Portfolio 8

Sam Bowyer

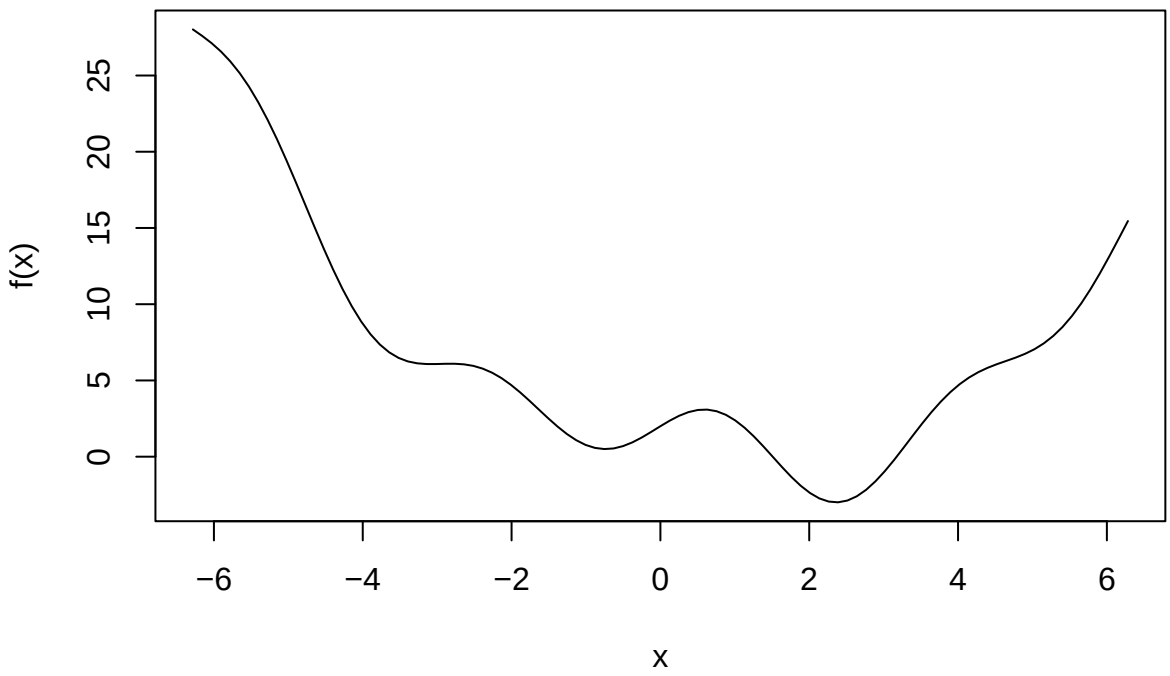
Numerical Optimization

Many tasks in data science boil down to optimising some objective function—that is, finding the best solution out of a space of possible solutions where 'best' is measured by some function. This is an enormous topic that we will be briefly covering in this portfolio, starting with one-dimensional optimization.

One-Dimensional Optimization

Say we have the function $f(x) = 2sin(2x) + 2\cos(x) + 0.5x^2 - x$.

```
f <- function(x) 2*sin(2*x) + 2*cos(x) + 0.5*x^2 - x
curve(f, from=-2*pi, 2*pi)
```



We can use the `R` function `optimize` on functions of one variable to (hopefully) find the minimum as below:

```
optimize(f, interval=c(-2*pi, 2*pi))
```

```
## $minimum
```

```
## [1] 2.361778
##
## $objective
## [1] -2.994743
```

However, note that this function doesn't always work, in particular if we instead tell it to consider the interval $[-3, 3]$ rather than $[-2\pi, 2\pi]$ we instead get a local minimum as output and not the global minimum.

```
optimize(f, interval=c(-3, 3))
```

```
## $minimum
## [1] -0.7361458
##
## $objective
## [1] 0.4989203
```

In order to improve our method of optimization within `R` we will move on to the functions `nlm` and `optim` which, unlike `optimize`, work on functions with multi-dimensional inputs.

Multi-Dimension Optimization

In this section we will minimize the 2D objective function $f(x_1, x_2) = \cos(2x_1 + x_2 - 1) + \sin(3x_1x_2) + x_1^2 + x_2^2 + 1$, however, the code presented can easily be used for higher dimensions too.

```
library(pracma) # for meshgrid function
library(plotly)
f <- function(x1, x2) cos(2*x1+x2-1) + sin(3*x1*x2) + x1^2 + x2^2 + 1
```

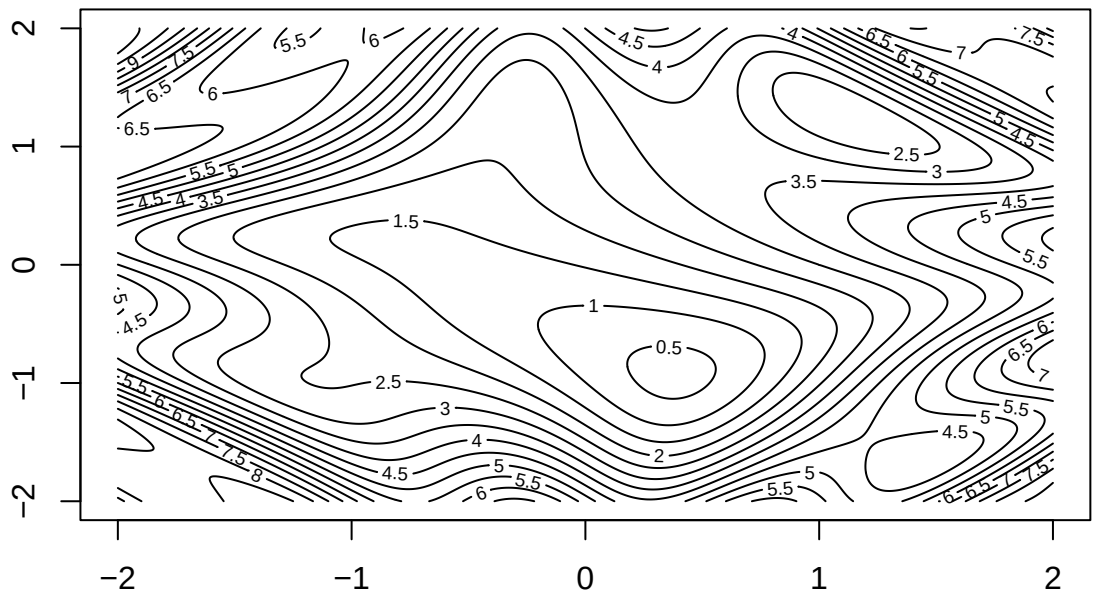
Since this is a faily simple function we can visualise this in `R` by evaluating it on a mesh of 2D points, and can then print out the minimal such evaluation, however, this minimal evaluation will not be the exact minimum of the function, so we will go on to examine much better techniques for optimization that would also work for much more complicated (e.g. much higher-dimensional) functions.

```
x <- seq(-2, 2, length=1001)
grid_XY <- meshgrid(x)

z <- matrix(mapply(f, grid_XY$X, grid_XY$Y), nrow=1001)
print(min(z))
```

```
## [1] 0.3474042
```

```
contour(x, x, z, nlevels=20)
```



Note that the function f will have minima (whether local or global) when its derivative is 0, so we can use the function `deriv` to calculate this derivative symbolically from the expression of f which will be more accurate than having the function `nlm` derive it numerically.

```
f1 <- deriv(expression(cos(2*x1+x2-1) + sin(3*x1*x2) + x1^2 + x2^2 + 1),
            namevec = c('x1', 'x2'), function.arg=T, hessian=T)
f1(0,0)
```

```
## [1] 1.540302
## attr(,"gradient")
##            x1       x2
## [1,] 1.682942 0.841471
## attr(,"hessian")
## , , x1
##
##              x1       x2
## [1,] -0.1612092 1.919395
##
## , , x2
##
##            x1       x2
## [1,] 1.919395 1.459698
```

Calling `nlm` below, we can see that a more accurate minimum has been found than through the simple mesh-grid-evaluation.

```
opt = nlm(function(x) f1(x[1], x[2]), c(0,0))
cat(opt$minimum, " @ ", opt$estimate)
```

```
## 0.3473823  @  -0.9180633 0.378447
```

The function `nlm` uses a Newton-type method which iteratively improves upon a starting solution (here we use $(0,0)$) by examining an estimate of the Hessian of the function at the current solution. Many other quai-Newton methods are possible too through the `optim` function, which by default uses the Nelder-Mead function:

```
opt = optim(c(0,0), function(x) f1(x[1], x[2]))
```

However, we can also use CG, BFGS and L-BFGS-B:

```
optim(c(0,0), function(x) f1(x[1], x[2]), method="CG")
```

```
## $par
## [1] -0.9180633  0.3784469
##
## $value
## [1] 0.3473823
##
## $counts
## function gradient
##       44       17
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
optim(c(0,0), function(x) f1(x[1], x[2]), method="BFGS")
```

```
## $par
## [1] -0.9180663  0.3784515
##
## $value
## [1] 0.3473823
##
## $counts
## function gradient
##       18       11
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
optim(c(0,0), function(x) f1(x[1], x[2]), method="L-BFGS-B")
```

```
## $par
## [1] -0.9180632  0.3784467
##
## $value
## [1] 0.3473823
##
## $counts
## function gradient
```

```
##       16       16
##
## $convergence
## [1] 0
##
## $message
## [1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

We note here that although each of these methods managed to find the same minimum, BFGS and L-BFGS-B managed to do so in significantly fewer iterations than NM and CG. It is also worth noting that the choice of start-point can have a big impact on the result achieved, for instance, if we start L-BFGS-B $(100, 100)$ instead of $(0, 0)$, we see that it ends up stuck in a less optimal local minimum.

```
optim(c(100,100), function(x) f1(x[1], x[2]), method="L-BFGS-B")
```

```
## $par
## [1] 99.21817 98.99340
##
## $value
## [1] 19644.63
##
## $counts
## function gradient
##       32       32
##
## $convergence
## [1] 0
##
## $message
## [1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

Solving a Magic-Word-Grid with Simulated Annealing

Many other methods of optimization exist and will have different advantages and disadvantages based on the problem at hand. For example, we could have also discussed gradient descent and its many variations which are extremely popular for training neural networks. However, we will finish this portfolio by implementing *simulated annealing*, which can sometimes be useful if the function to be optimized is difficult to fully write down, is not differentiable or perhaps if the search space is discrete. In particular we'll try to optimize square grids of letters so that the rows, columns and diagonals contain as many valid English words as possible (we will call these magic-word-grids)—note that for an $n \times n$ grid the maximum number of valid words we could achieve is $2n + 2$, since we have n rows, n columns and 2 diagonals (which we'll read from top-left to bottom-right and bottom-left to top-right).

We'll first check 4x4 grids and define a fitness function that counts how many words can be found in the rows and columns of the grid:

```
n = 4

# install.packages("qdapDictionaries")
library(qdapDictionaries)
isWord  <- function(x) tolower(x) %in% GradyAugmented

fitness <- function(grid){
  # Add row scores
  score = sum(isWord(apply(grid, 1, function(x) paste(unlist(x), collapse=""))))
```

```
  # Add column scores
  score = score + sum(isWord(apply(grid, 2, function(x) paste(unlist(x), collapse=""))))

  # Add diagonal scores
  score = score + isWord(paste(diag(as.matrix(grid)), collapse=""))
  score = score + isWord(paste(diag(as.matrix(rev(grid[n:1,n:1]))), collapse=""))

  return(score)
}
```

Now to generate a random grid:

```
getGrid <- function(){
  grid = data.frame(matrix(sample(LETTERS, n*n, replace=TRUE), ncol=n, nrow=n))
  colnames(grid) = 1:n
  return(grid)
}
grid = getGrid()
grid
```

```
##   1 2 3 4
## 1 D Y A M
## 2 W G C A
## 3 R O N G
## 4 M F O N
```

```
fitness(grid)
```

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

And we also need a method by which to move from one grid to a neighboring grid (in which case we'll say two grids are neighbours if they differ by at most a total of `dist` letters):

```
getNeighbour <- function(grid, dist=3){
  newGrid = grid
  for (d in 1:dist){
    pos = sample(1:n, 2, rep=TRUE)
    val = grid[pos[1], pos[2]]

    newGrid[pos[1], pos[2]] = sample(setdiff(LETTERS, val), 1)
  }
    return(newGrid)
}
neighbour = getNeighbour(grid)
neighbour
```

```
##   1 2 3 4
## 1 D Y H M
## 2 W J C A
## 3 R O N G
## 4 A F O N
```

```
fitness(neighbour)
```

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

Finally we define the simulated annealing algorithm, which is essentially random search where at each iteration we move to a neighbouring solution either if the neighbour is fitter than the previous solution or if a random

number is greater than the *temperature* variable `temp` at that particular iteration. The *cooling schedule* of this temperature is vital to the implementation and success of simulated annealing, in this case we'll use the schedule $temp_i = \frac{temp_0}{1+i/\alpha}$ with $\alpha = 700$. (Note that the cooling schedule is here simulating the decrease in particle movement as a metal cools in the metallurgic technique of *annealing*.)

```
SA <- function(s0, t0, tempUpdate, maxIter, neighbourhoodSize){
  best = s0
  bestFitness = fitness(s0)

  temp = t0

  iter = 1
  while(bestFitness < (2*n + 2) & iter <= maxIter){
    neighbour = getNeighbour(best, neighbourhoodSize)
    neighbourFitness = fitness(neighbour)

    temp = tempUpdate(t0, iter)
    rand = runif(1)

    if (neighbourFitness >= bestFitness | rand < exp((neighbourFitness - bestFitness)/temp)){
      best = neighbour
      bestFitness = neighbourFitness
    }

    if (iter %% 1000 == 0) {
      print(best)
      cat("Iteration ", iter, ": ", bestFitness, "\n")
    }

    iter = iter + 1
  }
  print(best)
  print(bestFitness)
}
set.seed(-1)
SA(grid, 1, function(t, i) t/(1+i/700), 10000, 3)
```

```
##   1 2 3 4
## 1 B S F X
## 2 Q D E P
## 3 H W T I
## 4 W W S V
## Iteration  1000 :  1
##   1 2 3 4
## 1 R M Y L
## 2 D O D O
## 3 W N C T
## 4 Z A G S
## Iteration  2000 :  5
##   1 2 3 4
## 1 B Y L A
## 2 S O U L
## 3 L U B M
## 4 O R T S
```

```
## Iteration  3000 :  5
##   1 2 3 4
## 1 A I L A
## 2 J B U G
## 3 A B R M
## 4 R J K A
## Iteration  4000 :  5
##   1 2 3 4
## 1 A I L A
## 2 L U U L
## 3 O B R M
## 4 W E K A
## Iteration  5000 :  6
##   1 2 3 4
## 1 A L P A
## 2 L U S T
## 3 O B R M
## 4 W E D A
## Iteration  6000 :  6
##   1 2 3 4
## 1 A J J A
## 2 L U S T
## 3 E S R M
## 4 E T T A
## Iteration  7000 :  7
##   1 2 3 4
## 1 A G N A
## 2 J U S T
## 3 E S R M
## 4 E T T A
## Iteration  8000 :  8
##   1 2 3 4
## 1 A G N A
## 2 J U S T
## 3 E S R M
## 4 E T T A
## Iteration  9000 :  8
##   1 2 3 4
## 1 A G N A
## 2 J U S T
## 3 E S R M
## 4 E T T A
## Iteration  10000 :  8
##   1 2 3 4
## 1 A G N A
## 2 J U S T
## 3 E S R M
## 4 E T T A
## [1] 8
```

One of the drawbacks of simulated annealing is that it involves many hyperparameters (which have only been roughly tuned here) and can take a large number of iterations to converge to a global optimum, hence why we do not actually reach an optimal 4x4 magic-word-square with fitness 10.