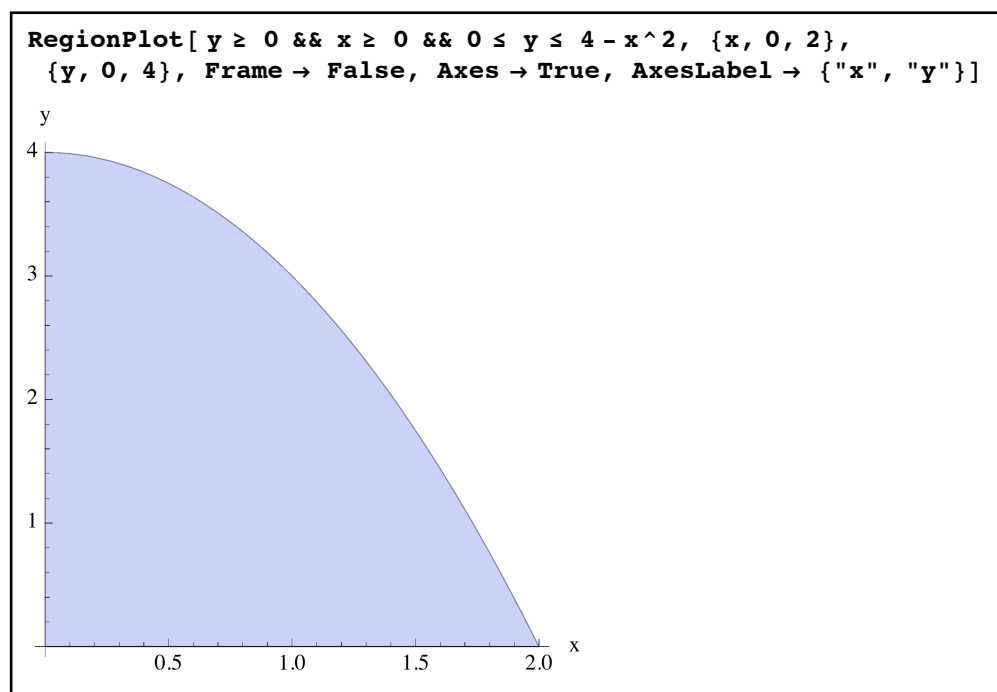
```
Plot3D[x^2 + 3 y^2, {x, 0, 3}, {y, 1, 2}, Filling -> Bottom,  
FillingStyle -> Opacity[.4], AxesLabel -> {"x", "y", "z"}]
```



*the net volume under the surface is 30 cubic units*

Note the use of Filling and FillingStyle to shade in the region under the surface which makes a big difference in the visual representation of the volume.

Suppose that instead of wanting the volume under the surface  $z = x^2 + 3y^2$  over a rectangle we wanted to find the volume under the surface which lay over the Quadrant I region bounded by the axes and the parabola  $y = 4 - x^2$ . This problem is more difficult because the region under the surface is not just a simple rectangle - so the first order of business is to understand the region and its boundary curves. We can use RegionPlot to get a good visualization of the region:



*the region in Quadrant I bounded by the axes and  $y=4-x^2$*

For each value of  $x$  from 0 to 2 this region is bounded below by  $y = 0$  and above by  $y = 4 - x^2$ .

So we can set up the integral for the volume as  $\int_{x=0}^{x=2} \int_{y=0}^{y=4-x^2} x^2 + 3y^2 dy dx$ :

```
Integrate[ x^2 + 3 y^2, {x, 0, 2}, {y, 0, 4 - x^2}]
```

$$\frac{6592}{105}$$

*the volume under the surface is 6592/105 cubic units*

A simpler way to get the same integral is to take advantage of region objects - this lets us find the integral and graph the region with a minimum of work:

```

region = ImplicitRegion[ 0 ≤ y ≤ 4 - x^2 ∧ 0 ≤ x ≤ 2, {x, y}];
Integrate[ x^2 + 3 y^2, {x, y} ∈ region]

```

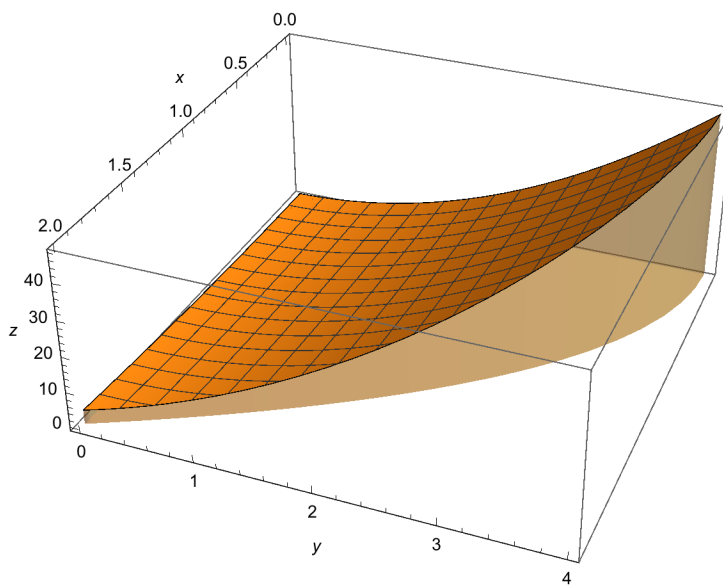
6592

105

```

Plot3D[x^2 + 3 y^2, {x, y} ∈ region, Filling → Axis,
FillingStyle → Opacity[.4], AxesLabel → {x, y, z}]

```



*a quick volume computation and visualization aided by region objects*

In many integral problems like the one above it may be the case that reversing the integration order (that is, integrating with  $x$  as the inner variable and  $y$  as the outer) may make the integral easier (or even possible). The hard part of this is always identifying the boundary curves. The `Reduce` command makes this a lot easier - use `Reduce` on the inequalities that define your region, but instead of ending it with  $\{x,y\}$  end it with  $\{y, x\}$ . For the region we have been using this would look like this:

```

Reduce[ x ≥ 0 && y ≥ 0 && 0 ≤ y ≤ 4 - x^2, {y, x}]

```

```

( 0 ≤ y < 4 && 0 ≤ x ≤ √(4 - y) ) || (y == 4 && x == 0)

```

*using Reduce to help switch the order of integration*

From this we see that for each value of  $y$  from 0 to 4 the variable  $x$  ranges from 0 to  $\sqrt{4-y}$ . So

the same volume can also be represented by 
$$\int_{y=0}^{y=4} \int_{x=0}^{x=\sqrt{4-y}} x^2 + 3y^2 \, dx \, dy :$$

```
Integrate[ x^2 + 3 y^2, {y, 0, 4}, {x, 0, Sqrt[4 - y]}]

6592
-----
105
```

*reversing the order of integration*

Reduce can also be very helpful with dealing with integrals involving 3 or more variables. Suppose you needed to find the integral of the function  $x^2 - y - z$  over the three-dimensional region in octant 1 inside the unit ball  $x^2 + y^2 + z^2 \leq 1$ . We can use Reduce to help us set up the boundaries and the order of integration:

```
Reduce[ x ≥ 0 && y ≥ 0 && z ≥ 0 && x^2 + y^2 + z^2 ≤ 1, {x, y, z}]

0 ≤ x ≤ 1 && ( ( 0 ≤ y < √(1 - x^2) && 0 ≤ z ≤ √(1 - x^2 - y^2) ) || ( y == √(1 - x^2) && z == 0 ) )
```

*using Reduce to find complicated boundaries*

As the part with  $z=0$  is two-dimensional we can ignore it and find that the three-dimensional region can be described as follows: for each value of  $x$  from 0 to 1,  $y$  can be any value from 0 to  $\sqrt{1 - x^2}$  and for those choices of  $x$  and  $y$  the variable  $z$  can be anything from 0 to  $\sqrt{1 - x^2 - y^2}$ .

This means our integral can be set up as  $\int_{x=0}^1 \int_{y=0}^{\sqrt{1-x^2}} \int_{z=0}^{\sqrt{1-x^2-y^2}} x^2 - y - z \, dz \, dy \, dx$ :

```
Integrate[ x^2 - y - z, {x, 0, 1}, {y, 0, Sqrt[1 - x^2]}, {z, 0, Sqrt[1 - x^2 - y^2]}]

11 π
-----
120
```

*finding the value of a complicated multivariate integral*

Not all regions and integrals will break down quite so nicely of course but using Reduce is often a huge help when trying to find boundaries and orders of integrations for complicated multivariate integrals. There will be times when using region objects won't work (as the integrals may not be computable in the standard integration order) but the "manual" approach to entering the boundary objects will.

Although series do not play the same role in multivariate calculus as they do in univariate calculus, Mathematica can compute series in more than 1 variable. Series[mess, {x, x0, n}, {y, y0, m}] will find the series for mess at the point (x0, y0) out to the power  $(x - x0)^n (y - y0)^m$  by first finding the series in terms of  $y$  and then finding series for the resulting coefficients in

terms of  $x$ . This generally results in several different "O" error terms. As in the previous cases we worked with series the error terms can be removed with the use of Normal and series which are centered at the same point can be added, subtracted, multiplied, and composed.

Here is how to find the series for  $x \sin(x + y) \sin(x+y)$  at the point  $(0,\pi)$ , out to the fourth degree  $x$  terms and third degree  $y$ -terms:

**Series[  $x \sin[x + y]$ , { $x$ , 0, 4}, { $y$ ,  $\pi$ , 3}]**

$$\left( - (y - \pi) + \frac{1}{6} (y - \pi)^3 + O[y - \pi]^4 \right) x + \left( -1 + \frac{1}{2} (y - \pi)^2 + O[y - \pi]^4 \right) x^2 +$$

$$\left( \frac{y - \pi}{2} - \frac{1}{12} (y - \pi)^3 + O[y - \pi]^4 \right) x^3 + \left( \frac{1}{6} - \frac{1}{12} (y - \pi)^2 + O[y - \pi]^4 \right) x^4 + O[x]^5$$

**Normal [%]**

$$-x^2 + \frac{x^4}{6} + \left( -x + \frac{x^3}{2} \right) (-\pi + y) + \left( \frac{x^2}{2} - \frac{x^4}{12} \right) (-\pi + y)^2 + \left( \frac{x}{6} - \frac{x^3}{12} \right) (-\pi + y)^3$$

*finding a series for a multivariate function*

## Section 5.8 Homework - Calculus Computations for Three (and More) Dimensions

- Find  $\lim_{x,y \rightarrow (1,3)} x^3 - 10y + xy$ .
- Show that  $\lim_{x,y \rightarrow (0,0)} \frac{x - y}{x + y}$  does not exist. Compute the two nested limits. Graph the surface  $z = \frac{x - y}{x + y}$  on the disk of radius 1 centered at the origin and use it to explain the nested limits. Does the graph of the surface show that other paths as  $(x, y) \rightarrow (0,0)$  can give different answers as well?
- Show that  $\lim_{(x,y,z) \rightarrow (0,0,0)} \frac{x + y}{y + z}$  does not exist. Find all 6 nested limits.
- The Limit command is more flexible than presented here. Try evaluating the command `Limit[(x^2-y^2)/(x^2+y^2), {x->0, y->k x}]` for several different values of  $k$ . What do you think the "meaning" of this tweaking of the limit notation is?
- If  $f(x, y) = \cos\left(\frac{x}{y}\right)$ , find  $f_{xyx}$  and  $f_{xyy}$ .
- If  $f(x, y, z) = x^3 e^{y-5z}$  find  $f_{xxy}, f_{xyz}$ , and  $f_{xyyz}$ .
- Using the techniques described in this section, find the equation of the tangent plane to  $z = xy$  at  $(1,-1)$ . Graph the surface and plane together.
- Using the techniques described in this section, find the tangent plane to  $z = \cos(x^2 + y^2)$  at  $(0,0)$ . Graph the surface and plane together.
- Find and classify the critical points of  $z = xy - x - y$ . Graph the surface and these points.
- Find and classify the critical points of  $z = x^3 - 3xy^2 + y^3$ . Graph the surface and the points.

- 11) Find and classify the critical points of  $z = \cos(x) + \cos(y) - \sin(x + y)$  where  $x$  and  $y$  range from 0 to  $2\pi$ . Graph the surface and the points.
- 12) Find the volume under  $z = x^2 + 5 \sin(xy)$  as  $x$  goes from 1 to 2 and  $y$  goes from 4 to 5. Estimate this to 6 places.
- 13) Find the net volume under  $z = x^3 + y^3$  as  $x$  and  $y$  go from -1 to 1.
- 14) Repeat problem 9, except find the volume over the unit circle. Do this both without and with region objects.
- 15) Repeat problem 9, except find the volume over the triangle whose corners are (0,0), (1,0), and (1,1). Do this both with and without region objects.
- 16) Repeat problem 9, except find the volume over the region bounded by  $y = x^2$  and  $y = x + 12$ . Try this problem using both orders of integration (so no region objects allowed).
- 17) Find the series for  $\sin(x + y^2)$  centered at the origin out to 4 powers of  $x$  and 6 powers of  $y$ . Use Normal and Expand on the result.



## Section 5.9 - Differential Equations

The last main area of calculus that we will discuss is the solution of differential equations. There are four principal commands for this - DSolve, DSolveValue, NDSolve, and NDSolveValue. The “N” commands are for numerical solutions to differential equations as opposed to the exact solutions. The “Value” commands give their results as straight answers/formulas as opposed to the “pure” Solve commands which return their answers as replacement rules.

Both DSolve and DSolveValue have 3 principal forms depending on whether you are trying to solve a single differential equation, a single ordinary differential equation with boundary conditions, or a system of differential equations:

DSolve[ *equation*,  $y[x]$ ,  $x$ ]: Attempt to solve the given differential equation for the function  $y[x]$  in terms of  $x$ . The solution is given as a set of replacement rules (DSolveValue will give the answer as a straight function).

DSolve[ *equationandconditionlist*,  $y[x]$ ,  $x$ ]: Attempt to solve the given differential equations and boundary conditions for  $y[x]$  in terms of  $x$ . The solution is given as a set of replacement rules (DSolveValue will give the answer as a straight function).

DSolve[ *equationsandconditionslist*, {  $y1[x]$ ,  $y2[x]$ , ... },  $x$ ]: Attempt to solve the given system of differential equations and boundary conditions for the functions  $y1[x]$ ,  $y2[x]$ , ... in terms of  $x$ . The solution is given as a set of replacement rules (DSolveValue will give the answer as a list of functions).

Before looking at some examples of differential equations it is worth taking a moment to describe both the structure of the inputs for DSolve and some of the difficulties with the solutions themselves. All derivatives have to be given using either the D notation or the ' notation ( $y'[x]$ ,  $y''[x]$ , and so on - make sure you haven't used the variable  $y$  previously or Clear it if you have). If you have a partial differential equation using the D command to define the derivatives is going to be necessary (say  $D[y[x,t],t]$  to represent the partial derivative  $y_t$ ). The equations and boundary conditions are given using the == notation just as we do for the Solve command ( $y''[x]==2y[x]-5$ ,  $y[0]==5$ ,  $y'[0]==-4$ , and so on). And because differential equations are closely related to integration you can expect to see all sorts of unusual functions in their solutions (and many equations which cannot be solved exactly). You may even see the Function command, which is Mathematica's internal representation of the “pure function” notation. With that said let's take a look at some examples of the DSolve and DSolveValue command which don't use boundary conditions:

```

DSolve[ y' [x] == x y[x]^2, y[x], x]


$$\left\{ \left\{ y[x] \rightarrow -\frac{2}{x^2 + 2 C[1]} \right\} \right\}$$


y[x] /. %[[1]]


$$-\frac{2}{x^2 + 2 C[1]}$$


DSolveValue[ y' [x] == x y[x]^2, y[x], x]


$$-\frac{2}{x^2 + 2 C[1]}$$


DSolve[ D[ y[x, t], t] == (x - t) D[y[x, t], t, t], y[x, t], {x, t}]


$$\{ \{ y[x, t] \rightarrow -\text{Log}[-t + x] C[1][x] + C[2][x] \} \}$$


```

*finding “general” solutions to differential equations with DSolve and NDSolve*

As long as there is a unique solution DSolveValue and y[x]/. DSolve[...][[1]] are more or less the same. DSolveValue presumes a unique solution so if there is more than one possible answer DSolve is the better command to use:

```

DSolveValue[ y' [x] == x / y[x]^2, y[x], x]

DSolveValue::dsvb : There are multiple solution branches for the equations,
but DSolveValue will return only one. Use DSolve to get all of the solution branches. >>


$$-\left(-\frac{3}{2}\right)^{1/3} \left(x^2 + 2 C[1]\right)^{1/3}$$


DSolve[ y' [x] == x / y[x]^2, y[x], x]


$$\left\{ \left\{ y[x] \rightarrow -\left(-\frac{3}{2}\right)^{1/3} \left(x^2 + 2 C[1]\right)^{1/3} \right\}, \right.$$


$$\left. \left\{ y[x] \rightarrow \left(\frac{3}{2}\right)^{1/3} \left(x^2 + 2 C[1]\right)^{1/3} \right\}, \left\{ y[x] \rightarrow (-1)^{2/3} \left(\frac{3}{2}\right)^{1/3} \left(x^2 + 2 C[1]\right)^{1/3} \right\} \right\}$$


```

*DSolve allows for multiple solutions; DSolveValue only gives one*

Note that as we have not specified any boundary conditions all of the solutions involve arbitrary constants C[1], C[2], etc. Also worth noting is the appearance of complex numbers in the last set of solutions.

We can easily add in boundary conditions as well. To solve the boundary value problem  $y'' - 4y' + 5y = t, y(0) = 3, y'(0) = 5$ , we simply enter the following:

```

DSolve[ {y''[t] - 4 y'[t] + 5 y[t] == t, y[0] == 3, y'[0] == 5}, y[t], t]
{ {y[t] -> 1/25 (4 + 5 t + 71 e^2 t Cos[t] - 22 e^2 t Sin[t])} }
DSolveValue[ {y''[t] - 4 y'[t] + 5 y[t] == t, y[0] == 3, y'[0] == 5}, y[t], t]
1/25 (4 + 5 t + 71 e^2 t Cos[t] - 22 e^2 t Sin[t])

```

*solving an ODE with boundary conditions*

Differential equations with boundary conditions tend to have unique solutions which makes the simpler results in DSolveValue a bit more attractive than DSolve.

Working with simple systems of differential equations often yield very complicated solutions, even if the individual equations are simple. For example suppose you had the system of differential equations

$$y_1''(y) + 2y_1(t) = -y_2(t), y_2''(t) + y_2(t) = y_1(t), y_1(0) = 0, y_1'(0) = 2, y_2(0) = -2, y_2'(0) = 0$$

Entering this in via DSolveValue gives the following unpleasant (but accurate!) result:

```

DSolveValue[{y1''[t] + 2 y1[t] == -y2[t], y2''[t] + y2[t] == y1[t],
  y1[0] == 0, y1'[0] == 2, y2[0] == -1, y2'[0] == 0}, {y1[t], y2[t]}, t]
{ 1/2 (RootSum[3 + 3 #1^2 + #1^4 &, (e^t #1)/(3 + 2 #1^2) &] + 2 RootSum[3 + 3 #1^2 + #1^4 &, (e^t #1 + e^t #1 #1^2)/(3 #1 + 2 #1^3) &]),
  1/2 (-RootSum[3 + 3 #1^2 + #1^4 &, (2 e^t #1 + e^t #1 #1^2)/(3 + 2 #1^2) &] +
  2 RootSum[3 + 3 #1^2 + #1^4 &, (e^t #1)/(3 #1 + 2 #1^3) &]) }

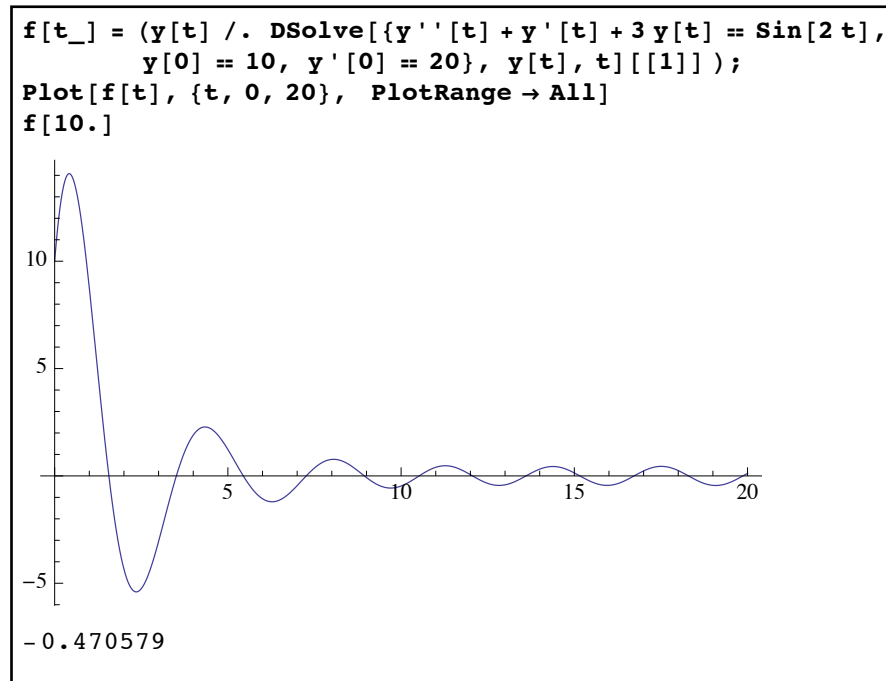
```

*systems of simple differential equations can have complicated solutions*

In this solution RootSum[*polynomial, function*] stands for “add up the values of *function* where the underlying variable is evaluated at each root of *polynomial*”. So the expression RootSum[#1^2 - 3#1 &, #1^6 &] means “add up the values of  $x^6$ , where  $x$  can be any of the roots of  $x^2 - 3x$ ”. So in this solution each individual RootSum stands for the sum of 4 terms, each which is a root of  $x^4 + 3x^2 + 3$ . It is very common to see RootSum and more complicated functions show up in the solution to both systems of differential equations and a single differential equation of high order.

If you want to graph the solution to a differential equation you either need to have the function given via DSolveValue or you’ll have to convert the “replacement rule” form of a DSolve solution to something Plot and similar commands can handle. One common way to do

this is to define a function which is the solution and then simply plot the function. For example suppose you wanted to plot the solution to the initial value problem  $y'' + 3y' = \sin(2t)$ ,  $y(0) = 10$ ,  $y'(0) = 20$  (this problem could represent a mass-spring system that has both friction accounted for and an outside forcing function). The solution to a DSolve command is going to be a list of replacement rule lists, so we can use the replacement notation /. along with the [[1]] notation to define a function for plotting (alternatively we could use DSolveValue):



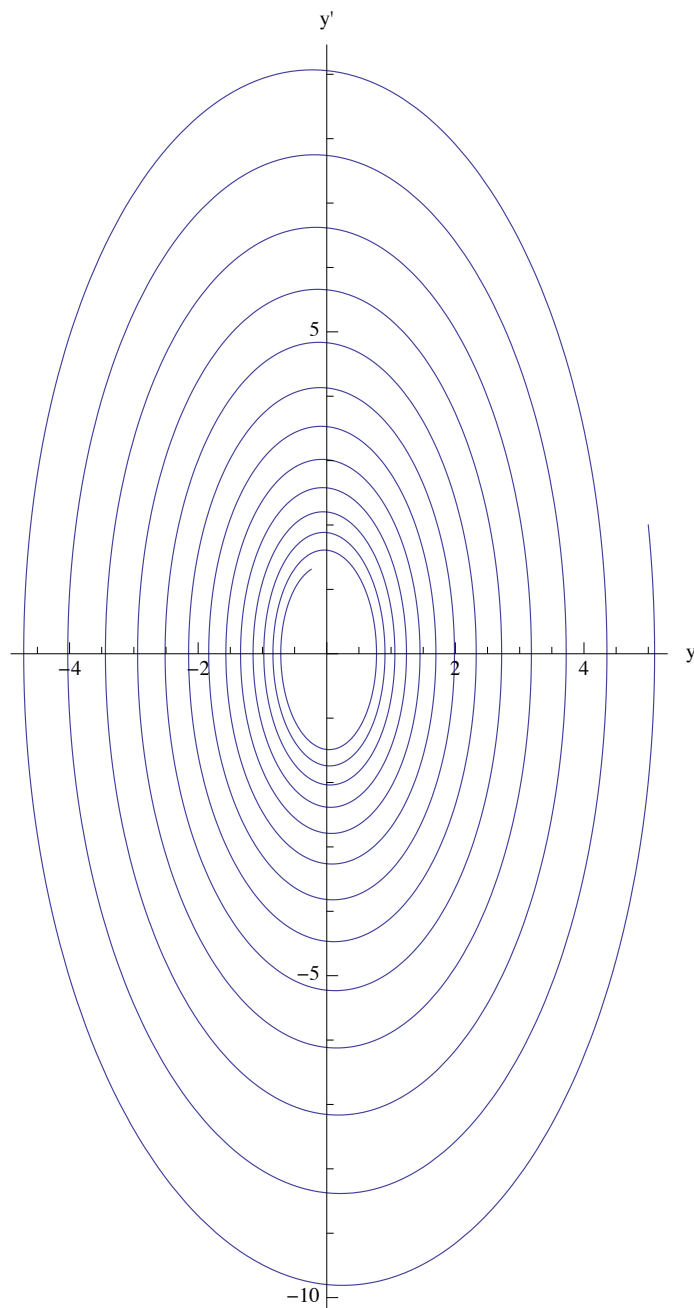
*graphing the solution to a boundary value problem*

You may have noticed that in the command that defines  $f[t]$  we have used the  $=$  sign rather than the  $:=$  sign as we usually do for functions. The “immediate evaluation” used in the  $=$  sign means that the differential equation is solved one time only and its solution is stored in  $f[t]$ . If we had used  $:=$  in the definition of  $f[t]$  then the differential equation would have to be resolved every time the function is used - which would be a lot slower (and cause problems for the Plot command as well). We could have sidestepped the definition of  $f[t]$  as well, simply evaluating  $y[t] /. DSolve...$  as one line and then using  $Plot[\%, \{t, 0, 20\}, PlotRange \rightarrow All]$ . However in many applications you will want to take the solution to the differential equation and plug in a value like  $t = 10$  and having the solution defined as a function makes this very easy to do.

Another common type of plot for second-order differential equations is to use the “phase plane”. Rather than graph the solution  $y(t)$  in terms of  $t$  as we normally would we parametrically graph the point  $(y(t), y'(t))$  (this is useful in many applications as both  $y$  and  $y'$  may have some kind of meaning - for example in the differential equation for a mass-spring system  $y$  would be the position of the object and  $y'$  the velocity) For example, suppose we took the differential

equation  $y'' + \frac{1}{10}y' + 4y = 0, y(0) = 5, y'(0) = 2$ . We can graph this by defining a function  $f[t]$  to represent  $y$ , and then doing a `ParametricPlot` for the point  $\{f[t], f'[t]\}$ :

```
f[t_] = y[t] /. DSolve[{y''[t] + y'[t]/10 + 4 y[t] == 0,
  y[0] == 5, y'[0] == 2}, y[t], t][[1]];
ParametricPlot[{f[t], f'[t]}, {t, 0, 40}, AxesLabel -> {"y", "y'"}]
```



*the phase plane for our solution from time 0 to time 40*

The graph in the phase plane “starts” at the point (5,2) (this corresponds to our initial conditions  $y(0)=5, y'(0)=2$ ) and then spirals inward towards the origin. This makes sense if we interpret the differential equation as representing a physical model. The equation  $y'' + \frac{1}{10}y' + 4y = 0$  could represent a mass-spring system with a mass of 1 unit, a friction unit of  $1/10$ , and a spring constant of 4. The boundary conditions would then translate to the fact that at time 0, the mass is 5 units out from the rest position and given an initial velocity of 2 units. Since there is a friction term we would expect the mass to bounce back and forth less each time and do so more slowly - this would indicate that both  $y$  (the position) and  $y'$  (the velocity) to decrease over time, which is consistent with the spiral in towards the origin (which would represent the mass at complete rest).

The plot above requires that we are able to find an explicit formula for the solution  $y$  even if it is not shown (if we don't have a formula for  $y$ , it is going to be hard to find  $y'$ ). As the complexity and order of a differential equation rises it may not be possible to find an exact solution. For example, consider the equation  $(y')^2 + y^2 = 1 + \sin(x)$ ,  $y(0) = \frac{1}{2}$ ,  $y'(0) = \frac{1}{3}$ :


```
DSolve[{y'[x]^2 + y[x]^2 == 1 + Sin[x], y[0] == 1/2, y'[0] == 1/3}, y[x], x]
DSolve[{y[x]^2 + y''[x]^2 == 1 + Sin[x], y[0] == 1/2, y'[0] == 1/3}, y[x], x]
```


*not terribly helpful*

In this case Mathematica is unable to find the solution exactly so it returns the DSolve command unevaluated. This happens in practice quite often (and parallels problems we have seen previously with the Solve and Integrate commands). If an exact solution is not possible in many cases you can get a numerical estimate for the solution through the command NDSolve (or NDSolveValue). NDSolve and NDSolveValue work a little differently than DSolve. Both NDSolve and NDSolveValue require that you have boundary conditions along with your equations (the boundary conditions form the starting point for the estimation). In addition both commands require that you give a range of values for the independent variable (it is not enough to say “x”, you need to specify something like {x,0,20}). The estimated solution needs to be fairly accurate and to guarantee that accuracy Mathematica needs to know what range of values you will be using for “x”. The range of values you use in DSolve must include the value used by the boundary conditions and should in practice be as limited as possible (the wider the range the harder Mathematica has to work to be accurate over the entire range). And finally the results from NDSolve and NDSolveValue are not normal functions but rather a type of expression called an InterpolatingFunction. Without going into the details of how InterpolatingFunction works suffice it to say that an InterpolatingFunction more or less represented an “estimated” function and it can be used in place of a “normal” formula when defining functions or making a graph. Redoing the differential equation  $(y')^2 + y^2 = 1 + \sin(x)$ ,  $y(0) = \frac{1}{2}$ ,  $y'(0) = \frac{1}{3}$  with NDSolve on the interval [0,10] yields:

```

sols = NDSolve[
  {y''[x]^2 + y[x]^2 == 1 + Sin[x], y[0] == 1/2, y'[0] == 1/3, y[x], {x, 0, 10}}
NDSolve::mxst : Maximum number of 482822 steps reached at the point x == 2.189123600937721'. >>
NDSolve::mxst : Maximum number of 482822 steps reached at the point x == 1.1280485329797558'. >>

{ {y[x] -> InterpolatingFunction[
   Domain: {{0., 2.19}} Output: scalar ][x] },

{y[x] -> InterpolatingFunction[
   Domain: {{0., 1.13}} Output: scalar ][x] } }


```


*using NDSolve on a difficult boundary value problem*

Here NDSolve is telling us it has found 2 solutions - one which is valid from 0 to 2.19, and another which is valid from 0 to 1.13 (note that even though we gave it the range  $\{x, 0, 10\}$  it was only able to guarantee the accuracy on smaller ranges). We can define the 2 solutions as  $f[x]$  and  $g[x]$  and then either plot them or get estimates for specific values of  $x$ :

```

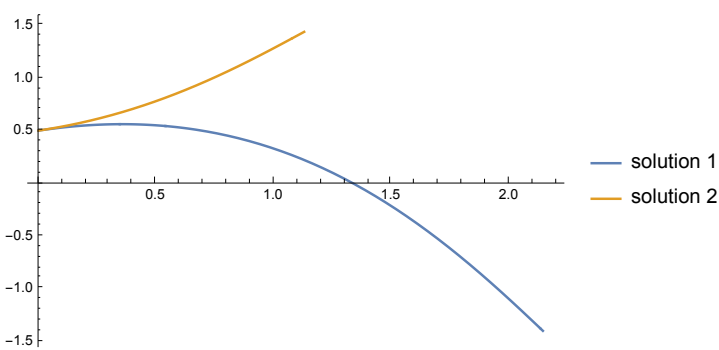
f[x_] = y[x] /. sols[[1]]
g[x_] = y[x] /. sols[[2]]

InterpolatingFunction[ Domain: {{0., 2.19}} Output: scalar ][x]

InterpolatingFunction[ Domain: {{0., 1.13}} Output: scalar ][x]

Plot[{f[x], g[x]}, {x, 0, 2.19}, PlotLegends -> {"solution 1", "solution 2"}]

```



```

f[1]
0.32802 + 0. i

g[.3]
0.640415 + 0. i

```

*the two estimated solutions to our boundary value problem and some of their values*

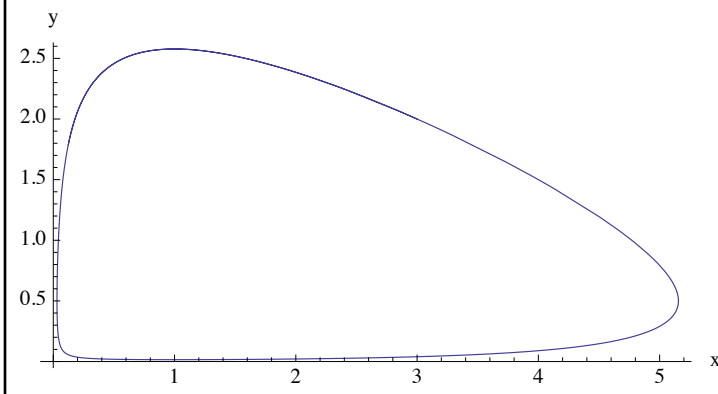
We can use NDSolve with systems of differential equations as well - we just need to modify the inputs the same way we did for DSolve. For example suppose we had the system  $x' = x - 2xy$ ,  $y' = -y + xy$ ,  $x(0) = 3$ ,  $y(0) = 2$  (again, here both  $x$  and  $y$  depend on a third variable  $t$ ). We can estimate a solution to this system as  $t$  goes from say 0 to 20 as follows:

```
sols = NDSolve[ {x'[t] == x[t] - 2 x[t] y[t],  
  y'[t] == -y[t] + x[t] y[t], y[0] == 2, x[0] == 3}, {x[t], y[t]}, {t, 0, 20}]  
{ {x[t] → InterpolatingFunction[{{0., 20.}}, <>][t],  
  y[t] → InterpolatingFunction[{{0., 20.}}, <>][t] }
```

*the approximate solution to the system of equations, stored in "sols"*

We can get values for both  $x$  and  $y$  by defining functions for them just as we did before (in this case we would define one function for  $x$  and another one for  $y$ ). We can even graph them against each other using ParametricPlot:

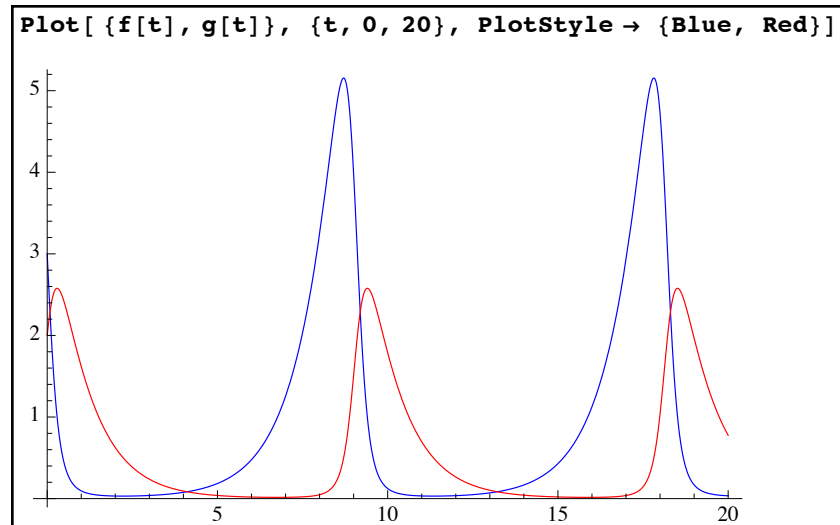
```
f[t_] = x[t] /. sols[[1]];  
g[t_] = y[t] /. sols[[1]];  
ParametricPlot[ {f[t], g[t]}, {t, 0, 10}, AxesLabel → {"x", "y"}]
```



*the graph of our solutions against each other rather than as over time*

As the parametric plot seems to form a closed loop it would appear that  $x$  and  $y$  repeat over and over again, which we can verify by plotting them normally versus the variable  $t$ :

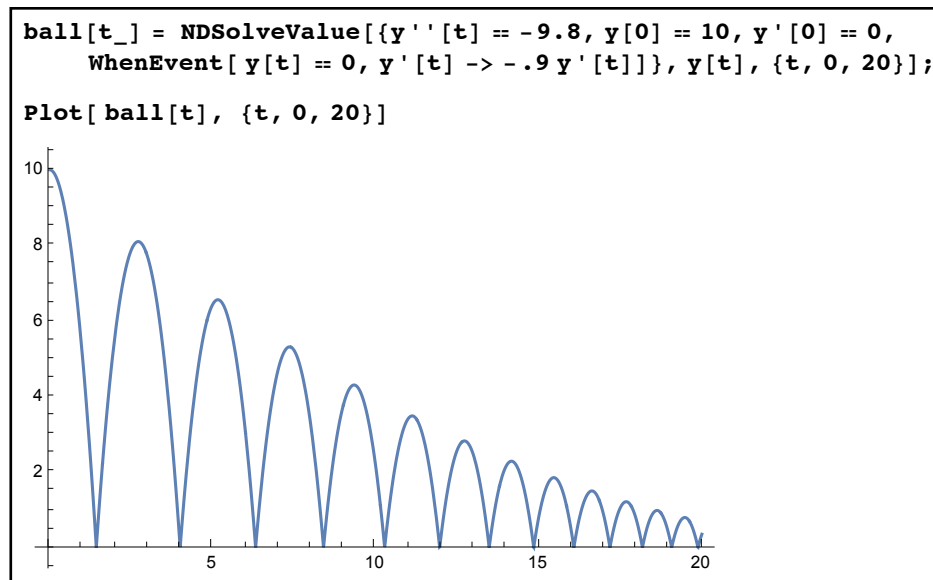




*the graphs of  $x$  and  $y$  over time  $t$ , which appear to be periodic*

Those who have studied differential equations will recognize this system as a predator-prey model where the population of predators ( $x$  in this case, graphed in blue), follows the population of prey ( $y$ , graphed in red) with a time lag between the two population's growth and collapse.

In Mathematica 9 a special command was introduced to help with a special kind boundary condition in `NDSolve` and `NDSolveValue` - `WhenEvent`. `WhenEvent` specifies a boundary condition of the form “when this condition is met perform the following action” via `WhenEvent[ condition, action as a replacement rule or list of rules]`. A great example of this is a falling ball. As long as a ball is in free fall (and we simplify things by ignoring air resistance, etc.) the only acceleration on the object is due to gravity. In the metric system the acceleration due to gravity is roughly 9.8 meter/sec<sup>2</sup> downward, so we could say  $a = -9.8$ . But as acceleration is the derivative of velocity we can write this as  $v'[t] = -9.8$  (or even better as  $y''[t] = -9.8$ , where  $y[t]$  is the height of the ball and  $y'[t]$  is the velocity). This only holds as long as the ball is in free fall though - when the ball hits the ground (that is when  $y[t] = 0$ ) it bounces back, say with 90% of its original velocity. In this case we have a triggered event (with trigger  $y[t] = 0$ ) which changes the velocity to  $-0.9$  of its “pre-trigger” value ( $0.9$  for 90%, the minus sign for a direction change). This would be the boundary condition `WhenEvent[ y[t]==0, y'[t]→-.9 y'[t] ]`. If we assume the ball was dropped from a height of 10 meters with no initial velocity we could model this over the first 20 seconds as `NDSolveValue[ {y''[t]==-9.8, y[0]==10, y'[0]==0, WhenEvent[ y[t]==0, y'[t]→-.9 y'[t] ]}, y[t], {t,0,20}]`:



*a model of a bouncing ball using WhenEvent*

The WhenEvent condition is very useful in modeling complex systems - modeling the height of the ball would have been much more complicated just a few years ago, requiring either programming specific to this problem or setting up and solving a new differential equation at the velocity discontinuity created by each bounce.

Although these examples have worked without too many errors popping up the numerical solution of differential equations can be very difficult. In many cases you will get no solutions or multiple solutions. Some solutions may involve complex numbers (which obviously don't make sense in some physical models and graphs). And the estimation process itself may break down if you use an overly large range for the independent variable or at singularities. NDSolve has options which you can alter to try to handle some of these situations, but they are beyond the scope of our discussion.

All of "DSolve" commands presented here also work with "partial differential equations" - that is, similar problems where the function has more than one independent variable and the differential equation involves partial derivatives. These types of problems get very complicated (the "boundary conditions" often specify what happens along an entire boundary rather than just a single point) and almost always require numerical solutions. In some cases rather than using the D command to get partial derivatives the full Derivative command (which we have not introduced) can be used for simplicity. While these problems are beyond the scope of our course, here is an example to illustrate the basic idea:

Suppose you had a string 10 inches long which is pinned down at both ends, and the material from which it is made makes the natural speed of a wave in the string 1/2 inch per second. In addition suppose that the string is initially forced into the shape

$f(x) = \frac{x(10-x)(x-3)}{40}$  (which is 0 at both  $x=0$  and  $x=10$ ) and then let go. If  $y(x, t)$  is the height of the vibrating string over the point  $x$  at time  $t$ , then  $y$  satisfies the following:

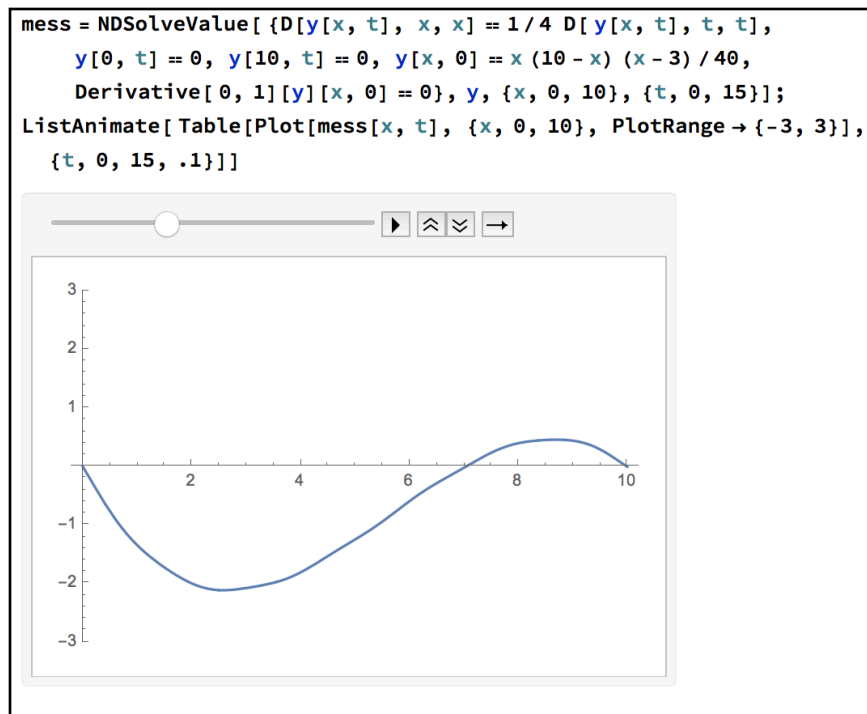
$$y_{xx}(x, t) = \frac{1}{4} y_{tt}(x, t) \text{ (this is the "wave equation" with speed } 1/2\text{)}$$

$$y(0, t) = 0 \text{ and } y(10, t) = 0 \text{ (the string is pinned at the ends)}$$

$$y_t(x, 0) = 0 \text{ (the string is initially motionless)}$$

$$y(x, 0) = \frac{x(10-x)(x-3)}{40} \text{ (initial string shape)}$$

To get a numerical solution for  $y$  over the length of the string and the first 15 seconds of motion, you could use the command `mess=NDSolveValue[{D[y[x,t],x,x]==1/4 D[y[x,t],t,t], y[0,t]==0, y[10,t]==0, Derivative[0,1][y][x,0]==0, y[x,0]==x(10-x)(x-3)/40}, y, {x,0,10}, {t,0,15}];`. You could then animate the behavior of the string over time using the command `ListAnimate[Table[mess[x,t], {x,0,10}, PlotRange->{-3,3}], {t,0,15,.1}];`.



*animating a vibrating string over time*

## Section 5.9 Homework - Differential Equations

- 1) Solve the differential equation  $y'' + 6y' + 10y = e^t$ .
- 2) Solve the differential equation  $y'' + xy' - y = 0$ .
- 3) Solve the initial value problem  $y' = \frac{5x}{y^2}, y(1) = 1$ .
- 4) Solve the differential equation  $y'' + 3y' + y = t^2, y(0) = 0, y'(0) = 4$ .
- 5) Graph the solution to problem 4 both normally and in the phase plane.
- 6) Graph the solution to  $y'' + y' - xy = 0, y(0) = 1, y'(0) = 1$ .
- 7) Graph the solution to  $y''' + y = 0, y(0) = 0, y'(0) = 1, y''(0) = 0$ .
- 8) Graph the solutions to  $y' = x - 3y, x' = y - x, y(0) = 1, x(0) = 1$ , where  $t$  is the underlying variable for  $x$  and  $y$ .
- 9) Graph the solutions to problem 8 in the phase plane.
- 10) Use NDSolve to graph the solution to  $y'' + (y')^2 = y, y(1) = 1, y'(1) = 1$  over the range  $[0, 10]$ .
- 11) Repeat problem 10 over the range  $[-10, 10]$ . What happens?
- 12) Use NDSolve to graph the solution to  $y'' + \sin(y) = 0, y(0) = 1, y'(0) = 0$  from 0 to 100 in blue.
- 13) For “small” values of  $y$ ,  $\sin(y)$  is approximately the same as  $y$ . So for “small” values of  $y$ , the equation in problem 12 can be approximated by the equation  $y'' + y = 0, y(0) = 1, y'(0) = 0$ . Graph this solution in red together with the solution from problem 12. Do they look the same?
- 14) Repeat problems 12 and 13, but replace  $y(0)=1$  with  $y(0)=1/10$ . Do they look the same?
- 15) Repeat problems 12-13 but graph both solutions in phase space instead of as functions of  $t$ .
- 16) Solve the differential equation  $y'' + \frac{g}{l}y = 0$ , where  $g$  and  $l$  are (positive) constants.

Note: The differential equation in problem 12 is the equation of a pendulum whose arm length is equal to the gravitational acceleration, lifted 1 radian from the rest position, and let go. It is not solvable exactly, so it is often replaced with the differential equation in 13. The approximation is pretty close if  $|y| < .1$ , which is why the graphs in problem 14 look much closer than the ones in problem 13.