Doing Applied Math with *Mathematica*

Douglas E. Baldwin — August 19, 2010

What is *Mathematica*?

Mathematica is a computer algebra system (CAS) and a high-level programming language. *Mathematica* and *Maple* are the most popular CASs internationally; the other CASs sometimes used by applied mathematicians are *Maxima* (formerly *Macsyma*), *Derive* (discontinued in 2007), *Axiom* (forked in 2007 to *OpenAxion* and *FriCAS*), and *FORM*.

Frequently, as an applied mathematician, you'll run into problems where the algebra is too complicated for you to do it by hand; problems where it'd take *Mathematica* less than a minute and it'd take you 4 months to do the calculation and another 6 months to check that it's correct. While *Mathematica* is a very powerful tool, it isn't a panacea. Usually the best way to get *Mathematica* to solve your problem is to figure out how you'd do it by hand and then tell *Mathematica* what to do at each step.

Defining functions

Suppose you want to define $f(x,y) = \sqrt{x^2+y^2}$, then there are three main ways to implement this in *Mathematica*:

```
myF1[x_, y_] := Sqrt[x^2 + y^2];
myF2 = Function[{x,y}, Sqrt[x^2 + y^2]];
myF3 = Sqrt[#1^2 + #2^2]&;
{myF1[a, b], myF2[c, d], myF3[e, f]}
```

$$\left\{\sqrt{a^2+b^2}, \sqrt{c^2+d^2}, \sqrt{e^2+f^2}\right\}$$

The first is the standard way of defining a function and the second two are *pure* function definitions — pure function are very useful when using functional and rule-based programming.

The first thing to notice in *Mathematica* is that all built-in functions start with a capital letter and use square brackets; so begin all your functions and variables with a lower-case letter. Parentheses are only used to group algebraic or logical expressions, such as `(a + b)^15` or `(a && b) || (c && d)`. Curly braces are only used to make ordered lists, such as `{eskimo, Pi, {4, u}}`.

There are several types of 'equals' in *Mathematica* that you'll likely use:

- `a = 5` assigns `a` the value 5 and `a=.` clears `a`;
- `a*x^2 + b*x + c == 0` is used in `Solve`, `DSolve`, etc.;
- `1 == 1.` returns `True` even though the LHS is an integer and the RHS is a float;
- `1 === 1.` returns `False` because an integer isn't the `SameQ` as a float — note that `===` always returns either `True` or `False` while `==` only does when both sides simplify to literal values;
- `a := Random[]` is a delayed equal and will wait until `a` is evaluated to compute the RHS, so *Mathematica* will give a different random real number in $[0,1]$ each time `a` is evaluated;
- there's also not equal `!=`, add to `+=`, subtract from `-=`, and up set `^=`.

Let's start with the Fibonacci sequence, which is defined by $F_n = F_{n-1} + F_{n-2}$, $F_0 = 0$, and $F_1 = 1$. You could just type:

```
fiboBad[n_] := fiboBad[n - 1] + fiboBad[n - 2];
fiboBad[0] = 0; fiboBad[1] = 1;
```

Here `n_` is a 'pattern object': whenever *Mathematica* sees `fiboBad` with a single object enclosed in square brackets, it associates that object with the variable `n` when it evaluates the RHS. There's also the pattern object `__` that matches one or more objects and `___` that matches one, more, or no objects.

I called this definition of F_n "`fiboBad`" because it takes $\mathcal{O}(F_n)$ operations to evaluate:

```
Timing[fiboBad[30]]
```

$$\{3.766 \text{ seconds}, 832\,040\}$$

A better definition is

```
fiboGood[n_Integer] := fiboGood[n] =
  fiboGood[n - 1] + fiboGood[n - 2] /; n > 1;
fiboGood[0] = 0; fiboGood[1] = 1;
Timing[fiboGood[30]]
```

$$\{5.42101 \times 10^{-19} \text{ seconds}, 832\,040\}$$

since it only takes $\mathcal{O}(n)$ operations. By typing `n_Integer` you restrict the definition to objects that are integers and "`/; n > 1`" further restricts it to integers that are greater than one. Restricting the pattern in function definitions is a very good programming habit; but it's only necessary if you're planning to 'overload' the function or plan to distribute your code. If you want to overload the definition of `fiboGood`, you might type

```
fiboGood[x_] := (GoldenRatio^x -
  (1 - GoldenRatio)^x)/Sqrt[5];
```

Since *Mathematica* uses the first pattern that it matches, `fiboGood` will use the recursive definition for integers and the the closed-form expression for everything else. If you'd entered the closed-form definition first, then the recursive definition would never be used.

Mathematica loves lists

Mathematica loves functional programming and functional programming loves lists — so *Mathematica* loves lists!

One of my favorite things is to `Map` functions onto the entries of a list. For instance, I can square each entry of a list by mapping the pure function `#^2&` onto each entry of the list using `/@`:

```
#^2& /@ {I, like, Pi, 2}
```

$$\{-1, \text{like}^2, \pi^2, 4\}$$

Let's suppose you want to discuss the concept of error with your class (à la *The Feynman Lectures on Physics*). Say you want *Mathematica* to do 100, 400, and 1 600 trials of flipping a coin 30 times and comparing their histograms with the binomial distribution.

You might model a fair coin toss in *Mathematica* by

```
coinToss := If[Random[] < 0.5, 1, 0];
```

where 1 represents a "head" and 0 represents a "tail". To get the number of heads in 30 tosses, you might define

```
trial := Sum[coinToss, {30}];
```

or equivalently

```
trial := Plus @@ Table[coinToss, {30}];
```

The first definition of `trial` just uses `Sum`. The second definition uses `Table` — one of the functions you'll use the most in *Mathematica* — and the lower level functions `Plus` and `Apply` (`@@`). `Table` is used to make lists of objects — for instance, `Table[Prime[ii], {ii, 100}]` makes a list of the first one hundred prime numbers. In the definition of `trial` you get a list like $\{1, 0, 0, 1, 1, \ldots, 0, 0\}$; to get a sum, you can just replace the `Head` (which is `List`) with `Plus` by typing "`Plus @@`". To see why this works, it is helpful to see how *Mathematica* internally represents what you type using `FullForm`:

```
FullForm[{a + b*x + c*x^2 == 0}]
```

gives

```
List[Equal[Plus[a,Times[b,x],
  Times[c,Power[x,2]]],0]]
```

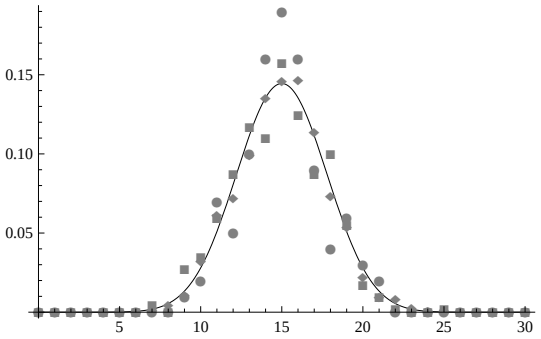
Now, let's write a little function that does n trials and returns the ratio of trials that have 0, 1, …, 30 heads:

```
coinHisto[n_Integer] :=
  Module[{trials}, (* Local Variable *)
    trials = Table[trial, {n}];
    Return[
      Table[{ii, Count[trials, ii]/n}, {ii, 0, 30}]
    ]] /; n > 0;
```

When you need to write a function with several steps, you can use `Module` to hold these steps and protect any local variables that you might need. In this case, you have two steps and you'll want to protect the local variable `trials`. Here, `Table[trial, {n}]` just makes a list with `n` trials and `Table[{ii, Count[trials, ii]/n}, {ii, 0, 30}]` makes a list of lists where the first entry is the number of heads and the second is the frequency that number of heads appeared. You don't actually need to use `Return` — since `Module` returns the value of the last line — but it's good programming practice.

If you want to plot a few histograms, say with 100, 400 and 1 600 trials, you can use `ListPlot`:

```
Show[Plot[Binomial[30, x]/2^30, {x, 0, 30}],
 ListPlot[{coinHisto[100], coinHisto[400],
   coinHisto[1600]}, PlotMarkers -> Automatic],
 PlotRange -> All]
```



The `Show` function allows you to combine the plots into a single plot. Here `Binomial[30, x]/2^30` is the probability mass function (PMF) of the binomial distribution when each trial consists of 30 tosses of a fair coin and `PlotRange -> All` ensures that the all the dots show up.

While this plot is nice, it doesn't show that you expect the difference between your histograms (of N trials) and the PMF of the binomial distribution to be proportional to $1/\sqrt{N}$. So let's do a hundred sets of trials for $N = 100$, 400, 1 600, and 6 400 and make a quartile plot of the histograms.

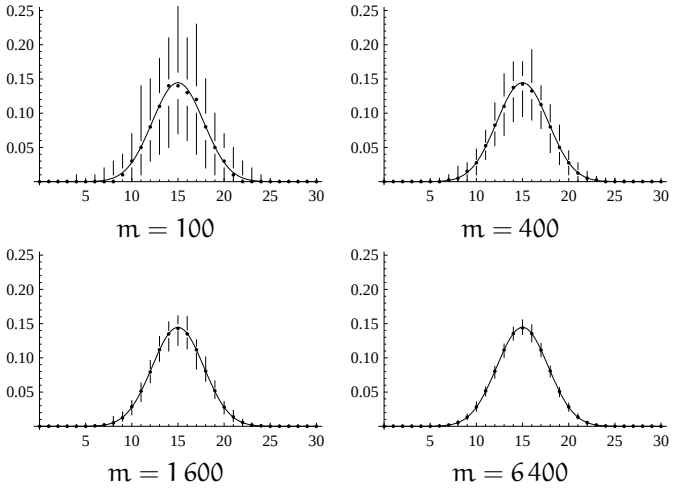
```
coinHistoQuartileLines[n_,m_]:=
Module[{data,quartiles,linesAndDots},
  data = Transpose /@
    Transpose[Table[coinHisto[n], {m}]];
  quartiles =
    { #[[1, 1]],
      Quantile[#[[2]], {0, 1/4, 1/2, 3/4, 1}]
    }& /@ data;
  linesAndDots =
    { Line[{{#[[1]],#[[2,1]]},{#[[1]],#[[2,2]]}}],
      Point[{#[[1]],#[[2,3]]}],
      Line[{{#[[1]],#[[2,4]]},{#[[1]],#[[2,5]]}}]
    }& /@ quartiles;
  Return[Flatten[linesAndDots]]];
```

Here I use the graphical primitives `Line` and `Point`, the function `Quartile`, the `Part` (`[[…]]`) function to select elements from the lists, and a bunch of `Maps` (`/@`) and pure functions.

Then using $m = 100$, 400, 1 600, and 6 400 in

```
Show[Plot[Binomial[30, x]/2^30, {x, 0, 30}],
  Graphics[{PointSize[Small],
    coinHistoQuartileLines[m, 100]}],
  PlotRange -> {0, 0.25}]
```

gives the Tufte-style quartile plots:



(If you haven't already, I highly recommend reading Edward R. Tufte's *The Visual Display of Quantitative Information* 2nd ed. and *Envisioning Information*.)

Pattern matching is your friend!

Suppose you're given the Korteweg–de Vries equation,

$$u_t + 6uu_x + u_{xxx} = 0, \qquad \text{(KdV)}$$

which models phenomena with weak nonlinearity and weak dispersion, and you want to verify that

$$u(x,t) = 2\kappa^2 \operatorname{sech}^2\{\kappa(x - 4\kappa^2 t - x_0)\}$$

satisfies (KdV). You can enter (KdV) into *Mathematica* as

```
kdv = D[u[x,t], t] + 6*u[x,t]*D[u[x,t], x] +
  D[u[x,t], {x, 3}];
```

If you try to type:

```
kdv /. u[x,t] :> 2*k^2*Sech[k*(x - 4*k^2*t - x0)]^2
```

you'll get

$$u_t + 12\kappa^2\,\mathrm{sech}^2\{\kappa(x-4\kappa^2t-x_0)\}u_x + u_{xxx},$$

which isn't what you wanted at all! Let's use our friend `FullForm` to figure out what went wrong:

```
FullForm[kdv]
```

gives

```
Plus[Derivative[0,1][u][x,t],
  Times[6,u[x,t],Derivative[1,0][u][x,t]],
  Derivative[3,0][u][x,t]]
```

Ah ha! So *Mathematica* didn't replace the derivatives because `[u][x,t]` is not the same pattern as `u[x,t]`. You can get around this by using a pure function in your rule:

```
Simplify[kdv /. u -> Function[{x, t},
    2*k^2*Sech[k*(x - 4*k^2*t - x0)]^2]]
```

returns 0, which is exactly what you wanted.

Real world example: Painlevé analysis

The Painlevé test is a popular method for determining if a nonlinear partial differential equation (PDE) is likely to be completely integrable — that is, if it has special properties that allow it to be solved using the inverse scattering transform. A differential equation has the Painlevé property if all the movable singularities of all its solutions are poles — a singularity is movable if it depends on the constants of integration. We'll test this by assuming that the solution is a Laurent series,

$$u(x,t) = g^\alpha(x,t)\sum_{k=0}^{\infty} u_k(x,t)g^k(x,t),$$

where α is a negative integer, and verifying that it's the general solution; since a Laurent series only has movable poles (and possibly a movable essential singularity) then such a solution doesn't have any movable algebraic or logarithmic branch cuts and so you say that it passes the Painlevé test.

First you need to determine what α is. So you substitute $u(x,t) = u_0(x,t)g^\alpha(x,t)$ into the PDE and pull of the exponents of $g(x,t)$ (since there must be a balance from at least two terms):

```
Exponent[kdv /.
    u -> Function[{x, t}, u0[x,t]*g[x, t]^alpha],
  g[x, t], List]
```

$$\{-3+\alpha, -2+\alpha, -1+\alpha, -1+2\alpha\}$$

Clearly the balance must come from $\alpha-3$ and $2\alpha-1$ so you find that $\alpha=-2$ and the dominant power of $g(x,t)$ is -5. With $\alpha=-2$, you can solve for u_0:

```
Solve[
  Coefficient[kdv /. u -> Function[{x, t},
    u0[x, t]*g[x, t]^-2], g[x, t], -5] == 0,
  u0[x, t]]
```

gives $u_0(x,t) = -2g_x^2(x,t)$. (We require that $g_x(x,t)\not\equiv 0$ on the manifold $g(x,t)=0$ so we can use the Cauchy–Kovalevskaya theorem to give local existence and uniqueness of the solution.)

Then, for the series to be a general solution, you need three arbitrary functions in your series solution because (KdV) is a third order PDE. Thus, you substitute

$$u(x,t) = -2g_x^2(x,t)g^{-2}(x,t) + u_r(x,t)g^{r-2}(x,t)$$

into (KdV) and determine at which r that $u_r(x,t)$ is arbitrary by setting the coefficients of the dominant term (g^{r-5}) to zero:

```
Factor[
  Coefficient[
    kdv /. u -> Function[{x, t},
      -2*D[g[x, t], x]^2*g[x, t]^-2 +
        ur[x, t]*g[x, t]^(r - 2)],
  g[x, t], r - 5]] == 0
```

gives

$$(r-6)(r-4)(r+1)u_r(x,t)g_x^3(x,t) = 0$$

Thus, $u_r(x,t)$ should be arbitrary when $r=-1$, 4, and 6; the universal *resonance* at $r=-1$ corresponds to the arbitrary function $g(x,t)$. For more complicated equation, you can simplify your calculations by taking $g(x,t) = x - h(t)$ by the implicit function theorem (since $g(x,t)$ is non-characteristic, $g_x\not\equiv 0$, on the manifold $g(x,t)=0$).

Finally, you must verify that $u_r(x,t)$ really is arbitrary at $r=4$ and 6 by substituting $u(x,t) = -2g_x^2(x,t)g^{-2}(x,t) + u_1(x,t)g^{-1}(x,t) + \cdots + u_6(x,t)g^4(x,t)$ into (KdV) and solving for $u_1(x,t)$, …, $u_6(x,t)$.

```
theCoefRules =
  {u[0] -> Function[{x, t}, -2*D[g[x, t], x]^2]};
seriesSoln =
  Sum[u[ii][x, t]*g[x, t]^(ii - 2), {ii, 0, 6}] /.
    theCoefRules;
eqs =
  Table[
    Coefficient[
      kdv /. u -> Function[{x, t},
        Evaluate[seriesSoln]], (* Why do I use *)
      g[x, t], ii - 5] == 0,   (* "Evaluate"?  *)
    {ii, 1, 6}];
Do[
  Solve[eqs[[ii]] /. theCoefRules, u[ii][x, t]
    ] /. Rule[a_[var__], b_] :>
      AppendTo[theCoefRules,
        a -> Function[{var}, b]],
  {ii, 1, 6}];
theCoefRules
```

gives $u_1(x,t) = 2g_{xx}$, $u_2(x,t) = -(g_tg_x^2 + 3g_xg_{xx}^2 - 4g_x^2g_{xxx})/(6g_x^3)$, etc. Solve doesn't find expressions for $u_4(x,t)$ and $u_6(x,t)$; checking the equation for $r=4$ and 6 you see that $u_4(x,t)$ and $u_6(x,t)$ are indeed arbitrary:

```
Simplify[{eqs[[4]], eqs[[6]]} /. theCoefRules]
```

$$\{\texttt{True}, \texttt{True}\}$$

Therefore, you conclude that the KdV equation passes the Painlevé test — you should check that it doesn't have an essential movable singularity, but no one actually does.