Plotting, Derivatives, and Integrals for Teaching Calculus in R

Daniel Kaplan

March 25, 2012

The `mosaic` package provides a command notation in R designed to make it easier to teach and to learn introductory calculus, statistics, and modeling. The principle behind `mosaic` is that a notation can more effectively support learning when it draws clear connections between related concepts, when it is concise and consistent, and when it suppresses extraneous form. At the same time, the notation needs to mesh clearly with R, facilitating students' moving on from the basics to more advanced or individualized work with R.

This document describes the calculus-related features of `mosaic`. As they have developed historically, and for the main as they are taught today, calculus instruction 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.

At the core of the `mosaic` calculus features are a set of operators that work with mathematical functions of one or more variables and implement 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 numbers.

Traditionally, the calculus curriculum has emphasized symbolic algorithms and rules (such as $x^n \rightarrow nx^{n-1}$ and $\sin(x) \rightarrow \cos(x)$). Computer algebra systems provide a way to automate such symbolic algorithms and extend them beyond human capabilities. The `mosaic` package provides only limited symbolic capabilities, so it will seem to many instructors that there is no mathematical reason to consider using `mosaic` for teaching calculus. For such instructors, non-mathematical reasons may not be very compelling for instance that R is widely available, that students can carry their R skills from calculus to statistics, that R is clearly superior to the most widely encountered systems used in practice, viz. graphing calculators. Indeed, instructors will often claim that it's good for students to learn *multiple* software systems — a claim that they don't enforce

on themselves nearly so often as on their students.

Section **??** outlines an argument that computer algebra systems are in fact a mis-step in teaching introductory calculus. Whether that argument applies to any given situation depends on the *purpose* for teaching calculus. Of course, purpose can differ from setting to setting.

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, economics,

- familiarize students with the basics of functions of multiple variables,

- give students computational skills that apply outside of calculuus,

- 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.[**?**] 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.

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 `makeFun()` operator from `mosaic`.

```
f <- makeFun(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 `makeFun()` notation.

More important, though, `makeFun()` 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
```

3

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 <- makeFun(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 <- makeFun(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] 1.129
```

Mathematically, the `makeFun()` 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 **??**.)

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 alongside 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)
```
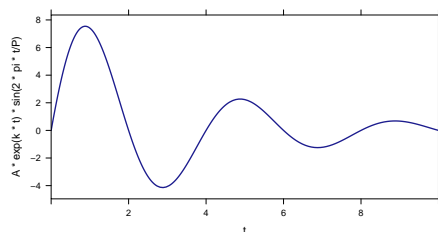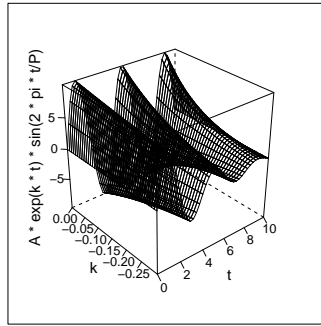


```
plotFun(A * exp(k * t) * sin(2 * pi * t/P) ~ t + k, t.lim =
range(0,
    10), k.lim = range(-0.3, 0), A = 10, P = 4)
```
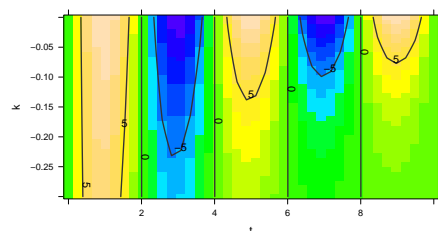

For functions of two variables, you can override the default with `surface=TRUE`,

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


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:
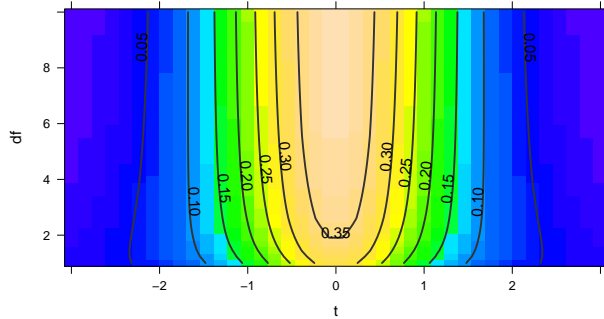
```
plotFun(A * exp(k * t) * sin(2 * pi * t/P) ~ t + k, t.lim =
range(0,
    10), k.lim = range(-0.3, 0), A = 10, P = 4)
plotFun(t + 1/k <= 0 ~ t + k, add = TRUE, npts = 300, alpha =
0.2)
```
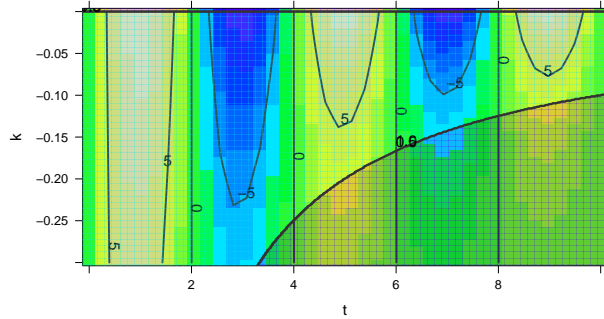
```

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 `makeFun()`.

```
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.[1] masks the original D) operator from the `stats` package.[2]

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

---

[1]`mosaic D`

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

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 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()`:

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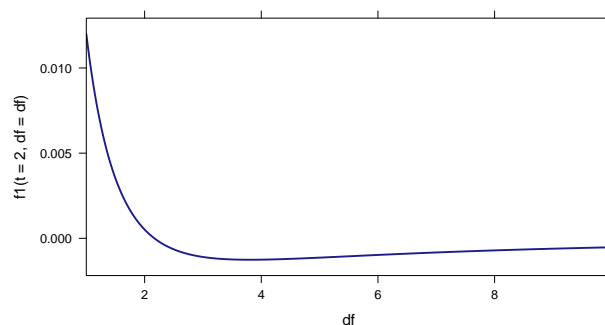f1 = D(dt(t, df) ~ df)
f1(t = 2, df = 1)

## [1] 0.012

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


Numerical differentiation, especially high-order differentiation, has problematic numerical properties. For this reason, only second-order numerical differ-

entiation 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 $\frac{a}{3}x^3$):

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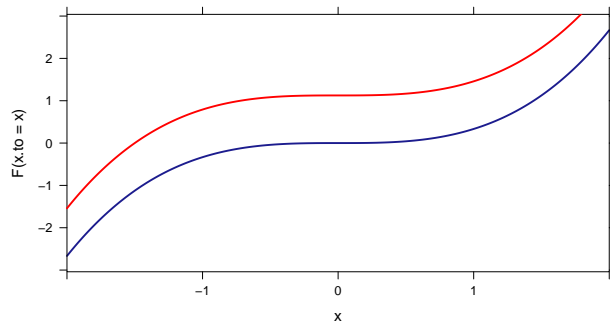F = antiD(a * x^2 ~ x, a = 1)
F

## function (x.to = NaN, x.from = 0, initVal = 0, a = 1)
## {
##     args <- lapply(as.list(match.call())[-1L], eval, parent.frame())
##     names <- if (is.null(names(args)))
##         character(length(args))
##     else names(args)
##     dovec <- names %in% vectorize.args
##     do.call("mapply", c(FUN = FUN, args[dovec], MoreArgs = list(args[!dovec]),
##         SIMPLIFY = SIMPLIFY, USE.NAMES = USE.NAMES))
## }
## <environment: 0x104c9b468>

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

## [1] 0.3333
```

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

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

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] 3.142
```

11

4.1 Example: Jumping Off a Diving Board

The `antiD()` function allows you to specify the "constant of integration." This is another style of providing information redundant with the `x.from` argument, but the constant of integration provides perhaps a clearer way to think about integrals where there is an initial condition.

As an example, imagine jumping off a diving board. Let the initial velocity be 1 m/s and the height of the board 5 m. Acceleration due to gravity is $-9.9m/s$. The velocity function is the integral of acceleration due to gravity over time, plus the initial velocity:

```
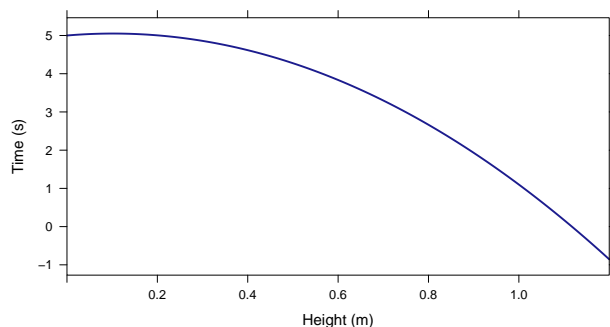vel <- antiD(-9.8 ~ t, Const = 1)
```

The position is, of course, the integral of velocity plus the initial position of 5 m — the height of the board.

```
pos <- antiD(vel(t.to = t) ~ t, Const = 5)
```

```
plotFun(pos(t.to = t) ~ t, t.lim = range(0, 1.2), xlab = "Height
(m)",
    ylab = "Time (s)")
```


The differential equation solver provides another way to approach this problem.

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:

- Find the zeros within a specified range:

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

```
## [1] -6.283 -3.142  0.000  3.142
```

- Find the nearest several zeros to a point:

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

```
## [1]  3.142  6.283  9.425 12.566 15.708
```

- Specify a range via a center and width:

```
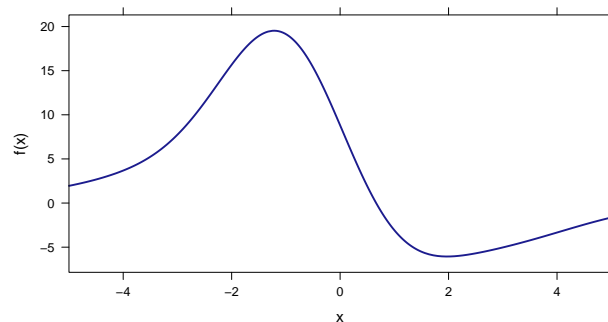findZeros(sin(t) ~ t, near = 0, within = 8)
```

```
## [1] -9.425 -6.283 -3.142  0.000  3.142  6.283  9.425
```

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.

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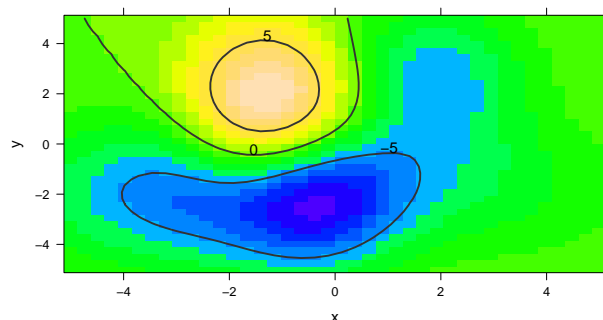f = rfun(~x, seed = 345)
plotFun(f(x) ~ x, x.lim = range(-5, 5))
```

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)
plotFun(f(x, y) ~ x + y, x.lim = range(-5, 5), y.lim = range(-5,
    5))
```


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.

7.1 Interpolators

Interpolating function connect data points. Different interpolating functions have different properties of smoothness, monotonicity, end-points, etc. These properties can be important in modeling.

At present, `mosaic` implements only interpolating functions of one variable.

To illustrate, here are 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.

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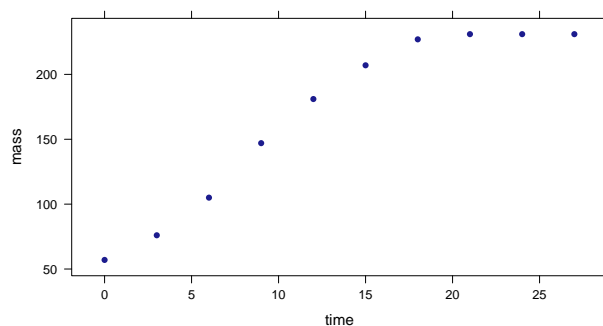water <- data.frame(mass = c(57, 76, 105, 147, 181, 207, 227,
231,
    231, 231), time = c(0, 3, 6, 9, 12, 15, 18, 21, 24, 27))
```

Plotting out the data can be done in the usual way (using `lattice` graphics)

```
xyplot(mass ~ time, data = water)
```


Of course, the mass in the cup varied continuously with time. It's just the recorded data that are discrete. Here's how to create a cubic-spline interpolant that connects the measured data:[3]

```
f <- spliner(mass ~ time, data = water)
```

The function `f()` created has input `time`. It's been arranged so that when `time` is one of the values in the data `water`, the output will be the corresponding value of `mass` in the data.

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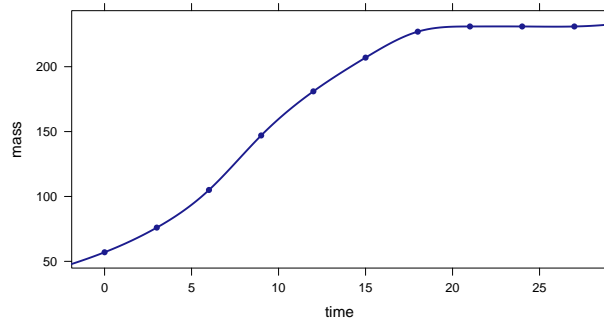f(time = c(0, 3, 6))
```

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

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

```
f(time = c(0, 0.5, 1, 1.5, 2, 2.5, 3))
```

```
## [1] 57.00 59.71 62.59 65.64 68.89 72.33 76.00
```

---

[3]Should this function be changed to `makeSpline()`?

15

```
xyplot(mass ~ time, data = water)
plotFun(f(t) ~ t, add = TRUE, t.lim = range(0, 27))
```


Like any other smooth function, `f()` can be differentiated:

```
Df <- D(f(t) ~ t)
```

There are, of course, other interpolating functions. In situations that demand monotonicity (remember, the water was being poured into the cup, not spilling or draining out), monotonic, smooth splines can be created

```
fmono <- spliner(mass ~ time, data = water, monotonic = TRUE)
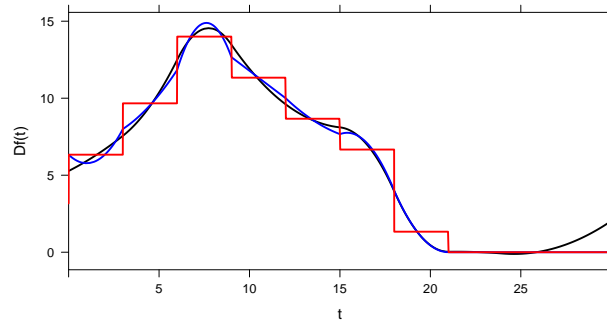```

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

```
fline <- connector(mass ~ time, data = water)
```

The mathematical issues of smoothness and monotonicity are illustrated by these various interpolating functions in a natural way. Sometimes these are better ways to think about the choice of functions for modeling — you're not going to find a global polynomial or exponential or any other classical function to represent these data.

Consider, for instance, the question of determining the rate of flow from the bottle. This is the derivative of the mass measurement. Here are plots of the derivatives of the three interpolating functions:

```
Df <- D(f(t) ~ t)
Dfmono <- D(fmono(t) ~ t)
Dfline <- D(fline(t) ~ t)
plotFun(Df(t) ~ t, t.lim = range(0, 30), lwd = 2, col = "black")
plotFun(Dfmono(t) ~ t, t.lim = range(0, 30), add = TRUE, col =
"blue")
plotFun(Dfline(t) ~ t, t.lim = range(0, 30), add = TRUE, col =
"red")
```

It's a worthwhile classroom discussion: Which of the three estimates of flow is best? The smoothest one has the unhappy property of negative flow near time 25 and positive flow even after the pouring stopped.

Currently, only interpolating functions of one variable are available through the `mosaic` interface.

7.2 Smoothers

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 `CPS` that gives wage and demographic data for 534 people.

```
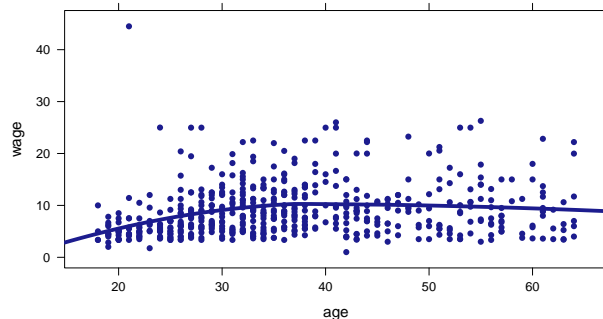data(CPS)
```

There is no definite relationship between wage and age, but there are general trends:

```
xyplot(wage ~ age, data = CPS)
f <- smoother(wage ~ age, span = 0.9, data = CPS)
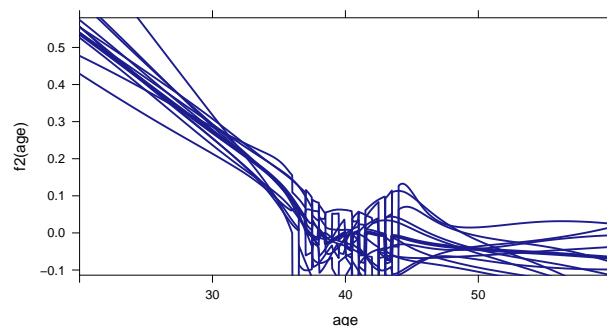plotFun(f(age) ~ age, add = TRUE, lwd = 4)
```


17

There appears to be a slight decline in wage — a negative derivative — for people older than 40. Statistically, one might wonder whether the data provide good evidence for this small effect and the extent to which the outlier affects matters. Let's look at the resampling distribution of the second derivative, stripping away the outlier:

```
CPS2 <- subset(CPS, wage < 30)
f <- smoother(wage ~ age, span = 0.9, data = CPS2)
f2 <- D(f(age) ~ age)
plotFun(f2(age) ~ age, age.lim = range(20, 60), lwd = 4)
do(10) * {
    fr <- smoother(wage ~ age, span = 0.9, data = resample(CPS2))
    fr2 <- D(fr(age) ~ age)
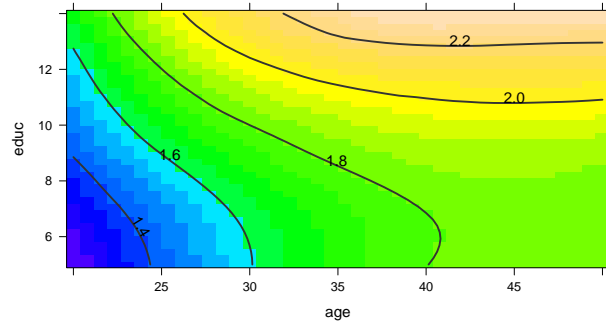    plotFun(fr2(age) ~ age, add = TRUE)
}

## list()

# just to display the plot
plotFun(1 ~ age, add = TRUE)
```


Pretty good evidence for wage going up with age, particularly for the young. The rate of increase per year of age gets smaller and smaller. After about 35 years of age, there's weak evidence for any systematic effect.

Smoothers can construct functions of more than one variable. Here, for instance, is a representation of the relationship between wage, age, and education.

```
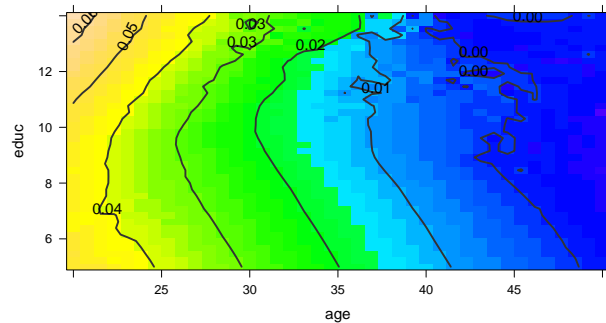g <- smoother(log(wage) ~ age + educ + 1, span = 0.9, data =
CPS2)
plotFun(g(age = age, educ = educ) ~ age + educ, age.lim =
range(20,
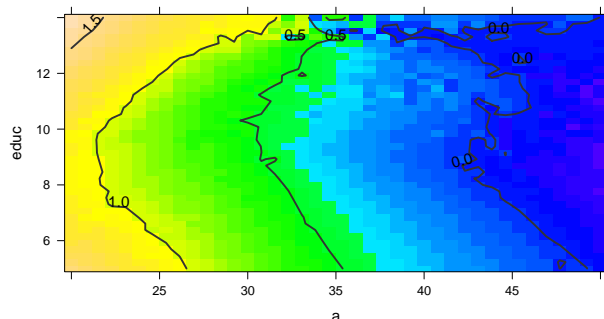    50), educ.lim = range(5, 14))
```

The graph suggests that people with a longer education see a steeper and more prolonged increase in wage with age. To see this in a different way, here's the partial derivative of log wage with respect to age, holding education constant:

```
DgAge <- D(g(age = age, educ = educ) ~ age)
plotFun(DgAge(age = age, educ = educ) ~ age + educ, age.lim =
range(20,
    50), educ.lim = range(5, 14))
```


For economists, perhaps worthwhile to think about the elasticity of wages with respect to age for different levels of education.

```
CPS2$logage <- log(CPS2$age)
g2 <- smoother(log(wage) ~ logage + educ + 1, span = 0.9, data =
CPS2)
elasticity <- D(g2(logage = logage, educ = educ) ~ logage)
plotFun(elasticity(logage = log(a), educ = educ) ~ a + educ,
a.lim = range(20,
    50), educ.lim = range(5, 14))
```

19

Elasticity seems to fall off with age at pretty much the same rate for different levels of education. It just starts out more positive for the more highly educated.

7.3 Fitted Functions

Statisticians will be familiar with parametric functions fitted to data. The `lm()` operator returns information about the fitted model. `mosaic` provides `linearModel()`, which takes the output of `lm()` and packages it up into a model function.

```
g <- linearModel(log(wage) ~ age * educ + 1, data = CPS)
g(age = 40, educ = 12)

##     1
## 2.014

dgdeduc <- D(g(age = age, educ = educ) ~ educ)
dgdeduc(age = 40, educ = 12)

##       1
## 0.08422
```

The `mosaic` function `nlsModel()` provides similar capabilities for nonlinear models, that is, models that are nonlinear in their parameters such as exponentials and sums of exponentials.

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.

```
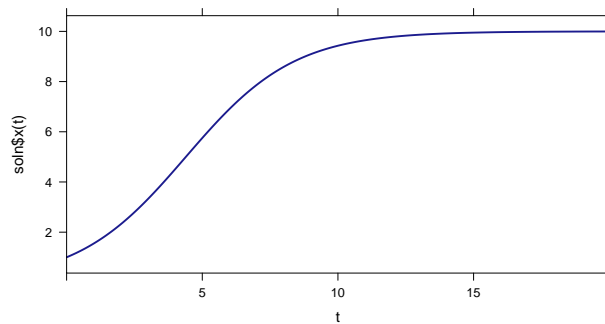soln <- integrateODE(dx ~ r * x * (1 - x/K), x = 1, K = 10, r =
0.5,
    tdur = list(from = 0, to = 20))
```

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] 1.000 1.548 2.320 3.324 4.509 5.751
```

Often, you will plot out the solution against time:

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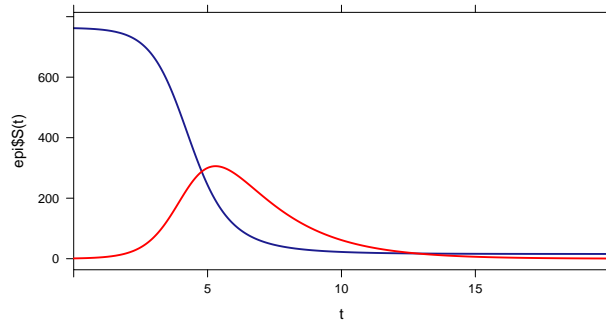plotFun(soln$x(t) ~ t, t.lim = range(0, 20))
```


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 = 0.5, S = 762, I = 1, tdur = 20)
```

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

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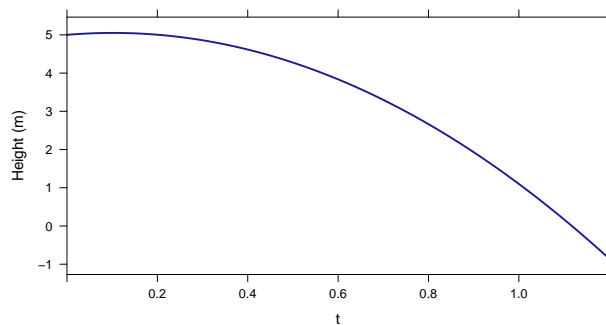plotFun(epi$I(t) ~ t, add = TRUE, col = "red")
```

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.

8.1 Example: Another Dive from the Board

Let's return to the diving-board example, solving it as a differential equation with state variables v and x.

```
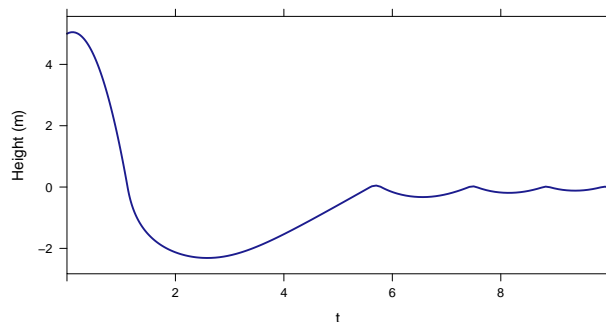dive = integrateODE(dv ~ -9.8, dx ~ v, v = 1, x = 5, tdur = 1.2)
plotFun(dive$x(t) ~ t, t.lim = range(0, 1.2), ylab = "Height
(m)")
```


What's nice about the differential equation format is that it's easy to add features like the buoyancy of water and drag of the water. We'll do that here by changing the acceleration (the dv term) so that when $x < 0$ the acceleration is slightly positive with a drag term proportional to v^2 in the direction opposed to the motion.

```
diveFloat = integrateODE(dv ~ ifelse(x > 0, -9.8, 1 - sign(v) *
v^2),
    dx ~ v, v = 1, x = 5, tdur = 10)
```

23

```
plotFun(diveFloat$x(t) ~ t, t.lim = range(0, 10), ylab = "Height
(m)")
```


According to the model, the diver resurfaces after slightly more than 5 seconds, and then bobs in the water. One might adjust the parameters for buoyancy and drag to try to match the observed trajectory.

9 Algebra and 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. As stated in the Mathematical Association of America's CRAFTY report, "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." [?, p.1] 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, in the 2010 *Notices of the American Mathematical Society*, 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."

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 last hundred years or more, however, there have been numerical approaches to calculus problems.

The numerical approaches are rarely emphasized in introductory calculus, except as demonstrations when trying to help students visualize operations like the integral that are otherwise too abstract. 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.[4] 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

---

[4]The Ryacas package already provides an R interface to a computer algebra system.
[5]NSF DUE 0920350