Chapter 8. Optimization

8.1. Introduction

The contents of the first seven chapters prepare the scientist and engineer to accomplish an extraordinarily broad set of tasks—solving systems of algebraic equations, either linear or nonlinear, solving systems of ordinary differential equations, numerical differentiation, integration and linear regression. For many undergraduate introductions to numerical methods, this would be sufficient. However there is one task that undergraduates frequently face that we have not yet covered and that task is optimization. Optimization means finding a maximum or minimum. In mathematical terms, optimization means finding where the derivative is zero.

$$\frac{df(x)}{dx} = 0 \tag{8.1}$$

Typically, the function, $f(x)$, is called the objective function, since the objective is to optimize it.

It is true that linear regression is one kind of optimization. We observed in the derivation of the regression techniques in Chapter 2 that we analytically differentiated the sum of the squares of the error (SSE) with respect to the regression parameters and then set the expressions for those partial derivatives to zero, resulting in a system of linear algebraic equations. If we can linearize the model (that is, massage the equation so that the unknown parameters appear in a linear form), then this is absolutely the way to proceed because it is always much easier to solve a system of linear algebraic equations than a system of nonlinear algebraic equations.

However, life is nonlinear and sometimes we are asked to optimize parameters for nonlinear models. To this end we provide a brief chapter on optimization of nonlinear systems. There is a vast and ever-expanding literature on optimization techniques. This quaint chapter is intended to provide a couple rudimentary techniques and introduce the student to the subject. One virtually universally acknowledged source on optimization is “Numerical Recipes” by Press et al. Students who find this chapter does not sate their curiosity are encouraged to seek out Chapter 10 of “Numerical Recipes”.

8.2. Optimization vs Root-finding in One-Dimension

One can compare the goal of optimization in equation (8.1) with the goal of root-finding in equation (4.1)

$$f(x) = 0 \tag{4.1}$$

The two equations are essentially the same, both setting a function equal to zero. This motivates the idea that we can use our existing single-equation root-finding tools to optimize nonlinear equations. If the function is simple, we can perform the differentiation by hand and obtain the functional form of the derivative. At that point we can apply any single-equation root-finding technique, such as the bisection method or the Newton-Raphson method, without modification, using as an input $f'(x)$ rather than $f(x)$.

If we either cannot or will not differentiate the function analytically, we can still use the framework of the bisection method or the Newton-Raphson method where we use the finite difference formula to provide the first derivative of the function. If we are using a technique like the bisection method, that is all we require as input. If we are using a technique like the Newton-Raphson method, which requires derivatives of the function, then we shall require the second derivative as well. Fortunately, in Chapter 3, we provided finite difference formulae for the second derivative as well.

Later in this chapter, we provide subroutines for using the bisection and Newton-Raphson method for one-dimensional optimization. The only change in the bisection code necessary to convert it from a root-finding routine to an optimization routine is that, where we previously evaluated the function at the brackets, we now evaluate the first derivative of the function at the brackets using a finite difference formula. The only changes in the Newton-Raphson with Numerical derivatives method to convert it from a root-finding routine to an optimization routine are that, (i) where we previously evaluated the function, we now evaluate the first derivative of the function using a finite difference formula and (ii) where we previously evaluated the first derivative of the function, we now evaluate the second derivative of the function using a finite difference formula.

Example 8.1. Bisection Method

Consider the single nonlinear algebraic equation as our objective function,

$$f(x) = \exp(x) - sqrt(x) \tag{8.2}$$

Although the derivative of this function can easily be evaluated, we will not do so for the sake of this application. We will take as our brackets,

$$x_- = 0.1 \text{ and } x_+ = 0.55.$$

How did we find these brackets? It was either by trial and error or we plotted *f*(*x*) vs *x* to obtain some idea where the minimum was. For optimization, we need the **slope** to be negative at x_- and the slope to be positive at x_+. We will use a relative error on *x* as the criterion for convergence and we will set our tolerance at 10^{-6}.

	x_-	x_+	$f'(x_-)$	$f'(x_+)$	error
1	0.100000	0.550000	-4.76E-01	1.06E+00	8.18E-01
2	0.100000	0.325000	-4.76E-01	5.07E-01	6.92E-01
3	0.100000	0.212500	-4.76E-01	1.52E-01	5.29E-01
4	0.156250	0.212500	-9.58E-02	1.52E-01	2.65E-01
5	0.156250	0.184375	-9.58E-02	3.80E-02	1.53E-01
6	0.170313	0.184375	-2.59E-02	3.80E-02	7.63E-02
7	0.170313	0.177344	-2.59E-02	6.72E-03	3.96E-02
8	0.173828	0.177344	-9.41E-03	6.72E-03	1.98E-02
9	0.175586	0.177344	-1.30E-03	6.72E-03	9.91E-03
10	0.175586	0.176465	-1.30E-03	2.72E-03	4.98E-03
11	0.175586	0.176025	-1.30E-03	7.12E-04	2.50E-03
12	0.175806	0.176025	-2.95E-04	7.12E-04	1.25E-03
13	0.175806	0.175916	-2.95E-04	2.09E-04	6.25E-04
14	0.175861	0.175916	-4.30E-05	2.09E-04	3.12E-04
15	0.175861	0.175888	-4.30E-05	8.29E-05	1.56E-04
16	0.175861	0.175874	-4.30E-05	1.99E-05	7.81E-05
17	0.175868	0.175874	-1.15E-05	1.99E-05	3.90E-05
18	0.175868	0.175871	-1.15E-05	4.22E-06	1.95E-05
19	0.175869	0.175871	-3.65E-06	4.22E-06	9.76E-06
20	0.175869	0.175870	-3.65E-06	2.88E-07	4.88E-06
21	0.175870	0.175870	-1.68E-06	2.88E-07	2.44E-06

So in 21 iterations, we see the optimum value lies at $x = 0.175870$. The bisection method guarantees a root. However, since we are invoking a finite difference formula to estimate the derivative of the function, we are subject to the limitations in the accuracy of the numerical differentiation. As for virtually any numerical method, it is conceivable that ill-posed problems exist for this procedure will not converge.

Example 8.2. Newton-Raphson with Numerical Derivatives Method

Consider the same objective function as used in the previous example. We will provide an initial guess for the Newton-Raphson method of $x = 1$. We will use a relative error on x as the criterion for convergence and we will set our tolerance at 10^{-6}.

	x_{old}	$f(x_{old})$	$f'(x_{old})$	x_{new}	error
1	2.000000	7.035625	7.48E+00	1.06E+00	1.00E+02
2	1.059095	2.397952	3.11E+00	2.89E-01	2.67E+00
3	0.288832	0.404506	2.95E+00	1.52E-01	9.06E-01
4	0.151500	-0.121024	5.40E+00	1.74E-01	1.29E-01
5	0.173898	-0.009088	4.64E+00	1.76E-01	1.11E-02
6	0.175858	-0.000054	4.58E+00	1.76E-01	6.75E-05
7	0.175870	0.000000	4.58E+00	1.76E-01	3.25E-09

So we converged to 0.175870 in only seven iterations.

8.4. Other One Dimensional Optimization Techniques

Nonlinear optimization can be a tricky business. There is always the possibility of falling into local minimum rather than the desired global minimum. Therefore, an exhaustive optimization search involves numerous initial guesses. If the various initial guesses lead to the different minima, then the global minimum is the one with the lowest value of the objective function.

Sometimes it is also worth trying different optimization techniques in order to avoid pitfalls of one method. For this reason, two additional optimization techniques are described below. The practical procedure in optimization is thus to try a given method. If it fails to converge, then we move on to another method. We naturally try the fastest methods (with quadratic convergence) first. If that fails, we then try the slower methods. Once we have a minimum, it is often useful to confirm the minimum by starting the optimization again (with the same method and with different methods) at a point very near the minimum to confirm the existence of the minimum (as opposed to simply having run out of iterations before convergence). Thus we might try to optimize using the Newton-Raphson method first. If it fails, we might then move on to Brent's method, described below.

A seminal resource in Numerical Methods is the book "Numerical Recipes" by Press, Teukolsky, Vetterling & Flannery. They provide various excellent routines for one-dimensional optimization. The codes are provided in either Fortran or C, depending on the version of the text book. Because these tools are so refined, we have translated some of the routines into Matlab and made the translated codes available on the course website. None of the codes translated from "Numerical Recipes" are reproduced in this book.

Relevant to the topic of one-dimensional optimization, the Brent's method of one-dimensional optimization (brent.m) and Brent's method of one-dimensional optimization with numerical derivatives (dbrent.m) are provided in Matlab. These methods incorporate not simply theory but a variety of rules of thumb and tricks of the trade for optimization, gained over many years of experience by the authors of "Numerical Recipes". The two codes, brent.m and dbrent.m require three brackets. Thus the translation of the bracket generating routine for one-dimensional optimization, mnbrak1.m is also included on the course website. These routines require that the objective function be entered in a file funkeval.m.

Example 8.3. Brent Method

Consider the same objective function in equation (8.2). We first create the objective function, funkeval.m,

```
function f = funkeval(x);
f = exp(x) -  sqrt(x);
```

We next generate brackets,

```
» [ax, bx, cx] = mnbrak1(0.1,0.2,'min')

ax =    0.1000
bx =    0.2000
```

```
cx =    0.3618
```

Note that the first two input arguments in mnbrak1 are two values of x, hopefully close to the root. The third argument, 'min', specifies minimization. The outputs are three brackets required by both brent.m and dbrent.m.

We next run the optimization code. For example, if we use brent.m, with a relative tolerance on x (fourth argument) of 10^{-6} and the print option (fifth argument) turned on (1 = on, 0 = off), we have

```
» [f,xmin] = brent(ax,bx,cx,1.0e-6,1,'min');
boundary 1:  a = 1.76e-001 f(a) =  7.73e-001
boundary 2:  b = 1.76e-001 f(b) =  7.73e-001
best guess:  x = 1.76e-001 f(x) =  7.73e-001
2nd best guess:  w = 1.76e-001 f(w) =  7.73e-001
previous w:  v = 1.76e-001 f(v) =  7.73e-001
most recent point:  u = 1.76e-001 f(u) =  7.73e-001
error = 4.35e-008 iter =    28

ANSWER = 1.758669e-001
```

So in 28 iterations, Brent's method found the minimum at 0.17587 with an error of 4.35×10^{-8}.

Alternatively, we could use the dbrent.m code.

```
» [f,xmin] = dbrent(ax,bx,cx,1.0e-6,1,'min');
boundary 1:  a = 1.76e-001 f(a) =  7.73e-001
boundary 2:  b = 1.76e-001 f(b) =  7.73e-001
best guess:  x = 1.76e-001 f(x) =  7.73e-001
2nd best guess:  w = 1.76e-001 f(w) =  7.73e-001
previous w:  v = 1.76e-001 f(v) =  7.73e-001
most recent point:  u = 1.76e-001 f(u) =  7.73e-001
error = 4.05e-008 iter =    20

ANSWER = 1.758669e-001
```

So in 20 iterations, Brent's method with derivatives found the same minimum at 0.17587 with an error of 4.05×10^{-8}.

8.5. Multivariate Nonlinear Optimization

Real systems and materials are not only nonlinear but also are dependent on multiple parameters. Therefore multivariate nonlinear optimization is a task that is practically necessary. The objective in multivariate nonlinear optimization is to minimize one objective function with respect to many variables,

$$\left(\frac{\partial f(x_1, x_2 \ldots x_n)}{\partial x_i} \right)_{x_{j \neq i}} = 0 \qquad \text{for } i = 1 \text{ to } n \tag{8.3}$$

It is often said that multivariate nonlinear optimization is an art not a science. The challenge of finding any minimum in multi-dimensional space can be difficult. The challenge of routinely finding a global minimum in multi-dimensional space is exceedingly difficult. This task is still an active area of research.

In this chapter we will provide discussion of three useful techniques for multivariate nonlinear optimization. The first is the simple conversion of a root-finding technique. The latter two are optimization techniques with codes translated from "Numerical Recipes".

8.6. Optimization vs Root-finding in Multiple Dimensions

Just as we converted the single-variable version of the Newton-Raphson with numerical derivatives method from a root-finding technique to an optimization technique, so too can we convert the multivariate Newton-Raphson method with numerical derivatives method from a root-finding technique to an optimization technique.

In this case, we have one objective function, $f_{obj}(\underline{x})$, which a function of n variables. At the k^{th} iteration, the i^{th} element of the residual in the Newton-Raphson method is first partial derivative of the objective function with respect to variable x_i, evaluated at the current values of $\underline{x}^{(k)}$,

$$R_i^{(k)} = \left(\frac{\partial f_{obj}}{\partial x_i} \right)_{x_{m \neq i}} \Bigg|_{\underline{x}^{(k)}} \tag{8.4}$$

The i,j element of the Jacobian matrix at the k^{th} iteration is the matrix of second partial derivatives, which in mathematical is called the Hessian matrix,

$$J_{i,j}^{(k)} = \left(\frac{\partial}{\partial x_j} \left(\frac{\partial f_{obj}}{\partial x_i} \right)_{x_{m \neq i}} \right)_{x_{m \neq j}} \Bigg|_{\underline{x}^{(k)}} \tag{8.5}$$

By inspection, the Hessian matrix is symmetric.

It is likely that the derivatives required in the residual vector and Jacobian matrix will be evaluated numerically. To do so, we employ the centered-finite difference formulae for first and second partial derivatives provided in Section 3.4.

Example 8.4. Multivariate Newton-Raphson with Numerical Derivatives Method

Consider the objective function,

$$f_{obj}(x_1, x_2) = (x_1 - 3)^2 + (x_2 - 7)^2 + x_1 x_2^2 - x_1^2 x_2 + 4 \tag{8.6}$$

If we provide an initial guess of $(x_1, x_2) = (4,8)$, we find that this method produces the following results

iteration	x_1	x_2	error on residual	error on x
0	4	8	-	-
1	2.1373	4.4902	35.4	2.81
2	1.3797	3.5274	6.81	0.866
3	1.2308	3.4759	0.73	0.111
4	1.2302	3.4781	1.02e-02	1.53e-03
5	1.2302	3.4781	5.56e-6	8.22e-07

So in 5 iterations, the Newton-Raphson method with numerical derivatives found the same minimum at $x_1 = 1.23$ and $x_2 = 3.47$ with a relative error on x of 8.22×10^{-7} and an RMS error on the residual of 5.56×10^{-6}. The value of the objective function at the minimum is 29.154, determined by substitution of the converged solution into the objective function.

8.7. Other Multivariate Optimization Techniques

As was the case for the single-variable nonlinear optimization, a seminal resource in multivariate nonlinear optimization is the book "Numerical Recipes" by Press, Teukolsky, Vetterling & Flannery. They provide various excellent routines for multi-dimensional optimization. The codes are provided in either Fortran or C, depending on the version of the text book. Because these tools are so refined, we have translated some of the routines into Matlab and made the translated codes available on the course website. None of the codes translated from "Numerical Recipes" are reproduced in this book.

We discuss two completely different approaches to multivariate nonlinear optimization presented in "Numerical Recipes". The first is the called either the "Nelder and Mead's Downhill Simplex Method" or the "amoeba method". In the application of the amoeba method to an n-dimensional optimization problem, one creates an n-dimensional volume using n+1 points. For example a two-dimensional volume (typically called an area) can be created from 3 points. The resulting two-dimensional shape is called a triangle. Similarly, a three-dimensional volume can be created from 4 points. The resulting three-dimensional shape is called a tetrahedron. The same concept applies to n-dimensional space. One creates this n-dimensional object and then allows it to explore n-dimensional space through a series of reflection, contraction and extrapolation operations. The process can be liked to an amoeba who moves away from its least favorable of the n+1 points by extending a pseudopod in the opposite direction (reflection). If it likes what it finds (a more hospitable environment, in this case indicated by a lower value of the objective function), it moves even further (extrapolation). If it doesn't like what it finds, it shrinks away (contraction).

Eventually all n+1 points are within a given tolerance of the minimum and the process is converged. The advantage of this method is that it is simple and hard to crash. The disadvantage is that it is slow and can require thousands or hundreds of thousands of iterations to converge. The code amoeba.m that appears on the course website is a translation of the Fortran version of the code that appears in "Numerical Recipes". Because the amoeba method is slow, it should be used only as a last resort, when other quicker methods have failed to find the optimum. This routine require that the objective function be entered in a file funkeval.m.

Example 8.5. Amoeba Method

Consider the objective function,

$$f_{obj}(x_1, x_2) = (x_1 - 3)^2 + (x_2 - 7)^2 + x_1^2 x_2^2 + 4 \tag{8.7}$$

For this two-dimensional problem, the amoeba method requires three initial guesses. We only want to provide one initial guess. Therefore, the code in general generates an additional n initial guess to the one guess provided by increase each variable in term by some fraction. For example, if we provide an initial guess of $(x_1, x_2) = (1,3)$ and our factor is 1%, then our additional two initial guesses are $(x_1, x_2) = (1.01,3)$ and $(x_1, x_2) = (1,3.03)$. The initial value of the objective function for these three initial guesses are respectively, 33, 33.141 and 32.942. We set two tolerances in the amoeba code, one for the unknowns and one for the objective function, both are set to 10^{-6} in this example. These values constrain the range of x and f_{obj} values across the n+1 points.

The code is issued at the MATLAB command line prompt with the following command:

```
» [f,x] = amoeba([1; 3],1.0e-6,1.0e-6)
```

In the following table values of x and f_{obj} that correspond to the best of the n+1 points are provided by iteration. The errors on x and f_{obj} are also reported.

iteration	x_1	x_2	f_{obj}	error on x	error on f_{obj}
1	1.00000000	3.03000000	32.941800	9.95E-03	6.03E-03
2	0.98000000	3.04500000	32.627278	1.77E-02	1.14E-02
3	0.97000000	3.11250000	32.348672	2.87E-02	1.82E-02
4	0.92500000	3.17625000	31.558717	5.06E-02	3.33E-02
5	0.88250000	3.34312500	30.560857	8.38E-02	5.68E-02
…	…	…	…	…	…
62	0.06043011	6.97453100	12.819358	5.27E-06	5.58E-13
63	0.06043011	6.97453100	12.819358	2.88E-06	5.31E-13
64	0.06043011	6.97453100	12.819358	3.36E-06	1.58E-13
65	0.06043026	6.97453030	12.819358	1.33E-06	1.21E-13
66	0.06043017	6.97452980	12.819358	6.98E-07	2.06E-14

So in 66 iterations, the amoeba method found a minimum at x_1 = 0.6043 and x_2 = 6.9745 with a relative error on x of 6.98×10^{-7} and an RMS error on the objective function of 2.06×10^{-14}. The value of the objective function at the minimum is 12.819.

The second method for multivariate nonlinear optimization that has been translated from "Numerical Recipes" is the conjugate-gradient method. The procedure of a conjugate gradient method is to perform a series of one-dimensional line minimizations. We have already been introduced to good tools for one-dimensional line minimizations earlier in the chapter. However, the directions (or gradients) of these lines do not correspond to the variable axes. Instead these gradients are selected to be as independent of (or orthogonal to) each other as possible. In the best case, the conjugate gradient method turns an *n*-dimensional optimization problem into a series of *n* one-dimensional optimization problems. Because the conjugate gradient method uses information about derivatives (obtained numerically in this code as in the Newton-Raphson code), it will converge rapidly when one is near the optimum.

Example 8.6. Conjugate Gradient Method

Consider the same objective function as was used in the previous example, equation (8.7). We again provide an initial guess of $(x_1, x_2) = (1,3)$. We set a tolerance for the relative error on x to 10^{-6} in this example. The code is issued at the MATLAB command line prompt with the following command:

```
» [f,x] = conjgrad([1;3],1.0e-6,1,'min')
```

The third and fourth arguments are a printing variable (1 for printing intermediate information from each iteration) and a key set to minimization. In the following table values of x and f_{obj} are provided by iteration. The error on x is also reported.

iteration	x_1	x_2	f_{obj}	error on x
0	1.000000	2.000000	37.000000	1.00E+02
1	-0.127925	3.691887	24.950576	6.24E+00
2	-0.220721	6.266219	16.824407	4.16E-01
3	0.060911	6.967095	12.819419	3.27E+00
4	0.060641	6.974728	12.819360	3.24E-03
5	0.060430	6.974531	12.819358	2.47E-03
6	0.060430	6.974530	12.819358	4.16E-07

So in 6 iterations, the conjugate gradient method found a minimum at $x_1 = 0.6043$ and $x_2 = 6.9745$ with a relative error on x of 4.16×10^{-7} and the value of the objective function at the minimum is 12.819.

8.8. Subroutine Codes

In this section, we provide routines for implementing the various optimization methods described above that are not translated from "Numerical Recipes". Note that these codes correspond to the theory and notation exactly as laid out in this book. These codes do not contain

extensive error checking, which would complicate the coding and defeat their purpose as learning tools. That said, these codes work and can be used to solve problems.

As before, on the course website, two entirely equivalent versions of this code are provided and are titled *code.m* and *code_short.m*. The short version is presented here. The longer version, containing instructions and serving more as a learning tool, is not presented here. The numerical mechanics of the two versions of the code are identical.

Code 8.1. Bisection Method for optimization – 1 variable (bisect_opt1_short)

```
function [x0,err] = bisect_opt1(xn,xp);
maxit = 100;
tol = 1.0e-6;
err = 100.0;
icount = 0;
h = min(0.01*xn,0.01);
fn = dfunkeval(xn,h);
h = min(0.01*xp,0.01);
fp = dfunkeval(xp,h);
while (err > tol & icount <= maxit)
   icount = icount + 1;
   xmid = (xn + xp)/2;
   h = min(0.01*xp,0.01);
   fmid = dfunkeval(xmid,h);
   if (fmid > 0)
      fp = fmid;
      xp = xmid;
   else
      fn = fmid;
      xn = xmid;
   end
   err = abs((xp - xn)/xp);
   fprintf(1,'i = %i xn = %e xp = %e fn = %e fp = %e err = %e \n',icount, xn,
xp, fn, fp, err);
end
x0 = xmid;
if (icount >= maxit)
   fprintf(1,'Sorry.  You did not converge in %i iterations.\n',maxit);
   fprintf(1,'The final value of x was %e \n', x0);
end

function df = dfunkeval(x,h)
fp = funkeval(x+h);
fn = funkeval(x-h);
df = (fp - fn)/(2*h);

function f = funkeval(x)
f = exp(x)-sqrt(x);
```

An example of using bisect_opt1_short is given below.

```
» [x0,err] = bisect_opt1_short(0.1,1);

i = 1 xn = 1.000000e-001 xp = 5.500000e-001 fn = -4.759875e-001 fp =
1.059054e+000 err = 8.181818e-001
…
i = 23 xn = 1.758699e-001 xp = 1.758700e-001 fn = -2.037045e-007 fp =
2.877649e-007 err = 6.100434e-007

x0 =   0.17586992979050
err =    6.100434297642754e-007
```

So in 23 iterations, the bisection method found the same minimum at 0.17587 with an error of 6.10×10^{-7}.

Code 8.2. Newton-Raphson Method with numerical derivatives for optimization – 1 variable (nrnd_opt1_short)

```
function [x0,err] = nrnd_opt1(x0);
maxit = 100;
tol = 1.0e-6;
err = 100.0;
icount = 0;
xold =x0;
while (err > tol & icount <= maxit)
   icount = icount + 1;
   h = min(0.01*xold,0.01);
   f = dfunkeval(xold,h);
   df = d2funkeval(xold,h);
   xnew = xold - f/df;
   if (icount > 1)
      err = abs((xnew - xold)/xnew);
   end
   fprintf(1,'icount = %i xold = %e f = %e df = %e xnew = %e  err = %e
\n',icount, xold, f, df, xnew, err);
   xold = xnew;
end
x0 = xnew;
if (icount >= maxit)
   % you ran out of iterations
   fprintf(1,'Sorry.  You did not converge in %i iterations.\n',maxit);
   fprintf(1,'The final value of x was %e \n', x0);
end

function df = dfunkeval(x,h)
fp = funkeval(x+h);
fn = funkeval(x-h);
df = (fp - fn)/(2*h);

function d2f = d2funkeval(x,h)
fp = funkeval(x+h);
```

```
fo = funkeval(x);
fn = funkeval(x-h);
d2f = (fp - 2*fo + fn)/(h*h);

function f = funkeval(x)
f = exp(x)-sqrt(x);
```

An example of using nrnd_opt1_short is given below.

```
» [x0,err] = nrnd_opt1_short(2)
icount = 1 xold = 2.000000e+000 f = 7.035625e+000 df = 7.477507e+000 xnew =
1.059095e+000  err = 1.000000e+002
…
icount = 7 xold = 1.758700e-001 f = -2.621053e-009 df = 4.582021e+000 xnew =
1.758700e-001  err = 3.252574e-009

x0 =   0.17586997424458
err =    3.252573753217557e-009
```

So in 7 iterations, the Newton-Raphson method with numerical derivatives found the same minimum at 0.17587 with an error of 3.25×10^{-9}.

Code 8.3. Newton-Raphson Method with numerical derivatives for optimization – n variables (nrnd_optn_short)

```
function [x,err,f] = nrnd_optn(x0,tol,iprint)
maxit = 1000;
n = max(size(x0));
Residual = zeros(n,1);
Jacobian = zeros(n,n);
InvJ = zeros(n,n);
dx = zeros(n,1);
x = zeros(n,1);
xold = zeros(n,1);
dxcon = zeros(n,1);
dxcon(1:n) = 0.01;
x = x0;
err = 100.0;
iter = 0;
while ( err > tol )
   for j = 1:1:n
      dx(j) = min(dxcon(j)*x(j),dxcon(j));
   end
   Residual = dfunkeval(x,dx,n);
   Jacobian = d2funkeval(x,dx,n);
   xold = x;
   invJ = inv(Jacobian);
   deltax = -invJ*Residual;
   for j = 1:1:n
```

```
      x(j) = xold(j) + deltax(j);
   end
   iter = iter +1;
   err = sqrt( sum(deltax.^2) /n  ) ;
   f = sqrt(sum(Residual.*Residual)/n);
   if (iprint == 1)
      fprintf (1,'iter = %4i, err = %9.2e f = %9.2e \n ', iter, err, f);
   end
   if ( iter > maxit)
      Residual
      error ('maximum number of iterations exceeded');
   end
end

function df = dfunkeval(x,dx,n)
df = zeros(n,1);
for i = 1:1:n
   xtemp(1:n) = x(1:n);
   xtemp(i) = x(i) + dx(i);
   fp = funkeval(xtemp);
   xtemp(i) = x(i) - dx(i);
   fn = funkeval(xtemp);
   df(i) = (fp - fn)/(2*dx(i));
end

function Jacobian = d2funkeval(x,dx,n)
Jacobian = zeros(n,n);
for i = 1:1:n
   xtemp(1:n) = x(1:n);
   xtemp(i) = x(i) + dx(i);
   fp = funkeval(xtemp);
   xtemp(i) = x(i);
   fo = funkeval(xtemp);
   xtemp(i) = x(i) - dx(i);
   fn = funkeval(xtemp);
   Jacobian(i,i) = (fp - 2*fo + fn)/(dx(i)*dx(i));
end
for i = 1:1:n-1
   for j = i+1:1:n
      xtemp(1:n) = x(1:n);
      xtemp(i) = x(i) + dx(i);
      xtemp(j) = x(j) + dx(i);
      fpp = funkeval(xtemp);
      xtemp(i) = x(i) + dx(i);
      xtemp(j) = x(j) - dx(i);
      fpn = funkeval(xtemp);
      xtemp(i) = x(i) - dx(i);
      xtemp(j) = x(j) + dx(i);
      fnp = funkeval(xtemp);
      xtemp(i) = x(i) - dx(i);
      xtemp(j) = x(j) - dx(i);
      fnn = funkeval(xtemp);
      Jacobian(i,j) = (fpp - fpn -fnp + fnn)/(4*dx(i)*dx(j));
```

```
   end
end
for i = 2:1:n
   for j = 1:1:i-1
      Jacobian(i,j) = Jacobian(j,i);
   end
end

function fobj = funkeval(x)
fobj = (x(1)-3)^2 + (x(2)-7)^2 + x(1)*x(2)^2 - x(1)^2*x(2) + 4;
```

An example of using nrnd_optn_short is given below.

```
» [x,err,f] = nrnd_optn_short([4,8],1.0e-6,1)
iter =    1, err = 2.81e+000 f = 3.54e+001
 iter =    2, err = 8.66e-001 f = 6.81e+000
 iter =    3, err = 1.11e-001 f = 7.30e-001
 iter =    4, err = 1.58e-003 f = 1.02e-002
 iter =    5, err = 8.22e-007 f = 5.56e-006

x =   1.23017202549369   3.47805528788098
err =    8.220328680806535e-007
f =    5.555092652529348e-006
```

So in 5 iterations, the Newton-Raphson method with numerical derivatives found the same minimum at $x_1 = 1.23$ and $x_2 = 3.47$ with a relative error on x of 8.22×10^{-7} and an RMS error on the residual of 5.56×10^{-6}. The value of the objective function at the minimum is 29.154, determined by substitution of the converged solution into the objective function.

8.9. Problems

Problems are located on course website.

References

Chapra, S.C., Canale, R.P., Numerical Methods for Engineers, 2nd Ed., McGraw-Hill, New York, 1988.

Montgomery, D.C., Runger, D.C., Applied Statistics and Probability for Engineers, 2nd Edition, John Wiley & Sons, New York, 1999.

Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P., Numerical Recipes in Fortran 77: The Art of Scientific Computing, 2nd Edition, Volume 1 of Fortran Numerical Recipes, Cambridge University Press, 1992.

Walpole, R.E., Myers, R.H, Myers, S.L., Probability and Statistics for Engineers and Scientists, 6th Edition, Prentice Hall, Upper Saddle River, New Jersey, 1998.