COSC/DATA 405/505

Modelling and Simulation

More Optimization

Integer Programming

Quadratic Programming

Nonlinear Optimization

Nelder-Mead Simplex Method

Integer programming

Decision variables are often restricted to be integers. For example, we might want to minimize the cost of shipping a product by using 1, 2 or 3 different trucks.

It is not possible to use a fractional number of trucks, so the number of trucks must be integer-valued.

Problems involving integer-valued decision variables are called integer programming problems.

Simple rounding of a non-integer solution to the nearest integer is *not* good practice; the result of such rounding can be a solution which is quite far from the optimal solution.

Integer programming

The `lp()` function has a facility to handle integer valued variables using a technique called the branch and bound algorithm.

The `int.vec` argument can be used to indicate which variables have integer values.

Integer Programming: Example

Find nonnegative x_1, x_2, x_3 and x_4 to minimize

$$C(x) = 2x_1 + 3x_2 + 4x_3 - x_4$$

subject to the constraints

$$x_1 + 2x_2 \geq 9$$

$$3x_2 + x_3 \geq 9.$$

and

$$x_2 + x_4 \leq 10.$$

Furthermore, x_2 and x_4 can only take integer values.

Integer Programming: Example

```
library(lpSolve)
integ.lp <- lp(objective.in=c(2, 3, 4, -1),
    const.mat=matrix(c(1, 0, 0, 2, 3, 1, 0,
                1, 0, 0, 0, 1), nrow=3),
    const.dir=c(">=", ">=", "<="),
     const.rhs=c(9, 9, 10),
    int.vec=c(2, 4))
 integ.lp

## Success: the objective function is 8
```

```
 integ.lp$solution

## [1] 1 4 0 6
```

Integer Programming: Example

Here is what happens when the integer variables are ignored:

```
wrong.lp <- lp(objective.in=c(2, 3, 4, -1),
    const.mat=matrix(c(1, 0, 0, 2, 3, 1, 0,
    1, 0, 0, 0, 1), nrow=3), const.dir=
    c(">=", ">=", "<="), const.rhs=c(9, 9, 10))
wrong.lp

## Success: the objective function is 8
```

```
wrong.lp$solution

## [1] 0.0 4.5 0.0 5.5
```

Alternatives to `lp()`

The `lp()` function provides an interface to code written in C.

There is another function in the linprog package called solveLP() which is written entirely in R; this latter function solves large problems much more slowly than the `lp()` function, but it provides more detailed output.

We note also the function simplex() in the boot package.

It should also be noted that, for very large problems, the simplex method might not converge quickly enough.

You might consider the interior point methods which are implemented in the *intpoint* package.

Quadratic programming

Nonlinear programming problems are a special case of optimization problems in which a possibly nonlinear function is minimized subject to constraints.

Such problems are typically more difficult to solve and are beyond the scope of this course; an exception is the case where the objective function is quadratic and the constraints are linear.

This is a problem in quadratic programming.

Quadratic programming

A quadratic programming problem with k constraints is often of the form

$$\min_{\beta} \frac{1}{2}\beta^T D\beta - d^T\beta$$

subject to constraints $A^T\beta \geq b$.

Here β is a vector of p unknowns, D is a positive definite $p \times p$ matrix, d is vector of length p, A is a $p \times k$ matrix and b is a vector of length k.

Quadratic Programming: Example

Consider the following 20 pairs of observations on the variables x and y. A scatterplot is displayed later.

```
x <- c(0.45,  0.08, -1.08,  0.92,  1.65,  0.53, 0.52, -2.15, -2.20,
       -0.32, -1.87, -0.16, -0.19, -0.98, -0.20, 0.67,  0.08,  0.38,
        0.76, -0.78)
y <- c(1.26,  0.58, -1.00,  1.07,  1.28, -0.33, 0.68, -2.22, -1.82,
       -1.17, -1.54,  0.35, -0.23, -1.53,  0.16, 0.91,  0.22,  0.44,
        0.98, -0.98)
```

Quadratic Programming: Example

Our problem is to pass a regression line through these data. We seek a line of the form

$$y = \beta_0 + \beta_1 x$$

where β_0 is the y-intercept and β_1 is the slope.

However, we have additional background information about these data that indicate that the slope β_1 of the required line is at least 1.

Quadratic Programming: Example

The line we want is the one that minimizes the sum of the squared vertical distances between the observed points and the line itself:

$$\min_{\beta_0,\beta_1} \sum_{i=1}^{20} (y_i - \beta_0 - \beta_1 x_i)^2$$

Our extra information about the slope tells us that this minimization is subject to the constraint $\beta_1 \geq 1$.

Quadratic Programming: Example

This is an example of a *restricted least-squares* problem and is equivalent to

$$\min_{\beta} \quad \beta^T X^T X \beta - 2 y^T X \beta$$

subject to

$$A\beta \geq b,$$

where $A = [0 \;\; 1]$, $\beta = [\beta_0 \;\; \beta_1]^T$.

Quadratic Programming: Example

y is a column vector consisting of the 20 y measurements, and X is a matrix consisting of two columns, where the first column contains only 1's and the second column contains the 20 x observations:

$$X = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \dots & \dots \\ 1 & x_n \end{bmatrix} = \begin{bmatrix} 1 & 0.45 \\ 1 & 0.08 \\ \dots & \dots \\ 1 & -0.78 \end{bmatrix}$$

Quadratic Programming: Example

We then have

$$X^T X = \begin{bmatrix} n & \sum_{i=1}^n x_i \\ \sum_{i=1}^n x_i & \sum_{i=1}^n x_i^2 \end{bmatrix} = \begin{bmatrix} 20 & -3.89 \\ -3.89 & 21.4 \end{bmatrix}$$

and

$$y^T X = [\sum_{i=1}^n y_i \quad \sum_{i=1}^n x_i y_i] = [-2.89 \;\; 20.7585].$$

This is a quadratic programming problem with $D = X^T X$ and $d = y^T X$.

Quadratic Programming: Example

A scatterplot of the 20 observations with a line of slope 1 and intercept 0.05 overlaid.

Quadratic Programming Functions in R

The `solve.QP()` function is in the quadprog library. Parameters:

- `Dmat` **– a matrix containing the elements of the matrix (D) of the quadratic form in the objective function.**
- `dvec` **– the coefficient vector of the decision variables in the objective function.**
- `Amat` **– the constraint coefficient matrix; each row of the matrix corresponds to a constraint.**
- `bvec` **– the constant vector on the right-hand side of the constraints.**
- `mvec` **– the number of equality constraints.**

The output from this function is a list whose first two elements are the vector that minimizes the function and the minimum value of the function.

Quadratic Programming: Example

For the restricted least squares problem considered earlier, we must first set up the matrices D and A as well as the vectors b and d. Here, $D = X^T X$ and $d = X^T y$:

```
> library(quadprog)
> X <- cbind(rep(1, 20), x)
> XX <- t(X) %*% X
> Xy <- t(X) %*% y
> A <- matrix(c(0, 1), ncol=1)
> b <- 1
```

Quadratic Programming: Example

```
> solve.QP(Dmat=XX, dvec=Xy, Amat=A, bvec=b)

$solution
[1] 0.05 1.00

$value
[1] -10.08095

$unconstrainted.solution
[1] 0.04574141 0.97810494

$iterations
[1] 2 0

$iact
[1] 1
```

Quadratic Programming: Example

From the output, we see that the required line is

$$\widehat{y} = 0.05 + x.$$

The rest of the output is indicating that the constraint is *active*.

If the unconstrained problem had yielded a slope larger than 1, the constraint would have been *inactive*, and the solution to the unconstrained problem would be the same as the solution to the constrained problem.

Quadratic Programming: Example

The decision variables in the example were restricted in sign.

Nonnegativity conditions must be explicitly set when using the `solve.QP()` function. Inequality constraints are all of the form `>=`.

If your problem contains some inequality constraints with `<=`, then the constraints should be multiplied through by -1 to convert them to the required form.

There are more efficient ways to solve restricted least squares problems in other computing environments.

The matrix D in the preceding example is a diagonal matrix, and this special structure can be used to reduce the computational burden.

The following example involves a full matrix.

This example also places a restriction on the sign of the decision variables.

Quadratic Programming: Example

Quadratic programming can be applied to the problem of finding an optimal portfolio for an investor who is choosing how much money to invest in each of a set of n stocks. A simple model for this problem boils down to maximizing

$$x^T\beta - \frac{k}{2}\beta^T D\beta$$

subject to the constraints $\sum_{i=1}^{n} \beta_i = 1$ and $\beta_i \geq 0$ for $i = 1, \ldots, n$.

The ith component of the β vector represents the fraction of the investor's fortune that should be invested in the ith stock.

Quadratic Programming: Example

Each element of this vector must be nonnegative, since the investor cannot allocate a negative fraction of her portfolio to a stock.

The vector x contains the average daily *returns* for each stock; the daily return value for a stock is the difference in closing price for the stock from one day to the next.

Therefore, $x^T \beta$ represents the average daily return for the investor.

Quadratic Programming: Example

Most investors do not want to take large risks; the second term in the objective function takes this fact into account.

The factor k quantifies the investor's tolerance for risk.

If the investor's goal is purely to maximize the average daily return without regard for the risk, then $k = 0$.

The value of k is larger for an investor who is concerned about taking risks.

Quadratic Programming: Example

The D matrix quantifies the underlying variability in the returns; it is called a *covariance* matrix.

The diagonal elements of the D matrix are the variances of the returns for each of the stocks.

An off-diagonal element (i, j) is the covariance between returns of the ith and jth stocks.

Quadratic Programming: Example

For a specific example, we consider three stocks and set $k = 2$ and

$$D = \begin{bmatrix} 0.010 & 0.002 & 0.002 \\ 0.002 & 0.010 & 0.002 \\ 0.002 & 0.002 & 0.010 \end{bmatrix}.$$

We assume the mean daily returns for the three stocks are 0.002, 0.005, and 0.01, respectively, so $x^T = [0.002\ 0.005\ 0.01]$.

Quadratic Programming: Example

The requirement that $\beta_1 + \beta_2 + \beta_3 = 1$ and the nonnegativity restrictions on the β variables can be written as

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} \begin{matrix} = \\ \geq \\ \geq \\ \geq \end{matrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Therefore, we take

$$A^T = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Quadratic Programming: Example

To set this up in R, we note first that the maximization problem is equivalent to minimizing the negative of the objective function, subject to the same constraints. This fact enables us to employ `solve.QP()`**.**

```
> A <- cbind(rep(1, 3), diag(rep(1, 3)))
> D <- matrix(c(0.01, 0.002, 0.002, 0.002, 0.01,
                0.002, 0.002, 0.002, 0.01), nrow=3)
> x <- c(0.002, 0.005, 0.01)
> b <- c(1, 0, 0, 0)
> # meq specifies the number of equality constraints;
> # these are listed before the inequality constraints
> solve.QP(2 * D, x, A, b, meq=1)
```

Quadratic Programming: Example

```
$solution
[1] 0.1041667 0.2916667 0.6041667

$value
[1] -0.002020833

$unconstrainted.solution
[1] -0.02678571  0.16071429  0.47321429

$iterations
[1] 2 0

$iact
[1] 1
```

Quadratic Programming: Example

The optimal investment strategy (for this investor) is to put 10.4% of her fortune into the first stock, 29.2% into the second stock and 60.4% into the third stock.

The optimal value of the portfolio is 0.0020 (from `$value` above).

Recall that the negative sign appears in the output, because we were minimizing the negative of the objective function.

Other Numerical Optimization Methods

- Line Search - e.g. Golden Section
- Gradient Methods - e.g. Steepest Descent
- Hessian Methods - e.g. Newton's Method
- Derivative Free Methods - e.g. Nelder-Mead

Most of these methods have been around for awhile. If you want to find out about more modern approaches, check out some of the mathematicians at UBC-O - this is one of the top optimization departments in Canada!

The Golden Section Search Method

The golden section search method is used to find the minimizer of a single-variable function which has a single minimum on a given interval $[a, b]$**.**

The function does not need to be smooth.

Consider minimizing the function

$$f(x) = x^2 + x - 2\sqrt{x}.$$

on the interval $[0, 2]$**.**

The Golden Section Search Method

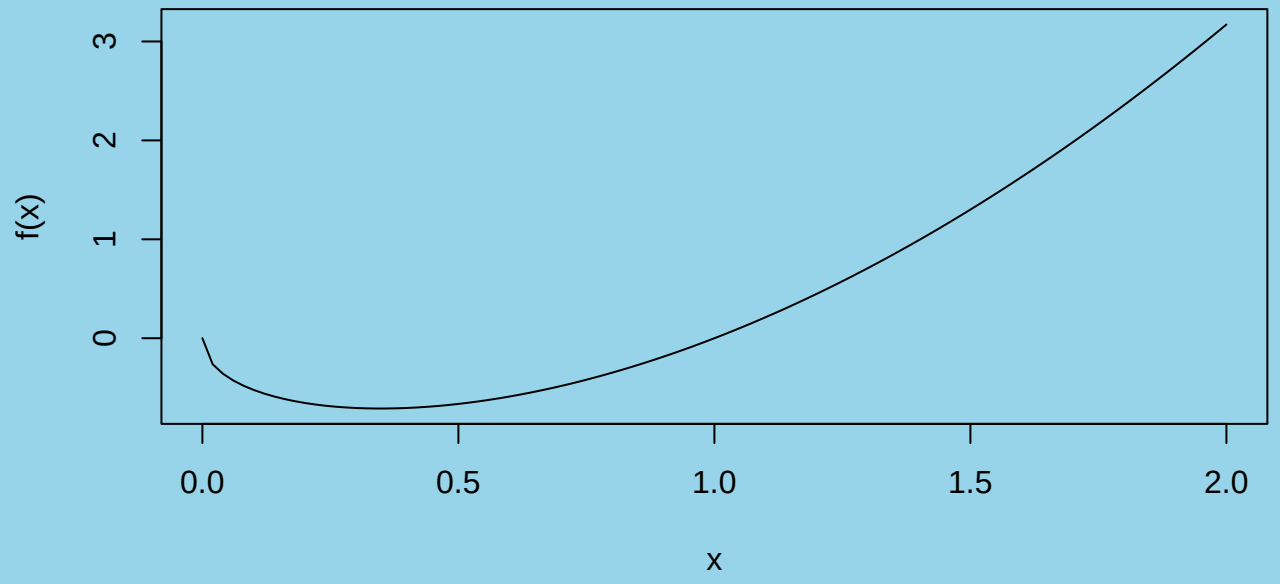
We can write an R function to evaluate $f(x)$ as follows:

```
f <- function(x) {
      x^2 + x - 2*sqrt(x)
}
```

We can use the `curve()` function to verify that there is a single minimum in $[0, 2]$:

The Golden Section Search Method: Example

```
curve(f, from = 0, to = 2)
```



the minimizer is located near $x = 0.3$.

The Golden Section Search Method

The golden section search method is an iterative method:

1. **Start with the interval $[a, b]$, known to contain the minimizer.**
2. **Repeatedly shrink the interval, finding smaller and smaller intervals $[a', b']$ which still contain the minimizer.**
3. **Stop when $b' - a'$ is small enough, i.e. when the interval length is less than a pre-set tolerance.**

When the search stops, the midpoint of the final interval will serve as a good approximation to the true minimizer, with a maximum error of $(b' - a')/2$.

The Golden Section Search Method

The shrinkage step begins by evaluating the function at two points $x_1 < x_2$ in the interior of the interval $[a, b]$.

Because we have assumed that there is a unique minimum, we know that if $f(x_1) > f(x_2)$, then the minimum must lie to the right of x_1, i.e. in the interval

$$[a', b'] = [x_1, b].$$

If $f(x_1) < f(x_2)$, the minimum must lie in

$$[a', b'] = [a, x_2].$$

The Golden Section Search Method

The choice of the points between a and b makes use of properties of the golden ratio $\phi = (\sqrt{5}+1)/2$.

We make use of the fact that $1/\phi = \phi - 1$ and $1/\phi^2 = 1 - 1/\phi$ in the following.

The Golden Section Search Method

We locate the interior points at $x_1 = b - (b - a)/\phi$ and $x_2 = a + (b - a)/\phi$.

The reason for this choice is as follows.

After one iteration of the search, it is possible that we will discard a and replace it with $a' = x_1$.

The Golden Section Search Method

Then the new value to use as x_1 will be

$$\begin{aligned} x_1' &= b - (b - a')/\phi \\ &= b - (b - x_1)/\phi \\ &= b - (b - a)/\phi^2 \\ &= a + (b - a)/\phi \\ &= x_2 \end{aligned}$$

The Golden Section Search Method

In other words, we can re-use a point we already have.

We do not need a new calculation to find it, and we don't need a new evaluation of $f(x_1')$.

We can re-use $f(x_2)$.

Similarly, if we update to $b' = x_2$, then $x_2' = x_1$, and we can re-use that point.

The Golden Section Search Method

We put this together into the following R function.

```
golden <- function (f, a, b, tol = 1e-7) {
     ratio <- 2 / (sqrt(5) + 1)
     x1 <- b - ratio * (b - a)
     x2 <- a + ratio * (b - a)
     f1 <- f(x1)
     f2 <- f(x2)
      while(abs(b - a) > tol) {
         if (f2 > f1) {
              b <- x2
              x2 <- x1
              f2 <- f1
              x1 <- b - ratio * (b - a)
              f1 <- f(x1)
         } else {
              a <- x1
              x1 <- x2
              f1 <- f2
              x2 <- a + ratio * (b - a)
              f2 <- f(x2)
         }
      }
      return((a + b) / 2)
 }
```

The Golden Section Search Method

We test and see that `golden()` works, at least on one function.

```
golden(f, 0, 2)


## [1] 0.3478104
```

Steepest Descent

The Golden Section method will work on any kind of function as long as the starting interval contains a single local minimum.

If the function is known to be smooth, other methods can be used which will converge to the solution faster.

The method of steepest descent works on functions which have a single derivative. It is used most often in problems involving more than 1 variable.

Example

A simple example of a function of 2 variables to be minimized is

$$f(x) = f(x_1, x_2) = \frac{(2 - x_1)^2}{2x_2^2} + \frac{(3 - x_1)^2}{2x_2^2} + \log(x_2)$$

Note that x_2 should be positive, so we might need to protect against negative values of x_2.

Steepest Descent Algorithm

The essential idea of steepest descent is that the function decreases most quickly in the direction of the negative gradient.

The method starts at an initial guess x.

The next guess is made by moving in the direction of the negative gradient.

The location of the minimum along this line can then be found by using a one-dimensional search algorithm such as golden section search.

Steepest Descent Algorithm

The nth update is then

$$x_n = x_{n-1} - \alpha f'(x_{n-1})^*$$

where α is chosen to minimize the one-dimensional function:

$$g(\alpha) = f\left(x_{n-1} - \alpha f'(x_{n-1})\right).$$

In order to use golden section, we need to assume that α is in an interval. We take the interval to be $[0, h]$ where h is a value that we need to choose.

*Remember: x_n and f' are vectors here. α is scalar.

Steepest Descent Code

```
steepestdescent <- function(f, fprime, start, h,
            tol=1e-7, maxiter=100) {
    x <- start
    g <- function(alpha) {
        f(x - alpha*fpx)
    }
    niter <- 0
    while(niter < maxiter & sum(abs(fprime(x))) > tol) {
        fpx <- fprime(x)
        alpha <- golden(g, 0, h)
        x <- x - alpha*fpx
        niter <- niter + 1
    }
    if (niter == maxiter) print("Warning: Maximum number
                    of iterations reached")
    c("Minimizer" = x)
}
```

Steepest Descent Example

This time, we need functions for both the function and the gradient:

```
f1 <- function(x) {
    (2-x[1])^2/(2*x[2]^2) +
        (3-x[1])^2/(2*x[2]^2) + log(x[2])
}

f1prime <- function(x) {
    c(-(2-x[1])/x[2]^2 - (3-x[1])/x[2]^2,
     -(2-x[1])^2/x[2]^3 -
         (3-x[1])^2/x[2]^3 + 1/x[2])
}
```

Steepest Descent Example

Let's try a starting value of $x = (.1, .1)$.

```
 steepestdescent(f1, f1prime, start=c(.1,.1), h=.1)


## [1] "Warning: Maximum number \n                    of iterat
## Minimizer1 Minimizer2
##  2.4992967  0.7123675
```

We haven't converged yet.

Steepest Descent Example

One possibility is to run the procedure again, using the most recent result as our starting guess:

```
steepestdescent(f1, f1prime,
    start=c(2.4992967, 0.7123675), h=.1)

## Minimizer1 Minimizer2
##  2.5000000  0.7071068
```

Success!

The Newton-Raphson Method

If the function to be minimized has two continuous derivatives and we know how to evaluate them, we can make use of this information to give a faster algorithm than the golden section search.

We want to find a minimizer x^* of the function $f(x)$ in the interval $[a, b]$. Provided the minimizer is not at a or b, x^* will satisfy $f'(x^*) = 0$.

Other solutions of $f'(x^*) = 0$ are maximizers and points of inflection. One sufficient condition to guarantee that our solution is a minimum is to check that $f''(x^*) > 0$.

The Newton-Raphson Method

If we have a guess x_0 at a minimizer, we use the fact that $f''(x)$ is the slope of $f'(x)$ and approximate $f'(x)$ using a Taylor series approximation:

$$f'(x) \approx f'(x_0) + (x - x_0) f''(x_0)$$

Finding a zero of the right hand side should give us an approximate solution to $f'(x^*) = 0$.

The Newton-Raphson Minimization Method

Start with an initial guess x_0, and compute an improved guess using the solution

$$x_1 = x_0 - \frac{f'(x_0)}{f''(x_0)}$$

Then use x_1 in place of x_0, to obtain a new update x_2.

Continue with iterations of the form

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}.$$

This iteration stops when $f'(x_n)$ is close enough to 0. Usually, we set a tolerance ε and stop when $|f'(x_n)| < \varepsilon$.

The Newton-Raphson Method

It can be shown that the Newton-Raphson method is guaranteed to converge to a local minimizer, provided the starting value x_0 is close enough to the minimizer.

As with other numerical optimization techniques, where there are multiple minimizers, Newton-Raphson won't necessarily find the best one. However, when $f''(x) > 0$ everywhere, there will be only one minimizer.

The Newton-Raphson Method

In actual practice, implementation of Newton-Raphson can be tricky. We may have $f''(x_n) = 0$, in which case the function looks locally like a straight line, with no solution to the Taylor series approximation to $f'(x^*) = 0$. In this case a simple strategy is to move a small step in the direction which decreases the function value, based only on $f'(x_n)$.

In cases where x_n is far from the true minimizer, the Taylor approximation may be inaccurate, and $f(x_{n+1})$ is larger than $f(x_n)$. When this happens, replace x_{n+1} with $(x_{n+1} + x_n)/2$ (or another value between x_n and x_{n+1}), since a smaller stepsize may produce better results.

The Newton-Raphson Method

Here is a basic implementation:

```
Newton <- function(fprime, f2prime, x, tol=1e-7, maxit=10) {
    niter <- 0
    while(niter < maxit) {
        fpx <- fprime(x)
        f2px <- f2prime(x)
        x <- x - fpx/f2px
        if (abs(fpx) < tol) break
        niter <- niter+1
    }
    if (niter == maxit) print("Warning: Maximum number
        of iterations reached in Newton iteration")
    x
}
```

Newton-Raphson: Example

Newton-Raphson can be used in place of the golden search in the steepest descent algorithm:

```
steepestdescentNewton <- function(f, fprime, f2prime,
                start, alpha0, tol=1e-7, maxiter=100) {
    x <- start
    g <- function(alpha) {
        f(x - alpha*fpx)
    }
    gprime <- function(alpha) {
        -(fpx%*%fprime(x-alpha*fpx))[1,1]
    }
    g2prime <- function(alpha) {
        (fpx%*%f2prime(x-alpha*fpx)%*%fpx)[1,1]
    }
    niter <- 0
    while(niter < maxiter & sum(abs(fprime(x))) > tol) {
        fpx <- fprime(x)
        alpha <- Newton(fprime=gprime, f2prime=g2prime, x=alpha0)
        x <- x - alpha*fpx
        niter <- niter + 1
    }
    if (niter == maxiter) print("Warning: Maximum number
                                    of iterations reached")
    c("Minimizer" = x)
}
```

Newton-Raphson: Example

We need the second derivative of f. This is a 2×2 matrix:

```
f2prime <- function(x) {
    matrix(c(2/x[2]^2, 2*(2-x[1])/x[2]^3 +2*(3-x[1])/x[2]^3,
    2*(2-x[1])/x[2]^3 +2*(3-x[1])/x[2]^3,
    3*(2-x[1])^2/x[2]^4 + 3*(3-x[1])^2/x[2]^4), nrow=2)
}
```

Newton-Raphson: Example

What happens if we start far from the solution?

```
steepestdescentNewton(f1, f1prime, f2prime, c(1.1,.5), .01)
```

If you run this code, you will receive many warnings and the solution will not be correct.

Newton-Raphson: Example

It could be worse:

```
steepestdescentNewton(f1, f1prime, f2prime, c(1.1,.5), .1)
```

This one brings you extremely far from the solution.

Newton-Raphson: Example

What happens if we try starting a bit closer to the solution?

```
steepestdescentNewton(f1, f1prime, f2prime, c(2.4,.7),.1)


## Minimizer1 Minimizer2
##   2.5000000  0.7071068
```

Success!

The Nelder-Mead Simplex Method

In the previous sections, we have talked about two different methods for optimizing a function of one variable.

However, when a function depends on multiple inputs, optimization becomes much harder.

It is hard even to visualize the function once it depends on more than two inputs.

The Nelder-Mead simplex algorithm is one method for optimization of a function of several variables.

The Nelder-Mead Simplex Method

This method is coded up as the default option in the function `optim()`.

We do not give all of the details of the algorithm here, but only a rough sketch.

In p dimensions, the algorithm starts with $p + 1$ points $x_1, \ldots, x_{p+1}$, arranged so that when considered as vertices of a p-dimensional "simplex".

For example, in two dimensions the three points would form a triangle.

The points are labelled in order from smallest to largest values of $f(x_i)$, so that $f(x_1) \leq f(x_2) \leq \cdots \leq f(x_{p+1})$.

The Nelder-Mead Simplex Method

To minimize $f(x)$, we would like to drop x_{p+1} and replace it with a point that gives a smaller value.

We do this by calculating several proposed points from the existing points.

The midpoint of $x_1, \ldots, x_p$ is calculated as $x_{mid} = (x_1 + \cdots + x_p)/p$.

There are four kinds of proposals for new points based on modifying the simplex by:

- **reflection of x_{p+1} through x_{mid} – searching in vicinity of solution**
- **reflection through x_{mid}, and expansion – expanding search region**
- **contraction towards x_{mid} – focusing search**
- **contraction of points toward x_1 – refining search near solution**

Nelder-Mead: Example

Minimize the function

$$f(x, y) = \log(1 + |x - 2y|) + e^{-(x-2)^2} + (y - 3)^2 + (x - 2)^2$$

using the Nelder-Mead algorithm.

This function is easily coded as:

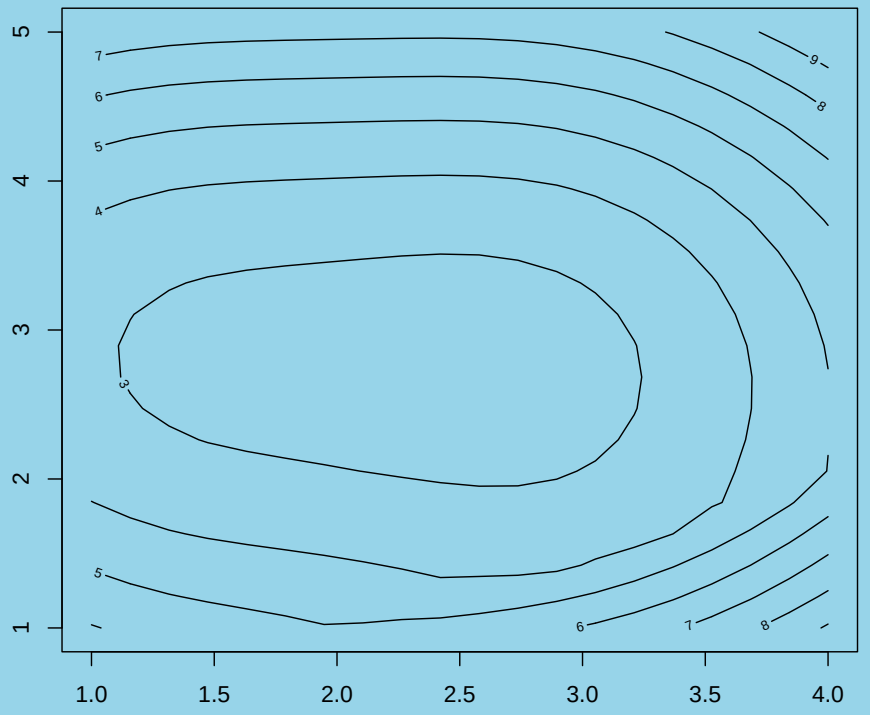
```
f <- function(x, y) {
    log(1 + abs(x - 2*y)) + exp(-(x-2)^2) +
        (y-3)^2 + (x - 2)^2
}
```

Nelder-Mead: Example

We start by drawing a contour plot of the function, in order to get approximate starting values.

After some experimentation, we obtain the plot on the right, with the following code:

```
x <- seq(1, 4,  len=20)
y <- seq(1, 5,, len=20)
z <- outer(x, y, f)
contour(x, y, z)
```



The optimum is near $(2.5, 3)$**.**

Running Nelder-Mead in R

Nelder-Mead is implemented in the `optim()` function.

Basic usage requires two arguments: `x` a vector of the unknown parameters, and `f`, a function of `x$`.
Our function `f` needs to be converted into a function we call `fNM` which takes only one argument instead of two.

```
fNM <- function(x)   f(x[1], x[2])
```

```
optim(c(2.5, 3), fNM)

## $par
## [1] 2.524009 2.748390
##
## $value
## [1] 2.477244
##
## $counts
## function gradient
##       51       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

The optimizer is $x = 2.52$ and $y = 2.75$ and the minimum value is 2.477.