WROCLAW UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF ELECTRONICS

Field:	Electronics
Speciality:	Advanced Applied Electronics

Optimization Methods: Nonlinear Equations

Author:
Jaroslaw M. Szumega

Supervisor:
Rafal Zdunek, D.Sc, K-4/W4

Grade:

WROCŁAW 08.06.2017

Contents

1 Solution to the given problems **1**

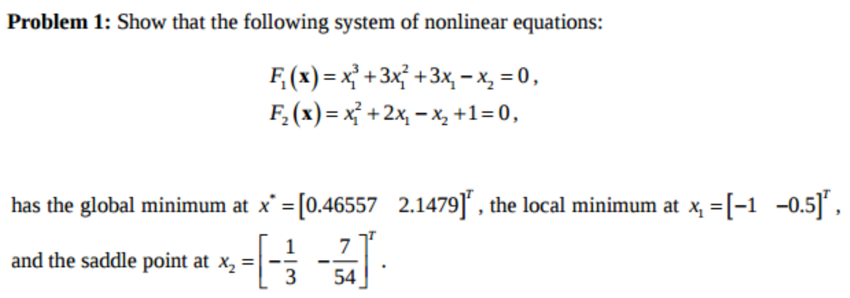
2 Listings of algorithms **9**

2.1 Coded selected algorithms 9

Bibliography **11**

Chapter 1

Solution to the given problems



Problem 1: Show that the following system of nonlinear equations:

$$F_1(\mathbf{x}) = x_1^3 + 3x_1^2 + 3x_1 - x_2 = 0,$$
$$F_2(\mathbf{x}) = x_1^2 + 2x_1 - x_2 + 1 = 0,$$

has the global minimum at $x^* = [0.46557 \quad 2.1479]^T$, the local minimum at $x_1 = [-1 \quad -0.5]^T$, and the saddle point at $x_2 = \left[-\frac{1}{3} \quad -\frac{7}{54}\right]^T$.

In general, the unconstrained nonlinear least squares problem is equivalent to:

$$\min_x \sum_{i=1}^{N} F_i^2(x)$$

with the objective function:

$$f(x) = \frac{1}{2}\sum_{i=1}^{N} F_i^2(x)$$

In the case of the following task, objective function will be

$$\begin{aligned}
f(x) &= \frac{1}{2}(F_1^2 + F_2^2) \\
&= \frac{1}{2}[(x_1^6 + 6x_1^5 + 15x_1^4 - 2x_1^3x_2 + 18x_1^3 - 6x_1^2x_2 + 9x_1^2 - 6x_1x_2 + x_2^2) \\
+ &\quad (x_1^4 + 4x_1^3 - 2x_1^2x_2 + 6x_1^2 - 4x_1x_2 + 4x_1 + x_2^2 - 2x_2 + 1)] \\
&= \frac{1}{2}(x_1^6 + 6x_1^5 + 16x_1^4 - 2x_1^3x_2 + 22x_1^3 - 8x_1^2x_2 + 15x_1^2 - 10x_1x_2 + 4x_1 + 2x_2^2 - 2x_2 + 1)
\end{aligned}$$

And the points to check are:

$$\begin{aligned}
x^* &= [0.46557 \; 2.1479]^T \\
x_1 &= [-1 \; -0.5]^T \\
x_2 &= [-\frac{1}{3} \; -\frac{7}{54}]^T
\end{aligned}$$

Firstly, we can calculate the cost function gradient to verify above–mentioned points:

$$\nabla f(x^*) = 0$$

$$\frac{\delta f(x)}{\delta x_1} = 3x_1^5 + 15x_1^4 + 32x_1^3 - 3x_1^2 x_2 + 33x_1^2 - 8x_1 x_2 + 15x_1 - 5x_2 + 2$$
$$\frac{\delta f(x)}{\delta x_2} = -x_1^3 - 4x_1^2 - 5x_1 + 2x_2 - 1$$

And in fact, the mentioned in task description points are recognized as a solutions of the cost function gradient.

To check their nature, we need to calculate the Hessian and evaluate it according to the stationary points.

$$\nabla^2 f(x) = \begin{bmatrix} \frac{\delta^2 f(x)}{\delta x_1^2} & \frac{\delta^2 f(x)}{\delta x_1 x_2} \\ \frac{\delta^2 f(x)}{\delta x_1 x_2} & \frac{\delta^2 f(x)}{\delta x_2^2} \end{bmatrix}$$

$$\nabla^2 f(x) = \begin{bmatrix} 15x_1^4 + 60x_1^3 + 96x_1^2 - 6x_1 x_2 + 66x_1 - 8x_2 + 15 & -3x_1^2 + 8x_1 + 5 \\ -3x_1^2 + 8x_1 + 5 & 2 \end{bmatrix}$$

$$for\ point\ [0.46557\ 2.1479] = \begin{bmatrix} 50.11 & -9.37 \\ -9.37 & 2 \end{bmatrix}$$
$$det > 0, M1 > 0, f(x) = 5.01e-11,\ there\ is\ a\ minimum\ (global)$$

$$for\ point\ [-1\ \ -0.5] = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$
$$det > 0, M1 > 0, f(x) = 0.25,\ there\ is\ a\ minimum$$

$$for\ point\ [-\frac{1}{3}\ \ -\frac{7}{54}] = \begin{bmatrix} 2.41 & -2.66 \\ -2.66 & 2 \end{bmatrix}$$
$$det < 0, M1 > 0, f(x) = 0.33,\ there\ is\ a\ saddle\ point$$

To make all the calculations above, the following script in Python was written to perform symbolic computations:

```python
#!/usr/bin/python

from sympy import diff, Symbol, latex, Matrix


def optimization(function, x1, x2):

    print latex(function)

    #calculating the gradient's elements
    firstx1 = diff(function, x1)
    firstx2 = diff(function, x2)

    print firstx1.expand()
    print firstx2.expand()


    #calculations for Hessian elements
    secondx1x1 = diff(function, x1, x1)
    secondx1x2 = diff(function, x1, x2)
    secondx2x1 = diff(function, x2, x1)
    secondx2x2 = diff(function, x2, x2)

    print "\n Hessian elements:\n\n"
    print latex(secondx1x1) +"\t" + latex(secondx1x2)
    print latex(secondx2x1) + "\t"+latex(secondx2x2)


    point1 = [0.46557, 2.1479]
    point2 = [-1.0, -0.5]
    point3 = [-1.0/3, -7.0/54]

    point = point3
    print secondx1x1.subs(x1, point[0]).subs(x2, point[1]);
    print secondx1x2.subs(x1, point[0]).subs(x2, point[1]);
    print secondx2x1.subs(x1, point[0]).subs(x2, point[1]);
    print secondx2x2.subs(x1, point[0]).subs(x2, point[1]);
    print "\n\n"
    print function.subs(x1, point[0]).subs(x2, point[1]);


def main():
    x1 = Symbol('x1')
    x2 = Symbol('x2')

    f1 = x1**3 + 3*x1**2 + 3*x1 - x2
    f2 = x1**2+2*x1 -x2 + 1
    f = (f1**2 + f2**2)/2

    function = f.expand()
    optimization(function, x1,x2)

main()
```

Problem 2: Solve the following system of nonlinear equations with the selected nonlinear least squares methods.

$$F_1(\mathbf{x}) = x_1^3 - 3x_1^2 + 3x_1 - x_2 - 3 = 0,$$
$$F_2(\mathbf{x}) = x_1^2 - 2x_1 - x_2 = 0.$$

Draw the contour lines of the cost function in the nonlinear least square problem. Check the optimality points.

To resolve this nonlinear LS problem, the Newton and Broyden algorithm were used.

```
Newton's Method
iteration =  5

x =
2.4656  1.1479

Elapsed time is 0.000832081 seconds.


Broyden Method
iteration =  424

x =
2.4656 1.1479

Elapsed time is 0.055002 seconds.
```

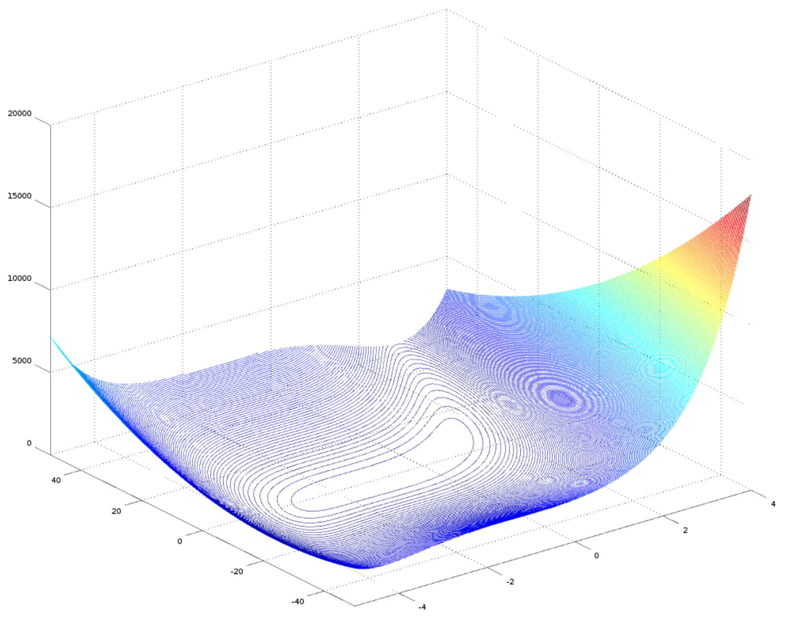


Figure 1.1 Contour plot of cost function.

Problem 3: Solve the following system of nonlinear equations:

$$2x_1 - x_2 = \exp(-x_1),$$
$$-x_1 + 2x_2 = \exp(-x_2),$$

starting from $\mathbf{x}_0 = [-5 \quad -5]^T$.

```
Newton Method
x =
0.56714
0.56714

iter_newton =  8
Elapsed time is 0.00742817 seconds.


Broyden Method
x =
0.56714
0.56714

iter_broyden =  8
Elapsed time is 0.00764704 seconds.
```

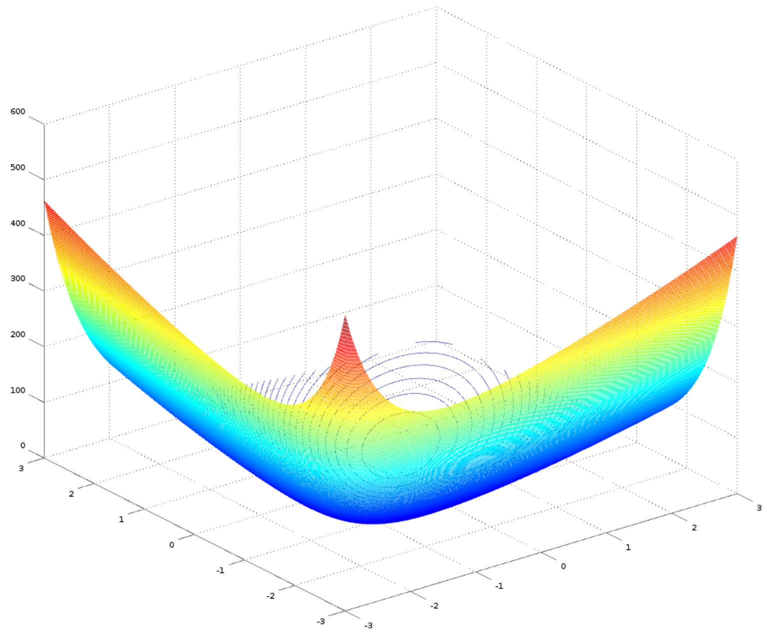


Figure 1.2 Contour plot of cost function

Problem 4: Find $\mathbf{x}$ that minimizes $\sum_{k=1}^{10}\left(2+2k-\exp(kx_1)-\exp(kx_2)\right)^2$, starting at the point $\mathbf{x}_0=[0.3 \quad 0.4]^T$.

To complete this task, the **lsqnonlin** function was used.
In order to make this experiment more interesting, here will be shown its usage in two ways:

- without using Jacobian calculations,
- with Jacobian calculations. It will be turned on using Octave options.

There fore there was prepared two versions of evaluation function to pass to the lsgnonlin:

```
function [F,J] = fun_4(x)
    k = 1:10;
    F = 2 + 2*k-exp(k*x(1))-exp(k*x(2));
endfunction

function [F,J] = fun_4_Jacobian(x)
    k = 1:10;

    F = 2 + 2*k-exp(k*x(1))-exp(k*x(2));

    J = zeros(10,2);
    J(k,1) = -k.*exp(k*x(1));
    J(k,2) = -k.*exp(k*x(2));
endfunction
```

The call end execution looked like this:

```
x0 = [0.3; 0.4];
tic
[x,resnorm,res,eflag, output]= lsqnonlin(@fun_4,x0);
toc


opts = optimset ("Jacobian", "on")
tic
[x,resnorm,res,eflag,output_jacobian] = lsqnonlin(@fun_4_Jacobian,x0
    ,[],[],opts);
toc
```

And the following results were obtained:

```

Calculations without Jacobian:

Elapsed time is 0.0594881 seconds.
x = 0.25803     0.25993

output =
niter =  17



```

```
Calculations with Jacobian

Elapsed time is 0.1645 seconds.
x = 0.25782      0.25823

output_jacobian =
niter =  122
```

As it can be observed, both methods obtained the same values, however Jacobian calculations lasted 3 times more and also performed more iterations (almost 7 times).

Problem 5: The Shockley ideal diode equation is given by: $I = I_s\left(\exp\left\{\frac{U}{n\varphi_T}\right\}-1\right)$. Assuming $I_s = 10^{-10} A$, $n = 1.2$, $\varphi_T = 26mV$, determine the values of the forward current for the across voltages $U = [0 \quad 0.01 \quad 0.02 \quad \dots \quad 1]^T$ V. Then solve the inverse problem: having the I-V look-up table, the thermal voltage φ_T, and the Shockley ideal diode equation, try to estimate the reverse bias saturation current I_s and the ideality factor *n*.

To calculate this task, the model of Shockley diode was coded into a function:

```
function [F]=fun_5_Shockley(x,U)
fi=0.026;

F = x(1) * (exp(U/(x(2)*fi))-1);
end
```

After projection of the data to the **I** current values, it was possible to use Octave curve fitting:

```
x_est = lsqcurvefit(@fun_5_Shockley, x0, Uvalues, Ivalues);
```

The result of curve fitting is quite promising. For x = [1.0540e-09, 1.1924e+00], we obtained:

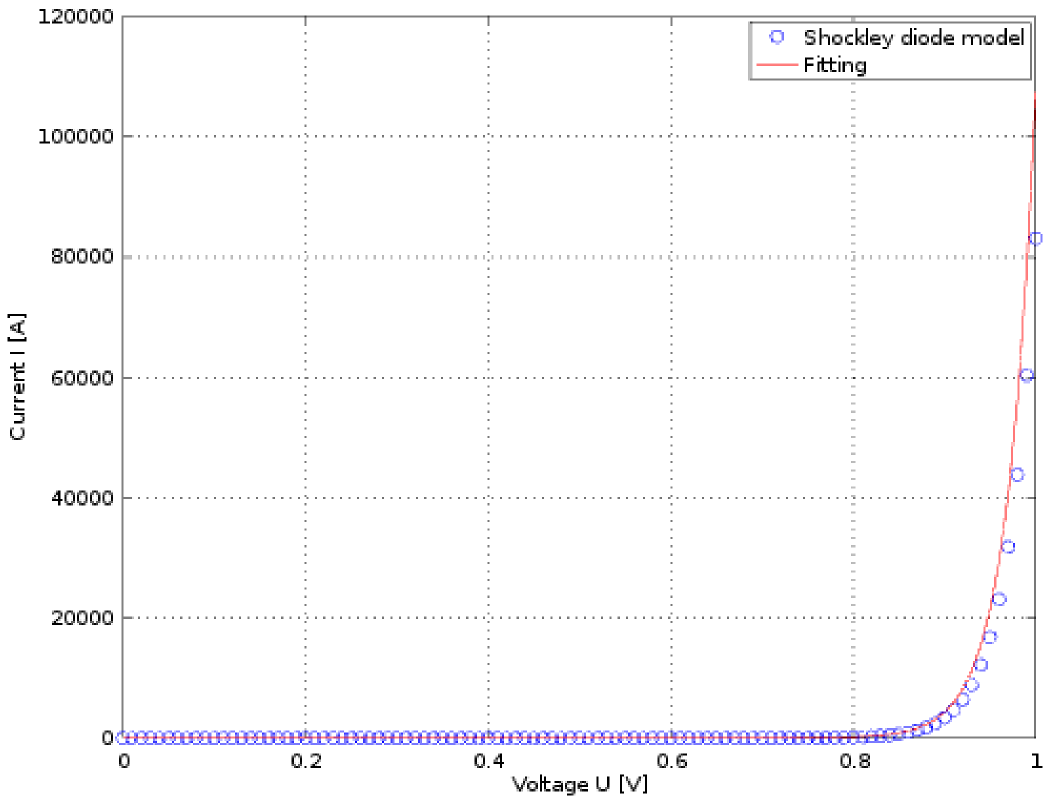


Figure 1.3 Curve fitting (LS) with Octave.

Chapter 2

Listings of algorithms

2.1 Coded selected algorithms

Algorithm 1 - Newton algorithm

```
function [x,k] = newton(fun,x,err)
    stop = 2*err;
    iter = 0;
    while stop > err
        [F,J] = fun(x);
        dx = J\(-F);
        x = x+dx;
        [F,J] = fun(x);
        stop = max(abs(F));
        iter = iter + 1;
    endwhile

    iteration = iter
endfunction
```

Algorithm 2 - Broyden method

```
function [xv,it]=broyden(f,x,tol,n)

    fr=zeros(n,1); it=0; xv=x;

    Br=eye(n);
    fr=f(xv);

    while norm(fr)>tol
        it=it+1;
        pr=-Br*fr;
        tau=1;

        xv1=xv+tau*pr;
        xv=xv1;

        oldfr=fr;
        fr=f(xv);

        %Update approximation to Jacobian using Broydens formula
        y=fr-oldfr;
        oldBr=Br;
        oyp=oldBr*y-pr;
        pB=pr'*oldBr;

        for i=1:n
            for j=1:n
                M(i,j)=oyp(i)*pB(j);
            end;
        end;
        Br=oldBr-M./(pr'*oldBr*y);
    end;
endfunction
```

Bibliography

[1] J. Nocedal, S. J. Wright, Numerical Optimization, Springer, 1999,

[2] Zdunek R., Optimization Methods - lecture slides.

[3] Mathworks webpage, ”Unconstrained Optimization Algorithms”, https://www.mathworks.com/help/optim/ug/unconstrained-nonlinear-optimization-algorithms.html

[4] Nonlinear Programming course webpage, North Carolina State University, http://www4.ncsu.edu/ kksivara/ma706/