Modeling-Based Calculus with R/mosaic

Daniel Kaplan
Dept. of Mathematics, Statistics, and Computer Science
Macalester College
1600 Grand Ave.
St. Paul, MN 55105
`kaplan@macalester.edu`

Cecylia Bocovich
Cheriton School of Computer Science
University of Waterloo
200 University Ave W
Waterloo, ON N2L 3G1
Canada
`cbocovic@uwaterloo.ca`

Randall Pruim
Dept. of Mathematics and Statistics
Calvin College
Grand Rapids, MI
49546
`rpruim@calvin.edu`

Introduction: Software for Calculus

The choice of software for teaching calculus depends on several objectives and constraints:

- availability;
- instructor experience;
- graphics, symbolic, and numerical capabilities;
- steepness of the learning curve;
- etc.

When calculus is taught from a modeling perspective, other objectives ought to be considered:

- support for a broad range of modeling techniques, and
- the potential for use of the software in other, non-calculus, modeling courses.

R/mosaic for Calculus

We describe a powerful, expressive, and free computing environment that few calculus instructors have explored: R with the *mosaic* package [Pruim 2011]

Although R is most closely associated with statistics and data analysis, R is designed to be extensibl, and it contains capabilities for symbolic manipulation, differentiation, and integration.

The *mosaic* package provides commands to make it easier to teach and to learn introductory calculus, statistics, and modeling. The principle behind *mosaic* is that a notation can support learning more effectively when it

- draws clear connections between related concepts,
- is concise and consistent, and
- avoids distracting verbiage.

At the same time, R/mosaic facilitates bringing data into the modeling process, thus broadening the repertoire of functions that can be used in modeling, adding, for instance, fitted functions, smoothers, interpolators, and discontinuous functions.

Although *mosaic* also includes extensive facilities for teaching and doing statistics, here we describe just the calculus-related features and their use in modeling.

Calculus, as developed historically and in the main as taught today, has little or nothing to do with statistics. Calculus software is generally associated with computer algebra systems (CAS) such as Mathematica, which provide the ability to carry out the operations of differentiation, integration, and solving algebraic expressions. Traditionally, the calculus curriculum has emphasized symbolic algorithms and rules (such as $x^n \to nx^{n-1}$ and $\sin x \to \cos x$). Computer algebra systems provide a way to automate such symbolic algorithms and extend them beyond human capabilities, but they are not designed to support the process of modeling.

The *mosaic* package provides computer functions implementing the core operations of calculus—differentiation and integration—as well plotting, modeling, fitting, interpolating, smoothing, solving, etc. The notation is designed to emphasize the roles of different kinds of mathematical objects—variables, functions, parameters, data—without unnecessarily turning one into another. For example, the derivative of a function in *mosaic*, as in mathematics, is itself a function. The result of fitting a functional form to data is similarly a function, not a set of numerical coefficients.

Goals of mosaic for Calculus

The *mosaic* calculus features were developed to support a calculus course with these goals:

- introduce the operations and applications of differentiation and integration (which is what calculus is about);
- provide students with the skills needed to construct and interpret useful models that can apply *inter alia* to biology, chemistry, physics, and economics;
- familiarize students with the basics of functions of multiple variables;
- give students computational skills that apply outside of calculus; and
- prepare students for the statistical interpretation of data and models relating to data.

These goals are very closely related to the objectives stated by the Mathematical Association of American in its series of reports on Curriculum Reform and the First Two Years [Mathematical Association of America n.d.]. As such, even though they may differ from the goals of a typical calculus class, they are likely a good set of goals to aspire to in most settings.

We begin by introducing the variety of ways to describe mathematical functions in *mosaic*. In addition to the usual algebraic formula functions there are functions based on data—fitted functions, interpolators, and smoothers—and generic, smooth, wavy functions. Following giving the *mosaic* notation for graphing functions, we introduce the *mosaic* differentiation and anti-differentiation functions, which provide both symbolic and numerical capabilities. We then show how to use *mosaic* to solve differential equations.

We address some questions that arise when constructing functions from data, and we comment on the links between calculus and algebra when seen from a modeling perspective.

Functions and Notation

Functions, the transformation of an input to an output, are the building blocks of modeling. A common and accepted notation for calculus students is exemplified by $f(x) = mx + b$. In standard R syntax, `f()` can be defined as

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

We refer to the function as $f()$, where the empty parentheses are merely a typographical reminder that `f` is a function and not some other sort of object such as a number or a table of data.

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, these necessities are an important component of doing technical computing and transfers to future work, e.g., in statistics. It's also helpful to remind students that functions don't have to be named f, g, and h, but can have more descriptive names, such as `O2_uptake`.

Variables and Parameters

In using a function like `f()`, a very real problem is the meaning of m and b. The convention, which students somehow pick up, is that letters such as a, b, c, and m are fixed parameters, while x, y, and z are variables. This convention dates back at least to Newton [??? 1718]. But how to assign numerical values to parameters like m and b?

Mosaic gives a straightforward notation to distinguish between variables and parameters, while providing a sensible way to assign specific values to parameters. With *mosaic*, the function `f()` can be defined using `makeFun()`:

```
library(mosaic)
f <- makeFun(m * x + b ~ x)
```

The command, `library(mosaic)` loads the *mosaic* package, making the *mosaic* functions available to the user. (To be able to loaded, a package must first be installed on the user's computer. Installation can be accomplished with the command `install.packages("mosaic")`. Installation need only be done once, but loading must be done for each session.)

In computing terms, `f()` is an ordinary R object. You can examine any object by using its name as a command:

```
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 as well. This is done using the standard named-argument syntax in R:

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

```
[1] 17
```

Typically, you assign values to the symbolic parameters when the function is created:

```
f <- makeFun(m*x + b~x, m=3.5, b=10)
```

Doint so 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 values of the parameters to be changed later.

```
f(x=2)
```

```
[1] 17
```

A function can have more than one input variable. To identify symbolic quantities as input variables, list them to the right of the formula tilde:

```
g <- makeFun(A * x * cos(pi * x * y) ~ x + y, A=3)
g
```

```
function (x, y, A = 3)
A * x * cos(pi * x * y)
```

Other symbolic quantities in the expression to the left of the tilde are treated as "parameters." The only operational difference between an input variable and a parameter involves warning the user whenever a parameter with is not given a default value . For example, the following two functions are equivalent except for the warning generated in making the second one:

```
g1 <- makeFun(A * x * cos(pi * x * y) ~ x + y + A)
g2 <- makeFun(A * x * cos(pi * x * y) ~ x + y)
```

It is entirely a matter of style whether to include a symbolic quantity such as A as a "parameter" or as an "input variable." The warning message encourages a style where parameters are assigned default values. (The symbol pi is handled specially; it's always treated as the number π and never treated as a parameter.)

In constructing a function, the order of the input variables and parameters is more or less arbitrary:

- Quantities identified as input variables, by being on the left side of the tilde, appear first on its right-hand side, listed in the same order.

- Parameters are listed after the variables, in no particular order.

When you "manufacture" a new function—for instance, by differentiation—the order of variables in the new function may be different from that in the parent function.

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

```
g(x=1,y=2)
```

```
[1] 3
```

Using `makeFun()` to define functions has the following advantages for introductory calculus students:

- The notation highlights the distinction between parameters and inputs to functions.
- Inputs are distinguished explicitly, reflecting the distinction between parameters and inputs; but parameter values too can be changed.
- R's formula syntax is introduced 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.

Functions from Data

Traditionally, calculus instruction has emphasized functions that can be described by a single algebraic formula. In modeling, however, functions often stem from data. *Mosaic* provides a variety of ways to construct functions from data: interpolators, smoothers, and linear and nonlinear fitted functions.

Interpolating Functions

Interpolating functions connect data points. Different dnterpolating functions have differing properties of smoothness, monotonicity, endpoints, etc., which can be important in modeling. *Mosaic* implements several interpolation methods for functions of one variable.

To illustrate, we present some data from a classroom example intended to illustrate the measurement of flow using derivatives. Water was poured out of a bottle into a cup set on a scale. Every three seconds, a student read off the digital reading from the scale, in grams. Thus, the data indicate the mass of the water in the cup.

```
Water <- data.frame(
  mass=c(57,76,105,147,181,207,227,231,231,231),
  time=c(0,  3,  6,  9, 12, 15, 18, 21, 24, 27))
```

Plotting the data can be done with *mosaic*, which employs R's `lattice` graphics (an implementation of trellis graphics) [Sarkar n.d.].

Of course, the mass in the cup varied continuously with time. It's just the recorded data that are discrete. The *mosaic* `spliner()` function creates a cubic-spline interpolant that connects the measured data:

```
waterMass <- spliner(mass ~ time, data = Water)
```

Following the notation for functions in *mosaic*, the formula `mass ~ time` indicates that `time` is the input and `mass` is the output. When `time` matches one of the cases in the `Water` data, the output is the corresponding value of `mass` from that data.

```
waterMass(time=c(0,3,6))
```

```
[1]  57  76 105
```

At intermediate values of `time`, the function takes on interpolated values:

```
waterMass(time=c(0,.5,1,1.5,2,2.5))
```

```
[1] 57.00000 59.71408 62.59067 65.64367 68.88697 72.33445
```

Like any other smooth function, `waterMass()` can be plotted (**Figure 1**), differentiated, integrated, etc.

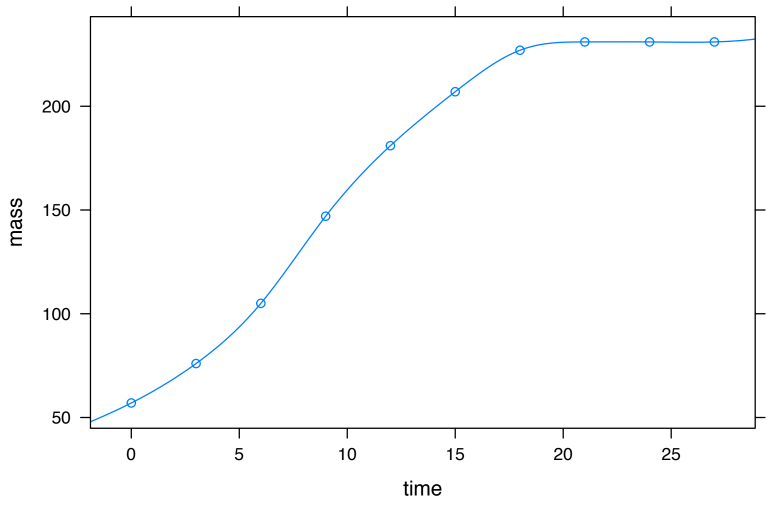


Figure 1. The water-pouring data and a smooth interpolator.

Mosaic provides other interpolating functions. For situations that demand monotonicity (remember, the water was being poured into the cup, not spilling or draining out), monotonic, smooth splines can be created

```
monoWaterFun <- spliner(mass ~ time, data = Water,
                                              monotonic = TRUE)
```

If smoothness isn't important, the straight-line connector might be an appropriate interpolant:

```
lineWaterFun <- connector(mass ~ time, data = Water)
```

Smoothers

Interpolating functions "connect the dots." In contrast, a smoother is a function that follows general trends of data. Unlike an interpolating function, a smoother need not replicate the data exactly. To illustrate, consider a moderate-sized data set `CPS85` that gives hourly wage and demographic data for 534 people in 1985. (`CPS85` is in the `mosaicData` package available through CRAN, the Comprehensive R Archive Network [2015].)

```
library(mosaicData)
data(CPS85)
```

There is no definite relationship between wage and age, but there are general trends, as indicated in **Figure 2**.

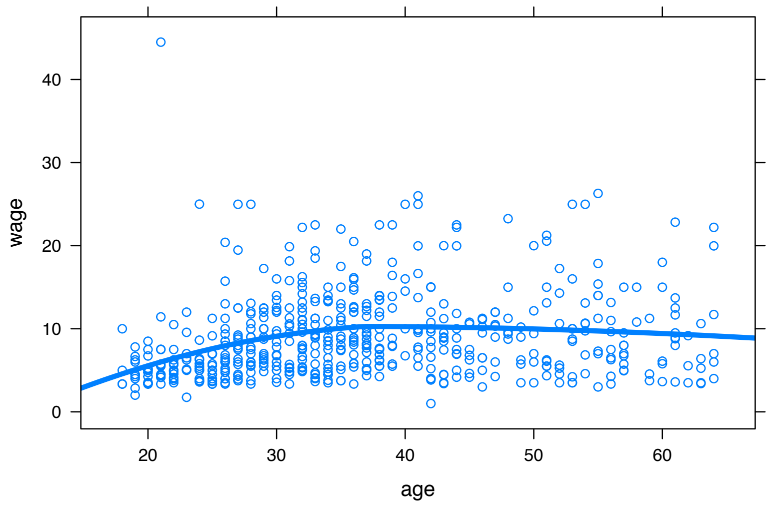


Figure 2. The `CPS85` data for wage and age plotted along with a smoother function.

Smoothers can construct functions of more than one variable. Here, for instance, is a functional representation of the relationship between wage, age, and education. The function is graphed in **Figure 3**.

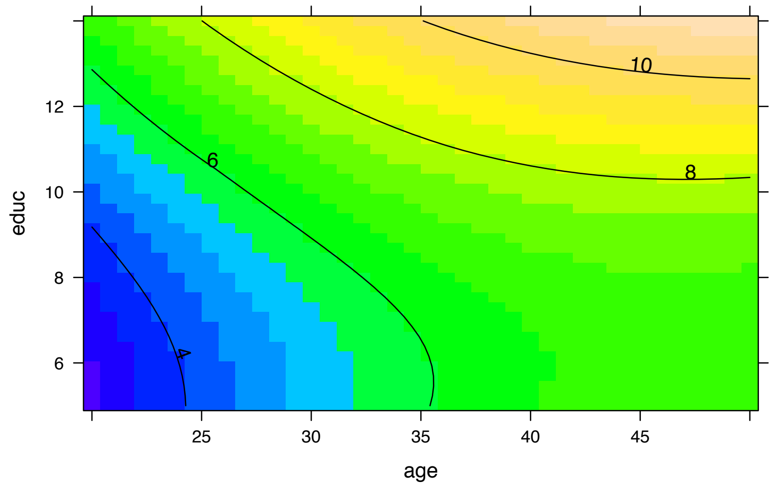


Figure 3. A smoother modeling the relationship between wage and age and education.

```
wageModel <- smoother(wage ~ age + educ, span = 1.0,
                                         data = CPS85)
```

Fitted Functions

R has powerful facilities for fitting parametric functions to data. Two of the most wide used parametric functions result from linear and logistic fitting. These are carried out with the standard R `lm()` and `glm()` functions.

The *mosaic* function `makeFun()` takes the output of `lm()` (linear models), `glm()` (generalized linear models), or `nls()` (nonlinear least squares), and packages it up into a model function. For instance, we have:

```
wageModelLinear <- lm(log(wage) ~ age + educ + age:educ + 1,
                                         data = CPS85)
wageFunLinear <- makeFun(wageModelLinear)
wageFunLinear(age = 55, educ = 12)
```

```
       1
2.187433
```

In addition to applying `makeFun()` to models produced using the standard statistical modeling functions `lm()`, `glm()`, and `nls()`, *mosaic* also provides `fitModel()` and `fitSpline()`. These fit nonlinear and spline models and return the model as a function.

- The splines used by `fitSpline()` are not interpolating splines but splines chosen by the method of least squares. In typical applications, one sets the number of *knots* (points where the piecewise polynomial function is “stiched together”).

- The command `fitModel()` is roughly equivalent to first fitting the model with `nls()` and then applying `makeFun()`; but the command provides an easier syntax for setting starting points for the least-squares search algorithm.

Among other things, `fitModel` enables students to refine eyeballed fits to data. For example, here's an exponential model fitted to the `CoolingWater` data. From **Figure 4**, it's easy to see that the exponential "half-life" is approximately 30 minutes and the asymptotic temperature is about 30°C, so these can be used as starting guesses.

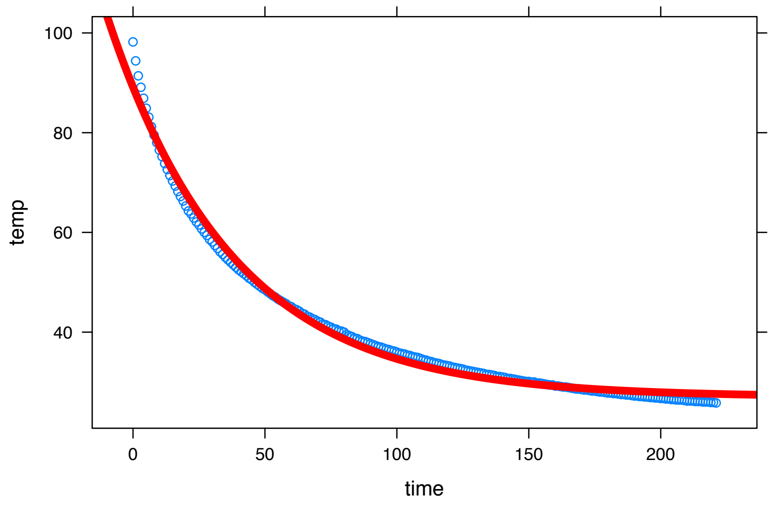


Figure 4. An exponential model fitted to the `CoolingWater` data with `fitModel()`.

```
waterCoolingMod <- fitModel(temp ~ A + B * exp( - k * time),
                            start = list(A = 30, k = log(2)/30),
                            data = CoolingWater)
```

The function returned by `fitModel()` shows the specific numerical values of the parameters:

```
waterCoolingMod
```

```
function (time, ..., transform = identity)
return(transform(predict(model, newdata = data.frame(time = time),
    ...)))
<environment: 0x100d537c0>
attr(,"coefficients")
          A           k           B
27.00447664  0.02096445 62.13547142
attr(,"class")
[1] "nlsfunction" "function"
```

As a rule, starting guesses for linear parameters don't need to be provided. For complicated models, the fitting process may fail to converge unless the starting guess is good. Commonly encountered examples of nonlinear parameters include the exponential decay constant k in the above, and the period for a sine wave.

Sketchy Functions

In teaching, it's helpful to have a set of functions that can be employed to illustrate various concepts. Sometimes, all that's needed 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." **Figure 5** shows some examples. An optional random seed provides a reproducible function that can be selected by an instructor.

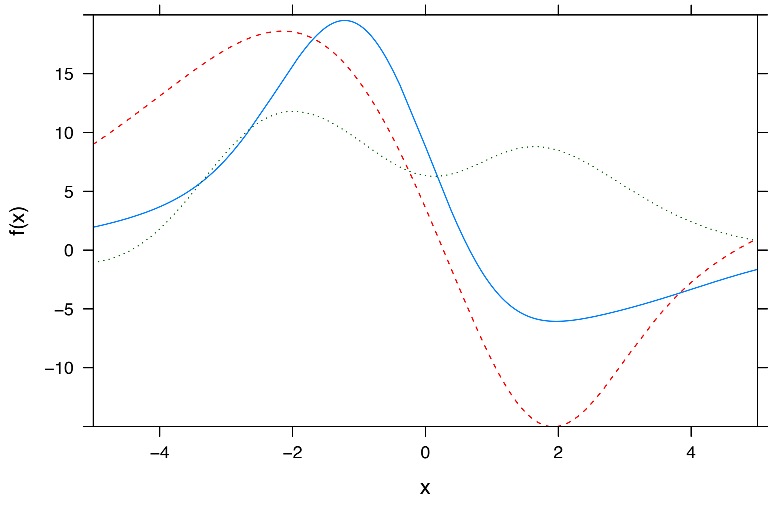


Figure 5. Three randomly generated smooth functions with local maxima and minima. `g()` and `h()` are shown with dashed and dotted curves, respectively.

```
f <- rfun( ~ x, seed = 345)
g <- rfun( ~ x)
h <- rfun( ~ x)
```

These random functions are particularly helpful to develop intuition about functions of two variables, since they are readily interpreted as a landscape. For instance, **Figure 6** shows a round hill adjacent to a cresent-shaped lake.

```
f <- rfun( ~ x + y, seed=345)
```

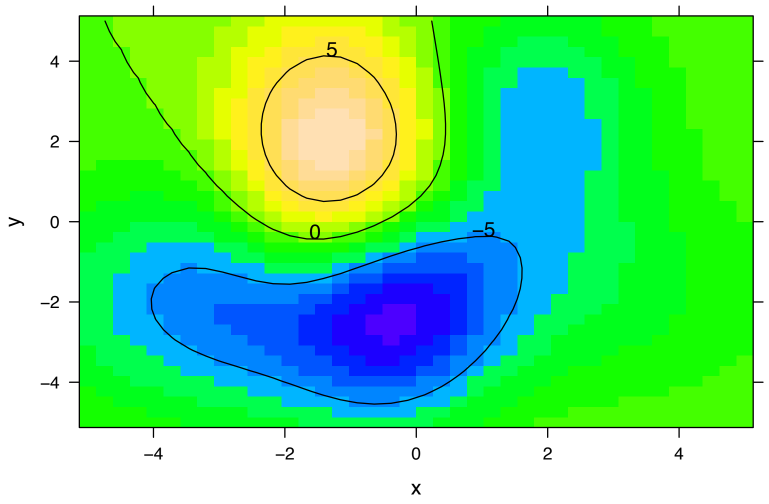


Figure 6. A sketchy function of two variables.

Graphics

Functions generated by *mosaic* can be evaluated and plotted using any of the R graphics systems; but to simplify graphing functions of one or two variables, *mosaic* provides `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, or
- a surface plot of a function of two variables.

The `plotFun()` interface is similar to that of `makeFun()`. 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, in **Figure 7** is a conventional line plot of a function of t and in **Figure 8** is a contour plot of two variables.

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


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)
```

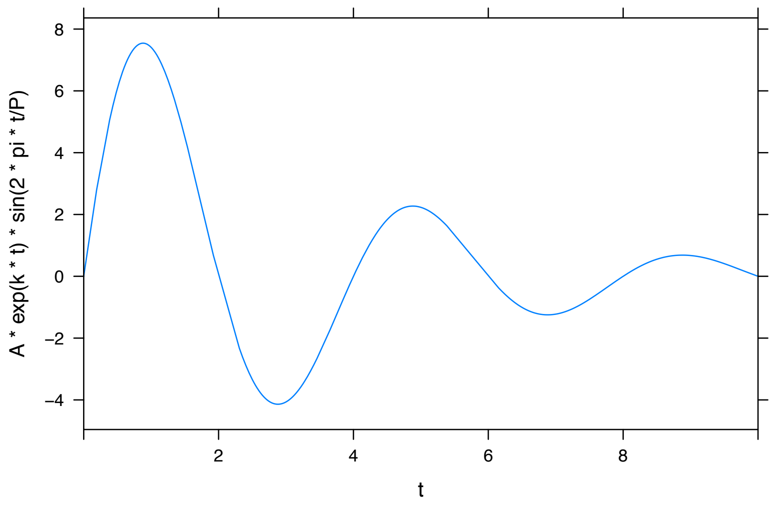


Figure 7. Conventional line plot.

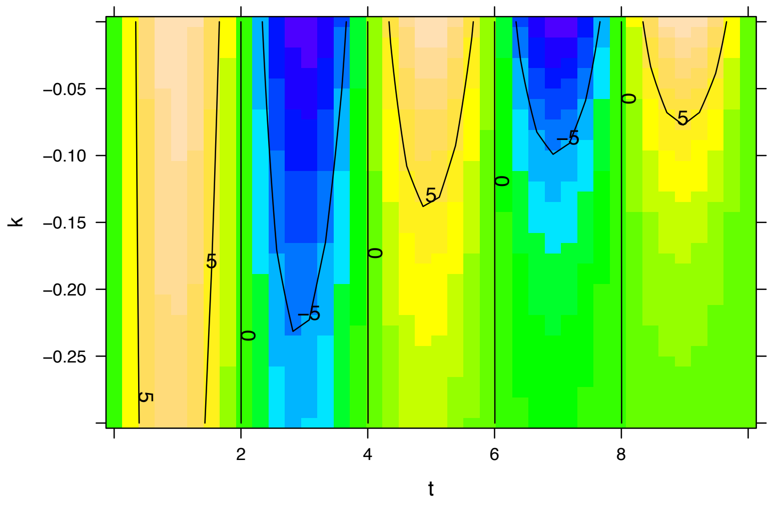


Figure 8. Contour plot of two variables.

Functions of two variables can be plotted as pseudo-3D surface plots by specifying the option `surface = TRUE`. Typically, surface plots are hard to interpret, but they are useful in teaching students how to interpret contour plots.

To simplify overlaying plots, `plotFun()` has an `add` argument to control whether to make a new plot or overlay an old one. Similarly, there is an `plotPoints()` function that works like the standard lattice scatterplotter, `xyplot()`, but knows about `add`.

For instance, **Figure 9**, which shows three sketchy functions, is made like this:

```
plotFun(f(x)~x, x.lim=range(-5,5), ylim=c(-15,20))
plotFun(g(x) ~ x, add=TRUE, col='red', lty=2)
plotFun(h(x) ~ x, add=TRUE, col='darkgreen', lty=3)
```

The `plotPoints()` function is useful for comparing data to functions. For example, **Figure 9** shows the model function `wageModel()` for wage as a function of age and education along with the age and education data from `CPS85`.

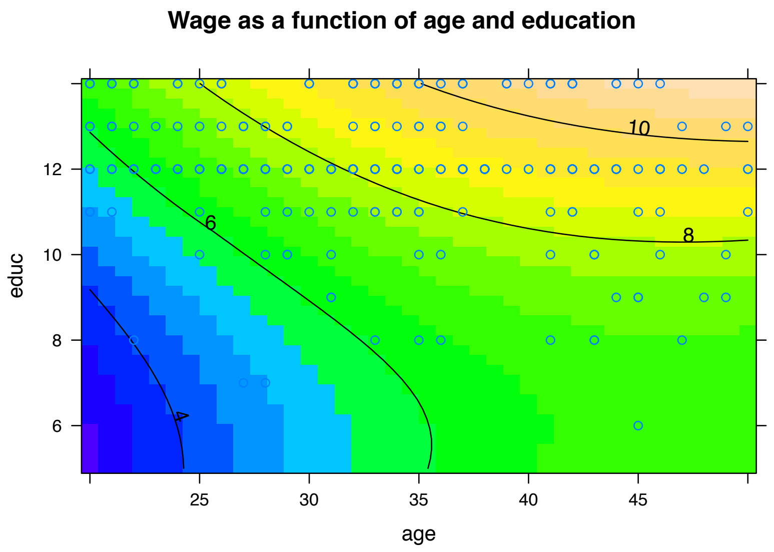


Figure 9. The `educ` vs. `age` data added to the plot of `wageModel()`.

```
plotFun( wageModel(age=age,educ=educ)~age+educ,
         age.lim=range(20,50),
         educ.lim=range(5,14),
         main='Wage as a function of age and education')
plotPoints(educ ~ age, data=CPS85, add=TRUE)
```

Differentiation and Antidifferentiation

Differentiation and antidifferentiation are provided by two *mosaic* functions: `D()` and `antiD()`. Both of these are set up in the same way as `makeFun()`: They take a ~ expression as input and produce a function as output. Here is an example:

```
antiD(1 / (a * x + b * y) ~ x)
```

```
function (x, C = 0, a, b, y)
1/(a) * log(((a * x + b * y))) + C
```

Default values of parameters are retained in the functions created by `D()` and `antiD()`, as we demonstrate:

```
antiD(sqrt(b + a * x) ~ x, a = 7, b=20)

function (x, a = 7, b = 20, C = 0)
1/(a) * 2/3 * sqrt(b + a * x)^3 + C
```

Such algebraic forms are a staple of introductory calculus courses, but they are not the only sort of functions used in modeling; and we want to apply the operations of differentiation and antidifferentiation to those, too. For example, recall the `waterMass()` function constructed by a spline interpolation through discrete data shown in **Figure 1**; `waterMass()` gives the mass of water accumulated in a glass over time as water is poured into it. Suppose you want to know the flow of water. This will be the derivative of `waterMass()` with respect to time:

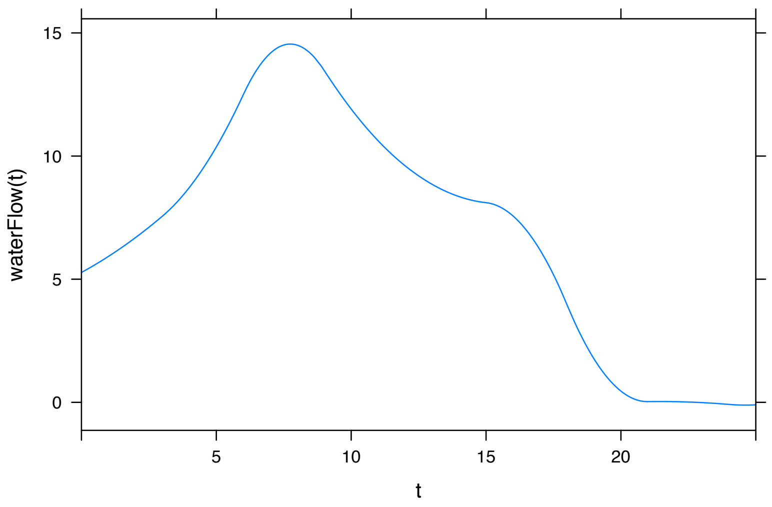


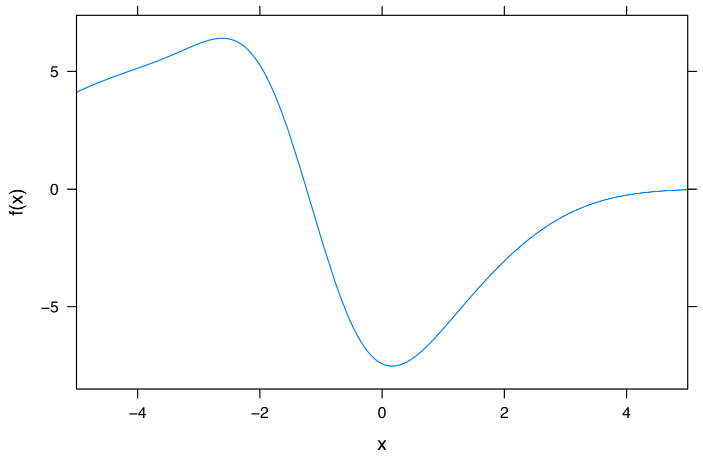
Figure 10. The rate of flow of water into a glass calculated as the derivative of the spline interpolating function `waterMass()`.

```
waterFlow <- D( waterMass(t) ~ t)
plotFun( waterFlow(t) ~ t, t.lim=range(0,25))
```

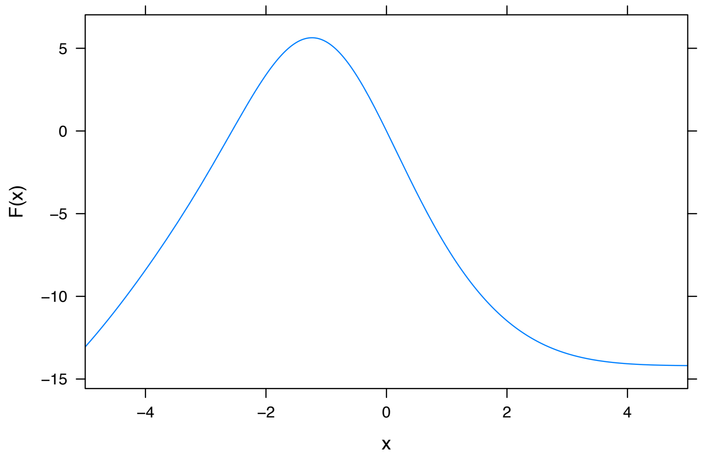
Being able to differentiate and antidifferentiate functions that do not necessarily have an "approachable" algebraic expression can be useful for demonstrating the properties of differentiation and antidifferentiation. For instance, the derivative of an antiderivative is the original function, as students can see for themselves by working with a "sketchy" function such as the following, with steps exhibited in **Figure 11**:

```
f <- rfun( ~ x, seed=98)
F <- antiD(f(x) ~ x)
g <- D(F(x) ~ x)
```

```
plotFun(f(x)~x, x.lim=c(-5,5))
```



```
plotFun(F(x)~x, x.lim=c(-5,5))
```



```
plotFun(g(x)~x, x.lim=c(-5,5))
```

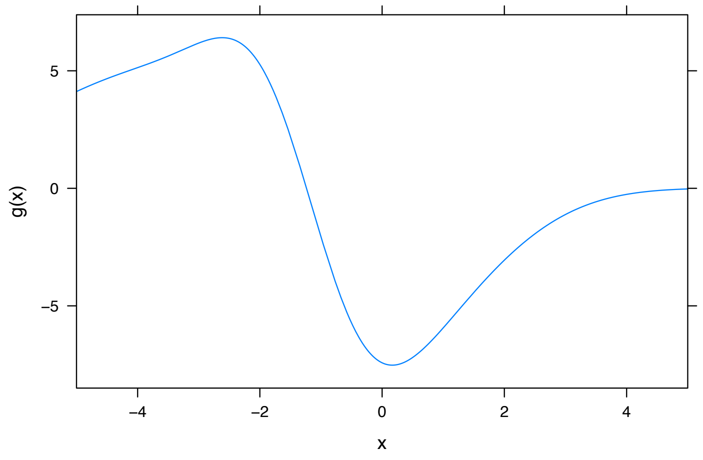


Figure 11. Operating on a sketchy function to demonstrate that the derivative of an antiderivative is the original function.

Of course, the opposite is not always true: The anti-derivative of a derivative is not necessarily the same as the original function, since they can differ by a constant (**Figure 12**).

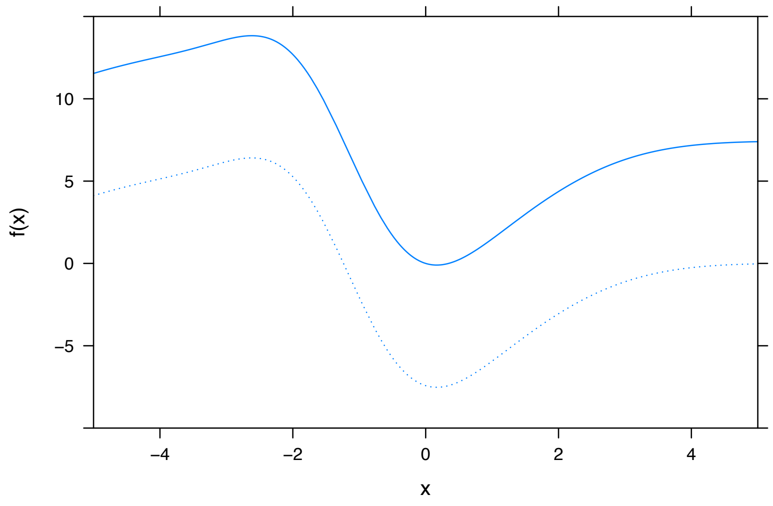


Figure 12. The antiderivative of the derivative of $f(x)$ is not necessarily the same as $f(x)$.

```
df <- D(f(x) ~ x)
DF <- antiD(df(x) ~ x)
plotFun( f(x) ~ x,
         x.lim=range(-5, 5), ylim=c(-10, 15),
         lty = 3)
plotFun( DF(x) ~ x, add=TRUE)
```

A nice activity for novice modelers is estimating the capacity of traffic on a road regulated by a traffic light. Students generally have a good idea about the length of a car, the distance between cars stopped at a light, and the delay between one car starting to move and the one behind it moving. They claim that they don't know how far a car will travel as a function of time. But they will nod their heads when reminded that a typical car accelerates from zero to 60 mph in 10 seconds. This suggests the following model of car velocity as the car accelerates to the speed limit (`pmin()` means "the lesser of two values"), with the functions shown in **Figure 13**.

```
car_velocity <- makeFun( pmin(speed_limit, (60/10)*t) ~ t,
                                          speed_limit = 40)
```

Distance travelled is the antiderivative of speed:

```
car_distance <- antiD(car_velocity(t) ~ t)
plotFun(car_distance(t) ~ t, t.lim=c(0,10))
```

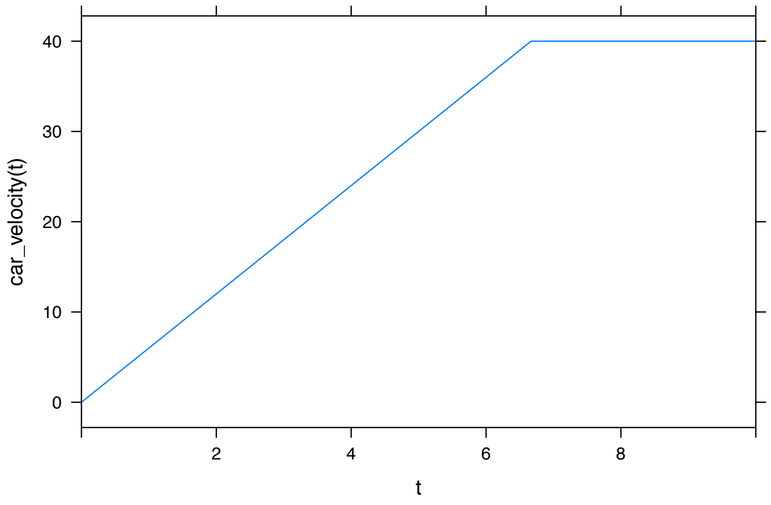


Figure 13a. the `car_velocity()` function.

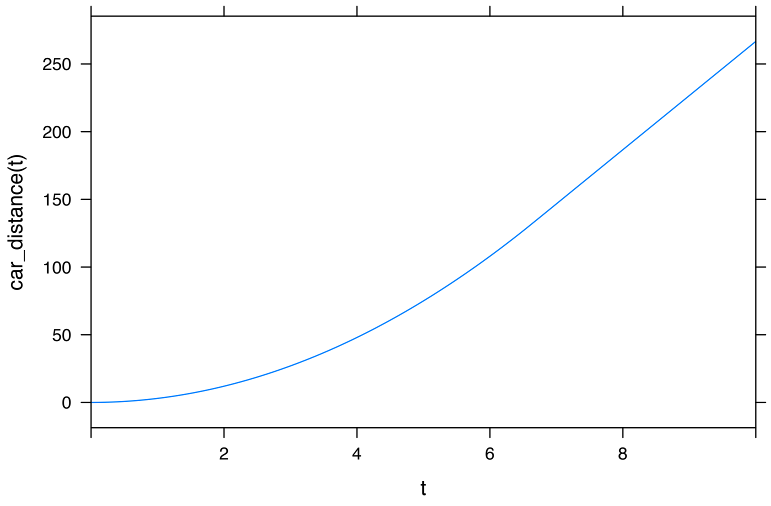


Figure 13b. The anti-derivative of the `car_velocity()` function.

citation here to Banks?

Traditional introductory calculus deals with functions of a single variable. In modeling-based calculus, it's appropriate to start students off with functions of two variables. After all, models of genuine phenomena are generally multivariate. The "sketchy" functions and the data-based functions in *mosaic* provide a means for students to start working with two-variable functions without any algebraic overhead. Partial derivatives provide an important way of interpreting such functions, answering as they do the common question: How would the output change if one input were changed with all the others held constant? (Working with partial change invites another question: What happens if changing one input variable causes a change in another input variable? The distinction between partial change and total change is an important concept in interpreting models (see Kaplan [2011, Chapter 10]).

To illustrate how *mosaic* facilitates working with such concepts, consider the example of prices of used cars. (This is a good object lesson in open-ended modeling: What determines the price of a car?) Students can easily collect data from used-car services on the Web. One such dataset is available at `http://tiny.cc/mosaic/used-hondas.csv`. This example will work with those data. `used-hondas.csv` comprises 92 cars. Here's a short excerpt:

Price	Year	Mileage	Location	Color	Age
7977	2002	100979	St.Paul	White	5
22988	2006	15075	Durham	Black	1
17900	2006	29927	Durham	Brown	1

Mosaic provides a `read.file()` function that can handle data in a variety of formats, including the widely used CSV (comma-separated values).

```
hondas <- read.file("http://tiny.cc/mosaic/used-hondas.csv")
```

Here's a simply constructed model of price as a function of *both* mileage and age (see **Figure 14**).

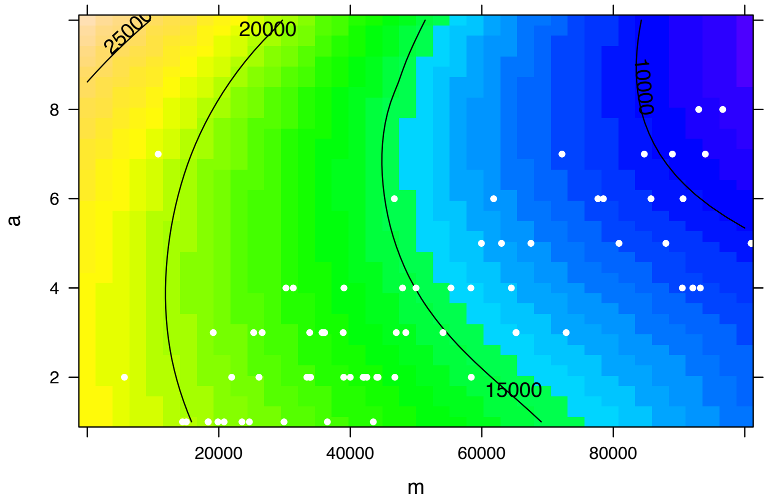


Figure 14. Price of used cars as a function of both mileage and age.

```
mod1 <- smoother(Price ~ Mileage + Age, data = hondas,
                                                span = 1)
plotFun(mod1(Mileage = m, Age = a) ~ m + a,
        m.lim = range(0, 100000), a.lim = range(1, 10))
plotPoints( Age ~ Mileage, data = hondas, add = TRUE,
            col = "white", pch = 20)
```

As expected, price goes down with age and mileage. (Note the paradoxical increase in price for low-mileage cars as they age. A good object lesson: It's dangerous to extrapolate a function far from the data used to construct it!)

How do prices change for cars as mileage increases, holding age constant? See **Figure 15**.

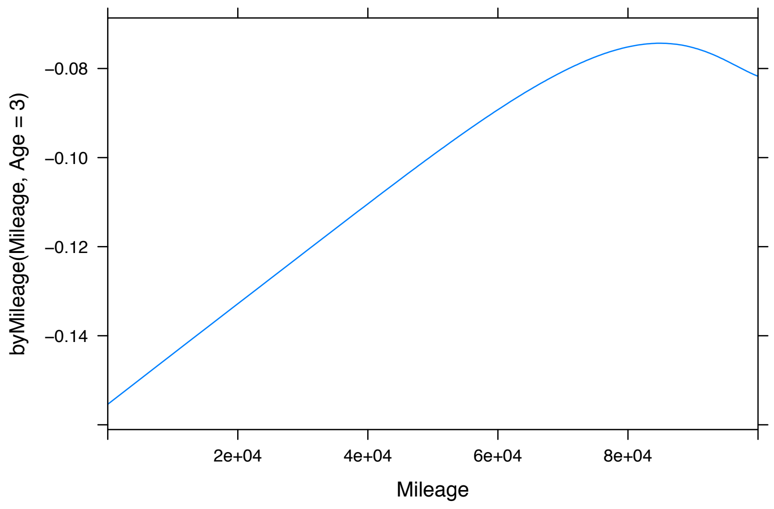


Figure 15. Loss of value per mile as a function of total mileage, for cars 3 years old.

```
byMileage <- D(mod1(Mileage = Mileage, Age = Age) ~ Mileage)
plotFun(byMileage(Mileage, Age = 3) ~ Mileage,
        Mileage.lim=range( 0, 100000))
```

Low mileage cars lose value at about 14 cents/mile, but high-mileage cars lose value at about 8 cents/mile.

A similar analysis can be performed for change in price as a function of age.

Even on a function such as a smoother, the mixed partial derivatives can be computed. How does the effect of age on price change as mileage increase?

```
mixed1 <- D(mod1(Mileage=Mileage, Age=Age) ~ Age + Mileage)
```

Or inversely, how does the effect of mileage on price change as a car ages?

```
mixed2 <- D(mod1(Mileage=Mileage, Age=Age) ~ Mileage + Age)
```

Of course, these two different-sounding questions have the same answer:

```
mixed1( Mileage = 50000, Age = 5)
```

```
          1
-0.01236913
```

```
mixed2( Mileage = 50000, Age = 5)
```

```
          1
-0.01236913
```

Differential Equations

A general problem-solving strategy is "divide and conquer," or, as George Polya [1945] phrased it, "decomposing and recombining." Calculus provides important methods for implementing divide and conquer. For instance, complex areas are divided into simple rectangles, whose areas are easily calculated and recombined. Volumes are decomposed into areas and then recombined.

For the modeler, divide and conquer often takes the form of an differential equation. The model consists of an instantaneous "state" and a formulation of how, at any state, the state will change over a very small interval of time. The differential equation decomposes the system into usually simple instantaneous relationships of this form. The simple systems are easily solved. The recombining phase sequences the simple, short-duration solutions into an overall, often complicated, long-duration solution.

For example, consider a diver jumping from a diving board. The overall motion is complicated: first up, then slowing near the peak, then falling, then hitting the water, which both slows the diver's downward speed and provides bouyancy to lift the diver to the surface. The "state" of the diver in a simple model is the vertical position y and the vertical velocity v. A differential equation model specifies how the instantaneous change of state depends on the state itself, that is, how dy/dt and dv/dt each depend on the state y and v.

An important definition for any modeler to know: Velocity is the change in state of position. One way to write this is $dy/dt = v$. A shorthand for this in R notation is

```
dy ~ v.
```

The instantaneous change in velocity is more complicated. A standard, simple model is $dy/dt = -g$, where g is the acceleration due to gravity on the Earth's surface (where most diving takes place!): $g \approx (9.8 \text{ m/s}^2$. This model will be close to reality in the air; but once the diver hits the water, there are two new important terms that come into play: water resistance to the diver's motion, and the bouyancy of the diver. Model bouyancy in the water as a constant upward force compensating gravity: $B - g \gtrsim 0$. Water resistance is often modeled as proportional to velocity squared in a direction opposite to velocity. The instantaneous change of velocity can be modeled as

```
dv ~ ifelse(y > 0,
            -g, # above the water
            B - g - r * sign(v) * v^2 # in the water
            )
```

To integrate this pair of differential equations is to recombine the relatively simple `dy` and `dv` into overall functions of time. The *mosaic* function `integrateODE()` carries this out:

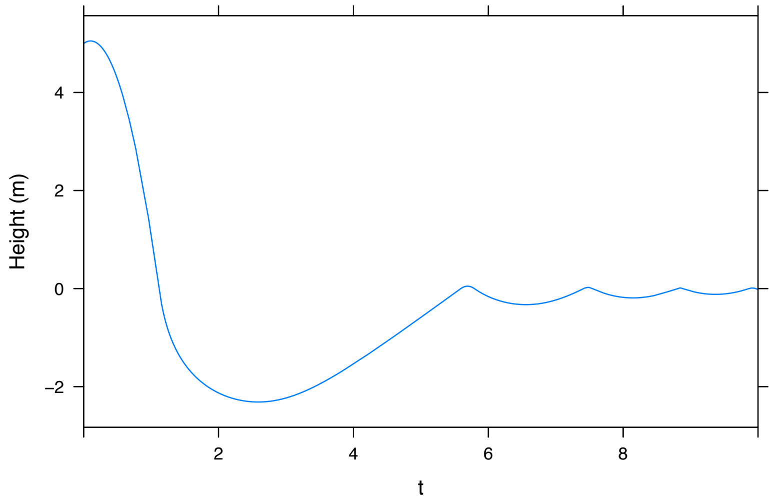


Figure 16. A solution to the diving problem.

```
diveFloat <-
  integrateODE(
               dy ~ v,
               dv ~ ifelse(y > 0,
                           -g,
               B - g - r * sign(v) * v^2),
               v = 1, y = 5,  # initial conditions
               tdur = 10, # duration of solution
               g = 9.8, B = 10.8, r = 1 # parameters
               )
plotFun(diveFloat$y(t) ~ t,
        t.lim = range(0, 10),
        ylab = "Height (m)" )
```

As seen in **Figure 16**, the diver resurfaces after slightly more than 5 seconds, and then bobs in the water. One can adjust the parameters for buoyancy B and drag r to try to match observed trajectories.

Calculus and Data

The *mosaic* package was developed to support a particular goal: uniting modeling, statistics, computation, and calculus into a coherent and useful whole.

In some ways, "mainstream calculus" and data-analysis/statistics are conflicting paradigms. Mainstream calculus emphasizes exact answers and (often) proof. Statistics is about variation and modeling. Calculus is deductive, statistics is inductive.

Some of the conflicts can be seen in approaches to constructing approximations to functions. Mainstream calculus introduces Taylor series: a way of constructing a polynomial approximation to an algebraically specified function that matches first, second, third, and higher derivatives at a particular point. Data analysts, however, know that estimating a second or third derivative from data is perilous and unreliable. They also know that high-order polynomials have very poor modeling properties; for instance, extrapolation of constructed polynomials is dangerous. Taylor polynomials match functions exactly at one point, data analysts look for good approximations over a region.

Yet there are important links between calculus and statistics. A distinctive feature of *mosaic* is the connection between data and calculus. *Mosaic* provides a tight interface to use calculus as a tool to work with smoothers, interpolators, and fitted parametric functions. Once constructed, the shapes of derived-from-data functions can be interpreted with calculus techniques such as partial differentiation or integration over a region.

Calculus tools on their own, however, cannot address a critical question: How to evaluate the strength of evidence that data provide to support a claim of any specific feature of a function derived from data? The techniques of statistics are important here and often need little specialized knowledge.

Consider, for example, the model of wage as a function of age and education introduced earlier based on a smoother (see **Figure 3** on p. 31).

```
wageModel <- smoother(wage ~ age + educ,
                      span = 1.0,
                      data = CPS85)
```

One question a data analyst might want to address from these data and the model is how wage changes with age. Typically, a person's education is set at the time he or she starts work, so a relevant quantity is the partial derivative of wage with respect to age (holding education constant). For instance, in **Figure 17** we show the result for workers with 12 years of education.

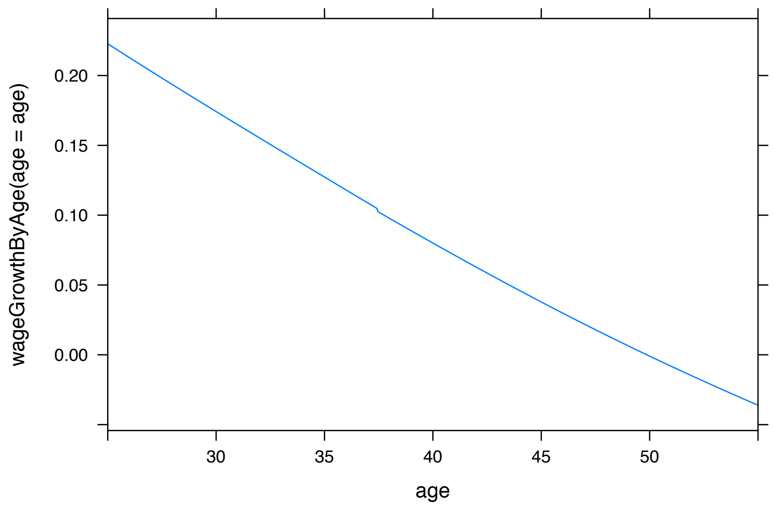


Figure 17. Wage growth by age for workers with 12 years of education.

```
wageGrowthByAge <- D(wageModel(age = age, educ = educ) ~ age,
                                                educ = 12)
plotFun(wageGrowthByAge(age = age) ~ age,
        age.lim = range(25,55))
```

The growth of wage appears to be positive, but the growth declines with age and even becomes negative after roughly 45 years of age. At age 55, the hourly wage is declining 3.6 cents per year.

```
wageGrowthByAge(age = 55)
```

```
          1
-0.03609313
```

How strong is the evidence for this wage decline? One way to answer this question is to simulate the variation in data that can be expected to be encountered if the study had been done with a new set of data. This can be done using a technique called "resampling" where new random samples of cases are drawn from the original data. The following R/mosaic statements fit the wage model to the resampled data, construct the partial derivative of wage with respect to age at age 55, and overlay it on the original plot. This is repeated 100 times to get an idea of the variation that might be anticipated if new data were collected.

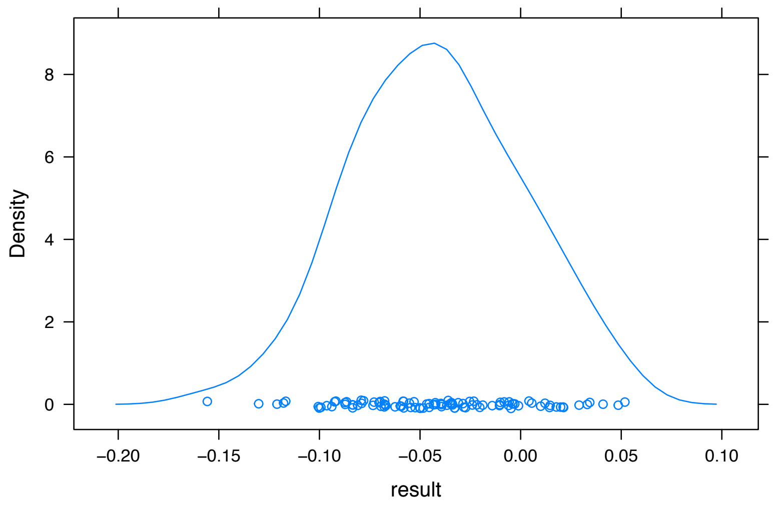


Figure 18. Dot plot and density plot for 100 resamples of partial derivative of wage with respect to age at age 55.

```
trials <-
  do(100) *
    {
      wageModelResamp <- smoother(wage ~ age + educ,
                                  span = 1.0,
                                  data = resample(CPS85))
      wageGrowthResamp <- D(wageModelResamp(age = age,
                            educ = educ) ~ age, educ = 12)
      data.frame(result = wageGrowthResamp(age = 55))
    }
densityplot( ~ result, data = trials)
```

Algebra, Calculus, and Modeling-Based Calculus

We've tried to make the case that calculus is useful as a strategy for framing models (decomposition and recombination) as well as for interpreting models. We call this "modeling-based calculus." Yet introductory calculus is typically about performing algebraic operations and it's widely thought that students need a solid grounding in algebra and trigonometry before tackling calculus. Indeed, almost all of the secondary-school mathematics curriculum is devoted to preparing for calculus. The acronym often used to describe the secondary-school mathematics curriculum is GATC: Geometry, Algebra, Trigonometry, and Calculus. Until just a half-century ago, calculus was an advanced topic first encountered in the university. Trigonometry was a practical subject, useful for navigation and surveying and design. Geometry also related to design and construction; it served as well as an introduction to proof. Calculus was a filter, helping to sort out which students were deemed suited for continuing studies in science and engineering and even medicine.

Nowadays, calculus is widely taught in high-school rather than university. Trigonometry, having lost its clientelle of surveyers and navigators, has become an algebraic prelude to calculus. Indeed, the goal of GAT has become C — it's all a preparation for doing calculus.

There is a broad dissatisfaction. Instructors fret that students are not prepared for the calculus they teach. Students fail calculus at a high rate. Huge resources of time and student effort are invested in college algebra, "remedial courses intended to prepare students for a calculus course that the vast majority—more than 90% --- will never take."

quote? source?

> "Students do not see the connections between mathematics and their chosen disciplines; instead, they leave mathematics courses with a set of skills that they are unable to apply in non-routine settings and whose importance to their future careers is not appreciated. Indeed, the mathematics many students are taught often is not the most relevant to their chosen fields.
>
> [Mathematical Association of America n.d., p. 1]

need specific MAA document

Seen in this context, college algebra is a filter that keeps students away from career paths for which they are otherwise suited.

Accept, for the sake of argument, that calculus is important, or at least is potentially important if students are brought to relate calculus concepts to inform their understanding of the world.

Is algebra helpful for most students who study it? It's not so much the direct applications. The nursing students who are examined in completing the square will never use it or any form of factoring in their careers. Underwood Dudley wrote, "I keep looking for the uses of algebra in jobs, but I keep being disappointed. To be more accurate, I used to keep looking until I became convinced that there were essentially none" [Dudley 2010, 610].

There was a time when algebra was essential to calculus, when performing calculus relied on algebraic manipulation. The use of the past tense may surprise many readers. The way calculus is taught, algebra is still essential to teaching calculus. Most people who study calculus think of the operations in algebraic terms. For the 25 years or more, however, there have been easily accessible numerical approaches to calculus problems.

The numerical approaches are rarely emphasized in introductory calculus. There are both good and bad reasons for this lack of emphasis on numerics. Tradition and aesthetics both play a role. The preference for exact solutions of algebra rather than the approximations of numerics is understandable. Possibly also important is the lack of a computational skill set for students and instructors; very few instructors and almost no high-school students learn about technical computing in a way that would make it easier for them to do numerical calculus rather than algebraic calculus. (Here's a test for instructors: In some computer language that you know, how do you write a computer function that will return a computer function that provides even a rough and ready approximation to the derivative of an arbitrary mathematical function?)

There are virtues to teaching calculus using numerics rather than algebra. Approximation is important and should be a focus of courses such as calculus. As John Tukey said, "Far better an approximate answer to the right question, which is often vague, than an exact answer to the wrong question, which can always be made precise." And computational skill is important. Indeed, it can be one of the most useful outcomes of a calculus course.

In terms of the basic calculus operations themselves, the need to compute derivatives and integrals using algebra-based algorithms limits the sorts of functions that can be employed in calculus. Students and textbooks have to stay on a narrow track which allows the operations to be successfully performed. That's why there are so many calculus optimization problems that amount to differentiating a global cubic and solving a quadratic.(When was the last time you used a global cubic to represent something in the real world?) There's little room for realism, innovation, and creativity in modeling. Indeed, so much energy and time is needed for algebra, that its conventional for functions of multiple variables to be deferred to a third-semester of calculus, a level reached by only a small fraction of students who will use data intensively in their careers.

With time, it's likely that more symbolic capabilities will be picked up in the *mosaic* package. (The `Ryacas package` [Goedman et al. 2014] already provides an R interface to a computer algebra system.) This will speed up some computations, add precision, and be gratifying to those used to algebraic expressions. But it will not fundamentally change the pedagogical issues and the desirability of applying the operations of calculus to functions that often may not be susceptible to symbolic calculation.

Acknowledgments

The *mosaic* package is one of the initiatives of Project MOSAIC 92015], an NSF-sponsored project (NSF DUE 0920350) that aims to make stronger connections among modeling, statistics, computation, and calculus in the undergraduate curriculum.

Editor's Note

Kaplan [2013] offers a further introduction to doing calculus in R, featuring greater detail and more examples.

References

Banks, Robert B. 1998, *Towing Icebergs, Falling Dominoes, and Other Adventures in Applied Mathematics*. Princeton, NJ: Princeton University Press.

Comprehensive R Archive Network (CRAN). 2015. `http://cran.r-project.org/`.

Dudley, Underwood. 2010. What is mathematics for? *Notices of the American Mathematical Society* 57 (5) (May 2010): 608–613.

Goedman, Rob, Gabor Grothendieck, Søren Højsgaard, and Ayal Pinkus. 2014. `Ryacas`—An R interface to the `yacas` computer algebra system. `http://cran.r-project.org/web/packages/Ryacas/vignettes/Ryacas.pdf`.

Kaplan, Daniel T. 2011. *Statistical Modeling: A Fresh Approach*. 2nd ed. Project Mosaic.

________. 2013. *Start R in Calculus*. Project Mosaic.

Mathematical Association of America. n.d. Curriculum Renewal Across the First Two Years (CRAFTY). `http://www.maa.org/programs/faculty-and-departments/curriculum-department-guidelines-recommendations/crafty`.

need details

Newton, Isaac. 1718. ???.

Polya, George. 1945. *How to Solve It: A New Aspect of Mathematical Method*. Princeton, NJ: Princeton University Press. 1957. 2nd ed. New York: Doubleday. 2004. Reprinted with foreword by John Conway. Princeton, NJ: Princeton University Press. 2009. Reprinted with foreword by Sam Sloan. San Rafael, CA: Ishi Press.

Project Mosaic: Modeling, Statistics, Calculus, and Computation. `http://mosaic-web.org/`.

Pruim, Randall. 2011. A `mosaic` sampler. `http://www.calvin.edu/~rpruim/talks/MosaicLightning/useR-2011/LightningMosaic.pdf`.

Sarkar, Deepayan. n.d. Getting started with lattice graphics. `http://lattice.r-forge.r-project.org/Vignettes/src/lattice-intro/lattice-intro.pdf`.

About the Authors

Danny Kaplan specializes in applying mathematics, including statistics and computation, to problems in the sciences. His professional training is in biomedical engineering, and he came to Macalester from the physiology department at McGill Medical School. He has written textbooks in nonlinear dynamics (chaos theory) applied to biology and medicine, in scientific computation, and most recently in statistical modeling. He won Macalester's 2006 Excellence in Teaching Award for creating an introductory quantitative curriculum consisting of calculus and statistics courses that are among the most heavily subscribed at the college. Prof. Kaplan also participates in Macalester's public health concentration, teaching an introductory course in epidemiology. He has a B.A. (Physics) from Swarthmore College, an M.A. (Engineering-Economic Systems) from Stanford, and a Ph.D. (Biomedical Physics) from Harvard.

Cecylia Bocovich is Ph.D. student in Computer Science at the University of Waterloo in Ontario, Canada, where she is a member of the Cryptography, Security, and Privacy (CrySP) lab and the Centre for Applied Cryptographic Research. She received a Master's of Mathematics from the University of Waterloo and a Bachelor of Arts in Mathematics and Computer Science from Macalester College.

Randall Pruim…