Solving non-linear equations

In hydraulics the design of a channel is a common task. Based on the the shape, material, steepness, width and the height of water in an open channel the water flow could be determined.

![Diagram of channel](image)

Let's have a look on the relevant formula:

$$Q = \frac{\sqrt{S}}{n} \cdot \frac{(b \cdot h)^3}{(b + 2n \cdot h)^{3/2}}$$

where $Q$ - water flow, $n$ - Gauckler-Manning coefficient, $S$ - steepness, $b$ - the width of the channel, $h$ - height of the water. This is an exact formula for $Q$, but if we are interested in the height of water to grant a specific amount of water flow, we can't express it explicitly. What level of water flow belongs to the water height of 1 and 2 meters? Determine the water height ($h$), if: $S = 0.0008; n = 0.02; Q = 3 \text{ m}^3 / \text{s}; b = 2 \text{ m}$

To solve a non-linear equation, we need to change the form a little bit according to the followings:

$$f(x) = 0$$

In this case this means: $f(x) = 0$. The solution of this equation is called the root, or zero point. If we substitute back $x$ into the expression, the equation will be true, or at least the value will be close to zero. If you visualize it graphically, the solution is the point, where the function crosses the horizontal axis:

![Graphs showing different number of solutions](image)

The solution of the equation $f(x) = 0$ could be find mostly only numerically, using iterations. This means we accept the result if a certain bounds on error ($\Delta$) is reached, $f(x) \leq \Delta$. The solutions are mostly local methods, which means to start the process you will need one or more initial guesses. In the next lessons you will here more about the different methods, because there are many different algorithms. A common property of these will be, that you can determine one solution at a time. So if you want to find more solutions, you need to apply the same method several times, with different initial
Design of an open flow channel

Design of an open flow channel

% Design of an open flow channel
clear all; clc; close all;
% Initializing the input variables
S = 0.008; n = 0.02; Q = 3; b = 2;
% Water flow defined as an anonymous function: h is the argument
Q=@(h) sqrt(S/n*(b*h).^(5/3))/(b+2*h).^(2/3);
% What levele of water flow belongs to the water height of 1 and 2 meters?
Q(1), Q(2)

ans = 1.7818
ans = 4.3170

We're searching for the location where this expression is equal to the given Q value, therefore 3. This means the solution will be between 1 and 2 meters. Let's visualize the current known state:

% Plotting the function in the range of [0;2]
figure(1); hold on;
fpplot(Q, [0 2]);
% plotting Q=3 in the range of [0;2]
h = fpplot('3', [0 2]); % plotting the line y=3

Warning: Char input to fpplot will be removed in a future release. Use fpplot(@t*ones(size(t))) instead.

set(h, 'Color', 'r'); % coloring line y=3 in red
% Labeling the diagram
title('Design of an open flow channel');
xlabel('Water height [m]');
ylabel('Water flow [m^3/s]');

Based on the figure it could be stated, that we have a solution somewhere in the range of 1.4 and 1.6. To solve this problem, you'll need an initial guess, or initial interval, that contains the solution. To find the solution of non-linear equations you can use for example the built-in function fzero in MATLAB. To use the function fzero some changes must be done on the previous expression, you need to transform the equation Q(h)=3 into the form of f(h)=0:

f = @(h) Q(h)-3

f = function_handle with value:
   @(h)Q(h)-3

Let's define an initial guess, and solve the example using function fzero:

x0 = 1.4

x0 = 1.4000

x = fzero(f, x0)

x = 1.4929

Q(x)-3 % check the result

ans = 4.4409e-16

plot(x, Q(x), 'k*') % plotting the result on the same figure

Closed interval methods

We can apply closed interval methods only, if we can determine a range [a,b], where on the two end the function values \(f(a) \text{ et } f(b)\) has different signs, therefore \(f(a) \cdot f(b) < 0\). In this case there is for sure a true solution in the interval (c), if \(f(x)\) is continous. Closed interval methods are always converging, but usually slower than the open interval methods. Using
this method we decrease the range of the interval in a way till we reach the desired $f(x)=0$ result (with a reasonable bounds on errors).

Restrictions for the initial interval:

1. There should be at least (and preferably) one solution in the given range
2. The function should be interpretable at the ends of the interval
3. The sign of the function should differ on the ends of the interval

**Bisection method**

By halving the initial interval, we check the function values at the ends of each section. This way we can determine with which half of the interval should we deal later, and use that part in the next step. We repeat this process, till we reach the desired approximation (with a pre-defined ∆ error).

![Bisection method diagram]

1. $c = (a + b)/2$
2. if $|f(c)| < ∆$ → stop the process
3. if $f(a) \cdot f(c) < 0$, then $b = c$, otherwise: $a = c$

**Regula falsi method**

This method is more effective, than the bisection method (it usually converges faster to the solution). It calculates where the line between the end points $((a, f(a)), (b, f(b)))$ of the interval crosses the horizontal axis and that location will be used as a new initial guess. The intersection of the line and the horizontal axis could be calculated using the principles of similar triangles:

\[
\frac{b-a}{f(b)-f(a)} = \frac{c-a}{0-f(a)}
\]

![Regula falsi method diagram]
1. the result: $x = c$: 
   
   $$ c = \frac{a \cdot f(b) - b \cdot f(a)}{f(b) - f(a)} $$

2. if $|f(c)| < \Delta \rightarrow$ stop the process
3. if $f(a) \cdot f(c) < 0$, then $b = c$, otherwise $a = c$

**Brent method (inverse quadratic interpolation)**

It is also called as Brent-Dekker method, because it's an improved version of the Brent method by Dekker. It's an efficient robust method, that combines the bisection method, the regula falsi method and the inverse quadratic interpolation. This is used by the built-in fzero function too.

The inverse quadratic interpolation needs 3 points in the $[a, b]$ interval. In the following expression the coordinates of the points is given in a reverse order ($f(x_i), x_i$), therefor the independent variable is the function value. For this three points, we fit a second order polynomial:

$$ x(f) = a_2 \cdot f^2 + a_1 \cdot f + a_0 $$

For this we determine the coefficients $a_2, a_1, a_0$, then substitute the $f=0$ value, to get the next point for our calculations ($x=c$ location), where the function crosses the horizontal axis. In case of $f=0$: $x = x(0) = a_0$

![Image](image.png)

The built-in function fzero, is a well-optimized version of the previous methods. It starts the process at first to determine an interval around the initial guess, where the function values on the ends have different signs, and then it solves the problem with a closed interval method using the Brent-Dekker method. We can also set several options with fzero-via optimset. E.g. to display the iterations ('Display'), or the accuracy of the calculations ('TolFun', or 'TolX').

```matlab
%% Solving the example with closed interval methods
[xbis, ibis] = bisection(f, 1.4, 1.6, 1e-9, 100)  % bisection method

xbis = 1.4929
ibis = 28

[xreg, ireg] = regulafalsi(f, 1.4, 1.6, 1e-9, 100)  % regula falsi method

xreg = 1.4929
ireg = 4

% built-in function: fzero
x = fzero(f, [1.4 1.6], optimset('Display', 'iter', 'TolFun', 1e-9))
```

<table>
<thead>
<tr>
<th>Func-count</th>
<th>x</th>
<th>f(x)</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.4</td>
<td>-0.233494</td>
<td>initial</td>
</tr>
<tr>
<td>3</td>
<td>1.49242</td>
<td>-0.00119159</td>
<td>interpolation</td>
</tr>
<tr>
<td>4</td>
<td>1.49289</td>
<td>2.92975e-07</td>
<td>interpolation</td>
</tr>
<tr>
<td>5</td>
<td>1.49289</td>
<td>-6.47749e-12</td>
<td>interpolation</td>
</tr>
<tr>
<td>6</td>
<td>1.49289</td>
<td>4.44889e-16</td>
<td>interpolation</td>
</tr>
<tr>
<td>7</td>
<td>1.49289</td>
<td>4.44889e-16</td>
<td>interpolation</td>
</tr>
</tbody>
</table>

Zero found in the interval [1.4, 1.6]

x = 1.4929
% if we want to store the iteration number:
[X,FV,EX,OUTPUT] = fzero(f,[1.4, 1.6], optimset('TolFun',1e-9))

X = 1.4929
FV = 4.4409e-16
EX = 1
OUTPUT = struct with fields:
    iterations: 5
    funcCount: 7
    algorithm: 'bisection, interpolation'
    message: 'Zero found in the interval [1.4, 1.6]'

i = OUTPUT.iterations

i = 5

% Checking the result, by substitution
f(xbisl), f(xreg), f(x) 

ans = -4.5629e-10
ans = -1.3299e-10
ans = 4.4409e-16

% Visualizing the result graphically
plot(x, f(x), 'ro');

% Displaying the result as text:
fprintf('Bisection method\n')

Bisection method

fprintf('Root location: %.4f, Value: %e, number of iterations:%d\n', xbisl, f(xbisl), ibisl)

Root location: 1.4929, Value: -4.562932e-10, number of iterations:28

fprintf('Regula falsi method\n')

Regula falsi method
Open interval method

For the open interval methods we only need one initial guess around the true solution. These methods usually converge faster to the solution than the closed interval methods (if they converge), but there are more stringent convergence criteria. There are multiple type of open interval methods, e.g. the gradient based methods (Newton and secant method), the fix-point methods which do not need gradients. You can find description about the fix-point methods (and also about an improved fix-point method, the Wegstein method) at the end of this chapter, they use instead of the form $f(x) = 0$ the $g(x) = x$ form of the examined function.

Newton's method

The Newton's method (or Newton-Raphson method) could be applied only if the function is continuous and differentiable and if we know, that around the initial guess a solution exists. In the first step we calculate the function value at the initial guess, than we search for the intersection of the horizontal axis and the local tangent at point $(x_0, f(x_0))$. This will be the next approximation of the solution. We repeat this process, till we reach the desired approximation (with a pre-defined $\Delta$ error), or we reach a maximum number of iteration. This last restriction is necessary, to avoid entering an infinite loop if it doesn't converges to anywhere.

$f'(x)$ the slope of the local tangent based on the figure: $f'(x_i) = \frac{f(x_i) - 0}{x_i - x_{i+1}}$.

From this the following iteration formula (the general form of the Newton's method) could be derived: $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$.

The Newton's method could be derived from the functions Taylor-series based approximation too, if we linearize it only including the first two terms: $f(x) \approx f(x_i) + f'(x_i)(x_{i+1} - x_i) = 0$, where $f(x) = 0$.

If the Newton's method could be applied, it is usually converging fast to the solution. The disadvantage is, that to apply this method, the functions derivative is necessary. This could be sometimes too complicated, for those the secant method is a better choice, which is an approximation of the Newton's method.

Secant method

The secant method is basically the Newton's method, where we approximate local derivative via finite difference.
approximation. In this case we don’t have to know the derivative of the function. Usually this converges slower and it needs two initial points \((x_0\text{ and } x_1\text{ in the Figure})\), but unlike the regula falsi method, these do not necessary have to interfere with the solution. The method is the same, just substitute into the formula of the Newton’s method the finite difference approximation of the derivative using the two initial point in the first iteration!

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

from this the general formula of the secant method could be derived:

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

Solving the example via Newton’s method:

Determine the derivative of function \(f\) by symbolically! For this the function \texttt{diff} could be used, after creating a symbolical expression from the function with the function \texttt{sym}.

\[
s = \text{diff(sym(f), 'h')}
\]

\[
s = \frac{10 \sqrt{2}}{3} \frac{(2h)^{2/3}}{(2h+2)^{2/3}} - 4 \sqrt{2} \frac{(2h)^{5/3}}{(2h+2)^{5/3}}
\]

The result of this is not a function, instead a symbolical variable. Lets define it as a function, which could be used in later calculations! You can simply copy and paste the expression after an anonymous function definition:

\[
df = @(h)(10*2^(1/2)*(2*h)^(2/3))/(3*(2*h + 2)^(2/3)) - (4*2^(1/2)*(2*h)^(5/3))/(3*(2*h + 2)^(5/3))
\]

\[
df = \text{function_handle with value:}
\]

\[
@(h)(10*2^(1/2)*(2*h)^(2/3))/(3*(2*h+2)^(2/3)) - (4*2^(1/2)*(2*h)^(5/3))/(3*(2*h+2)^(5/3))
\]

Remark: This task could be also done without the manual copy/paste work:

1. You could create a joined string array from the expressions you want to process in which you convert the symbolical result itno characters (\texttt{char}), then with the function \texttt{eval} you can evaluate the created line:

\[
\text{eval}(['df'=='@(h)' char(s)]) \% alternative for the previous operation
\]

\[
df = \text{function_handle with value:}
\]

\[
@(h)(10*2^(1/2)*(2*h)^(2/3))/(3*(2*h+2)^(2/3)) - (4*2^(1/2)*(2*h)^(5/3))/(3*(2*h+2)^(5/3))
\]

2. You could use the built-in function: \texttt{matlabFunction}

\[
\text{matlabFunction(s)}
\]

\[
\text{ans} = \text{function_handle with value:}
\]

\[
@(h)sqrt(2.0).*1.0./((h.*2.0+2.0).^2.0./3.0).*((h.*2.0).^2.0./3.0).*((1.0e1./3.0)-sqrt(2.0).*1.0./((h.*2.0+2.0).^5
\]
Let's use now 10 degrees as an initial guess:

```matlab
>> x0 = 10;
>> [xnew, inew] = newton(f, df, x0, 1e-9, 100)
xnew = 1.4929
inew = 4
```

```matlab
fprintf('Newton's method\n')
```

```matlab
Root location: 1.4929, Value: -4.440892e-16, number of iterations:4
```

Result:

Newton's method

Root location: 6.0636, Value: 1.421085e-14, number of iterations:4

The solution is the same as earlier, but the interesting thing is, that the method converges to the solution faster than any method before, even though we choose a worse initial guess on purpose. The disadvantage is, that to apply this method, we need to calculate the derivative of the function.

### Finding the roots of a single variate polynomial

It is a common that the non-linear equation of which we want to find the solution is an algebraic polynomial (an expression consisting variables and coefficients that involves only operations of addition, subtraction, multiplication, and non-negative integer exponents of the variables). To find the roots of these, there are multiple built-in functions in MATLAB, which can find all the roots without any initial guess. One of them is the function `roots` which finds the roots of the single variate polynomial numerically using only the coefficients of the polynomial (in a vector, in decreasing order starting from the highest order term). E.g.: in case of the polynomial $3x^3 - 4x^2 - 23 = 0$ the coefficient vector is: [3, -4, 0, -23]. Another function is the `solve` that solves this task symbolically and results in exact values. Let's look at an example from the topic of elasticity, where we need to find the roots of a polynomial.

### Homogeneous system of linear equations

When right side of the system is zero ($b=0$) it is called a homogeneous system of linear equations. In this case there is a nontrivial solution $x \neq 0$ only when the determinant of matrix $A$ is zero: $\text{det}(A)=0$.

Let us consider an example from engineering (mechanics), the Cauchy stress tensor $F$ (see figure).

![Cauchy stress tensor](image)

Determination of principal stresses and axes of principal stresses is an eigenvalue-eigenvector problem which is the following homogeneous system

$$(F - \sigma_e \cdot I) \cdot \epsilon = 0$$
where $\sigma_i$ is a principal stress, $I$ is the identity matrix and $e$ is unit vector along the principal stress axis. We seek a nontrivial solution $e \neq 0$. Eigenvalues are solutions of the characteristic equation $\det(F - \sigma_i \cdot I) = 0$.

Consider an example in Matlab. Elements of stress tensor $F$ at point P of the solid body are: $\sigma_x = 50$ MPa, $\sigma_y = 80$ MPa, $\sigma_z = -20$ MPa, $\tau_{xy} = \tau_{xy} = 20$ MPa, $\tau_{xz} = \tau_{zx} = -40$ MPa, $\tau_{yz} = \tau_{zy} = -30$ MPa. Let us determine both principal stresses and axes.

We can write characteristic equation symbolically by expanding the determinant

```matlab
syms de
F = [50 20 -40; 20 80 -30; -40 -30 -20]

F =
50  20 -40
20  80 -30
-40 -30 -20
eq = det(F-eye(3)*de)
eq = -de^3 + 110 de^2 + 1500 de - 197000

Roots of the equation are:

c=sym2poly(eq)
c =
-1  110  1500 -197000
de = roots(c)
de =
186.7674
-41.3691
44.6017

sort(de, 'ascend')

ans =
-41.3691
44.6017
186.7674

shortcut in Matlab to obtain eigenvalues by the eig function:

eig(F)

ans =
-41.3691
44.6017
186.7674

Check whether matrix is really singular:

A1 = F-eye(3)*de(1)

A1 =
-56.7674  20.0000 -40.0000
 20.0000 -26.7674 -30.0000
-40.0000 -30.0000 -126.7674

% Singular? YES
det(A1)
ans = 7.9048e-11

3-rank(A1) % Number of basis vectors: 1

ans = 1

The eigenvector is a basis vector of the null space of matrix A1

v1=null(A1)

v1 =
    -0.5203
    -0.7795
     0.3487

Check our solution by substituting into the original homogeneous system of equations

(F-de(1)*eye(3))*v1

ans =
   1.0e-13 *
    -0.2132
    -0.0577
     -0.2018

Matlab built-in function eig with two outputs gives all eigenvalues and eigenvectors as columns of V:

[V D]=eig(F)

V =
   0.3647    0.7722    0.5203
   0.1664   -0.6038    0.7795
   0.9162   -0.1977   -0.3487

D =
  -41.3691     0     0
   0   44.6017     0
   0     0  106.7674

function [x2, i] = newton(f, df, x0, delta, N)
% Newton method
% Input parameters:
%  f - single variate function
%  df - the derivative
%  x0 - initial guess
%  delta - terminating condition, threshold of the error
%  N - terminating condition, maximal iteration number
% Outputs:
%  x2 - location of the zero point - the solution
%  i - number of iterations
    x1 = x0;
    x2 = x1 - f(x1)/df(x1);
    i = 1;
    while abs(f(x2))>delta && i<=N
        x1 = x2;
        x2 = x1 - f(x1)/df(x1);
        i = i + 1;
    end
end
function [c, i] = regulafalsi(f, a, b, delta, N)
% Regula falsi method

c = (a*f(b) - b*f(a))/(f(b) - f(a)); % 1st iteration
i = 1; % number of iteration

% Stopping Criteria:
% error is smaller then the given tolerance, or the maximum iteration number is reached
while abs(f(c)) > delta && i <= N
    if f(c)*f(a) < 0
        b = c;
    else
        a = c;
    end;
    i = i + 1;
    c = (a*f(b) - b*f(a))/(f(b) - f(a));
end; % end

function [c, i] = bisection(f, a, b, delta, N)
% Bisection method

c = (a+b)/2; % 1st iteration
i = 1; % number of iteration

% Stopping Criteria:
% error is smaller then the given tolerance, or the maximum iteration number is reached
while abs(f(c)) > delta && i <= N
    if f(c)*f(a) < 0
        b = c;
    else
        a = c;
    end;
    i = i + 1;
    c = (a+b)/2;
end; % end