Optimization

An optimization problem consists of finding extrema of a function. This occurs frequently in engineering problems, for instance when the place of maximum deflection of a structure is to be determined, the position of a point with minimum error is looked for in a surveying adjustment, or to find the highest contamination of samples in a water quality test.

Different procedures are invented to solve such problems. These mostly involve minimization, therefore when the type of the extremum is maximum, the idea is to find minimum of the negative of the function, i.e. \( \max(f(x)) = \min(-f(x)) \). Extrema are always considered in a given interval or domain. When there are several local minima, the smallest one is called the global minimum (point \( P_2 \) in figure below).

Optimization may be unconditional or conditional (unconstrained or constrained). In constrained optimization problems those minima are searched that also meet an additional constraint or condition. One or several of such conditions may be given, they may be equalities or inequalities, linear or nonlinear. Different procedures are applied in different cases (e.g. Lagrange method, penalty method, Karush-Kuhn-Tucker conditions, linear programming).

**Optimization of functions of one variable**

Let's have a look on the following example:

![Diagram of a beam with linear loading](image)

In this case we have a fixed ended beam with an \( l \) cross-section which is loaded linearly.
distributed. The deflection in the y axis could be determined with the following formula:

\[ y = \frac{q_0}{120LEI} (x^5 - 5Lx^4 + 7L^2x^3 - 3L^3x^2), \]

where \( x \) denotes the position along the beam, \( L = 3000 \text{mm} \) denotes the beam length, \( q_0 \) is 15kN/m=15N/mm (distributed load), and \( EI \) is the flexural rigidity, where \( E=70000 \text{ N/mm}^2 \), \( I = 5.29 \times 10^7 \text{ mm}^4 \).

What is the deflection at 1 and 2 meters?

```matlab
% Deflection
clear all; clc; close all;
E = 70000; I = 5.29e7; q0 = 15; L = 3000; EI = E*I;
y = @(x) q0/(120*L*EI)*(x^5-5*L*x^4+7*L^2*x^3-3*L^3*x^2)

y = function_handle with value:
    @(x)q0/(120*L*EI)*(x^5-5*L*x^4+7*L^2*x^3-3*L^3*x^2)

Let us plot the deflection along the beam:

figure(1); fplot(y,[0 L])
```

Warning: Function behaves unexpectedly on array inputs. To improve performance, properly vectorize function to return an output with the same size and shape as the input arguments.

\[ y(1000), y(2000) \text{ % deflection at 1 and 2 meters: } -0.3601 \text{ and } -0.3151 \text{ mm} \]
We will consider two ways to find minimum of a function of one variable.

**Ternary search**

In ternary search an interval \([a, b]\) must be specified that contains a minimum of the function. In this interval the function is unimodal: monotonically decreases until it reaches the minimum and after that monotonically increases. The starting interval then should be shrunk till we find the solution. Choose two internal points \((x_1, x_2)\) and evaluate our function there.

Due to monotonicity of the function the place of the minimum must be in the interval between the point with the least function value and of its two neighbors. Hence if \(f(x_1) < f(x_2)\), the minimum must be in the interval \([a, x_2]\), or if \(f(x_1) > f(x_2)\) it must be in the interval \([x_1, b]\). See figure. Next we choose again two points in the new interval and repeat this process until interval length drops below a specified threshold.

Ternary search algorithm works by uniform placement of points at 1/3 and 2/3 of the interval length. Matlab implementation of this algorithm is in file `ternary.m`.

Let us find the maximum deflection by this method. The initial interval from figure is chosen as \([1000, 2000]\).

```matlab
% ternary search - uniform placement of points
[x1 i1] = ternary(y,1000,2000,1e-6)
```

\[x1 = 1.4259e+03\]
\[i1 = 50\]

\[y1 = y(x1) \approx -0.4293\]

\[y1 = -0.4293\]

50 iterations were required to find the location of the minimum point. The location is at 1425.9 mm.
and it's value is $-0.4293$ mm.

**Golden-section search**

We improve ternary search by using the golden ratio. By using golden ratio an interval $L$ can be divided

\[ L = L_1 + L_2 \]

such that the ratio of the longer section to the whole is the same as the ratio of the shorter section to the longer one.

\[ R = \frac{L_2}{L} = \frac{L_1}{L_2} \]

From the above two equations we get

\[ R^2 + R - 1 = 0 \]

The golden ratio is the only one positive root of this quadratic equation.

\[ R = \frac{\sqrt{5} - 1}{2} \approx 0.618 \]

Let us use this ratio for improved minimum search by modifying selection of the internal points within our interval.

Place internal points symmetrically such both have distances $0.618 \cdot L$ from the interval endpoints. Shrink our interval depending on function values. Thanks to the special property of golden ratio one internal point of the new interval will be the same as one of the internal points of the old interval. We see on the figure that point $x_2$ of the new interval coincides with point $x_1$ of the old interval. Hence it is not necessary to evaluate function value at this point again. For complicated functions this can be a considerable speedup. The Matlab implementation of this procedure is in golden.m.

Let us find and plot minimum oxygen concentration also by this method.

% golden-section search
The number of iterations reduced from 50 to 42. The number of function evaluations is also reduced from 100 (50 iterations * 2) to 44 (2 evaluation in the first iteration, and 1 in the rest). This is a great advantage over ternary search for complicated functions.

**Newton’s method**

Let us use Newton’s method for optimization. Instead of solving equation \( f(x) = 0 \) for the roots we will solve equation \( f'(x) = 0 \). The formula for optimization is

\[
x_{i+1} = x_i - \frac{f'(x_i)}{f''(x_i)}
\]

as a reminder the formula for the zero point: \( x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \)
Both first and second derivatives of the function are required for optimization by Newton's method. Let us solve our problem by Newton's method. Specify one endpoint of the interval as initial value for comparison, e.g. 5. (Of course a better initial value, e.g. 4 could be selected from the figure). The \texttt{matlabFunction} procedure is used for converting symbolic derivatives into functions.

```matlab
%% Newton's method
syms EI L q0 x
ys = q0/(120*L*EI)*(x.^5-5*L*x.^4+7*L^2*x.^3-3*L^3*x.^2)

ys =
\frac{q_0 \left( 3 L^3 x^2 - 7 L^2 x^3 + 5 L x^4 - x^5 \right)}{120 EI L}

dx = \text{diff}(ys, x) \% -(q0*(6*L^3*x - 21*L^2*x^2 + 20*L*x^3 - 5*x^4))/(120*EI*L)

dx =
\frac{-q_0 \left( 6 L^3 x - 21 L^2 x^2 + 20 L x^3 - 5 x^4 \right)}{120 EI L}

ddx = \text{diff}(ys, x, 2) \% -(q0*(6*L^3 - 42*L^2*x + 60*L*x^2 - 20*x^3))/(120*EI*L)

ddx =
\frac{-q_0 \left( 6 L^3 - 42 L^2 x + 60 L x^2 - 20 x^3 \right)}{120 EI L}

% Convert symbolic expressions into functions
E = 70000; I = 5.29e7; q0 = 15; L = 3000; EI = E*I;
dx = \text{subs}(dx), ddx = \text{subs}(ddx)

dx =
\frac{x^4}{17774400000000000} - \frac{x^3}{1481200000000} + \frac{9 x^2}{42320000000} - \frac{27 x}{148120000}

ddx =
\frac{x^3}{4443600000000000} - \frac{3 x^2}{1481200000000} + \frac{9 x}{21160000000} - \frac{27}{148120000}

dxf = \text{matlabFunction}(dx)

dxf = \text{function_handle with value:}
@(x)x.*(-1.822846340804753e-6)+x.^2.*2.126654064272212e-9-x.^3.*6.751282743721307e-13+x.^4/

dddf = \text{matlabFunction}(ddx)

dddf = \text{function_handle with value:}
 @(x)x.*4.253308128544423e-9-x.^2.*2.025384823116392e-12+x.^3.*2.250427581240436e-16-1.8221

% solution by Newton's method
[xn in] = newton(dxf, dddf, 2000, 1e-6, 100) \% cn = 3.3912; in = 6
\[ x_n = 1.4257 \times 10^3 \]
\[ i_n = 3 \]

Now only 3 iterations were needed for the solution. This shows that Newton's method in case of convergence converges much more quickly. Try to change the initial value, give first the other interval endpoint 0, then try a better approximation 4 and finally use a more distant value, e.g. 10. Check the results.

**Using Matlab's Built-in function**

There is a built-in function of Matlab for optimization, e.g. `fminsearch`. This function uses the Nelder-Mead simplex method.

```matlab
% built-in Matlab function - fminsearch
xmin = fminsearch(y,2000) % 3.3912
```

\[ xmin = 1.4259 \times 10^3 \]

```matlab
% with detailed output
[x,fval,exitflag,output] = fminsearch(y,5)
```

\[ x = 1.4259 \times 10^3 \]
\[ fval = -0.4293 \]
\[ exitflag = 1 \]
\[ output = struct with fields: \]
\[ \quad \text{iterations: 36} \]
\[ \quad \text{funcCount: 72} \]
\[ \quad \text{algorithm: 'Nelder-Mead simplex direct search'} \]
\[ \quad \text{message: 'Optimization terminated: the current x satisfies the termination criteria ['} \]

\[ i = output.iterations \% i = 36 \]

\[ i = 36 \]

```matlab
% Algorithm: Nelder-Mead simplex direct search
```

**Finding maxima**

A slight modification of ternary/golden-section search algorithm would make them capable of finding maxima; however, it is easier to find the minimum of the negative of the function. Consider an example. Let us find the maximum of the function \( f(x) = -x^2 + 6x + 1 \)

```matlab
clear all; clc; close all;
f = @(x) -x^2 + 6*x + 1

f = function_handle with value:
   @(x)-x^2+6*x+1

ezplot(f)
```
Warning: Function failed to evaluate on array inputs; vectorizing the function may speed up its evaluation and avoid the need to loop over array elements.

\[
fm = @(x) -f(x)
\]

\[
fm = \text{function_handle with value:}
\]
\[
@(x)-f(x)
\]

\[
xm = \text{fminsearch(fm,4)} \% 3.0000
\]

\[
xm = 3.0000
\]

\[
fmax = f(xm) \% 10.0000
\]

\[
fmax = 10.0000
\]

\[
\text{hold on; plot(xm, fmax, 'ro')}
\]

For finding the value of maximum make substitution should be made into the original function.

**Optimization of a function in several variables**

Optimization is frequently required of a function of not only one but of several variables. Such are problems of finding extremal points of a surface, finding maximum deflections of a 3D truss in \(x,y\) directions, optimal placing of crossings of a traffic network by minimizing ranges, etc. We have several methods for solving unconstrained multivariate optimization problems as well. For example Newton method in several variables, gradient method, Nelder-Mead simplex methods
Positioning in multivariate case

Let us now consider an optimization problem in two variables. Revisit our former problem of mobile phone positioning by intersection using distances. We had to solve a system of two nonlinear equations for the two unknowns, but in practice we frequently have additional measurements. For 3 or more distances, when we have small measurement errors, there are misfits in positioning and we need adjustments. Very similar to overdetermined linear systems here also residuals (square sum of residuals) are minimized by least squares. This problem can be solved by linearization as well as by multivariate optimization algorithms. Now there are distance measurements for 4 mobile masts and our task is to get our most probable location.

<table>
<thead>
<tr>
<th>Mobile mast number</th>
<th>Coordinate X [m]</th>
<th>Coordinate Y [m]</th>
<th>Mast-terminal distance [m]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>561</td>
<td>1487</td>
<td>2130</td>
</tr>
<tr>
<td>2</td>
<td>5203</td>
<td>4625</td>
<td>5620</td>
</tr>
<tr>
<td>3</td>
<td>5067</td>
<td>-5728</td>
<td>6040</td>
</tr>
<tr>
<td>4</td>
<td>1012</td>
<td>5451</td>
<td>5820</td>
</tr>
</tbody>
</table>

Locii of measured distances are circles with the following implicit equation:

\[(x - x_i)^2 + (y - y_i)^2 - r_i^2 = 0\]
where \( x_i, y_i \) denote coordinates of mobile masts, \( x, y \) are that of the unknown position.

Plot circles in Matlab and check their intersections. Place measurements into vectors and plot mobile mast positions.

```matlab
clear all; clc; close all;
xt = [561; 5203; 5067; 1012]

xt =
    561
    5203
    5067
    1012

yt = [487; 4625; -5728; 5451]

yt =
    487
    4625
   -5728
   5451

rm = [2130; 5620; 6040; 5820]

rm =
    2130
    5620
    6040
    5820

% centers of circles
figure(1); hold on;
plot(xt, yt, 'r*')
```

Define a generic function of a circle and by using symbolic variables \( x, y \) plot all circles.

```matlab
% Generic function of a circle
eq = @(x,y,xt,yt,rm) (x-xt).^2 + (y-yt).^2 - rm.^2

eq = function_handle with value:
    @(x,y,xt,yt,rm)(x-xt).^2+(y-yt).^2-rm.^2

% plot circles
sym x y
E = eq(x,y,xt,yt,rm)
```

\[
\begin{pmatrix}
(x - 561)^2 + (y - 487)^2 - 4536900 \\
(x - 5203)^2 + (y - 4625)^2 - 31584400 \\
(x - 5067)^2 + (y + 5728)^2 - 36481600 \\
(x - 1012)^2 + (y - 5451)^2 - 33872400
\end{pmatrix}
\]
for i = 1:4
    fimplicit(E(i),[-5000 12000])
end
axis equal
title('Positioning by adjustment')

Zoom in the area of intersection.

% Area of intersection
figure(2); hold on;
for i = 1:4
    fimplicit(E(i),[2000 3000 -500 100])
end
axis equal
title('Positioning by adjustment')
These four circles do not have a common intersection point but define an area where our assumed position is. This most probable position can be computed by minimizing sum of squared residuals. The function \( f \) to be minimized is the sum of squared residuals

\[
f(x, y) = \sum_{i=1}^{n} (x - x_i)^2 + (y - y_i)^2 - r_i^2
\]

Let us define our cost function and make contour and 3D mesh plots of it in Matlab.

% cost function to be minimized
f = sum((eq(x,y,xt,yt,rm)).^2)

\( f = ((x - 561)^2 + (y - 487)^2 - 4536900)^2 + ((x - 5203)^2 + (y - 4625)^2 - 31584400)^2 + ((x - 1012)^2 + (y - 4625)^2 - 31584400)^2 \)

F = matlabFunction(f)  \% convert symbolic \( f \) into Matlab function

F = function_handle with value:
@(x,y)((x-5.61e2).^2+(y-4.87e2).^2-4.5369e6).^2+((x-5.203e3).^2+(y-4.625e3).^2-3.15844e7).^2

fcontour(f,[2000 3000 -500 100]);
Positioning by adjustment

```matlab
figure(3)
fsurf(f, [2000 3000 -500 100])
```
The minimum of this function is our most probable position. To find it use Newton’s method in several variables.

**Newton’s Method in several variables**

The Newton’s method formula for a single variate case is the following:

\[ x_{i+1} = x_i - \frac{f(x_i)}{f''(x_i)} \]

Generalization of univariate Newton’s formula for several variables is possible by replacing first derivative with gradient vector \((\nabla f)\) and second derivative with Hessian matrix \((H)\). Variables must be specified as a vector \((x)\). Our formula is

\[ x_{i+1} = x_i - H^{-1}(x_i) \cdot \nabla f(x_i) \]

where components of the Hessian are second partial derivatives of \(f(x)\), components of the gradient vector are first partial derivatives. The Hessian can also be computed as the Jacobian matrix of the gradient vector of a function.

\[
\nabla f(x,y) = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} ; \quad H(x,y) = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix}
\]

The function `gradmulti.m` implements multivariate Newton's method in Matlab. Its input arguments are the gradient vector, the Hessian, an initial guess \(x_0\), tolerance \(\text{eps}\) for stopping iteration and maximum iteration number \(\text{nmax}\).

Outputs of the function are the solution \(x1\), iteration number \(i\) and iteration steps are stored in matrix \(X\).

**Positioning by Multivariate Newton’s Method**

Let us compute gradient and Hessian with Matlab. Matlab function `gradient` is suitable for both numeric and symbolic computation of the gradient vector. Let us plot numerically computed gradient vectors for better visualization. First we use `meshgrid` to create a grid and for grid points we then plot computed gradient vectors with the `quiver` function.

```
% Visualization of numerically computed gradient vectors
[X,Y] = meshgrid(2000:100:3000, -500:50:100);
Z = F(X,Y);
[px,py] = gradient(Z); % numerical computation of gradient vectors
figure(2)
quiver(X,Y,px,py)
```

Optimization by multivariate Newton's method requires gradient vectors and the Hessian not numerically but given as functions of vector variables. These functions can be determined symbolically with the commands `gradient` and `hessian` applied for the symbolic function \(f\).

```
% gradient vector symbolically
```
\( G = \text{gradient}(f) \)

\[
G = \\
\left( \begin{array}{l}
(2 \cdot (2 \cdot x - 1122) \cdot \sigma_4 + 2 \cdot (2 \cdot x - 10406) \cdot \sigma_3 + 2 \cdot (2 \cdot x - 2024) \cdot \sigma_2 + 2 \cdot (2 \cdot x - 10134) \cdot \sigma_1) \\
(2 \cdot (2 \cdot y - 974) \cdot \sigma_4 + 2 \cdot (2 \cdot y - 9250) \cdot \sigma_3 + 2 \cdot (2 \cdot y - 10902) \cdot \sigma_2 + 2 \cdot (2 \cdot y + 11456) \cdot \sigma_1) \\
\end{array} \right)
\]

where

\[
\sigma_1 = (x - 5067)^2 + (y + 5728)^2 - 36481600
\]

\[
\sigma_2 = (x - 1012)^2 + (y - 5451)^2 - 33872400
\]

\[
\sigma_3 = (x - 5203)^2 + (y - 4625)^2 - 31584400
\]

\[
\sigma_4 = (x - 561)^2 + (y - 487)^2 - 4536900
\]

\[
% \text{Hessian computed symbolically} \text{H} = \text{hessian}(f)
\]

\[
H = \\
\left( \begin{array}{l}
\sigma_5 + \sigma_4 + \sigma_2 + \sigma_8 + \sigma_7 + \sigma_6 + 2 \cdot (2 \cdot x - 1122)^2 + 2 \cdot (2 \cdot x - 2024)^2 + 2 \cdot (2 \cdot x - 10134)^2 + \\
\sigma_1
\end{array} \right)
\]

where

\[
\sigma_1 = 2 \cdot (2 \cdot x - 1122) \cdot (2 \cdot y - 974) + 2 \cdot (2 \cdot x - 2024) \cdot (2 \cdot y - 10902) + 2 \cdot (2 \cdot x - 10406) \cdot (2 \cdot y - 925)
\]

\[
\sigma_2 = 4 \cdot (x - 5203)^2
\]

\[
\sigma_3 = 4 \cdot (x - 5067)^2
\]

\[
% \text{Rewrith the H and G as functions of a vector variable} \text{G} = \text{matlabFunction}(G) \text{ % convert symbolic expression of G into a function}
\]

\[
G = \text{function\_handle with value:}
\]

\[
@(x,y)\left((x\cdot 2.0\cdot 1.122\cdot e3)\cdot ((x\cdot 5.61\cdot e2)\cdot ^2\cdot (y\cdot 4.87\cdot e2)\cdot ^2\cdot 4.536\cdot e6\cdot 2.0\cdot (x\cdot 2.0\cdot 1.040\cdot e4)\cdot (\right)
\]

\[
H = \text{matlabFunction}(H) \text{ % convert symbolic expression of H into a function}
\]

\[
H = \text{function\_handle with value:}
\]

\[
@(x,y)\text{reshape}([x\cdot 5.61\cdot e2\cdot 2.0\cdot 4.0\cdot 1.012\cdot e3\cdot 2.0\cdot 4.0\cdot x\cdot 5.067\cdot e3\cdot 2.0\cdot 4.0\cdot 5.203\cdot e3\cdot 2.0\cdot 4.0\cdot (x\cdot 5.203\cdot e3\cdot 2.0\cdot 4.0\cdot x\cdot 5.203\cdot e3\cdot 2.0\cdot 4.0/((x\cdot 5.61\cdot e2)\cdot ^2\cdot (y\cdot 4.87\cdot e2)\cdot ^2\cdot 4.536\cdot e6\cdot 2.0\cdot (x\cdot 2.0\cdot 1.040\cdot e4)\cdot (\right)
\]

\[
G = @(x) \text{G}(x(1),x(2)) \text{ % rewrite function with vectorial argument}
\]

\[
G = \text{function\_handle with value:}
\]
\[(x)G(x(1),x(2))\]

\[H = @(x) H(x(1),x(2)) \% rewrite function with vectorial argument\]

\[H = \text{function-handle with value:}\]
\[
@(x)H(x(1),x(2))
\]

Find initial value from the figure and call gradmulti.m.

\[x0 = [2400; -300]\]

\[x0 =\]
\[
2400
-300
\]

\[[p\ i\ pp] = \text{gradmulti}(G,H,x0,1e-6,100)\]

\[p =\]
\[
1.0e+03 *
2.4547
-0.2469
\]
\[i = 4\]
\[pp =\]
\[
1.0e+03 *
2.4000 2.4529 2.4547 2.4547 2.4547
-0.3000 -0.2476 -0.2469 -0.2469 -0.2469
\]

\[% Plot solution\]
\[
\text{plot}(p(1),p(2),'r*')
\]

Our computation converged to the minimum within specified tolerance very quickly in only 4 iterations.

**Using Matlab's Built-in fminsearch Function**

There are built-in Matlab functions for multivariate optimization, namely \texttt{fminsearch}, which uses Nelder-Mead simplex method and \texttt{fminunc}, which uses quasi-Newton minimization. Simplex method is preferred in the case when it is hard to compute derivatives of the function. This method works by starting from an initial polyhedron (simplex), which is a triangle in 2 dimensions. Then 3 vertices of this simplex is changed (by stretching, shrinking, mirroring) to follow the shape of the function and eventually it shrinks to the minimum point. To grasp the idea the following animation can help: [https://en.wikipedia.org/wiki/File:Nelder-Mead_Himmelblau.gif](https://en.wikipedia.org/wiki/File:Nelder-Mead_Himmelblau.gif)

Let us solve this problem by simplex method. We must rewrite function \(F\) in terms of a vector variable.

\[x0 = [2400; -300]\]
x0 =
    2400
    -300

F = @(x) F(x(1),x(2)) % rewrite function with vectorial argument

F = function_handle with value:
    @(x)F(x(1),x(2))

sol = fminsearch(F,x0)

sol =
    1.0e+03 *
    2.4547
    -0.2469

plot(sol(1),sol(2), 'ks')

Let us check residuals of measured distances and distances between mobile masts and adjusted position.

% residuals
ex = xt - sol(1);
ey = yt - sol(2);
er = rm - sqrt(ex.^2+ey.^2)
er =
   99.0847
   26.3188
  -31.7595
  -57.7440

%  99.0847
%  26.3188
% -31.7595
% -57.7440

Plot solution in 3D:

```matlab
figure(3); hold on;
plot3(sol(1),sol(2),F(sol),'r*')
```

Remark: We considered local optimization algorithms that are designed to find local optimum near a given initial guess. To find global minimum several local minima must be searched and finally one with the smallest value have to be chosen. There are direct methods for global optimization over a specified domain (e.g. genetic algorithms) as well.

```matlab
function [x, i] = ternary(f, a, b, tol)
i = 1;
x1 = a + 1/3*(b-a);
x2 = b - 1/3*(b-a);
```
while abs(x2-x1) > tol
    if f(x1) < f(x2)
        b = x2;
    else
        a = x1;
    end
    i = i+1;
    x1 = a + 1/3*(b-a);
    x2 = b - 1/3*(b-a);
end
x = (x1+x2)/2;
end

function [x, i] = golden(f, a, b, tol)
i = 1;
R = (sqrt(5)-1)/2;
x1 = b - R*(b-a);
x2 = a + R*(b-a);
f1 = f(x1); f2 = f(x2);

while abs(x2-x1)>tol
    if f1 < f2
        b = x2;
        x2 = x1; f2 = f1; % taken from previous iteration
        x1 = b - R*(b-a);
        f1 = f(x1); % this must be evaluated
    else
        a = x1;
        x1 = x2; f1 = f2; % taken from previous iteration
        x2 = a + R*(b-a);
        f2 = f(x2); % this must be evaluated
    end
    i = i+1;
    x = (x1+x2)/2;
end
end

function [x2, i] = newton(f, df, x0, delta, N)
x1 = x0;
x2 = x1 - f(x1)/df(x1); % first approximation
i = 1; % number of iterations
while abs(f(x2))>delta && i<=N
    x1 = x2;
    x2 = x1 - f(x1)/df(x1);
    i = i + 1;
end
end

function [x i X] = gradmulti(grad, hesse, x0, eps, nmax)
% Two-variable minimization by Newton's method
%
% Input:
%   grad - function to compute gradient vector
%   hesse - function to compute Hessian
%   x0 - vector of initial values
%   eps - stopping criterion, step size
%   nmax - stopping criterion, maximum number of iterations
%
% Output:
%   x - solution vector
%   i - number of iterations
%   X - matrix of iteration steps

x1 = x0 - pinv(esse(x0))*grad(x0);
i=1;
X=[x0 x1];

% Stopping criteria:
% 1. step size is smaller than 'eps', or
% 2. maximum number of iterations, 'nmax' is reached
while and(norm(x1 - x0) > eps, i < nmax)
    x0 = x1;
    x1 = x0 - pinv(esse(x0))*grad(x0);
    i = i + 1;
    X = [X x1];
end
x = x1;
end