```
# 2.2 Numerical Methods and Optimisation

# Numerical methods are a key part of computations for Economics among other
# quantitative fields.

# Required packages: rootSolve, numDeriv, pracma, R2Cuba, alabama
#=============================================================================>

# Systems of linear equations: matrix solve
# The equations can be formulated as Ax = B
options(digits=3)
set.seed(3)
A = matrix(runif(16), nrow = 4)
A
set.seed(3)
B = runif(4)
B
solve(A,B) #solve for the unkowns x

A%*%solve(A,B) # Should recover b
#=============================================================================>

# Roots of a polynomial

# Roots of a complex polynomial
# f(x)=1+ 2ix + (3-7i)x^2
polyroot(c(1, 2i, 3-7i)) # R recognises undefined i as complex polynomial

# Roots of a real polynomial
# f(x)=-6x - 7x^2 + x^4
polyroot(c(0, -6, -7, 0, 1))

#=============================================================================>

# Finding the zeros of a function

f = function(x,a) x^(1/3)*sin(5*x) - a*x^(1/2)
curve(f(x,a=0.5),0,5)
abline(h=0, lty=3) # this curve has several zeros

#install.packages("rootSolve") #install the package if you do not have it
require(rootSolve) #load rootSolve package.

zpts=uniroot.all(f,c(0,5),a=0.5)
zpts # all zeros of the function f over the interval 0 to 5

yz=rep(0,length(zpts))
points(zpts,yz) # Locate roots on graph of function

#=============================================================================>

# Matrix decompositions
# These have applicability in econometrics, statistics, etc.

#------------------------------------------->
# Single value decomposition
# A = UDV, D is a non-negative diagonal matrix
set.seed(13)
A = matrix(rnorm(30), nrow=6)
```

```
svd(A)

#------------------------------------------->
# Eigendecomposition - A = VDV^-1
options(digits=3)
M = matrix(c(2,-1,0,-1,2,-1,0,-1,2), nrow=3, byrow=TRUE)
eigen(M)

#------------------------------------------->
# LU decomposition
# The LU decomposition factors a square matrix into a lower triangular matrix
L
# and an upper triangular matrix U.
options(digits=3)
set.seed(1)
require(Matrix)
mm = Matrix(round(rnorm(9),2), nrow = 3)
mm

lum = lu(mm) #take the LU decomposition and save as object lum
str(lum) #view its structure
elu = expand(lum) #expand to view elements of object lum
elu

#------------------------------------------->
# Choleski decomposition
# a special case of the LU decomposition for real, symmetric, positive-
# definite square matrices.
chol(M)

#============================================================================>

# Systems of non-linear equations
# multiroot in the rootSolve package
# s^3 -3s^2 + 4r=0; r=0.96

fs = function(s) s^3 - 3*s^2 +4*rho
rho=0.96
curve(fs(x),0,3); abline(h=0)

#Thus we search for roots between 1.5 and 2.5. (See figure in plot)
options(digits=3)
multiroot(fs, c(1.5,2.5))

require(rootSolve)
model = function(x) c(F1 = 10*x[1]+3*x[2]^2-3,
                      F2 = x[1]^2 -exp(x[2]) -2)
# Note: input x is a vector

(ss = multiroot(model,c(1,1)))

#============================================================================>

# Numerical differentiation

# Numerical differentiation is important where taking analytical ones is
# either infeasible or difficult

#------------------------------------------->
# Example:
```

```
f = function(x) x^3 * sin(x/3) * log(sqrt(x))

# Using the fundamental definition

df = function(f,x0,h) (f(x0+h)-f(x0))/h
# f is the function, x0 is the point, h is the deviation
# Note: in this case, df() is a function which takes another function f()

# set values
x0 = 1; h = 1e-5

df(f,x0,h) #take derivative
# with positive h, this is a forward derivative

#-------------------------------------------->
# backward derivative
dfb = function(f,x0,h) (f(x0)-f(x0-h))/h
dfb(f,x0,h)
# with positive negative, this is a backward derivative
#-------------------------------------------->
# combining forward and backward delivers the central difference formula

dfc<- function(f,x0,h) (f(x0+h)-f(x0-h))/(2*h)
dfc(f,x0,h)

#-------------------------------------------->
# Numerical differentiation using the numDeriv package
require(numDeriv)
options(digits=16)

# trying different methods
grad(f, 1, method = "simple")

grad(f, 1, method = "Richardson")

grad(f, 1, method = "complex")

#-------------------------------------------->
# grad() for multivariate functions

# Let us compute the gradient of the likelihood function at c(rep(1,4))
# If not in memory, refer to 1.1 and run the function like()

# Gradient of likelihood function like()
lfun<- function(pars) like(y=dat$nonwife,x=xx,pars)#define as function of
pars
grad(lfun,c(rep(1,4))) # compute the gradient of the likelihood function

# Gradient of function f
f = function(u){
  x = u[1]; y = u[2]; z = u[3]
return(2*x + 3*y^2 - sin(z))
}

grad(f,c(1,1,0)) # gradient of f at c(1,1,0)

#-------------------------------------------->
# jacobian() of a system of equations
require(numDeriv)
F = function(x) c(x[1]^2 + 2*x[2]^2 - 3, cos(pi*x[1]/2) -5*x[2]^3)
```

```
jacobian(F, c(2,1))

#------------------------------------------->
# hessian()
# The hessian matrix may be thought of as the jacobian of the gradient
# of the function.
options(digits = 5) #set number of digits to display
hessian(f,c(1,1,0))
hessian(lfun,c(rep(1,4))) # compute the gradient of the likelihood function
eigen(hessian(lfun,rep(1,4)))#hessian of the likelihood function at rep(1,4)
# is negative (semi-)definite
#------------------------------------------->
# Higher order derivatives

require(pracma)
f = function(x) x^3 * sin(x/3) * log(sqrt(x))
x = 1:4
fderiv(f,x) # 1st derivative at 4 points

fderiv(f,x,n=2,h=1e-5) # 2nd derivative at 4 points

#============================================================================>

# Numerical integration

#------------------------------------------->
# integrate: Basic integration in R
f = function(x) exp(-x) * cos(x)
( q = integrate(f, 0, pi) )

# The integrand function needs to be vectorized, otherwise one will get
# an error message, e.g., with the following nonnegative function:

f1 = function(x) max(0, x)
integrate(f1, -1, 1)

# now vectorize
f1=Vectorize(f1)
integrate(f1, -1, 1)

#------------------------------------------->
# Integrating discretized functions
# Discretised functions occur when the function is not explicitly known, but
# is represented by a number of discrete points,

require(pracma)
f = function(x) exp(-x) * cos(x)
xs = seq(0, pi, length.out = 101) # what does the function seq() do?
ys = f(xs)
trapz(xs, ys)

#use the integrate() with the explicit form of the function
integrate(f,0,pi)

#------------------------------------------->
# Integration over infinite integration domains
fgauss = function(t) exp(-t^2/2) # specify a function
( q = integrate(fgauss, -Inf, Inf) )

q$value / sqrt(2*pi) # is our approximation accurate?
```

```
#------------------------------------------->
# Integrals in higher dimensions
# For multidimensional integration two packages on CRAN, cubature and
# R2Cuba, provide this functionality on hyperrectangles using adaptive
# procedures internally.

require(R2Cuba)

# computing the volume of a sphere
f = function(u, w) { x = u[1]; y = u[2]; z = u[3]
 if (x^2 + y^2 + z^2 <= 1) 1 else 0
 }
ndim = 3; ncomp = 1 # the number of dimensions of the integral ndim
q = vegas(ndim, ncomp, f, lower = c(0,0,0), upper = c(1,1,1))
( V = 8 * q$value )

# computing the volume of a sphere
f = function(x) prod(1/sqrt(2*pi)*exp(-x^2))
require(R2Cuba)
ndim = 10; ncomp = 1 # the number of components of the integrand
cuhre(ndim, ncomp, f, lower=rep(0, 10), upper=rep(1, 10))

#------------------------------------------->
# Monte Carlo and sparse grid integration
# As a naive example, we try to compute the volume of the unit sphere in R3.
# A set of N uniformly distributed points in [0;1]3 is generated and the
# number of points is counted that lie in the volume of the sphere. Because
# the unit cube has volume one, the fraction of points falling into the
sphere
# is also the volume of the sphere in [0;1]3 or one eighth of the total
volume.
# set.seed(4321)

N = 10^6
x = runif(N); y = runif(N); z = runif(N)
V = 8 * sum(x^2 + y^2 + z^2 <= 1) / N
V

#=============================================================================>

# Optimisation

#------------------------------------------->
# One-dimensional optimization
# The base R function for finding minima (the default) or maxima of functions
of
# a single variable is optimize()

# Example with an anonymous function:
optimize(function(x) x*(20-2*x)*(16-2*x), c(0,8), maximum=T)
# can you plot the curve?

#Consider next the use of optimize with the function
f = function(x) x*sin(4*x)
curve(f,0,3)

# Applying optimize() in the simplest way yields
optimize(f,c(0,3))
```

```
# Because we have a plot of the function, we can see that we must exclude the
# local minimum from the lower and upper endpoints of the search interval.
(op<-optimize(f,c(1.5,3)))
abline(v=op$minimum) # a vertical line through the minimum point

# To find the global maximum we enter
optimize(f,c(1,3),maximum=TRUE)

# Exercises:
# Suppose a utility function in wealth: u(w) = -w^2 + 1000w + 10000
# Draw a curve of the utility function
# What is the level of wealth that maximises utility?
# What is the maximum level of utility?


#------------------------------------------->
# Multi-dimensional optimization with optim()

# Optimization in more than one dimension is harder to visualize and to
compute.
# The surface defined by the function may be visualized by the persp function

x1 = x2 = seq(.1,.9,.02)
z = outer(x1,x2,FUN=function(x1,x2) 1/x1 + 1/x2 +
            (1-x2)/(x2*(1-x1)) + 1/((1-x1)*(1-x2)))
persp(x1,x2,z,theta=45,phi=0)

# Can you explain the role of the above functions?

# Write out the function for minimisation,
f = function(x) {
   x1 = x[1]
   x2 = x[2]
   return(1/x1 + 1/x2 + (1-x2)/(x2*(1-x1)) +
             1/((1-x1)*(1-x2)))
  }
#To minimize f with respect to x1 and x2, we write
optim(c(.5,.5),f)

# Let us maximise our likelihood function from 1.1
optp<-optim(par=c(rep(0,3),2),fn=like,y=dat$nonwife,x=xx,
          control=list(fnscale=-1),hessian = T)
optp
# Because the problem is a maximisation problem, we use
# control=list(fnscale=-1), otherwise, we return a negative function value
# and use minimisation.


#=========================================================================>

# Constrained Optimisation
# R has two external packages, alabama and Rsolnp, that implement the
augmented
# Lagrange multiplier method for general nonlinear optimization.

# An example with the ALABAMA package:
f = function(x) sin(x[1]*x[2]+x[3]) # objective function

heq = function(x) -x[1]*x[2]^3 + x[1]^2*x[3]^2 -5 # equality constraint
```

```
hin = function(x) { # inequality constraint
   h = rep(NA,2)
   h[1] = x[1]-x[2] # set as non-negativity constraints
   h[2] = x[2] -x[3]
   h
   }

p0 = c(3,2,1) # starting values for constrained optimisation
require(alabama) # Also loads numDeriv package
(ans = constrOptim.nl(par=p0, fn = f, heq=heq, hin = hin))

# A more robust option from the ALABAMA package is the auglag() function.
# It allows starting values that violate the inequality constraints.
# Example:
p0 = c(1,2,3) # starting values for constrained optimisation
(ans = constrOptim.nl(par=p0, fn = f, heq=heq, hin = hin))
# now try:
(ans = auglag(par=p0, fn = f, heq=heq, hin = hin))



# Exercises:
# Constrained utility maximisation problem
# u = (c^r + (1.2m)^r)^(1/r) subject to c>=20, m>=20, c + m <=100 where
# r = -4

uF<- function(a) - (a[1]^(-4) + (1.2*a[2])^(-4))^(-0.25) # return fn value
for minimisation
p0=c(20,20) # starting values
uF(p0) # function value at starting values
hin<- function(a){
  h = rep(NA,3)
  h[1]=a[1]-20
  h[2]=a[2]-20
  h[3]=100-a[1] - a[2]
  h
} # non-linear constraints
(opF<- auglag(par = p0,fn=uF,hin = hin))
```