Numerical Optimisation

Dom Owens

19/11/2019

How can we find the optimal value of a function from all feasible solutions? Without loss of generality, we consider minimisation for continuous problems.

The standard form of the **constrained problem** is

$$\min f(x)$$

$$s.t. g_i(x) \leq 0, i = 1, ..., m$$

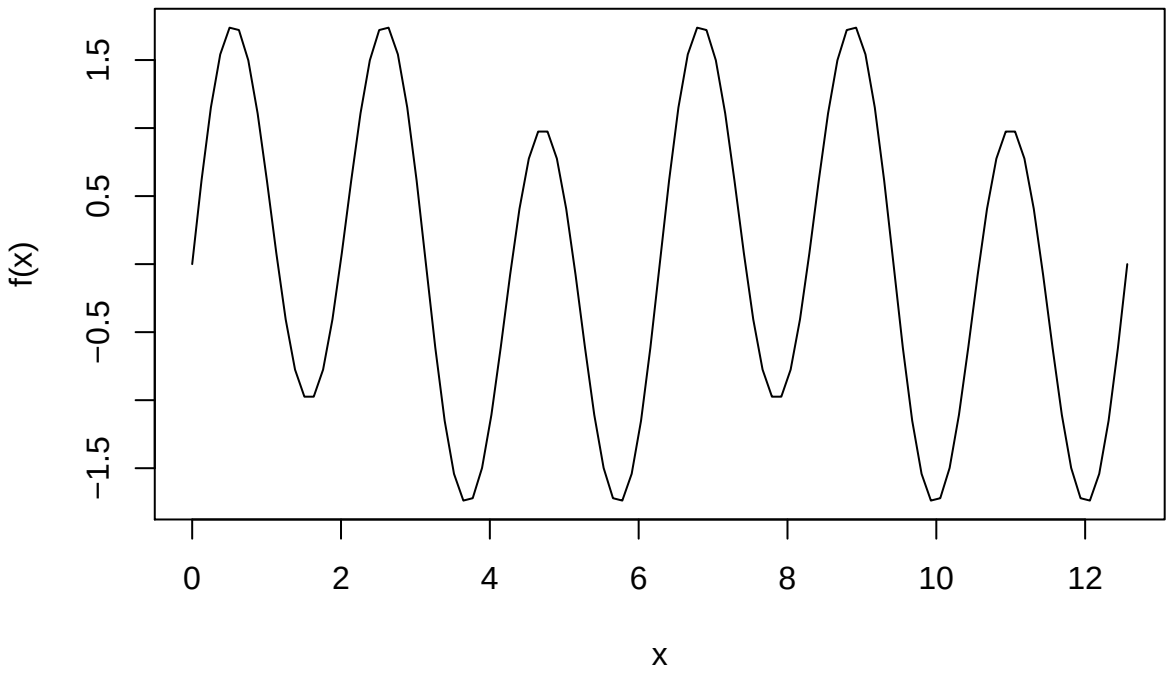
$$h_j(x) = 0, j = 1, ..., p$$

though we will focus mainly on unconstrained minimisation.

Optimisation in One Dimension

Consider this function

```
f <- function(x) cos(x)*sin(2*x)  + sin(3*x)
curve(f, from=0, to=4*pi)
```



How can we find the minimum? We could solve this minimisation with base R's `optimize`. This only works for one dimensional problems.

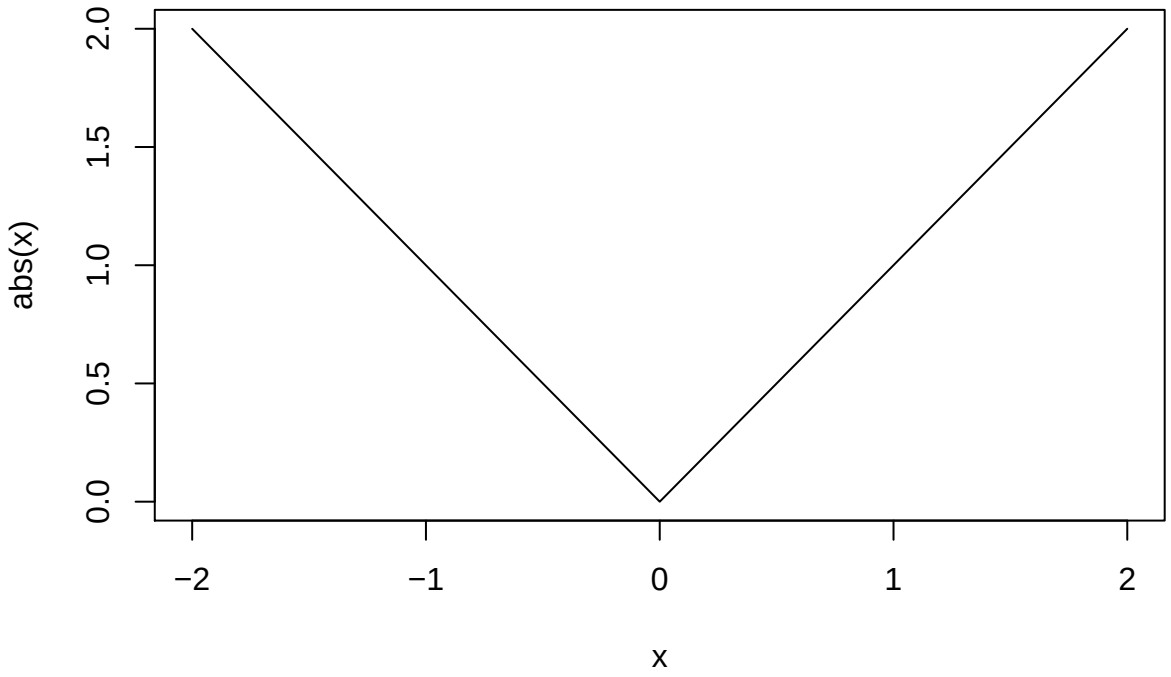
```
optimize(f, interval=c(0,4*pi))
```

```
## $minimum
## [1] 7.854002
##
```

```
## $objective
## [1] -1
```

This finds a local, not nessecarily a global, minimimum using **golden section search**. One benefit of this is that no derivative needs to be calculated, so works on continous but non-differentiable functions such as `abs`.

```
#library(soobench)
#fw <- weierstrass_function(1)
#fw <- function(x){
#  n <- 10
#  a <- 0.9
#  b <- 7
#  y <- 1:n
#  for (i in 1:n) {
#    y[i] <- a^(i-1)*cos(b^(i-1) *pi * x)
#  }
#  sum(y)
#}
#y <-
#curve(fw, -2,2)
curve(abs, from = -2, to = 2)
```




```
optimize(abs, interval=c(-2,2)) #find min
```

```
## $minimum
## [1] 5.551115e-17
##
## $objective
## [1] 5.551115e-17
```

Newton's Method

Based on the Taylor expansion, we obtain the iterative formula

$$x \leftarrow x - \frac{f'(x)}{f''(x)}$$

This allows us to find optima for f using knowledge on the gradient and

```
f_sym <- expression(cos(x)*sin(2*x)  + sin(3*x)) #define f
f1_sym <- D(f_sym, 'x') #f'
f1 <- function(x) eval(f1_sym) #f'(x)
f2_sym <- D(f1_sym, 'x') #f''
f2 <- function(x) eval(f2_sym) #f''(x)
x <- 2 #x_0
for (i in 1:6) {
  x <- x - f1(x) / f2(x); cat(x, '\n') #iterate
}
```

```
## 0.7602888
## 0.5173089
## 0.5551037
## 0.5551212
## 0.5551212
## 0.5551212
```

This converges in 4 steps to a maximum; this highlights that Newton's Method does not discriminate between minima and maxima.

Optimisation in Multiple Dimensions

How can we approach problems for $d > 1$?

We delineate the approaches to optimisation into three categories:

- **Simplex methods** using only function values
- **Gradient methods** using the first derivatives (gradient)
- **Newton-type methods** using the second derivatives (Hessian) or an approximation

These are largely implemented by `nlm` or `optim`.

Simplex Methods

Exemplified by the Nelder-Mead method of direct search. Morphs a $n + 1$ vertex in n-dimensional space based on the function values at each vertex. This is slow, converges to local minima only, and can converge to non-stationary points.

Gradient Methods

We iterate on

$$\mathbf{x} \leftarrow \mathbf{x} - t\mathbf{d}$$

where $\mathbf{d}$ is a *descent direction*, and t is a step size (chosen with e.g. line search). This is slow, so isn't used by the canonical optimisation functions, but is simple. The **Conjugate Gradient** method selects descent directions which are conjugate to the previous direction, avoiding zigzag behaviour. Tends to outperform gradient descent, but not Newton-type methods.

Newton-type Methods

We iterate on

$$\mathbf{x} \leftarrow \mathbf{x} - [H(f(\boldsymbol{x}))]^{-1} \nabla f(\mathbf{x})$$

We should consider that finding H, the Hessian matrix, in high dimensions is hard, and inversion is expensive. **Quasi-Newton** methods replace this with a reasonable approximation. **BFGS** is the most commonly used method in R, which avoids calculating any inverses but does require storing large dense matrices B_k which approximate H. The **L-BFGS** algorithm overcomes storage problems by using a low-dimensional representation of B_k.

Simulated Annealing

The three previous methods discussed were deterministic methods. **Simulated Annealing** works like the Metropolis-Hastings algorithm, jumping around the function between states according to acceptance probabilities until a minimum is reached. This finds the global minimum, but operates slowly.

Optimisation Functions in R

R has useful utilities for optimising functions. We initiate a 2D function $f(x_1,x_2) = \cos(x_1) sin(2x_2) + \sin(3x_1x_2) + (x_1-x_2)^3$ for demonstrating some methods.

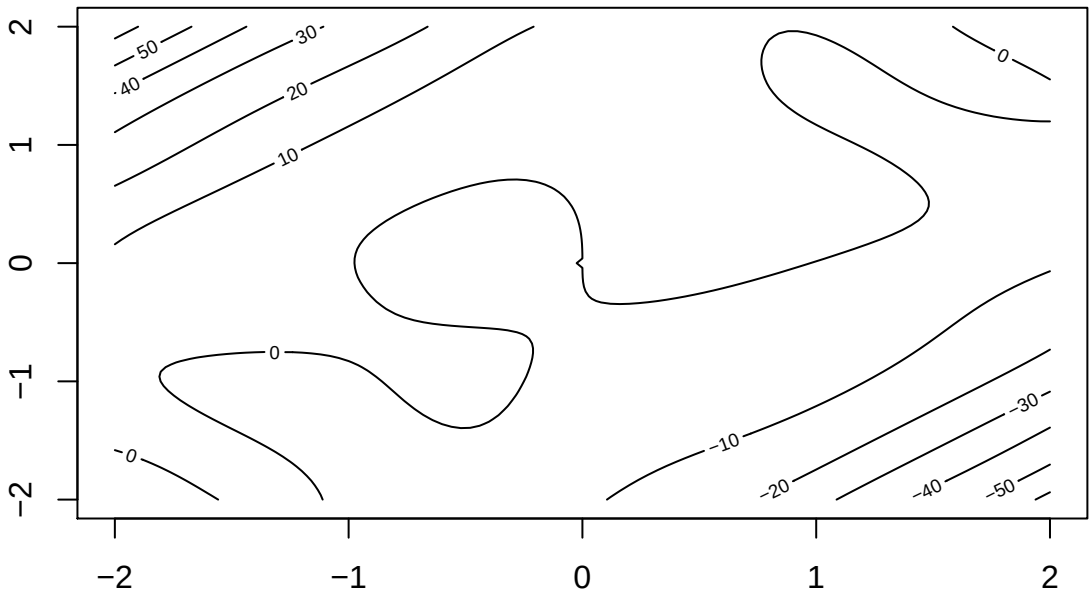
```
#install.packages("pracma")
library(pracma) # for meshgrid function
f <- function(x1, x2) cos(x1)*sin(2*x2)  + sin(3*x1*x2) + (x1-x2)^3 #define function to optimise
meshgrid(seq(-2, 2, length=5)) #output coordinate grid
```

```
## $X
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   -2   -1    0    1    2
## [2,]   -2   -1    0    1    2
## [3,]   -2   -1    0    1    2
## [4,]   -2   -1    0    1    2
## [5,]   -2   -1    0    1    2
##
## $Y
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   -2   -2   -2   -2   -2
## [2,]   -1   -1   -1   -1   -1
## [3,]    0    0    0    0    0
## [4,]    1    1    1    1    1
## [5,]    2    2    2    2    2
```

```
x <- seq(-2, 2, length=101) #store finer grid
grid_XY <- meshgrid(x)
z <- matrix(mapply(f, grid_XY$X, grid_XY$Y), nrow=101) #apply function
min(z) #print minimum
```

```
## [1] -63.14849
```

```
contour(x, x, z) #show contour plot
```



nlm

For Newton-type methods, we require the gradient and Hessian matrix.

```
f1 <- deriv(expression(cos(x1)*sin(2*x2)  + sin(3*x1*x2) + (x1-x2)^3), namevec = c('x1', 'x2'), functio
f1(0,0) #evaluate at origin
```

```
## [1] 0
## attr(,"gradient")
##      x1 x2
## [1,]  0  2
## attr(,"hessian")
## , , x1
##
##      x1 x2
## [1,]  0  3
##
## , , x2
##
##      x1 x2
## [1,]  3  0
```

We try `nlm` at two different initial conditions:

```
nlm(function(x) f1(x[1], x[2]), c(0,0)) #start nlm at (0,0)
```

```
## $minimum
## [1] -0.7844471
##
## $estimate
## [1]  0.1545715 -0.5313069
##
## $gradient
## [1] 6.494805e-13 1.813640e-06
##
## $code
```

```
## [1] 2
##
## $iterations
## [1] 15
```

```
nlm(function(x) f1(x[1], x[2]), c(10, 10),  stepmax = 1000) #start at (10,10)
```

```
## $minimum
## [1] -1.897023
##
## $estimate
## [1]  9.880883 10.120652
##
## $gradient
## [1]  0.02399007 -1.06034777
##
## $code
## [1] 4
##
## $iterations
## [1] 100
```

These find different minima.

If we don't specify the gradient or Hessian to `nlm`, numerical approximations are used. This impacts accuracy.

```
nlm(function(x) f(x[1], x[2]), c(10,10)) #start nlm at (0,0) without gradient or Hessian
```

```
## $minimum
## [1] -1.142578e+15
##
## $estimate
## [1] -52271.70  52271.38
##
## $gradient
## [1]  32787751456 -32787784247
##
## $code
## [1] 5
##
## $iterations
## [1] 11
```

optim

The canonical optimisation function in R, `optim`, uses Nelder-Mead by default.

```
optim(c(0,0), function(x) f(x[1], x[2])) #run optim using Nelder-Mead, starting at origin
```

```
## $par
## [1]  0.1545715 -0.5313072
##
## $value
## [1] -0.7844471
##
## $counts
```

```
## function gradient
##      113       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

We can alternatively use e.g. conjugate gradients, BFGS, and L-BFGS. In general, BFGS works best:

```
optim(c(0, 0), function(x) f(x[1], x[2]), method="BFGS") #run optim using BFGS algorithm, starting at o
```

```
## $par
## [1]  0.1545717 -0.5313071
##
## $value
## [1] -0.7844471
##
## $counts
## function gradient
##       24       10
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Simulated annealing will find the global minimum, but perform poorly. Indeed, we should control the maximum number of iterations.

```
optim(c(0, 0), function(x) f(x[1], x[2]), method="SANN", control=list(maxit=3000)) #run soptim with sim
```

```
## $par
## [1] -122.7966  121.5459
##
## $value
## [1] -14588047
##
## $counts
## function gradient
##     3000       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Nonlinear Least Squares

For a statistical modelling problem, we often minimise residuals (via the **residue function**)

$$r_i(\beta) = y_i - f(x_i, \beta)$$

where f is a non-linear function.

We wish to minimise the **sum of squares**

$$\min_{\beta} \sum_{1}^{m} r_i^2$$

A specific subclass of algorithms exists for this form of problem. We discuss two popular methods: **The Gauss-Newton Method** and **The Levenberg-Marquardt Algorithm**.

The Gauss-Newton Method adapts Newton's general method by approximating the Hessian of the objective function with the Jacobian matrix J_r of the residue function. This gives increments

$$\beta \leftarrow \beta - (J_r^T J_r)^{-1} J_r^T r(\beta)$$

Convergence is not guaranteed, but performance is empirically strong. When J_r is singular, this may not work.

The Levenberg-Marquardt Algorithm addresses this by damping. Using the approximation

$$2(J_r^T J_r + \lambda I)$$

with the damping parameter λ, we iterate

$$\beta \leftarrow \beta - (J_r^T J_r + \lambda I)^{-1} J_r^T r(\beta)$$

For $\lambda = 0$ this is the Gauss-Markov method; as $\lambda \to \infty$ this reaches the steepest-descent gradient method. We hence wish to reduce λ as much as possible without introducing instability.

Stochastic Gradient Descent works online, so can handle big data or time-ordered data. This iterates

$$\beta \leftarrow \beta - \gamma r_i \nabla r_i(\beta)$$

where i is deterministically or randomly selected.

We demonstrate the two methods below.

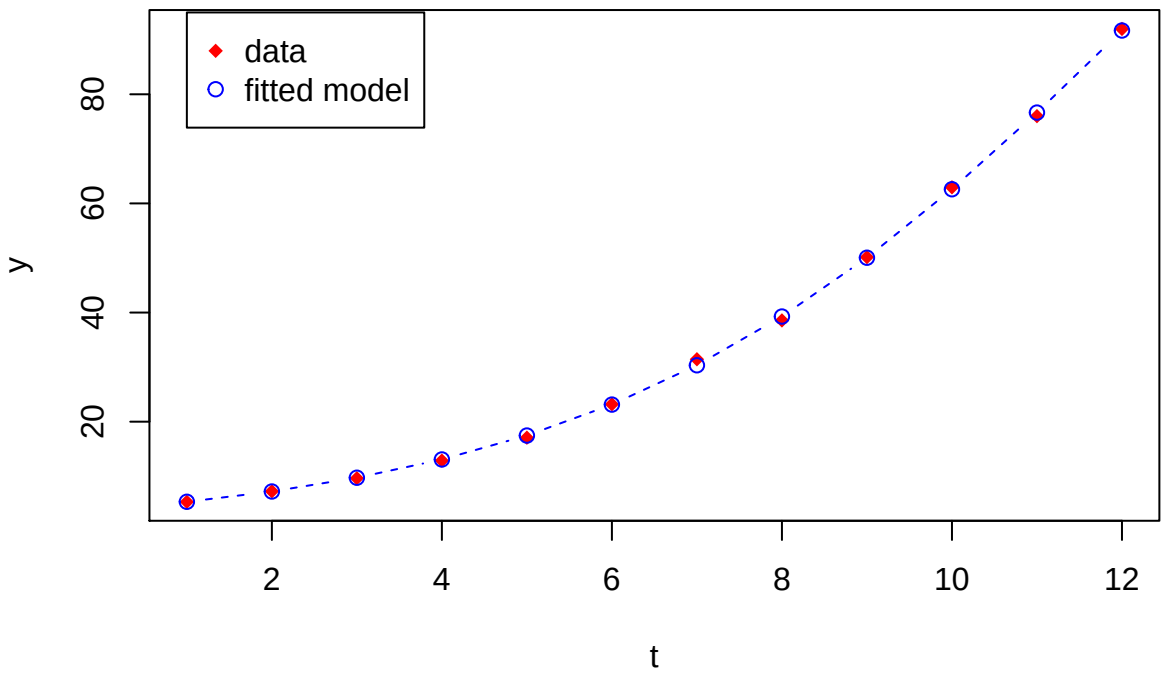
```
ydat <- c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192, 31.443, 38.558, 50.156, 62.948, 75.995, 91.972) #
tdat <- seq_along(ydat) #1:12
my_data <- data.frame(y=ydat, t=tdat) #store as frame
start1 <- c(b1=1, b2=1, b3=1) #start from (1,1,1)
my_model <- y ~ b1/(1+b2*exp(-b3*t)) #specify nonlinear model
try(nls(my_model, start=start1, data=my_data)) #incur gradient error
```

```
## Error in nls(my_model, start = start1, data = my_data) :
##   singular gradient
```

```
library(minpack.lm)
out_lm <- nlsLM(my_model, start=start1, data=my_data) #fit model using Lev.-Marq. method
out_lm
```

```
## Nonlinear regression model
##   model: y ~ b1/(1 + b2 * exp(-b3 * t))
##    data: my_data
##       b1       b2       b3
## 196.1863  49.0916   0.3136
##  residual sum-of-squares: 2.587
##
## Number of iterations to convergence: 17
## Achieved convergence tolerance: 1.49e-08
```

```r
plot(tdat, ydat, pch=18, xlab='t', ylab='y', col='red') #view performance
lines(tdat, out_lm$m$fitted(), pch=1, col='blue', type='b', lty=2)
legend(1, 95, legend=c('data', 'fitted model'), col=c('red', 'blue'), pch=c(18,1))
```



GM fails due to the gradient, but LM handles the singular gradient and works well.

We define the residue function and compare methods.

```r
f <- function(b, mydata) sum((mydata$y-b[1]/(1+b[2]*exp(-b[3]*mydata$t)))^2) #specify residue
nlm(f, mydata=my_data, p=start1) #run nlm
```

```
## $minimum
## [1] 2.587305
##
## $estimate
## [1] 196.2832646  49.0967439   0.3135034
##
## $gradient
## [1]  2.383761e-08 -6.473640e-08  3.165912e-05
##
## $code
## [1] 2
##
## $iterations
## [1] 45
```

```r
optim(par=start1, fn=f, mydata=my_data) #run optim/Nelder-Mead
```

```
## $par
##        b1        b2        b3
##  35.52886 -28.58678  18.73764
##
## $value
## [1] 9205.436
```

```
##
## $counts
## function gradient
##      156       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
optim(par=start1, fn=f, mydata=my_data, method="CG") #run optim/Conjugate Gradient
```

```
## $par
##         b1         b2         b3
## 39.7789687  5.2889870  0.4890772
##
## $value
## [1] 5336.906
##
## $counts
## function gradient
##      447      101
##
## $convergence
## [1] 1
##
## $message
## NULL
```

```
optim(par=start1, fn=f, mydata=my_data, method="BFGS") #run optim/BFGS
```

```
## $par
##          b1          b2          b3
## 196.5079544  49.1138533   0.3133611
##
## $value
## [1] 2.587543
##
## $counts
## function gradient
##      196       63
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
optim(par=start1, fn=f, mydata=my_data, method="L-BFGS-B") #run optim/ L-BFGS
```

```
## $par
##          b1          b2          b3
## 196.5325081  49.1219542   0.3133583
##
## $value
```

```
## [1] 2.587556
##
## $counts
## function gradient
##      103      103
##
## $convergence
## [1] 0
##
## $message
## [1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```