Plotting, Derivatives, and Integrals for Teaching Calculus in R

Daniel Kaplan

March 8, 2012

It's fair to ask at the beginning: Why do calculus in R? The answers to that question shape much of the design of the calculus operators in the `mosaic` package.

The `mosaic` package is one of the initiatives of Project MOSAIC, an NSF-sponsored[1] project that aims to make stronger connections among modeling, statistics, computation, and calculus in the undergraduate curriculum. R is an understandable choice for software for teaching statistics in a computationally serious way. It's also an extremely good choice for teaching statistical modeling, having a highly consistent and expressive set of modeling operators.

But for calculus, other choices are more obvious and much more widely used, particular so-called "Computer Algebra Systems" (CAS). Widely used CAS packages are Mathematica and Maple, MATLAB (with a CAS extension), and even CAS-enabled hand-held calculators and smart-phone apps. According to the MAA (reference ...), about 50% of university-level calculus courses are taught using calculators, and about 20% are taught using computers.

It's the rare statistics course that is taught using a CAS such as Mathematica. The MAA survey indicates that a slight majority of university-level statistics classes use computers. Commonly used systems include Minitab, JMP, and Excel, among others. (The high-school AP statistics curriculum does not involve computers because of the perceived impracticality of using them in an exam setting.) Based on anecdotal evidence, R is just starting to enter the mainstream for computing in introductory statistics.

One reason why CAS is infrequently used in teaching statistics is that the major symbolic operations — solving, differentiation, integration — are not at all central to the introductory statistics curriculum. Introductory statistics is generally cast as a pre-calculus subject. CAS user interfaces are not designed around data, and there is a syntactical overhead involved in providing CAS capabilities.

The curricular separation of statistics and calculus causes problems. Mathematics-oriented students often take few statistics courses and see very little statistical content in their mathematics courses. Conversely, students in introductory

[1] NSF DUE 0920350

statistics courses have not been exposed to important concepts from calculus that would help them understand statistical methods. For example, the ideas of linear and low-order polynomial approximation are important in statistical modeling, and the concept of "partial derivative" or "partial change" is crucial to understanding statistical adjustment to deal with confounding. In addition, the computational education of students is wanting ... in calculus and algebra courses they don't learn computing in a way that might be useful when working with data. In computing in statistics courses, students often see only a menu of statistics-related options. Calculus students see a set of functions largely restricted to those that can be readily integrated and differentiated using symbolic techniques, ignoring approaches such as splines and smoothers that might be more readily applicable to real-world modeling problems. Parameter estimation from data is not a standard topic in calculus.

Making a transition from the present arrangement which segregates statistics from calculus is challenging in part because of the different software used in the two disciplines. In my experience, there is a small proportion of university-level faculty who are conversant both with statistical software and CAS. This report describes an implementation of basic calculus operations in R. The point of this is to provide a means for greater integration and exchange between the calculus and statistics curricula. Such exchange might also be provided by implementing statistics operations in CAS software, or by software such as Sage that provides both R and CAS functionality.

An obvious goal of the calculus operators in the `mosaic` package is

- To support teaching calculus in a computational setting.

R has obvious advantages here, being free, open-source, and available on many platforms (including web servers). With R, and given that the vast majority of today's students, even in high school, have internet access, there is no budget excuse for not using computation in teaching calculus. (The same might be said for services such as Wolfram Alpha.)

It seems self-evident that the calculus operators should be straightforward to use and avoid unnecessary pitfalls and complications. At the same time, the notation used should reinforce the mathematical concepts.

But what does "reinforcing the mathematical concepts" mean in practice? For most students, learning differential calculus means mastering a set of algebraic transformation rules of the sort $x^2 \to 2x$ and $\sin(x) \to \cos(x)$, remembering intepretations such as "the derivative is the slope of a tangent line," and applying the techniques of differentiation to problems such as finding local extrema of $x^3 - 3x^2 + 7x + 4$ or resolving quantities such as $\lim_{x\to 0} \sin(x)/x$.

Insofar as these sorts of things are taken as the concepts of calculus, there may be little point to replacing paper-and-pencil techniques with computer techniques.

That's actually a tribute to the generations of mathematicians and teachers who developed the pedagogy of calculus. They were working in a technological environment of paper and pencil — originally, paper and quill! — of printed tables, of slow arithmetic calculation, of very slow and tedious graphing. Using

the technology of their day, they created representations of calculus ideas and settings where those ideas could be exercised.

The technology is different now. One obvious difference is the ability to carry out arithmetic very rapidly. Another is the ability to generate graphics. Still another is in notation: computer notation provides the capability of representing mathematical concepts clearly and unambiguously

The needs for calculus and the context in which it is taught are also different, in large respect. Filtering .... High school ... not taken as part of a curriculum where one is using calculus at the same time as studying it.

This line of thinking introduces another goal of the `mosaic` calculus operators:

- To support a context in which calculus can be used.

Increasingly, the use of calculus is in the development and interpretation of models. These models of often statistical in nature, meaning not just that they need to connect to data, but that decisions about the appropriate form of the model often need to be drawn from data or to balance data and first principles. And, these models will usually involve multiple variables. Consequently, it's important to provide a means for students to engage the mathematics of multiple variables.

Finally, we recognize that today's students will need to use computers for myriad technical tasks, and that the analysis and interpretation of data — statistics, if you will — is now central to science as well as commercial and governmental activity. The need to teach statistical computing underlies another goal of the `mosaic` calculus operators:

- Smooth the transition from calculus to statistics, by developing concepts and habits in calculus-related computing that will apply well in statistical computing.

In particular, the `mosaic` package is designed to support the application of calculus operations to mathematical objects that are outside the conventional bounds of calculus courses but which find use in statistics, e.g., fitted functions, smoothers, etc.

1 Functions at the Core

In introducing calculus to a lay audience, mathematician Steven Strogatz wrote:

> *The subject is gargantuan — and so are its textbooks. Many exceed 1,000 pages and work nicely as doorstops.*
>
> *But within that bulk you'll find two ideas shining through. All the rest, as Rabbi Hillel said of the Golden Rule, is just commentary. Those two ideas are the "derivative" and the "integral." Each dominates its own half of the subject, named in their honor as differential and integral calculus.* — New York Times, April 11, 2010

Although generations of students have graduated calculus courses with the ideas that a derivative is "the slope of a tangent line" and the integral is the 'area under a curve," these are merely interpretations of the application of derivatives and integrals — and limited ones at that.

More basically, a derivative is a function, as is an integral. What's more, the operation of differentiation takes a function as an input and produces a function as an output. Similarly with integration. The "slope" and "area" interpretations relate to the values of those output functions when given a specific input.

The traditional algebraic notation is problematic when it comes to reinforcing the function → function operation of differentiation and integration. There is often a confusion between a "variable" and a "function." The notation doesn't clearly identify what are the inputs and what is the output. Parameters and constants are identified idiomatically: a, b, c for parameters, x, y for variables. When it comes to functions, it's usually implicit that x is the input and y is the output.

R has a standard syntax for defining functions, for instance:

```
> f <- function(x){m*x + b}
```

This syntax is nice in many respects. It's completely explicit that a function is being created. The input variable is also explicitly identified. To use this syntax, students need to learn how computer notation for arithmetic differs from algebraic notation: `m*x + b` rather than $mx+b$. This isn't hard, although it does take some practice. Assignment and naming must also be taught. Far from being a distraction, this is an important component of doing technical computing and transfers to future work, e.g. in statistics.

The native syntax also has problems. In the example above, the parameters `m` and `b` pose a particular difficulty. Where will those values come from? This is an issue of scoping. Scoping is a difficult subject to teach and scoping rules differ among languages.

For many years I taught introductory using the native function-creation syntax, trying to finesse the matter of scoping by avoiding the use of symbolic parameters. This sent the wrong message to students: they concluded that computer notation was not as flexible as traditional notation.

In the `mosaic` calculus operators we provide a simple means to step around scoping issues while retaining the use of symbolic parameters. Here's an example using the `makeFunction()` operator from `mosaic`.

```
> f <- makeFunction(m*x + b ~ x)
```

One difference is that the input variable is identified using the R `~` syntax: the "body" of the function is on the left of `~` and the input variable to the right.

This is perhaps a slightly cleaner notation than `function`, and indeed my experience with introductory calculus students is that they make many fewer errors with the `makeFunction()` notation.

More important, though, `makeFunction()` provides a simple framework for scoping of symbolic parameters: they are all explicit arguments to the function being created. You can see this by examining the function itself:

```
> f

function (x, m, b)
m * x + b
```

When evaluating `f()`, you need to give values not just to the independent variables (`x` here), but to the parameters. This is done using the standard named-argument syntax in R:

```
> f(x=2, m=3.5, b=10)

[1] 17
```

Typically, you will assign values to the symbolic parameters at the time the function is created:

```
> f <- makeFunction(m*x + b^x, m=3.5, b=10)
```

This allows the function to be used as if the only input were `x`, while allowing the roles of the parameters to be explicit and self-documenting and enabling the parameters to be changed later on.

```
> f(x=2)

[1] 17
```

The variable `pi` is handled differently; it's always treated as the number π.

In general, functions can have more than one input. The `mosaic` package handles this using an obvous extension to the notation:

```
> g <- makeFunction(A*x*sin(x*y)^x&y, A=10)
> g

function (x, y, A = 10)
A * x * sin(x * y)
```

In evaluating functions with multiple inputs, it's helpful to use the variable names to identify which input is which:

```
> g(x=0.2,y=3)

[1] 1.129285
```

Mathematically, the `makeFunction()` notation highlights the distinction between parameters and inputs to functions. It allows the inputs to be identified explicitly, but it also does not enforce an artificial distinction between parameters and "inputs." Sometimes, you want to study what happens as you vary a parameter.

It also introduces the `~` notation early and in a fundamental way. The `mosaic` package builds on this to enhance functionality while maintaining a common

theme. In addition, the notation sets students up for a natural transition to functions of multiple variables.

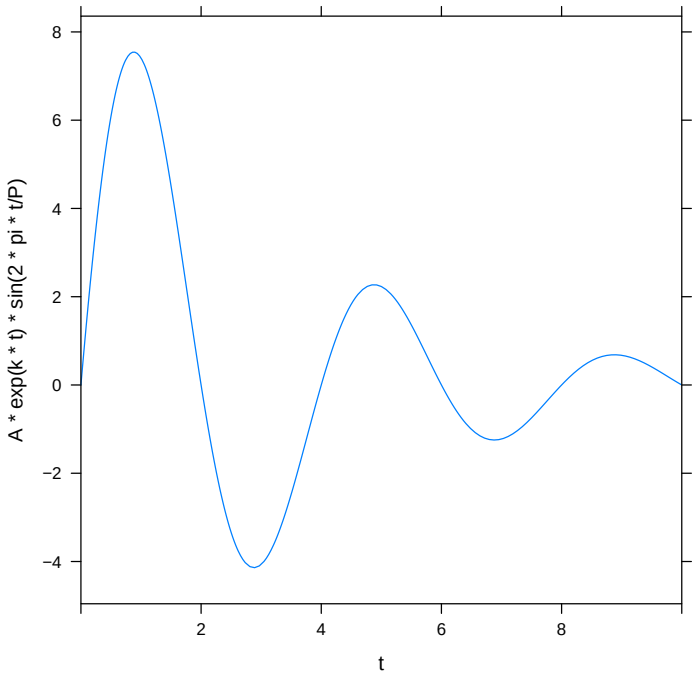
You can, of course, use functions constructed using `function()` or in any other way in the `mosaic` package operators. Indeed, `mosaic` is designed to make it straightforward to employ calculus operations to construct and interpret functions that do not have a simple algebraic expression, for instance splines, smoothers, and fitted functions. (See Section 7.)

2 Graphs

The `mosaic` package provides a basic operator for graphing functions: `plotFun()`. This one function handles three different formats of graph: the standard line graph of a function of one variable; a contour plot of a function of two variables; and a surface plot of a function of two variables.

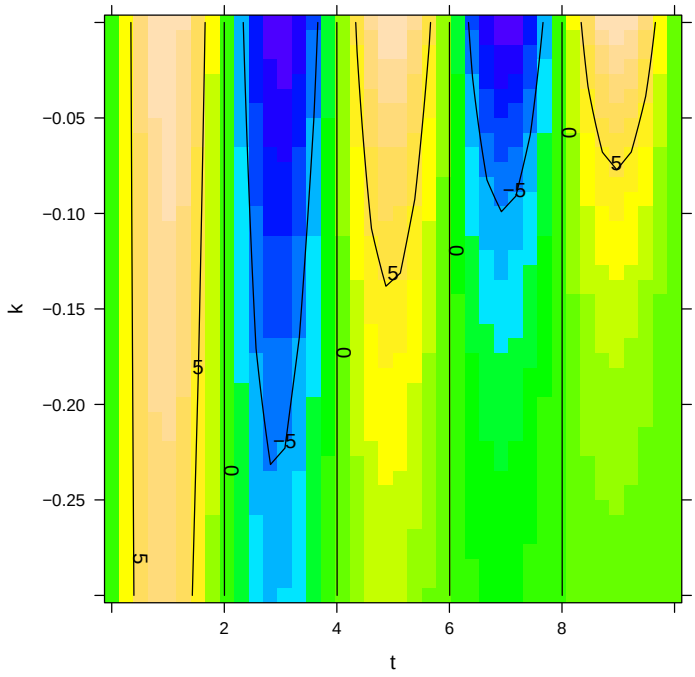
The variables to the right of `~` set the independent axes plotting variables. The plotting domain can be specified by a `lim` argument whose name is constructed to be prefaced by the variable being set. For example, here's a conventional line plot of a function of t:

```
> p=plotFun(A*exp(k*t)*sin(2*pi*t/P) ~ t,
+           t.lim=range(0,10), k=-0.3, A=10, P=4)
> print(p)
```



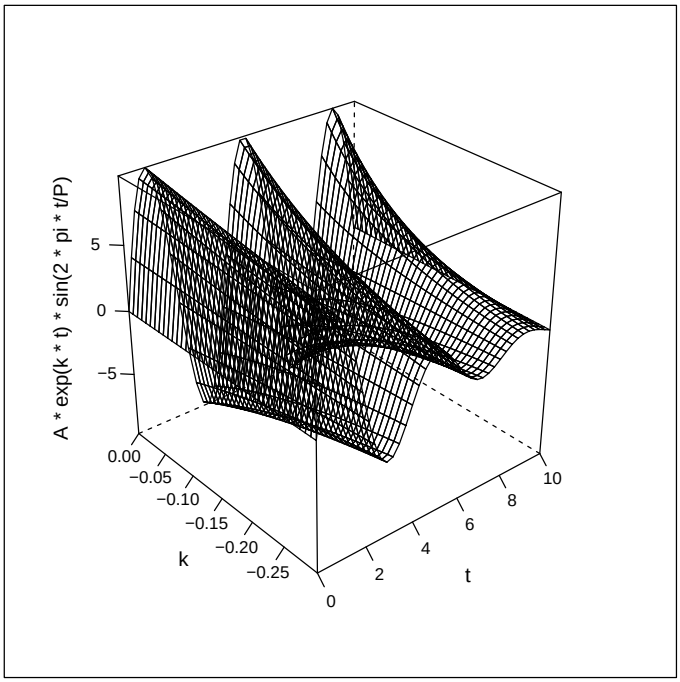
For functions of two variables, the default format is a contour plot:

```
> p=plotFun(A*exp(k*t)*sin(2*pi*t/P) ~ t&k,
+           t.lim=range(0,10), k.lim=range(-0.3,0.0), A=10, P=4)
> print(p)
```



You can override the default with `surface=TRUE`,

```
> p=plotFun(A*exp(k*t)*sin(2*pi*t/P) ~ t&k,
+           t.lim=range(0,10), k.lim=range(-0.3,0.0), A=10, P=4,
+           surface=TRUE)
> print(p)
```



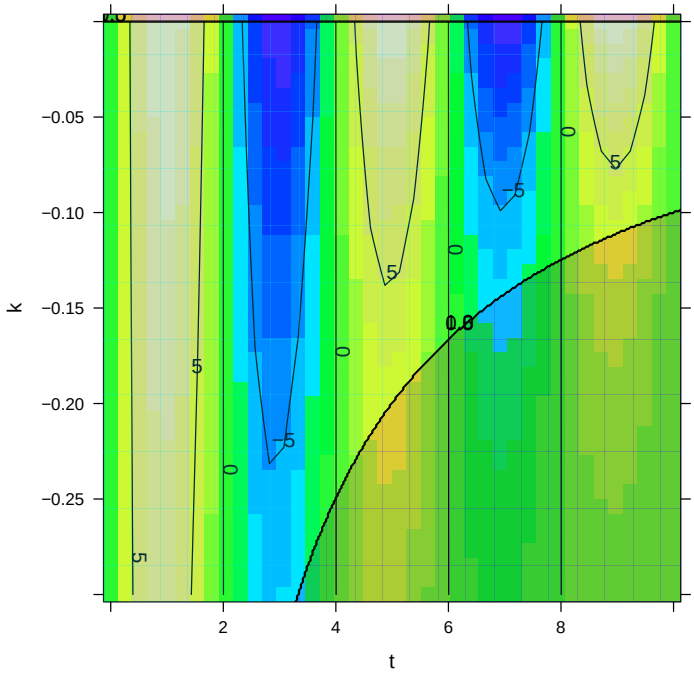
In general, surface plots are hard to interpret, but they are useful in teaching students how to interpret contour plots.

The resolution of the two-variable plots can be changed with the `npts` argument. By default, it's set to be something that's rather chunky in order to enhance the speed of drawing. A value of `npts=300` is generally satisfactory for publication purposes.

The `lattice` graphics package is used to implement `plotFun`. We hope eventually to use the `lattice` panel capabilities to provide support for displaying functions of three variables.

Common graphical tasks are comparing two functions or plotting a function along with data. The standard `lattice` approach to this can be daunting for students. To make the task of overlaying plots easier, `plotFun()` has an `add` argument to control whether to make a new plot or overlay an old one. Here's an example of laying a constraint $t + 1/k \leq 0$ over another function:

```
> plotFun(A*exp(k*t)*sin(2*pi*t/P) ~ t&k,
+           t.lim=range(0,10), k.lim=range(-0.3,0.0), A=10, P=4)
> p= plotFun( t + 1/k <= 0 ~ t&k, add=TRUE, npts=300, alpha=.2)
> print(p)

NULL
```



The lighter region shows where the constraint is satisfied. Note also that a high resolution (`npts=300`) was used for plotting the constraint. At the default resolution, such contraints are often distractingly chunky.

The `mosaic` graphics operators are built on `lattice` but provide an interface similar to that of `makeFunction()`.

```
> plotFun( dt(t,df)~t&df, t.lim=range(-3,3),df.lim=range(1,10))
```

3 Differentiation

A derivative is an operation that takes a function as input and returns a function as an output. In `mosaic`, differentiation is implemented by the `D()` operator.[2] masks the original `D`) operator from the `stats` package.[3]

A function is not the only input to differentiation; one also needs to specify the variable with respect to which the derivative is taken. Traditionally, this is represented as the variable in the denominator of the Leibniz quotient, e.g. x in $\partial/\partial x$.

To enable functions of multiple variables to be differentiated flexibly, the `mosaic D()` operator takes not a bare function name, like `sin`, but an expression

[2] `mosaic D`

[3] The `stats` operator can be accessed, if desired, by using the double-colon notation `stats::D`.

that applies the function to a variable. For instance,

```
> D(sin(x)~x)

function (x)
cos(x)
```

The use of expressions in this way makes it straightforward to move on to functions of multiple variables and functions with symbolic parameters. For example,

```
> D( A*x^2*sin(y) ~ x )

function (x, A, y)
A * (2 * x) * sin(y)

> D( A*x^2*sin(y) ~ y )

function (y, A, x)
A * x^2 * cos(y)
```

Notice that the object returned by `D()` is a function. The function takes as arguments both the variables of differentiation and any other variables or symbolic paremeters in the expression being differentiated. Default values for parameters will be retained in the return function. Even parameters or variables that are eliminated in the process of differentiation will be retained in the function. For example:

```
> D(A*x + b~y, A=10,b=5)

function (y, A = 10, x, b = 5)
0
```

The controlling rule here is that the derivative of a function should have the same arguments as the function being differentiated.

Second- and higher-order derivatives can be handled using an obvious extention to the notation:

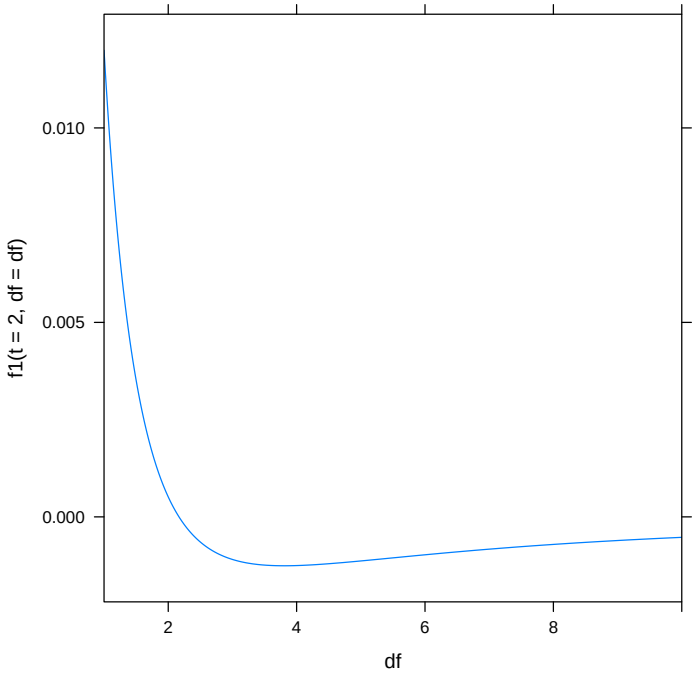
```
> D( A*x^2*sin(y) ~ x&x )

function (x, A, y)
A * 2 * sin(y)

> D( A*x^2*sin(y) ~ y&y )

function (y, A, x)
-(A * x^2 * sin(y))

> D( A*x^2*sin(y) ~ x&y ) #mixed partial
```

```
function (x, y, A)
A * (2 * x) * cos(y)
```

The ability to carry out symbolic differentiation is inherited from the `stats::deriv()` operator. This is valuable for two reasons. First, seeing R return something that matches the traditional form can be re-assuring for students and instructors. Second, derivatives — especially higher-order derivatives — can have noticeable pathologies when evaluated through non-symbolic methods such as simple finite-differences. Fortunately, `stats::deriv()` is capable of handling the large majority of sorts of expressions encountered in calculus courses.

Not every function has an algebraic form that can be differentiated using the algebraic rules of differentiation. In such cases, numerical differentiation can be used. `D()` is designed to carry out numerical differentiation and to package up the results as a function that can be used like any other function. To illustrate, consider the derivative of the density of the t-distribution. The density is implemented in R with the `dt(t, df)` function, taking two parameters, t and the "degrees of freedom" df. Here's the derivative of density with respect to df constructed using `D()`:

```
> f1 = D( dt(t,df) ~ df)
> f1(t=2,df=1)

[1] 0.01199571

> p = plotFun(f1(t=2,df=df)~df, df.lim=range(1,10))
> print(p)
```



Numerical differentiation, especially high-order differentiation, has problematic numerical properties. For this reason, only second-order numerical differentiation is directly supported. You can, of course, construct a higher-order numerical derivative by iterative application of `D()` to a derivative function, but don't expect very accurate results.

4 Anti-Differentiation

The `antiD()` operator carries out anti-differentiation. The syntax and return of a function is very similar to `D()` with two exceptions:

- Only first-order integration is directly supported.
- The returned function retains the same arguments as the function being integrated, but splits the variable of integration into a "from" and a "to" part.

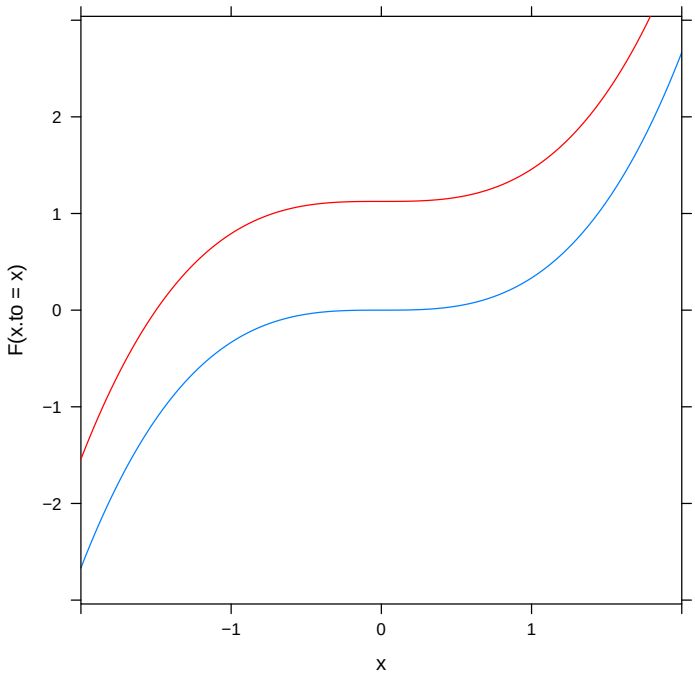
To illustrate, here is $\int ax^2 dx$ (which should give $\frac{a}{3}x^2$):

```
> F = antiD( a*x^2~x, a=1 )
> F

function (x.to = NaN, x.from = 0, a = 1)
{
```

```
    numerical.integration(.newf, .wrt, as.list(match.call())[-1],
        formals())
}
<environment: 0x104824bf8>

> F(x.to=1) #should be 1/3

[1] 0.3333333
```

Being a function, the output of `antiD()` can be plotted:

```
> plotFun(F(x.to=x)~x, x.lim=range(-2,2))
> p = plotFun(F(x.from=-1.5,x.to=x)~x, add=TRUE, col="red")
> print(p)

NULL
```



At this time, integration is purely numerical. The mechanics are hidden behind the function `numerical.integration()`, which is not intended to be used directly.

Unlike differentiation, integration has good numerical properties. Even integrals out to infinity can often be handled with great precision. Here, for instance, is a calculation of the mean of a normal distribution via integration from $-\infty$ to ∞:

```
> F = antiD( x*dnorm(x,mean=3,sd=2)~x)
> F(x.from=-Inf,x.to=Inf)

[1] 3

> F = antiD(x*dexp(x,rate=rate)~x)
> F(x.from=0,x.to=Inf,rate=10)

[1] 0.1

> F(x.from=0,x.to=Inf,rate=100)

[1] 0.01
```

Because anti-differentiation is done numerically, you can compute the anti-derivative of any function that's numerically well behaved, even when there is no simple algebraic form. In particular, you can take the anti-derivative of a function that is itself an anti-derivative. Here, for example, is a double integral for the area of a circle of radius 1:

```
> one = makeFun(1~x&y)
> by.x = antiD( one(x=x, y=y) ~x )
> by.xy = antiD(by.x(x.from=-sqrt(1-y^2), x.to=sqrt(1-y^2), y=y)~y)
> by.xy(y.from=-1, y.to=1)

[1] 3.141593
```

5 Solving

The `findZeros()` function will locate zeros of a function in a flexible way that's easy to use. The syntax is very similar to that of `plotFun()`, `D()`, and `antiD()`: You specify an expression and the values of any symbolic parameters. The search for zeros is conducted over a range that can be specified in a number of ways. To illustrate:

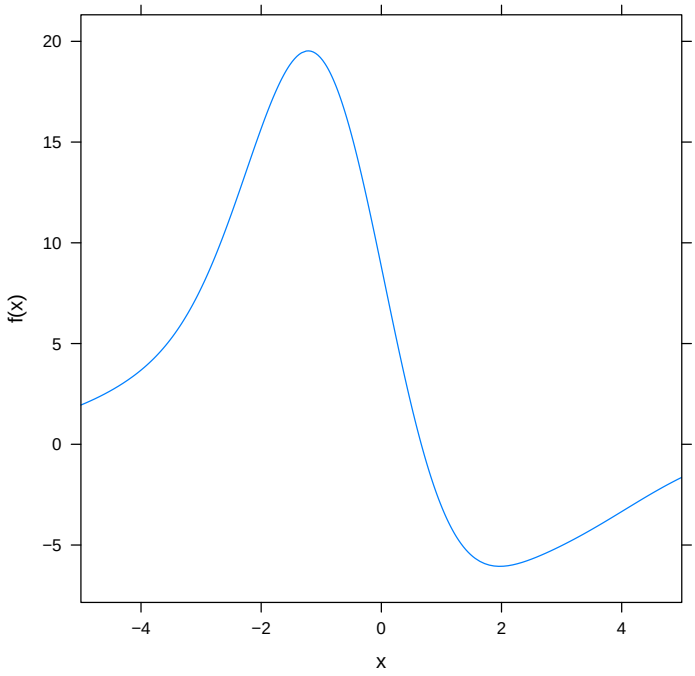
```
> findZeros( sin(t)~t, t.lim=range(-5,1))

[1] -3.141593  0.000000

> findZeros( sin(t)~t, nearest=5, near=10)

[1]  3.141592  6.283185  9.424778 12.566371 15.707964

> findZeros( sin(t)~t, near=0, within=10)

[1] -9.424778 -6.283185 -3.141593  0.000000  3.141593  6.283185  9.424778
```

We hope to extend `findZeros()` to work with multiple functions of multiple variables.

6 Random-Example Functions

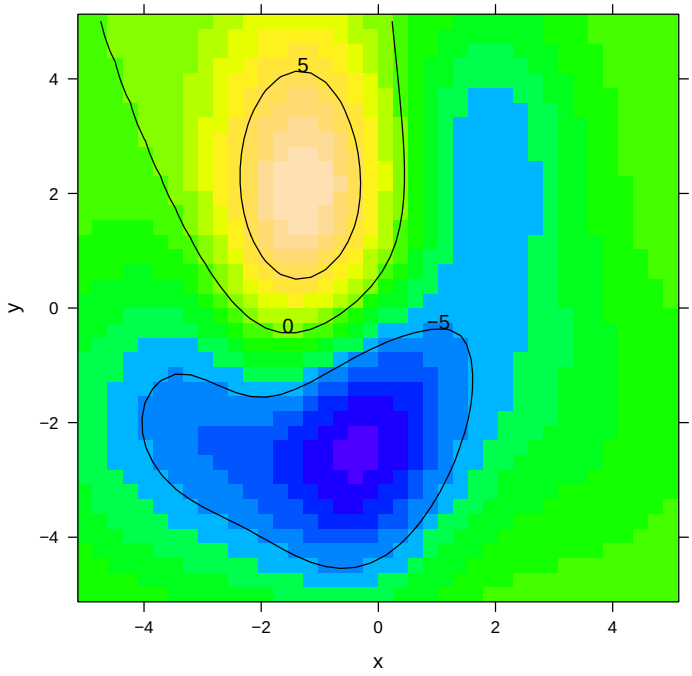
In teaching, it's helpful to have a set of functions that can be employed to illustrate various concepts. Sometimes, all you need is a smooth function that displays some ups and downs and has one or two local maxima or minima. The `rfun()` function will generate such functions "at random." That is, a random seed can be used to control which function is generated.

```
> f = rfun(~x, seed=345)
> p = plotFun(f(x)~x, x.lim=range(-5,5))
> print(p)
```



These random functions are particularly helpful to develop intuition about functions of two variables, since they are readily interpreted as a landscape:

```
> f = rfun(~x&y, seed=345)
> p = plotFun(f(x,y)~x&y, x.lim=range(-5,5), y.lim=range(-5,5))
> print(p)
```



7 Functions from Data

Aside from `rfun()`, all the examples to this point have involved functions expressed algebraically, as is traditional in calculus instruction. In practice, however, functions are often created from data. The `mosaic` package supports three different types of such functions:

1. Interpolators: functions that connect data points.
2. Smoothers: smooth functions that follow general trends in data.
3. Fitted functions: parametrically specified functions where the parameters are chosen to approximate the data in a least-squares sense.

8 Differential Equations

A basic strategy in calculus is to divide a challenging problem into easier bits, and then put together the bits to find the overall solution. Thus, areas are reduced to integrating heights. Volumes come from integrating areas.

Differential equations provide an important and compelling setting for illustrating the calculus strategy, while also providing insight into modeling approaches and a better understanding of real-world phenomena. A differential

equation relates the instantaneous "state" of a system to the instantaneous change of state. "Solving" a differential equation amounts to finding the value of the state as a function of independent variables. In an "ordinary differential equations," there is only one independent variable, typically called time. In a "partial differential equation," there are two or more dependent variables, for example, time and space.

The `integrateODE()` function solves an ordinary differential equation starting at a given initial condition of the state.

To illustrate, here is the differential equation corresponding to logistic growth:

$$\frac{dx}{dt} = rx(1 - x/K).$$

There is a state x. The equation describes how the change in state over time, dx/dt is a function of the state. The typical application of the logistic equation is to limited population growth; for $x < K$ the population grows while for $x > K$ the population decays. The state $x = K$ is a "stable equilibrium." It's an equilbrium because, when $x = K$, the change of state is nil: $dx/dt = 0$. It's stable, because a slight change in state will incur growth or decay that brings the system back to the equilibrium. The state $x = 0$ is an unstable equilibrium.

The algebraic solution to this equation is a staple of calculus books. It is

$$x(t) = \frac{Kx(0)}{x(0) + (K - x(0)e^{-rt})}.$$

The solution gives the state as a function of time, $x(t)$, whereas the differential equation gives the change in state as a function of the state itself. The initial value of the state (the "initial condition") is $x(0)$, that is, x at time zero.

The logistic equation is much beloved because of this algebraic solution. Equations that are very closely related in their phenomenology, do not have analytic solutions.

The `integrateODE()` function takes the differential equation as an input, together with the initial value of the state. Numerical values for all parameters must be specified, as they would in any case to draw a graph of the solution. In addition, must specify the range of time for which you want the function $x(t)$. For example, here's the solution for time running from 0 to 20.

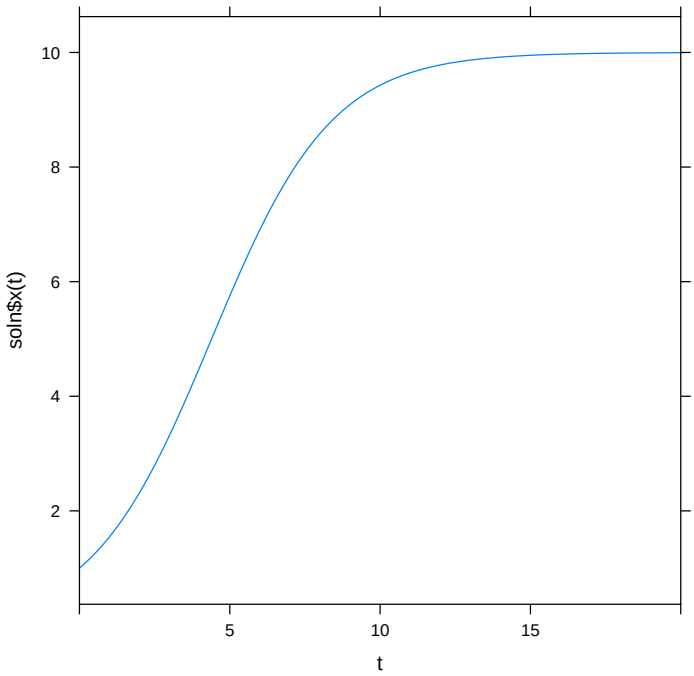
```
> soln <- integrateODE( dx ~ r*x*(1-x/K),
+                       x=1, K=10, r=.5,
+                       tdur=list(from=0,to=20))
```

The object that is created by `integrateODE()` is a function of time. Or, rather, it is a set of solutions, one for each of the state variables. In the logistic equation, there is only one state variable x. Finding the value of x at time t means evaluating the function at some value of t. Here are the values at $t = 0, 1, \ldots, 5$.

```
> soln$x(0:5)

[1] 1.000000 1.548281 2.319693 3.324279 4.508531 5.751209
```

Often, you will plot out the solution against time:

```
> p = plotFun(soln$x(t)~t, t.lim=range(0,20))
> print(p)
```

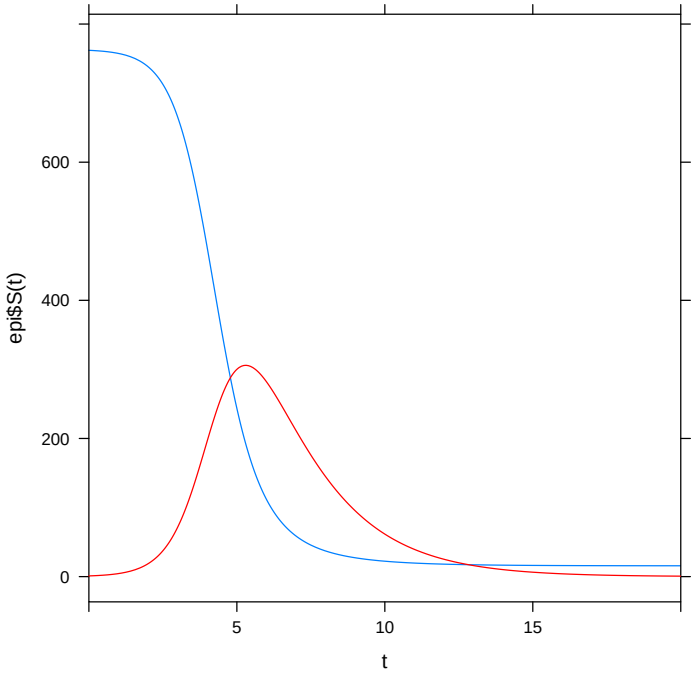


Differential equation systems with more than one state variable can be handled as well. To illustrate, here is the SIR model of the spread of epidemics, in which the state is the number of susceptibles S and the number of infectives I in the population. Susceptibles become infective by meeting an infective, infectives recover and leave the system. There is one equation for the change in S and a corresponding equation for the change in I. The initial $I = 1$, corresponding to the start of the epidemic.

```
> epi = integrateODE( dS~-a*S*I, dI~a*S*I - b*I,
+   a=0.0026,b=.5,S=762,I=1,tdur=20)
```

This system of differential equations is solved to produce two functions, $S(t)$ and $I(t)$.

```
> plotFun( epi$S(t)~t, t.lim=range(0,20))
> p = plotFun( epi$I(t)~t, add=TRUE, col="red")
> print(p)

NULL
```



In the solution, you can see the epidemic grow to a peak near $t = 5$. At this point, the number of susceptibles has fallen so sharply that the number of infectives starts to fall as well. In the end, almost every susceptible has been infected.

9 Supporting a Pedagogy of Calculus

A quantitative curriculum with data and statistics at the center. Modeling, linear algebra, interpretation.

Teaching computation. What does it mean to teach computation? Sensitivity to syntax and the structure of computer commands. The idea of assignment and sequences of steps. For statistics: summarization, iteration, randomization, accumulation. Identification of the "types" of objects: numbers, functions, vectors.

Community of users: basic knowledge. Everyone is expected to learn algebra and it is used in many fields in science. Sometimes this just amounts to $\sqrt{n}$. Similarly, we need a common language in computation, even if it's very simple operations. There's a question of what that language should be, and whether there should be any language at all. Let's not compromise on using many different languages, none of them well.

Some people will argue that the "language" should be user-friendly interfaces that can be learned intuitively. I think there are good reasons to think that this

will never happen. QUOTE OF LESLEY GROVES: and I speak as an expert on explosives. If this is right, it will become self-evident as soon as such an interface is constructed. In the meantime, we need to work with what we can get.

Calculus taught using Mathematica doesn't seem to do it. Few students learn how to define a function, let alone to iterate, randomize, accumulate, summarize. Also, Mathematica has failed to make inroads into other fields.

The great deficiency of `mosaic` calculus operators is that they don't replicate the traditional symbolic integration capabilities and symbolic solutions. (The symbolic differentiation properties are already pretty close.) But is this really a deficiency?

A lot of the difficulty of integral calculus comes from identifying the situations where an analytic integral does not exist and distinguishing them from the cases where one just hasn't figured out how to arrive at the analytic form. For many important functional forms, there is no analytic form. But there is always an integral. Doing integration numerically may be slow in certain cases, but it is numerically stable and no less liable to error (due to singularities, etc.) than the traditional symbolic algorithms as implemented by students.

The calculus features of the `mosaic` package were written to support teaching calculus to beginners, not for the professional who is using calculus in his or her technical work. Much more powerful calculus capabilities are provided by widely accessible software such as Mathematica and Wolfram Alpha. What the `mosaic` functions provide is an *interface* to calculus functionality that is intended to be straightforward, to relate well to statistical operations that are not a part of the traditional calculus curriculum, and to help students learn basics of technical computing while they are learning calculus.

For many calculus instructors, the reasons not to use R/`mosaic` will be obvious: the symbolic capabilities of `mosaic` are very limited, there are more powerful general-mathematics programs that can be used, and calculus instruction is not necessarily improved by electronic computation. This last reason perhaps explains why so many calculus courses are being taught without modern computing. If the point of teaching calculus is to support advanced calculations, then use software designed for this purpose. If the point is to exercise logical reasoning abilities, then don't use electronic computation.[4]

The crux of the issue is the purpose of teaching calculus. This is not such a simple matter. The plain answer — not necessarily a good answer — is that the topics of calculus are useful, either directly in their technical application or as a means to train the mind. But utility depends on context. Many of the topics of the traditional calculus curriculum are there to support the technology of traditional calculus. Doing integrals? Partial fractions might be useful. Trying to find a tricky limit? L'Hopital's rule can simplify things, and for that you need symbolic differentiation. Need to find an extremum? Differentiate and solve.

In science education, the curriculum gradually shifts to support new dis-

[4]Somehow, from this dilemma, graphing calculators, ubiquitous in high schools, have become the compromise of choice.

coveries. My mother's high-school education, in the 1940s, did not include any mention of DNA. That would be absurd now. Yet my daughter's calculus course was very similar to my father's. There's little internal need for change in mathematics education. Mathematics is always right. In contrast, science is always wrong. But science, even if wrong, can be useful. Mathematics, even if right, is not always so.

The big outside changes that need to influence mathematics are the emergence, both largely in the second half of the 20th century, of electronic computation and of statistics. Both computation and statistics are so widely used, in so many aspects of civic, commercial, and technical work, that the re-alignment of mathematics education to serve those needs ought to be both compelling and obvious. Yet it has not happened very broadly.

Calculus played a major role in the emergence of electronic computation. The name of the first general-purpose electronic computer, ENIAC, announced in 1946, stood for "Electronic Numerical Integrator and Computer." Why feature integration in the name? Because integration is both important and difficult.

Imagine a negotiation where the science/mathematics curriculum were being devised *de novo*. What arguments would proponents of calculus be able to bring to the table to support allocating student time and energy to calculus? Point to the value of l'Hopital's rule in science? Hardly. Argue that trigonometric substitution is the foundation of physics? No. Instead, they might reasonably claim that calculus provides an important way to describe the world, that calculus concepts (differentiation, partial derivatives, accumulation, ...) are important in analyzing systems and in understanding approximation, that calculus helps in modeling. There are important ways in which calculus supports statistics. These have to do with modeling, approximation, and optimization and not so much with "area under a curve."