# APTS 2019/20
# Statistical Computing

Notes originally by Simon Wood, University of Bath, 2012
https://people.maths.bris.ac.uk/~sw15190/

Updated by Darren Wilkinson, Newcastle University, 2018, 2019
https://darrenjw.github.io/

On-line resources: https://www.staff.ncl.ac.uk/d.j.wilkinson/teaching/apts-sc/

Last updated: December 21, 2019

# Contents

# Chapter 1

# Introduction

In combination, modern computer hardware, algorithms and numerical methods are extraordinarily impressive. Take a moment to consider some simple examples, all using R.

1. Simulate a million random numbers and then sort them into ascending order.

```
x <- runif(1000000)
system.time(xs <- sort(x))

##    user  system elapsed
##   0.107   0.015   0.124
```

   The sorting operation took 0.1 seconds on the laptop used to prepare these notes. If each of the numbers in `x` had been written on its own playing card, the resulting stack of cards would have been 40 metres high. Imagine trying to sort `x` without computers and clever algorithms.

2. Here is a matrix calculation of comparatively modest dimension, by modern statistical standards.

```
set.seed(2)
n <- 1000
A <- matrix(runif(n*n),n,n)
system.time(Ai <- solve(A))  # invert A

##    user  system elapsed
##   0.282   0.060   0.108

range(Ai%*%A-diag(n))        # check accuracy of result

## [1] -1.222134e-12  1.292300e-12
```

   The matrix inversion took under half a second to perform, and the inverse satisfied its defining equations to around one part in $10^{11}$. It involved around 2 billion ($2 \times 10^9$) basic arithmetic calculations (additions, subtractions multiplications or divisions). At a very optimistic 10 seconds per calculation this would take 86 average working lives to perform without a computer (and that's assuming we use the same algorithm as the computer, rather than Cramer's rule, and made no mistakes). Checking the result by multiplying the `A` by the resulting inverse would take another 86 working lives.

3. The preceding two examples are problems in which smaller versions of the same problem can be solved more or less exactly without computers and special algorithms or numerical methods. In these cases, computers allow much more useful sizes of problem to be solved very quickly. Not all problems are like this, and some problems can really only be solved by computer. For example, here is a scaled

version of a simple delay differential equation model of adult population size, $n$, in a laboratory blowfly culture.

$$\frac{\mathrm{d}n}{\mathrm{d}t} = PT_D n(t-1)e^{-n(t-1)} - \delta T_D n(t).$$

The dynamic behaviour of the solutions to this model is governed by parameter groups $PT_D$ and $\delta T_D$, but the equation can not be solved analytically, from any interesting set of initial conditions. To see what solutions actually look like we must use numerical methods.

```
install.packages("PBSddesolve") ## install package from CRAN
```

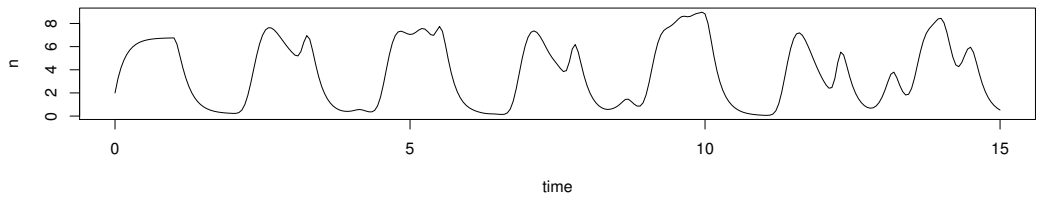
```
library(PBSddesolve)

bfly <- function(t,y,parms) {
## function defining rate of change of blowfly population, y, at
## time t, given y, t, lagged values of y form 'pastvalue' and
## the parameters in 'parms'. See 'ddesolve' docs for more details.

## making names more useful ...
  P <- parms$P;d <- parms$delta;y0 <- parms$yinit

  if (t<=1) y1 <- P*y0*exp(-y0)-d*y else { ## initial phase
    ylag <- pastvalue(t-1)          ## getting lagged y
    y1 <- P*ylag*exp(-ylag) - d*y   ## the gradient
  }
  y1  ## return gradient
}

## set parameters...
parms <- list(P=150,delta=6,yinit=2.001)
## solve model...
yout <- dde(y=parms$yinit,times=seq(0,15,0.05),func=bfly,parms=parms)
## plot result...
plot(yout$t,yout$y,type="l",ylab="n",xlab="time")
```




Here the solution of the model (to an accuracy of 1 part in $10^8$), took a quarter of a second.

## 1.1 Things can go wrong

The amazing combination of speed and accuracy apparent in many computer algorithms and numerical methods can lead to a false sense of security and the wrong impression that in translating mathematical and statistical ideas into computer code we can simply treat the numerical methods and algorithms we use as infallible 'black boxes'. This is not the case, as the following three simple examples should illustrate.

1. The `system.time` function in R measures how long a particular operation takes to execute. First generate two $3000 \times 3000$ matrices, $\mathbf{A}$ and $\mathbf{B}$ and a vector, $\mathbf{y}$.

```
n <- 3000
A <- matrix(runif(n*n),n,n)
B <- matrix(runif(n*n),n,n)
y <- runif(n)
```

Now form the product $\mathbf{ABy}$ in two slightly different ways

```
system.time(f0 <- A%*%B%*%y)

##    user  system elapsed
##   2.975   0.247   0.919

system.time(f1 <- A%*%(B%*%y))

##    user  system elapsed
##   0.105   0.023   0.033
```

`f0` and `f1` are identical, to machine precision, so why is one calculation so much quicker than the other? Clearly anyone wanting to compute with matrices had better know the answer to this.

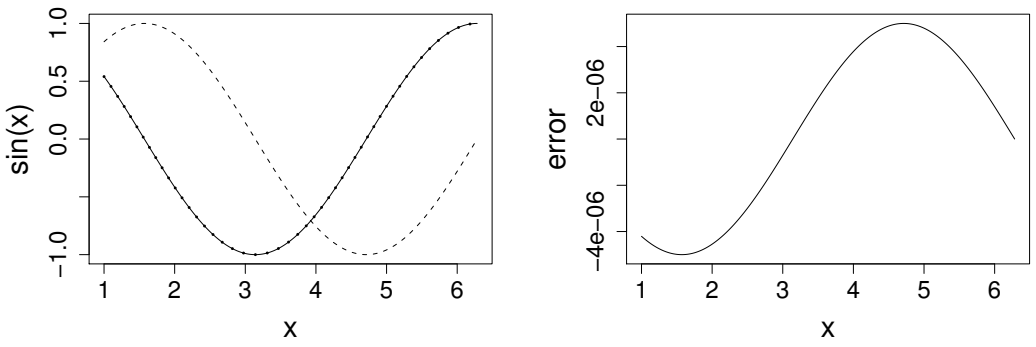
2. It is not unusual to require derivatives of complicated functions, for example when estimating models. Such derivatives can be very tedious or difficult to calculate directly, but can be approximated quite easily by 'finite differencing'. i.e. if $f$ is a sufficiently smooth function of $x$ then

$$\frac{\partial f}{\partial x} \simeq \frac{f(x+\Delta) - f(x)}{\Delta}$$

where $\Delta$ is a small interval. Taylor's theorem implies that the left hand side tends to the right hand side, above, as $\Delta \to 0$. It is easy to try this out on an example where the answer is known, so consider differentiating $\cos(x)$ w.r.t. x.

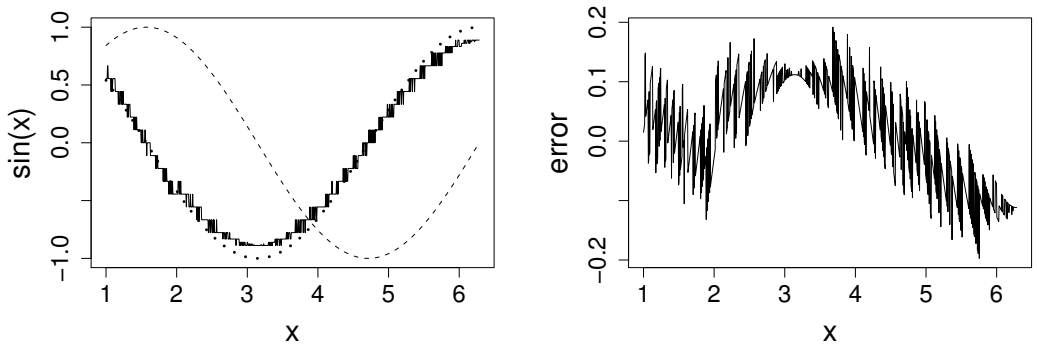
```
x <- seq(1,2*pi,length=1000)
delta <- 1e-5       # FD interval
dsin.dx <- (sin(x+delta)-sin(x))/delta # FD derivative
error <- dsin.dx-cos(x)      # error in FD derivative
```

The following plots the results. On the LHS $\sin(x)$ is dashed, it's true derivative $\cos(x)$ is dotted and the FD derivative is continuous. The RHS shows the difference between the FD approximation and the truth.



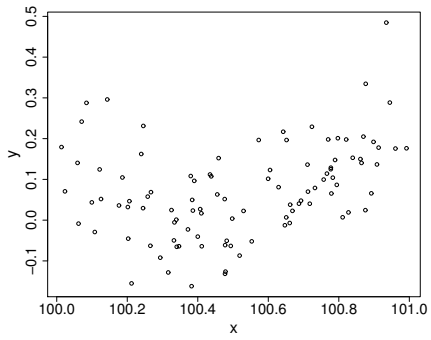
Note that the error appears to be of about the same magnitude as the finite difference interval $\Delta$. Perhaps we could reduce the error to around the machine precision by using a much smaller $\Delta$. Here is the equivalent of the previous plot, when

```
delta <- 1e-15
```




Clearly there is something wrong with the idea that decreasing $\Delta$ must always increase accuracy here.

3. These data



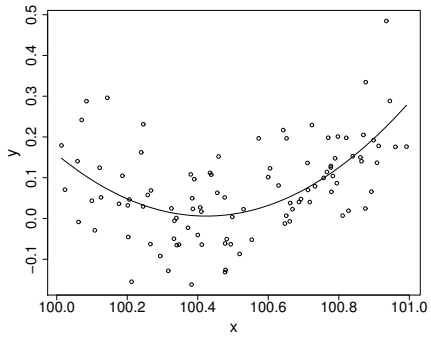

were generated with this code. . .

```
set.seed(1); n <- 100
xx <- sort(runif(n))
y <- .2*(xx-.5)+(xx-.5)^2 + rnorm(n)*.1
x <- xx+100
```

An appropriate and innocuous linear model for the data is $y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \epsilon_i$, where the $\epsilon_i$ are independent, zero mean constant variance r.v.s and the $\beta_j$ are parameters. This can be fitted in R with

```
b <- lm(y~x+I(x^2))
```

which results in the following fitted curve.



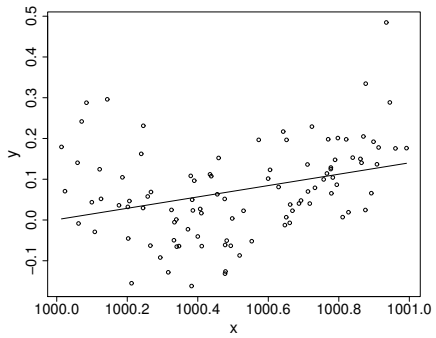

From basic statistical modelling courses, recall that the least squares parameter estimates for a linear model are given by $\hat{\boldsymbol{\beta}} = (\mathbf{X}^{\mathsf{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathsf{T}}\mathbf{y}$ where $\mathbf{X}$ is the model matrix. An alternative to use of `lm` would be to use this expression for $\hat{\boldsymbol{\beta}}$ directly.

```
X <- model.matrix(b)     # get same model matrix used above
beta.hat <- solve(t(X)%*%X,t(X)%*%y) # direct solution of 'normal equations'

## Error in solve.default(t(X) %*% X, t(X) %*% y):  system is computationally
singular:  reciprocal condition number = 3.98647e-19
```

Something has gone badly wrong here, and this is a rather straightforward model. It's important to understand what has happened before trying to write code to apply more sophisticated approaches to more complicated models.

In case you are wondering, I can make `lm` fail to. Mathematically it is easy to show that the model fitted values should be unchanged if I simply add 900 to the `x` values, but if I refit the model to such shifted data using `lm`, then the following fitted values result, which can not be correct.



# Chapter 2

# Matrix Computation

Matrices are ubiquitous in statistics. But most statistical models and methods are only useful once turned into computer code, and this often means computing with matrices. It is easy to ruin a beautiful piece of theory by poor use of numerical linear algebra. This section covers some basic numerical linear algebra, including the important topics of efficiency and stability.

## 2.1 Efficiency in matrix computation

Recall the simple example

```
n <- 3000
A <- matrix(runif(n*n),n,n)
B <- matrix(runif(n*n),n,n)
y <- runif(n)
system.time(f0 <- A%*%B%*%y)     # case 1

##    user  system elapsed
##   2.816   0.173   0.819

system.time(f1 <- A%*%(B%*%y))   # case 2

##    user  system elapsed
##   0.083   0.041   0.033
```

What happened here? The answer is all to do with how the multiplications were ordered in the two cases, and the number of *floating point operations* (flop) required by the two alternative orderings.

1. In the first case $\mathbf{AB}$ was formed first, and the resulting matrix used to pre-multiply the vector $\mathbf{y}$.
2. In the second case, the vector $\mathbf{By}$ was formed first, and was then pre-multiplied by $\mathbf{A}$.

Now we need to count the floating point operations. Try a couple of simple examples first

1. How many flop (+, -, *, /) are involved in the multiplication

$$\begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}?$$

2. How many operations are involved in the multiplication

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}?$$

Generalising

1. In terms of $n$, what is the flop cost of forming $\mathbf{AB}$?
2. What is the cost of forming $\mathbf{By}$

Now write down the flop cost of the two ways of forming $\mathbf{ABy}$

1.
2.

Another simple matrix operation is the evaluation of the trace of a matrix product. Again different ways of computing the same quantity lead to radically different computation times. Consider the computation of $\mathrm{tr}\,(\mathbf{AB})$ where $\mathbf{A}$ is $5000 \times 1000$ and $\mathbf{B}$ is $1000 \times 5000$.

```
n <- 5000; m <- 1000
A <- matrix(runif(n*m), n, m)
B <- matrix(runif(n*m), m, n)
system.time(sum(diag(A%*%B)))

##    user  system elapsed
##   2.858   0.262   0.852

system.time(sum(diag(B%*%A)))

##    user  system elapsed
##   0.559   0.021   0.150

system.time(sum(A*t(B)))

##    user  system elapsed
##   0.165   0.115   0.096
```

1. The first method forms $\mathbf{AB}$, at a flop cost of $2n^2m$ and then extracts the leading diagonal and sums it.
2. The second method uses the fact that $\mathrm{tr}\,(\mathbf{AB}) = \mathrm{tr}\,(\mathbf{BA})$. It forms $\mathbf{BA}$ at flop cost of $2nm^2$ and then extracts the leading diagonal of the result and sums it.
3. The third method makes direct use of the fact that $\mathrm{tr}\,(\mathbf{AB}) = \sum_{ij} A_{ij}B_{ji}$, at a cost of $2nm$. It is fastest as no effort is wasted in calculating off diagonal elements of the target matrix (but here does come at the cost of computing a transpose).

Notice that method 1 is not just wasteful of flops, it also requires storage of an $n \times n$ matrix, which is much larger than either $\mathbf{A}$ or $\mathbf{B}$.

### 2.1.1 Flop counting in general

Clearly it is important to count flops, but it is also tedious to do a complete count for complex algorithms. In many cases an exact count is not necessary. Instead we simply need to know how the computation scales with problem size, as it becomes large. For this reason the *leading order* operation count is all that is usually reported. For example the naive computation of $\mathrm{tr}\,(\mathbf{AB})$, above, required at least $n^2m + n^2(m-1) + n - 1$ operations. As $n$ and $m$ tend to infinity this is dominated by a term proportional to $n^2m$, so we write that the computational cost is $O(n^2m)$ ('order $n^2m$').

Note also that flops are not always defined in the same way. For some authors one flop is one floating point addition, subtraction, division or multiplication, but for others one flop is one multiplication/division with one addition/subtraction.

### 2.1.2 There is more to efficiency than flop counting

Flop count is clearly important, but it is not the only aspect of efficiency. With current computing technology (driven by the computer games market) a floating point operation often takes the same amount of time as an operation on an integer or a memory address. Hence how matrices are stored and addressed in memory is often as important as the flop count in determining efficiency.

As an example consider the multiplication of the elements of two matrices as part of a matrix multiplication performed in C. The code

```
C[i][j] += A[i][k]*B[k][j]
```

requires at least 6 address/integer calculations in order to figure out where the relevant elements of `A`, `B` and `C` are located, before the 2 floating point operations can be performed. To avoid this, efficient code stores pointers which keep a record of the memory locations of the 'current' relevant elements of `A`, `B` and `C`. The multiplication loops are then structured so as to minimise the work needed to update such pointers, for each new element multiplication.

To further complicate matters, computer processors often contain features that can be exploited to increase speed. For example, fetching data from the processor memory cache is much quicker than fetching data from general memory (not to be confused with the yet slower disk). An algorithm structured to break a problem into chunks that fit entirely into cache can therefore be very quick. Similarly, some processors can execute multiple floating point operations simultaneously, but careful coding is required to use such a feature.

To avoid having to re-write whole matrix algebra libraries to maximise speed on every new processor that comes along, standard numerical linear algebra libraries, such as LAPACK (used by R), adopt a highly structured design. Wherever possible, the high level routines in the library (e.g. for finding eigenvalues), are written so that most of the flop intensive work is performed by calling a relatively small set of routines for comparatively simple basic matrix operations (e.g. multiplying a matrix by a vector), known as the Basic Linear Algebra Subprograms (BLAS). The BLAS can then be tuned to make use of particular features of particular computers, without the library that depends on it needing to change. This provides one way of exploiting multi-core computers. Multi-threaded BLAS can gain efficiency by using multiple CPU cores, without any of the higher level code having to change.

Whatever language or library you are using for statistical computing, you should be using native BLAS and LAPACK libraries "under–the–hood" for your matrix computations. But it is important to be aware that not all BLAS and LAPACK libraries are the same, and "default" versions shipping with Linux distributions and similar operating systems are not always the best. You can very often swap out a default BLAS library and replace it with an optimised BLAS for your system without changing or recompiling any code. This can easily make a factor of 2 or more difference to the running time of codes for which matrix computation is a bottleneck, so it is well worth investigating how to do this for an easy way of speeding up your code.

### 2.1.3 Efficiency conclusions

In conclusion:

- Think about the flop count when computing with matrices (or computing in general).
- If you are writing compiled code for manipulating matrices (or other arrays), then be aware of memory management efficiency as well as flops.
- For optimal performance try and use libraries such as LAPACK, with a fast BLAS, where-ever possible.

## 2.2 Some terminology

Now consider some more interesting things to do with matrices. First recall the following basic terminology.

| | | |
|---|---|---|
| $\begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{pmatrix}$ **square** | $\begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{12} & x_{22} & x_{23} \\ x_{13} & x_{23} & x_{33} \end{pmatrix}$ **symmetric** | $\mathbf{y}^\mathsf{T}\mathbf{X}\mathbf{y} > 0 \; \forall \; \mathbf{y} \neq \mathbf{0}$ $(\mathbf{y}^\mathsf{T}\mathbf{X}\mathbf{y} \geq 0 \; \forall \; \mathbf{y} \neq \mathbf{0})$ **positive (semi−)definite** |
| $\begin{pmatrix} x_{11} & 0 & 0 \\ x_{21} & x_{22} & 0 \\ x_{31} & x_{32} & x_{33} \end{pmatrix}$ **lower triangular** | $\begin{pmatrix} x_{11} & x_{12} & x_{13} \\ 0 & x_{22} & x_{23} \\ 0 & 0 & x_{33} \end{pmatrix}$ **upper triangular** | $\begin{pmatrix} x_{11} & 0 & 0 \\ 0 & x_{22} & 0 \\ 0 & 0 & x_{33} \end{pmatrix}$ **diagonal** |

$\mathbf{Q}$ is an **orthogonal matrix** iff $\mathbf{Q}^\mathsf{T}\mathbf{Q} = \mathbf{Q}\mathbf{Q}^\mathsf{T} = \mathbf{I}$.

## 2.3 Cholesky decomposition: a matrix square root

Positive definite matrices are the 'positive real numbers' of matrix algebra. They have particular computational advantages and occur frequently in statistics, since covariance matrices are usually positive definite (and always positive semi-definite). So let's start with positive definite matrices, and their matrix square roots. To see why matrix square roots might be useful, consider the following.

**Example:** Generating multivariate normal random variables. There exist very quick and reliable methods for simulating i.i.d. $N(0,1)$ random deviates, but suppose that $N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ random vectors are required? Clearly we can generate vectors $\mathbf{z}$ from $N(\mathbf{0}, \mathbf{I})$. If we could find a square root matrix $\mathbf{R}$ such that $\mathbf{R}^\mathsf{T}\mathbf{R} = \boldsymbol{\Sigma}$ then

$$\mathbf{y} \equiv \mathbf{R}^\mathsf{T}\mathbf{z} + \boldsymbol{\mu} \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma}),$$

since the covariance matrix of $\mathbf{y}$ is $\mathbf{R}^\mathsf{T}\mathbf{I}\mathbf{R} = \mathbf{R}^\mathsf{T}\mathbf{R} = \boldsymbol{\Sigma}$ and $\mathbb{E}(\mathbf{y}) = \mathbb{E}(\mathbf{R}^\mathsf{T}\mathbf{z} + \boldsymbol{\mu}) = \boldsymbol{\mu}$.

In general the square root of a positive definite matrix is not uniquely defined, but there is a unique *upper triangular* square root of any positive definite matrix: its *Cholesky factor*. The algorithm for finding the Cholesky factor is easily derived. Consider a $4 \times 4$ example first. The defining matrix equation is

$$\begin{pmatrix} R_{11} & 0 & 0 & 0 \\ R_{12} & R_{22} & 0 & 0 \\ R_{13} & R_{23} & R_{33} & 0 \\ R_{14} & R_{24} & R_{34} & R_{44} \end{pmatrix} \begin{pmatrix} R_{11} & R_{12} & R_{13} & R_{14} \\ 0 & R_{22} & R_{23} & R_{24} \\ 0 & 0 & R_{33} & R_{34} \\ 0 & 0 & 0 & R_{44} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{12} & A_{22} & A_{23} & A_{24} \\ A_{13} & A_{23} & A_{33} & A_{34} \\ A_{14} & A_{24} & A_{34} & A_{44} \end{pmatrix}$$

If the component equations of this expression are written out and solved in the right order, then each contains only one unknown, as the following illustrates (unknown in grey)

$$\begin{aligned} A_{11} &= {R_{11}}^2 \\ A_{12} &= R_{11}R_{12} \\ A_{13} &= R_{11}R_{13} \\ A_{14} &= R_{11}R_{14} \\ A_{22} &= R_{12}^2 + {R_{22}}^2 \\ A_{23} &= R_{12}R_{13} + R_{22}R_{23} \\ &\vdots \\ &\vdots \end{aligned}$$

Generalising to the $n \times n$ case, and using the convention that $\sum_{k=1}^{0} x_i \equiv 0$, we have

$$R_{ii} = \sqrt{A_{ii} - \sum_{k=1}^{i-1} R_{ki}^2}, \quad \text{and} \quad R_{ij} = \frac{A_{ij} - \sum_{k=1}^{i-1} R_{ki}R_{kj}}{R_{ii}}, \quad j > i.$$

Working through these equations in row order, from row one, and starting each row from its leading diagonal component ensures that all r.h.s. quantities are known at each step. Cholesky decomposition requires $n^3/3$ flops and $n$ square roots. In R it is performed by function `chol`, which calls routines in LAPACK or LINPACK.

**Example (continued)**: The following simulates a random draw from

$$N\left(\begin{pmatrix}1\\-1\\3\end{pmatrix}, \begin{pmatrix}2&-1&1\\-1&2&-1\\1&-1&2\end{pmatrix}\right)$$

```
  V <- matrix(c(2,-1,1,-1,2,-1,1,-1,2),3,3)
  mu <- c(1,-1,3)

  R <- chol(V)                ## Cholesky factor of V
  z <- rnorm(3)               ## iid N(0,1) deviates
  y <- t(R)%*%z + mu          ## N(mu,V) deviates
  y
```

```
##           [,1]
## [1,]  1.621485
## [2,] -2.069023
## [3,]  3.939876
```

The following simulates 1000 such vectors, and checks their observed mean and covariance.

```
  Z <- matrix(rnorm(3000),3,1000) ## 1000 N(0,I) 3-vectors
  Y <- t(R)%*%Z + mu              ## 1000 N(mu,V) vectors
  ## and check that they behave as expected...
  rowMeans(Y)                     ## observed mu
```

```
## [1]  1.033903 -1.017923  2.970056
```

```
  (Y-mu)%*%t(Y-mu)/1000           ## observed V
```

```
##            [,1]       [,2]       [,3]
## [1,]  2.0160784 -1.008641  0.9942745
## [2,] -1.0086408  1.970159 -1.0416559
## [3,]  0.9942745 -1.041656  1.9563587
```

As a second application of the Cholesky decomposition, consider evaluating the log likelihood of $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$

$$l = -\frac{n}{2}\log(2\pi) - \frac{1}{2}\log(|\boldsymbol{\Sigma}|) - \frac{1}{2}(\mathbf{y}-\boldsymbol{\mu})^{\mathsf{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{y}-\boldsymbol{\mu}).$$

If we were simply to invert $\boldsymbol{\Sigma}$ to evaluate the final term, it would cost $2n^3$ flops, and we would still need to evaluate the determinant. A Cholesky based approach is much better. It is easy to see that $\boldsymbol{\Sigma}^{-1} = \mathbf{R}^{-1}\mathbf{R}^{-\mathsf{T}}$, where $\mathbf{R}$ is the Cholesky factor of $\boldsymbol{\Sigma}$. So the final term in the log likelihood can be written as $\mathbf{z}^{\mathsf{T}}\mathbf{z}$ where $\mathbf{z} = \mathbf{R}^{-\mathsf{T}}(\mathbf{y}-\boldsymbol{\mu})$. Notice that we don't actually need to evaluate $\mathbf{R}^{-\mathsf{T}}$, but simply to solve $\mathbf{R}^{\mathsf{T}}\mathbf{z} = \mathbf{y}-\boldsymbol{\mu}$ for $\mathbf{z}$. To see how this is done, consider a $4\times 4$ case again:

$$\begin{pmatrix}R_{11}&0&0&0\\R_{12}&R_{22}&0&0\\R_{13}&R_{23}&R_{33}&0\\R_{14}&R_{24}&R_{34}&R_{44}\end{pmatrix}\begin{pmatrix}z_1\\z_2\\z_3\\z_4\end{pmatrix} = \begin{pmatrix}y_1-\mu_1\\y_2-\mu_2\\y_3-\mu_3\\y_4-\mu_4\end{pmatrix}$$

If this system of equations is solved from the top down, then there is only one unknown (shown in grey) at each stage:

$$\begin{aligned} R_{11}z_1 &= y_1 - \mu_1 \\ R_{12}z_1 + R_{22}z_2 &= y_2 - \mu_2 \\ R_{13}z_1 + R_{23}z_2 + R_{33}z_3 &= y_3 - \mu_3 \\ R_{14}z_1 + R_{24}z_2 + R_{34}z_3 + R_{44}z_4 &= y_4 - \mu_4 \end{aligned}$$

The generalisation of this *forward substitution* process to $n$ dimensions is obvious, as is the fact that it cost $O(n^2)$ flops — much cheaper than explicit formation of $\mathbf{R}^{-\mathsf{T}}$, which would involve applying forward substitution to find each column of the unknown $\mathbf{R}^{-\mathsf{T}}$ in the equation $\mathbf{R}^{\mathsf{T}}\mathbf{R}^{-\mathsf{T}} = \mathbf{I}$, at $O(n^3)$ cost.

In R there is a routine `forwardsolve` for doing forward substitution with a lower triangular matrix (and a routine `backsolve` for performing the equivalent *back substitution* with upper triangular matrices). Before using it, we still need to consider $|\mathbf{\Sigma}|$. Again the Cholesky factor helps. From general properties of determinants we know that $|\mathbf{R}^{\mathsf{T}}||\mathbf{R}| = |\mathbf{\Sigma}|$, but because $\mathbf{R}$ is triangular $|\mathbf{R}^{\mathsf{T}}| = |\mathbf{R}| = \prod_{i=1}^{n} R_{ii}$. So given the Cholesky factor the calculation of the determinant is $O(n)$.

**Example:** The following evaluates the log likelihood of the covariance matrix `V` and mean vector `mu`, from the previous example, given an observed $\mathbf{y}^{\mathsf{T}} = (1, 2, 3)$.

```
  y <- 1:3
  z <- forwardsolve(t(R),y-mu)
  logLik <- -length(y)*log(2*pi)/2 -
      sum(log(diag(R))) - sum(z*z)/2
  logLik

## [1] -6.824963
```

Note that Cholesky decomposition of a matrix that is not positive definite will fail. Even positive semi-definite will not work, since in that case a leading diagonal element of the Cholesky factor will become zero, so that computation of the off diagonal elements on the same row is impossible. Positive semi-definite matrices are reasonably common, so this is a practical problem. For the positive semi-definite case, it is possible to modify the Cholesky decomposition by *pivoting*: that is by re-ordering the rows/columns of the original matrix so that the zeroes end up at the end of the leading diagonal of the Cholesky factor, in rows that are all zero. This will not be pursued further here. Rather we will consider a more general matrix decomposition, which provides matrix square roots along with much else.

## 2.4 Eigen-decomposition (spectral-decomposition)

Any symmetric matrix, $\mathbf{A}$ can be written as

$$\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{\mathsf{T}} \tag{2.1}$$

where the matrix $\mathbf{U}$ is orthogonal and $\mathbf{\Lambda}$ is a diagonal matrix, with $i^{th}$ leading diagonal element $\lambda_i$ (conventionally $\lambda_i \geq \lambda_{i+1}$). Post-multiplying both sides of the decomposition by $\mathbf{U}$ we have

$$\mathbf{A}\mathbf{U} = \mathbf{U}\mathbf{\Lambda}.$$

Considering this system one column at a time and writing $\mathbf{u}_i$ for the $i^{\text{th}}$ column of $\mathbf{U}$ we have:

$$\mathbf{A}\mathbf{u}_i = \lambda_i \mathbf{u}_i.$$

i.e. the $\lambda_i$ are the eigenvalues of $\mathbf{A}$, and the columns of $\mathbf{U}$ are the corresponding eigenvectors. (2.1) is the *eigen-decomposition* or *spectral decomposition* of $\mathbf{A}$.

We will not go over the computation of the eigen-decomposition in detail, but it is *not* done using the determinant and characteristic equation of $\mathbf{A}$. One practical scheme is as follows: (i) the matrix $\mathbf{A}$ is first

reduced to tri-diagonal form using repeated pre and post multiplication by simple rank-one orthogonal matrices called *Householder rotations*, (ii) an iterative scheme called *QR-iteration* then pre-and post-multiplies the tri-diagonal matrix by even simpler orthogonal matrices, in order to reduce it to diagonal form. At this point the diagonal matrix contains the eigenvalues, and the product of all the orthogonal matrices gives $\mathbf{U}$. Eigen-decomposition is $O(n^3)$, but a good symmetric eigen routine is around 10 times as computationally costly as a Cholesky routine.

An immediate use of the eigen-decomposition is to provide an alternative characterisation of positive (semi-) definite matrices. All the eigenvalues of a positive (semi-)definite matrix must be positive (non-negative) and real. This is easy to see. Were some eigenvalue, $\lambda_i$ to be negative (zero) then the corresponding eigenvector $\mathbf{u}_i$ would result in $\mathbf{u}_i^\mathsf{T}\mathbf{A}\mathbf{u}_i$ being negative (zero). At the same time the existence of an $\mathbf{x}$ such that $\mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x}$ is negative (zero) leads to a contradiction unless at least one eigenvalue is negative (zero)*.

### 2.4.1 Powers of matrices

Consider raising $\mathbf{A}$ to the power $m$.

$$\mathbf{A}^m = \mathbf{A}\mathbf{A}\mathbf{A}\cdots\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\mathsf{T}\mathbf{U}\mathbf{\Lambda}\mathbf{U}^\mathsf{T}\cdots\mathbf{U}\mathbf{\Lambda}\mathbf{U}^\mathsf{T} = \mathbf{U}\mathbf{\Lambda}\mathbf{\Lambda}\cdots\mathbf{\Lambda}\mathbf{U}^\mathsf{T} = \mathbf{U}\mathbf{\Lambda}^m\mathbf{U}^\mathsf{T}$$

where $\mathbf{\Lambda}^m$ is just the diagonal matrix with $\lambda_i^m$ as the $i^{\text{th}}$ leading diagonal element. This suggests that any real valued function, $f$, of a real valued argument, which has a power series representation, has a natural generalisation to a symmetric matrix valued function of a symmetric matrix argument, i.e.

$$f'(\mathbf{A}) \equiv \mathbf{U}f'(\mathbf{\Lambda})\mathbf{U}^\mathsf{T}$$

where $f'(\mathbf{\Lambda})$ denotes the diagonal matrix with $i^{\text{th}}$ leading diagonal element $f(\lambda_i)$. E.g. $\exp(\mathbf{A}) = \mathbf{U}\exp(\mathbf{\Lambda})\mathbf{U}^\mathsf{T}$.

### 2.4.2 Another matrix square root

For matrices with non-negative eigenvalues we can generalise to non-integer powers. For example it is readily verified that $\sqrt{\mathbf{A}} = \mathbf{U}\sqrt{\mathbf{\Lambda}}\mathbf{U}^\mathsf{T}$ has the property that $\sqrt{\mathbf{A}}\sqrt{\mathbf{A}} = \mathbf{A}$. Notice (i) that $\sqrt{\mathbf{A}}$ is not the same as the Cholesky factor, emphasising the non-uniqueness of matrix square roots, and (ii) that, unlike the Cholesky factor, $\sqrt{\mathbf{A}}$ is well defined for positive semi-definite matrices (and can therefore be applied to *any* covariance matrix). Also, the symmetry of this square root can sometimes be convenient.

### 2.4.3 Matrix inversion, rank and condition

Continuing in the same vein we can write

$$\mathbf{A}^{-1} = \mathbf{U}\mathbf{\Lambda}^{-1}\mathbf{U}^\mathsf{T}$$

where the diagonal matrix $\mathbf{\Lambda}^{-1}$ has $i^{\text{th}}$ leading diagonal element $\lambda_i^{-1}$. Clearly we have a problem if any of the $\lambda_i$ are zero, for the matrix inverse will be undefined. A matrix with no zero eigen-values is termed *full rank*. A matrix with any zero eigenvalues is *rank deficient* and does not have an inverse. The number of non-zero eigenvalues is the *rank* of a matrix.

For some purposes it is sufficient to define a *generalised inverse* or *pseudo-inverse* when faced with rank deficiency, by finding the reciprocal of the non-zero eigenvalues, but setting the reciprocal of the zero eigenvalues to zero. We will not pursue this here.

It is important to understand the consequences of rank deficiency quite well when performing matrix operations involving matrix inversion/matrix equation solving. This is because near rank deficiency is rather easy to achieve by accident, and in finite precision arithmetic is as bad as rank deficiency. First consider trying to solve

$$\mathbf{A}\mathbf{x} = \mathbf{y}$$

*We can write $\mathbf{x} = \mathbf{U}\mathbf{b}$ for some vector $\mathbf{b}$. So $\mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x} < 0 \Rightarrow \mathbf{b}^\mathsf{T}\mathbf{\Lambda}\mathbf{b} < 0 \Rightarrow \sum b_i^2\Lambda_i < 0 \Rightarrow \Lambda_i < 0$ for some $i$.

for $\mathbf{x}$ when $\mathbf{A}$ is rank deficient. In terms of the eigen-decomposition the solution is

$$\mathbf{x} = \mathbf{U}\boldsymbol{\Lambda}^{-1}\mathbf{U}^{\mathsf{T}}\mathbf{y}$$

i.e. $\mathbf{y}$ is rotated to become $\mathbf{y}' = \mathbf{U}^{\mathsf{T}}\mathbf{y}$, the elements of $\mathbf{y}'$ are then divided by the eigenvalues, $\lambda_i$, and the reverse rotation is applied to the result. The problem is that $y'_i/\lambda_i$ is not defined if $\lambda_i = 0$. This is just a different way of showing something that you already know: rank deficient matrices can not be inverted. But the approach also helps in the understanding of near rank deficiency and *ill conditioning*.

An illustrative example highlights the problem. Suppose that $n \times n$ symmetric matrix $\mathbf{A}$ has $n-1$ distinct eigenvalues ranging from 0.5 to 1, and one much smaller magnitude eigenvalue $\epsilon$. Further suppose that we wish to compute with $\mathbf{A}$, on a machine that can represent real numbers to an accuracy of 1 part in $\epsilon^{-1}$. Now consider solving the system

$$\mathbf{A}\mathbf{x} = \mathbf{u}_1 \tag{2.2}$$

for $\mathbf{x}$, where $\mathbf{u}_1$ is the dominant eigenvector of $\mathbf{A}$. Clearly the correct solution is $\mathbf{x} = \mathbf{u}_1$, but now consider computing the answer. As before we have a formal solution

$$\mathbf{x} = \mathbf{U}\boldsymbol{\Lambda}^{-1}\mathbf{U}^{\mathsf{T}}\mathbf{u}_1,$$

but although analytically $\mathbf{u}'_1 = \mathbf{U}^{\mathsf{T}}\mathbf{u}_1 = (1,0,0,\ldots,0)^{\mathsf{T}}$, the best we can hope for computationally is to get $\mathbf{u}'_1 = (1+e_1, e_2, e_3, \ldots, e_n)^{\mathsf{T}}$ where the number $e_j$ are of the order of $\pm\epsilon$. For convenience, suppose that $e_n = \epsilon$. Then, approximately, $\boldsymbol{\Lambda}^{-1}\mathbf{u}'_1 = (1,0,0,\ldots,0,1)^{\mathsf{T}}$, and $\mathbf{x} = \mathbf{U}\boldsymbol{\Lambda}^{-1}\mathbf{u}'_1 = \mathbf{u}_1 + \mathbf{u}_n$, which is not correct. Similar distortions would occur if we used any of the other first $n-1$ eigenvectors in place of $\mathbf{u}_1$: they all become distorted by a spurious $\mathbf{u}_n$ component, with only $\mathbf{u}_n$ itself escaping.

Now consider an *arbitrary* vector $\mathbf{y}$ on the r.h.s. of (2.2). We can always write it as some weighted sum of the eigenvectors $\mathbf{y} = \sum w_i \mathbf{u}_i$. This emphasises how bad the ill-conditioning problem is: all but one of $\mathbf{y}$s components are seriously distorted when multiplied by $\mathbf{A}^{-1}$. By contrast multiplication by $\mathbf{A}$ itself would lead only to distortion of the $\mathbf{u}_n$ component of $\mathbf{y}$, and not the other eigen-vectors, but the $\mathbf{u}_n$ component is the component that is so heavily shrunk by multiplication by $\mathbf{A}$ that it makes almost no contribution to the result, unless we have the misfortune to choose a $\mathbf{y}$ that is proportional to $\mathbf{u}_n$ and nothing else.

A careful examination of the preceding argument reveals that what really matters in determining the seriousness of the consequences of near rank deficiency is the ratio of the largest magnitude to the smallest magnitude eigenvalues, i.e.

$$\kappa = \max|\lambda_i| / \min|\lambda_i|.$$

This quantity is a *condition number* for $\mathbf{A}$. Roughly speaking it is the factor by which errors in $\mathbf{y}$ will be multiplied when solving $\mathbf{A}\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$. Once $\kappa$ begins to approach the reciprocal of the machine precision we're in trouble[†]. A system with a large condition number is referred to as *ill-conditioned*. Orthogonal matrices have $\kappa = 1$, which is why numerical analysts like them so much.

**Example.** We are now in a position to understand what happened when we tried to use the 'normal equations' to fit the simple quadratic model in the introduction. Let's explicitly calculate the condition number of $\mathbf{X}^{\mathsf{T}}\mathbf{X}$.

```
set.seed(1)
xx <- sort(runif(100))
x <- xx + 100  # regenerated same x data
X <- model.matrix(~x + I(x^2)) # and the model matrix
XtX <- crossprod(X)           # form t(X)%*%X (with half the flop count)
lambda <- eigen(XtX)$values
lambda[1]/lambda[3]  # the condition number of X'X
```

```
## [1] 2.355538e+18
```

Of course this raises a couple of obvious questions. Could we have diagnosed the problem directly from $\mathbf{X}$? And how does the `lm` function avoid this problem? Answers soon, but first consider a trick for reducing $\kappa$.

[†]Because the condition number is so important in numerical computation, there are several methods for getting an approximate condition number more cheaply than via eigen decomposition — e.g. see `?kappa` in R.

### 2.4.4 Preconditioning

The discussion of condition numbers given above was for systems involving unstructured matrices (albeit *presented* only in the context of symmetric matrices). Systems involving matrices with special structure are sometimes less susceptible to ill-conditioning than naive computation of the condition number would suggest. For example if $\mathbf{D}$ is a diagonal matrix, then we can accurately solve $\mathbf{D}\mathbf{y} = \mathbf{x}$ for $\mathbf{y}$, however large $\kappa(\mathbf{D})$ is: overflow or underflow are the only limits.

This basic fact can sometimes be exploited to re-scale a problem to improve computational stability. As an example consider *diagonal* preconditioning of the computation of $\mathbf{X}^\mathsf{T}\mathbf{X}$, above. For the original $\mathbf{X}^\mathsf{T}\mathbf{X}$ we have

```
solve(XtX)

## Error in solve.default(XtX): system is computationally singular:  reciprocal
condition number = 3.98647e-19
```

But now suppose that we create a diagonal matrix $\mathbf{D}$, with elements $D_{ii} = 1/\sqrt{(\mathbf{X}^\mathsf{T}\mathbf{X})_{ii}}$. Clearly

$$(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1} = \mathbf{D}(\mathbf{D}\mathbf{X}^\mathsf{T}\mathbf{X}\mathbf{D})^{-1}\mathbf{D},$$

but $(\mathbf{D}\mathbf{X}^\mathsf{T}\mathbf{X}\mathbf{D})^{-1}$ turns out to have a much lower condition number than $\mathbf{X}^\mathsf{T}\mathbf{X}$...

```
  D <- diag(1/diag(XtX)^.5)
  DXXD <- D%*%XtX%*%D
  lambda <- eigen(DXXD)$values
  lambda[1]/lambda[3]

## [1] 429397494367
```

As a result we can now compute the inverse of $\mathbf{X}^\mathsf{T}\mathbf{X}$...

```
  XtXi <- D%*%solve(DXXD,D) ## computable inverse of X'X
  XtXi %*% XtX              ## how accurate?

##         (Intercept)             x       I(x^2)
## [1,]  9.999640e-01 -3.262288e-03 -0.352586841
## [2,]  2.920875e-07  1.000032e+00  0.002340679
## [3,] -1.151890e-09 -1.143870e-07  0.999992137
```

...not perfect, but better than no answer at all.

### 2.4.5 Asymmetric eigen-decomposition

If positive definite matrices are the positive reals of the square matrix system, and symmetric matrices are the reals, then asymmetric matrices are the complex numbers. As such they have complex eigenvectors and eigenvalues. It becomes necessary to distinguish right and left eigenvectors (one is no longer the transpose of the other), and the right and left eigenvector matrices are no longer orthogonal matrices (although they are still inverses of each other). Eigen-decomposition of asymmetric matrices is still $O(n^3)$, but is substantially more expensive than the symmetric case. Using R on the laptop used to prepare these notes, eigen-decomposition of a symmetric $1000 \times 1000$ matrix took 0.5 seconds. The asymmetric equivalent took 3 seconds.

The need to compute with complex numbers somewhat reduces the practical utility of the eigen-decomposition in numerical methods for statistics. It would be better to have a decomposition that provides some of the useful properties of the eigen-decomposition without the inconvenience of complex numbers. The singular value decomposition meets this need.

## 2.5 Singular value decomposition

The *singular values* of $r \times c$ matrix, $\mathbf{A}$ ($r \geq c$) are the positive square roots of the eigenvalues of $\mathbf{A}^{\mathsf{T}}\mathbf{A}$. If $\mathbf{A}$ is positive semi-definite then its singular values are just its eigenvalues, of course. For symmetric matrices, eigenvalues and singular values differ only in sign, if at all. However the singular values are also well defined, and real, for matrices that are not even square, let alone symmetric.

Related to the singular values is the *singular value decomposition*

$$\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^{\mathsf{T}}$$

where $\mathbf{U}$ has orthogonal columns and is the same dimension as $\mathbf{A}$, while $\mathbf{D}$ is a $c \times c$ diagonal matrix of the singular values (usually arranged in descending order), and $\mathbf{V}$ is a $c \times c$ orthogonal matrix.

The singular value decomposition is computed using a similar approach to that used for the symmetric eigen problem: orthogonal bi-diagonalization, followed by QR iteration at $O(rc^2)$ cost (it does *not* involve forming $\mathbf{A}^{\mathsf{T}}\mathbf{A}$) . It is more costly than symmetric eigen-decomposition, but cheaper than the asymmetric equivalent. Example R timings for a $1000 \times 1000$ matrix are 0.5 seconds for symmetric eigen, 1 seconds for SVD and 3 seconds for asymmetric eigen.

```
n = 1000
XA = matrix(rnorm(n*n),ncol=n)
XXt = XA %*% t(XA)
system.time(chol(XXt))

##    user  system elapsed
##   0.031   0.020   0.013

system.time(eigen(XXt, symmetric=TRUE))

##    user  system elapsed
##   0.898   0.377   0.328

system.time(eigen(XA))

##    user  system elapsed
##   6.185   2.657   2.351

system.time(svd(XA))

##    user  system elapsed
##   2.311   0.553   0.756

system.time(qr(XA))

##    user  system elapsed
##   0.398   0.044   0.311
```

The number of its non-zero singular values gives the rank of a matrix, and the SVD is the most reliable method for numerical rank determination (by examination of the size of the singular values relative to the largest singular value). In a similar vein, a general definition of condition number is the ratio of largest and smallest singular values: $\kappa = d_1/d_c$.

**Example** Continuing the simple regression disaster from the introduction, consider the singular values of `X`

```
  d <- svd(X)$d # get the singular values of X
  d

## [1] 1.010455e+05 2.662169e+00 6.474081e-05
```

Clearly, numerically X is close to being rank 2, rather than rank 3. Turning to the condition number...

```
  d[1]/d[3]

## [1] 1560769713
```

$\kappa \approx 2 \times 10^9$ is rather large, especially since it is easy to show that the condition number of $\mathbf{X}^\mathsf{T}\mathbf{X}$ must be the square of this. So we now have a pretty clear diagnosis of the cause of the original problem.

In fact the SVD not only provides a diagnosis of the problem, but one possible solution. We can re-write the solution to the normal equations in terms of the SVD of X.

$$\begin{aligned}(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y} &= (\mathbf{V}\mathbf{D}\mathbf{U}^\mathsf{T}\mathbf{U}\mathbf{D}\mathbf{V}^\mathsf{T})^{-1}\mathbf{V}\mathbf{D}\mathbf{U}^\mathsf{T}\mathbf{y}\\ &= (\mathbf{V}\mathbf{D}^2\mathbf{V}^\mathsf{T})^{-1}\mathbf{V}\mathbf{D}\mathbf{U}^\mathsf{T}\mathbf{y}\\ &= \mathbf{V}\mathbf{D}^{-2}\mathbf{V}^\mathsf{T}\mathbf{V}\mathbf{D}\mathbf{U}^\mathsf{T}\mathbf{y}\\ &= \mathbf{V}\mathbf{D}^{-1}\mathbf{U}^\mathsf{T}\mathbf{y}\end{aligned}$$

Notice two things...

1. The condition number of the system that we have ended up with is exactly the condition number of $\mathbf{X}$, i.e. the square root of the condition number involved in the direct solution of the normal equations.
2. Comparing the final RHS expression to the representation of an inverse in terms of its eigen-decomposition, it is clear that $\mathbf{V}\mathbf{D}^{-1}\mathbf{U}^\mathsf{T}$ is a sort of *pseudo-inverse* $\mathbf{X}$.

The SVD has many uses. One interesting one is low rank approximation of matrices. In a well defined sense, the best rank $k \leq \mathrm{rank}(\mathbf{X})$ approximation to a matrix $\mathbf{X}$ can be expressed in terms of the SVD of $\mathbf{X}$ as

$$\tilde{\mathbf{X}} = \mathbf{U}\tilde{\mathbf{D}}\mathbf{V}^\mathsf{T}$$

where $\tilde{\mathbf{D}}$ is $\mathbf{D}$ with all but the $k$ largest singular values set to 0. Using this result to find low rank approximations to observed covariance matrices is the basis for several dimension reduction techniques in multivariate statistics (although, of course, a symmetric eigen decomposition is then equivalent to SVD). One issue with this sort of approximation is that the full SVD is computed, despite the fact that part of it is then to be discarded (don't be fooled by routines that ask you how many eigen or singular vectors to return — I saved .1 of a second, out of 13 by getting R routine `svd` to only return the first columns of $\mathbf{U}$ and $\mathbf{V}$). Look up Lanczos methods and Krylov subspaces for approaches that avoid this sort of waste.

## 2.6 The QR decomposition

The SVD provided a stable solution to the linear model fitting example, but at a rather high computational cost, prompting the question of whether similar stability could be obtained without the full cost of SVD? The QR decomposition provides a positive answer. We can write any $r \times c$ rectangular matrix $\mathbf{X}$ ($r \geq c$) as the product of columns of an orthogonal matrix and an upper triangular matrix.

$$\mathbf{X} = \mathbf{Q}\mathbf{R}$$

where $\mathbf{R}$ is upper triangular and $\mathbf{Q}$ is of the same dimension as $\mathbf{X}$, with orthogonal columns (so $\mathbf{Q}^\mathsf{T}\mathbf{Q} = \mathbf{I}$, but $\mathbf{Q}\mathbf{Q}^\mathsf{T} \neq \mathbf{I}$). The QR decomposition has a cost of $O(rc^2)$, but it is about 1/3 of the cost of SVD.

Consider the linear modelling example again

$$\begin{aligned}(\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y} &= (\mathbf{R}^\mathsf{T}\mathbf{Q}^\mathsf{T}\mathbf{Q}\mathbf{R})^{-1}\mathbf{R}^\mathsf{T}\mathbf{Q}^\mathsf{T}\mathbf{y}\\ &= (\mathbf{R}^\mathsf{T}\mathbf{R})^{-1}\mathbf{R}^\mathsf{T}\mathbf{Q}^\mathsf{T}\mathbf{y}\\ &= \mathbf{R}^{-1}\mathbf{R}^{-\mathsf{T}}\mathbf{R}^\mathsf{T}\mathbf{Q}^\mathsf{T}\mathbf{y}\\ &= \mathbf{R}^{-1}\mathbf{Q}^\mathsf{T}\mathbf{y}\end{aligned}$$

Now the singular values of $\mathbf{R}$ are the same as the singular values of $\mathbf{X}$, so this system has the same condition number as $\mathbf{X}$. i.e. the QR method provides the stability of SVD without the computational cost (although it still has about twice the cost of solving the normal equations via Cholesky decomposition). In fact the QR method for solving least squares estimates can be derived from first principles, without using the normal equations, in which case it also provides a very compact way of deriving much of the distributional theory required for linear modelling. R routine `lm` uses the QR approach, which is why, in the Introduction, it was able to deal with the case for which the normal equations failed. Of course it is not magic: the subsequent failure even of `lm` arose because I had caused the model matrix condition number to become so bad that even the QR approach fails (I should have centred $x$, by subtracting its mean, or used orthogonal polynomials, via `poly`).

It is also worth noting the connection between the QR and Cholesky decompositions. If $\mathbf{X} = \mathbf{Q}\mathbf{R}$, then $\mathbf{X}^\mathsf{T}\mathbf{X} = \mathbf{R}^\mathsf{T}\mathbf{R}$, where $\mathbf{R}$ is upper triangular, so the QR decomposition of $\mathbf{X}$ gives the Cholesky factorisation of $\mathbf{X}^\mathsf{T}\mathbf{X}$ "for free". Note that most QR routines don't bother ensuring that the diagonal elements of $\mathbf{R}$ are positive. If the positivity of the diagonal elements is an issue (it rarely matters), it can easily be fixed by flipping the signs of the relevant rows of $\mathbf{R}$. This relationship between the QR and Cholesky can be used to directly construct the Cholesky decomposition of a covariance matrix in a stable way, without ever directly constructing the covariance matrix.

Another application of the QR decomposition is determinant calculation. If $\mathbf{A}$ is square and $\mathbf{A} = \mathbf{Q}\mathbf{R}$ then

$$|\mathbf{A}| = |\mathbf{Q}||\mathbf{R}| = |\mathbf{R}| = \prod_i R_{ii}$$

since $\mathbf{R}$ is triangular, while $\mathbf{Q}$ is orthogonal with determinant 1. Usually we need

$$\log|\mathbf{A}| = \sum_i \log|R_{ii}|$$

which underflows to zero much less easily than $|\mathbf{A}|$.

## 2.7 Woodbury's formula

There are a large number of matrix identities that are useful for simplifying computations, and the "matrix cookbook" is an excellent source of these. However, there is a set of identities relating to the updating of the inverse of a large matrix that is worthy of particular mention. There are several different variants, and each goes by different names. But names such as the *Woodbury identity*, *Sherman-Morrison-Woodbury formula*, *Sherman-Morrison formula*, etc., are all common names for various versions and special cases. Here I'll just give the most commonly encountered variants.

The context is a (large) $m \times m$ matrix, $\mathbf{A}$, and its inverse, $\mathbf{A}^{-1}$. We suppose that $\mathbf{A}^{-1}$ is already known, but that now $\mathbf{A}$ is subject to a low-rank update of the form $\mathbf{A} + \mathbf{U}\mathbf{V}^\mathsf{T}$, for $\mathbf{U}, \mathbf{V}$ both $m \times n$ with $n << m$. The problem is how to compute the updated inverse $(\mathbf{A} + \mathbf{U}\mathbf{V}^\mathsf{T})^{-1}$ in an efficient way. Woodbury's formula gives the answer.

$$(\mathbf{A} + \mathbf{U}\mathbf{V}^\mathsf{T})^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{I}_n + \mathbf{V}^\mathsf{T}\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^\mathsf{T}\mathbf{A}^{-1}$$

Note that crucially, the inverse on the LHS is $m \times m$, whereas the new inverse required on the RHS is $n \times n$. This is what makes the result so useful.

The result is easy to prove by direct multiplication. First, pre-multiply the RHS by $\mathbf{A} + \mathbf{U}\mathbf{V}^\mathsf{T}$, multiply out, and simplify, to get $\mathbf{I}_m$. This tells us that the RHS is a right-inverse. Next post-multiply the RHS by

$\mathbf{A} + \mathbf{U}\mathbf{V}^\mathsf{T}$, multiply out, simplify to get $\mathbf{I}_m$, thus confirming that the RHS is both a left- and right-inverse, and hence the inverse.

The form given above is sufficiently general for most applications in statistical computing, but it is worth noting a modest generalisation, often referred to as the *Sherman-Morrison-Woodbury formula*, which introduces an additional $n \times n$ matrix, $\mathbf{C}$.

$$(\mathbf{A} + \mathbf{U}\mathbf{C}\mathbf{V}^\mathsf{T})^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{C}^{-1} + \mathbf{V}^\mathsf{T}\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^\mathsf{T}\mathbf{A}^{-1}$$

This can be proved by direct multiplication, as before. There is also a very commonly encountered special case, where $\mathbf{U}$ and $\mathbf{V}$ are just $m \times 1$ vectors, $\mathbf{u}, \mathbf{v}$. In that case the identity is often referred to as the *Sherman-Morrison formula*, and reduces to

$$(\mathbf{A} + \mathbf{u}\mathbf{v}^\mathsf{T})^{-1} = \mathbf{A}^{-1} - \frac{\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^\mathsf{T}\mathbf{A}^{-1}}{1 + \mathbf{v}^\mathsf{T}\mathbf{A}^{-1}\mathbf{u}}.$$

Writing it this way makes it clear that no new matrix inversion is required to perform the update.

There are numerous applications of these identities in statistical computing, but updating a precision matrix based on a low-rank update of a covariance matrix is a typical example.

## 2.8 Further reading on matrices

These notes can only scratch the surface of numerical linear algebra. Hopefully you now have a clearer idea of the importance of stability and efficiency in matrix computation, and some understanding of a few key decompositions and some of their applications. My hope is that when writing any code involving matrices, you'll at least think about flops and condition numbers. If you want to find out more, here are some starting points. See section 6.3 as well.

- Golub, GH and CF Van Loan (1996) *Matrix Computations* 3rd ed. John Hopkins. This is *the* reference for numerical linear algebra, but not the easiest introduction. It is invaluable if you are writing carefully optimised matrix computations, directly calling BLAS and LAPACK functions.
- Watkins, DS (2002) *Fundamentals of Matrix Computations* 2nd ed. Wiley. This is a more accessible book, but still aimed at those who really want to understand the material (the author encourages you to write your own SVD routine, and with this book you can).
- Press WH, SA Teukolsky, WT Vetterling and BP Flannery (2007) *Numerical Recipes (3rd ed)*, Cambridge. Very easily digested explanation of how various linear algebra routines work, with code. But there are better versions of most of the algorithms in this book.
- The GNU scientific library is at `http://www.gnu.org/software/gsl/`, and is a better bet than Numerical Recipes for actual code.
- For state of the art numerical linear algebra use LAPACK: `http://www.netlib.org/lapack/`.
- Matlab offers a convenient, but expensive, platform for matrix computation. For an introduction see e.g. Pratap, R (2006) *Getting Started with Matlab*, Oxford.
- Octave is Matlab like free software: `http://www.gnu.org/software/octave/`.
- Julia: `http://julialang.org/` is another dynamically typed scripting language aimed at scientific computing.
- Monahan, JF (2001) *Numerical Methods of Statistics* provides a useful treatment of numerical linear algebra for statistics.
- Harville, DA (1997) *Matrix Algebra From a Statistician's Perspective*, Springer, provides a wealth of helpful theoretical material.

# Chapter 3

# Optimisation

Many problems in statistics can be characterised as solving

$$\min_{\theta} f(\boldsymbol{\theta}) \tag{3.1}$$

where $f$ is a real scalar valued function of vector $\boldsymbol{\theta}$, usually known as the *objective function*. For example: $f$ might be the negative log-likelihood of the parameters $\boldsymbol{\theta}$ of a statistical model, the maximum likelihood estimates of which are required; in a Bayesian setting $f$ might be a (negative) posterior distribution for $\boldsymbol{\theta}$, the posterior modes of which are required; $f$ might be a dissimilarity measure for the alignment of two DNA sequences, where the alignment is characterised by a parameter vector $\boldsymbol{\theta}$; $f$ might be total distance travelled, and $\boldsymbol{\theta}$ a vector giving the order in which a delivery van should visit a set of drop off points. Note that maximising $f$ is equivalent to minimising $-f$ so we are not losing anything by concentrating on minimisation here. In the optimisation literature, the convention is to mimimise an objective, with the idea being that the objective is some kind of *cost* or *penalty* to be minimised. Indeed, the objective function is sometimes referred to as the *cost function*.

Let $\Theta$ denote the set of all possible $\boldsymbol{\theta}$ values. Before discussing specific methods, note that in practice we will only be able to solve (3.1) if one of two properties holds.

1. It is practical to evaluate $f(\boldsymbol{\theta})$ for all elements of $\Theta$.
2. It is possible to put some structure onto $\Theta$, giving a notion of distance between any two elements of $\Theta$, such that closeness in $\Theta$ implies closeness in $f$ value.

In complex situations, requiring complicated stochastic approaches, it is easy to lose sight of point 2 (although if it isn't met we may still be able to find ways of locating $\theta$ values that reduce $f$).

Optimisation problems vary enormously in difficulty. In the matrix section we saw algorithms with operation counts which might be proportional to as much as $n^3$. There are optimisation problems whose operations count is larger than $O(n^k)$ for any $k$: in computer science terms there is no *polynomial time algorithm*[*] for solving them. Most of these really difficult problems are discrete optimisation problems, in that $\boldsymbol{\theta}$ takes only discrete values, rather than consisting of unconstrained real numbers. One such problem is the famous *Travelling salesman problem*, which concerns finding the shortest route between a set of fixed points, each to be visited only once. Note however, that simply having a way of improving approximate solutions to such problems is usually of considerable practical use, even if exact solution is very taxing.

Continuous optimisation problems in which $\boldsymbol{\theta}$ is a real random vector are generally more straightforward than discrete problems, and we will concentrate on such problems here. In part this is because of the centrality of optimisation methods in practical likelihood maximisation.

[*]Computer scientists have an interesting classification of the 'difficulty' of problems that one might solve by computer. If $n$ is the problem 'size' then the easiest problems can be solved in polynomial time: they require $O(n^k)$ operations where $k$ is finite. These problems are in the P-class. Then there are problems for which a proposed solution can be *checked* in polynomial time, the NP class. NP-complete problems are problems in NP which are at least as hard as all other problems in NP in a defined sense. NP hard problems are at least as hard as all problems in NP, but are not in NP: so you can't even check the answer to an NP hard problem in polynomial time. No one knows if P and NP are the same, but if any NP-complete problem turned out to be in P, then they would be.

## 3.1 Local versus global optimisation

Minimisation methods generally operate by seeking a *local* minimum of $f$. That is they seek a point, $\boldsymbol{\theta}^*$, such that $f(\boldsymbol{\theta}^*+\boldsymbol{\Delta}) \geq f(\boldsymbol{\theta}^*)$, for any *sufficiently small* perturbation $\boldsymbol{\Delta}$. Unless $f$ is a strictly convex function, it is not possible to guarantee that such a $\boldsymbol{\theta}^*$ is a *global* minimum. That is, it is not possible to guarantee that $f(\boldsymbol{\theta}^* + \boldsymbol{\Delta}) \geq f(\boldsymbol{\theta}^*)$ for *any arbitrary* $\boldsymbol{\Delta}$.

## 3.2 Optimisation methods are myopic

Before going further it is worth stressing another feature of optimisation methods: they are generally very short-sighted. At any stage of operation all that optimisation methods 'know' about a function are a few properties of the function *at the current best* $\boldsymbol{\theta}$, (and, much less usefully, those same properties at previous best points). It is on the basis of such information that a method tries to find an improved best $\boldsymbol{\theta}$. The methods have no 'overview' of the function being optimised.

## 3.3 Look at your objective first

Before attempting to optimise a function with respect to some parameters, it is a good idea to produce plots in order to get a feel for the function's behaviour. Of course if you could simply plot the function over the whole of $\Theta$ it is unlikely that you would need to worry about numerical optimisation, but generally this is not possible, and the best that can be achieved is to look at some one or two dimensional transects through $f$.

To fix ideas, consider the apparently innocuous example of fitting a simple dynamic model to a single time series by least squares/maximum likelihood. The dynamic model is

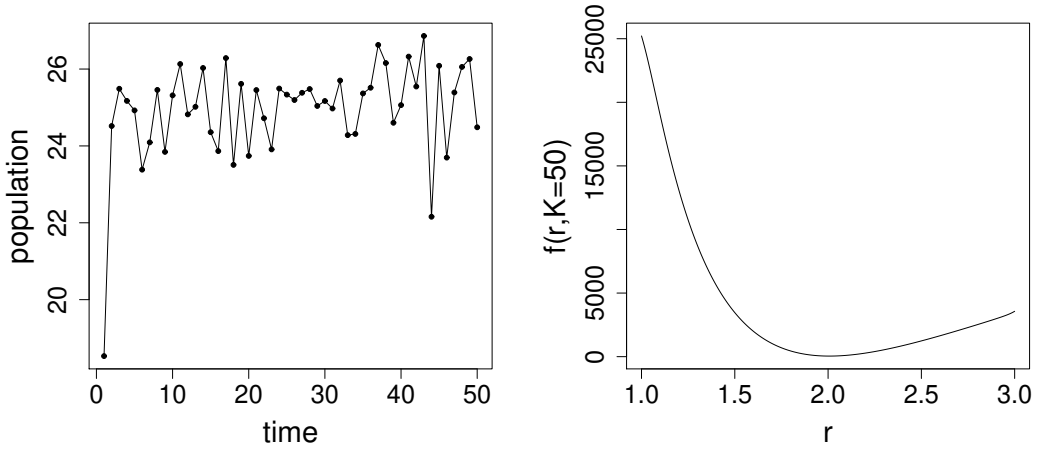
$$n_{t+1} = rn_t(1 - n_t/K), \quad t = 0, 1, 2, \ldots$$

where $r$ and $K$ are parameters and we will assume that $n_0$ is known. Further suppose that we have observations $y_t = n_t + \epsilon_t$ where $\epsilon_t \underset{\text{i.i.d.}}{\sim} N(0, \sigma^2)$. Estimation of $r$ and $K$ by least squares (or maximum likelihood, in this case) requires minimisation of

$$f(r, K) = \sum_i \{y_i - n_i(r, K)\}^2$$

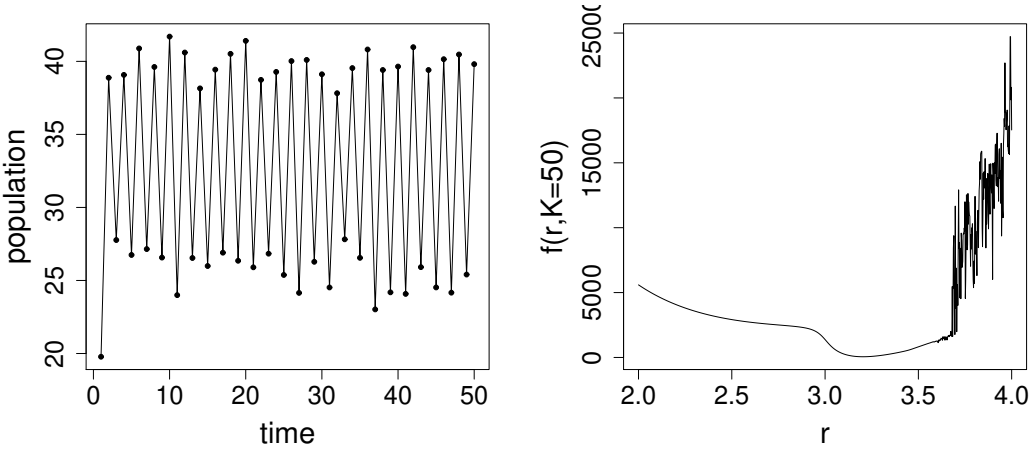
w.r.t. $r$ and $K$. We should try to get a feel for the behaviour of $f$. To see how this can work consider some simulated data examples. In each case I used $n_0 = 20$, $K = 50$ and $\sigma = 1$ for the simulations, but I have varied $r$ between the cases.

- In the first instance data were simulated with $r = 2$. If we now pretend that we need to estimate $r$ and $K$ from such data, then we might look at some $r$-transects and some $K$-transects through $f$. This figure shows the raw data and an $r$-transect.



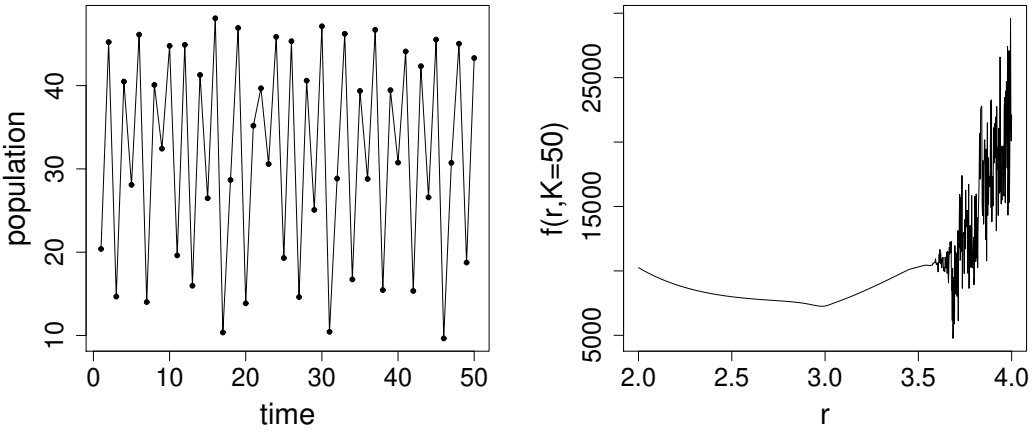
$r$-transects with other $K$ values look equally innocuous, and $K$-transects also look benign over this $r$ range. So in this case $f$ appears to be a nice smooth function of $r$ and $K$, and any half decent optimisation method ought to be able to find the optimum.

- In the second example data were simulated with $r = 3.2$. Again pretend that we only have the data and need to estimate $r$ and $K$ from it. Examination of the left hand plot below, together with some knowledge of the model's dynamic behaviour, indicates that $r$ is probably in the 3 to 4 range.




Notice that $f$ still seems to behave smoothly in the vicinity of its minimum, which is around 3.2, but that something bizarre starts to happen above 3.6. Apparently we could be reasonably confident of locating the function's minimum if we start out below or reasonably close to 3.2, but if we stray into the territory above 3.6 then there are so many local minima that locating the desired global minimum would be very difficult.

- Finally data were simulated with $r = 3.8$.

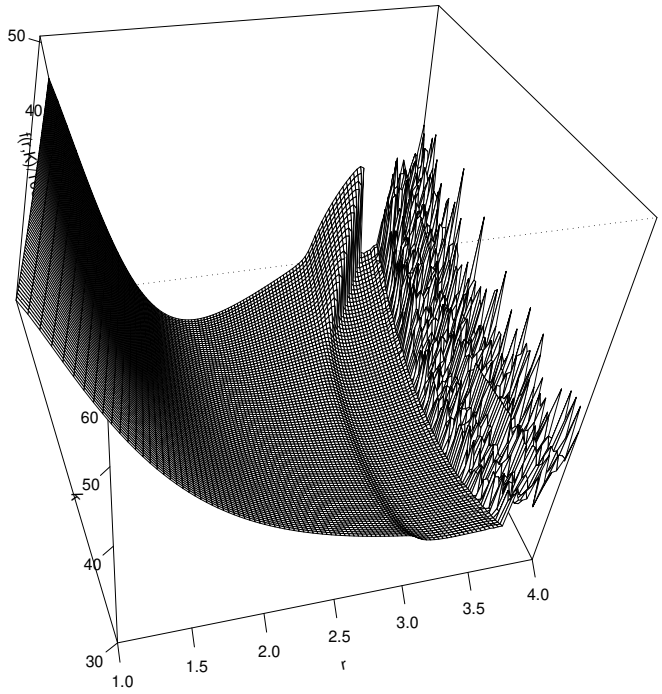



Now the objective has a minimum somewhere around 3.7, but it is surrounded by other local minima, so that locating the actual minima would be a rather taxing problem. In addition it is now unclear how we would go about quantifying uncertainty about the 'optimal' $\boldsymbol{\theta}$: it will certainly be of no use appealing to asymptotic likelihood arguments in this case.

So, in each of these examples, simple transects through the objective function have provided useful information. In the first case everything seemed OK. In the second we would need to be careful with the optimisation, and careful to examine how sensible the optimum appeared to be. In the third case we would

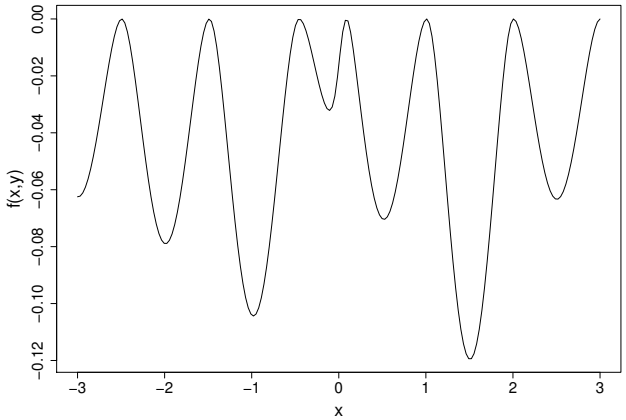
need to think very carefully about the purpose of the optimisation, and about whether a re-formulation of the basic problem might be needed.

Notice how the behaviour of the objective was highly parameter dependent here, something that emphasises the need to understand models quite well before trying to fit them. In this case the dynamic model, although very simple, can show a wide range of complicated behaviour, including chaos. In fact, for this simple example, it is possible to produce a plot of the fitting objective over pretty much the whole plausible parameter space. Here is an example when data were simulated with $r = 3.2$

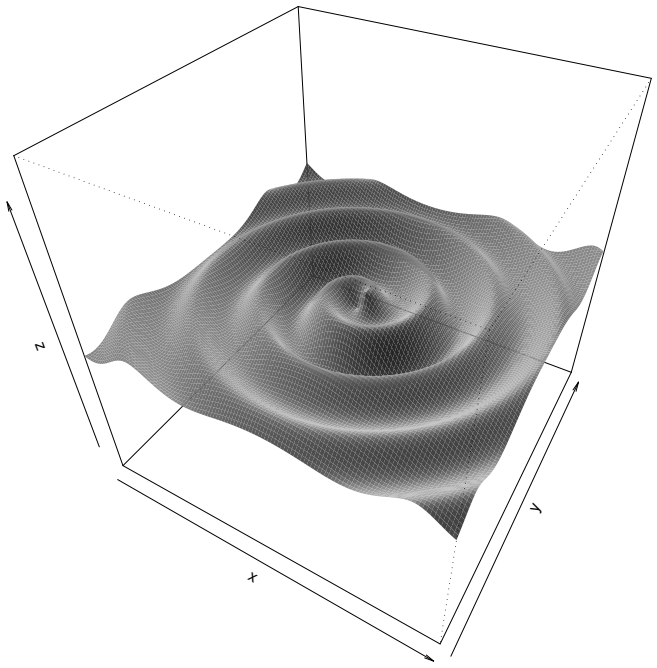



### 3.3.1 Objective function transects are a partial view

Clearly plotting some transects through your objective function is a good idea, but they can only give a limited and partial view when the $\theta$ is multidimensional. Consider this $x$-transect through a function, $f(x, y)$

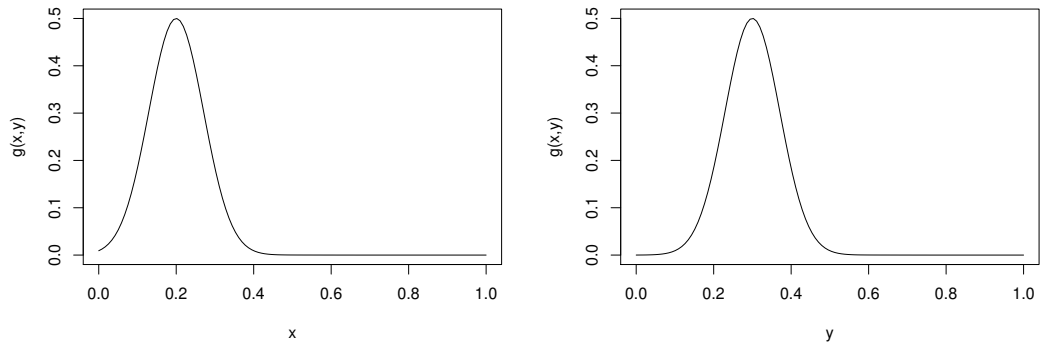



Apparently the function has many local minima, and optimisation will be difficult. Actually the function is this. . .

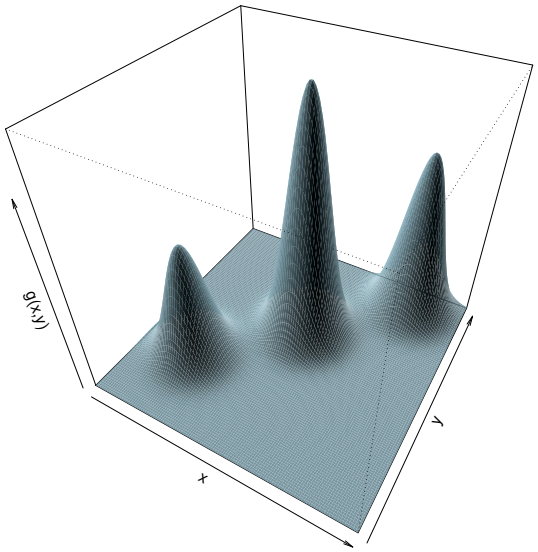



It has one local minimum, its global minimum. Head downhill from any point $x, y$ and you will eventually reach the minimum.

The opposite problem can also occur. Here are $x$ and $y$ transects through a second function $g(x, y)$.




. . . apparently no problem. From these plots you would be tempted to conclude that $g$ is a well behaved and uni-modal. The problem is that $g$ looks like this. . .




So, generally speaking, it is a good idea to plot transects through your objective function before optimisation, and to plot transects passing through the apparent optimum after optimisation, but bear in mind that transects only give partial views.

## 3.4 Some optimisation methods

Optimisation methods start from some initial guesstimate of the optimum, $\boldsymbol{\theta}^{[0]}$. Starting from k=0, most methods then iterate the following steps.

1. Evaluate $f(\boldsymbol{\theta}^{[k]})$, and possibly the first and second derivatives of $f$ w.r.t. the elements of $\boldsymbol{\theta}$, at $\boldsymbol{\theta}^{[k]}$.
2. Either
   (a) use the information from step 1 (and possibly previous executions of step 1) to find a *search direction*, $\boldsymbol{\Delta}$, such that for some $\alpha$, $f(\boldsymbol{\theta}^{[k]} + \alpha\boldsymbol{\Delta})$ will provide sufficient decrease relative to $f(\boldsymbol{\theta}^{[k]})$. Search for such an $\alpha$ and set $\boldsymbol{\theta}^{[k+1]} = \boldsymbol{\theta}^{[k]} + \alpha\boldsymbol{\Delta}$. Or
   (b) use the information from step 1 (and possibly previous executions of step 1) to construct a local model of $f$, and find the $\boldsymbol{\theta}$ minimising this model within a defined region around $\boldsymbol{\theta}^{[k]}$. If this value of $\boldsymbol{\theta}$ leads to sufficient decrease in $f$ itself, accept it as $\boldsymbol{\theta}^{[k+1]}$, otherwise shrink the search region until it does.
3. Test whether a minimum has yet been reached. If it has, terminate, otherwise increment $k$ and return to 1.

Methods employing approach (a) at step 2 are known as *search direction methods*, while those employing approach (b) are *trust region methods*. In practice the difference between (a) and (b) may be rather small: the search direction is often based on a local model of the function. Here we will examine some search direction methods.

### 3.4.1 Taylor's Theorem (and conditions for an optimum)

We need only a limited version of Taylor's theorem. Suppose that $f$ is a twice continuously differentiable function of $\boldsymbol{\theta}$ and that $\boldsymbol{\Delta}$ is of the same dimension as $\boldsymbol{\theta}$. Then

$$f(\boldsymbol{\theta}+\boldsymbol{\Delta}) = f(\boldsymbol{\theta}) + \nabla f(\boldsymbol{\theta})^{\mathsf{T}}\boldsymbol{\Delta} + \frac{1}{2}\boldsymbol{\Delta}^{\mathsf{T}}\nabla^2 f(\boldsymbol{\theta}+t\boldsymbol{\Delta})\boldsymbol{\Delta}$$

for some $t \in (0,1)$, where

$$\nabla f(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial f}{\partial \theta_1} \\ \frac{\partial f}{\partial \theta_2} \\ \cdot \\ \cdot \end{pmatrix} \text{ and } \nabla^2 f(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial^2 f}{\partial \theta_1^2} & \frac{\partial^2 f}{\partial \theta_1 \partial \theta_2} & \cdot & \cdot \\ \frac{\partial^2 f}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 f}{\partial \theta_2^2} & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

From Taylor's theorem it is straightforward to establish that the conditions for $f(\boldsymbol{\theta}^*)$ to be a minimum of $f$ are that $\nabla f(\boldsymbol{\theta}^*) = \mathbf{0}$ and $\nabla^2 f(\boldsymbol{\theta}^*)$ is positive semi-definite.

Here we have given an *exact* form of Taylor's theorem for a real-valued many-variable function, for a first order approximation plus a second order remainder term in Langrange's form. In practice, in applications, we typically work with approximate versions of the above result. In some cases we work directly with a linear (first-order) approximation, by setting the remainder term to zero. In other cases we work with the quadratic (second-order) approximation, obtained by evaluating the remainder term at $t = 0$.

### 3.4.2 Steepest descent (AKA gradient descent)

Given some parameter vector $\boldsymbol{\theta}^{[k]}$, it might seem reasonable to simply ask which direction would lead to the most rapid decrease in $f$ for a sufficiently small step, and to use this as the search direction. If $\boldsymbol{\Delta}$ is small enough then Taylor's theorem implies that

$$f(\boldsymbol{\theta}^{[k]}+\boldsymbol{\Delta}) \simeq f(\boldsymbol{\theta}^{[k]}) + \nabla f(\boldsymbol{\theta}^{[k]})^{\mathsf{T}}\boldsymbol{\Delta}.$$

For $\boldsymbol{\Delta}$ of fixed length the greatest decrease will be achieved by minimising the inner product $\nabla f(\boldsymbol{\theta}^{[k]})^{\mathsf{T}}\boldsymbol{\Delta}$, which is achieved by making $\boldsymbol{\Delta}$ parallel to $-\nabla f(\boldsymbol{\theta}^{[k]})$, i.e. by choosing

$$\boldsymbol{\Delta} \propto -\nabla f(\boldsymbol{\theta}^{[k]}).$$

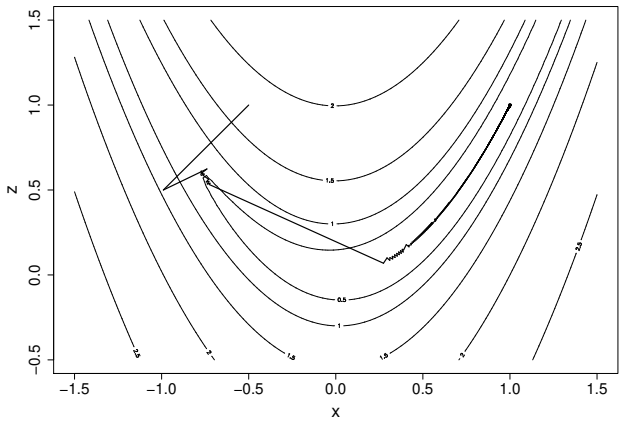
To firm things up, let's set $\boldsymbol{\Delta} = -\nabla f(\boldsymbol{\theta}^{[k]})/\|\nabla f(\boldsymbol{\theta}^{[k]})\|$, and consider a new proposed value of the form $\boldsymbol{\theta}^{[k]} + \alpha\boldsymbol{\Delta}$ for $\alpha > 0$. Then, by plugging back into our linear approximation, we have

$$\begin{aligned} f(\boldsymbol{\theta}^{[k]}+\alpha\boldsymbol{\Delta}) &\simeq f(\boldsymbol{\theta}^{[k]}) - \alpha\frac{\nabla f(\boldsymbol{\theta})^{\mathsf{T}}\nabla f(\boldsymbol{\theta})}{\|\nabla f(\boldsymbol{\theta}^{[k]})\|} \\ &= f(\boldsymbol{\theta}^{[k]}) - \alpha\|\nabla f(\boldsymbol{\theta}^{[k]})\|, \end{aligned}$$

and so as we increase $\alpha$ from zero, we decrease $f(\cdot)$ at a rate proportional to the magnitude of the gradient vector. Any other choice of $\boldsymbol{\Delta}$ will decrease the function value at a slower rate.

Given a search direction, we need a method for choosing how far to move from $\boldsymbol{\theta}^{[k]}$ along this direction. Here there is a trade-off. We need to make reasonable progress, but do not want to spend too much time choosing the exactly optimal distance to move, since the point will only be abandoned at the next iteration anyway. Usually it is best to try and choose the step length $\alpha$ so that $f(\boldsymbol{\theta}^{[k]}+\alpha\boldsymbol{\Delta})$ is sufficiently much lower that $f(\boldsymbol{\theta}^{[k]})$, and also so that the magnitude of the gradient of $f$ in the $\boldsymbol{\Delta}$ direction is sufficiently reduced by the step. See e.g. Section 3.1 of Nocedal and Wright (2006) for further details. For the moment presume that we have a step selection method.

The following illustrates the progress of the steepest descent method on Rosenbrock's function. The contours are $\log_{10}$ of Rosenbrock's function, and the thick line shows the path of the algorithm.




It took 14,845 steps to reach the minimum. This excruciatingly slow progress is typical of the steepest descent method. Another disadvantage of steepest descent is **scale dependence**: imagine linearly re-scaling so that the $z$ axis in the above plot ran from -5 to 15, but the contours were unchanged. This would change all the steepest descent directions. By contrast, the methods considered next are scale invariant.

### 3.4.3 Newton's method

As steepest descent approaches the minimum, the first derivative term retained in the Taylor approximation of $f$ becomes negligible relative to the neglected second derivative term. This largely explains the method's poor performance. Retaining the second derivative term in the Taylor expansion yields the method of choice for a wide range of problems: Newton's method. Taylor's theorem implies that

$$f(\boldsymbol{\theta}^{[k]} + \boldsymbol{\Delta}) \simeq f(\boldsymbol{\theta}^{[k]}) + \nabla f(\boldsymbol{\theta}^{[k]})^{\mathsf{T}} \boldsymbol{\Delta} + \frac{1}{2} \boldsymbol{\Delta}^{\mathsf{T}} \nabla^2 f(\boldsymbol{\theta}^{[k]}) \boldsymbol{\Delta}.$$

Differentiating the r.h.s. w.r.t. $\boldsymbol{\Delta}$ and setting the result to $\mathbf{0}$ we have

$$\nabla^2 f(\boldsymbol{\theta}^{[k]}) \boldsymbol{\Delta} = -\nabla f(\boldsymbol{\theta}^{[k]})$$

as the equation to solve for the step $\boldsymbol{\Delta}$ which will take us to the turning point of the (quadratic) Taylor approximation of $f$. This turning point will be a minimum (of the approximation) if $\nabla^2 f(\boldsymbol{\theta}^{[k]})$ is positive definite, and can also be derived directly without vector calculus by completing the square.

Notice how Newton's method automatically suggests a search direction *and* a step length. Usually we simply accept the Newton step, $\boldsymbol{\Delta}$, unless it fails to lead to sufficient decrease in $f$. In the latter case the step length is repeatedly halved until sufficient decrease is achieved.

The second practical issue with Newton's method is the requirement that $\nabla^2 f(\boldsymbol{\theta}^{[k]})$ is positive definite. There is no guarantee that it will be, especially if $\boldsymbol{\theta}^{[k]}$ is far from the optimum. This would be a major problem, were it not for the fact that the solution to

$$\mathbf{H} \boldsymbol{\Delta} = -\nabla f(\boldsymbol{\theta}^{[k]})$$

is a descent direction for *any* positive definite $\mathbf{H}$. To see this, put $\boldsymbol{\Delta} = -\mathbf{H}^{-1} \nabla f(\boldsymbol{\theta}^{[k]})$ as a search direction in our first-order approximation,

$$\begin{aligned} f(\boldsymbol{\theta} + \alpha \boldsymbol{\Delta}) &\simeq f(\boldsymbol{\theta}) + \alpha \nabla f(\boldsymbol{\theta})^{\mathsf{T}} \boldsymbol{\Delta} \\ &= f(\boldsymbol{\theta}) - \alpha \nabla f(\boldsymbol{\theta})^{\mathsf{T}} \mathbf{H}^{-1} \nabla f(\boldsymbol{\theta}) \\ &< f(\boldsymbol{\theta}), \end{aligned}$$

for small $\alpha > 0$ due to the positive-definiteness of $\mathbf{H}^{-1}$. This gives us the freedom to modify $\nabla^2 f(\boldsymbol{\theta}^{[k]})$ to make it positive definite, and still be guaranteed a descent direction. One approach is to take the symmetric eigen-decomposition of $\nabla^2 f(\boldsymbol{\theta}^{[k]})$, reverse the sign of any negative eigenvalues, replace any

zero eigenvalues with a small positive number and then replace $\nabla^2 f(\boldsymbol{\theta}^{[k]})$ by the matrix with this modified eigen-decomposition. Computationally cheaper alternatives simply add a multiple of the identity matrix to $\nabla^2 f(\boldsymbol{\theta}^{[k]})$, sufficiently large to make the result positive definite. Note that steepest descent corresponds to the choice $\boldsymbol{H} = \boldsymbol{I}$, and so inflating the diagonal of the Hessian corresponds to nudging the search direction from that of the Newton method towards that of steepest descent, and this is usually a fairly safe and conservative thing to do.

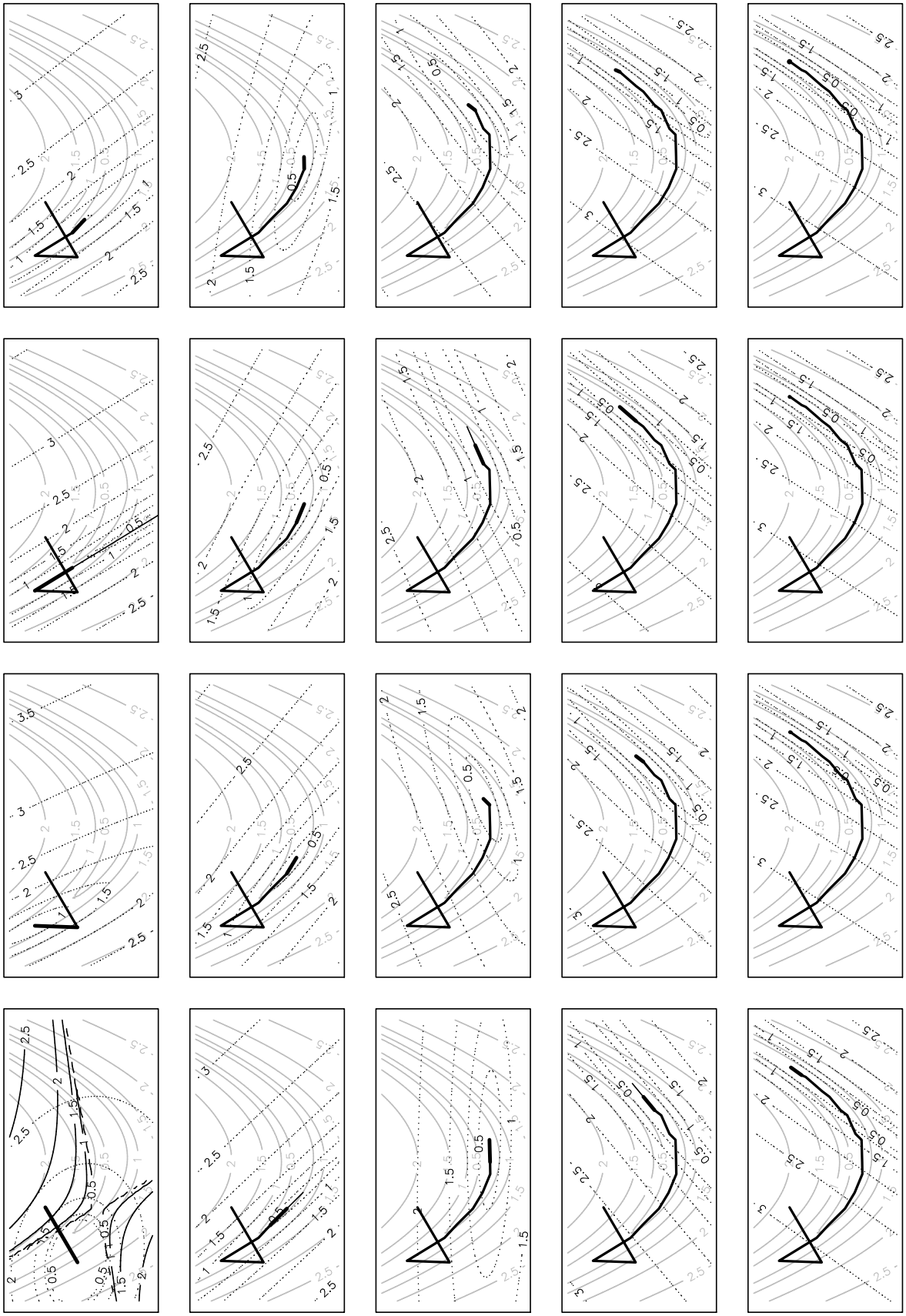
Combining these steps gives a practical Newton algorithm. Starting with $k = 0$ and a guesstimate $\boldsymbol{\theta}^{[0]}$...

1. Evaluate $f(\boldsymbol{\theta}^{[k]})$, $\nabla f(\boldsymbol{\theta}^{[k]})$ and $\nabla^2 f(\boldsymbol{\theta}^{[k]})$, implying a quadratic approximation to $f$, via Taylor's theorem.
2. Test whether $\boldsymbol{\theta}^{[k]}$ is a minimum, and terminate if it is.
3. If $\nabla^2 f(\boldsymbol{\theta}^{[k]})$ is not positive definite, perturb it so that it is (thereby modifying the quadratic model of $f$).
4. The search direction $\boldsymbol{\Delta}$ is now defined by the solution to

   $$\nabla^2 f(\boldsymbol{\theta}^{[k]})\boldsymbol{\Delta} = -\nabla f(\boldsymbol{\theta}^{[k]}),$$

   i.e. the $\boldsymbol{\Delta}$ minimising the current quadratic model.
5. If $f(\boldsymbol{\theta}^{[k]} + \boldsymbol{\Delta})$ is not sufficiently lower than $f(\boldsymbol{\theta}^{[k]})$, repeatedly halve $\boldsymbol{\Delta}$ until it is.
6. Set $\boldsymbol{\theta}^{[k+1]} = \boldsymbol{\theta}^{[k]} + \boldsymbol{\Delta}$, increment $k$ by one and return to step 1.

The following figure illustrates the progress of the method on Rosenbrock's function, working across rows from top left. Grey contours show the function, dotted show the quadratic approximation at the current best point, and, if present, black contours are the quadratic approximation before modification of the Hessian. The thick black line tracks the steps accepted, while the thin black line shows any suggested steps that were reduced. Note the fact that the method converged in 20 steps, compared to the 14,845 steps taken by steepest descent.



### 3.4.4 Quasi-Newton methods

Newton's method usually performs very well, especially near the optimum, but to use it we require the second derivative (*Hessian*) matrix of the objective. This can be tedious to obtain. Furthermore, if the

dimension of $\boldsymbol{\theta}$ is large, then solving for the Newton direction can be a substantial part of the computational burden of optimisation. Is it possible to produce methods that only require first derivative information, but with efficiency rivalling Newton's method, rather than steepest descent?

The answer is yes. One approach is to replace the exact Hessian, of Newton's method, with an approximate Hessian based on finite differences of gradients. The finite difference intervals must be chosen with care (see section 4.2), but the approach is reliable (recall that any positive definite matrix in place of the Hessian will yield a descent direction, so an approximate Hessian should be unproblematic). However, finite differencing for the Hessian is often more costly than exact evaluation, and there is no saving in solving for the search direction.

The second approach is to use a *quasi-Newton* method. The basic idea is to use first derivative information gathered from past steps of the algorithm to accumulate an approximation to the Hessian at the current best point. In principle, this approximation can be used exactly like the exact Hessian in Newton's method, but if structured carefully, such an algorithm can also work directly on an approximation to the inverse of the Hessian, thereby reducing the cost of calculating the actual step. It is also possible to ensure that the approximate Hessian is *always* positive definite.

Quasi-Newton methods were invented in the mid 1950's by W.C. Davidon (a physicist). In the mathematical equivalent of not signing the Beatles, his paper on the method was rejected (it was eventually published in 1991). There are now many varieties of Quasi-Newton method, but the most popular is the BFGS method[†], which will be briefly covered here.

We suppose that at step $k$ we have already computed $\boldsymbol{\theta}^{[k]}$ and an approximate Hessian $\mathbf{H}^{[k]}$. So, just as with a regular Newton method, we solve

$$\mathbf{H}^{[k]}\boldsymbol{\Delta} = -\nabla f(\boldsymbol{\theta}^{[k]})$$

to obtain a new search direction, leading either to a new $\boldsymbol{\theta}^{[k+1]}$ directly, or more likely via line search. We now have the problem of how to appropriately update the approximate Hessian, $\mathbf{H}^{[k+1]}$. We want the approximate Hessian to give us a good quadratic approximation of our function about $\boldsymbol{\theta}^{[k+1]}$. If $\mathbf{H}^{[k+1]}$ is our new approximate positive definite Hessian at the $(k+1)^{\text{th}}$ step, we have the function approximation

$$f(\boldsymbol{\theta}) \simeq f(\boldsymbol{\theta}^{[k+1]}) + \nabla f(\boldsymbol{\theta}^{[k+1]})^{\mathsf{T}}(\boldsymbol{\theta} - \boldsymbol{\theta}^{[k+1]}) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}^{[k+1]})^{\mathsf{T}}\mathbf{H}^{[k+1]}(\boldsymbol{\theta} - \boldsymbol{\theta}^{[k+1]}).$$

The basic requirement of a quasi Newton method is that the new Hessian should be chosen so that the approximation should exactly match $\nabla f(\boldsymbol{\theta}^{[k]})$ — i.e. the approximation should get the gradient vector at the previous best point, $\boldsymbol{\theta}^{[k]}$, exactly right. Taking the gradient of the above approximation gives

$$\nabla f(\boldsymbol{\theta}) \simeq \nabla f(\boldsymbol{\theta}^{[k+1]}) + \mathbf{H}^{[k+1]}(\boldsymbol{\theta} - \boldsymbol{\theta}^{[k+1]}).$$

Evaluating this at $\boldsymbol{\theta}^{[k]}$ leads to our requirement

$$\nabla f(\boldsymbol{\theta}^{[k]}) = \nabla f(\boldsymbol{\theta}^{[k+1]}) + \mathbf{H}^{[k+1]}(\boldsymbol{\theta}^{[k]} - \boldsymbol{\theta}^{[k+1]}),$$

and hence

$$\mathbf{H}^{[k+1]}\mathbf{s}_k = \mathbf{y}_k \tag{3.2}$$

where $\mathbf{s}_k = \boldsymbol{\theta}^{[k+1]} - \boldsymbol{\theta}^{[k]}$ and $\mathbf{y}_k = \nabla f(\boldsymbol{\theta}^{[k+1]}) - \nabla f(\boldsymbol{\theta}^{[k]})$. The so-called *secant equation* (3.2) will only be feasible for positive definite $\mathbf{H}^{[k+1]}$ under certain conditions on $\mathbf{s}_k$ and $\mathbf{y}_k$, but we can always arrange for these to be met by appropriate step length selection (see Wolfe conditions, Nocedal & Wright, 2006, §3.1). Note, in particular, that pre-multiplying (3.2) by $\mathbf{s}_k^{\mathsf{T}}$ gives $\mathbf{s}_k^{\mathsf{T}}\mathbf{y}_k = \mathbf{s}_k^{\mathsf{T}}\mathbf{H}^{[k+1]}\mathbf{s}_k > 0$ for positive definite Hessian, so $\mathbf{s}_k^{\mathsf{T}}\mathbf{y}_k$ must be positive. Still, the secant equation is under-determined, so we need to constrain the problem further in order to identify an appropriate unique solution. For quasi Newton methods, this is typically done by proposing a low-rank update to the previous Hessian. Methods exist which use a rank 1 update, but BFGS uses a rank 2 update of the form

$$\mathbf{H}^{[k+1]} = \mathbf{H}^{[k]} + \alpha\mathbf{u}\mathbf{u}^{\mathsf{T}} + \beta\mathbf{v}\mathbf{v}^{\mathsf{T}}.$$

[†]BFGS is named after Broyden, Fletcher, Goldfarb and Shanno all of whom independently discovered and published it, around 1970. Big Friendly Giant Steps is the way all Roald Dahl readers remember the name, of course (M.V.Bravington, pers. com.).

For (good) reasons that we don't have time to explore here, the choice $\mathbf{u} = \mathbf{y}_k$, $\mathbf{v} = \mathbf{H}^{[k]}\mathbf{s}_k$ is made, and substituting this update into the secant equation allows solving for $\alpha$ and $\beta$, giving the BFGS update,

$$\mathbf{H}^{[k+1]} = \mathbf{H}^{[k]} + \rho_k \mathbf{y}_k \mathbf{y}_k^\mathsf{T} - \frac{\mathbf{H}^{[k]}\mathbf{s}_k\mathbf{s}_k^\mathsf{T}\mathbf{H}^{[k]}}{\mathbf{s}_k^\mathsf{T}\mathbf{H}^{[k]}\mathbf{s}_k},$$

where $\rho_k = 1/\mathbf{s}_k^\mathsf{T}\mathbf{y}_k$. By substitution, we can directly verify that this satisfies the secant equation. Now let's work in terms of the inverse approximate Hessian, $\mathbf{B}^{[k]} \equiv \left(\mathbf{H}^{[k]}\right)^{-1}$, since this will allow us to avoid solving a linear system at each iteration. A naive re-write of our update would be

$$\mathbf{B}^{[k+1]} = \left[(\mathbf{B}^{[k]})^{-1} + \rho_k \mathbf{y}_k \mathbf{y}_k^\mathsf{T} - \frac{(\mathbf{B}^{[k]})^{-1}\mathbf{s}_k\mathbf{s}_k^\mathsf{T}(\mathbf{B}^{[k]})^{-1}}{\mathbf{s}_k^\mathsf{T}(\mathbf{B}^{[k]})^{-1}\mathbf{s}_k}\right]^{-1},$$

but this is a low-rank update of an inverse, and hence a good candidate for application of Woodbury's formula. It is actually a bit tedious to apply Woodbury directly, but if we can guess the solution,

$$\mathbf{B}^{[k+1]} = (\mathbf{I} - \rho_k \mathbf{s}_k \mathbf{y}_k^\mathsf{T})\mathbf{B}^{[k]}(\mathbf{I} - \rho_k \mathbf{y}_k \mathbf{s}_k^\mathsf{T}) + \rho_k \mathbf{s}_k \mathbf{s}_k^\mathsf{T},$$

we can verify that it is correct by direct multiplication. We can also double-check that this update satisfies the modified secant equation, $\mathbf{B}^{[k+1]}\mathbf{y}_k = \mathbf{s}_k$ (it does). The BFGS update is usually given in the form above, but for actually implementation on a computer, it can be expanded as

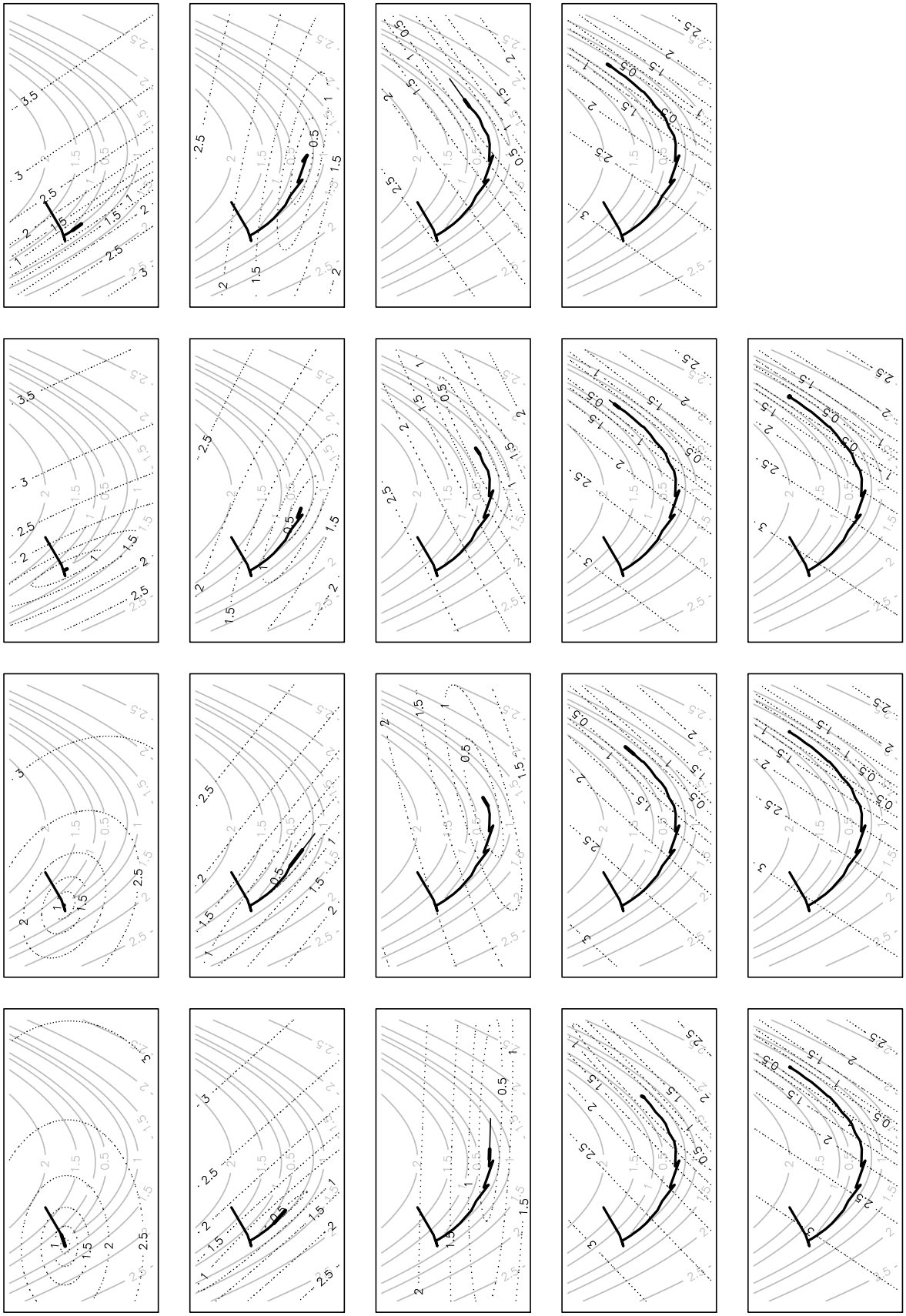
$$\mathbf{B}^{[k+1]} = \mathbf{B}^{[k]} - \rho_k \left[\mathbf{s}_k \mathbf{y}_k^\mathsf{T}\mathbf{B}^{[k]} + \mathbf{B}^{[k]}\mathbf{y}_k\mathbf{s}_k^\mathsf{T}\right] + \rho_k(1 + \rho_k \mathbf{y}_k^\mathsf{T}\mathbf{B}^{[k]}\mathbf{y}_k)\mathbf{s}_k\mathbf{s}_k^\mathsf{T},$$

and then some careful thinking about the order and re-use of computational steps within the update can lead to a very efficient algorithm.

The BFGS method then works exactly like Newton's method, but with $\mathbf{B}^{[k]}$ in place of the inverse of $\nabla^2 f(\boldsymbol{\theta}^{[k]})$, and without the need for second derivative evaluation or for perturbing the Hessian to achieve positive definiteness. It also allows direct computation of the search direction via

$$\boldsymbol{\Delta} = -\mathbf{B}^{[k]}\nabla f(\boldsymbol{\theta}^k).$$

A finite difference approximation to the Hessian is often used to start the method. The following figure illustrates its application (showing every second step of 38).



Notice how this quasi-Newton method has taken nearly twice as many steps as the Newton method, for the same problem, and shows a greater tendency to 'zig-zag'. However, it is still computationally cheaper than finite differencing for the Hessian, which would have required an extra 20 gradient evaluations, compared to quasi-Newton. Compared to steepest descent it's miraculous.

**L-BFGS**

It is also worth being aware of a variant of BFGS, known as L-BFGS, with the "L" standing for *limited memory*. This algorithm is mainly intended for very high-dimensional functions, where working with a full inverse Hessian would be an issue for either computational or memory reasons. Here, rather than propagating the full inverse Hessian matrix at each step, a low rank approximation of it is implicitly updated and propagated via the recent history of the process. The following Wikipedia pages provide a little additional context.

- `https://en.wikipedia.org/wiki/BFGS`
- `https://en.wikipedia.org/wiki/L-BFGS`
- `https://en.wikipedia.org/wiki/Wolfe_conditions`

### 3.4.5 The Nelder–Mead polytope method

What if even gradient evaluation is too taxing, or if our objective is not smooth enough for Taylor approximations to be valid (or perhaps even possible)? What can be done with function values alone? The Nelder–Mead polytope[‡] method provides a rather successful answer, and as with the other methods we have met, it is beautifully simple.

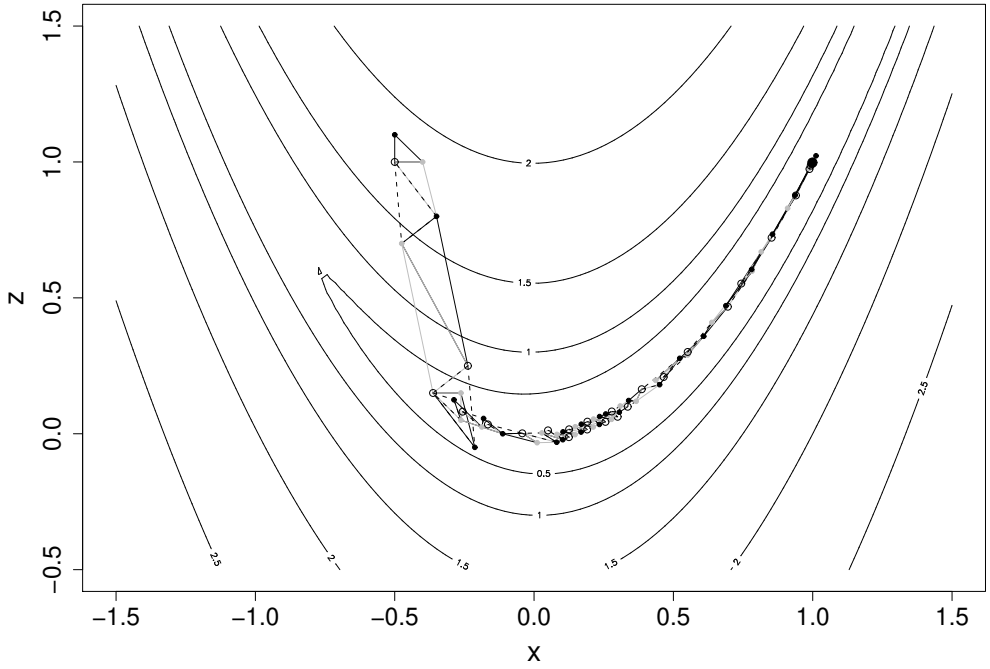
Let $n$ be the dimension of $\boldsymbol{\theta}$. At each stage of the method we maintain $n+1$ distinct $\boldsymbol{\theta}$ vectors, defining a polytope in $\Theta$ (e.g. for a 2-dimensional $\boldsymbol{\theta}$, the polytope is a triangle). The following is iterated until a minimum is reached/ the polytope collapses to a point.

1. The search direction is defined as the vector from the worst point (vertex of the polytope with highest objective value) through the average of the remaining $n$ points.
2. The initial step length is set to twice the distance from the worst point to the centroid of the others. If it succeeds (meaning that the new point is no longer the worst point) then a step length of 1.5 times that is tried, and the better of the two accepted.
3. If the previous step did not find a successful new point then step lengths of half and one and a half times the distance from the worst point to the centroid are tried.
4. If the last two steps failed to locate a successful point then the polytope is reduced in size, by linear rescaling towards the current best point (which remains fixed.)
5. When a point is accepted, the current worst point is discarded, to maintain $n+1$ points, and the algorithm returns to step 1.

Variations are possible, in particular with regard to the step lengths and shrinkage factors.

The following figure illustrates the polytope method in action. Each polytope is plotted, with the line style cycling through, black, grey and dashed black. The worst point in each polytope is highlighted with a circle.

[‡]This is also known as the *downhill simplex method*, which should not be confused with the completely different *simplex method* of linear programming.




In this case it took 79 steps to reach the minimum of Rosenbrock's function. This is somewhat more than Newton or BFGS, but given that we need no derivatives in this case, the amount of computation is actually less.

On the basis of this example you might be tempted to suppose that Nelder-Mead is all you ever need, but this is generally not the case. If you need to know the optimum very accurately then Nelder-Mead will often take a long time to get an answer that Newton based methods would give very quickly. Also, the polytope can get 'stuck', so that it is usually a good idea to restart the optimisation from any apparent minimum (with a new polytope having the apparent optimum as one vertex), to check that further progress is really not possible. Essentially, Nelder-Mead is good if the answer does not need to be too accurate, and derivatives are hard to come by, but you wouldn't use it as the underlying optimiser for general purpose modelling software, for example.

### 3.4.6 Other methods

These notes can only provide a brief overview of some important methods. There are many other optimisation methods, and quite a few are actually useful. For example:

- In likelihood maximisation contexts *the method of scoring* is often used. This replaces the Hessian in Newton's method with the expected Hessian.
- The *Gauss-Newton method* is another way of getting a Newton type method when only first derivatives are available. It is applicable when the objective is somehow related to a sum of squares of differences between some data and the model predictions of those data, and has a nice interpretation in terms of approximating linear models. It is closely related to the algorithm for fitting GLMs.
- *Conjugate gradient methods* are another way of getting a good method using only first derivative information. The problem with steepest descent is that if you minimise along one steepest descent direction then the next search direction is *always* at right angles to it. This can mean that it takes a huge number of tiny zig-zagging steps to minimise even a perfectly quadratic function. If $n$ is the dimension of $\boldsymbol{\theta}$, then Conjugate Gradient methods aim to construct a set of $n$ consecutive search directions such that successively minimising along each will lead you exactly to the minimum of a quadratic function.

- *Simulated annealing* is a gradient-free method for optimising difficult objectives, such as those arising in discrete optimisation problems, or with very spiky objective functions. The basic idea is to propose random changes to $\boldsymbol{\theta}$, always accepting a change which reduces $f(\boldsymbol{\theta})$, but also accepting moves which increase the objective, with a probability which decreases with the size of increase, and also decreases as the optimisation progresses.
- *Stochastic gradient descent* (SGD) is an increasingly popular approach to optimisation of likelihoods arising from very large data sets. Each step of SGD is a gradient descent based on a new objective derived from a random subset of the data. The stochastic aspect can speed up convergence and the use of a relatively small subset of the data can speed up each iteration: `https://en.wikipedia.org/wiki/Stochastic_gradient_descent`
- The *EM algorithm* is a useful approach to likelihood maximisation when there are missing data or random effects that are difficult to integrate out.

## 3.5 Constrained optimisation

Sometimes optimisation must be performed subject to constraints on $\boldsymbol{\theta}$. An obvious way to deal with constraints is to re-parameterise to arrive at an unconstrained problem. For example if $\theta_1 > 0$ we might choose to work in terms of the unconstrained $\theta_1'$ where $\theta_1 = \exp(\theta_1')$, and more complicated re-parameterisations can impose quite complex constraints in this way. While often effective re-parameterisation suffers from at least three problems:

1. It is not possible for all constraints that might be required.
2. The objective is sometimes a more awkward function of the new parameters than it was of the old.
3. There can be problems with the unconstrained parameter estimates tending to $\pm\infty$. For example is $\theta_1$ above is best estimated to be zero then the best estimate of $\theta_1'$ is $-\infty$.

A second method is to add to the objective function a penalty function penalising violation of the constraints. The strength of such a penalty has to be carefully chosen, and of course the potential to make the objective more awkward than the original applies here too.

The final set of approaches treat the constraints directly. For *linear* equality and inequality constraints constrained Newton type methods are based on the methods of *quadratic programming*. Non-linear constraints are more difficult, but can often be discretised into a set of approximate linear constraints.

## 3.6 Modifying the objective

If you have problems optimising an objective function, then sometimes it helps to modify the function itself. For example:

- Reparameterisation can turn a very unpleasant function into something much easier to work with (e.g. it is often better to work with precision than with standard deviation when dealing with normal likelihoods).
- Transform the objective. e.g. $\log(f)$ will be 'more convex' than $f$: this is usually a good thing.
- Consider perturbing the data. For example an objective based on a small sub sample of the data can sometimes be a good way of finding starting values for the full optimisation. Similarly resampling the data can provide a means for escaping small scale local minima in an objective if the re-sample based objective is alternated with the true objective as optimisation progresses, or if the objective is averaged over multiple re-samples. cf. SGD.
- Sometimes a problem with optimisation indicates that the objective is not a sensible one. e.g. in the third example in section 3.3, an objective based on getting the observed ACF right makes more sense, and is much better behaved.

## 3.7 Software

R includes a routine `optim` for general purpose optimisation by BFGS, Nelder-Mead, Conjugate Gradient or Simulated Annealing. The R routine `nlm` uses Newton's method and will use careful finite difference approximations for derivatives which you don't supply. R has a number of add on packages e.g. `quadprog`, `linprog`, `trust` to name but 3. See `http://neos-guide.org/` for a really useful guide to what is more generally available.

Should you ever implement your own optimisation code? Many people reply with a stern 'no', but I am less sure. For the methods presented in detail above, the advantages of using your own implementation are (i) the un-rivalled access to diagnostic information and (ii) the knowledge of exactly what the routine is doing. However, if you do go down this route it is *very* important to read further about *step length selection* and *termination criteria*. Where greater caution is needed is if you require constrained optimisation, large scale optimisation or to optimise based on approximate derivatives. All these areas require careful treatment.

## 3.8 Further reading on optimisation

- Nocedal and Wright (2006) *Numerical Optimization* 2nd ed. Springer, is a very clear and up to date text. If you are only going to look at one text, then this is probably the one to go for.
- Gill, P.E. , W. Murray and M.H. Wright (1981) *Practical Optimization* Academic Press, is a classic, particularly on constrained optimisation. I've used it to successfully implement quadratic programming routines.
- Dennis, J.E. and R.B. Schnabel (1983) *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (re-published by SIAM, 1996), is a widely used monograph on unconstrained methods (the `nlm` routine in R is based on this one).
- Press WH, SA Teukolsky, WT Vetterling and BP Flannery (2007) *Numerical Recipes (3rd ed.)*, Cambridge and Monahan, JF (2001) *Numerical Methods of Statistics*, Cambridge, both discuss Optimization at a more introductory level.

# Chapter 4

# Calculus by computer

As we saw in the previous section, almost any time that we want to maximise a likelihood, we need to differentiate it. Very often, merely evaluating a likelihood involves integration (to obtain the marginal distribution of the observable data). Unsurprisingly, there are times when it is useful to perform these tasks by computer. Broadly speaking, numerical integration is computationally costly but stable, while numerical differentiation is cheap but potentially unstable. We will also consider the increasingly important topic of 'automatic differentiation'.

## 4.1 Cancellation error

Consider the following R code

```
a <- 1e16
b <- 1e16 + pi
d <- b - a
```

so `d` should be $\pi$, right? Actually

```
  d; pi

## [1] 4
## [1] 3.141593
```

i.e. `d` is out by 27% as the following shows...

```
  (d-pi)/pi  ## relative error in difference is huge...

## [1] 0.2732395
```

This is an example of *cancellation error*. Almost any time we take the difference of two floating point numbers of similar magnitude (and sign) we find that the accuracy of the result is substantially reduced. The problem is rounding error. You can see what happens by considering computation to 16 places of decimals. Then

$$a = 1.0000000000000000 \times 10^{16} \quad \text{and} \quad b = 1.0000000000000003 \times 10^{16}$$

hence $d = b - a = 3$. (`d` was 4, above, because the computer uses a binary rather than a decimal representation).

As we will see, derivative approximation is an area where we can not possibly ignore cancellation error, but actually it is a pervasive issue. For example, cancellation error is the reason why $\sum_i x_i^2 - n\bar{x}^2$ is not a good basis for a 'one-pass' method for calculating $\sum_i (x_i - \bar{x})^2$. $\sum_i (x_i - x_1)^2 - n(\bar{x} - x_1)^2$ is a simple one pass alternative, and generally *much* better. It is also a substantial issue in numerical linear algebra.

For example, orthogonal matrix decompositions (used for QR, eigen and singular value decompositions), usually involve an apparently arbitrary choice at each successive rotation involved in the decomposition, about whether some element of the result of rotation should be positive or negative: one of the choices almost always involves substantially lower cancellation error than the other, and is the one taken.

Since approximation of derivatives inevitably involves subtracting similar numbers, it is self evident that cancellation error will cause some difficulty. The basic reason why numerical integration is a more stable process (at least for statisticians who usually integrate only positive functions) is because numerical integration is basically about adding things up, rather than differencing them, and this is more stable. e.g.

```
f <- a + b
(f-2e16-pi)/(2e16+pi) ## relative error in sum, is tiny...

## [1] 4.292037e-17
```

## 4.2 Approximate derivatives by finite differencing

Consider differentiating a sufficiently smooth function $f(\mathbf{x})$ with respect to the elements of its vector argument $\mathbf{x}$. $f$ might be something simple like $\sin(x)$ or something complicated like the mean global temperature predicted by an atmospheric GCM, given an atmospheric composition, forcing conditions etc.

A natural way to approximate the derivatives is to use:

$$\frac{\partial f}{\partial x_i} \simeq \frac{f(\mathbf{x}+\Delta \mathbf{e}_i) - f(\mathbf{x})}{\Delta}$$

where $\Delta$ is a small constant and $\mathbf{e}_i$ is a vector of the same dimension as $\mathbf{x}$, with zeroes for each element except the $i^{\text{th}}$, which is 1. In a slightly sloppy notation, Taylor's theorem tells us that

$$f(\mathbf{x}+\Delta \mathbf{e}_i) = f(\mathbf{x}) + \nabla f(\mathbf{x})^{\mathsf{T}} \mathbf{e}_i \Delta + \frac{1}{2}\Delta^2 \mathbf{e}_i^{\mathsf{T}} \nabla^2 f \mathbf{e}_i$$

Re-arranging while noting that $\nabla f(\mathbf{x})^{\mathsf{T}} \mathbf{e}_i = \partial f/\partial x_i$ we have

$$\frac{f(\mathbf{x}+\Delta \mathbf{e}_i) - f(\mathbf{x})}{\Delta} - \frac{\partial f}{\partial x_i} = \frac{1}{2}\Delta \mathbf{e}_i^{\mathsf{T}} \nabla^2 f \mathbf{e}_i.$$

Now suppose that $L$ is an upper bound on the magnitude of $\partial^2 f/\partial x_i^2$, we have

$$\left| \frac{f(\mathbf{x}+\Delta \mathbf{e}_i) - f(\mathbf{x})}{\Delta} - \frac{\partial f}{\partial x_i} \right| \leq \frac{L\Delta}{2}.$$

That is to say, we have an upper bound on the finite difference *truncation** error, assuming that the FD approximation is calculated exactly, without round-off error.

In reality we have to work in finite precision arithmetic and can only calculate $f$ to some accuracy. Suppose that we can calculate $f$ to one part in $\epsilon^{-1}$ (the best we could hope for here is that $\epsilon$ is the machine precision). Now let $L_f$ be an upper bound on the magnitude of $f$, and denote the computed value of $f$ by $\texttt{comp}(f)$. We have

$$|\texttt{comp}\{f(\mathbf{x})\} - f(\mathbf{x})| \leq \epsilon L_f$$

and

$$|\texttt{comp}\{f(\mathbf{x}+\Delta \mathbf{e}_i)\} - f(\mathbf{x}+\Delta \mathbf{e}_i)| \leq \epsilon L_f$$

combining the preceding two inequalities we get

$$\left| \frac{\texttt{comp}\{f(\mathbf{x}+\Delta \mathbf{e}_i) - f(\mathbf{x})\}}{\Delta} - \frac{f(\mathbf{x}+\Delta \mathbf{e}_i) - f(\mathbf{x})}{\Delta} \right| \leq \frac{2\epsilon L_f}{\Delta},$$

*So called because it is the error associated with *truncating* the Taylor series approximation to the function.

an upper bound on the *cancellation* error that results from differencing two very similar quantities using finite precision arithmetic. (It should now be clear what the problem was in section 1.1 example 2.)

Now clearly the dependence of the two types of error on $\Delta$ is rather different. We want $\Delta$ as small as possible to minimise truncation error, and as large as possible to minimise cancellation error. Given that the total error is bounded as follows,

$$\text{err.fd} \leq \frac{L\Delta}{2} + \frac{2\epsilon L_f}{\Delta}$$

it makes sense to chose $\Delta$ to minimise the bound. That is we should choose

$$\Delta \approx \sqrt{\frac{4\epsilon L_f}{L}}.$$

So **if** the typical size of $f$ and its second derivatives are similar then

$$\Delta \approx \sqrt{\epsilon}$$

will not be too far from optimal. This is why the square root of the machine precision is often used as the finite difference interval. However, note the assumptions: we require that $L_f \approx L$ *and* $f$ to be calculable to a relative accuracy that is a small multiple of the machine precision for such a choice to work well. In reality problems are not always *well scaled* and a complicated function may have a relative accuracy substantially lower than is suggested by the machine precision. In such cases, see section 8.6 of Gill, Murray and Wright (1981), or section 4.3.3 below.

### 4.2.1 Other FD formulae

The FD approach considered above is *forward differencing*. Centred differences are more accurate, but more costly...

$$\frac{\partial f}{\partial x_i} \simeq \frac{f(\mathbf{x} + \Delta \mathbf{e}_i) - f(\mathbf{x} - \Delta \mathbf{e}_i)}{2\Delta}$$

...in the well scaled case $\Delta \approx \epsilon^{1/3}$ is about right.

Higher order derivatives can also be useful. For example

$$\frac{\partial^2 f}{\partial x_i \partial x_j} \simeq \frac{f(\mathbf{x} + \Delta \mathbf{e}_i + \Delta \mathbf{e}_j) - f(\mathbf{x} + \Delta \mathbf{e}_i) - f(\mathbf{x} + \Delta \mathbf{e}_j) + f(\mathbf{x})}{\Delta^2}$$

which in the well scaled case will be most accurate for $\Delta \approx \epsilon^{1/4}$.

## 4.3 Automatic Differentiation: code that differentiates itself

Differentiation of complex models or other expressions is tedious, but ultimately routine. One simply applies the chain rule and some known derivatives, repeatedly. The main problem with doing this is the human error rate and the limits on human patience. Given that differentiation is just the repeated application of the chain rule and known derivatives, couldn't it be automated? The answer is yes. There are two approaches. The first is symbolic differentiation using a computer algebra package such as Maple or Mathematica. This works, but can produce enormously unwieldy expressions when applied to a complex function. The problem is that the number of terms in a symbolic derivative can be very much larger than for the expression being evaluated. It is also very difficult to apply to a complex simulation model, for example.

The second approach is automatic differentiation. This operates by differentiating a function based directly on the computer code that evaluates the function. There are several different approaches to this, but many use the features of *object oriented* programming languages to achieve the desired end. The key feature of an object oriented language, from the AD perspective, is that every data structure, or `object` in such a language has a *class* and the meaning of operators such as `+`, `-`, `*` etc depends on the class of the objects to which they are applied. Similarly the action of a function depends on the class of its arguments.

Suppose then, that we would like to differentiate

$$f(x_1, x_2, x_3) = (x_1 x_2 \sin(x_3) + e^{x_1 x_2}) / x_3$$

w.r.t. its real arguments $x_1$, $x_2$ and $x_3$.† In R the code

```
(x1*x2*sin(x3) + exp(x1*x2))/x3
```

would evaluate the function, if `x1`, `x2` and `x3` were initialised to be floating point numbers.

Now define a new type of object of class `"ad"` which has a value (a floating point number) and a `"grad"` attribute. In the current case this `"grad"` attribute will be a 3-vector containing the derivatives of the value w.r.t. `x1`, `x2` and `x3`. We can now define versions of the arithmetic operators and mathematical functions which will return class `"ad"` results with the correct value and `"grad"` attribute, whenever they are used in an expression.

Here is an R function to create and initialise a simple class `"ad"` object

```
ad <- function(x,diff = c(1,1)) {
## create class "ad" object. diff[1] is length of grad
## diff[2] is element of grad to set to 1.
  grad <- rep(0,diff[1])
  if (diff[2]>0 && diff[2]<=diff[1]) grad[diff[2]] <- 1
  attr(x,"grad") <- grad
  class(x) <- "ad"
  x
}
```

Here it is in use, initialising `x1` to the value 1, giving it a 3 dimensional `"grad"` attribute, and setting the first element of `grad` to 1 since $\partial x_1 / \partial x_1 = 1$.

```
  x1 <- ad(1,c(3,1))
  x1

## [1] 1
## attr(,"grad")
## [1] 1 0 0
## attr(,"class")
## [1] "ad"
```

Now the interesting part. I can define versions of mathematical functions and operators that are specific to class `"ad"` objects and correctly propagate derivatives alongside values. Here is a sin function for class `"ad"`.

```
sin.ad <- function(a) {
  grad.a <- attr(a,"grad")
  a <- as.numeric(a) ## avoid infinite recursion - change class
  d <- sin(a)
  attr(d,"grad") <- cos(a) * grad.a ## chain rule
  class(d) <- "ad"
  d
}
```

and here is what happens when it is applied to `x1`

†This example is taken from Nocedal and Wright, (2006).

```
sin(x1)

## [1] 0.841471
## attr(,"grad")
## [1] 0.5403023 0.0000000 0.0000000
## attr(,"class")
## [1] "ad"
```

i.e. the value of the result is $\sin(x_1)$, while the first element of its `"grad"` contains the derivative of $\sin(x_1)$ w.r.t. $x_1$ evaluated at $x_1 = 1$.

Operators can also be *overloaded* in this way. e.g. here is multiplication operator for class `"ad"`:

```
"*.ad" <- function(a,b) { ## ad multiplication
  grad.a <- attr(a,"grad")
  grad.b <- attr(b,"grad")
  a <- as.numeric(a)
  b <- as.numeric(b)
  d <- a*b    ## evaluation
  attr(d,"grad") <- a * grad.b + b * grad.a ## product rule
  class(d) <- "ad"
  d
}
```

Continuing in the same way we can provide a complete library of mathematical functions and operators for the `"ad"` class. Given such a library, we can obtain the derivatives of a function directly from the code that would simply evaluate it, given ordinary floating point arguments. For example, here is some code evaluating a function.

```
  x1 <- 1; x2 <- 2; x3 <- pi/2
  (x1*x2*sin(x3)+ exp(x1*x2))/x3

## [1] 5.977259
```

and here is the same code with the arguments replaced by `"ad"` objects

```
  x1 <- ad(1, c(3,1))
  x2 <- ad(2, c(3,2))
  x3 <- ad(pi/2, c(3,3))
  (x1*x2*sin(x3) + exp(x1*x2))/x3

## [1] 5.977259
## attr(,"grad")
## [1] 10.681278  5.340639 -3.805241
## attr(,"class")
## [1] "ad"
```

you can check that these results are correct (actually to machine accuracy).

This simple propagation of derivatives alongside the evaluation of a function is known as *forward* mode auto-differentiation. R is not the best language in which to try to do this, and if you need AD it is much better to use existing software libraries in C++, for example, which have done all the function and operator re-writing for you.

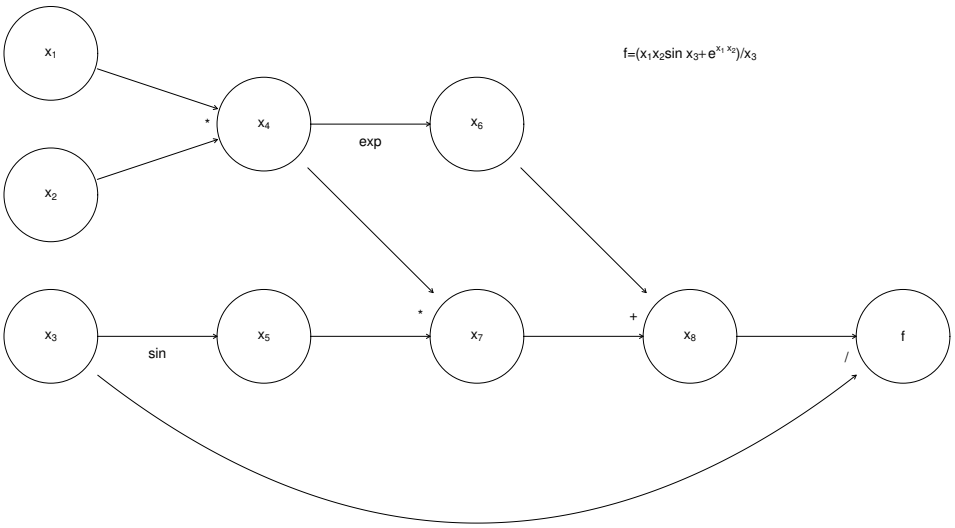
### 4.3.1 Reverse mode AD

If you require many derivatives of a scalar valued function then forward mode AD will have a theoretical computational cost similar to finite differencing, since at least as many operations are required for *each* derivative as are required for function evaluation. In reality the overheads associated with operator overloading make AD more expensive (alternative strategies also carry overheads). Of course the benefit of AD is higher accuracy, and in many applications the cost is not critical.

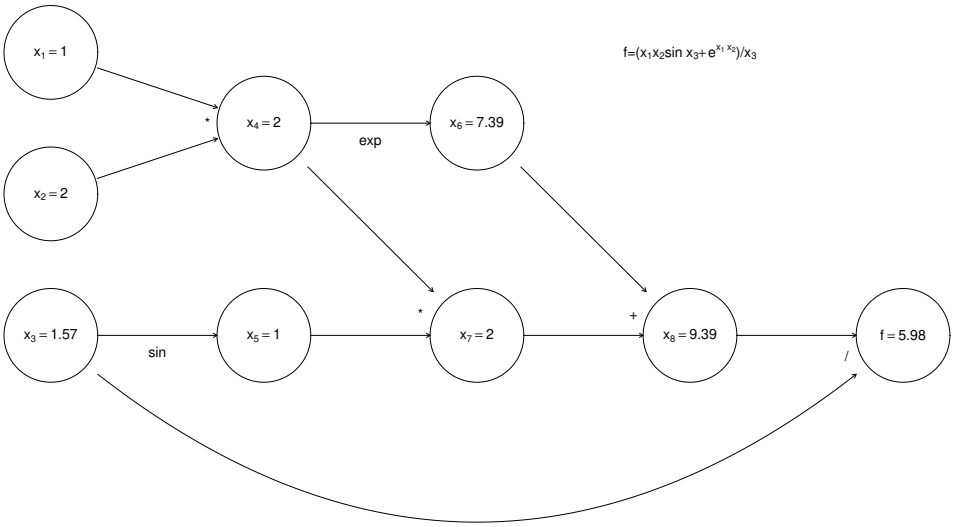
An alternative with the potential for *big* computational savings over finite differencing is *reverse* mode AD. To understand this it helps to think of the evaluation of a function in terms of a *computational graph*. Again concentrate on the example function given in Nocedal and Wright (2006):

$$f(x_1, x_2, x_3) = \left(x_1 x_2 \sin(x_3) + e^{x_1 x_2}\right) / x_3$$

Any computer evaluating this must break the computation down into a sequence of elementary operations on one or two floating point numbers. This can be thought of as a graph:



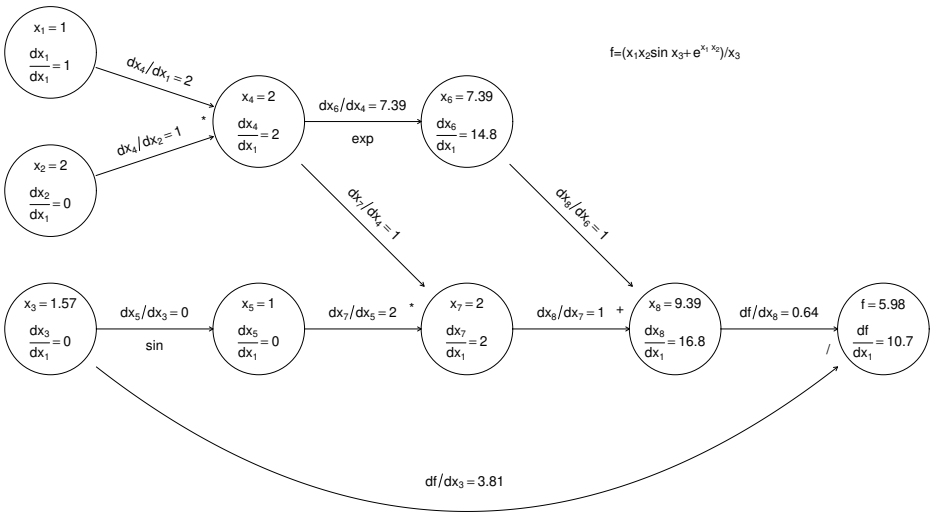

where the nodes $x_4$ to $x_8$ are the intermediate quantities that will have to be produced en route from the input values $x_1$ to $x_3$ to the final answer, $f$. The arrows run from *parent* nodes to *child* nodes. No child can be evaluated until all its parents have been evaluated.
Simple left to right evaluation of this graph results in this ...




Now, forward mode AD carries derivatives forward through the graph, alongside values. e.g. the derivative of a node with respect to input variable, $x_1$ is computed using

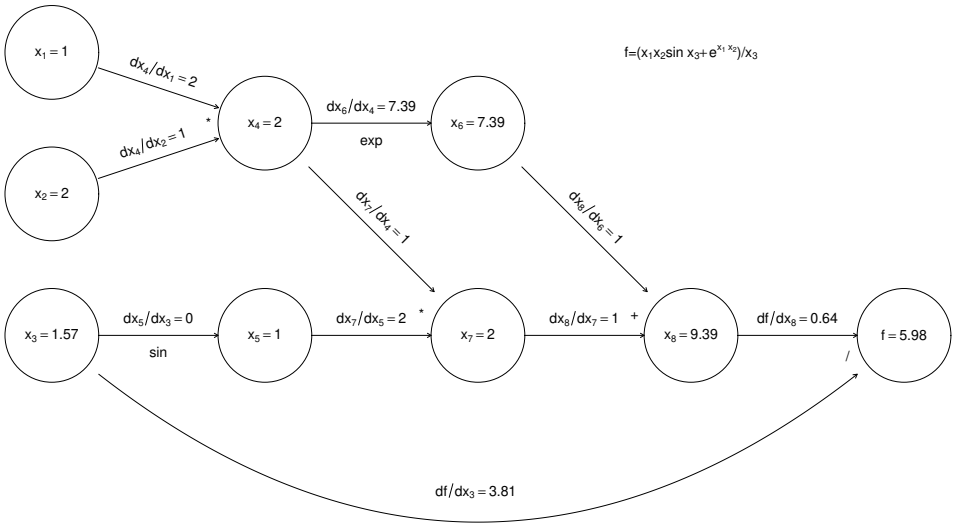
$$\frac{\partial x_k}{\partial x_1} = \sum_{j \text{ parent of } k} \frac{\partial x_k}{\partial x_j} \frac{\partial x_j}{\partial x_1}$$

(the r.h.s. being evaluated by overloaded functions and operators, in the object oriented approach). The following illustrates this process, just for the derivative w.r.t. $x_1$.




. . . Again computation runs left to right, with evaluation of a node only possible once all parent values are known.

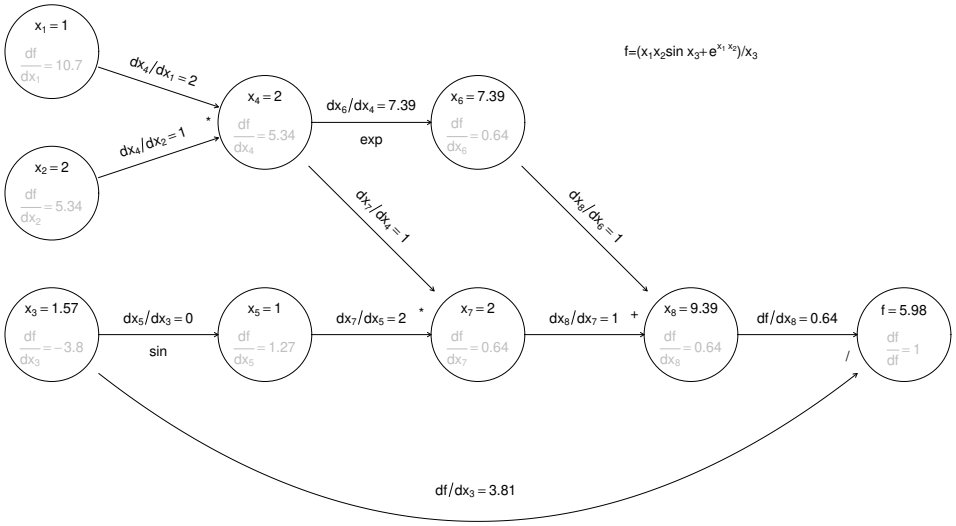
Now, if we require derivatives w.r.t. several input variables, then each node will have to evaluate derivatives w.r.t. each of these variables, and this becomes expensive (in terms of the previous graph, each node would contain multiple evaluated derivatives). Reverse mode therefore does something ingenious. It first executes a *forward sweep* through the graph, evaluating the function and all the derivatives of nodes w.r.t. their parents, as follows:




The *reverse* sweep then works backwards through the graph, evaluating the derivative of $f$ w.r.t. each node, using the following

$$\frac{\partial f}{\partial x_k} = \sum_{j \text{ is child of k}} \frac{\partial x_j}{\partial x_k} \frac{\partial f}{\partial x_j}$$

and the fact that at the terminal node $\partial f / \partial f = 1$.




The derivatives in grey are those calculated on the reverse sweep. The point here is that there is only one derivative to be evaluated at each node, but in the end we know the derivative of $f$ w.r.t. every input variable. Reverse mode can therefore save a large number of operations relative to finite differencing or forward mode. Once again, general purpose AD libraries automate the process for you, so that all you need to be able to write is the evaluation code.

Unfortunately reverse mode efficiency comes at a heavy price. In forward mode we could discard the values and derivatives associated with a node as soon as all its children were evaluated. In reverse mode the values of *all nodes* and the evaluated derivatives associated with *every connection* have to be stored during the forward sweep, in order to be used in the reverse sweep. This is a heavy storage requirement.

For example, if $f$ involved the inversion of a $1000 \times 1000$ matrix then we would have to store some $2 \times 10^9$ intermediate node values plus a similar number of evaluated derivatives. That is some 32 Gigabytes of storage before we even consider the requirements for storing the *structure* of the graph.

### 4.3.2 A caveat

For AD to work it is not sufficient that the function being evaluated has properly defined derivatives at the evaluated function value. We require that every function/operator used in the evaluation has properly defined derivatives at its evaluated argument(s). This can create a problem with code that executes conditional on the value of some variable. For example the Box-Cox transformation of a positive datum $y$ is

$$B(y;\lambda) = \begin{cases} (y^\lambda - 1)/\lambda & \lambda \neq 0 \\ \log(y) & \lambda = 0 \end{cases}$$

If you code this up in the obvious way then AD will never get the derivative of $B$ w.r.t. $\lambda$ right if $\lambda = 0$.

### 4.3.3 Using AD to improve FD

When fitting complicated or computer intensive models it can be that AD is too expensive to use for routine derivative calculation during optimisation. However, with the gradual emergence of highly optimised libraries, this situation is changing, and in any case it can still provide a useful means for calibrating FD intervals. A 'typical' model run can be auto-differentiated, and the finite difference intervals adjusted to achieve the closest match to the AD derivatives. As optimisation progresses one or two further calibrations of the FD intervals can be carried out as necessary.

## 4.4 Numerical Integration

Integration is at least as important as optimisation in statistics. Whether integrating random effects out of a joint distribution to get a likelihood, or just evaluating a simpler expectation, it occurs everywhere. Unfortunately integration is generally numerically expensive, and much statistical research is devoted to clever ways of avoiding it. When it can't be avoided then the game is typically to find ways of approximating an integral of a function with the minimum number of function evaluations.

- Integration w.r.t. 1 or 2 (maybe even up to 4) variables is not too problematic. There are two 'deterministic' lines of attack.
  1. Use differential equation solving methods since, for example
     $$y = \int_a^b f(x)dx$$
     is the solution of
     $$\frac{\mathrm{d}y}{\mathrm{d}x} = f(x), \quad y(a) = 0$$
     integrated to $b$.
  2. Quadrature rules. These evaluate the function to be integrated at some set of points in the integration domain and combine the resulting values to obtain an approximation to the integral. For example
     $$\int_a^b f(x)dx \approx \frac{b-a}{N}\sum_{i=1}^{N} f(a + (i - .5)(b-a)/N)$$
     is the *Midpoint rule* for approximate integration. Simpson's Rule is a higher order variant, while *Gaussian Quadrature* rules aim for substantially higher order accuracy by careful choice of the locations at which to evaluate.

- Integration with respect to several-many variables is usually more interesting in statistics and it is difficult. Maintenance of acceptable accuracy with quadrature rules or differential equation methods typically requires an unfeasible number of function evaluations for high dimensional domains of integration. There are two main approaches.

  1. Approximate the function to be integrated by one that can be integrated analytically.
  2. Take a statistical view of the integration process, and construct statistical estimators of the integral. Note that, for a fixed number of function evaluations $N$,
     (a) any bias (such as the error in a quadrature rule) scales badly with dimension.
     (b) any estimator variance need not depend on dimension at all.
     (c) as usual, we can completely eliminate either bias or variance, but not both.

     These considerations suggest using unbiased estimators of integrals, based on random samples from the function being integrated. The main challenge is then to design statistical sampling schemes and estimators which minimise the estimator variance.

### 4.4.1 Quadrature rules

In one dimension we might choose to approximate the integral of a continuous function $f(x)$ by the area under a set of step functions, with the midpoint of each matching $f$:

$$\int_a^b f(x)dx = \frac{b-a}{N}\sum_{i=1}^{N} f(a+(i-.5)(b-a)/N) + O(h^2)$$

where $h$ in the final error term is $(b-a)/N$. Let's see what the error bound means in practice by approximating $\int_0^1 e^x dx$.

```
  I.true <- exp(1) - 1
  N <- 10
  I.mp <- sum(exp((1:N-.5)/N))/N
  c(I.mp,I.true,(I.true-I.mp)/I.true)

## [1] 1.7175660865 1.7182818285 0.0004165452
```

A $.04\%$ error may be good enough for many purposes, but suppose we need higher accuracy...

```
  N <- 100
  I.mp <- sum(exp((1:N-.5)/N))/N
  c(I.mp,I.true,(I.true-I.mp)/I.true)

## [1] 1.718275e+00 1.718282e+00 4.166655e-06

  N <- 1000
  I.mp <- sum(exp((1:N-.5)/N))/N
  c(I.mp,I.true,(I.true-I.mp)/I.true)

## [1] 1.718282e+00 1.718282e+00 4.166667e-08
```

Exactly as the error bound indicated, each 10 fold increase in the number of function evaluations buys us a 100 fold reduction in the approximation error. Incidentally, notice that in statistical terms this error is not a random error — it is really a bias.

If $f$ is cheap to evaluate, accuracy is not paramount and we are only interested in 1 dimensional integrals then we might stop there (or more likely with the slightly more accurate but equally straightforward 'Simpson's Rule'), but life is not so simple. If one is very careful about the choice of $x$ points at which to

evaluate $f(x)$ *and* about the weights used in the integral approximating summation, then some very impressive results can be obtained, at least for smooth enough functions, using *Gauss(ian) Quadrature*. The basic idea is to find a set $\{x_i, w_i : i = 1, \ldots, n\}$ such that if $w(x)$ is a fixed weight function and $g(x)$ is *any* polynomial up to order $2n - 1$ then

$$\int_a^b g(x)w(x)dx = \sum_{i=1}^{n} w_i g(x_i)$$

Note that the $x_i, w_i$ set depends on $w(x)$, $a$ and $b$, but not on $g(x)$. The hope is that the r.h.s. will still approximate the l.h.s. very accurately if $g(x)$ is not a polynomial of the specified type, and there is some impressive error analysis to back this up, for any function with $2n$ well behaved derivatives (see e.g. Monahan, 2001, section 10.3).

The R package `statmod` has a nice `gauss.quad` function for producing the weights, $w_i$ and nodes, $x_i$, for some standard $w(x)$s. It assumes $a = -1$, $b = 1$, but we can just linearly re-scale our actual interval to use this. Here is the previous example using Gauss Quadrature (assuming $w(x)$ is a constant).

```
  library(statmod)
  N <- 10
  gq <- gauss.quad(N) # get the x_i, w_i
  gq # take a look

## $nodes
##  [1] -0.9739065 -0.8650634 -0.6794096 -0.4333954 -0.1488743  0.1488743
##  [7]  0.4333954  0.6794096  0.8650634  0.9739065
##
## $weights
##  [1] 0.06667134 0.14945135 0.21908636 0.26926672 0.29552422 0.29552422
##  [7] 0.26926672 0.21908636 0.14945135 0.06667134

  sum(gq$weights) # for an interval of width 2

## [1] 2

  I.gq <- sum(gq$weights*exp((gq$nodes+1)/2))/2
  c(I.gq,I.true,(I.true-I.gq)/I.true)

## [1]  1.718282e+00  1.718282e+00 -1.292248e-15
```

Clearly 10 evaluation points was a bit wasteful here

```
  N <- 5
  gq <- gauss.quad(N) ## get the x_i, w_i
  I.gq <- sum(gq$weights*exp((gq$nodes+1)/2))/2
  c(I.gq,I.true,(I.true-I.gq)/I.true)

## [1] 1.718282e+00 1.718282e+00 3.805670e-13
```

even 5 seems slightly wasteful

```
  N <- 3
  gq <- gauss.quad(N) ## get the x_i, w_i
  I.gq <- sum(gq$weights*exp((gq$nodes+1)/2))/2
  c(I.gq,I.true,(I.true-I.gq)/I.true)

## [1] 1.718281e+00 1.718282e+00 4.795992e-07
```

. . . still more accurate than the midpoint rule with 100 evaluations. This example should be treated with caution however, $e^x$ is a *very* smooth function and such impressive results will not hold for less smooth cases.

### 4.4.2 Quadrature rules for multidimensional integrals

We can integrate w.r.t. several variables by recursive application of quadrature rules. For example consider integrating $f(x, z)$ over a square region, and assume we use basically the same node and weight sequence for both dimensions. We have

$$\int f(x_j, z)dz \approx \sum_i w_i f(x_j, z_i)$$

and so

$$\int\int f(x, z)dzdx \approx \sum_j w_j \sum_i w_i f(x_j, z_i) = \sum_j \sum_i w_j w_i f(x_j, z_i)$$

Clearly, in principle we can repeat this for as many dimensions as we like. In practice however, to maintain a given level of accuracy, the number of function evaluations will have to rise as $N^d$ where $N$ is the number of nodes for each dimension, and $d$ is the number of dimensions. Even if we can get away with a 3 point Gaussian quadrature, we will need 1.5 million evaluations by the time we reach a 13 dimensional integral.

Here is a function which takes the node sequence `x` and weight sequence `w` of some quadrature rule and uses them to produce the `d` dimensional set of nodes, and the corresponding weight vector resulting from recursive application of the 1D rule.

```
mesh <- function(x,d,w=1/length(x)+x*0) {
  n <- length(x)
  W <- X <- matrix(0,n^d,d)
  for (i in 1:d) {
   X[,i] <- x;W[,i] <- w
   x <- rep(x,rep(n,length(x)))
   w <- rep(w,rep(n,length(w)))
  }
 w <- exp(rowSums(log(W))) ## column product of W gives weights
 list(X=X,w=w) ## each row of X gives co-ordinates of a node
}
```

Here's how it works.

```
mesh(c(1,2),3)
```

```
## $X
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    2    1    1
## [3,]    1    2    1
## [4,]    2    2    1
## [5,]    1    1    2
## [6,]    2    1    2
## [7,]    1    2    2
## [8,]    2    2    2
##
## $w
## [1] 0.125 0.125 0.125 0.125 0.125 0.125 0.125 0.125
```

Let's look at it in action on $f(\mathbf{x}) = \exp(\sum x_i)$, integrated over $[-1, 1]^d$.

```
fd <- function(x) {exp(rowSums(x))}
```

First try the midpoint rule over a two dimensional domain

```
  N <- 10; d <- 2; N^d # number of function evaluations

## [1] 100

  I.true <- (exp(1)-exp(-1))^d
  mmp <- mesh((1:N-.5)/N*2-1,d,rep(2/N,N))
  I.mp <- sum(mmp$w*fd(mmp$X))
  c(I.mp,I.true,(I.true-I.mp)/I.true)

## [1] 5.506013515 5.524391382 0.003326677
```

Now over a 5D domain

```
  N <- 10; d <- 5; N^d # number of function evaluations

## [1] 1e+05

  I.true <- (exp(1)-exp(-1))^d
  mmp <- mesh((1:N-.5)/N*2-1,d,rep(2/N,N))
  I.mp <- sum(mmp$w*fd(mmp$X))
  c(I.mp,I.true,(I.true-I.mp)/I.true)

## [1] 71.136612879 71.731695754  0.008295954
```

. . . which illustrates the enormous rate of increase in function evaluations required to achieve a rather disappointing accuracy.

Of course Gauss Quadrature is a better bet

```
  N <- 4; d <- 2; N^d

## [1] 16

  I.true <- (exp(1)-exp(-1))^d
  gq <- gauss.quad(N)
  mgq <- mesh(gq$nodes,d,gq$weights)
  I.gq <- sum(mgq$w*fd(mgq$X))
  c(I.gq,I.true,(I.true-I.gq)/I.true)

## [1] 5.524390e+00 5.524391e+00 2.511325e-07
```

with the increase in dimension being slightly less painful . . .

```
  N <- 4; d <- 5; N^d

## [1] 1024

  I.true <- (exp(1)-exp(-1))^d
  gq <- gauss.quad(N)
  mgq <- mesh(gq$nodes,d,gq$weights)
  I.gq <- sum(mgq$w*fd(mgq$X))
  c(I.gq,I.true,(I.true-I.gq)/I.true)

## [1] 7.173165e+01 7.173170e+01 6.278311e-07
```

... even so, by the time we reach 20 dimensions we would be requiring $10^{12}$ function evaluations, and most functions we might want to integrate will not be so smooth and well behaved that 4 quadrature points give us anything like the sort of accuracy seen in this example.

### 4.4.3 Approximating the integrand

Beyond a few dimensions, quadrature rules really run out of steam. In statistical work, where we are often interested in integrating probability density functions, or related quantities, it is sometimes possible to approximate the integrand by a function with a known integral. Here, just one approach will be illustrated: *Laplace approximation*. The typical application of the method is when we have a joint density $f(\mathbf{y}, \mathbf{b})$ and want to evaluate the marginal p.d.f. of $\mathbf{y}$,

$$f_y(\mathbf{y}) = \int f(\mathbf{y}, \mathbf{b}) d\mathbf{b}.$$

For a given $\mathbf{y}$ let $\hat{\mathbf{b}}_y$ be the value of $\mathbf{b}$ maximising $f$. Then Taylor's theorem implies that

$$\log\{f(\mathbf{y}, \mathbf{b})\} \simeq \log\{f(\mathbf{y}, \hat{\mathbf{b}}_y)\} - \frac{1}{2}(\mathbf{b} - \hat{\mathbf{b}}_y)^{\mathsf{T}} \mathbf{H} (\mathbf{b} - \hat{\mathbf{b}}_y)$$

where $\mathbf{H}$ is the Hessian of $-\log(f)$ w.r.t. $\mathbf{b}$ evaluated at $\mathbf{y}, \hat{\mathbf{b}}_y$. i.e.

$$f(\mathbf{y}, \mathbf{b}) \simeq f(\mathbf{y}, \hat{\mathbf{b}}_y) \exp\left\{ -\frac{1}{2}(\mathbf{b} - \hat{\mathbf{b}}_y)^{\mathsf{T}} \mathbf{H} (\mathbf{b} - \hat{\mathbf{b}}_y) \right\}$$

and so

$$f_y(\mathbf{y}) \simeq f(\mathbf{y}, \hat{\mathbf{b}}_y) \int \exp\left\{ -\frac{1}{2}(\mathbf{b} - \hat{\mathbf{b}}_y)^{\mathsf{T}} \mathbf{H} (\mathbf{b} - \hat{\mathbf{b}}_y) \right\} d\mathbf{b}.$$

From the properties of normal p.d.f.s we have

$$\int \frac{|\mathbf{H}|^{1/2}}{(2\pi)^{d/2}} e^{-\frac{1}{2}(\mathbf{b} - \hat{\mathbf{b}}_y)^{\mathsf{T}} \mathbf{H} (\mathbf{b} - \hat{\mathbf{b}}_y)} d\mathbf{b} = 1 \Rightarrow \int e^{-\frac{1}{2}(\mathbf{b} - \hat{\mathbf{b}}_y)^{\mathsf{T}} \mathbf{H} (\mathbf{b} - \hat{\mathbf{b}}_y)} d\mathbf{b} = \frac{(2\pi)^{d/2}}{|\mathbf{H}|^{1/2}}$$

so

$$f_y(\mathbf{y}) \simeq f(\mathbf{y}, \hat{\mathbf{b}}_y) \frac{(2\pi)^{d/2}}{|\mathbf{H}|^{1/2}}.$$

Notice that the basic ingredients of this approximation are the Hessian of the log integrand w.r.t $\mathbf{b}$ and the $\hat{\mathbf{b}}_y$ values. Calculating the latter is an optimisation problem, which can usually be solved by a Newton type method, given that we can presumably obtain gradients as well as the Hessian. The highest cost here is the Hessian calculation, which scales with the square of the dimension, at worst, or for cheap integrands the determinant evaluation, which scales as the cube of the dimension. i.e. when it works, this approach is *much* cheaper than brute force quadrature.

### 4.4.4 Monte-Carlo integration

Consider integrating $f(\mathbf{x})$ over some region $\Omega$ of volume $V(\Omega)$. Clearly

$$I_\Omega = \int_\Omega f(\mathbf{x}) d\mathbf{x} = \mathbb{E}\{f(\mathbf{X})\} V(\Omega)$$

where $\mathbf{X} \sim$ uniform over $\Omega$. Generating $N$ uniform random vectors, $\mathbf{x}_i$, over $\Omega$, we have the obvious estimator

$$\hat{I}_\Omega = \frac{V(\Omega)}{N} \sum_{i=1}^{N} f(\mathbf{x}_i),$$

which defines basic *Monte-Carlo Integration*. ‡ Clearly $\hat{I}_\Omega$ is unbiased, and if the $\mathbf{x}_i$ are independent then

$$\mathrm{var}(\hat{I}_\Omega) = \frac{V(\Omega)^2}{N^2} N \mathrm{var}\{f(\mathbf{X})\}$$

i.e. the variance scales as $N^{-1}$, and the standard deviation scales as $N^{-1/2}$ (it's $O_p(N^{-1/2})$). This is not brilliantly fast convergence to $I_\Omega$, but compare it to quadrature. If we take the midpoint rule then the (bias) error is $O(h^2)$ where $h$ is the node spacing. In $d$ dimensions then at best we can arrange $h \propto N^{-1/d}$, implying that the bias is $O(N^{-2/d})$. i.e. once we get above $d = 4$ we would prefer the stochastic convergence rate of Monte-Carlo Integration to the deterministic convergence rate of the midpoint rule.

Of course using Simpson's rule would buy us a few more dimensions, and for smooth functions the spectacular convergence rates of Gauss quadrature may keep it as the best option up to a rather higher $d$, but the fact remains that its bias is still dependent on $d$, while the sampling error of the Monte Carlo method is dimension independent, so Monte Carlo will win in the end.

### 4.4.5 Stratified Monte-Carlo Integration

One tweak on basic Monte-Carlo is to divide (a region containing) $\Omega$ into $N$ equal volumed sub-regions, $\Omega_i$, and to estimate the integral within each $\Omega_i$ from a single $\mathbf{X}_i$ drawn from a uniform distribution over $\Omega_i$. The estimator has the same form as before, but its variance is now

$$\mathrm{var}(\tilde{I}_\Omega) = \frac{V(\Omega)^2}{N^2} \sum_{i=1}^{N} \mathrm{var}\{f(\mathbf{X}_i)\}$$

which is usually lower than previously, because the range of variation of $f$ within each $\Omega_i$ is lower than within the whole of $\Omega$.

The underlying idea of trying to obtain the nice properties of uniform random variables, but with more even spacing over the domain of integration, is pushed to the limit by *Quasi-Monte-Carlo* integration, which uses deterministic space filling sequences of numbers as the basis for integration. For fixed $N$, the bias of such schemes grows much less quickly with dimension than the quadrature rules.

### 4.4.6 Importance sampling

In statistics an obvious variant on Monte-Carlo integration occurs when the integrand factors into the product of a p.d.f. $f(\mathbf{x})$ and another term in which case

$$\int \phi(\mathbf{x}) f(\mathbf{x}) d\mathbf{x} \approx \frac{1}{n} \sum_{i=1}^{n} \phi(\mathbf{x}_i)$$

where the $\mathbf{x}_i$ are observations on r.v.s with p.d.f. $f(\mathbf{x})$. A difficulty with this approach, as with simple MC integration, is that a high proportion of the $\mathbf{x}_i$s often fall in places where $\phi(\mathbf{x}_i)$ makes no, or negligible, contribution to the integral. For example, in Bayesian modelling, it is quite possible for the $f(\mathbf{y}|\boldsymbol{\theta})$ to be have non-vanishing support over only a tiny volume of the parameter space over which $f(\boldsymbol{\theta})$ has non-vanishing support.

One approach to this problem is *importance sampling*. Rather than generate the $\mathbf{x}_i$ from $f(\mathbf{x})$ we generate from a distribution with the same support as $f$, but with a p.d.f., $g(\mathbf{x})$, which is designed to generate points concentrated in the places where the integrand is non-negligible. Then the estimator is

$$\int \phi(\mathbf{x}) f(\mathbf{x}) d\mathbf{x} = \int \phi(\mathbf{x}) \frac{f(\mathbf{x})}{g(\mathbf{x})} g(\mathbf{x}) d\mathbf{x} \approx \hat{I}_{is} = \frac{1}{n} \sum_{i=1}^{n} \phi(\mathbf{x}_i) \frac{f(\mathbf{x}_i)}{g(\mathbf{x}_i)}.$$

The expected value of the r.h.s. of the above is its l.h.s., so this estimator is unbiased.

‡Note that in practice, if $\Omega$ is a complicated shape, it may be preferable to generate uniformly over a simpler enclosing region $\tilde{\Omega}$, and just set the integrand to zero for points outside $\Omega$ itself.

What would make a good $g$? Suppose that $g \propto \phi f$. Then $g(\mathbf{x}) = \phi(\mathbf{x})f(\mathbf{x})/k$, where $k = \int \phi(\mathbf{x})f(\mathbf{x})d\mathbf{x}$. So $\hat{I}_{is} = nk/n = \int \phi(\mathbf{x})f(\mathbf{x})d\mathbf{x}$, and the estimator has zero variance. Clearly this is impractical. If we knew $k$ then we would not need to estimate, but there are useful insights here anyway. Firstly, the more constant is $\phi f/g$, the lower the variance of $\hat{I}_{is}$. Secondly, the idea that the density of evaluation points should reflect the contribution of the integrand in the local neighbourhood is quite generally applicable.

### 4.4.7 Laplace importance sampling

We are left with the question of how to choose $g$? When the integrand is some sort of probability density function, a reasonable approach is to approximate it by the normal density implied by the Laplace approximation. That is we take $g$ to be the p.d.f. of the multivariate normal $N(\hat{\mathbf{x}}, \mathbf{H}^{-1})$ where $\hat{\mathbf{x}}$ denotes the $\mathbf{x}$ value maximising $\phi f$, while $\mathbf{H}$ is the Hessian of $-\log(\phi f)$ w.r.t. $\mathbf{x}$, evaluated at $\hat{\mathbf{x}}$.

In fact, as we have seen previously, multivariate normal random vectors are generated by transformation of standard normal random vectors $\mathbf{z} \sim N(\mathbf{0}, \mathbf{I})$. i.e. $\mathbf{x} = \hat{\mathbf{x}} + \mathbf{R}^{-1}\mathbf{z}$ where $\mathbf{R}$ is a Cholesky factor of $\mathbf{H}$ so that $\mathbf{R}^{\mathsf{T}}\mathbf{R} = \mathbf{H}$. There is then some computational saving in using a change of variable argument to re-express the importance sampling estimator in terms of the density of $\mathbf{z}$, rather than $g$ itself...

$$\int \phi(\mathbf{x})f(\mathbf{x})d\mathbf{x} \approx \frac{(2\pi)^{d/2}}{n|\mathbf{R}|}\sum_{i=1}^{n} \phi(\hat{\mathbf{x}} + \mathbf{R}^{-1}\mathbf{z}_i)f(\hat{\mathbf{x}} + \mathbf{R}^{-1}\mathbf{z}_i)e^{\mathbf{z}_i^{\mathsf{T}}\mathbf{z}_i/2}$$

Some tweaks may be required.

1. It might be desirable to make $g$ a little more diffuse than the Laplace approximation suggests, to avoid excessive importance weights being assigned to $\mathbf{z}_i$ values in the tails of the target distribution.
2. If the integration is in order to obtain a likelihood, which is to be maximised, then it is usually best to condition on a set of $\mathbf{z}_i$ values, in order to avoid random variability in the approximate likelihood, which will make optimisation difficult.
3. Usually the log of the integrand is evaluated rather than the integrand itself, and it is the log of the integral that is required. To avoid numerical under/overflow problems note that

$$\log\left\{\sum_i w_i \exp(\psi_i)\right\} = \log\left[\sum_i w_i \exp\{\psi_i - \max(\psi_i)\}\right] + \max(\psi_i)$$

   the r.h.s. will not overflow or underflow when computed in finite precision arithmetic (for reasonable $w_i$).
4. As with simple Monte-Carlo integration, some variance reduction is achievable by stratification. The $\mathbf{z}_i$ can be obtained by suitable transformation of random vectors from a uniform distribution on $(0,1)^d$. Instead $(0,1)^d$ can be partitioned into equal volume parts, with one uniform vector generated in each...

Tweak 3. is an example of a general technique for avoiding numerical underflow and overflow which has many applications in statistical computing. It is therefore worth examining more carefully.

### 4.4.8 The log–sum–exp trick

In statistical computing, we rarely want to work with raw probabilities and likelihoods, but instead work with their logarithms. Log-likelihoods are much less likely to numerically underflow or overflow than raw likelihoods. For likelihood computations involving iid observations, this is often convenient, since a product of raw likelihood terms becomes a sum of log-likelihood terms. However, sometimes we have to evaluate the sum of raw probabilities or likelihoods. The obvious way to do this would be to exponentiate the logarithms, evaluate the sum, and then take the log of the result to get back to our log scale. However, the whole point to working on the log scale is to avoid numerical underflow and overflow, so we know that exponentiating raw likelihoods is a dangerous operation which is likely to fail in many problems. Can we do the log–sum–exp operation safely? Yes, we can.

Suppose that we have log (unnormalised) weights (typically probabilities or likelihoods), $l_i = \log w_i$, but we want to compute the (log of the) sum of the raw weights. We can obviously exponentiate the log weights and sum them, but this carries the risk that all of the log weights will underflow (or, possibly, overflow). So, we want to compute

$$L = \log \sum_i \exp(l_i),$$

without the risk of underflow. We can do this by first computing $m = \max_i l_i$ and then noting that

$$L = \log \sum_i \exp(l_i) = \log \sum_i \exp(m) \exp(l_i - m)$$

$$= \log \left[ \exp(m) \sum_i \exp(l_i - m) \right] = m + \log \sum_i \exp(l_i - m).$$

We are now safe, since $l_i - m$ cannot be bigger than zero, and hence no term can overflow. Also, since we know that $l_i - m = 0$ for some $i$, we know that at least one term will not underflow to zero. So we are guaranteed to get a sensible finite answer in a numerically stable way. Note that exactly the same technique works for computing a sample mean (or other weighted sum) as opposed to just a simple sum (as above).

### 4.4.9 An example

Consider a Generalised Linear Mixed Model:

$$y_i|\mathbf{b} \sim \text{Poi}(\mu_i), \quad \log(\mu_i) = \mathbf{X}_i\boldsymbol{\beta} + \mathbf{Z}_i\mathbf{b}, \quad \mathbf{b} \sim N(\mathbf{0}, \boldsymbol{\Lambda}^{-1})$$

where $\boldsymbol{\Lambda}$ is diagonal and depends on a small number of parameters. From the model specification it is straightforward to write down the joint probability function of $\mathbf{y}$ and $\mathbf{b}$, i.e.

$$\log\{f(\mathbf{y}, \mathbf{b})\} = \sum_i \{y_i(\mathbf{X}_i\boldsymbol{\beta} + \mathbf{Z}_i\mathbf{b}) - \exp(\mathbf{X}_i\boldsymbol{\beta} + \mathbf{Z}_i\mathbf{b}) - \log(y_i!)\}$$

$$- \frac{1}{2} \sum_j \lambda_j b_j^2 + \frac{1}{2} \sum_j \log(\lambda_j)$$

but likelihood based inference actually requires that we can evaluate the marginal p.f. $f_y(\mathbf{y})$ for any values of the model parameters. Now

$$f_y(\mathbf{y}) = \int f(\mathbf{y}, \mathbf{b}) d\mathbf{b}$$

and evaluation of this requires numerical integration.

To tie things down further, suppose that the fixed effects part of the model is simply $\beta_0 + \beta_1 x$ where $x$ is a continuous covariate, while the random effects part is given by two factor variables, $z_1$ and $z_2$, each with two levels. So a typical row of $\mathbf{Z}$ is a vector of 4 zeros and ones, in some order. Suppose further that the first two elements of $\mathbf{b}$ relate to $z_1$ and the remainder to $z_2$ and that $\text{diag}(\boldsymbol{\Lambda}) = [\sigma_1^{-2}, \sigma_1^{-2}, \sigma_2^{-2}, \sigma_2^{-2}]$. The following simulates 50 data from such a model.

```
set.seed(0);n <- 50
x <- runif(n)
z1 <- sample(c(0,1),n,replace=TRUE)
z2 <- sample(c(0,1),n,replace=TRUE)
b <- rnorm(4) ## simulated random effects
X <- model.matrix(~x)
Z <- cbind(z1,1-z1,z2,1-z2)
eta <- X%*%c(0,3) + Z%*%b
mu <- exp(eta)
y <- rpois(mu,mu)
```

Here is a function evaluating $\log\{f(\mathbf{y},\mathbf{b})\}$ (log is used here to avoid undesirable underflow to zero)

```
lf.yb0 <- function(y, b, theta, X, Z, lambda) {
  beta <- theta[1:ncol(X)]
  theta <- theta[-(1:ncol(X))]
  eta <- X%*%beta + Z%*%b
  mu <- exp(eta)
  lam <- lambda(theta, ncol(Z))
  sum(y*eta - mu - lfactorial(y)) - sum(lam*b^2)/2 +
      sum(log(lam))/2
}
```

`theta` contains the fixed effect parameters, i.e. $\boldsymbol{\beta}$ and the parameters of $\boldsymbol{\Lambda}$. So in this example, `theta` is of length 4, with the first 2 elements corresponding to $\boldsymbol{\beta}$ and the last two are $\sigma_1^{-1}, \sigma_2^2$. `lambda` is the name of a function for turning the parameters of $\boldsymbol{\Lambda}$ into its leading diagonal elements, for example

```
var.func <- function(theta, nb) c(theta[1], theta[1],
                                  theta[2], theta[2])
```

Actually, for integration purposes, `lf.yb0` can usefully be re-written to accept a matrix `b` where each column is a different $\mathbf{b}$ at which to evaluate the log p.d.f. Suppose `lf.yb` is such a function. Lets try evaluating $f_y(\mathbf{y})$ at the simulated data, and the `theta` vector used for simulation, using the various integration methods covered above.

### *Quadrature rules*

First try brute force application of the midpoint rule. We immediately have the slight problem of the integration domain being infinite, but since the integrand will vanish at some distance from $\mathbf{b} = \mathbf{0}$, we can probably safely restrict attention to the domain $[-5, 5]^4$.

```
  theta <- c(0,3,1,1)
  nm <- 10;m.r <- 10
  bm <- mesh((1:nm-(nm+1)/2)*m.r/nm,4,rep(m.r/nm,nm))
  lf <- lf.yb(y,t(bm$X),theta,X,Z,var.func)
  log(sum(bm$w*exp(lf-max(lf)))) + max(lf)

## [1] -112.8546
```

Of course, we don't actually know how accurate this is, but we can get some idea by doubling `nm`, in which case

```
  log(sum(bm$w*exp(lf-max(lf)))) + max(lf)

## [1] -114.9476
```

. . . clearly we are not doing all that well, even with 10000-160000 function evaluations.

On very smooth functions Gauss quadrature rules were a much better bet, so let's try a naive application of these again.

```
  nm <- 10
  gq <- gauss.quad(nm)
  w <- gq$weights*m.r/2 ## rescale to domain
  x <- gq$nodes*m.r/2
  bm <- mesh(x,4,w)
  lf <- lf.yb(y,t(bm$X),theta,X,Z,var.func)
  log(sum(bm$w*exp(lf-max(lf)))) + max(lf)
```

```
## [1] -133.2743
```

— this is a long way from the midpoint estimate: what happens if we double `nm` to 20?

```
  log(sum(bm$w*exp(lf-max(lf)))) + max(lf)

## [1] -121.6612
```

Increasing further still indicates problems. The difficulty is that the integrand is not very smooth, in the way that $e^x$ was, and it is not well approximated by a tensor product of high order polynomials. The midpoint rule actually works better here. In fact this integral *could* be performed reasonably accurately by skilful use of Gauss quadrature, but the example illustrates that naive application can be *much* worse than doing something simpler.

### *Laplace approximation*

To use the Laplace approximation we need to supplement `lf.yb` with functions `glf.yb` and `hlf.yb` which evaluate the gradient vector and Hessian of the $\log(f)$ w.r.t. $\mathbf{b}$. Given these functions we first find $\hat{\mathbf{b}}$ by Newton's method

```
b <- rep(0,4)
while(TRUE) {
  b0 <- b
  g <- glf.yb(y,b,theta,X,Z,var.func)
  H <- hlf.yb(y,b,theta,X,Z,var.func)
  b <- b - solve(H,g)
  if (sum(abs(b-b0))<1e-10*sum(abs(b))) break
}
```

after which `b` contains $\hat{\mathbf{b}}$ and `H` the negative of the required $\mathbf{H}$. It remains to evaluate the approximation

```
  lf.yb(y,b,theta,X,Z,var.func) + 2*log(2*pi) -
    0.5*determinant(H,logarithm=TRUE)$modulus
## [1] -110.0599
```

This is closer to the truth than anything else so far, but the method itself provides us with no way of knowing how close.

### *Monte Carlo*

Simple brute force Monte Carlo Integration follows

```
  N <- 100000
  bm <- matrix((runif(N*4)-.5)*m.r,4,N)
  lf <- lf.yb(y,bm,theta,X,Z,var.func)
  log(sum(exp(lf-max(lf)))) + max(lf) - log(N) + log(m.r^4)

## [1] -116.1722
```

The next replicate gave -110.5, and that degree of variability is with $10^5$ function evaluations. Stratification improves things a bit, but let's move on.

***Laplace Importance Sampling***

```
  N <- 10000
  R <- chol(-H)
  z <- matrix(rnorm(N*4),4,N)
  bm <- b + backsolve(R,z)
  lf <- lf.yb(y,bm,theta,X,Z,var.func)
  zz.2 <- colSums(z*z)/2
  lfz <- lf +zz.2
  log(sum(exp(lfz - max(lfz)))) + max(lfz) + 2 * log(2*pi) -
      sum(log(diag(R))) - log(N)
## [1] -110.0580
```

subsequent replicates gave -110.0602, -110.0592 and -110.0587. Stratification can reduce this variability still further, and recall that we know that the estimator is unbiased (although not on the log scale, of course).

***Conclusions***

The example emphasises that integration is computationally expensive, and that multi-dimensional integration is best approached by approximating the integrand with something analytically integrable or by constructing unbiased estimators of the integral. The insight that variance is likely to be minimised by concentrating the 'design points' of the estimator on where the integrand has substantial magnitude is strongly supported by this example, and leads rather naturally, beyond the scope of this course, to MCMC methods. Note that in likelihood maximisation settings you might also consider avoiding intractable integration via the EM algorithm (see e.g. Lange, 2010 or Monahan, 2001).

## 4.5 Further reading on computer integration and differentiation

- Chapter 8 of Nocedal, J. & S.J. Wright (2006) *Numerical Optimization* (2nd ed.), Springer, offers a good introduction to finite differencing and AD, and the AD examples above are adapted directly from this source.
- Gill, P.E. , W. Murray and M.H. Wright (1981) *Practical Optimization* Academic Press, Section 8.6 offers the treatment of finite differencing that you should read before using it on anything but the most benign problems. This is one of the few references that tells you what to do if your objective is not 'well scaled'.
- Griewank, A. (2000) *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* SIAM, is *the* reference for Automatic Differentiation. If you want to know exactly how AD code works then this is the place to start. It is very clearly written and should be consulted before using AD on anything seriously computer intensive.
- `http://www.autodiff.org` is a great resource for information on AD and AD tools.
- If you want to see AD in serious action, take a look at e.g. Kalnay, E (2003) *Atmospheric Modelling, Data Assimilation and Predictability*.
- Although most widely-used AD libraries are based on object–oriented programming approaches, arguably the most elegant approaches exploit functional programming languages with strong static type systems. See, for example: Elliott, C. (2018) The simple essence of automatic differentiation. Proc. ACM Program. Lang. 2, ICFP, Article 70. DOI: `https://doi.org/10.1145/3236765`

- Chapter 10 of Monahan, J.F (2001) *Numerical Methods of Statistics*, offers more information on numerical integration, including much of the detail skipped over here, although it is not always easy to follow, if you don't already have quite a good idea what is going on.
- Lange, K (2010) *Numerical Analysis for Statisticians, second edition*, Springer. Chapter 18 is good on one dimensional quadrature.
- Ripley, B.D. (1987, 2006) *Stochastic Simulation* Chapters 5 and section 7.4 are good discussions of stochastic integration and Quasi Monte Carlo integration.
- Robert, C.P. and G. Casella (1999) *Monte Carlo Statistical Methods* Chapter 3 (and beyond) goes into much more depth than these notes on stochastic integration.
- Press et al. (2007) *Numerical Recipes (3rd ed)*, Cambridge, has a nice chapter on numerical integration, but with emphasis on 1D.

# Chapter 5

# Random number generation

The stochastic integration methods of the last section took it for granted that we can produce random numbers from various distributions. Actually we can't. The best that can be done is to produce a completely deterministic sequence of numbers which appears indistinguishable from a random sequence with respect to any relevant statistical property that we choose to test. In other words we may be able to produce a deterministic sequence of numbers that can be very well modelled as being a random sequence from some distribution. Such deterministic sequences are referred to as sequences of *pseudorandom* numbers, but the *pseudo* part usually gets dropped at some point.

The fundamental problem, for our purposes, is to generate a pseudorandom sequence that can be extremely well modelled as i.i.d. $U(0,1)$. Given such a sequence, it is fairly straightforward to generate deviates from other distributions, but the i.i.d. $U(0,1)$ generation is where the problems lie. Indeed if you read around this topic then most books will pretty much agree about how to turn uniform random deviates into deviates from a huge range of other distributions, but advice on how to get the uniform deviates in the first place is much less consistent.

## 5.1 Simple generators and what can go wrong

Since the 1950s there has been much work on linear congruential generators. The intuitive motivation is something like this. Suppose I take an integer, multiply it by some enormous factor, re-write it in base - 'something huge', and then throw away everything except for the digits after the decimal point. Pretty hard to predict the result, no? So, if I repeat the operation, feeding each step's output into the input for the next step, a more or less random sequence might result. Formally the pseudorandom sequence is defined by

$$X_{i+1} = (aX_i + b) \mathrm{mod} M$$

where $b$ is 0 or 1, in practice. This is started with a *seed* $X_0$. The $X_i$ are integers ($< M$, of course), but we can define $U_i = X_i/M$. Now the intuitive hope that this recipe might lead to $U_i$ that are reasonably well modelled by i.i.d. $U(0,1)$ r.v.s is only realised for some quite special choices of $a$ and $M$, and it takes some number theory to give the generator any sort of theoretical foundation (see Ripley, 1987, Chapter 2).

An obvious property to try to achieve is *full period*. We would like the generator to visit all possible integers between $1-b$ and $M-1$ once before it starts to repeat itself (clearly the first time it revisits a value, it starts to repeat itself). We would also like successive $U_i$s to appear uncorrelated. A notorious and widely used generator called RANDU, supplied at one time with IBM machines, met these basic considerations with

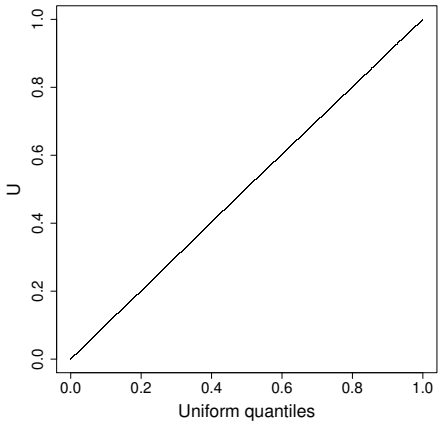
$$X_{i+1} = (65539 X_i) \mathrm{mod} 2^{31}$$

This appears to do very well in 1 dimension.

```
n <- 100000 ## code NOT for serious use
x <- rep(1,n)
a <- 65539;M <- 2^31;b <- 0 ## Randu
```

```
for (i in 2:n) x[i] <- (a*x[i-1]+b)%%M
u <- x/(M-1)
```

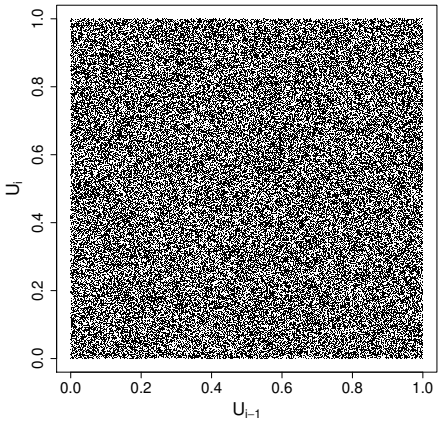
```
qqplot((1:n-.5)/n,sort(u))
```




Similarly a 2D plot of $U_i$ vs $U_{i-1}$ indicates no worries with serial correlation

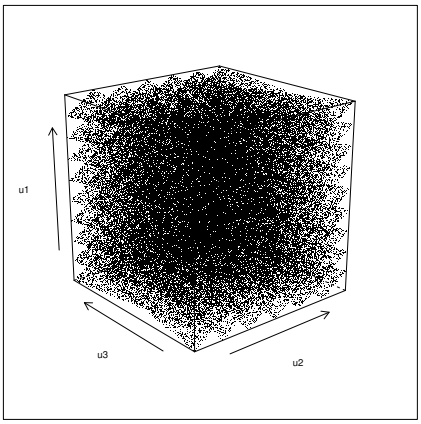
```
## Create data frame with U at 3 lags...
U <- data.frame(u1=u[1:(n-2)],u2=u[2:(n-1)],u3=u[3:n])
```

```
plot(U$u1,U$u2,pch=".")
```



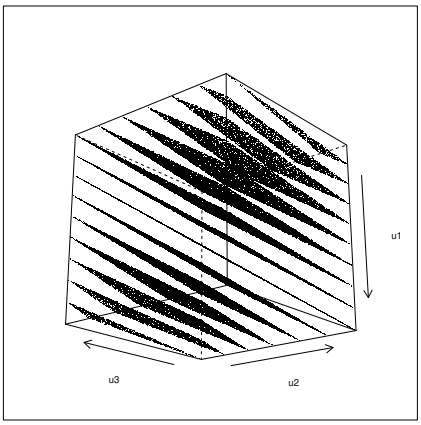

We can also check visually what the distribution of the triples $(U_i, U_{i-1}, U_{i-2})$ looks like.

```
library(lattice)
cloud(u1~u2*u3,U,pch=".",col=1,screen=list(z=40,x=-70,y=0))
```




. . . not quite so random looking. Experimenting a little with rotations gives

```
cloud(u1~u2*u3,U,pch=".",col=1,screen=list(z=40,x=70,y=0))
```



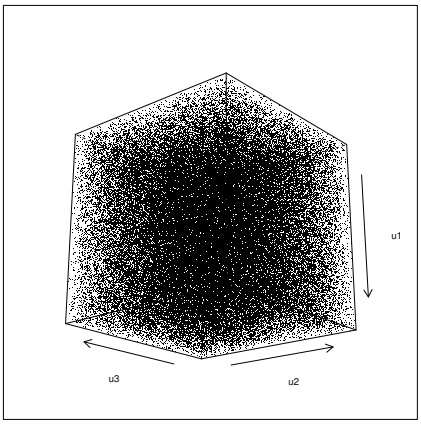

i.e. the triples lie on one of 15 planes. Actually you can show that this must happen (see Ripley, 1987, section 2.2).

Does this deficiency matter in practice? When we looked at numerical integration we saw that quadrature rules based on evaluating integrands on fixed lattices become increasingly problematic as the dimension of the integral increases. Stochastic integration overcomes the problems by avoiding the biases associated with fixed lattices. We should expect problems if we are performing stochastic integration on the basis of random numbers that are in fact sampled from some coarse lattice.

So the first lesson is to use generators that have been carefully engineered by people with a good understanding of number theory, and have then been empirically tested (Marsaglia's *Diehard* battery of tests provides one standard test set). For example, if we stick with simple congruential generators, then

$$X_i = (69069X_{i-1} + 1) \mathrm{mod} 2^{32} \tag{5.1}$$

is a much better bet. Here is its triples plot. No amount of rotating provides any evidence of structure.




Although this generator is *much* better than RANDU, it is still problematic. An obvious infelicity is the fact that a *very* small $X_i$ will always be followed by an unusually small $X_{i+1}$ (consider $X_i = 1$, for example). This is not a property that would be desirable in a time series simulation, for example. Not quite so obvious is the fact that for any congruential generator of period $M$, then $k-$tuples, $U_i, U_{i-1}, \ldots, U_{i-k+1}$ will tend to lie on a finite number of $k-1$ dimensional planes (e.g. for RANDU we saw 3-tuples lying on 2 dimensional planes.) There will be *at most* $M^{1/k}$ such planes, and as RANDU shows, there can be far fewer. The upshot of this is that if we could visualise 8 dimensions, then the 8-tuple plot for (5.1) would be just as alarming as the 3D plot was for RANDU. 8 is not an unreasonably large dimension for an integral.

Generally then, it might be nice to have generators with better behaviour than simple congruential generators, and in particular we would like generators where $k-$tuples appear uniformly distributed on $[0,1]^k$ for as high a $k$ as possible (referred to as having a high *k-distribution*).

## 5.2 Building better generators

An alternative to the congruential generators are generators which focus on generating random sequences of 0s and 1s. In some ways this seems to be the natural *fundamental* random number generation problem when using modern digital computers, and at time of writing it also seems to be the approach that yields the most satisfactory results. Such generators are often termed *shift-register* generators. The basic approach is to use bitwise binary operations to make a binary sequence 'scramble itself'. An example is Marsaglia's (2003) *Xorshift* generator as recommended in Press et al. (2007).

Let $x$ be a 64-bit variable (i.e. an array of 64 0s or 1s). The generator is initialised by setting to any value (other than 64 0s). The following steps then constitute one iteration (update of $x$)

$$\begin{aligned} x &\leftarrow x \wedge (x >> a) \\ x &\leftarrow x \wedge (x << b) \\ x &\leftarrow x \wedge (x >> c) \end{aligned}$$

each iteration generates a new random sequence of 0s and 1s. $\wedge$ denotes 'exclusive or' (XOR) and $>>$ and $<<$ are right-shift and left shift respectively, with the integers $a, b$ and $c$ giving the distance to shift. $a = 21$, $b = 35$ and $c = 4$ appear to be good constants (but see Press *et al.*, 2007, for some others).

If you are a bit rusty on these binary operators then consider an 8-bit example where `x=10011011` and `z=01011100`.

- `x<<1` is `00110110`, i.e. the bit pattern is shifted leftwards, with the leftmost bit discarded, and the rightmost set to zero.
- `x<<2` is `01101100`, i.e. the pattern is shifted 2 bits leftwards, which also entails discarding the 2 leftmost bits and zeroing the two rightmost.
- `x>>1` is `01001101`, same idea, but rightwards shift.

- `x^z` is `11000111`, i.e. a 1 where the bits in `x` and `z` disagree, and a 0 where they agree.

The Xorshift generator is very fast, has a period of $2^{64} - 1$, and passes the Diehard battery of tests (perhaps unsurprising as Marsaglia is responsible for that too). These shift register generators suffer similar granularity problems to congruential generators (there is always some $k$ for which $[0, 1]^k$ can not be very well covered by even $2^{64} - 1$ points), but tend to have all bit positions 'equally random', whereas lower order bits from congruential generator sequences often have a good deal of structure.

Now we reach a fork in the road. To achieve better performance in terms of longer period, larger k-distribution, and fewer low order correlation problems, there seem to be two main approaches, the first pragmatic, and the second more theoretical.

1. Combine the output from several 'good', well understood, simple generators using operations that maintain randomness (e.g. XOR and addition, but not multiplication). When doing this, the output from the combined generators is *never* fed back into the driving generators. Preferably combine rather different types of generator. Press et al. (2007) make a convincing case for this approach. Wichmann-Hill, available in R is an example of such a combined generator, albeit based on 3 very closely related generators.

2. Use more complicated generators — non-linear or with a higher dimensional state that just a single $X_i$ (see Gentle, 2003). For example, use a shift register type generator based on maintaining the history of the last $n$ bit-patterns, and using these in the bit scrambling operation. Matsumoto and Nishimura's (1998) *Mersenne Twister* is of this type. It achieves a period of $2^{19937} - 1$ (that's not a misprint: $2^{19937} - 1$ is a 'Mersenne prime'*), and is 623-distributed at 32 bit accuracy. i.e. its 623-tuples appear uniformly distributed (each appearing the same number of times in a full period) and spaced $2^{-32}$ apart (without the ludicrous period this would not be possible). It passes Diehard, is the default generator in R, and C source code is freely available.

## 5.3 Uniform generation conclusions

I hope that the above has convinced you that random number generation, and use of pseudo-random numbers are non-trivial topics that require some care. That said, most of the time, provided you pick a good modern generator, you will probably have no problems. As general guidelines...

1. Avoid using black-box routines supplied with low level languages such as C, you don't know what you are getting, and there is a long history of these being botched.

2. Do make sure you know what method is being used to generate any uniform random deviates that you use, and that you are satisfied that it is good enough for purpose.

3. For any random number generation task that relies on $k-$tuples having uniform distribution for high $k$ then you should be particularly careful about what generator you use. This includes any statistical task that is somehow equivalent to high dimensional integration.

4. The Mersenne Twister is probably the sensible default choice in most cases, at present. For high dimensional problems it remains a good idea to check answers with a different high-quality generator. If results differ significantly, then you will need to find out why (probably starting with the 'other' generator).

Note that I haven't discussed methods used by cryptographers. Cryptographers want to use (pseudo)random sequences of bits (0s and 1s) to scramble messages. Their main concern is that if someone were to intercept the random sequence, and guess the generator being used, they should not be able to infer the state of the generator. Achieving this goal is quite computer intensive, which is why generators used for cryptography are usually over engineered for simulation purposes.

*Numbers this large are often described as being 'astronomical', but this doesn't really do it justice: there are probably fewer than $2^{270}$ atoms in the universe.

## 5.4 Other deviates

Once you have a satisfactory stream of i.i.d. $U(0,1)$ deviates then generating deviates from other standard distributions is much more straightforward. Conceptually, the simplest approach is inversion. We know that if $X$ is from a distribution with continuous c.d.f. $F$ then $F(X) \sim U(0,1)$. Similarly, if we define the inverse of $F$ by $F^-(u) = \min(x|F(x) \geq u)$, and if $U \sim U(0,1)$, then $F^-(U)$ has a distribution with c.d.f. $F$ (this time with not even a continuity restriction on $F$ itself).

As an example here is inversion used to generate 1 million i.i.d. $N(0,1)$ deviates in R.

```
  system.time(X <- qnorm(runif(1e6)))

##    user  system elapsed
##   0.068   0.004   0.072
```

For most standard distributions (except the exponential) there are better methods than inversion, and the happy situation exists where textbooks tend to agree what these are. Ripley's (1987) Chapter 3 is a good place to start, while the lighter version is provided by Press et al. (2007) Chapter 7. R has many of these methods built in. For example an alternative generation of 1 million i.i.d. $N(0,1)$ deviates would use.

```
  system.time(Z <- rnorm(1e6))

##    user  system elapsed
##   0.069   0.001   0.070
```

Here the difference is marginal, due to the highly optimised `qnorm` function in R, but for other distributions the difference can be substantial. We have already seen how to generate multivariate normal deviates given i.i.d. $N(0,1)$ deviates, and for other (simple) multivariate distributions see chapter 4 of Ripley (1987). For more complicated distributions see the *Computationally-Intensive Statistical Methods* APTS module.

## 5.5 Further reading on pseudorandom numbers

- Gentle, J.E. (2003) *Random Number Generation and Monte Carlo Methods (2nd ed)*, Springer, Chapters 1 and 2 provides up to date summaries both of uniform generators, and of generator testing. Chapter 3 covers Quasi Monte Carlo, and the remaining chapters cover simulation from other distributions + software.
- Matsumoto, M. and Nishimura, T. (1998) Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, ACM Transactions on Modeling and Computer Simulation, 8, 3-30. This gives details and code for the Mersenne Twister.
- `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html` gives the Mersenne Twister source code.
- Marsaglia, G. (2003) Xorshift RNGs, Journal of Statistical Software, 8(14):1-6.
- DIEHARD is George Marsaglia's Diehard battery of tests for random number generators: `https://en.wikipedia.org/wiki/Diehard_tests`
- Press, W.H., S.A. Teukolsky, W.T. Vetterling and B.P Flannery (2007) *Numerical Recipes* (3rd ed.) Cambridge. Chapter 7, on random numbers is good and reasonably up to date, but in this case you do need the third edition, rather than the second (and certainly not the first).
- Ripley (1987, re-issued 2006) *Stochastic Simulation*, Wiley. Chapter's 2 and 3 are where to go to really get to grips with the issues surrounding random number generation. The illustrations alone are

an eye opener. You probably would not want to use any of the actual generators discussed and the remarks about mixed generators now look dated, but the given principles for choosing generators remain sound (albeit with some updating of what counts as simple, and the integers involved). See chapter 3 and 4 for simulating from other distributions, given uniform deviates.

- Gamerman, D. and H.F. Lopes (2006) *Markov Chain Monte Carlo: Stochastic simulation for Bayesian inference*, CRC. Chapter 1 has a nice compact introduction to simulation, given i.i.d. $U(0, 1)$ deviates.
- Devroye, L., (2003) Non-Uniform Random Variate Generation, Springer. This is a classic reference. Now out of print, it is available on-line. `http://www.nrbook.com/devroye/`

# Chapter 6

# Other topics

This module has concentrated on some topics in numerical analysis and numerical computation that are quite important for statistics. It has not covered a great many other topics in numerical analysis that are also important. Despite its name the module has made no attempt whatsoever to provide an overview of 'statistical computing' more generally, which would surely have to talk about data handling, pattern recognition, sorting, matching, searching, algorithms for graphs and trees, computational geometry, discrete optimisation, the EM algorithm, dynamic programming, linear programming, fast Fourier transforms, wavelets, support vector machines, function approximation, sparse matrices, huge data sets, graphical processing units, multi-core processing and a host of other topics.

Rather than attempt the impossible task of trying to remedy this deficiency, I want to provide a couple of examples of the ingenuity of the less numerical algorithms available, which serve, I hope, to illustrate the value of looking to see what is available when confronted with any awkward computational problem.

## 6.1 An example: sparse matrix computation

Many statistical computations involve matrices which contain very high proportions of zeroes: these are known as *sparse matrices*. Most of the computational time used by most numerical linear algebra routines is devoted to additions, subtractions and multiplications of pairs of floating point numbers: if we know, in advance, that one of the pair is a zero, we don't actually need to perform the operation, since the answer is also known in advance. In addition, there is little point in storing all the zeroes in a sparse matrix: it is often much more efficient to store only the values of the non-zero elements, together with some encoding of their location in the matrix.

Efficient storage and simple matrix manipulations are easy to implement. Here is an example, based on simulating the model matrix for a linear model with 2 factors, and a large number of observations.

```
  n <- 100000 ## number of obs
  pa <- 40;pb <- 10 # numbers of factor levels
  a <- factor(sample(1:pa,n,replace=TRUE))
  b <- factor(sample(1:pb,n,replace=TRUE))
  X <- model.matrix(~a*b) # model matrix a + b + a:b
  y <- rnorm(n) # a random response
  object.size(X) # X takes lots of memory!

## 326428560 bytes

  sum(X==0)/prod(dim(X)) # proportion of zeros

## [1] 0.9906167
```

. . . so the simulated $\mathbf{X}$ is $100,000 \times 400$ and is about 99% zeroes. It requires about 0.3Gb of computer memory, if stored as a normal 'dense' matrix. The `Matrix` library, supplied with R, supports sparse matrix

computations, so the following produces a sparse version of $\mathbf{X}$...

```
  library(Matrix)
  Xs <- Matrix(X) ## a sparse version of X
  ## much reduced memory footprint
  as.numeric(object.size(Xs))/as.numeric(object.size(X))
```

```
## [1] 0.03349214
```

Because `Xs` doesn't store the zeroes, it uses much less memory than the dense version `X`. Note that although `X` is only 1% non-zeroes, `Xs` uses 3% of the storage of `X`, because it needs to store the locations of the non-zero elements, in addition to their values. The sparse approach also saves computer time when performing basic matrix operations, by avoiding the floating point operations that are not needed because they involve a zero. For example...

```
  system.time(Xy <- t(X)%*%y)
```

```
##    user  system elapsed
##   0.312   0.090   0.330
```

```
  system.time(Xsy <- t(Xs)%*%y)
```

```
##    user  system elapsed
##   0.015   0.000   0.015
```

Again computational overheads involved in only storing the non-zeroes, and their locations, means that the cost of the sparse operation is more than 1% of the dense cost, but the saving is still substantial.

Any competent programmer could produce code for the examples given so far, but any serious application will involve *solving* linear systems, and this is *much* less straightforward. Sparse versions of some of the decompositions that we met in section 2 are needed, but naive implementations using sparse matrices flounder on the problem of *infill*. You can see the problem in the following dense calculation...

```
  system.time({
    XX <- t(X)%*%X
    R <- chol(XX)
})
```

```
##    user  system elapsed
##   1.994   0.137   0.812
```

```
  range(t(R)%*%R-XX)
```

```
## [1] -3.637979e-12  1.455192e-11
```

```
  sum(XX == 0)/prod(dim(XX))
```

```
## [1] 0.97935
```

```
  sum(R != 0) ## the upper triangle is dense (not good)
```

```
## [1] 80123
```

```
  R[1:5, 1:5] ## top left corner, for example
```

```
##             (Intercept)        a2        a3        a4        a5
## (Intercept)    316.2278  8.063808  7.877234  7.956291  7.810826
## a2               0.0000 49.849524 -1.274245 -1.287033 -1.263503
## a3               0.0000  0.000000 49.267895 -1.305385 -1.281518
## a4               0.0000  0.000000  0.000000 49.490776 -1.322352
## a5               0.0000  0.000000  0.000000  0.000000 49.030640
```

Although $\mathbf{X}^T\mathbf{X}$ is 98% zeroes, the upper triangle of its Cholesky factor is 0% zeroes, when computed in the usual way. i.e. we have lost all sparsity. Fortunately, it is possible to re-arrange the rows and columns of $\mathbf{X}^T\mathbf{X}$ (or any other matrix for which we need a sparse Cholesky factor), in a way that reduces infill in the Cholesky factor. This *pivoting* means that we get a re-ordered version of the solution of the system of interest: a minor inconvenience. Unfortunately, finding the *optimal* pivoting in terms of infill reduction is NP hard, but there are never the less practical computational pivoting strategies that work well in practice. These rely on some rather ingenious graph theory based symbolic analysis of the matrix to be factorised, which is applied before anything numerical is computed. See Davis (2006). Here's what happens when a sparse Cholesky factorisation, with infill reducing pivoting, is used on $\mathbf{X}^T\mathbf{X}$...

```
  system.time({
    XXs <- t(Xs)%*%Xs
    Rs <- chol(XXs,pivot=TRUE) ## pivot to reduce infill
  })

##    user  system elapsed
##   0.045   0.000   0.046

  range(t(Rs)%*%Rs-XXs[Rs@pivot,Rs@pivot]) ## factorization works!

## [1] -4.547474e-13  5.684342e-14

  sum(Rs!=0) ## the upper triangle is sparse (much better)

## [1] 1730

  Rs[1:5,1:5] ## top left corner, for example

## 5 x 5 sparse Matrix of class "dtCMatrix"
##             (Intercept)       a2       a3       a4       a5
## (Intercept)    15.13275  .        .        .        .
## a2              .       16.21727  .        .        .
## a3              .        .       16.09348  .        .
## a4              .        .        .       15.96872  .
## a5              .        .        .        .       16.24808
```

with sparsity maintained in the factor, sparse backward or forward substitution is also very efficient. Note that while the sparse Cholesky routines in the `Matrix` package are state-of-the-art, the QR routines are not, and carry heavy memory overheads associated with computation of $\mathbf{Q}$. You'll probably need to look for *sparse multi-frontal QR* routines if you need sparse QR.

## 6.2 Hashing: another example

Suppose that you want to store data, which is to be accessed using arbitrary *keys*. For example, you want to store data on a large number of people, using their names (as a text string) as the key for finding the record in memory. i.e., data about someone is read in from somewhere, and is stored at a location in computer memory, depending on what their name string is. Later you want to retrieve this information using their name string. How can you actually do this?

You could store some sort of directory of names, which would be searched through to find the location of the data. But such a search is always going to take some time, which will increase with the number of people's data that is stored. Can we do better? With hashing the answer is yes. The pivotal idea is to create a function which will turn an arbitrary key into an index which is either unique or shared with only a few other keys, and has a modest predefined range. This index gives a location in a (hash) table which in turn stores the location of the actual data (plus some information for resolving 'clashes'). So given a key,

we use the (hash) function to get its index, and its index to look up the data location, with no, or very little searching. The main problem is how to create a hash function.

An elegant general strategy is to use the ideas of pseudorandom number generation to turn keys into random integers. For example, if the key were some 64bit quantity, we could use it to seed the Xorshift random number generator given earlier, and run this forward for a few steps. More generally we would need an approach that successively incorporated all the bits in a key into the 'random number' generation process (see e.g. section 7.6 of Press et al., 2007, for example code). So, the hash function should return numbers that appear random, which will mean that they tend to be nicely spread out, irrespective of what the keys are like. Of course, despite this, the function always returns the same number for a given key.

It remains to turn the number returned by the hash function, $h$, into an index, $i_h$ , somewhere between 0 and the length, $n_h$, of the hash table minus 1. $i_h = h \bmod n_h$ is the usual approach. At position $i_h$ in the hash table we might store the small number of actual keys (or possibly just their $h$ values) sharing the $i_h$ value, and the actual location in memory of the corresponding data.

The really neat thing here is that the randomness of the $h$'s means that whatever the keys look like, you usually end up with a fairly uniform distribution of $i_h$'s, and only a small number of keys sharing each slot in the hash table. It is this that makes data retrieval so efficient, relative to searching a directory of keys.

If that doesn't sound like it has any application in statistics, consider the construction of triangulations in spatial statistics (a triangulation being a way of joining a set of points up with triangles, in such a way that the whole complex hull of the points is covered in triangles). Efficient construction of triangulations is tricky, and several efficient algorithms work by incrementally adding triangles. You usually know at least one neighbour of each triangle at its creation, but you actually need to know all its neighbours in the existing set. The task of finding those neighbours would be prohibitive if you had to search each existing triangle for edges shared with the new triangle. Instead one can maintain a hash table for unpaired triangle edges, where the hash key is based on the end co-ordinates of each edge. This allows any newly created edge to be paired with any existing edge in little more than the length of time it takes to create the hash key.

## 6.3 Further reading

1. For a good introduction to the sparse methods underlying Martin Maechler and Douglas Bates' R package `Matrix`, see Davis, T. A. (2006) *Direct Methods for Sparse Linear Systems*, SIAM.
2. Tim Davis's home page is worth a look for more recent sparse matrix developments `http://faculty.cse.tamu.edu/davis/research.html`.
3. If you think hashing is clever, take a look at the rest of Cormen, T., C.E. Leiserson, R.L. Rivest and C. Stein (2001) *Introduction to Algorithms* (2nd ed.) MIT.
4. Once again, Press, W.H., S.A. Teukolsky, W.T. Vetterling and B.P Flannery (2007) *Numerical Recipes* (3rd ed.) provides a readable introduction to a much wider range of topics in scientific computing than has been covered here.