Statistical Computing 2

Anthony Stephenson

11/19/2020

Contents

Debugging

Bugs are an inevitable part of programming. As exemplified by the line Andrew Jackson stated in the film *Alpha Go*: “If DeepMind has figured out how to write code that doesn’t have bugs, that is a bigger news story then AlphaGo.”. As such, it is essential to devise an efficient procedure for finding and fixing bugs.

General debugging strategies

Google: when an error is thrown and it isn’t immediately obvious why, google the error message as the vast majority of the time you will not be the first person to have your problem, and there may well be a stackexchange/overflow post about it, including a fix.

Unit testing is one good way to by default incorporate minimal working examples to reproduce and subsequently fix bugs. Unit tests force the coder to consider the expected output from various inputs, and check whether the code behaves as intended as well as to try and guess how the function might be misused/fed unexpected input and how the function does and should behave in those circumstances. Of course, one can't forsee everything, but this tends to reduce the amount of unexpected behaviour and bugs.

Once an error is found, figuring out precisely what causes the bug follows a similar protocol to writing good tests, in that unless it turns out to be immediate and trivial, a minimal working example is useful to construct, from which inputs/outputs can be checked and behaviour monitored to try and narrow down the cause. The example can then also be used as the basis for a new test to make sure it doesn't get reintroduced in future modifications.

This argument is also why it makes sense to make code as modular as possible, so that bugs can be narrowed down to a particular function that does a singular task as quickly as possible.

Using an interactive debugger designed for the language being written also tends to be advisable, as they allow one to pause execution at a particular line, analyse the variables in the workspace at that place and scope and step through line-by-line to see how those variables change with each computation. Conditional breakpoints can be very useful in, say, for loops, where code is executing the same line repeatedly and it could take a long time to manually inspect the variables at each iteration. Instead, a breakpoint that acts only when something that you suspect *might* reproduce the error occurs.

If an interactive debugger is not available or cannot be used, then printing out objects at particular lines can work reasonably well to check whether the object has the value/structure you expect at that point given the rest of the code.

Debugging in R

Use `traceback()` to find more information about a bug, by providing the call stack that preceeded the error, since the default R error messages can be somewhat opaque.

If a bug causes R to crash, then the cause is probably due to a bug in compiled C/++ code, which may require the use of an interactive C debugger.

Profiling

Profiling is a useful way to measure the speed at which different segments of code run. This enables the writer to determine where performance bottlenecks lie and hence improve the performance of the code by addressing the effiency of the implementation in these places.

In R, the `profvis` package is a profile implementing `statistical profiling`. This method interrupts the execution at regular intervals to record what line the code is running to build up a picture of how the running time is distributed over the lines of code.

When modifying code to improve the performance, one must be careful to ensure that changes do not break behaviour, e.g. in corner cases. Here, again, unit tests are helpful.

Performance

Being a high-level language, R's performance for certain tasks is not the most efficient, compared to (say) C. For this reason, many packages in R are actually written in C, and if a particularly computationally task is required, an R user can write the code using Rcpp and thereby use a compiled C++ function in R. Note this can be done in other high level languages (Matlab -> mex, Python -> Cython, . . .).

Vectorisation

We have already seen that vectorisation improves performance; this is due to the routines implementing matrix multiplication are implemented in C++ and called in R, whereas writing out the same functionality in loops in R is much slower.

Storage

Data arrays are stored either by *column-major* or *row-major* ordering. This describes the location of values with respect to each other in memory. In particular, in *column-major* (which is R's default) values in an array are stored as contiguous chunks of memory, by column, so that values read sequentially (i.e. down each column, left to right) will be accessed faster.

```
# define a function for later
looper <- function (x, n=100, f=function(x) x^2) for (i in 1:n) { f(x) }
```

Dense Matrices

The default matrix object in R is constructed with the `matrix` function and is stored by column-major order in memory (which means that the elements of the structure, when ordered columnwise, form a contiguous chunk of memory and hence are faster to access sequentially in this order). In R, we can choose to store in row-major order, by specifying `row=T` at construction. In R a matrix can also be assigned row and column names.

For example, imagine we want to generate a matrix of (fake/random) covariates to test statistical algorithms on:

```
set.seed(5)
d <- rnorm(1000, 0, 1)
m <- matrix(d, nrow=100, ncol=10)
m <- cbind(rep(1, 10), m)
cols <- c()
rows <- c()
for (i in 1:10) cols <- c(cols, paste("x", i, sep=""))
for (i in 1:100) rows <- c(rows, paste("r", i, sep=""))
colnames(m) <- c("intercept", cols)
rownames(m) <- rows
print(head(m))
```

```
##    intercept          x1          x2         x3         x4         x5
## r1         1 -0.84085548 -1.99538697 -0.1136782  1.2101461 -0.4701264
## r2         1  1.38435934  1.13531128 -0.2951008  0.1886572  0.4693224
## r3         1 -1.25549186  0.67579457  0.9891685  1.9624987 -1.4753401
## r4         1  0.07014277  0.20848326 -0.7751318  0.1387119 -0.9159547
## r5         1  1.71144087 -0.05784564  0.2758983 -1.5786274  0.4579751
## r6         1 -0.60290798  0.89381141  0.4107816 -0.7970213 -0.7125025
##              x6         x7          x8          x9         x10
## r1 -0.084716818  1.3294962 -0.44111991 -0.23756597  0.77225283
## r2  0.199059827  0.9897694  0.56334117 -0.91430359 -0.99452192
## r3  1.229825830  1.6645658  1.18008944  1.44336433  0.34033713
## r4  0.004552531 -2.1198836 -0.93377333 -0.01370158 -0.43664151
## r5 -1.294607637  0.1414808 -0.02200709 -0.53039735  0.19505235
## r6 -0.397161380 -0.1778062 -0.23324656  0.80979975 -0.08121796
```

If we extract a row of the matrix, we return a list, rather than another matrix, unless we index with an additional argument:

```
dim(m[1,])
```

```
## NULL
```

```
dim(m[1, ,drop=F])
```

```
## [1]  1 11
```

If we want higher dimensional arrays (i.e. with dimension greater than 2) we can use the `array` structure in R.

Solving linear systems

Now imagine we use this to solve a linear model, using `solve`:

```
w <- rnorm(ncol(m), 0, 1)
y <- m %*% w
C <- t(m) %*% m
w_hat_1 <- solve(C) %*% t(m) %*% y
w_hat_2 <- solve(C, t(m) %*% y)
median(w - w_hat_1)
```

```
## [1] 0
```

```
median(w - w_hat_2)
```

```
## [1] 0
```

```
system.time(looper(c(C, m, y), n=10000, function(x) solve(x[1]) %*% t(x[2]) %*% x[3]))
```

```
##    user  system elapsed
##   0.185   0.001   0.186
```

```
system.time(looper(c(C, m, y), n=10000, function(x) solve(x[1], t(x[2]) %*% x[3])))
```

```
##    user  system elapsed
##   0.145   0.003   0.148
```

This example has failed to demonstrate the expected behaviour, that in general, using `solve(A, b)` is more numerically stable than `solve(A)` (i.e. has a smaller error) and should be used most of the time, for dense or sparse matricies `A`.

Other useful R functions for linear algebra include `qr` for QR decomposition, `eigen` for eigen decomposition, `chol` for cholesky decomposition etc.

For example, we can check the rank using `qr`:

```
qr(C)$rank
```

```
## [1] 11
```

Numerical precision and stability

The standard data storage type in R is the *double* which is a 64-bit signed representation of a floating point value, with 1 bit to represent the sign, 11 to represent an exponent and 52 bits to represent the precision. Any floating point calculations will introduce errors of the order of the *machine epsilon* which is defined as the difference between 1 and the next larger floating point value which is given by `b^(p-1)` which for a double is

```
epsilon <- 2^(-52)
epsilon
```

```
## [1] 2.220446e-16
```

Due to the precision issue, to compare floating points, we should use the function `all.equal` and specify the tolerance we want to test "equality" upto. When dealing with integer type objects (*long* and can be set by `L` e.g. `1L`), however, we can directly test equality with the usual `==` operator. Most values in R are stored as doubles.

Sparse Matrices

Sparse matrices are matrices whose entries are mostly 0, meaning that if we implement sparse methods (i.e. we compute operations only on the non-zero entries whose locations are recorded) we should save significant computational overhead in memory and CPU time. This allows us to deal with matrices with very large dimensionality relatively easily (albeit with some care to ensure we don't convert back to a dense representation accidentally) if they are sparse. In R we need to use the package `Matrix` to utilise sparse functionality.

The `Matrix` package stores objects as one of several underlying data types:

- `dgeMatrix` - column-major order dense matrix
- `dgCMatrix` - compressed sparse column (CSC) matrix: stores contiguous blocks of non-zero values along columns, with a pointer to the first non-zero value and row indices of the values.
- `dgRMatrix` - compressed sparse row (CSR) matrix: same logic as CSC, but by row
- `dgTMatrix` - coordinate *triplet format* storage, where each non-zero element is recorded as a triplet `(i, j, x)` with row and column index `[i, j]` and value `x`.

Storage size of different representations

```r
#  construct tridiagonal matrix of random normal values
constructTriDiagM <- function(n=100) {
  x <- rnorm(n, 0, 1)
  y <- rnorm(n-1, 0, 1)

  A <- diag(x)
  A[row(A) - col(A) == 1] <- A[row(A) - col(A) == -1] <- y
  return(A)
}
A <- constructTriDiagM()
sum(A != 0)
```

```
## [1] 298
```

```r
# check the size of A
object.size(A)
```

```
## 80216 bytes
```

```r
# coerce A into a default Matrix (sparse) object, and measure its size
As <- Matrix(A)
str(As)
```

```
## Formal class 'dsCMatrix' [package "Matrix"] with 7 slots
##   ..@ i       : int [1:199] 0 0 1 1 2 2 3 3 4 4 ...
##   ..@ p       : int [1:101] 0 1 3 5 7 9 11 13 15 17 ...
##   ..@ Dim     : int [1:2] 100 100
##   ..@ Dimnames:List of 2
##   .. ..$ : NULL
##   .. ..$ : NULL
##   ..@ x       : num [1:199] 0.197 1.792 1.024 1.381 -0.954 ...
##   ..@ uplo    : chr "U"
##   ..@ factors : list()
```

```r
object.size(As)
```

```
## 4520 bytes
```

```r
# convert into a triplet format sparse matrix
At <- as(A, 'dgTMatrix')
object.size(At)
```

```
## 6264 bytes
```

```r
# convert into a dense Matrix object (and check the size)
object.size(as(A, 'dgeMatrix'))
```

```
## 81176 bytes
```

```r
# convert into a CSR Matrix (and check the size)
Ar <- as(A, 'dgRMatrix')
object.size(Ar)
```

```
## 5480 bytes
```

```r
# we can't convert a column-major order into a row-major order object, or vice versa:
as(as(A, 'dgCMatrix'), 'dgRMatrix')
```

```
## Error in as(as(A, "dgCMatrix"), "dgRMatrix"): no method or default for coercing "dgCMatrix" to "dgRMa
```

Operations on sparse (and dense) matricies

Basic operations that convert sparse into dense matricies:

- Scalar addition (e.g. `A + 10`) since it adds 10 elementwise
- Matrix multiplication with dense vector
- Matrix inverse:

If we try to invert a sparse matrix, we find that the inverted matrix is no longer sparse, as now *all* of the entries are non-zero:

```r
Ainv <- solve(As)
sum(Ainv != 0)
```

```
## [1] 10000
```

```r
prod(dim(A))
```

```
## [1] 10000
```

How long does it take to calculate a larger version of `A^2` in different representations?

```r
A <- constructTriDiagM(n=1000)
As <- as(A, 'dgCMatrix')
Ar <- as(A, 'dgRMatrix')
At <- as(A, 'dgTMatrix')

system.time(looper(A))
```

```
##    user  system elapsed
##   0.170   0.021   0.191
```

```
system.time(looper(As))
```

```
##    user  system elapsed
##   0.003   0.000   0.004
```

```
system.time(looper(At))
```

```
##    user  system elapsed
##   0.021   0.000   0.020
```

```
system.time(looper(Ar))
```

```
##    user  system elapsed
##   0.019   0.001   0.020
```

So we can see that CSC and triplet representations are the fastest, followed by CSR at an order of magnitude slower, and dense at another order of magnitude slower.

How about `B^2` (where `B` is fully dense) in different representations?

```
B <- matrix(rnorm(1e6, 0, 1), nrow=1000, ncol=1000)
system.time(looper(B))
```

```
##    user  system elapsed
##   0.161   0.024   0.185
```

```
B_s <- as(B, 'dgCMatrix')
system.time(looper(B_s))
```

```
##    user  system elapsed
##   0.177   0.024   0.202
```

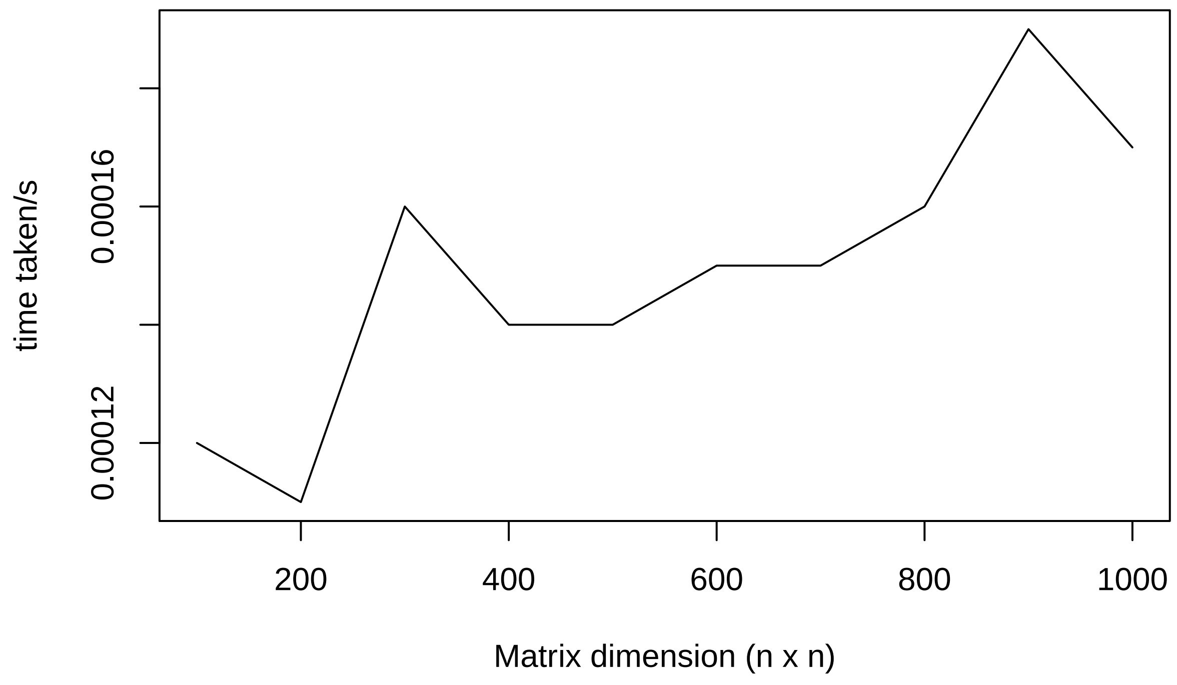
```
B_r <- as(B, 'dgRMatrix')
system.time(looper(B_r))
```

```
##    user  system elapsed
##   1.670   0.221   1.901
```

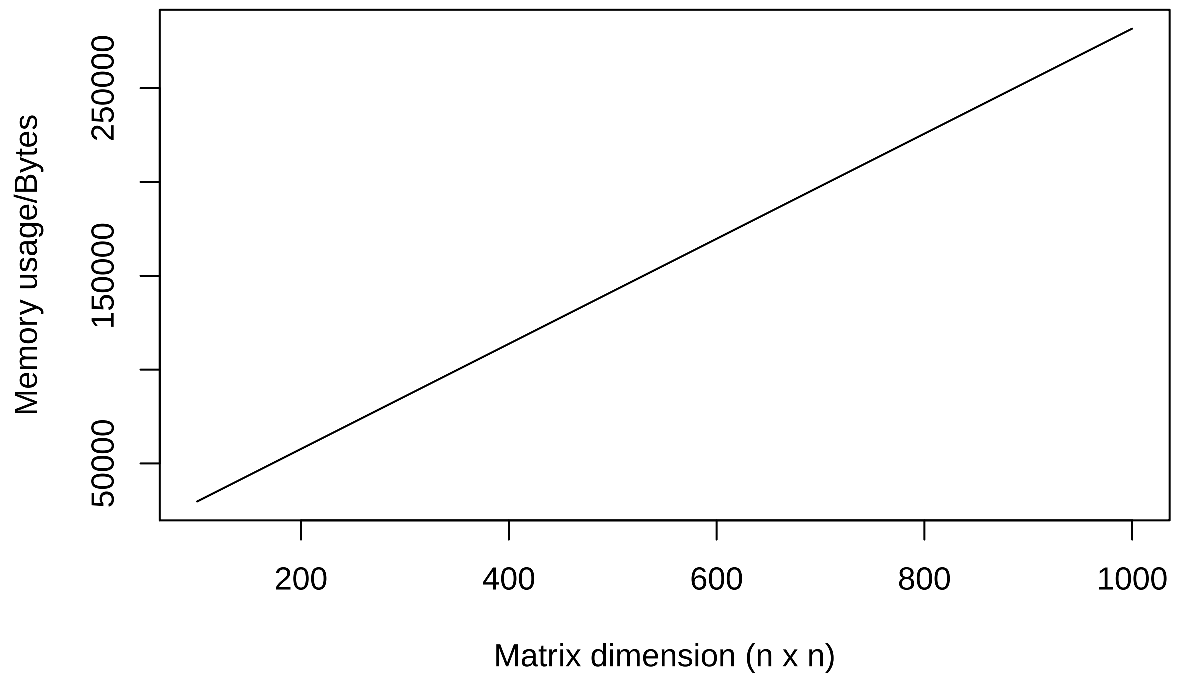
In comparison, in the fully dense case, the CSC and dense representations are effectively equivalent (which is to be expected) whilst the CSR representation is an order of magnitude slower.

We can plot CPU time for different operations in different representations and how they scale with number of elements:

```
t_1 <- c()
s_1 <- c()
x <- 1:10
reps <- 100
for (i in x) {
  A <- Matrix(constructTriDiagM(n=i * 1000))
  t <- system.time(looper(A, n=reps, f=function(x) x^2))
  s <- object.size(A)
  t_1 <- c(t_1, t[3]/reps)
  s_1 <- c(s_1, s)
}
plot(x * 100, t_1, type="l", xlab="Matrix dimension (n x n)", ylab="time taken/s")
```



```
plot(x * 100, s_1, type="l", xlab="Matrix dimension (n x n)", ylab="Memory usage/Bytes")
```



Numerical Optimisation

Numerical optimisation is a substantial topic that deals with the problem of finding a minimum (or equivalently maximum) point of some objective function over a feasible set, with or without constraints.

One dimensional

The simplest form of optimisation we can carry out is in a single dimension. In R the function `optimize` can carry out one dimensional optimisation and uses the golden section search. It is gradient-free, which is useful but otherwise fairly limited in usefulness.

Multi-dimensional

Most problems are unsurprisingly multidimensional in the input and hence we want our optimisation algorithms to work well in high dimensions. For illustrative purposes, 2D problems are often used however, in order to facillitate a graphical representation of the procedure.

Simplex methods

Simplex methods work by defining a simplex (a polytope) and modifying its shape at each iteration based on the values at its vertices and attempt to shrink around the optimal point after some iterations. The most common such routine is known as the Nelder-Mead algorithm. It is gradient-free, which is a plus, but it can converge to non-stationary points and is not very fast in general.

Gradient methods

Gradient methods use the first order gradient only to pick the direction at each iteration. The simplest method simply goes in the direction of steepest gradient with some (possibly adaptive) stepsize γ, i.e. $x \leftarrow x - \gamma \nabla f(x)$. A more involved first order method is the conjugate gradient method which picks a search direction at each step that is conjugate (i.e. orthogonal w.r.t some matrix A) to previous search directions.

Newton type methods

Newton's method is a root finding approach that uses a Taylor expansion around some point x_0. Since finding a minimum is just equivalent to setting the first derivative to 0, it follows we can use Newton's method to find the root of the gradient and hence find an extremum.

$$\mathbf{g}(\mathbf{x}) \approx \mathbf{g}(\mathbf{x}_0) + (\mathbf{x} - \mathbf{x}_0)^T \nabla \mathbf{g}(\mathbf{x}_0)$$

Now setting $\mathbf{g}(\mathbf{x}) = \nabla f^T$ and setting the LHS to 0:

$$0 = \nabla f(\mathbf{x}_0)^T + (\mathbf{x} - \mathbf{x}_0)^T \nabla\nabla f(\mathbf{x}_0)^T$$

and defining the Hessian as $\mathbf{H} = \nabla\nabla f(\mathbf{x})^T$

$$\begin{aligned} -\nabla f(\mathbf{x}_0) &= \mathbf{H}(\mathbf{x} - \mathbf{x}_0) \\ \mathbf{x} &= \mathbf{x}_0 - \mathbf{H}^{-1} \nabla f(\mathbf{x}_0) \end{aligned}$$

Newton's method then iteratively applies this formula to update x at each step.

R has multiple routines that implement Newton's method which will be tested in the example section below.

First, though, here is an example implementation to solve the logistic regression problem, which has (after some algebra) Hessian $\boldsymbol{H} = -\mathbf{X}^T \boldsymbol{W} \mathbf{X}$ and gradient $\boldsymbol{g} = \mathbf{X}^T(\mathbf{y} - \boldsymbol{\mu})$ where $\mu = \text{logistic}(\mathbf{X}\text{b})$ and $\boldsymbol{W} = \text{diag}\left(\frac{e^x}{(e^x+1)^2}\right)$.

```
# fit a logistic regression model by IRWLS - same thing glm does; basically Newton's method.
library(Matrix)
logistic_reg <- function(X, y) {
    invlink <- logistic
    dinvlink <- function(x)  exp(x)/(1 + exp(x))^2
    loss <- function(y, eta) -(y * eta) + log(1 + exp(eta))
    d <- ncol(X)
    n <- nrow(X)

    b <- matrix(0, d, 1)
    eta <- matrix(0, n, 1)
    mu <- invlink(eta)
    L <- sum(loss(y, eta))

    maxIter <- 10
    tol <- 1e-6

    for (i in 1:maxIter) {
        w <- pmax(dinvlink(eta), 1e-6)
        W <- Diagonal(x = as.numeric(w))
        H <- -t(X) %*% W %*% X
        grad <- t(X) %*% (y - mu)

        b <- b - solve(H, grad)
        eta <- X %*% b
        mu <- invlink(eta)
        L_new <- sum(loss(y, eta))

        if (as.logical(abs(L - L_new) < tol)) {
            print(sprintf("No. iterations: %d", i))
            break
        }
        L <- L_new
    }
    return(b)
}
```

Simulated Annealing

Simulated annealing is a gradient-free method that is guaranteed to find the global minimum (in an infinite time limit). Unfortunately, the amount of time it takes to do this in practice means it is usually not a good choice to solve a problem.

Example

As an example, we'll use the function, the Rastrigin function $f(x, y) = 20 + (x^2 - 10\cos(\frac{3}{4}\pi x)) + (y^2 - 10\cos(\frac{3}{4}\pi y))$. This is highly non-convex with many local optima at regular intervals and one global minimum at the origin with optimal value 0.

```
library(pracma)
```

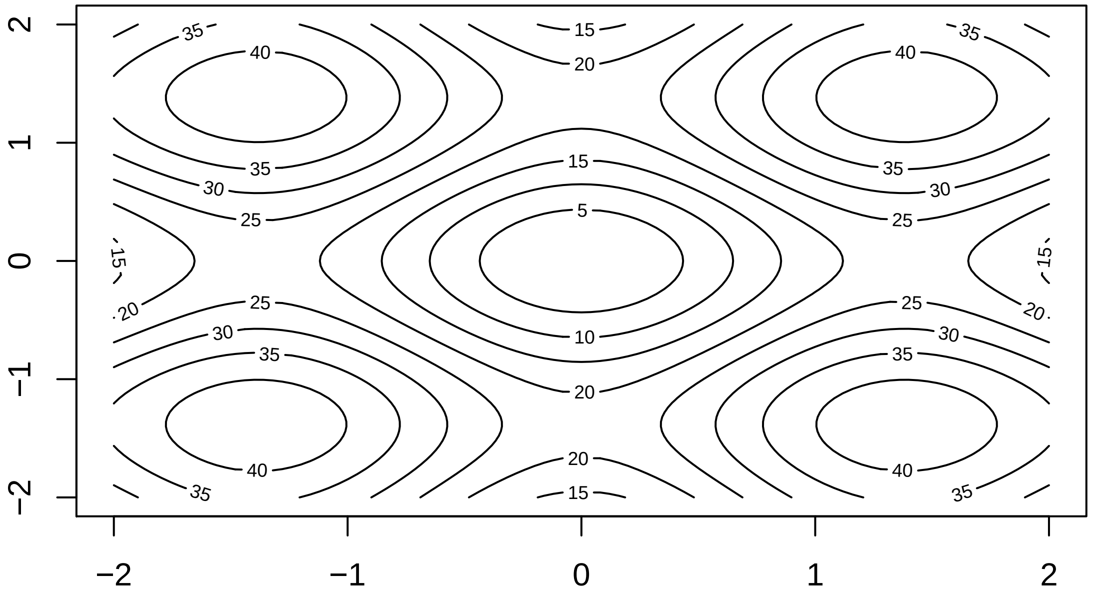
```
##
## Attaching package: 'pracma'

## The following objects are masked from 'package:Matrix':
##
##     expm, lu, tril, triu
```

```
looper <- function (x, n=100, f=function(x) x^2) for (i in 1:n) { f(x) }

f = function (x,y) 20 + (x^2 - 10*cos(0.75*pi*x)) + (y^2 - 10*cos(0.75*pi*y))
```

```r
x = seq(-2, 2, length=101)
grid_xy = meshgrid(x)
z = matrix(mapply(f, grid_xy$X, grid_xy$Y), nrow=101)
min(z)
```

```
## [1] 0
```

```r
contour(x, x, z)
```



```r
g = deriv(expression(20 + (x^2 - 10*cos(0.75*pi*x)) + (y^2 - 10*cos(0.75*pi*y))), namevec = c('x', 'y')
g(0,0)
```

```
## [1] 0
## attr(,"gradient")
##      x y
## [1,] 0 0
## attr(,"hessian")
## , , x
##
##             x y
## [1,] 57.51652 0
##
## , , y
##
##      x        y
## [1,] 0 57.51652
```

nlm

The `nlm` function implements a Newton-type algorithm. Since this relies on derivatives, we can choose to supply expressions for the derivative and hessian of our objective function (using `deriv` as above). This is always preferable to not doing so and relying on the numerical estimates of the derivative and hessian that the function will calculate otherwise.

If we use the origin as an initial guess, `nlm` recognises that this *a* minimum and does nothing:

```r
nlm(function(x) g(x[1], x[2]), c(0,0))
```

```
## $minimum
## [1] 0
##
## $estimate
## [1] 0 0
##
## $gradient
## [1] 0 0
##
## $code
## [1] 1
##
## $iterations
## [1] 0
```

If on the other hand we pick a random point further away, it can still find the global optimum so long as the point is close by. If we pick a point closer to one of the other optima, it will get stuck in them.

```r
nlm(function(x) g(x[1], x[2]), c(0.5,0.5))
```

```
## $minimum
## [1] 0
##
## $estimate
## [1] 2.41722e-19 2.41722e-19
##
## $gradient
## [1] 1.390301e-17 1.390301e-17
##
## $code
## [1] 1
##
## $iterations
## [1] 6
```

```r
system.time(looper(x, 1000, function(x) nlm(function(x) g(x[1], x[2]), c(0.5,0.5), gradtol=1e-8)))
```

```
##    user  system elapsed
##   0.111   0.003   0.114
```

optim

`optim` implements several algorithms, including the default Nelder-Mead, a simplex algorithm.

```r
optim(c(0.5,0.5), function(x) f(x[1], x[2]))
```

```
## $par
## [1] 7.40564e-05 1.22278e-05
##
## $value
## [1] 1.620203e-07
##
## $counts
## function gradient
##       67       NA
```

```
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
system.time(looper(x, 1000, function(x) optim(c(0.5,0.5), function(x) f(x[1], x[2]))))
```

```
##    user  system elapsed
##   0.118   0.002   0.119
```

```
optim(c(0.5, 0.5), function(x) f(x[1], x[2]), method="CG")
```

```
## $par
## [1] 1.061855e-08 1.061855e-08
##
## $value
## [1] 7.105427e-15
##
## $counts
## function gradient
##       50       15
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
system.time(looper(x, 1000, function(x) optim(c(0.5,0.5), function(x) f(x[1], x[2]), method="CG")))
```

```
##    user  system elapsed
##   0.192   0.001   0.194
```

The conjugate gradient method was mentioned earlier and is not as efficient in terms of iterations as a Newton-type method, but does not require any computation of second derivatives which can be helpful and, depending on the dimensionality, and may be a worthwhile trade off vs the number of iterations.

```
optim(c(-1.5, 1.5), function(x) f(x[1], x[2]), method="BFGS")
```

```
## $par
## [1] -2.573213  2.573213
##
## $value
## [1] 13.72575
##
## $counts
## function gradient
##       15        4
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
system.time(looper(x, 1000, function(x) optim(c(0.5,0.5), function(x) f(x[1], x[2]), method="BFGS")))
```

```
##    user  system elapsed
##   0.111   0.000   0.112
```

The BFGS algorithm is the most popular and generally most effective implementation of a quasi-newton method (i.e. one which approximates the inverse of the Hessian without ever actually requiring its direct computation).

```
optim(c(-1.5, 1.5), function(x) f(x[1], x[2]), method="L-BFGS-B")
```

```
## $par
## [1] -2.573212  2.573212
##
## $value
## [1] 13.72575
##
## $counts
## function gradient
##        9        9
##
## $convergence
## [1] 0
##
## $message
## [1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

```
system.time(looper(x, 1000, function(x) optim(c(0.5,0.5), function(x) f(x[1], x[2]), method="L-BFGS-B"))
```

```
##    user  system elapsed
##   0.079   0.001   0.080
```

The limited memory version of the BFGS algorithm runs in half the time of the usual method, reaching the same value (a local minimum).

Note that we can also supply lower and upper bounds on the variables - i.e. box constraints - for the L-BFGS-B algorithm. Also note that the starting point must be feasible.

```
optim(c(-1.5, 1.5), function(x) f(x[1], x[2]),
      method="L-BFGS-B", lower=c(-2, -2), upper=c(3, 3))
```

```
## $par
## [1] -2.000000  2.573212
##
## $value
## [1] 20.86288
##
## $counts
## function gradient
##       10       10
##
## $convergence
## [1] 0
##
## $message
## [1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

`optim` implements simulated annealing as well.

```r
optim(c(-1.5, 1.5), function(x) f(x[1], x[2]), method="SANN", control=list(maxit=1000))
```

```
## $par
## [1] -0.008244195 -0.002610170
##
## $value
## [1] 0.002150475
##
## $counts
## function gradient
##     1000       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```r
system.time(looper(x, 1000, function(x) optim(c(0.5,0.5), function(x) f(x[1], x[2]), method="SANN")))
```

```
##    user  system elapsed
##  16.369   0.081  16.563
```

As expected, simulated annealing runs for much longer than the other algorithms, without converging. It has ended up with a point lower than the immediate local minima found by the other methods, but in a very long time. It is still not the global minimum.

A better approach might be to run the faster algorithms at multiple initial points to try and search more of the space and learn the locality of different minima.

To summarise, using `optim` with L-BFGS-B is probably a sensible "go-to" choice.

Non-linear least squares

If we have a non-linear least squares problem, we can exploit the structure of the squared residual objective function and avoid directly computing the Hessian by instead approximating it with $H \approx 2J^T J$. There are two main algorithms that make use of this approximation; Gauss-Newton and Levenberg-Marquardt. These are implemented by `nls` and `minpack.lm::nlsM` in R. Levenberg-Marquardt is the better of the two algorithms, since it has guarantees on convergence and is robust to singular gradient matricies (due to the inclusion of a damping term to modify the update terms).

Stochastic Gradient Descent (SGD)

Stochastic gradient descent is effectively really a batch method that calculates the gradient over a subset of the dataset and then updates the parameters. As a result, it does not guarantee to reduce the objective at every step (which vanilla gradient descent would). The "stochastic" name results from the convention of picking batches of data at random. It's primary use is in optimizing Neural Networks with very large numbers of data and parameters such that other methods are computationally infeasible (e.g. we cannot fit all of our data in memory simultaneously, let alone optimize over it). In these sorts of environments it's quite effective, although it's worth noting that it is unlikely to actually converge, but will be stopped when out-of-sample performance begins to degrade.

Numerical Integration

Numerical integration is often useful in statistics, particularly in Bayesian methods which frequently involve computations of analytically intractable integrals.

Quadrature

Quadrature is a term that refers to the process of calculating an area - hence its relevance to integration. For a large class of functions, analytical integration is not actually possible, and so we need to resort to numerical methods. A family of algorithms that approximate these integrals involve first approximating the function with an elementary function that can be straightforwardly integrated; a polynomial. As a result we will first consider this procedure which must be carried out prior to the actual integration step.

In R, for one-dimensional integrals, the `integrate` function can be used. For multiple integrals, the package `cubature` is available.

Polynomial interpolation

We want to approximate a continuous function $f \in C^0([a, b])$ in some finite interval by some polynomial p, in order to facilitate integration.

In R, there are a few packages that can carry out function approximation without too much input from the user. For example, `splinefun`.

As an example, say I want to calculate the Area Under the Curve (AUC) of a Receiver Operating Characteristic (ROC) curve, I can use the combination of `splinefun` and `integrate` to get an approximate result:

```r
# calculate an approximate AUC given a set of false positive and true positive rates (for varying thres
compute_AUC <- function(FP, TP) {
    f <- splinefun(FP, TP)
    AUC <- integrate(f, 0, 1)
    return(AUC)
}
```

Weierstrass Approximation Theorem Given the setup above, this theorem states that there exists a sequence of polynomials (p_n) that converges uniformly to f on $[a, b]$;

$$\|f - p_n\|_\infty = \max_{x \in [a,b]} |f(x) - p_n(x)| \to 0.$$

In practice this doesn't provide enough information to really address the problem of approximating a given function with a polynomial, but it at least implies that the task is achievable.

Lagrange polynomials Lagrange polynomials are defined by

$$p_{k-1}(x) = \sum_{i=1}^{k} l_i(x) f(x_i)$$

with Lagrange basis polynomials $l_i(x) = \prod_{j=1, j\neq i}^{k} \frac{x - x_i}{x_i - x_j}$ for $i \in \{1, \dots, k\}$. The idea of the method is to use k points and essentially fit a degree $k-1$ curve between these points. Intuitively this form of the polynomial makes sense in this context, rather than the usual weighted sum of monomial form $p(x) = \sum_{i=1}^{k} a_i x^{i-1}$ for which we would also in order to solve a set of linear equations in order to compute the weights. Solving this linear system can also be numerically unstable for large k.

Polynomial interpolation error When we try to approximate some function with a polynomial, apart from polynomials of the same degree as the interpolating polynomial, we will inevitably have some error. The **Polynomial Interpolation Theorem** quantifies this error as

$$f(x) - p_{k-1}(x) = \frac{1}{k!} f^{(k)}(\xi) \prod_{i=1}^{k} (x - x_i).$$

where f is taken to be a C^k-smooth function on an interval $[a, b]$, p_{k-1} is the polynomial interpolation at points $x_1, \dots, x_k$ and $\xi \in (a, b)$ is a point guaranteed to exist that satisfies the above relation. Effectively this says that for any point $x \in [a, b]$, the error at that point is related to the k^{th} derivative of f at some point ξ and the distance between x and each of the interpolating points.

Theorem For an $f \in C^0([a, b])$ there exists a sequence of sets of interpolation points $X_1, X_2, \dots$ such that the corresponding sequence of interpolating polynomials converges uniformly to f on $[a, b]$. Unfortunately this theorem *does not* mean we can find some universal sequence of sets of interpolation points and retain uniform convergence. In fact, for any fixed sequence of such sets a function f exists for which the sequence of interpolating polynomials actually *diverges*. Fortunately, we can pick points that work *most* of the time. The *Chebyshev* points ($\cos(\frac{2j-1}{2k}\pi)$ for $j \in \{1, \dots, k\}$) minimise the maximum absolute value of the product term in the interpolation error and this seems to be effective for most functions.

Piece-wise polynomials In order to approximate complicated functions, we have two choices within polynomial interpolation. The first, which we have already seen, is to increase the degree of the polynomial. The second is to instead use a piece-wise polynomial function. To do this, we need to subdivide our interval $[a, b]$ into some number of subintervals and then estimate polynomials for each of these. If we want our resultant piece-wise function to be continuous then we need to ensure that the end points of each inner subinterval are included as interpolating points. As the number of subintervals increases, we can expect to be able to use smaller degree polynomials in them, since as we analyse our function at smaller and smaller scales, we expect it to look smoother and smoother and eventually be well approximated by a linear function.

Other polynomial schemes

- Hermite interpolation: A polynomial interpolation scheme that includes derivatives of f.
- Spline interpolation: A piece-wise polynomial interpolation scheme that matches the derivatives of the polynomials at the boundaries of their subintervals to provide a given number of continuous derivatives of the approximation function.
- Non-linear models: e.g. Neural Networks. Such models may well be difficult to integrate which defeats the point here.

Polynomial integration

We want to compute of some function $f \in C^0([a, b])$

$$I(f) = \int_a^b f(x)dx$$

Clearly for a poly omial approximation the integrals will be straightforward #### Changing the limits of integration To change the limits of a definite integral such that

$$\int_a^b f(x)dx = \int_c^d g(y)dy$$

for finite intervals $[a, b]$ and $[c, d]$ we simply define

$$g(y) = \frac{b-a}{d-c} f\left(a + \frac{b-a}{d-c}(y-c)\right)$$

which means that if we can find a suitable approximation for a given interval and function, we can transform it to any other interval.

For transition to and from (semi-)infinite intervals we can do a similar procedure, e.g.

$$\int_{-\infty}^{\infty} f(x)dx = \int_{-1}^{1} g(y)dy, \qquad g(y) = \frac{1+y^2}{1-y^2} f\left(\frac{y}{1-y^2}\right)$$

Newton-Cotes rules

- Rectangular rule (closed with $k = 1$) $\hat{I}_{\text{rectangular}}(f) = (b-a)f(a)$
- Midpoint rule (closed with $k = 1$) $\hat{I}_{\text{midpoint}}(f) = (b-a)f\left(\frac{1}{2}(a+b)\right)$
- Trapezoidal rule (cllosed with $k = 2$) $\hat{I}_{\text{trapezoidal}}(f) = \frac{1}{2}(b-a)(f(a)+f(b))$
- Simpson's rule (closed with $k = 3$) $\hat{I}_{\text{simpson}}(f) = \frac{1}{6}(b-a)(f(a)+4f(\frac{1}{2}(a+b))+f(b))$

Theorem Let $f \in C^k([a,b])$ then the integration error for interpolation points $x_1, \dots, x_k$ satisfies

$$|\hat{I}(f) - \hat{I}(f)| \le \max_{\xi \in [a,b]} |f^{(k)(\xi)}| \frac{(b-a)^{k+1}}{k!}$$

Composite rules For piece-wise polynomial approxmiations, we can calculate the integral by summing the approximate integrals over each subinterval. As such, we can apply of the Newton-Cotes rules to each subinterval.

Gaussian quadrature When we want to calculate the integral of some unknown function f, instead of directly estimating an approximation of f and then integrating, with Gaussian quadrature we instead aim to approximate the integral itself, without first estimating f. This means that the underlying function we integrate may be a poor approximator of f but we don't mind so long as their integrals are close.

This method provides an exact result for degree $2n-1$ or less polynomials, where the fundamental theorem of Gaussian quadrature states that for such a polynomial, the optimal points to evaluate the function f at, with some weight function $w(x)$, are exactly the roots of an orthogonal polynomial over the same interval as the integral.

Multiple integrals These quadrature methods tend to be highly efficient at approximating smooth one dimensional functions over finite intervals. For multi-dimensional integrals they can be much less efficient however and can suffer from the "curse of dimensionality".The most obvious method to attempt is to try and compute the multi-dimensional integration by iteratively integrating over each variable at a time, by making use of Fubini's Theorem. The efficiency of quadrature methods in high dimensional spaces is closely related to their smoothness and it turns out that for at least all functions $f \in C^r([0,1]^d)$ (with $r \in \mathbb{N}$ and all partial derivatives bounded by 1) suffer from the curse of dimensionality.

Monte Carlo Methods

Monte Carlo methods are useful for classes of functions that don't satisfy the smoothness requirements of high-efficiency quadrature algorithms or for high-dimensional integrals. They make use of random samples to make computations (as indicated by the name).

Theorem: Strong Law of Large Numbers (SLLN) Let $(X_n)_{n\ge1}$ be a sequence of i.i.d RVs distributed according to some measure μ, then

$$S_n(f) = \sum_{i=1}^{n} f(X_i) \lim_{n\to\infty} \frac{1}{n} S_n(f) = \mu(f)$$

$S_n(f)$ is a Monte Carlo approximation of $\mu(f)$.

Rejection sampling

To carry out rejection sampling, we do the following procedure: 1. Sample $X \sim \mu$ 2. With probability $\frac{1}{M}\frac{\pi(X)}{\mu(X)}$ output X otherwise go back to 1.

where M is the bound on $\sup_{x\in\mathcal{X}} \frac{\pi(x)}{\mu(x)}$

Importance sampling

Importance sampling asserts that $\frac{1}{n}S_n(f \cdot w)$ as an approximation for $\pi(f)$ where $w(x) = \frac{\pi(x)}{\mu(x)}$ based on an integral expression for $\pi(f)$:

$$\pi(f) = \int_{\mathcal{X}} f(x)\pi(dx) = \int_{\mathcal{X}} f(x)w(x)\mu(dx) = \mu(f \cdot w)$$

This method is useful when we do not know how to sample from π. ### Self-normalised importance sampling If we cannot calculate $w(x)$ exactly, then importance sampling will not work, in which case we can try a self-normalized approximation:

$$\frac{S_n(f \cdot w)}{S_n(w)} = \frac{\sum_i w(X_i)f(X_i)}{\sum_i w(X_i)}.$$

If $\pi(x) > 0 => \mu(x) > 0$ then this approximation should converge to $\pi(f)$ almost surely.

Markov Chains

A Markov Chain is a model that describes a sequence of states where the probability of entering the next state X_n is conditional only on the preceeding state X_{n-1} (the Markov property). In order to summarise the probability of moving from one state to another at any step of the chain we have a transition kernel, with elements P_{ij} corresponding to the probability of transitioning from state $i \to j$. This kernel encodes the properties of the Markov chain.

Properties of P:

- $\sum_{j \in S} P_{ij} = 1$
- $\pi P = \pi$ if P is π-invariant
- $P = \sum_{s \in S} w(s)P_s$
- $p(X_n \in A \mid X_{n-1} = x) = p(X_1 \in A \mid X_0 = x)$ is the MC $\mathbf{X}$ is *time-homogeneous.*
- P is *Harris recurrent* if the probability of returning to a region of the state-space in an infinite number of steps is unbounded.
- A Markov Chain is π-reversible if $\pi_i P_{ij} = \pi_j P_{ji}$.

MCMC

The general idea of MCMC is that one constructs a sampling mechanism whose sequence of samples forms a Markov chain and the stationary distribution of this Markov chain is the posterior distribution being estimated.

Metropolis-Hastings (MH) Proposal Markov kernel Q Algorithm: 1. Simulate $Z \sim Q$ 2. With probability α output Z, otherwise output x where $\alpha = \min\left(1, \frac{\pi(z)q(z,x)}{\pi(x)q(x,z)}\right)$

An example implementation from the notes:

```r
make.metropolis.hastings.kernel <- function(pi, Q) {
  q <- Q$density
  P <- function(x) {
    z <- Q$sample(x)
    alpha <- min(1, pi(z)*q(z,x)/pi(x)/q(x,z))
    ifelse(runif(1) < alpha, z, x)
  }
  return(P)
}
```

```r
# univariate normal proposal
make.normal.proposal <- function(sigma) {
  Q <- list()
  Q$sample <- function(x) {
    x + sigma*rnorm(1)
  }
  Q$density <- function(x,y) {
    dnorm(y-x, sd=sigma)
  }
  return(Q)
}

# simulate a Markov chain of length n of one-dimensional points
# initial point is x0, P simulates according to Markov kernel
simulate.chain <- function(P, x0, n) {
  xs <- rep(0, n)
  x <- x0
  for (i in 1:n) {
    x <- P(x)
    xs[i] <- x
  }
  return(xs)
}
```

Gibbs Gibbs sampling in its basic formulation is a special case of MH. It's extended formulation is more general and samples each of a set of variables in turn (possibly utilising MH for each variable) Algorithm: 1. Simulate $Z \sim Q$ 2. Sample each coordinate in turn conditioned on the other coordinates. e.g. given a model $y_i \sim \mu + \alpha_i + \beta_i$, do

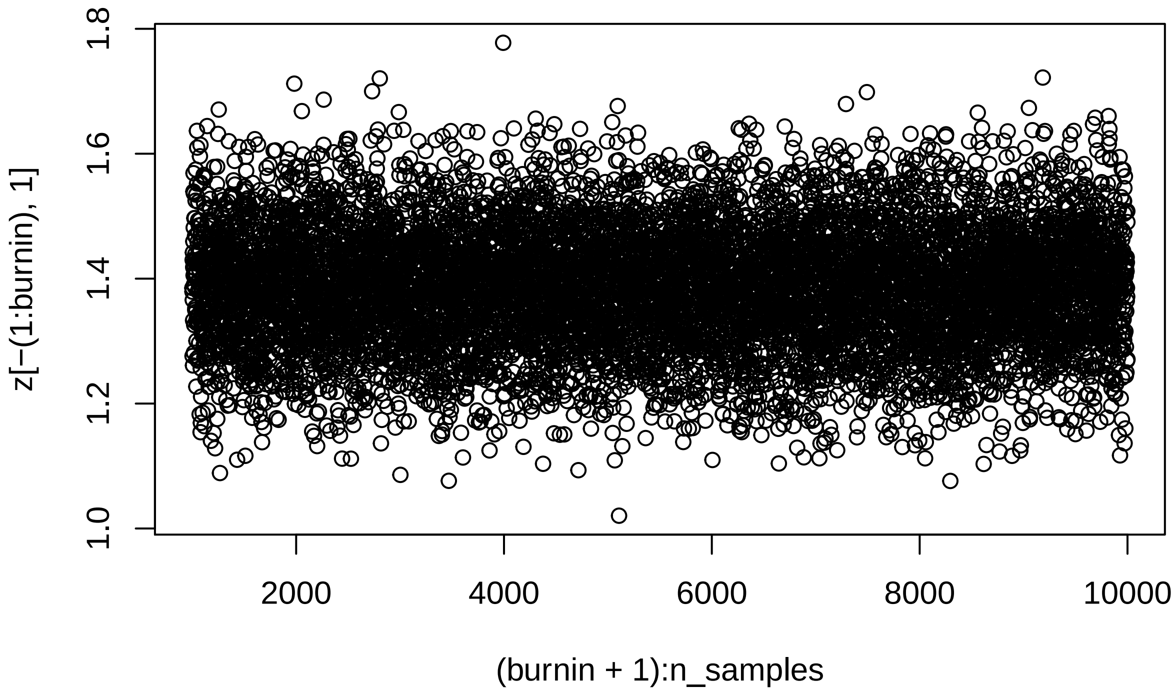
- $\mu^{(t+1)} \mid (\boldsymbol{\alpha}^{(t)}, \boldsymbol{\beta}^{(t)}, \mathbf{y})$
- $\alpha_i^{(t+1)} \mid (\mu^{(t+1)}, \boldsymbol{\beta}^{(t)}, \mathbf{y})$
- $\beta_i^{(t+1)} \mid (\mu^{(t+1)}, \boldsymbol{\alpha}^{(t)}, \mathbf{y})$

As an example, assume a model of the form $y \sim N(\mu, \sigma)$ with $\mu \sim N(\mu_0, \sigma_0)$ and $\frac{1}{\sigma^2} = \tau \sim \mathrm{Gamma(a, b)}$. Since these priors are conjugates, we can write out the posteriors directly, computing the posterior for μ first, holding τ constant and then the posterior for τ holding μ constant:

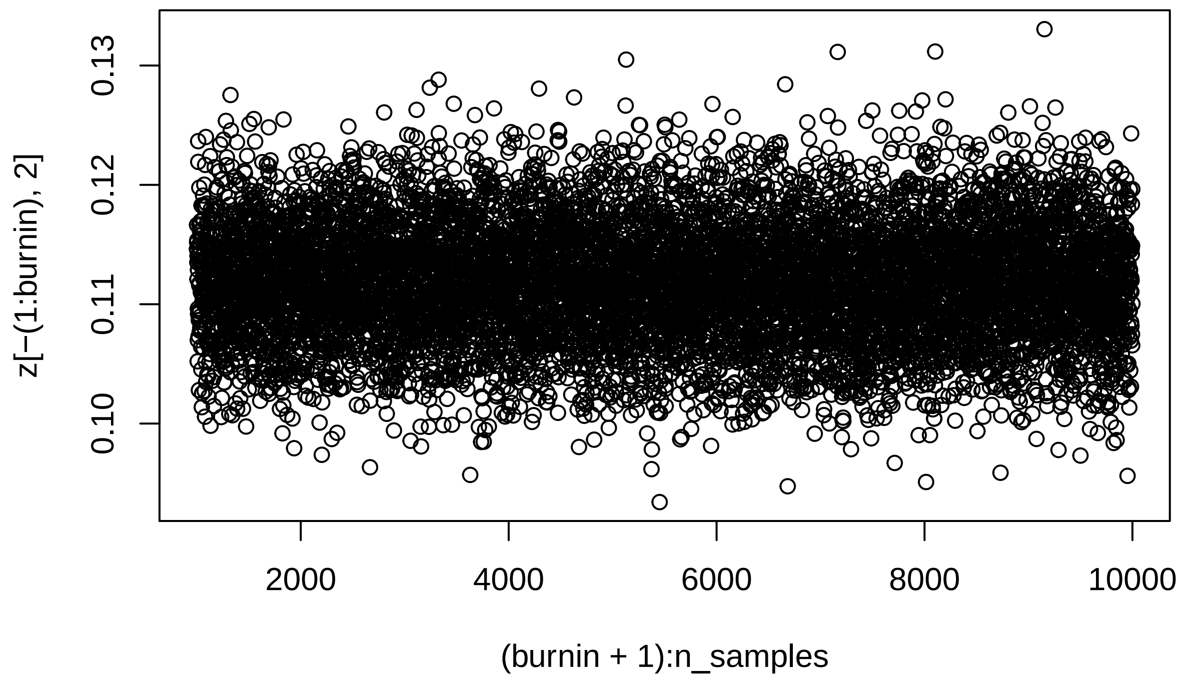
$$\begin{aligned}
N_\mu(\mu_0, \sigma_0) &\to N_\mu(\mu_*, \sigma_*) \\
\mu_* &= \frac{\tau_0 \mu_0 + \tau \sum_i^n x_i}{\tau_0 + n\tau}, \quad \tau_* = \tau_0 + n\tau \\
\mathrm{Gamma(a_0, b_0)} &\to \mathrm{Gamma(a_*, b_*)} \\
a_* &= a + \frac{n}{2} \\
b_* &= b + \frac{1}{2} \sum_i (x_i - \mu)^2 \\
&= b + \frac{1}{2}(n\hat{\sigma}^2 + n\bar{x}^2) - n\bar{x}\mu + \frac{1}{2} n\mu^2 \\
&= b + \frac{1}{2} n\hat{\sigma}^2 + \frac{1}{2} n(\bar{x} - \mu)^2
\end{aligned}$$

where $\bar{x}, \hat{\sigma}^2$ are the sample mean and variance respectively.

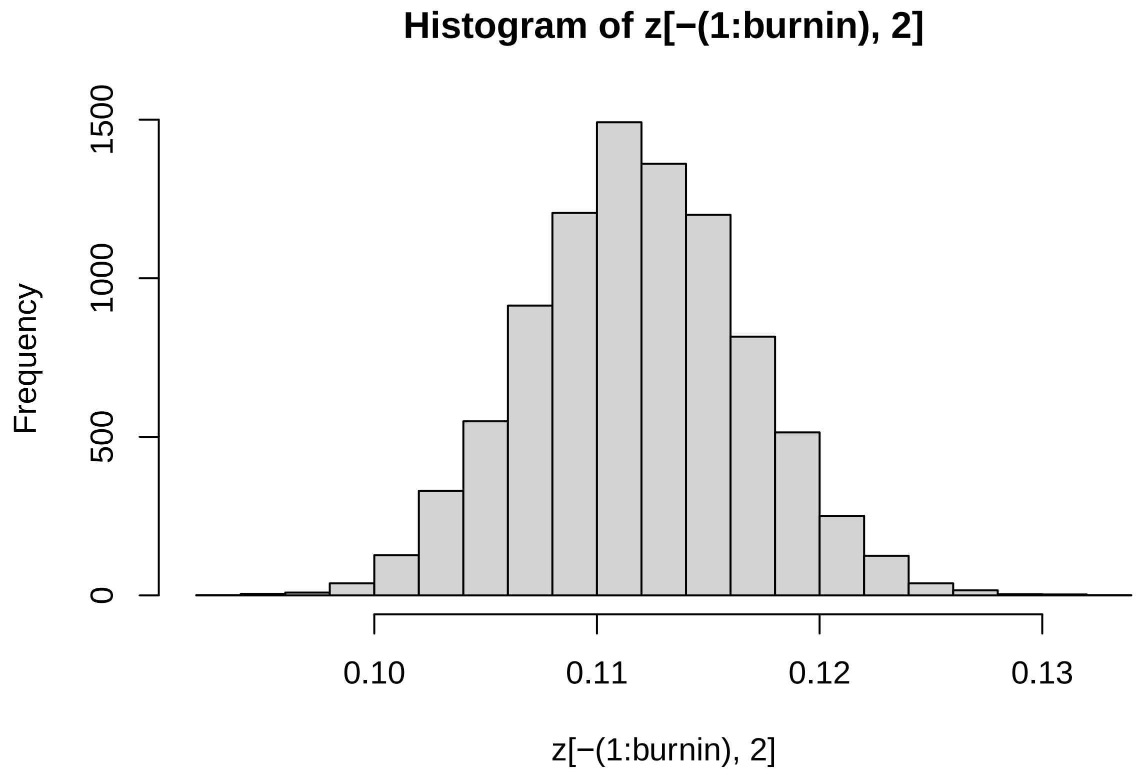
```r
set.seed(1234)
gibbs_sample <- function(init, m, data){
  n <- length(data)
  s <- sum(data)
  v <- var(data)

  out <- matrix(NA, nrow = m, ncol = 2)
  out[1,] <- init[1:2]
  for (i in 2:m){
    # update prior over mu
    out[i,1] <- rnorm(1, mean = (prod(init[1:2])  + out[i-1,2]*s)/(init[2] + n*out[i-1,2]), sd = 1/sqrt
    # update prior over tau
    out[i,2] <- rgamma(1, shape = init[3] + n/2, rate = init[4] + 0.5*n*v + 0.5*n*(s/n - out[i,1])^2)
  }
  return(out)
}
n_samples <- 10000
burnin <- 1000
# generate some data
x <- rnorm(1000, 1.5, 3.0)
# sample
z <- gibbs_sample(c(-0.5, 2.0, 1.0, 1.5), n_samples, x)
# plot traces
plot((burnin+1):n_samples, z[-(1:burnin),1])
```



```r
plot((burnin+1):n_samples, z[-(1:burnin),2])
```



```
# plot parameter
hist(z[-(1:burnin),2])
```



```
# check if the posterior samples reasonably approximate the parameters we used to generate the data
stopifnot(abs(mean(z[-(1:burnin),1]) - mean(x)) < 1e-1)
stopifnot(abs(1/mean(z[-(1:burnin),2]) - var(x)) < 1e-1)
```

An R implementation of Gibbs sampling used primarily for posterior estimation in Bayesian inference is the `rjags` package, where JAGS stands for "Just Another Gibbs Sampler".

To use it we need to define a model in the BUGS language:

```
model {
  for (i in 1:N) {
    Y[i] ~ dnorm(mu[i], tau)
    mu[i] <- alpha + beta * (x[i] - x.bar)
  }
  x.bar <- mean(x)
  alpha ~ dnorm(0.0, 1.0E-4)
  beta ~ dnorm(0.0, 1.0E-4)
  sigma <- 1.0/sqrt(tau)
  tau ~ dgamma(1.0E-3, 1.0E-3)
}
```

Then we can run this model in JAGS to estimate the parameters

```r
library(rjags)
```

```
## Loading required package: coda
```

```
## Linked to JAGS 4.3.0
```

```
## Loaded modules: basemod,bugs
```

```r
line_data <- list("x" = c(1, 2, 3, 4, 5), "Y" = c(1, 3, 3, 3, 5), "N"=5)
line_inits <- list(list("alpha" = 3, "beta"= 0, "tau" = 1.5),
  list("alpha" = 0, "beta" = 1, "tau" = 0.375))

model <- jags.model("line.bug", data=line_data, inits=line_inits, n.chains=2)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 5
##    Unobserved stochastic nodes: 3
##    Total graph size: 37
##
## Initializing model
```

```r
update(model, n.iter=1000)
samples <- coda.samples(model, variable.names=c("alpha", "beta", "sigma"),
                        n.iter=1000)

summary(samples)
```

```
##
## Iterations = 1001:2000
## Thinning interval = 1
## Number of chains = 2
## Sample size per chain = 1000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##         Mean     SD Naive SE Time-series SE
## alpha 3.0034 0.5306  0.01186       0.012256
## beta  0.7939 0.3752  0.00839       0.007774
## sigma 1.0063 0.6657  0.01489       0.025100
##
## 2. Quantiles for each variable:
##
##          2.5%    25%    50%    75% 97.5%
## alpha 1.90942 2.7568 3.0238 3.2785 3.994
## beta  0.06964 0.6229 0.7923 0.9694 1.520
## sigma 0.42666 0.6279 0.8185 1.1692 2.630
```