Sept. 16 2010

Lecture 5: Introduction to Mathematica IV

Graphics

Graphics are an important part of exploring mathematics and conveying its results. An informative plot or graphic that conveys a complex idea succinctly and naturally to an educated observer is a work of creative art. Indeed, art is sometimes defined as "an elevated means of communication," or "the means to inspire an observation, heretofore unnoticed, in another." Graphics are art; they are necessary. And, I think they are fun.

For graphics, we are limited to two and three dimensions, but, with the added possibility of animation, sound, and perhaps other sensory input in advanced environments, it is possible to usefully visualize more than three dimensions. Mathematics is not limited to a small number of dimensions; so, a challenge —or perhaps an opportunity—exists to use artfulness to convey higher dimensional ideas graphically.

The introduction to basic graphics starts with two-dimensional plots.

Lecture 05 MATHEMATICA® Example 1

Simple Plots

Download notebooks, pdf(color), pdf(bw), or html from http://pruffle.mit.edu/3.016-2011.

Here are some examples of simple x-y plots and how to decorate them. We start with very simple examples and add a little more at each step to show how a plot can be developed incrementally. We leave all the steps in as cut-and-paste examples.

```
Plot[Sin[x] / x, {x, -5 Pi, 5 Pi}]                                              1

Options[Plot]                                                                   2

Plot[Sin[x] / x, {x, -5 Pi, 5 Pi},
 PlotRange → {-0.25, 1.25},
 PlotStyle → {Red, Thick}]                                                      3

Plot[Sin[x] / x, {x, -5 Pi, 5 Pi},
 PlotRange → {-0.25, 1.25},
 PlotStyle → {Red, Thick},
 AxesLabel → {"x", "Sin (x) / x"}]                                              4

Plot[Sin[x] / x, {x, -5 Pi, 5 Pi},
 PlotRange → {-0.25, 1.25},
 PlotStyle → {Red, Thick},
 AxesLabel → {"x", "Sin (x) / x"}, BaseStyle →
  {Large, FontFamily → "Helvetica", Italic}]                                    5

Plot[Sin[x] / x, {x, -5 Pi, 5 Pi},
 PlotRange → {-0.25, 1.25},
 PlotStyle → {Red, Thick},
 AxesLabel → {"x", "Sin (x) / x"}, BaseStyle →
  {Large, FontFamily → "Helvetica", Italic},
 TicksStyle → {{Medium, Blue},
   {Medium, RGBColor[0.5, 0.2, 0]}}]                                            6
```

1: This is the simplest version of `Plot`: all it requires is an expression depending on a variable and a range over which to plot that variable. MATHEMATICA® has algorithms to select the region which is most likely to be of interest.

2: Tweaking the appearance of a plot will usually involve changing one of `Plot`'s options.

3: Here we change `PlotRange` and `PlotStyle` explicitly. `PlotStyle` takes a list of graphics directives, and the type of `PlotStyle` directives will generally depend on what is being plotted (i.e., lines, points, surfaces).

4: The `AxesLabel` option is used here. The `BasicMathInput`-palette is useful to typesetting mathematical expressions.

5: The option `BaseStyle` can be used to specify the basic size, font, font-shape, etc for the entire plot.

6: As a last example, we use a list of two styles for `TickStyle` to specify both x- and y-axis ticking characteristics.

Lecture 05 MATHEMATICA® Example 2

Plotting Precision and an Example of Interaction

Download notebooks, pdf(color), pdf(bw), or html from http://pruffle.mit.edu/3.016-2011.

Even for continuous functions, a graphical representation is a discrete object. The level of precision is associated with the *mesh*—which is the set where numerical evaluations are performed. More mesh points generally results in a smoother representation, but at the cost of longer computation and memory.

Mesh and MeshStyle

```
Plot[Sin[x] / x,
 {x, -5 Pi, 5 Pi}, PlotRange → All,
 PlotStyle → {Red, Thick}, Mesh → All,
 MeshStyle → {Black, PointSize[0.015]}]
```

1

MaxRecursion and PlotPoints

```
Plot[Sin[x] / x,
 {x, -5 Pi, 5 Pi}, PlotRange → All,
 PlotStyle → {Red, Thick}, , Mesh → All,
 MeshStyle → {Black, PointSize[0.015]},
 MaxRecursion → 2, PlotPoints → 8]
```

2

Interactive Graphics: An Example of Manipulate

```
Manipulate[Plot[Sin[x] / x, {x, -5 Pi, 5 Pi},
  PlotRange → All, PlotStyle → {Red, Thick},
  AxesLabel → {"x", "sin(x)/x"}, BaseStyle →
   {Large, FontFamily → "Helvetica", Italic},
  TicksStyle → {{Medium, Blue}, {Medium,
     RGBColor[0.5, 0.2, 0]}}, Mesh → All,
  MeshStyle → {Black, PointSize[0.015]},
  MaxRecursion → recursion,
  PlotPoints → plotpoints],
 {{recursion, 3}, 1, 15, 1},
 {{plotpoints, 4}, 2, 12, 1}]
```

3

1: The option `Mesh→All` shows the points where `Plot` made numerical evaluations. Note that the points are not equally spaced, but are adapted to the plot (in this case, to the curvature). `MeshStyle` permits specification of how the mesh is visualized.

2: A simple way to control the mesh is with `PlotPoints` (which specifies how many points to sample initially) and `MaxRecursion` (which specifies how many times to try to optimize the adaptation of the points on the curve).

3: This is a simple example of using `Manipulate` to change `PlotPoints` and `MaxRecursion` interactively. Here, both of the options point to variables (`recursion` and `plotpoints`) that can be adjusted via a graphical interface.

Lecture 05 MATHEMATICA® Example 3

Multiple Curves, Filling, and Excluding Points

Download notebooks, pdf(color), pdf(bw), or html from http://pruffle.mit.edu/3.016-2011.

Here, simple examples of plotting several curves at the same time, of filling between curves, or between curves and the axis, and of telling plot to ignore certain points, are demonstrated.

```
Plot[Sin[x] / x, {x, -5 Pi, 5 Pi},
 PlotRange → {-0.25, 1.25},
 PlotStyle → {Red, Thick},
 TicksStyle → {{Medium, Blue},
   {Medium, RGBColor[0.5, 0.2, 0]}},
 Filling → Automatic]
```
1

Combining several curves

```
Plot[{Sin[x] / x, Tan[x] / x},
 {x, -5 Pi, 5 Pi}, BaseStyle → {Thick}]
```
2

```
Plot[{Sin[x] / x, Tan[x] / x},
 {x, -5 Pi, 5 Pi}, PlotStyle → {{Red, Thick},
   {Hue[0.3, 1, .5], Thickness[0.005]}}]
```
3

Removing points with Exclusions

```
Plot[Tan[x] / x, {x, -5 Pi, 5 Pi},
 BaseStyle → {Thick, Medium},
 Exclusions → {-Pi / 2, Pi / 2}]
```
4

```
Plot[Tan[x] / x, {x, -5 Pi, 5 Pi},
 BaseStyle → {Thick, Medium}, Exclusions →
  Table[p, {p, -9 Pi / 2, 9 Pi / 2, Pi}]]
```
5

Multiple curves with exclusions

```
Plot[{Sin[x] / x, Tan[x] / x}, {x, -5 Pi, 5 Pi},
 PlotStyle → {{Red, Thick}, {Hue[0.3, 1, .5],
    Thickness[0.005]}}, Exclusions →
  Table[p, {p, -9 Pi / 2, 9 Pi / 2, Pi}]]
```
6

Filling between curves

```
Plot[{Sin[x] / x, Tan[x] / x},
 {x, -5 Pi, 5 Pi}, PlotStyle → {{Red, Thick},
   {Hue[0.3, 1, .5], Thickness[0.005]}},
 PlotRange → {-0.25, 1.25}, Exclusions →
  Table[p, {p, -9 Pi / 2, 9 Pi / 2, Pi / 2}],
 Filling → {2 → {{1}, {RGBColor[1, 0, 0, 0.2],
     RGBColor[0, 0, 1, 0.2]}}}]
```
7

1: Simple filling to the x-axis can be produced with `Filling→Automatic`.

2: When `Plot` gets a list of expressions as its first argument, it will superimpose the curves obtained from each. The curves' colors are chosen automatically, but can be specified. (n.b., if you find that the colors are not changing as you'd expect, try calling `Evaluate` on the list.) In this example, a vertical line appears for the $\tan(x)/x$ function where the values change as $\pm\infty$. To change the appearance of each curve, a list containing a style-directive list for each curve is used for the `PlotStyle` option. The first style, {`Red,Thick`}, uses simple directives for basic, easy-to-remember, control; the second style uses higher precision control with `Hue` and `Thickness`.

3: The singularities in the function produce vertical lines in the above plots. To remove these features, the option `Exclusions` can get a list of points where the curve should be sliced and not evaluated.

4: Here, we use `Table` to produce a list of all the singularities in $\tan(x)/x$. This list is passed via `Exclusions`.

7: This is a more complex example of filling: here we ask for the filling to take place between the second curve and the first—and to use different filling styles when the first curve lies above or below the second curve.

Lecture 05 MATHEMATICA® Example 4

Plotting Two Dimensional Parametric Curves and Mapped Regions

Download notebooks, pdf(color), pdf(bw), or html from http://pruffle.mit.edu/3.016-2011.

Here are simple examples of using `ParametricPlot` to plot functions for curves in the form $(x(t), y(t))$ and regions in the form $(x(s,t), y(s,t))$.

```
? ParametricPlot
```
1

```
MagicCircles[ t_, n_] :=
 { Cos[n t - Pi + 2 Pi Quotient[n t, 2 Pi] / n ] +
   Cos[2 Pi Quotient[n t, 2 Pi] / n],
  Sin[n t - Pi + 2 Pi Quotient[n t, 2 Pi] / n ] +
   Sin[2 Pi Quotient[n t, 2 Pi] / n]}
```
2

```
ParametricPlot[
 MagicCircles[t, 5], {t, 0, 2 Pi},
 PlotStyle → Thick, PlotRange → All]
```
3

```
Manipulate[
 ParametricPlot[MagicCircles[t, ncirc],
  {t, 0, lastp}, PlotStyle → Thick,
  PlotPoints → 6 ncirc, Axes → False],
 {{ncirc, 3}, 1, 36, 1},
 {{lastp, 2 Pi}, 0.0001, 2 Pi}]
```
4

```
OrbitOrbit[ r_, t_, n_] :=
 { r Cos[n t ] + Cos[t], r Sin[n t] + Sin[t]}
```
5

```
ParametricPlot[
 Evaluate[OrbitOrbit[.5, t, 12]],
 {t, -Pi, Pi}, PlotStyle → Thick]
```
6

Now we let both r and t vary. Some regions in the disk r ∈ (0.25,0.75) don't get covered, and others get covered one or more times.

```
ParametricPlot[Evaluate[OrbitOrbit[r, t, 12]],
 {t, -Pi, Pi}, {r, .25, .75},
 PlotStyle → {Thick, Red},
 Mesh → False, PlotPoints → 72]
```
7

```
ParametricPlot[Evaluate[OrbitOrbit[r, t, 6]],
 {t, -Pi, Pi}, {r, .25, .9},
 PlotStyle → {Thick, Red},
 Mesh → False, PlotPoints → 36,
 ColorFunction → (Hue[#3, 1, 1, 0.25] &)]
```
8

2: A function, *MagicCircles[t,n]* , is defined to produce some interesting parametric plots. It returns data in the form `{x(t),y(t)}` where $t \in (0, 2\pi)$. The second argument, `n`, is a parameter which will determine how many circles get drawn.

3: `ParametricPlot` is used with the `PlotStyle` option set for thick curves, and `PlotRange` set to `All`.

4: Here, we make `ParametricPlot` the first argument to `Manipulate` so that the number of circles can be varied (note, that we force `n` to iterate over integers). The trajectory of the curve can be visualized here by interactively changing the upper bound of t with `lastp`.

5: We cook up another function, *OrbitOrbit[r,t,n]* , to demonstrate filling a region. Data is returned in the form `{x(r,t),y(r,t)}`, and `n` is a parameter.

6: If `r` is fixed, `ParametricPlot` produces a curve as before.

7: Letting both `r` and `t` vary, produces a two-dimensional region—one might think of the region as the set of all the curves for different `r`.

8: This is a slightly advanced example where we use a *pure function* for the `ColorFunction` option. I'm including this example because I think it's pretty.

Lecture 05 MATHEMATICA® Example 5

Simple Plots of Data

Download notebooks, pdf(color), pdf(bw), or html from http://pruffle.mit.edu/3.016-2011.

One of MATHEMATICA® 's integrated data resources, `ElementData`, is used to demonstrate plotting of discrete data.

The next command uses Mathematicas Integrated Data Resources, it will not retrieve the data unless you have an active internet connection

```
ElementData[]
```
1

Here is a list of properties that we can access from ElementData

```
ElementData["Properties"]
```
2

However, one should always question the provenence and accuracy of data... Let's make a sanity check: the stable phase of carbon at STP is graphite which is hexagonal (but not close packed).

```
ElementData[6, "StandardName"]
ElementData[6, "CrystalStructure"]
```
3

We create a list of the densities of the first one hundred elements. Data that is missing is reported with Missing[NotAvailable] or Missing[Unknown].

```
Densities =
 Table[ElementData[i, "Density"], {i, 1, 100}]
```
4

```
ListPlot[Densities]
```
5

```
ListPlot[Densities,
 BaseStyle→{Large, FontFamily→"Helvetica",
   PointSize[0.025]}]
```
6

```
ListLinePlot[Densities,
 BaseStyle→{Large, FontFamily→"Helvetica",
   PointSize[0.025]}]
ListPlot[Densities, BaseStyle→
  {Large, FontFamily→"Helvetica",
   PointSize[0.025]}, Joined→True]
```
7

To see the data, we use the PlotMarkers Option.

```
ListLinePlot[Densities,
 BaseStyle→{Large, FontFamily→"Helvetica",
   PointSize[0.025]},
 PlotMarkers→Automatic, AxesLabel→
  {"Element Number", "Density (MKS)"},
 ImageSize→Large]
```
8

1: `ElementData` will download physical data for the elements via an internet connection. *This command won't work if you do not have an active connection.* However, similar data remain in the now obsolete `ChemicalElements` package.

2: This produces a list of properties that are available. One should always suspect data sources! The stable form of carbon and graphite, is hexagonal but not close-packed.

3: For example, this is how to access properties for carbon.

4: `Table` is used with `ElementData` to produce a list, `Densities`, of the first 100 elements for subsequent use. Missing data are indicated with the function `Missing`.

5: Simply using `ListPlot` produces an indexed scatter plot.

6: Like `Plot`, we can use options in `ListPlot` and `ListLinePlot` to change the appearance of the graphic.

7: A set of line segments are drawn (approximating a curve) in `ListLinePlot`—which is equivalent to using `ListPlot` with the option `PlotJoined` set to `True`.

8: Using the `PlotMarkers` option, both the data and the line segments are visualized.

Lecture 05 MATHEMATICA® Example 6

Getting More out of Displayed Data: Screen Interaction

Download notebooks, pdf(color), pdf(bw), or html from http://pruffle.mit.edu/3.016-2011.

Putting too much information on a single data graphic can make it difficult to understand. Using pop-up windows with the mouse can be a nice way to improve graphical information flow. Here, we show how this can be done using `Tooltip`. In these examples, *where* the extra information appears can be altered by replacing `Tooltip` with `StatusArea`, `Annotation`, or `PopupWindow`.

Example with Tooltip to make graphics interactive----put your mouse over a point and you get a pop-up with more information

```
ListLinePlot[Tooltip[Densities],
 BaseStyle→{Large, FontFamily→"Helvetica",
   PointSize[0.025]},
 PlotMarkers→Automatic, AxesLabel→
  {"Element Number", "Density (MKS)"},
 ImageSize→Large]
```

1

This is a slightly more complicated example of Tooltip. We create a data structure with {x(i),y(i)} = {density(i), bulkmodulus(i)} and then tell Tooltip to pop-up the element's symbol when the mouse is over it.

```
ListPlot[
 Table[Tooltip[{ElementData[i, "Density"],
    ElementData[i, "BulkModulus"]},
   ElementData[i, "Abbreviation"],
   LabelStyle→{Large}], {i, 1, 100}],
 BaseStyle→{Large, FontFamily→"Helvetica",
   PointSize[0.025]}, PlotMarkers→Automatic,
 AxesLabel→{"Density", "Bulk Modulus"},
 PlotLabel→"MKS Units",
 ImageSize→Full]
```

2

1: This is a simple example of `Tooltip`: wrapping the first argument to `ListPlot` or `ListLinePlot` inside `Tooltip` will show the value of each data point when the mouse is over it.

2: I like this example which uses `Tooltip[{xi,yi},labeli]` to produce an interesting way to pick material properties. Suppose we were interested in finding materials that are very stiff (large bulk modulus) but not very heavy (low density)—plotting modulus versus density will identify "interesting" elements in the northwest region of the plot. Using `Tooltip` with `ElementData[i,''Abbreviation'']` allows us to explore element properties without cluttering up the plot. I use `LabelStyle` as an option for `Tooltip` and `ImageSize` as an option for `ListPlot` to make things readable on the display.

Lecture 05 MATHEMATICA® Example 7

Graphical Data Exploration, continued

Download notebooks, pdf(color), pdf(bw), or html from http://pruffle.mit.edu/3.016-2011.

We use `BarChart` and `PieChart` in the `BarCharts` and `PieCharts` packages to explore the relative abundances of different crystal structures among the elements. A three-dimensional histogram of elements selected by their melting points and densities is produced with `Histogram3D` from the `Histograms` package.

Here we do a small exercise to get a graphical representation of which Crystal Structures the elements form, and represent the frequency of each type. First we create a list of known elemental crystal structures for the first 100 elements.

```
CrystalStructures = Table[ElementData[
   i, "CrystalStructure"], {i, 100}]
```
1

```
UniqueStructures = Tally[Cases[
   CrystalStructures, Except[Missing[_]]]]
MatrixForm[UniqueStructures]
```
2

Here is a bar chart showing the frequency of crystal structures.

```
Needs["BarCharts`"]
BarChart[Transpose[UniqueStructures][[2]],
 BarLabels ->
  Transpose[UniqueStructures][[1]],
 BaseStyle → {Large, FontFamily → "Helvetica"},
 BarOrientation → Horizontal, ImageSize → Full]
```
3

```
Needs["PieCharts`"]
PieChart[Transpose[UniqueStructures][[2]],
 PieLabels ->
  Transpose[UniqueStructures][[1]],
 BaseStyle → {Large, FontFamily → "Helvetica"},
 ImageSize → Full]
```
4

As a last example, we produce a 3D histogram. The height of each bar corresponds to the number of elements in a range of melting points and range of densities.

```
Needs["Histograms`"]
histdata = DeleteCases[Table[
   {ElementData[i, "AbsoluteMeltingPoint"],
    ElementData[i, "Density"]}, {i, 100}],
  {Missing[_], _} | {_, Missing[_]}]
Histogram3D[histdata, AxesLabel →
  {"Melting Point", "Density", "Number"},
 HistogramCategories → {16, 24}]
```
5

1: `CrystalStructures` will be a list of the crystal structures of the most stable solid phase. (I am not sure what is meant by most stable—this is ambiguous, but that is what it says in the documentation)

2: `UniqueStructures` will be a list of pairs—each item will be comprised of a crystal structure and how many times it appears. We use `Cases` to remove missing data by using a pattern, and then use `Tally` to create the data structure.

3: Because `BarChart` needs data of the form {`y1, y2, ...`}, we need to manipulate the data. To get the data, `Transpose` will put the abundances into the second row, which is also the list we need. We use the first row of the transpose for the `BarLabels` option. The plot is easier to read if horizontal, so we use the `BarOrientation` option.

4: Here we simply replace the barchart with `PieChart`.

5: As a final example, we create a histogram of elements with similar densities and melting points. We use a pattern with an "or" in `Cases` to remove missing data with `DeleteCases`, because we cannot plot data where either the density *or* the melting point is missing.

Lecture 05 MATHEMATICA® Example 8

Three-Dimensional Graphics

Download notebooks, pdf(color), pdf(bw), or html from http://pruffle.mit.edu/3.016-2011.

Here we show examples of three-dimensional graphics, although it would be better to say, 3D graphics projected onto a 2D screen.

```
EPot[x_, y_, z_, xo_, yo_] :=
                1
  ------------------------------
  √(x - xo)^2 + (y - yo)^2 + z^2
```
1

```
SheetOLatticeCharge[x_, y_, z_] :=
 Sum[EPot[x, y, z, xo, yo],
  {xo, -5, 5}, {yo, -5, 5}]
```
2

SheetOLatticeCharge represents the electric field produced by an 11 by 11 array of point charges arranged on the x-y plane at z = 0. The following command evaluates and plots the field variation in the plane z = 0.25:

```
Plot3D[
 Evaluate[SheetOLatticeCharge[x, y, 0.25]],
 {x, -6, 6}, {y, -6, 6}]
```
3

*Note below how **theplot** is set to contain the output of the Plot3D command---it is now a symbol assigned to a graphics object. The number of plotpoints is increased so that we can resolve all the bumps. This will take a while to compute on most machines.*

```
theplot = Plot3D[
   Evaluate[SheetOLatticeCharge[x, y, 0.25]],
   {x, -6, 6}, {y, -6, 6}, PlotPoints → 60]
```
4

This demonstrates the use of RegionFunction plot option which is pure function. Here, only the region inside a cylinder with radius 9 ($x^2 + y^2 \le 9^2$) is plotted.

```
Plot3D[
 Evaluate[SheetOLatticeCharge[x, y, 0.25]],
 {x, -9, 9}, {y, -9, 9}, PlotPoints → 60,
 RegionFunction → (#1^2 + #2^2 ≤ 81 &)]
```
5

This demonstrates the use of the ColorFunction plot option which is pure function. Here we use one of Mathematica ColorData functions.

```
Plot3D[
 Evaluate[SheetOLatticeCharge[x, y, 0.25]],
 {x, -9, 9}, {y, -9, 9}, PlotPoints → 60,
 RegionFunction → (#1^2 + #2^2 ≤ 81 &),
 ColorFunction →
  (ColorData["TemperatureMap"][#3] &)]
```
6

1: This is the electrostatic potential as a function of (x, y, z) due to a single positive charge located at $(x_o, y_o, z = 0)$ (i.e., anywhere on the $z = 0$ plane).

2: By summing over a square lattice of unit charges, this function (*SheetOLatticeCharge*) computes the electrostatic potential over a 11×11 square-lattice of point-charges centered on the z-plane as a function of x, y, and z.

3: `Plot3D` plots data of the form $f(x, y)$ (f is the height above a point (x, y)). Therefore, we must fix one of the coordinates; here we visualize the electrostatic potential at a fixed height ($z = 0.25$). Note that the bounds for both the "horizontal" and "into-screen" coordinates need to be specified.

You can rotate the graphics by dragging the mouse over the surface, translate by dragging with the shift-key held down, and zoom with the alt-key held down.

4: With sufficiently many `PlotPoints`, the structure of the potential at a fixed distance $z = 0.25$ is made apparent. The finer details are not resolved at lower resolutions, but using 60 points in each direction may be overkill and this will be slow on older computers and may not fit on machines with little memory.

5: `RegionFunction` is new as of MATHEMATICA® 6. This is an advanced examples, but it demonstrates how one can plot over non-rectangular domains.

6: As a last example, the use of the new `ColorData` functions for the `ColorFunction` option is demonstrated.

Lecture 05 MATHEMATICA® Example 9

Colors and Contours: Three-Dimensional Graphics in Two Dimensions

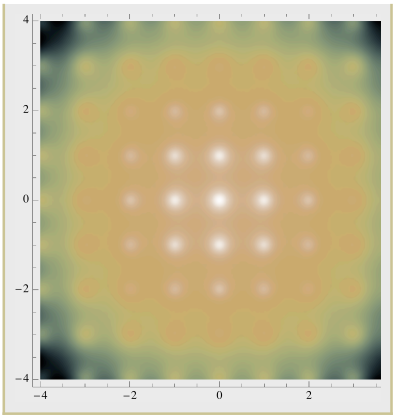
Download notebooks, pdf(color), pdf(bw), or html from http://pruffle.mit.edu/3.016-2011.

Three dimensions can also be visualized by drawing level sets (as in a topographical map) or by drawing colors (as in a relief map). The data burden is usually much smaller than a 3D graphics object, is sometimes easier to interpret, and is certainly easier to publish.

```
theconplot = ContourPlot[
  Evaluate[SheetOLatticeCharge[x, y, 0.25]],
  {x, -6, 6}, {y, -6, 6}, PlotPoints → 32]                    1

theconplot = ContourPlot[
  Evaluate[SheetOLatticeCharge[x, y, 0.25]],
  {x, -4, 4}, {y, -4, 4}, PlotPoints → 50,
  ColorFunction → Hue, Contours → 24]                         2

thedenplot = DensityPlot[
  Evaluate[SheetOLatticeCharge[x, y, 0.25]],
  {x, -4, 4}, {y, -4, 4},
  PlotPoints → 50, ColorFunction →
   ColorData["GreenBrownTerrain"]]                            3
```



1: We reproduce the 3D graphics object for the sheet of electric charges using `ContourPlot`. Here, the number of contours are picked arbitrarily, but `PlotPoints` has to be increased to resolve details of the function. Moving the mouse over one of the contours will give a pop-up window for the value along that contour.

2: In the representation above, we might conclude that a positive charge (such as a hole) confined to $z = 0.25$ could not be "trapped" because no minima are obvious. Increasing the number of contours with the `Contours` option improves the resolution so that local minima can be observed. Here we pass `Hue` to the `ColorFunction` option; however, I don't find this satisfactory because both the largest and the smallest values are red. In other words, the color scaling runs completely around the outside of a color wheel and ends up where it started.

Unless options are sent requesting otherwise, the values of the plot will be scaled so that the maximum and minimum values are 1 and 0. Thus, two plots would look the same whether the differences are very small or very large. This feature is controlled by `ColorFunctionScaling`.

3: Here, instead of a single color decorating the region between two neighboring contours, a color is plotted directly indicating the "height" of the function. `ColorData` is used with `GreenBrownTerrain` so that the high potentials look like snow-covered peaks and lower potentials look like green river-deltas.

Lecture 05 MATHEMATICA® Example 10
Graphics Primitives, Drawing on Graphics, and Combining Graphical Objects

Download notebooks, pdf(color), pdf(bw), or html from http://pruffle.mit.edu/3.016-2011.

Here, examples of placing *Graphics Primitives* into a *Graphics Object* are demonstrated by direct means: by a drawing tool, and by sequential combination.

It can be useful to be able to build up arbitrary graphics objects piece-by-piece using simple "graphics primitives" like ***Circle****:*

```
thecirc = Graphics[Circle[{2, 2}, 1.5]]
```
1

```
Show[thecirc, Axes -> True]
```
2

```
Show[thecirc, Axes -> True,
 AxesOrigin → {0, 0}, AspectRatio → 1]
```
3

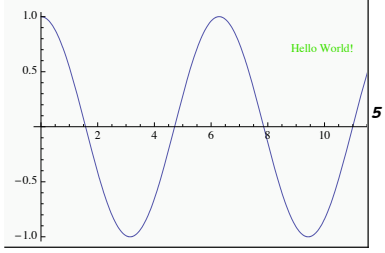
Now we take a simple plot…

```
cosplot = Plot[Cos[x], {x, 0, 4 Pi}]
```
4

Adding Graphics Primitives to Plots (or other graphics objects) using the built-in Drawing Tool

Mathematica6 now has a simple drawing editor that allows you add text, arrows, lines, and shapes to existing graphics. To do this, select the previous graphics output for the cosine plot. While the graphics are selected, use the Menu Item "Drawing Tools" under Graphics. After you have added shapes, text, etc., move the cursor to the left of the selected graphics object and type a symbol (below, I used "thenewplot") for the new (combined) graphics object to be assigned to.

```
thenewplot =
```



5

```
thenewplot
```
6

Combining Graphical Objects using Show.

and overlay some text in places of our own choosing…

```
Show[cosplot, Graphics[
  Text["One Wavelength", {2 Pi, 0.5}]],
 Graphics[Text["Two\nWavelengths",
   {4 Pi, 0.5}]], PlotRange → All]
```
7

```
Show[thenewplot, Graphics[
  Text["One Wavelength", {2 Pi, 1.1}]],
 Graphics[Text["Two Wavelengths",
   {4 Pi, 1.1}]], PlotRange → All]
```
8

1: A `Circle` is a graphics primitive, and making a primitive an argument to `Graphics` returns a "Graphics Object." When a graphics object is output, graphics appear. The graphical output can be suppressed by a trailing semicolon. In this case, `thecirc` is assigned to the graphics object and it is displayed. If a trailing semicolon appears (e.g., a unit circle `thecirc = Graphics[Circle[]];`), then the assignment is made to `thecirc`, but no graphics are sent to the display.

2–3: Additional options can be added to a graphics object with `Show`. The result is a new graphics object.

4: Here we create a graphics object and assign it to the symbol `cosplot` by simply using `Plot`.

5: If the mouse is clicked on the display of the graphics object, then it can be edited just like input. Clicking to the left of the object allows you to type a symbol for assignment to the graphics object. Shown here is the result of assigning a graphic to `thenewplot`. If the graphic is selected, then a *Drawing Tools Widget* can be pulled up under the Graphics menu item. With the widget, other primitives such as text, lines, arrows, and shapes can be combined. When the expression is evaluated, the combined graphics will be assigned to `thenewplot`.

7–8: Here, `Show` is used to add text via a graphics primitive to the original plot and to the new combined graphics object.

Lecture 05 MATHEMATICA® Example 11

A Worked Example: The Two-Dimensional Wulff Construction

Download notebooks, pdf(color), pdf(bw), or html from http://pruffle.mit.edu/3.016-2011.

The Wulff construction is a famous thermodynamic construction that predicts the equilibrium enclosing-surface of an anisotropic isolated body. The anisotropic surface tension, $\gamma(\hat{n})$, is the amount of work (per unit area) required to produce a planar surface with outward normal $\hat{n}$. The construction proceeds by drawing a bisecting plane at each point of the polar plot $\gamma(\hat{n})\hat{n}$. The interior of all bisectors is the resulting *Wulff shape*.
A working example of the Wulff construction for a $\gamma(\theta)$ in two dimensions is produced here.

This next example shows a clever way to perform a famous thermodynamic graphical construction called the Wulff construction.

```
wulffline[{x_, y_}, wulfflength_] :=
 Module[{θ, wulffhalf = wulfflength * 0.5,
   x1, x2, y1, y2}, θ = ArcTan[x, y];
  x1 = x + wulffhalf * Cos[θ + Pi / 2];
  x2 = x + wulffhalf * Cos[θ - Pi / 2];
  y1 = y + wulffhalf * Sin[θ + Pi / 2];
  y2 = y + wulffhalf * Sin[θ - Pi / 2];
  Graphics[Line[{{x1, y1}, {x2, y2}}]]
 ]
```

1

```
gammaplot[ theta_ , anisotropy_ , nfold_] :=
 {Cos[theta] + anisotropy *
    Cos[(nfold + 1) * theta], Sin[theta] +
   anisotropy * Sin[(nfold + 1) * theta]}
```

2

```
GammaPlot =
 ParametricPlot[gammaplot[t, 0.1, 4],
  {t, 0, 2 Pi}, PlotStyle →
   {{Thickness[0.01], RGBColor[1, 0, 0]}}]
```

3

```
Show[Table[wulffline[gammaplot[t, 0.1, 4], 2],
  {t, 0, 2 Pi, 2 Pi / 100}], GammaPlot]
```

4

```
ToutesDesLoups[anisotropy_, nfold_] :=
 Module[{GammaPlot}, GammaPlot =
   ParametricPlot[gammaplot[t, anisotropy,
     nfold], {t, 0, 2 Pi}, PlotStyle →
    {{Thickness[0.01], RGBColor[1, 0, 0]}}];
  Show[Table[wulffline[gammaplot[
     t, anisotropy, nfold], 3],
   {t, 0, 2 Pi, 2 Pi / 100}], GammaPlot]]

Manipulate[ToutesDesLoups[aniso, nfold],
 {{aniso, 0.1}, -0.9, 0.9},
 {{nfold, 6}, 2, 16, 1}]
```

5

1: This function takes a point {x,y} as an argument and then returns a graphics object of a line of specified length. The line is the perpendicular bisector required by the Wulff construction.

2: This is an example $\gamma(\hat{n})$ with the surface tension being smaller in the $\langle 11 \rangle$-directions (if the `anisotropy` parameter is positive).

3: A particular instance of a γ-plot is assigned to `GammaPlot`.

4: `Table` is used to produce a list of graphics objects by calling *wulfline* function at one hundred points on the γ-plot. The equilibrium shape is the interior of all the curves and the γ-plot from which it derives is superimposed by collecting all the graphics together with `Show`.

5: All the above steps are collected together and bundled into a `Module` to produce a single visualization function, *ToutesDesLoups* . The function depends on the prior definition of `gammaplot[t,`α`,n]`.

6: Here, *ToutesDesLoups* is used as the argument to `Manipulate` to visualize the effect of changing the anisotropy factor and the n-fold axis.

Graphical Animation: Using Time as a Dimension in Visualization

Animations can be very effective tools to illustrate time-dependent phenomena in scientific presentations. Animations are sequences of multiple images—called frames—that are written to the screen interatively at a constant rate: if one second of real time is represented by N frames, then a real-time animation would display a new image every $1/N$ seconds.

There are two important practical considerations for computer animation:

frame size An image is a an array of pixels, each of which is represented as a color. The amount of

memory each color requires depends on the current image depth, but this number is typically 2-5 bytes. Typical video frames contain 1024×768 pixel images which corresponds to about 2.5 MBytes/image and shown at 30 frames per second corresponding to about 4.5 GBytes/minute. Storage and editing of video is probably done at higher spatial and temporal resolution. Each frame must be read from a source—such as a hard disk—and transfered to the graphical memory (VRAM) before the screen can be redrawn with a new image. Therefore, along with storage space the rate of memory transfer becomes a practical issue when constructing an animation.

animation rate Humans are fairly good at extrapolating action between sequential images. It depends on the difference between sequential images, but animation rates below about 10 frames per second begin to appear jerky. Older Disney-type cartoons were typically displayed at about 15 frames per second, video is displayed at 30 frames per second. Animation rates above about 75 frames per second yield no additional perceptable "smoothness." The upper bound on computer displays is typically 60 hertz.

Lecture 05 MATHEMATICA® Example 12

Animation

Download notebooks, pdf(color), pdf(bw), or html from http://pruffle.mit.edu/3.016-2011.

Animations are a nice way to visualize an extra dimension, like time. An animation is composed of a sequence of displayed graphics (frames) that are displayed iteratively. Animations are fairly easy to create–and can be great fun.

```
Fxt[x_, t_] :=
 Sin[3 (x + 10 - t)] Exp[-(x + 10 - t)^2] -
  Sin[3 (x - 10 + t)] Exp[-(x - 10 + t)^2]
```
1

```
Animate[
 Plot[Fxt[xvar, timevar], {xvar, -15, 15},
  PlotRange → {-1, 1}, PlotStyle → {Thick, Red},
  Filling → Axis, FillingStyle →
   {RGBColor[0, 0.5, 0, 0.5], RGBColor[
     0, 0, 0.5, 0.5]}], {timevar, 0, 25}]
```
2

This is the solution to the temperature evolution equation (the diffusion equation) for a square of length L initially at 500K embedded in a plate initially at 100K , κ is the themal diffusivity (units length^2/time). We introduce a "normalized" time and space variables variable $\tau = \kappa\, t/L^2$ and ξ = x/L and η=y/L

```
TempSquare =
 100 + 400 Integrate[Exp[-((x-xo)^2 + (y-yo)^2)/(4 κ t)]/(4 π κ t),
   {xo, -L/2, L/2}, {yo, -L/2, L/2}]
NormalizeRules = {t → τ L^2/κ, x → ξ L,
   y → η L, xo → ξo L, yo → ηo L};
TempSquare = Simplify[TempSquare /.
   NormalizeRules, Assumptions → κ > 0 && L > 0]
```
3

We divide by 500 so that the temperatures should scale between zero and one, and then use ColorFunctionScaling->False so that the colors are consistent over time.

```
ListAnimate[
 Table[Plot3D[TempSquare / 500, {η, -1, 1},
   {ξ, -1, 1}, PlotRange → {0, 1}, PlotPoints →
    50, ColorFunction → "TemperatureMap",
   ColorFunctionScaling → False],
  {τ, 0.001, .1, 0.002}]]
```
4

1: We will create a simple animation by cooking up a function $f(x,t)$ and then plotting it for a range of x and for a sequence of t's.

2: This plot would be the frame associated with $t = 0$.

3: Using `Plot` as the argument to `Animate` produces the animation. Note, `xvar` 'belongs' to `Plot` while `timevar` belongs to `Animate`. Can you imagine what the animation would look like if we animated over x and plotted over t? No? Try it!

4: We will produce a three-dimensional animation of how the temperature would change in a flat plate, if at time $t = 0$ there is a square at a different temperature than the rest of the plate. The governing partial differential equation is $\partial T/\partial t = \kappa \nabla^2 T$ and for initial conditions $T(x, y, t = 0) = 500$ when $-L/2 < x, y < L/2$ and $T = 100$ otherwise, the closed form solution can be expressed as an integral. To make a plot, we must send a function that can be evaluated numerically. To do this, we must *non-dimensionalize variables* (also known, as *dimensional scaling* or *normalizing variables*). This is done by dividing variables having physical units (such as x), with a characteristic quantity in the model that has the same physical units (here, we will use the model's length L to produce a dimensionless variable $\xi = x/L$) *NormalizeRules* is a set of rules that can be applied to our physical problem. After the normalization rules are applied, the properly scaled solution should be a non-dimensional temperature-quantity as a function of non-dimensional space- and time-quantities.

5: Finally, we will use `Plot3D` inside `ListAnimate`. `Plot3D`'s argument is scaled by dividing by the maximum temperature, so that all temperature-like quantities scale between zero and one. We turn off `ColorFunctionScaling` so that the 'meaning' of each color remains constant in the animation. `ListAnimate` takes a list of frames that are produced via `Table`.

Lecture 05 MATHEMATICA® Example 13

An Example of Animating a Random Walk

Download notebooks, pdf(color), pdf(bw), or html from http://pruffle.mit.edu/3.016-2011.

A *random walk* process is an important concept in diffusion and other statistical phenomena. Functions to simulate a random walk in two dimensions are constructed and then visualized with animations.

```
randomwalk[0] = {0, {0, 0}}                                              1
```

```
randomwalk[nstep_Integer?Positive] :=
 randomwalk[nstep] =
  {nstep, randomwalk[nstep - 1][[2]] +
    RandomReal[0.5] {Cos[
       theta = RandomReal[2 π]], Sin[theta]}}                            2
```

Create a function that returns a graphic object putting the step number at the correct place:

```
gtext[nstep_Integer?NonNegative] :=
  gtext[nstep] = Graphics[
    Text[ToString[randomwalk[nstep][[1]]],
     randomwalk[nstep][[2]]]];                                           3
```

```
locations = Show[Table[gtext[i], {i, 0, 100}],
  PlotRange → All, AspectRatio → 1]                                      4
```

```
gline[nstep_Integer] := gline[nstep] =
   Graphics[Line[{randomwalk[nstep - 1][[2]],
      randomwalk[nstep][[2]]}]];                                         5
```

```
Show[Table[gtext[i], {i, 0, 100}],
 Table[gline[j], {j, 1, 100}],
 PlotRange → All, AspectRatio → 1]                                       6
```

```
Animate[Show[gtext[i], gline[i]],
 {i, 1, 49, 1}]                                                          7
```

If we use the PlotRange from a graphical object that contains all the points, we can fix the framesize, we use AbsoluteOptions

```
prange =
 PlotRange /. AbsoluteOptions[locations]                                 8
```

```
Animate[Show[gtext[i], gline[i],
  PlotRange → prange], {i, 1, 100, 1}]                                   9
```

```
Animate[
 Show[Table[{gtext[i], gline[i]}, {i, 1, j}],
  PlotRange → prange], {j, 2, 100}]                                      10
```

1–2: This is a recursive function that simulates a random walk process. Each step in the random walk is recorded as a list structure, { {`iteration number`}, { `x , y` }}, and assigned to *randomwalk* [iteration number]. For each step (or iteration), a number between 0 and 1/2 is selected (for the magnitude of the displacement), and an angle between 0 and 2π is selected (for the direction), with each of these numbers being selected randomly from a uniform distribution (using `RandomReal`). The function includes an assignment, so all previous values are stored in memory.

3: The function *gtext* calls *randomwalk* to create a text graphics-object located at the position corresponding to `nstep`.

4: This shows the history of a random walk after 50 iterations by combining the graphics objects created by *gtext* . The resulting graphics object gets assigned, because we will use some information contained in it later.

5: To improve the physical interpretation of the previous graphic, it would be an aid to the eye if the individual jumps were indicated. To do this, the function *gline* calls *randomwalk* to create a line graphics-object connecting the position corresponding to `nstep` to its previous position.

7: Thus, we could animate by combining the line and the text with `Show` and using that as the argument to `Animate`. *However, this result will be unsatisfactory because the "length scale" of each frame will not be consistent.*

8: To solve this problem, we find the bounds of a graphics object (`locations`) that contains all the points, and then query its `PlotRange` using `AbsoluteOptions` and this is assigned to a symbol `prange`.

9: The animation is consistent now, but could still use some improvement.

10: Here, we animate the graphics object that also contains the history of prior jumps. This is not a terribly efficient way to do this because we recreate the early steps many times over, but it works for our purposes.

Lecture 05 MATHEMATICA® Example 14

Worked Example (part A): Visualizing the Spinodal and Common Tangent Construction

Download notebooks, pdf(color), pdf(bw), or html from http://pruffle.mit.edu/3.016-2011.

The spinodal and common tangent construction is a fundamental thermodynamic concept used for the creation of an alloy phase diagram from molar-free energies. This construction appears repeatedly in studies of materials. An example of visualizing this construction as a function of temperature will be worked out in detail for the case of a single curve and a binary alloy.

First, we will work out all the steps in detail that are used to build up a single visualization, and then we will collect it all together in a reusable function.

A prototype molar free energy of mixing using the same xlogx function for the ideal entropy of mixing terms. The temperature term is a scaled energy (RT), and it is assumed that enthalpies have been scaled so that the temperatures of interest (if there are any) are between T=0 and T=10.

```
xlogx[0] =
 xlogx[1] = xlogx[0.0] = xlogx[1.0] = 0;
xlogx[x_] := x Log[x]
Gmolar[X_, T_] :=
 5 X (1 - X) + T (xlogx[X] + xlogx[1 - X]) + X / 2
```

1

Here is the shape of our prototype free energy at T=3/2

```
p1 = Plot[Gmolar[x, 3 / 2],
  {x, 0, 1}, PlotStyle → Thick]
```

2

We will need the bounds of the above graphics object:

```
{{graphxmin, graphxmax},
  {graphymin, graphymax}} =
 PlotRange /. AbsoluteOptions[p1, PlotRange]
```

3

First let's determine where the spinodal region (by finding where the second derivative with respect to composition is negative

```
ddg = D[Gmolar[x, 3 / 2], {x, 2}]
```

4

Then, use RegionPlot to illustrate the range over which spinodal decomposition is spontaneous

```
p2 = RegionPlot[ddg < 0,
  {x, graphxmin, graphxmax},
  {T, graphymin, graphymax},
  PlotStyle → RGBColor[0, 1, .5, 0.1]]
```

5

Show them both together to identify the spinodal region

```
Show[p1, p2]
```

6

1: We cook up a prototypical molar free-energy as a function of molar composition, `X`, and temperature `T`. The $x \log x$ terms are calculated with a handy function, *xlogx* , which will handle the zeroes without numerical difficulty at `0 Log[0]`.

2: The molar free-energy is plotted at a particular temperature ($T = 1.5$) and assigned to a symbol, `p1`.

3: We will need the bounds of the plot to create other graphical objects. We grab the bounds with `AbsoluteOptions` and assign them to variables using a handy assignment construction `{a,b} = List`.

4: The spinodal region is the easiest to visualize—it is the region where the second derivative of the molar free-energy is negative. The second derivative is assigned to `ddg`.

5: `RegionPlot` evaluates its first argument over a square region and fills where the argument is true. It is exactly what we need in order to visualize the spinodal region. We use the bounds that we calculated from the free energy curve as the bounds for `RegionPlot`.

6: Showing both plots together, we visualize the spinodal region.

Lecture 05 MATHEMATICA® Example 15

Worked Example (part B): Visualizing the Spinodal and Common Tangent Construction

Download notebooks, pdf(color), pdf(bw), or html from http://pruffle.mit.edu/3.016-2011.

The common tangent is any finite line segment that touches the molar free-energy at two points which have the same derivative. For phase diagrams, we are interested only in lower common tangents (i.e., lines that touch the molar free-energy, but always lie below all values). One can picture the common tangent by imagining that an elastic string is stretched along a molar free-energy curve; the common tangents are where the string pulls away from the the curves.

The common tangent is related to the *convex hull* that appears in computational geometry.

We can use the ConvexHull to find the common tangent lines; this function is in the Computational Geometry Package.

```
<< ComputationalGeometry`
```
1

First we compute a list of values along the molar free energy curve, then compute those that lie outside the common tangent(s) (i.e., the convex hull). Because the points are given in order, we might as well sort them on the way back out. Note, the convex hull program gives the indices of the vertices that are on the hull.

```
npoints = 100;
gvals = Table[{x, Gmolar[x, 3 / 2]},
   {x, 0, 1, 1 / N[npoints - 1]}];
```
2

We only want the lower convex hull; therefore we add some "fictive" points to the beginning and the end of the data. The the fictive points add a rectangle to the top of the curve that should be part of the computed convex hull.

```
gmax = Max[Transpose[gvals][[2]]];
PrependTo[gvals, {0, 10 * Abs[gmax]}];
AppendTo[gvals, {1, 10 * Abs[gmax]}];
```
3

After we compute this hull, we shift the hull by one and take off its first and last element. We strip the first and last element from the discrete values of free energy as well.

```
chull = Sort[ConvexHull[gvals]];
chull = Drop[Drop[chull - 1, 1], -1]
gvals = Drop[Drop[gvals, 1], -1]
```
4

The common tangent(s) correspond to gaps in the vertex list of the common tangent. We will use Split to find the set of continous sequences.

```
convexparts = Split[chull, (#2 - #1 < 2) &]
```
5

```
{{1, 2, 3, 4, 5, 6}, {95, 96, 97, 98, 99, 100}}
```

1: To calculate convex hulls, the `ComputationalGeometry` package is needed.

2: `ConvexHull` operates on discrete data. Discrete data are created by evaluating *Gmolar* at `npoints` evenly-spaced mesh-points. We use `Table` and assign the discrete data list to `gvals`.

3: `ConvexHull` calculates the *entire hull* (i.e., the polygon that encloses all other points), and we are only interested in the lower hull. Thus, we add a rectangle to the top of the data which is guaranteed to be part of the hull, calculate the hull and discard the upper parts. Here we use `PrependTo` to add a point ten times higher than the maximum value on the left side of the region, and use `AppendTo` to add a corresponding point to the right side of the region. We have thus added a known rectangle that we will remove later.

4: `ConvexHull` returns a *list of indices of points* from the original data. Because the original data was created in an orderly left-to-right way, we can use `Sort` to put the data in a predictable form. Because there was an additional point added at the beginning of `gvals`, we will need to shift the indices down by one (by subtracting 1 from each index), and then we use `Drop` to remove the first and last elements of both `chull` and `gvals`.

5: Thinking about the indices on the convex hull, any ordered sequence of the sorted list must be part of original discrete data and also part of the convex hull. We are interested in connecting the last point of any isolated sequence to the first point of the next sequence. We can use `Split` to find the isolated sequences.

Lecture 05 MATHEMATICA® Example 16

Worked Example (part C): Visualizing the Spinodal and Common Tangent Construction

Download notebooks, pdf(color), pdf(bw), or html from http://pruffle.mit.edu/3.016-2011.

With the information contained in the convex hull data, graphical objects are created to represent the gaps in that data. The gaps coincide with the common tangents.

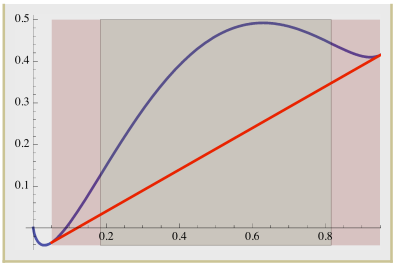
Now we create graphics objects for each of the two-phase regions (i.e., the gaps in the convex hull) and collect them all into a graphics list for subsequent display.

```
len = Length[convexparts];
graphicslist = {};
i = 1;
While[i + 1 ≤ len, leftpoint =
  gvals[[Last[ convexparts[[i]] ]]];
 rightpoint = gvals[[
   First[ convexparts[[i + 1]] ]]];
 ctline = {Red, Thick,
   Line[{leftpoint, rightpoint}]};
 twophaseregion = {RGBColor[0.5, 0, 0, 0.2],
   Rectangle[{leftpoint[[1]], graphymin},
    {rightpoint[[1]], graphymax}]};
 AppendTo[graphicslist, ctline];
 AppendTo[graphicslist, twophaseregion];
 i++
]
p3 = Graphics[graphicslist]
```

1

```
Show[p1, p2, p3]
```

2



1: We traverse the list `convexparts` and construct graphical objects corresponding to the regions of isolated sequences. Because it is possible that a curve may have any number of common tangents, we accumulate graphics primitives in a list as we encounter common tangents. A graphics object is created from the list of graphics primitives.

The number of isolated sequences is assigned to `len` and we start with an empty list `graphicslist`. Then, we loop over the list of length `len`. At each iteration in the loop, we identify the last vertex on the previous point of the convex hull sequence and the first part of the next sequence. We use those indices to extract the points on the curve that have been stored in `gvals`. With the two points, we create red lines for the common tangents—and with the extra graphical information about the original plot, draw a rectangle for the region.

Finally, a new graphics object (`p3`) is created.

2: Our final visualization is obtained by showing all three graphics objects together.

Lecture 05 MATHEMATICA® Example 17

Worked Example (part D): Visualizing the Spinodal and Common Tangent Construction

Download notebooks, pdf(color), pdf(bw), or html from http://pruffle.mit.edu/3.016-2011.

The previous three parts illustrate how one might actually go about developing a complex visualization: create simple working parts and then integrate them together into something more complex. (Don't get the impression that I didn't make any errors or silly conceptual mistakes as I created this example! It was very time consuming and, while it looks fairly straightforward in hindsight, it was a challenge to build.) However, once finished, it is useful to collect everything into a single function that can be reused.

```
Needs["ComputationalGeometry`"];
CommonTangentConstruction[
  Gm_, T_, npts_:100] :=
 Module[{x, y, p1, p2, p3, gxmin, gxmax,
   gymin, gymax, ddg, gvals, gmax,
   chull, conprts, len, glist = {}, i = 1,
   lftpt, rtpt, ctline, twophasreg},
  p1 = Plot[Gm[x, T], {x, 0, 1},
    PlotStyle → Thick];
  {{gxmin, gxmax}, {gymin, gymax}} =
   PlotRange /.
    AbsoluteOptions[p1, PlotRange];
  ddg = D[Gm[x, T], {x, 2}];
  p2 = RegionPlot[ddg < 0,
    {x, gxmin, gxmax}, {y, gymin, gymax},
    PlotStyle → RGBColor[0, 1, .5, 0.1]];
  gvals = Table[{x, Gm[x, T]},
    {x, 0, 1, 1 / N[npts - 1]}];
  gmax = Max[Transpose[gvals][[2]]];
  PrependTo[gvals, {0, 10 * Abs[gmax]}];
  AppendTo[gvals, {1, 10 * Abs[gmax]}];
  chull = Sort[ConvexHull[gvals]];
  chull = Drop[Drop[chull - 1, 1], -1];
  gvals = Drop[Drop[gvals, 1], -1];
  conprts = Split[chull, (#2 - #1 < 2) &];
  len = Length[conprts];
  While[i + 1 ≤ len,
   lftpt = gvals[[Last[ conprts[[i]] ]]] ;
   rtpt = gvals[[ First[ conprts[[i + 1]] ]]];
   ctline =
    {Red, Thick, Line[{lftpt, rtpt}]};
   twophasreg = {RGBColor[0.5, 0, 0, 0.2],
     Rectangle[{lftpt[[1]], gymin},
      {rtpt[[1]], gymax}]};
   AppendTo[glist, ctline];
   AppendTo[glist, twophasreg]; i++];
  p3 = Graphics[glist]; Show[p1, p2, p3]]
```

1

1: Here is the result, *CommonTangentConstruction* , which collects the previous three examples together and returns a single graphical object. *CommonTangentConstruction* takes two arguments for the molar free-energy function, `Gm`, and temperature `T`, and an optional third argument for the precision to calculate the hull. The optional argument is indicated by the `:100` and will default to 100 if not passed to the function.

The first argument must be the name of a defined function of composition and temperature.

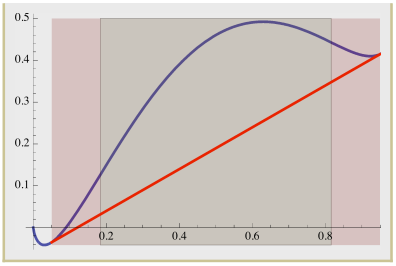
Lecture 05 MATHEMATICA® Example 18

Worked Example (part E): Visualizing the Spinodal and Common Tangent Construction

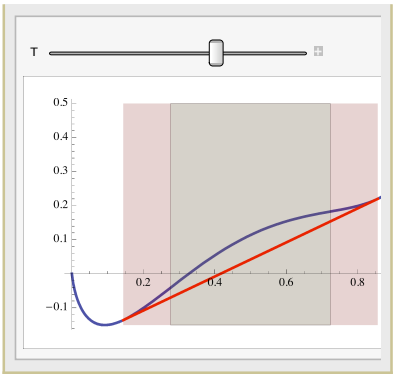
Download notebooks, pdf(color), pdf(bw), or html from http://pruffle.mit.edu/3.016-2011.

Examples of visualizing with *CommonTangentConstruction* are presented here.

```
CommonTangentConstruction[Gmolar, 1.5]
```
1




```
Manipulate[CommonTangentConstruction[
  Gmolar, T, 300], {{T, 2}, 0, 3}]
```
2



1: This is the construction at $T = 1.5$.

2: Here we use the construction as an argument to `Manipulate` so that we can observe the effect of temperature on the spinodal and common tangent construction.