Proposed New Software for Plotting, Derivatives, and Integrals for Math 135

Danny Kaplan

May 24, 2011

The software we used for Math 135 in 2010-2011 included operators such as `D` and `antiD` that took functions as arguments and returned functions as values, a built-in plotting program for functions of one variable, `curve`, and a home-made function, `contour.plot`, for plotting functions of two variables. This worked reasonably well, but there were some shortcomings:

- An inconsistency in syntax between `curve` and the other functions `D`, `antiD`, and `contour.plot`. Whereas `curve` can take an expression as the function to be plotted, the other functions cannot. Students (and instructors!) found it convenient to be able to define functions with expressions.
- The argument **always** used in `curve` is x, which undermines our desire to give students a more general view of functions than "something that takes x."
- Although partial differentiation is part of the course, there was no way to take a partial derivative in the software.
- Parameters were often stored in the global environment. Changing them involved a separate assignment statement; the values could not be seen directly in the statement at hand.
- No connection to lattice. Randy Pruim wrote an `fplot` that works with lattice, but doesn't evaluate expressions like `curve`.

In an attempt to address these issues, I have drafted a new set of operators to replace `D`, `antiD`, `curve`, and `contour.plot`. I have provisionally (and awkwardly) called these new functions `newD`, `newAntiD`, and `plotFun` — they would likely be renamed `D`, `antiD`, and `plt`.

The goal of this note is to introduce these new operators and give us a chance to discuss the syntax being used. I've tried to make this syntax relatively flexible and consistent with the overall modeling paradigm within R. Some changes are possible, but limited by the language and by my limited understanding of the language.

The new software is contained in a file `plt.R` which you can source into your R session:

```
> source("plt.R")
```

1 Basic Syntax

The software is built around the idea of a formula of this form:

Mathematical expression ~ variables to work with.

For example

```
A*sin(2*pi*t/P)*exp(-k*t) ~ t&k
```

Any symbolic quantities — `A`, `t`, `P`, and `k` in the above — can be left as free variables or assigned values through the usual argument assignment mechanism. For some purposes, e.g., plotting out a function or finding zeros, it's essential that all free variables be given numerical values. For other purposes, e.g., integration or differentiation, the variables can be left free or can optionally be assigned default values.

For instance, here's the command to plot out the expression specified above as a function of variables `t` and `k`:

```
> plotFun(A*sin(2*pi*t/P)*exp(-k*t) ~ t&k, A=3, P=5, k=range(0.1,0.5), t=range(0,10) )
```

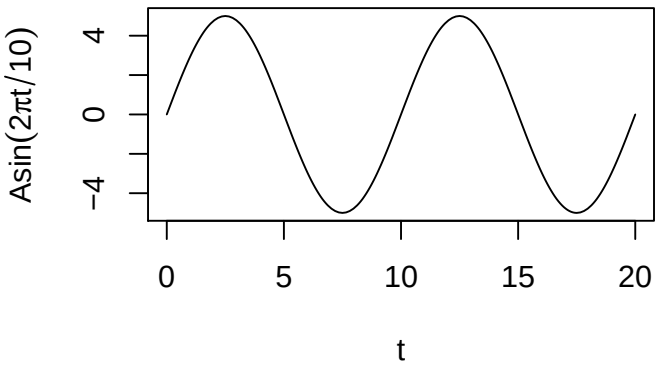
By design, `pi` can never be used as a free variable: it's always taken as a number.

DETAIL: Don't name a free variable with a starting period (.), e.g. `.a`. Other than that, all legitimate variable names should work. If you run into problems, let me know. Sometimes there is a conflict between free variables and variables used in the software — these are bugs.

2 Plotting Software

The plotting software is designed to allow expressions involving any variables to be used and designated as the axes. The same software handles both functions of one and two variables. For example, here is a plot involving two variables, A and t. The variable t is identified as the independent variable by putting it to the right of the `~`. Later in the statement, the independent variable is given a range of values and parameter A is given a specific numerical value.

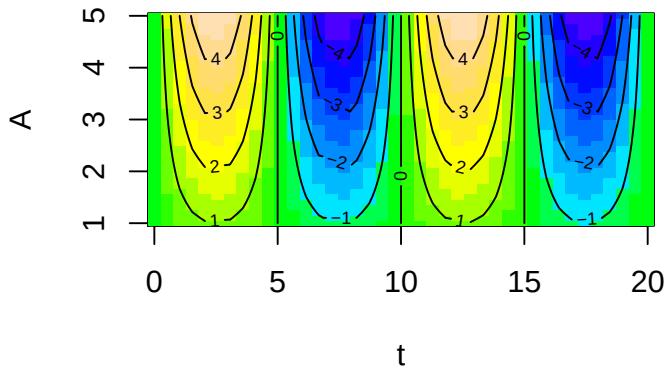
```
> plotFun( A*sin(2*pi*t/10) ~ t, t=range(0,20), A=5)
```



Failure to set A results in an error message: `object 'A' not found`.

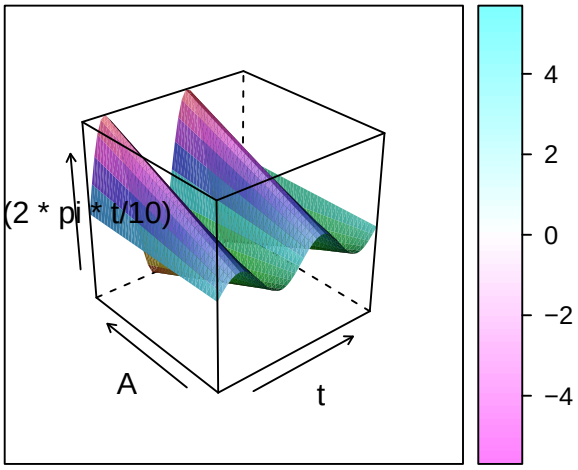
By specifying two independent variables, and giving them a range, a contour or surface plot will be made. (This function could be extended to serve as a panel function in the lattice graphics.)

```
> plotFun( A*sin(2*pi*t/10) ~ t&A, t=range(0,20), A=range(1,5))
```



Using the `surface=TRUE` optional argument will cause a surface plot to be generated. In RStudio, this comes with a slider to rotate the plot.

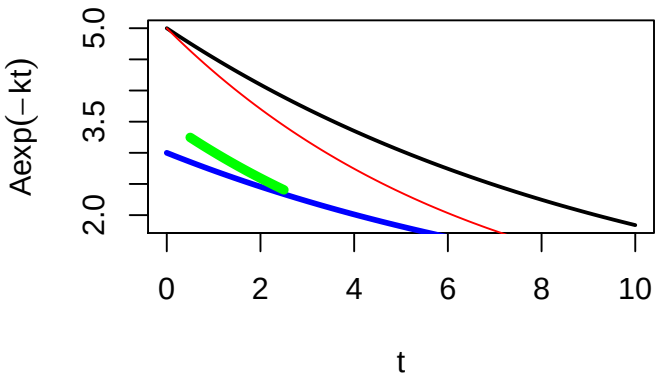
```
> plotFun( A*sin(2*pi*t/10) ~ t&A, t=range(0,20), A=range(1,5),surface=TRUE)
```



FOR THE FUTURE: I can imagine extending this to allow a range of one extra variable to be specified, with the result being a series of plots.

The value returned (invisibly) from `plotFun` is a function based on the expression given in the first argument. Each of the variables in the expression will be assigned a default value based on assignments in the call to `plotFun`. You can either use these default values, or change them. For example:

```
> f = plotFun( A*exp(-k*t) ~ t, A=5, k=.1, t=range(0,10),lwd=2)
> plotFun( f(t=t,k=.15) ~ t, col="red",add=TRUE)
> plotFun( f(t=t,A=3) ~ t, col="blue", lwd=3,add=TRUE)
> plotFun( f(t=t,k=.15,A=3.5) ~ t, add=TRUE, t=range(.5,2.5), col="green", lwd=5)
```

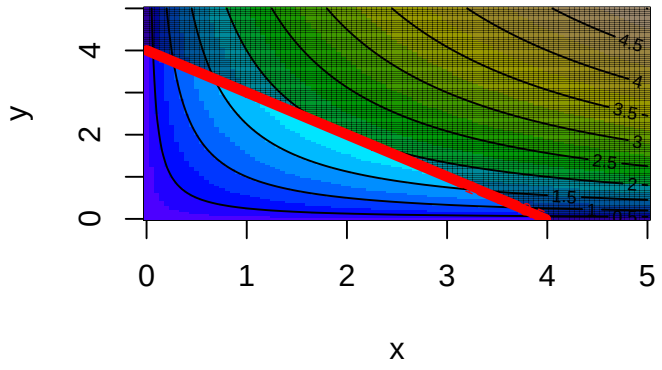


Notice that, since `add=TRUE` was used in the middle two calls to `plotFun`, the limits have been inferred from the existing plot. In the fourth call, limits have been set explicitly and the `add` argument set so that overplotting is done.

If the additional plot is outside of the current y-axis range, a warning message is given.

Example: Constraint Functions Inequality constraints can be plotted out by using a logical expression. Equality constraints are problematic, since they are typically a set of measure 0. But equality constraints can be plotted out as inequality constraints, then considering only the border.

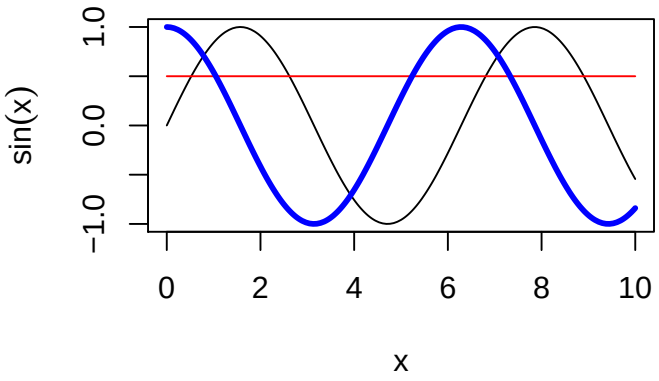
```
> plotFun( x^.5 * y^.5 ~ x&y, x=range(0,5), y=range(0,5), npts=100)
> plotFun( (x + y < 4) ~ x&y, add=TRUE,col="red", lwd=5,npts=100,filled=TRUE)
```



The bright area shows regions where the constraint is satisfied. It's effective to set `transparency=0` when using an inequality to represent an equality constraint.

Backward Compatibility The `plotFun` function tries to behave in a way that's compatible with `curve` or `fplot`: you can plot numerical constants, function objects, and strings that name functions.

```
> plotFun(sin, xlim=c(0,10))
> plotFun(.5, add=TRUE,col="red")
> plotFun("cos",add=TRUE, col="blue", lwd=3)
```



Of these, only the numerical constant is something to be encouraged, but the other forms might be employed by those used to `curve`.

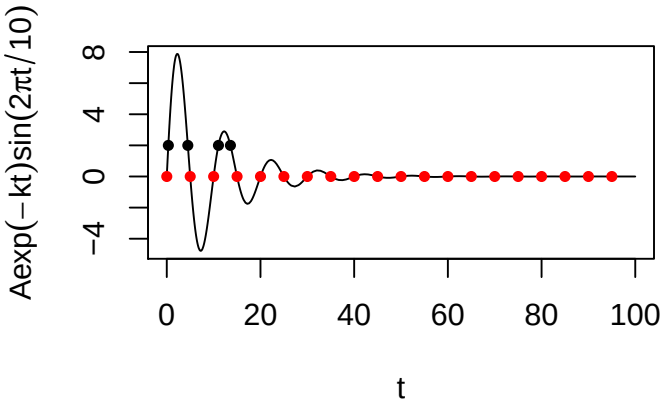
TO DO: Add a "label" argument along with a "where" argument that takes a value of the independent variable. It will put the label on the function at the point specified by "where".

TO DO: Produce a lattice version.

3 Finding Zeros

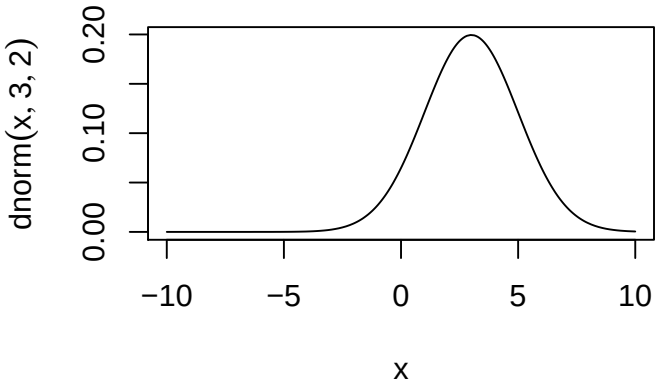
The `findZeros` operator takes a function of one variable, perhaps produced by an expression as in `plotFun`, and finds the roots of that function over some specified interval. If a plot is showing and the interval isn't explicitly stated, `findZeros` will look for the interval and solution variable in the plot.

```
> plotFun(A*exp(-k*t)*sin(2*pi*t/10) ~ t, A=10, k=.1,t=c(0,100))
> z = findZeros(A*exp(-k*t)*sin(2*pi*t/10) ~ t, A=10, k=.1)
> z

 [1]  0.000000  4.999984 10.000000 15.000000 20.000000
 [6] 25.000000 30.000000 35.000000 40.000000 45.000000
[11] 50.000000 55.000001 60.000001 65.000001 70.000001
[16] 75.000000 80.000000 85.000000 90.000000 94.999990

> # add them to the plot
> points(z,0+0*z,pch=20,col="red")
> # find the places where the function crosses 2
> z2 = findZeros(A*exp(-k*t)*sin(2*pi*t/10) - 2 ~ t, A=10, k=.1)
> points(z2,2+0*z2, pch=20, col="black")
```



The `findZeros` operator breaks up the interval into a "large" number of segments, and searches each segment for a zero. You can control the number of segments by setting the optional argument `npts`, which by default is 1000. It will make a reasonable attempt even when the interval requested is very large by using logarithmic spacing in the direction of very large endpoints.

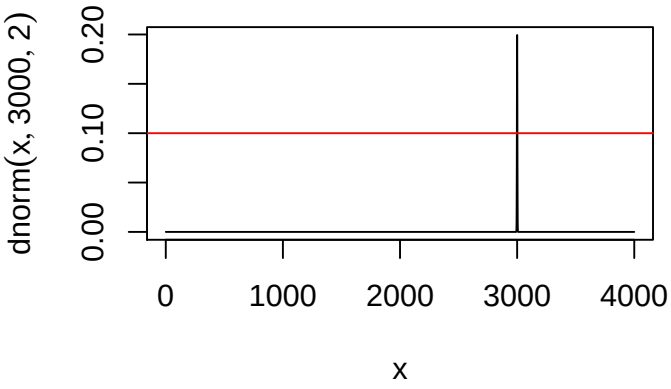
```
> plotFun( dnorm(x, mean=3, sd=2)~x, xlim=c(-10,10))
> findZeros( dnorm(x, mean=3, sd=2)-.1~x, x=c(-10,10))

[1] 0.6496804 5.3503241

> findZeros( dnorm(x, mean=3, sd=2)-.1~x, x=c(-Inf,Inf))

[1] 0.6496804 5.3503241
```



The goal is not to find every zero (although that would be nice!), but all the zeros that might be detectable in a plot. So, if the function stays on each side of zero for adjacent intervals that are bigger than about `1/npts` of the range, the intervening zero should be detected. You can fool `findZeros` by giving it a search range which is much larget than the interval over which the sign changes.

This interval is small enough to detect the crossing above the threshold of 0.1

```
> plotFun(dnorm(x, mean=3000, sd=2) ~ x, x=range(0,4000))
> abline(.1,0,col="red")
> findZeros( dnorm(x, mean=3000, sd=2)-0.1 ~ x, x=range(0,4000))

[1] 2997.65 3002.35
```



But the following interval is just a bit too big and the spike visible in the graph was too narrow to be detected:

```
> findZeros( dnorm(x, mean=3000, sd=0.2)- 0.1 ~ x, x=range(0,10000))

NULL
```

TO DO: It would be nice to extend `findZeros` to take functions of multiple (2 would do) variables.

4 Differentiation

The new operator, like the old one, returns a function. But unlike the old operator, the function returned by the new operator can have multiple variables as inputs. The variable with respect to which differentiation is to take place is explicitly identified with the ~ syntax.

```
> dfdt = newD( A*exp(-k*t)*sin(2*pi*t) ~ t )
```

Since there is no formal argument list, the order of arguments is somewhat arbitrary. It can be seen with the `args` command

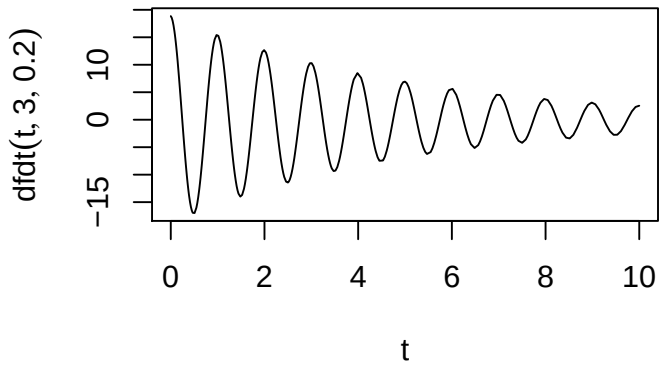
```
> args(dfdt)

function (t = NA, A = NA, k = NA)
NULL
```

In evaluating the function, the "named arguments" technique should be used to avoid dependence on the order of the arguments in the function:

```
> dfdt(t=10,k=.2,A=3)
[1] 2.550959
```

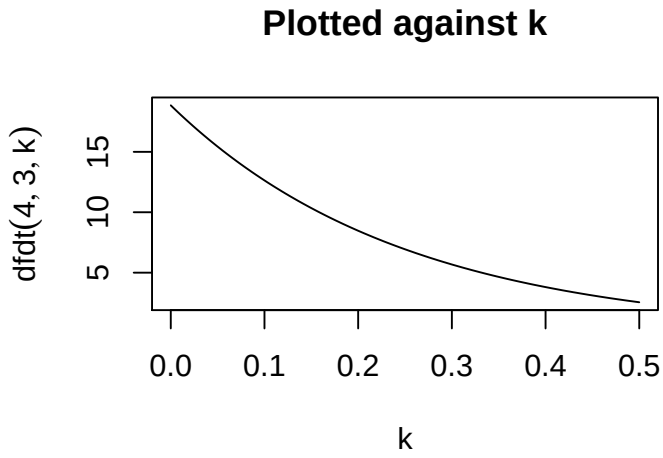
This can be relaxed somewhat in plotting the function:

```
> plotFun(dfdt(t,A=3,k=.2)~ t, t=range(0,10))
```



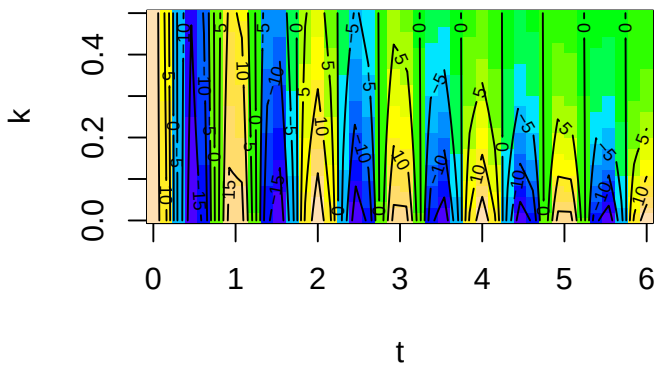
Of course, you might wish to plot the function against another variable, e.g. k:

```
> plotFun(dfdt(t=4,A=3,k=k)~ k, k=range(0,.5),main="Plotted against k")
```



Or, even to make a plot of the function against two variables:

```
> plotFun(dfdt(t=t,A=3,k=k)~t & k, t=range(0,6),k=range(0,.5))
```



Derivatives can, like any other function, be differentiated. With the new system, we can take partial derivatives including mixed partials.

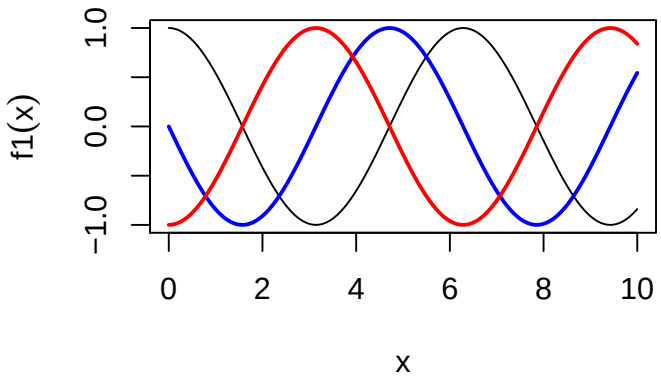
Here's a demonstration that the mixed partials are equal.

```
> f = function(t,A,k){A*exp(-k*t)*sin(2*pi*t)}
> dfdt = newD(f(t,A,k) ~ t)
> args(dfdt)

function (t = NA, A = NA, k = NA)
NULL

> dfdk = newD(f(t,A,k)~ k)
> args(dfdk)

function (k = NA, t = NA, A = NA)
NULL

> dfdtk = newD(dfdt(t=t,A=A,k=k)~ k)
> dfdkt = newD(dfdk(t=t,A=A,k=k)~t )
> dfdkt(t=1,A=3,k=.1)

[1] -17.05646

> dfdtk(t=1,A=3,k=.1)

[1] -17.05646
```

Notice that the assignment form for arguments is being used. This is important because the order of arguments in `dfdt` and `dfdk` is different.

The new function can be used in the same way as the old `D`, if desired:

```
> f1 = newD(sin)
> f1(0)

[1] 1

> f2 = newD(f1)
> f3 = newD(f2)
> f4 = newD(f3)
> plotFun(f1(x)~x, x=range(0,10))
> plotFun(f2(x)~x, col="blue", lwd=2, add=TRUE)
> plotFun(f3(x)~x, add=TRUE, col="red", lwd=2)
```



The choice of finite-difference $h = 0.0001$ is set to make things work reasonably for sin-like functions up to the third derivative. (A larger h is needed for higher-order functions.)

TO DO: Define higher-order derivatives with a notation like `~x + y` or `~x+x` or perhaps `~ x then y`.

TO DO: Possible capstone/honors project: implement the calculation via automatic differentiation.

5 Anti-Differentiation

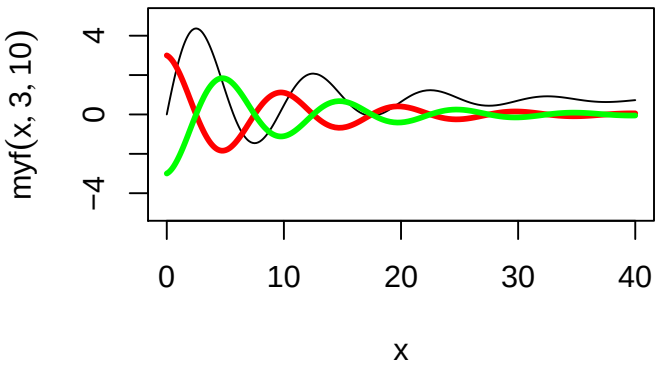
The new `newAntiD` operator lets you specify which variable is to be used as the variable of integration. It returns a function with **two** arguments to replace the variable of integration: `from` and `two`. (Any other variables in to the original function remain as variables to the function that is returned by `newAntiD`.) The default value of `from` is 0, but it's easy to change this.

```
> mylog = newAntiD(1/x ~ x, from=1)
> mylog( to = 1)
```

```
[1] 0

> mylog( to = exp(3))

[1] 3

> mylog( to = 10)

[1] 2.302585

> log(10)

[1] 2.302585

> mylog( from=10, to=1)

[1] -2.302585
```

Here's an example involving a "symbolic" variable, A, which can be held unset until the time that the integral needs to be evaluated, e.g., when plotting it.

```
> myf = newAntiD( A*exp(-.1*t)*cos(2*pi*t/P) ~ t)
> args(myf)

function (to, from = input0, A = NA, P = NA)
NULL

> plotFun(myf(to=x,A=3,P=10)~x, x=range(0,40), ylim=c(-5,5))
> myf2 = newD(myf(from=0,to=x,A=3,P=10)~x)
> args(myf2)

function (x = NA)
NULL

> plotFun(myf2(x)~x, col="red",lwd=3,add=TRUE)
> myf3 = newD(myf(from=x,to=0,A=3,P=10)~x)
> args(myf3)

function (x = NA)
NULL

> plotFun(myf3(x)~x, col="green",lwd=3,add=TRUE)
```



Notice that setting numerical values for the free variables removes them from the argument list of the resulting function. That is, `myf2` does **not** have `A` and `P` as arguments.

You can also leave the parameters free in the differentiation operation, evaluating them only when needed. This is done with the following assignment syntax

```
> myf2B = newD(myf(to=x,A=A,P=P) ~ x)
> args(myf2B)

function (x = NA, A = NA, P = NA)
NULL

> myf2B( x=1,A=3,P=10 )

[1] 2.196025

> myf2( x=1 )

[1] 2.196025
```

The point of the `A=A` is to make sure that assignment is being done to the right variable. In this case, we're making a new free variable `A`, and assigning it's value to the argument `A` in `myf`. You could, of course, choose to use other free variable names, for instance `Ralph` and `Fido`, like this:

```
> myf2B = newD(myf(to=x,A=Ralph,P=Fido) ~ x)
> args(myf2B)

function (x = NA, Ralph = NA, Fido = NA)
NULL
```

```
> myf2B( x=1, Ralph=3, Fido=10 )
```

```
[1] 2.196025
```

You still need to do assignment to `A` and `P`, since those are the arguments to `myf2B`, but the new function doesn't have to keep those names.

To assign default values to the parameters, give additional arguments setting the parameters, e.g.

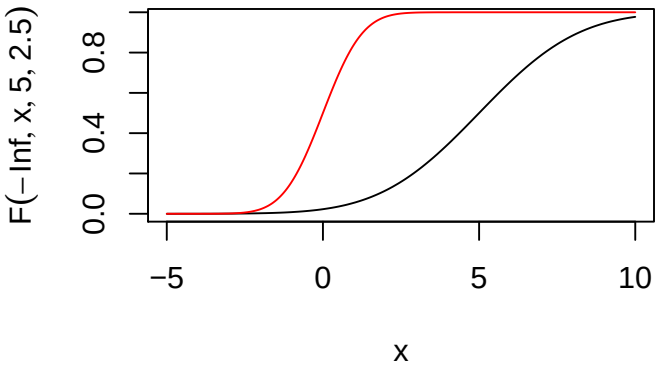
```
> myf2C = newD(myf(to=x,A=A,P=P) ~ x, A=3, P=10)
> myf2C( x=1 )
```

```
[1] 2.196025
```

Unfortunately, default parameters cannot be picked up automatically when you create a new function. I suspect it would be meaningless to do so, since the expression defining the new function might involve multiple functions with different values of the default parameters.

Example of free variables in an integral:

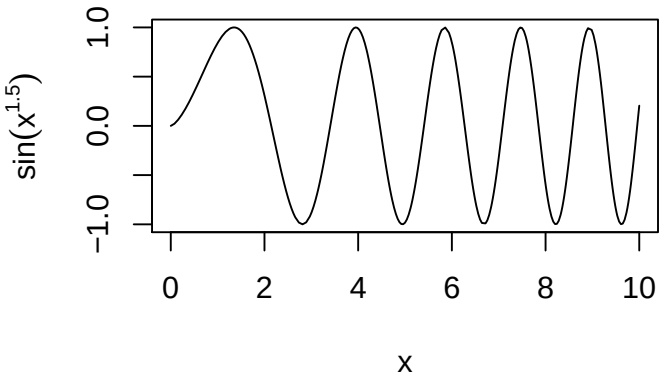
```
> F = newAntiD( dnorm(x, mean=m, sd=s) ~ x )
> plotFun(F(from=-Inf,to=x,m=5,s=2.5)~x, x=range(-5,10))
> plotFun(F(from=-Inf,to=x,m=0,s=1)~x, col="red",add=TRUE)
```



6 Using these new functions in the course

The `plotFun` works much like the existing `curve`. Indeed, it can be used with much the same syntax, just setting an `xlim=` argument and saying which variable to use to plot, even when that's obvious from context.

```
> plotFun( sin(x^1.5)~x, xlim=range(0,10))
```



By combining the contour and surface plots of two variables under the same scheme, I think that plotting overall will be just as easy or easier to do.

For the `newD` and `newAntiD`, there are both advantages and disadvantages over the old functions. We will have to establish what features of the new functions to introduce in order to gain from the advantages without suffering excessively from the disadvantages.

I have tried to define the `newD` and `newAntiD` functions in a way that they can be used more or less as substitutes for the existing `D` and `antiD`.

If we are to use them in a more general way, including "symbolic" (that is to say, "unbound") variables, we will have to be careful to introduce the "named argument" syntax from the beginning and encourage students to examine the arguments of the functions they create. This might be for the good, since it may encourage students to focus more explicitly on the question, "What is f a function of?"

7 Test Examples

1. $f(x,k,A) = A\exp -kx$ gives $\frac{df}{dx} = -Ak\exp -kx$ and $\int_0^t f(x,k,A)dx = -\frac{A}{k}(\exp -kt - 1)$

```
> f = plotFun( A*exp(-k*x)~x, A=3, k=.1, x=range(0,5))
> fprime = newD(f(x) ~ x)
> F = newAntiD(f(x) ~ x)
> fprime2 = function(x,A,k){-A*k*exp(-k*x)}
> F2 = function(t,A,k){-(A/k)*(exp(-k*t)-1)}
> F(1:5) - F2(1:5,3,.1)
```

```
[1] -1.332268e-15 -8.881784e-16  8.881784e-16  0.000000e+00
[5]  1.776357e-15
```

```
> fprime(1:5) - fprime2(1:5,3,.1)
```

```
[1] 1.357254e-06 1.228095e-06 1.111223e-06 1.005476e-06
[5] 9.097943e-07
```

```
> fprime = newD(f(x,k=.5,A=10)~x)
> fprime(x=1:5) - fprime2(x=1:5,k=.5,A=10)
```

```
[1] 7.581507e-05 4.598416e-05 2.789081e-05 1.691663e-05
[5] 1.026045e-05
```

You can even pass variables that remain unevaluated until you evaluate the integral itself. In order to avoid trouble with the order of arguments, it's best to use the named form of assignment.

```
> F = newAntiD(A*exp(-k*x)~x)
> F(from=0,to=1:5,k=.5,A=10) - F2(t=1:5,k=.5,A=10)
```

```
[1] 8.881784e-16 1.776357e-15 1.776357e-15 3.552714e-15
[5] 3.552714e-15
```

When you plot a function that comes from `plotFun`, you can redefine the values of parameters by including them as free variables in the function being given to `newAntiD`. Again, you want to be careful to get the order right.

```
> args(f)
```

```
function (x = NA, A = 3, k = 0.1)
NULL
```

If you don't want to worry about the order, used named assignment to the variables, like this:

```
> F = newAntiD(f(x,k=k,A=A)~x)
> F(from=0,to=1:5,k=.5,A=10) - F2(t=1:5,k=.5,A=10)
```

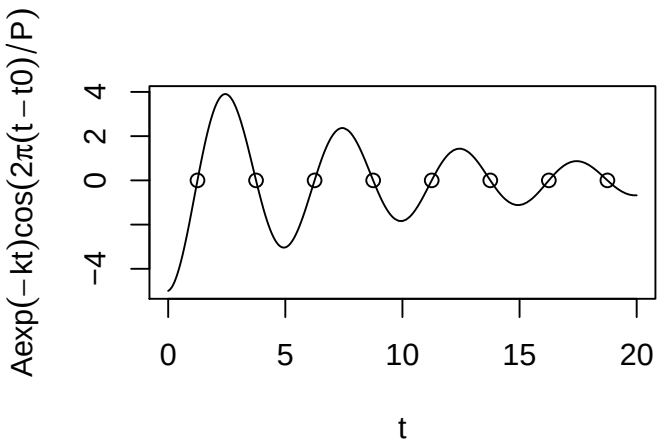
```
[1] 8.881784e-16 1.776357e-15 1.776357e-15 3.552714e-15
[5] 3.552714e-15
```

2. Finding zeros

```
> f = plotFun(A*exp(-k*t)*cos(2*pi*(t-t0)/P) ~ t,t=range(0,20),A=5,t0=2.5,k=.1,P=5)
> foo = findZeros(f(t)~t)
> f(t=foo)
```

```
[1]  5.505894e-05 -3.793452e-05  2.122519e-05 -6.440590e-06
[5]  4.565796e-06 -9.375027e-06  1.043940e-05 -9.440236e-06
```

```
> points(foo,0+0*foo)
```



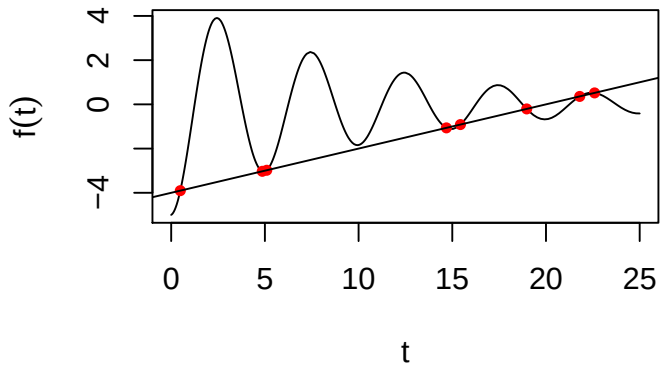
Change the period of the function so that the zeros will be spaced by 1

```
> findZeros(f(t,P=2)~t)
```

```
 [1]  1.000001  2.000003  3.000004  4.000005  5.000006
 [6]  6.000007  7.000008  8.000009  9.000010 10.000010
[11] 11.000010 12.000010 13.000010 14.000010 15.000009
[16] 16.000008 17.000006 18.000005 19.000002
```

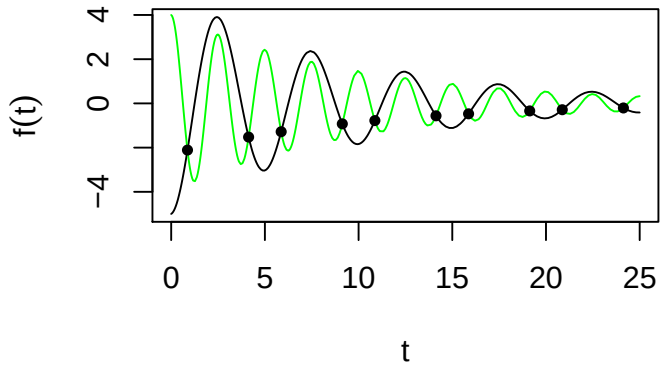
Find the places where the function equals $t/5 - 4$

```
> plotFun( f(t) ~ t, t=range(0,25))
> goo = findZeros(f(t) - (t/5 - 4) ~t)
> points( goo, goo/5 - 4, pch=20,col="red")
> abline(-4,1/5)
```



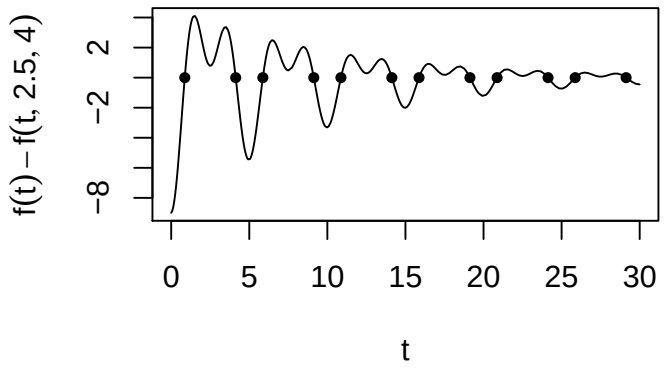
Find the places where two such functions intersect.

```
> plotFun( f(t) ~ t, t=range(0,25))
> f2 = plotFun( f(t,P=2.5,A=4) ~ t, add=TRUE, col="green")
> foo = findZeros(f(t)-f2(t)~t)
> points(foo,f(foo),pch=20)
```



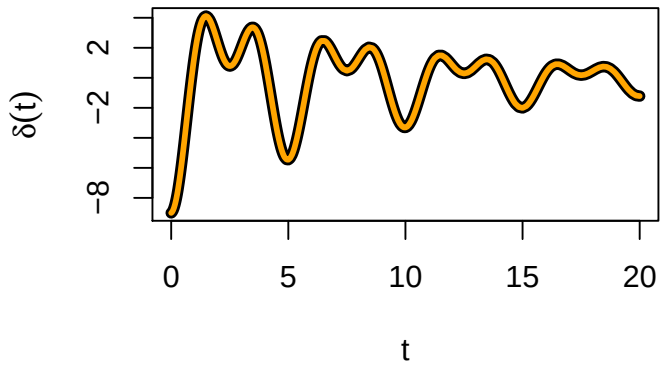
Using different default parameters with the same function:

```
> delta = plotFun( f(t) - f(t,P=2.5,A=4) ~ t,t=range(0,30))
> foo = findZeros(delta(t)~t)
> points(foo,0+0*foo, pch=20)
```



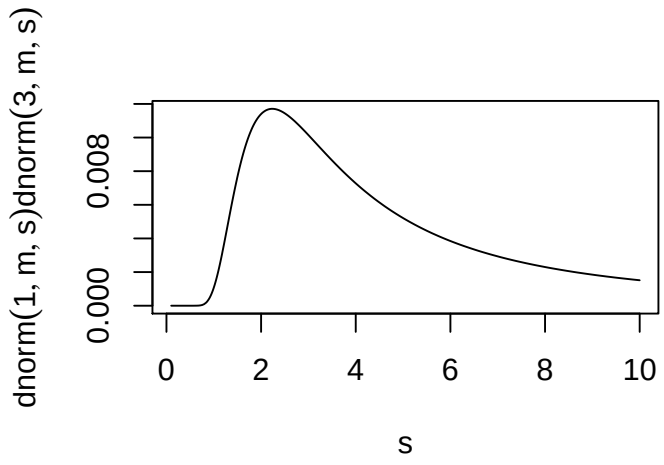
3. Showing that the derivative undoes the anti-derivative. Note the use of the named assignment to argument `to=t`.

```
> plotFun( delta(t) ~ t, t = range(0,20),lwd=6)
> g = newAntiD(delta(t) ~ t)
> h = newD(g(to=t)~t)
> x = seq(0,100,by=10)
> delta(x) - h(x)

 [1] -4.505512e-05 -1.657485e-05 -6.097515e-06 -2.243106e-06
 [5] -8.251680e-07 -3.035362e-07 -1.116404e-07 -4.104577e-08
 [9] -1.507563e-08 -5.521573e-09 -2.006930e-09

> plotFun(h(t)~t, lwd=3, add=TRUE, col="orange")
```



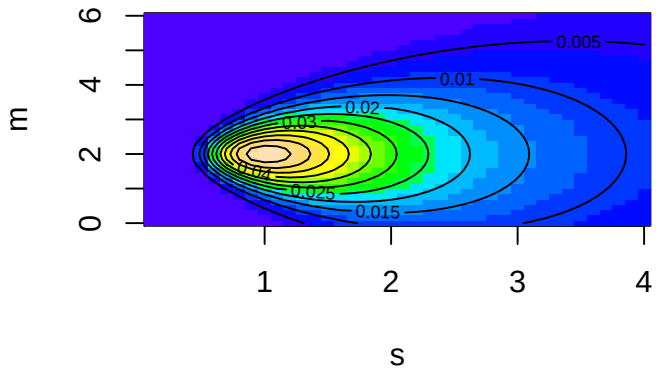
4. Find the maximum likelihood estimator of the standard deviation of two points, 1 and 3, if we assume that they were drawn from a population with mean 4:

```
> f = plotFun( dnorm(1,mean=m,sd=s)*dnorm(3,mean=m,sd=s) ~ s, s=range(.1,10), m=4)
```



Now let the mean be a free variable and maximize over both the mean and standard deviation:

```
> f = plotFun( dnorm(1,mean=m,sd=s)*dnorm(3,mean=m,sd=s) ~ s&m, s=range(.1,4), m=range(
```



ERRORS: Fill these in as I find them, remove as I fix them.

- ...