Numerical Integration and Differentiation

November 26, 2013

References:

- Gentle: Computational Statistics
- Monahan: Numerical Methods of Statistics
- Givens and Hoeting: Computational Statistics

Our goal here is to understand the basics of numerical (and symbolic) approaches to approximating derivatives and integrals on a computer. Derivatives are useful primarily for optimization. Integrals arise in approximating expected values and in various places where we need to integrate over an unknown random variable (e.g., Bayesian contexts, random effects models, missing data contexts).

1 Differentiation

1.1 Numerical differentiation

There's not much to this topic. The basic idea is to approximate the derivative of interest using finite differences.

A standard discrete approximation of the derivative is the forward difference

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

A more accurate approach is the central difference

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Provided we already have computed $f(x)$, the forward difference takes half as much computing as the central difference. However, the central difference has an error of $O(h^2)$ while the forward difference has error of $O(h)$.

For second derivatives, if we apply the above approximations to $f'(x)$ and $f'(x+h)$, we get an approximation of the second derivative based on second differences:

$$f''(x) \approx \frac{f'(x+h) - f'(x)}{h} \approx \frac{f(x+2h) - 2f(x+h) + f(x)}{h^2}.$$

The corresponding central difference approximation is

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}.$$

For multivariate x, we need to compute directional derivatives. In general these will be in axis-oriented directions (e.g., for the Hessian), but they can be in other directions. The basic idea is to find $f(x + he)$ in expressions such as those above where e is a unit length vector giving the direction. For axis oriented directions, we have e_i being a vector with a one in the ith position and zeroes in the other positions,

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x + he_i) - f(x)}{h}.$$

Note that for mixed partial derivatives, we need to use e_i and e_j, so the second difference approximation gets a bit more complicated,

$$\frac{\partial^2 f}{\partial x_i \partial x_j} \approx \frac{f(x + he_j + he_i) - f(x + he_j) - f(x + he_i) + f(x)}{h^2}.$$

We would have analogous quantities for central difference approximations.

Numerical issues Ideally we would take h very small and get a highly accurate estimate of the derivative. However, the limits of machine precision mean that the difference estimator can behave badly for very small h, since we lose accuracy in computing differences such as between $f(x+h)$ and $f(x-h)$ and from dividing by small h. Therefore we accept a bias in the estimate by not using h so small, often by taking h to be square root of machine epsilon (i.e., about 1×10^{-8} on most systems). Actually, we need to account for the order of magnitude of x, so what we really want is $h = \sqrt{\epsilon}|x|$ - i.e., we want it to be in terms relative to the magnitude of x. As an example, recall that if $x = 1 \times 10^9$ and we did $x + h = 1 \times 10^9 + 1 \times 10^{-8}$, we would get $x + h = 1 \times 10^9 = x$ because we can only represent 7 decimal places with precision.

Givens and Hoeting and Monahan point out that some sources recommend the cube root of machine epsilon (about 5×10^{-6} on most systems), in particular when approximating second

derivatives.

Let's assess these recommendations empirically in R. We'll use a test function, $\log \Gamma(x)$, for which we can obtain the derivatives with high accuracy using built-in R functions. This is a modification of Monahan's example from his *numdif.r* code.

```
# compute first and second derivatives of log(gamma(x)) at x=1/2
options(digits = 9, width = 120)
h <- 10^(-(1:15))
x <- 1/2
fx <- lgamma(x)
# targets: actual derivatives can be computed very accurately using
# built-in R functions:
digamma(x)  # accurate first derivative

## [1] -1.96351003

trigamma(x)  # accurate second derivative

## [1] 4.9348022

# calculate discrete differences
fxph <- lgamma(x + h)
fxmh <- lgamma(x - h)
fxp2h <- lgamma(x + 2 * h)
fxm2h <- lgamma(x - 2 * h)
# now find numerical derivatives
fp1 <- (fxph - fx)/h  # forward difference
fp2 <- (fxph - fxmh)/(2 * h)  # central difference
# second derivatives
fpp1 <- (fxp2h - 2 * fxph + fx)/(h * h)  # forward difference
fpp2 <- (fxph - 2 * fx + fxmh)/(h * h)  # central difference
# table of results
cbind(h, fp1, fp2, fpp1, fpp2)

##            h         fp1         fp2             fpp1             fpp2
##  [1,] 1e-01 -1.74131085 -1.99221980  3.67644733e+00  5.01817899e+00
##  [2,] 1e-02 -1.93911250 -1.96379057  4.77200996e+00  4.93561416e+00
##  [3,] 1e-03 -1.96104543 -1.96351283  4.91803003e+00  4.93481032e+00
```

```
##  [4,] 1e-04 -1.96326331 -1.96351005  4.93311987e+00  4.93480230e+00
##  [5,] 1e-05 -1.96348535 -1.96351003  4.93463270e+00  4.93480257e+00
##  [6,] 1e-06 -1.96350756 -1.96351003  4.93505237e+00  4.93483032e+00
##  [7,] 1e-07 -1.96350978 -1.96351003  4.91828800e+00  4.95159469e+00
##  [8,] 1e-08 -1.96351001 -1.96351003  7.77156117e+00  3.33066907e+00
##  [9,] 1e-09 -1.96351002 -1.96351002 -1.11022302e+02  0.00000000e+00
## [10,] 1e-10 -1.96351047 -1.96351047  2.22044605e+04  0.00000000e+00
## [11,] 1e-11 -1.96349603 -1.96350158 -3.33066907e+06  1.11022302e+06
## [12,] 1e-12 -1.96342942 -1.96348493 -1.11022302e+08  1.11022302e+08
## [13,] 1e-13 -1.96398453 -1.96398453  2.22044605e+10  0.00000000e+00
## [14,] 1e-14 -1.96509475 -1.97064587  1.11022302e+12  1.11022302e+12
## [15,] 1e-15 -1.99840144 -1.94289029  0.00000000e+00 -1.11022302e+14
```

What do we conclude about the advice about using h proportional to either the square root or cube root of machine epsilon?

1.2 Numerical differentiation in R

There are multiple numerical derivative functions in R. *numericDeriv()* will do the first derivative. It requires an expression rather than a function as the form in which the function is input, which in some cases might be inconvenient. The functions in the *numDeriv* package will compute the gradient and Hessian, either in the standard way (using the argument `method = 'simple'`) or with a more accurate approximation (using the argument `method = 'Richardson'`). For optimization, one might use the simple option, assuming that is faster, while the more accurate approximation might be good for computing the Hessian to approximate the information matrix for getting an asymptotic covariance. (Although in this case, the statistical uncertainty generally will ovewhelm any numerical uncertainty.)

```
x <- 1/2
numericDeriv(quote(lgamma(x)), "x")

## [1] 0.572364943
## attr(,"gradient")
##             [,1]
## [1,] -1.96351001
```

Note that by default, if you rely on numerical derivatives in *optim()*, it uses $h = 0.001$ (the *ndeps* sub-argument to *control*), which might not be appropriate if the parameters vary on a small

scale. This relatively large value of h is probably chosen based on *optim()* assuming that you've scaled the parameters as described in the text describing the *parscale* argument.

1.3 Symbolic differentiation

We've seen that we often need the first and second derivatives for optimization. Numerical differentiation is fine, but if we can readily compute the derivatives in closed form, that can improve our optimization. (Venables and Ripley comment that this is particularly the case for the first derivative, but not as much for the second.)

In general, using a computer program to do the analytic differentiation is recommended as it's easy to make errors in doing differentiation by hand. Monahan points out that one of the main causes of error in optimization is human error in coding analytic derivatives, so it's good practice to avoid this. R has a simple differentiation ability in the *deriv()* function (which handles the gradient and the Hessian). However it can only handle a limited number of functions. Here's an example of using *deriv()* and then embedding the resulting R code in a user-defined function. This can be quite handy, though the format of the result in terms of attributes is not the most handy, so you might want to monkey around with the code more in practice.

```
deriv(quote(atan(x)), "x")  # derivative of simple expression

## expression({
##     .value <- atan(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- 1/(1 + x^2)
##     attr(.value, "gradient") <- .grad
##     .value
## })

# derivative of a function; note we need to pass in an expression,
# not the entire function
f <- function(x,y) sin(x * y)+x^3+exp(y)
newBody <- deriv(body(f), c("x", "y"), hessian = TRUE)
# now create a new version of f that provides gradient
# and hessian as attributes of the output,
# in addition to the function value as the return value
f <- function(x, y) {} # function template
body(f) <- newBody
```

```
# try out the new function
f(3,1)

## [1] 29.8594018
## attr(,"gradient")
##                x            y
## [1,] 26.0100075 -0.251695661
## attr(,"hessian")
## , , x
##
##              x           y
## [1,] 17.85888 -1.41335252
##
## , , y
##
##                 x          y
## [1,] -1.41335252 1.44820176

attr(f(3,1), "gradient")

##                x            y
## [1,] 26.0100075 -0.251695661

attr(f(3,1), "hessian")

## , , x
##
##              x           y
## [1,] 17.85888 -1.41335252
##
## , , y
##
##                 x          y
## [1,] -1.41335252 1.44820176
```

For more complicated functions, both Maple and Mathematica do symbolic differentiation. Here are some examples in Mathematica, which is available on the SCF machines and through campus: http://ist.berkeley.edu/software-central:

```
# first partial derivative wrt x
D[ Exp[x^n] - Cos[x y], x]
# second partial derivative
D[ Exp[x^n] - Cos[x y], {x, 2}]
# partials
D[ Exp[x^n] - Cos[x y], x, y]
# trig function example
D[ ArcTan[x], x]
```

2 Integration

We've actually already discussed numerical integration extensively in the simulation unit, where we considered Monte Carlo approximation of high-dimensional integrals. In the case where we have an integral in just one or two dimensions, MC is fine, but we can get highly-accurate, very fast approximations by numerical integration methods known as quadrature. Unfortunately such approximations scale very badly as the dimension grows, while MC methods scale well, so MC is recommended in higher dimensions. Here's an empirical example in R, where the MC estimator is

$$\int_0^\pi \sin(x)dx = \int_0^\pi \pi \sin(x) \left(\frac{1}{\pi} \cdot 1\right) dx = E_f(\pi \sin(x))$$

for $f = \mathcal{U}(0, \pi)$:

```
f <- function(x) sin(x)
# mathematically, the integral from 0 to pi is 2
# quadrature through integrate()
integrate(f, 0, pi)

## 2 with absolute error < 2.2e-14

system.time(integrate(f, 0, pi))

##    user  system elapsed
##       0       0       0

# MC estimate
ninteg <- function(n) mean(sin(runif(n, 0, pi))*pi)
n <- 1000
ninteg(n)
```

```
## [1] 2.00825866

system.time(ninteg(n))

##    user  system elapsed
##   0.000   0.000   0.001

n <- 10000
ninteg(n)

## [1] 2.00712587

system.time(ninteg(n))

##    user  system elapsed
##       0       0       0

n <- 1000000
ninteg(n)

## [1] 2.00068404

system.time(ninteg(n))

##    user  system elapsed
##   0.084   0.012   0.098

# that was fairly slow,
# especially if you need to do a lot of individual integrals
```

More on this issue below.

2.1 Numerical integration methods

The basic idea is to break the domain into pieces and approximate the integral within each piece:

$$\int_a^b f(x)dx = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)dx,$$

where we then approximate $\int_{x_i}^{x_{i+1}} f(x)dx \approx \sum_{j=0}^{m} A_{ij} f(x_{ij}^*)$ where x_{ij}^* are the *nodes*.

2.1.1 Newton-Cotes quadrature

Newton-Cotes quadrature has equal length intervals of length $h = (b-a)/n$, with the same number of nodes in each interval. $f(x)$ is replaced with a polynomial approximation in each interval and A_{ij} are chosen so that the sum equals the integral of the polynomial approximation on the interval.

A basic example is the *Riemann rule*, which takes a single node, $x_i^* = x_i$ and the "polynomial" is a constant, $f(x_i^*)$, so we have

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx (x_{i+1} - x_i)f(x_i).$$

Of course using a piecewise constant to approximate $f(x)$ is not likely to give us high accuracy.

The *trapezoidal rule* takes $x_{i0}^* = x_i$, $x_{i1}^* = x_{i+1}$ and uses a linear interpolation between $f(x_{i0}^*)$ and $f(x_{i1}^*)$ to give

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx \left(\frac{x_{i+1} - x_i}{2}\right)(f(x_i) + f(x_{i+1})).$$

Simpson's rule uses a quadratic interpolation at the points $x_{i0}^* = x_i$, $x_{i1}^* = (x_i + x_{i+1})/2$, $x_{i2}^* = x_{i+1}$ to give

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx \left(\frac{x_{i+1} - x_i}{6}\right)\left(f(x_i) + 4f\left(\frac{x_i + x_{i+1}}{2}\right) + f(x_{i+1})\right).$$

The error of various rules is often quantified as a power of $h = x_{i+1} - x_i$. The trapezoid rule gives $O(h^2)$ while Simpson's rule gives $O(h^4)$.

Romberg quadrature There is an extension of Newton-Cotes quadrature that takes combinations of estimates based on different numbers of intervals. This is called Richardson extrapolation and when used with the trapezoidal rule is called *Romberg quadrature*. The result is greatly increased accuracy. A simple example of this is as follows. Let $\hat{T}(h)$ be the trapezoidal rule approximation of the integral when the length of each interval is h. Then $\frac{4\hat{T}(h/2)-\hat{T}(h)}{3}$ results in an approximation with error of $O(h^4)$ because the differencing is cleverly chosen to kill off the error term that is $O(h^2)$. In fact this approximation is Simpson's rule with intervals of length $h/2$, with the advantage that we don't have to do as many function evaluations ($2n$ vs. $4n$). Even better, one can iterate this approach for more accuracy as described in detail in Givens and Hoeting.

Note that at some point, simply making intervals smaller in quadrature will not improve accuracy because of errors introduced by the imprecision of computer numbers.

2.1.2 Gaussian quadrature

Here the idea is to relax the constraints of equally-spaced intervals and nodes within intervals. We want to put more nodes where the function is larger in magnitude.

Gaussian quadrature approximates integrals that are in the form of an expected value as

$$\int_a^b f(x)\mu(x)dx \approx \sum_{i=0}^{m} w_i f(x_i)$$

where $\mu(x)$ is a probability density, with the requirement that $\int x^k \mu(x)dx = E_\mu X^k < \infty$ for $k \geq 0$. Note that it can also deal with indefinite integrals where $a = -\infty$ and/or $b = \infty$. Typically μ is non-uniform, so the nodes (the quadrature points) cluster in areas of high density.The choice of node locations depends on understanding orthogonal polynomials, which we won't go into here.

It turns out this approach can exactly integrate polynomials of degree $2m + 1$ (or lower). The advantage is that for smooth functions that can be approximated well by a single polynomial, we get highly accurate results. The downside is that if the function is not well approximated by such a polynomial, the result may not be so good. The Romberg approach is more robust.

Note that if the problem is not in the form $\int_a^b f(x)\mu(x)dx$, but rather $\int_a^b f(x)dx$, we can reexpress as $\int_a^b \frac{f(x)}{\mu(x)}\mu(x)dx$.

Note that the trapezoidal rule amounts to μ being the uniform distribution with the points equally spaced.

2.1.3 Adaptive quadrature

Adaptive quadrature chooses interval lengths based on the behavior of the integrand. The goal is to have shorter intervals where the function varies more and longer intervals where it varies less. The reason for avoiding short intervals everywhere involves the extra computation and greater opportunity for rounding error.

2.1.4 Higher dimensions

For rectangular regions, one can use the techniques described above over squares instead of intervals, but things become more difficult with more complicated regions of integration.

The basic result for Monte Carlo integration (i.e., Unit 10 on simulation) is that the error of the MC estimator scales as $O(m^{-1/2})$, where m is the number of MC samples, regardless of dimensionality. Let's consider how the error of quadrature scales. We've seen that the error is often quantified as $O(h^q)$. In d dimensions, the error is the same as a function of h, but if in one dimension we need n function evaluations to get intervals of length h, in d dimensions, we need n^d function evaluations to get hypercubes with sides of length h. Let's re-express the error in terms

of n rather than h based on $h = c/n$ for a constant c (such as $c = b - a$), which gives us error of $O(n^{-q})$ for one-dimensional integration. In d dimensions we have $n^{1/d}$ function evaluations per dimension, so the error for fixed n is $O((n^{1/d})^{-q}) = O(n^{-q/d})$ which scales as $n^{-1/d}$. As an example, suppose $d = 10$ and we have $n = 1000$ function evaluations. This gives us an accuracy comparable to one-dimensional integration with $n = 1000^{1/10} \approx 2$, which is awful. Even with only $d = 4$, we get $n = 1000^{1/4} \approx 6$, which is pretty bad. This is one version of the curse of dimensionality.

2.2 Numerical integration in R

R implements an adaptive version of Gaussian quadrature in *integrate()*. The '...' argument allows you to pass additional arguments to the function that is being integrated. The function must be vectorized (i.e., accept a vector of inputs and evaluate and return the function value for each input as a vector of outputs).

Note that the domain of integration can be unbounded and if either the upper or lower limit is unbounded, you should enter **`Inf`** or **`-Inf`** respectively.

```
integrate(dnorm, -Inf, Inf, 0, 0.1)

## 1 with absolute error < 6.1e-07

integrate(dnorm, -Inf, Inf, 0, 0.001)

## 1 with absolute error < 2.1e-06

integrate(dnorm, -Inf, Inf, 0, 1e-04)  # THIS FAILS!

## 0 with absolute error < 0
```

2.3 Singularities and infinite ranges

A singularity occurs when the function is unbounded, which can cause difficulties with numerical integration. For example, $\int_0^1 \frac{1}{\sqrt{x}} = 2$, but $f(0) = \infty$. One strategy is a change of variables. For example, to find $\int_0^1 \frac{\exp(x)}{\sqrt{x}} dx$, let $u = \sqrt{x}$, which gives the integral, $2\int_0^1 \exp(u^2) du$.

Another strategy is to subtract off the singularity. E.g., in the example above, reexpress as

$$\int_0^1 \frac{\exp(x) - 1}{\sqrt{x}} dx + \int_0^1 \frac{1}{\sqrt{x}} dx = \int_0^1 \frac{\exp(x) - 1}{\sqrt{x}} dx + 2$$

where we do the second integral analytically. It turns out that the first integral is well-behaved at 0.

It turns out that R's *integrate()* function can handle $\int_0^1 \frac{\exp(x)}{\sqrt{x}} dx$ directly without us changing the problem statement analytically. Perhaps this has something to do with the use of adaptive quadrature, but I'm not sure.

```
# doing it directly with integrate()
f <- function(x) exp(x)/sqrt(x)
integrate(f, 0, 1)

## 2.92530349 with absolute error < 9.4e-06

# subtracting off the singularity
f <- function(x) (exp(x) - 1)/sqrt(x)
x <- seq(0, 1, len = 200)
integrate(f, 0, 1)

## 0.925303567 with absolute error < 7.6e-05


# analytic change of variables, followed by numeric integration
f <- function(u) 2 * exp(u^2)
integrate(f, 0, 1)

## 2.92530349 with absolute error < 3.2e-14
```

Infinite ranges Gaussian quadrature deals with the case that $a = -\infty$ and/or $b = \infty$. Another possibility is change of variables using transformations such as $1/x$, $\exp(x)/(1 + \exp(x))$, $\exp(-x)$, and $x/(1 + x)$.

2.4 Symbolic integration

Mathematica and Maple are able to do symbolic integration for many problems that are very hard to do by hand (and with the same concerns as when doing differentiation by hand). So this may be worth a try.

```
# one-dimensional integration
Integrate[Sin[x]^2, x]
```

```
# two-dimensional integration
Integrate[Sin[x] Exp[-y^2], x, y]
```