4. ROOTS OF NONLINEAR EQUATIONS

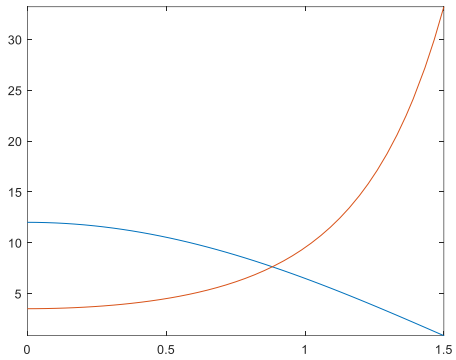
The numerical solution of the equation $f(x) = 0$ (finding its roots/zeros) often arises as a task to be solved in connection with engineering works.

$$f(x) = 0$$

The solution of this equation is also called the root of the equation. It is possible that our basic problem differs from the form $f(x) = 0$, but in many cases we can trace the solution of the problem back to this form by eliminating the right side. Let's look at some examples!

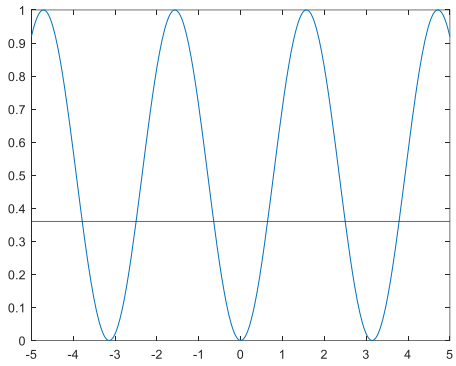
Finding the intersection of two univariate functions can also be traced back to finding the roots of a non-linear equation, after rearrangement!

$$3.5 \cdot e^{x^2} = 12\cos(x) \quad \rightarrow \quad 3.5 \cdot e^{x^2} - 12\cos(x) = 0$$

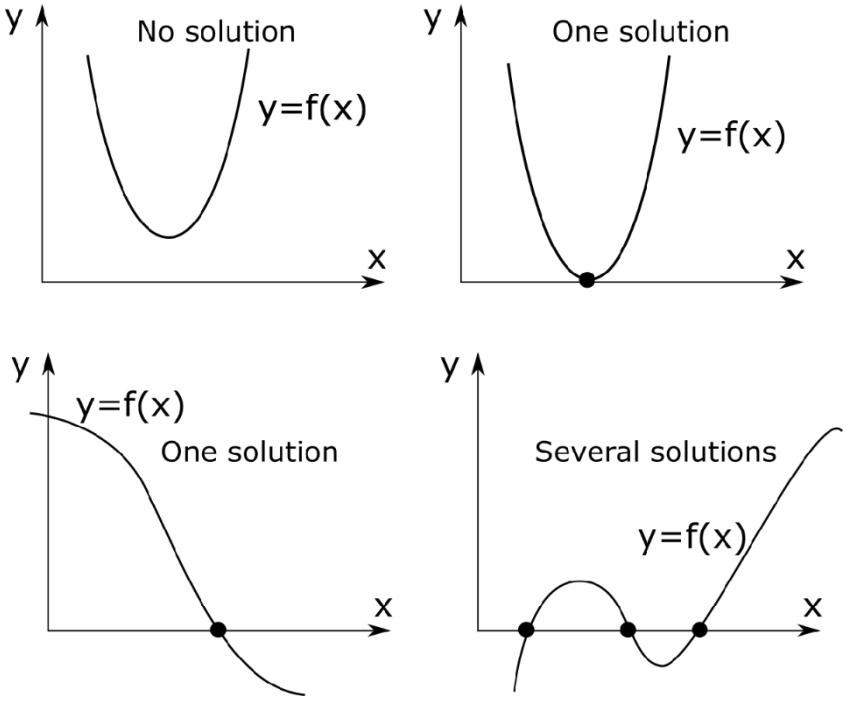


It can be solved similarly if we look for where a function takes a certain value:

$$\sin^2(x) = 0.36 \quad \rightarrow \quad \sin^2(x) - 0.36 = 0$$



The solution *x* satisfies the equation when substituted back, the value of the equation will be zero or approximately zero (within a given tolerance, see the round-off errors learned earlier). Graphically, the solution is the point where the function intersects the x-axis. Depending on the shape of the function, several cases are possible: there is no solution, there is 1 solution or there are several solutions.

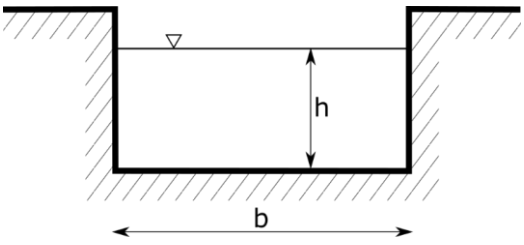


In many cases, the solution of the equation $f(x) = 0$ is only possible numerically, with iterations. In this case, during each iteration, we get closer and closer to the solution, until we are below a given error limit Δ (tolerance): $f(x) \leq \Delta$. In such cases, instead of the equation $f(x) = 0$, we actually solve the equation $f(x) \leq \Delta$, where Δ is the tolerance, which is mostly a small value.

Most of the solution methods are so-called local methods, when the iteration requires one or more initial values. Various algorithms have been developed for the solution. The characteristic of the algorithms is that they allow the determination of only one root at a time. In the case of several roots, they must be run with several initial values. The two main groups of root-finding algorithms are closed and open interval methods.

CIVIL ENGINEERING EXAMPLE OF SOLUTION OF A NONLINEAR EQUATION

In hydraulics the design of a channel is a common task. Depending on the shape, material, slope, width and water level of the open-surface channel, the amount of transported water can be deduced. For example, let's look at the formula for the water flow rate of a channel with an open surface and a rectangular cross-section!



$$Q = \frac{\sqrt{S}}{n} \cdot \frac{(b \cdot h)^{\frac{5}{3}}}{(b + 2 \cdot h)^{\frac{2}{3}}}$$

where Q – water flow rate (discharge), n - Manning's roughness coefficient, S - slope, b – channel width, h – water depth in the channel.

We would like to get answers to two questions:

1) How much discharge corresponds to a water depth of 1 and 2 meters?
2) Determine the height of the water in the channel, *h*, in the case of a standard water flow rate of 3 m³/s ($Q = 3\ {}^{m^3}/_{s}$)!

We know that

- The slope is 0.8 ‰ ($S = 0.0008$),
- 0.02 is the Manning roughness coefficient ($n = 0.02$) and
- 2 m is the channel width ($b = 2\ m$).

The first part of the task, how much discharge (water flow rate) belongs to a water depth of 1 and 2 meters, is easy to answer, it just means a simple substitution. First, enter the values of the variables and define the discharge (Q) as a function of the water depth (h).

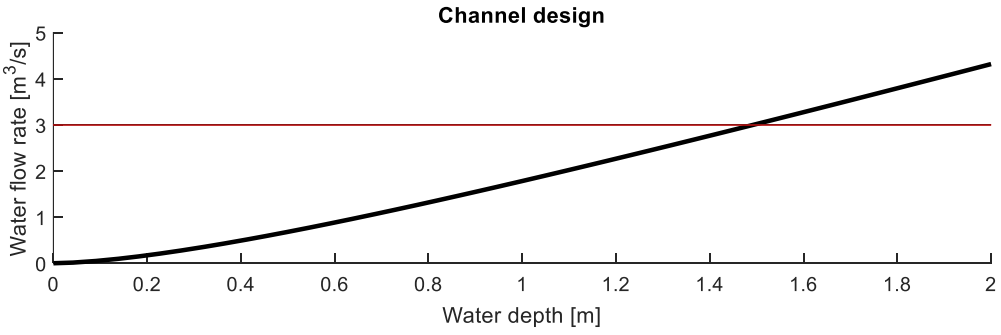
```
> clear all; clc;close all;
> % Assigning values to variables
> S = 0.0008; n = 0.02; Q = 3; b = 2;
> % Discharge as an inline function: h independent variable
> Q=@(h) sqrt(S)/n*(b*h).^(5/3)./(b+2*h).^(2/3)
```

How much discharge corresponds to a water depth of 1 and 2 m?

```
> Q(1), Q(2) %  1.7818 and 4.3170 m^3/s
```

The second question was what the water depth would be for a water flow rate of *Q*=3 m^3/s. Since we cannot express *h* from the equation as a function of *Q*, only an approximate, numerical solution can be considered here. Based on the two substituted values, 1 m has a water flow rate of 1.78 m^3/s, and 2 m has a water flow rate of 4.31 m^3/s, so the required water depth will be somewhere between 1 and 2 meters for the standard discharge of 3 m^3/s. Let's plot the function on the range 0-2 m and also draw the desired value Q=3 m^3/s! Functions can be displayed with the **fplot** command, and related pairs of points with the **plot** command.

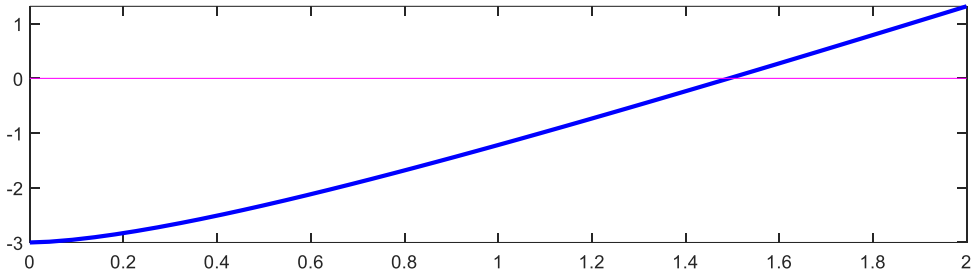
```
> % Representation of the function on the interval [0;2].
> figure(1); hold on;
> fplot(Q, [0 2],'Color','k','LineWidth',2);
> % Draw the line y=3 on the interval [0 2] specifying two points
> line([0,2],[3,3],'Color','r')
> % of course, the plot command can also be used for this
> plot([0,2],[3,3],'r')
> % or we can also use the xlim (figure x-direction limits) command
> plot(xlim,[3,3],'r')
> % Starting from version R2018b of Matlab, the yline command can also
  be used for the same
> yline(3)
> % labels and title
> title('Channel design');
> xlabel('Water depth [m]');
> ylabel('Water flow rate [m^3/s]');
```



The figure shows that the solution, Q=3, will be somewhere between 1.4 and 1.6. Algorithms that can be used to find the solution, on the other hand, always look for the zero point/root, that is, where the function intersects the x-axis, not where it takes a specific y value. Therefore, we must rearrange the equation in this form!

$$Q(h) = 3 \quad \rightarrow \quad f(h) = Q(h) - 3 = 0$$

```
> % Rearrange the equation
> f = @(h) Q(h)-3
> figure(2);
> fplot(f, [0 2],'Color','b','LineWidth',2);
> % Draw the line y=0 on the interval [0 2] specifying two points
> hold on; line([0,2],[0,0],'Color','m')
```



The solution, root or zero point can be determined using several methods, the two main types of which are the closed and open interval methods.

CLOSED INTERVAL METHODS

In the case of closed interval methods, we specify an interval [a,b] that contains the solution (and only one solution!). In this case, the function values will have opposite signs at the two endpoints of the interval, therefore $f(a) \cdot f(b) < 0$. Since the function changes sign between the two endpoints of the interval, it passes through zero. In this case, a zero point (*c*) can certainly be found within the interval, if *f(x)* is continuous. In this case, we gradually narrow the size of the interval, examining the function values of the end points, until the value of the function is either very close to 0 (within the specified tolerance), or the interval itself becomes very small. There are several types of closed interval methods, the difference between them is the method used to narrow the size of the interval. Closed interval methods always lead to results, only some methods are slower and others are faster.

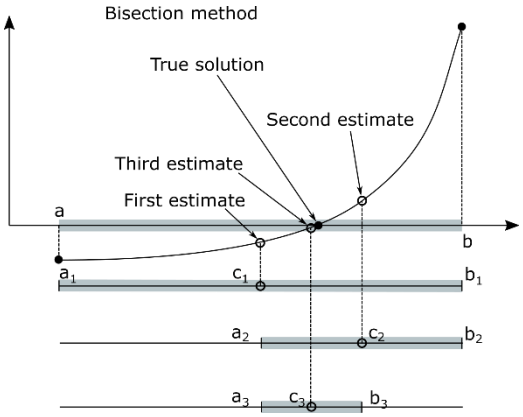
There are a few conditions that must be met in order to choose a good starting value:

1) There must be at least one (if chosen correctly, exactly one) solution in the interval.
2) The function must be continuous in the interval and interpreted at the endpoints.
3) The sign of the function values of the endpoints must be opposite.

BISECTION METHOD

By halving the initial interval, we examine the function values at the end points. Where the sign is different, that will be the new interval to deal with, and this is repeated until the desired Δ accuracy is reached.

1. $c = (a + b)/2$
2. if $|f(c)| < \Delta \rightarrow$ end
3. if $f(a) \cdot f(c) < 0$, then $b = c$, else $a = c$



REGULA FALSI METHOD

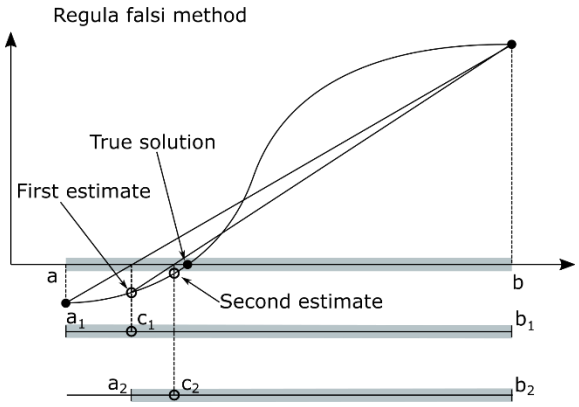
The regula falsi method is a more efficient solution than the bisection method (it usually converges faster). Here, the intersection point of the line connecting the points (*a,f(a)*) and (*b,f(b)*) with the x axis is determined as a new approximate value. The intersection of the line with the x axis (y=0) can be calculated from similar triangles:

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

1. The solution x=c:

$$c = \frac{a \cdot f(b) - b \cdot f(a)}{f(b) - f(a)}$$

2. if $|f(c)| < \Delta \rightarrow$ end
3. if $f(a) \cdot f(c) < 0$, then $b = c$, else $a = c$



BISECTION AND REGULA FALSI METHOD IN MATLAB

Let's see how we can write our own Matlab/Octave function for the bisection method! It will be necessary to specify a tolerance Δ (expected accuracy) and a maximum number of iterations to stop the condition controlled loop (**while** loop). Here it will be necessary to use a logical AND to test the two conditions together, this can be specified in Matlab in several ways: condition1 **&&** condition2 or **and**(condition1, condition2).

```
> 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
```

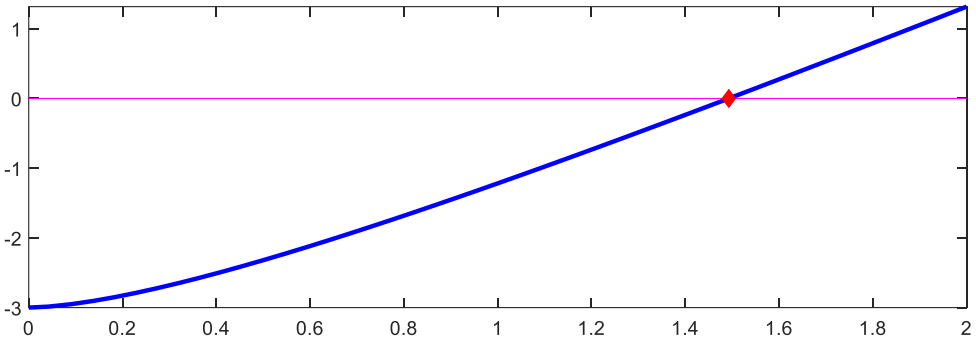
The regula falsi method can be implemented in the same way, only the calculation of c is different in the first and subsequent iterations, instead of `c=(a+b)/2`:

```
>   c = (a*f(b) - b*f(a))/(f(b) - f(a));
```

EXAMPLE FOR APPLICATION OF CLOSED INTERVAL METHODS

Let's use the functions we wrote (**bisection.m** and **regulafalsi.m**) to solve the channel design task. As a check of the solution, we should calculate the value of the function at the location of the solution and plot it in the figure as well. To be able to use our own functions (**bisection.m**, **regulafalsi.m**), they must be in the same working directory as our script!

```
>   [xbi, ibi]= bisection(f, 1.4, 1.6, 1e-9, 100) % bisection
>   % xbi = 1.4929, ibi = 28
>   [xreg, ireg]= regulafalsi(f, 1.4, 1.6, 1e-9, 100) % regulafalsi
>   % xreg = 1.4929, ireg = 4
>   % Check
>   f(xbi), f(xreg) %   -4.5629e-10,  -1.3299e-10
>   % Plot
>   plot(xreg, f(xreg), 'rd','MarkerFaceColor','r');
```



Let's evaluate the results! Are the solutions the same? Which was the faster algorithm? Which gave the more accurate result?

OPEN INTERVAL METHODS

In the case of open interval methods, instead of a closed interval, we only need to enter one initial guess (x_0) to search for the root. The convergence of these methods are usually much faster than that of the closed interval methods, if they converge! They require stricter convergence conditions and do not always reach the result. There are several open interval methods, for example the gradient-type Newton and secant methods. Another solution is the fixed-point method, which does not require a gradient, and its further developed form, the Wegstein method (transforming the equation f(x)=0 into the form g(x)=x).

NEWTON'S METHOD

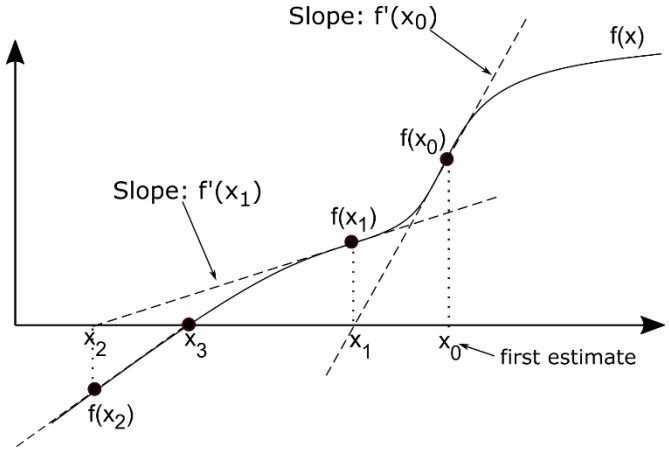
The Newton method (or Newton-Raphson method) can be applied if the function is continuous and differentiable and we know that the problem has a solution near a given initial value. First the method calculates the value of the function ($f(x_0)$) and its derivative ($f'(x_0)$) at the initial point x_0, and determines the point of intersection of the tangent line at this point with the x axis. This will be the next approximate value of the solution, x_1. The method continues until the value of $f(x)$ is smaller than a given tolerance Δ or the given maximum number of iterations is reached.

The slope of the tangent ($f'(x)$) can be calculated based on the figure with the following formula:

$$f'(x_i) = \frac{f(x_i) - 0}{x_i - x_{i+1}}$$

The iteration formula of Newton's method can be derived from this:

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



The Newton method can also be derived from the Taylor series approximation of the function, taking into account the first two terms (from the linearization of the function):

$f(x) \approx f(x_i) + f'(x_i) \cdot (x_{i+1} - x_i) = 0$, where $f(x) = 0$.

Newton's method, if it converges to a solution, usually converges quickly. However, we see that the derivative of the function must also be known in order to apply the method. This runs into difficulties in some cases, e.g. it can be too complicated to calculate the derivative. In this case, we can use the secant method, which is an approximation of the Newton method.

SECANT METHOD[1]

The secant method is a finite difference approximation of the Newton method. It can be used if we do not know the derivative of the function (or it would be difficult to calculate it). It usually converges more slowly (see the same example in the two figures) and at the beginning requires two initial guesses x_0 and x_1 in the figure (but unlike the closed interval methods, the solution does not have to lie between the two points).

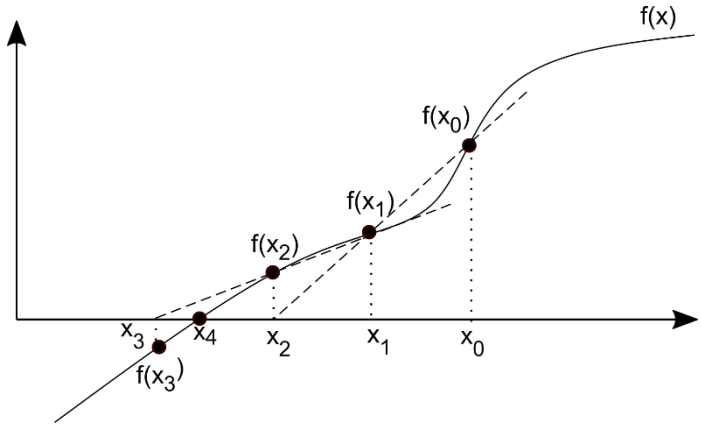
[1] Supplementary material

Let's use the finite difference approximation of the derivative:

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

Substituting back this to the Newton's method, the iteration formula of the secant method can be derived:

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



NEWTON'S METHOD IN MATLAB

For Newton's method, it is also necessary to know the derivative of the function (*df*), if this is available, we can solve the problem with the following function (**newton.m**):

```
function [x2, i] = newton(f, df, x0, delta, N)
% Newton method
% Input parameters:
%   f - single variate function
%   df - the derivative
%   x0 - initial guess
%   delta - tolerance of the error, stopping criteria
%   N - maximal iteration number, stopping criteria
%
% 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
```

EXAMPLE OF APPLICATION OF NEWTON'S METHOD

Let's solve the channel design problem with Newton's method! Determine the derivative of the function *f* with respect to *h* symbolically! To do this, we can use the **diff** command after converting the function *f* into symbolic expression with the **sym** command. or we can define *h*, as a symbolic variable with the **syms** command, then using this in an equation, the equation will also be a symbolic expression. Symbolic Toolbox must be installed for this task!

```
% Symbolic derivation
s=diff(sym(f),'h')
% s = (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))
```

```
> % Or defining first h, as a symbolic variable
> syms h % defining  h, as a symbolic variable
> eqs =  f(h)  % this will also be a symbolic expression
> s = diff(eqs)
```

However, the result is not a function but a symbolic expression into which concrete values cannot be substituted. Let's define it as a function so that we can work with it later! One solution to this is to simply copy the result after the function definition, or you can also use the **matlabFunction** command, which generates a function from a symbolic expression.

```
> % function from symbolic expression: definition then CTRL+C,CTRL+V
> 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))
> % another solution: use matlabFunction
> df = matlabFunction(s)
```

Let's solve it with Newton's method too!

```
> % solution with Newton's method
> [xnew, inew]= newton(f, df, 2, 1e-9, 100)
> % xnew = 1.4929, inew = 3
```

The root is of course the same as before. We can see that even with a starting value of 2 meters which is far from the solution, the procedure converged faster than any closed interval method, only 3 iterations were sufficient. A disadvantage of the method is that the derivative of the function had to be determined as well.

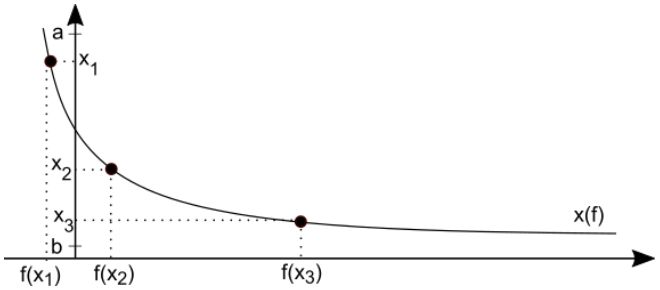
BUILT-IN MATLAB FUNCTION - FZERO

The previous algorithms presented the basics of numerical calculations very well. It matters a lot what algorithm we use and what starting value we choose. Depending on the above, we get a result faster or slower at the end of an iteration procedure (if there is a solution at all and the method converges). Finding the roots of a nonlinear equation is a basic task that comes up very often during our calculations. Of course, Matlab also has its own built-in function (**fzero**), which is a combination of several methods and is called the Brent-Dekker algorithm.

BRENT METHOD (INVERSE QUADRATIC INTERPOLATION)[2]

It is also called the Brent-Dekker method, since Dekker's earlier method was further developed by Brent. An efficient and robust method that combines the bisection method, regula falsi method and inverse quadratic interpolation. This is used by the built-in **fzero** function too.

For the inverse quadratic interpolation, we need to specify 3 points in the interval [a, b]. The coordinates of the points will be used in reversed order ($f(x_i), x_i$) so now the independent



[2] Supplementary material

variable will be the function value. These 3 points are used to fit a quadratic polynomial.

$$x(f) = \alpha_2 \cdot f^2 + \alpha_1 \cdot f + \alpha_0$$

When fitting the polynomial, the values of $\alpha_2, \alpha_1, \alpha_0$ can be determined, then, substituting back the value f=0, we immediately get the location x=c, where the function intersects the axis.

$$c = x(0) = \alpha_0$$

EXAMPLE TO USE THE FZERO FUNCTION

Let's solve the channel design task using the **fzero** command! The **fzero** command can be called with two types of initial values. We can use one initial value or specify an interval. In the case of two starting values, an interval must be chosen that contains the solution, so the function changes sign (closed interval method).

```
> % call with two initial values
> x = fzero(f,[1.4, 1.6]) % x = 1.4929
> % call with one initial value
> x = fzero(f,1.6) % x = 1.4929
```

If it is called with one initial value, the **fzero** function starts by searching for an interval around the initial value where the signs at the endpoints are different. After finding the interval, **fzero** will solve the problem using a closed interval method, the Brent-Dekker algorithm. The individual iteration steps can also be displayed using the **optimset** parameter. We can specify whether the iterations should be displayed ('Display','iter'), and what the calculation precision should be (using '**TolFun**' or '**TolX**', the tolerance of the function value or the independent variable).

```
> % call with one initial value, with additional options
> x = fzero(f,1.6, optimset('Display','iter','TolFun',1e-9))
> % Search for an interval around 1.6 containing a sign change:
> %  Func-count    a          f(a)             b          f(b)
   Procedure
> %     1            1.6      0.274131           1.6      0.274131
   initial interval
> %     3         1.55475     0.157999        1.64525     0.390694
   search
> %     5           1.536     0.110027          1.664     0.439097
   search
> %     7         1.50949     0.0423222       1.69051     0.507665
   search
> %     8           1.472    -0.0531432       1.69051     0.507665
   search
> %
> % Search for a zero in the interval [1.472, 1.69051]:
> %  Func-count    x          f(x)             Procedure
> %     8           1.472    -0.0531432        initial
> %     9         1.49271  -0.000458365        interpolation
> %    10         1.49289   4.30326e-08        interpolation
> %    11         1.49289   -3.6593e-13        interpolation
> %    12         1.49289   4.44089e-16        interpolation
> %    13         1.49289   4.44089e-16        interpolation
> %
> % Zero found in the interval [1.472, 1.69051]
> % x = 1.4929
```

INTERSECTION OF TWO UNIVARIATE FUNCTION

Finding the intersection of two univariate functions can also be traced back to finding the roots of a univariate function. Let's find the intersection point of the following two functions on the interval [-2,4]!

$$f(x) = (x-3)^3 + 20$$

$$g(x) = -5\,x + 6$$

At the point of intersection, the values of the two functions are equal to each other:

$$f(x) = g(x)$$

et's rearrange this to 0:

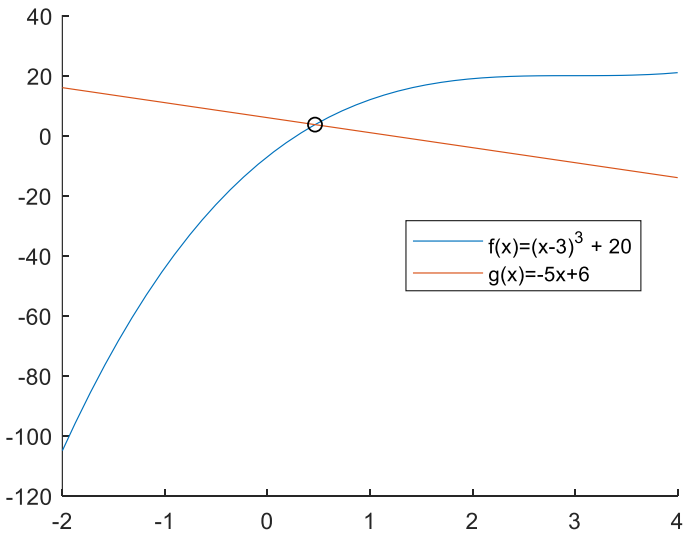
$$f(x) - g(x) = 0$$

Assign a new function to the 0-ordered form of the equation:

$$h(x) = f(x) - g(x)$$

The roots of the above function *h* can be determined using one of the previous methods.

```
> clc; clear all; close all;
> f = @(x) (x-3).^3 + 20
> g = @(x) -5*x+6
> figure(3); hold on;
> fplot(f,[-2,4]);
> fplot(g,[-2,4])
> h = @(x) f(x)-g(x) % rearrange the equation to 0
> x = fzero(h,0.5)  % 0.4636 - solution for x
> plot(x,f(x),'ko')
> legend('f(x)=(x-3)^3 + 20','g(x)=-5*x+6','Location','best')
```



ROOTS OF A UNIVARIATE ALGEBRAIC POLYNOMIAL

A common problem is that the non-linear equation whose roots we are looking for is an algebraic polynomial, i.e. it contains only integer powers of x.

The general form of an algebraic polynomial is:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

The coefficients $a_n, a_{n-1}, \cdots, a_1, a_0$ are real numbers, and *n* is a non-negative integer, the degree of the polynomial.

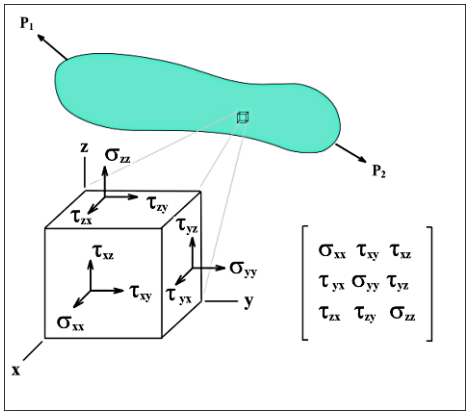
There are two commands in Matlab that can be used to determine all the roots of an algebraic polynomial at the same time without specifying initial values. One is the **roots** command, which numerically determines the roots of a univariate polynomial. The coefficients of the polynomial must be given in a vector from the term of the highest degree backwards. For example, the coefficients of the polynomial $3x^3 - 4x^2 - 23 = 0$ are [3, 4, 0, -23]. The other command is **solve**, it solves the task symbolically and provides exact values for the solution.

Let's look at an example of this from solid mechanics! Such an example is the determination of the roots of the characteristic polynomial in eigenvalue problems.

DETERMINATION OF PRINCIPAL STRESSES, SOLUTION OF AN EIGENVALUE PROBLEM

A common task in mechanics is to determine the principal stresses and principal stress axes from the Cauchy stress tensor! The tensor consists of nine components that completely define the state of stress at a point inside a material, see figure below[3]. Let F be the stress tensor.

$$F = \begin{pmatrix} \sigma_x & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \sigma_y & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \sigma_z \end{pmatrix},$$



Principal axes are the axes where only normal stresses occur, but no shear stresses (F_f).

$$F_{f(1,2,3)} = \begin{pmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{pmatrix}, \text{ where } \sigma_1 \geq \sigma_2 \geq \sigma_3.$$

From a mathematical point of view, the principal stress and principal axis problem can be considered an eigenvalue problem. The stress vector ρ_v can be written in the direction of the *v* principle unit vector by the product of the principle stress σ_f and the unit vector *v*: $\rho_v = \sigma_f \cdot v$, or by the projection of the stress tensor F in the *v* direction: $\rho_v = F \cdot v$. Equating the two (multiplying the first by identity matrix): $F \cdot v = \sigma_f \cdot I \cdot v$, the following formula can be derived: $(F - \sigma_f \cdot I) \cdot v = 0$

[3] figure: CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=591829

where I is the identity matrix. Let's solve the equation above:

$$\begin{pmatrix} \sigma_x - \sigma_f & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \sigma_y - \sigma_f & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \sigma_z - \sigma_f \end{pmatrix} \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

The above equation is a homogeneous system of linear equations (there are only zeros on the right side), where we are looking for a different solution than the trivial one (*v*=0). A non-trivial solution only exists if the determinant of the coefficient matrix is 0, i.e. $det(F - \sigma_v \cdot I) = 0$. The determined v vectors are the eigenvectors in the direction of the principal axes, and the σ_f values are the eigenvalues, the principal stresses. Let's expand the determinant, this will be the characteristic equation.

$$det(F - \sigma_f \cdot I) = \begin{vmatrix} (\sigma_x - \sigma_f) & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & (\sigma_y - \sigma_f) & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & (\sigma_z - \sigma_f) \end{vmatrix} = 0$$

Now let the stress tensor F be the following: $F = \begin{bmatrix} 50 & 20 & -40 \\ 20 & 80 & -30 \\ -40 & -30 & -20 \end{bmatrix} MPa$

Let's find the corresponding principal stresses, solve the equation a $det(F - \sigma_f \cdot I) = 0$! Express the determinant symbolically (**det** command)! This will be the characteristic equation, where *L* has denoted the principal stresses a σ_f, which are actually the eigenvalues:

```
> F = [50, 20, -40; 20, 80, -30; -40, -30, -20];
> syms L
> eq = det(F-eye(3)*L) % eq = - L^3 + 110*L^2 + 1500*L - 197000
```

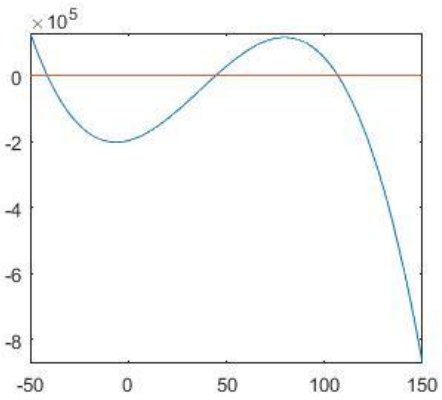
It is an algebraic polynomial of third degree, the roots of which give the eigenvalues:

$$eq = -{\sigma_f}^3 + 110\,{\sigma_f}^2 + 1500\,\sigma_f - 197000 = 0$$

Convert the symbolic expression back into a function and plot the function on the interval [-50,150].

```
> eq1 = matlabFunction(eq)
> %  @(L) L.*1.5e3+L.^2.*1.1e2-L.^3-1.97e5
> figure(3); fplot(eq1,[-50,150])
> % Plot the line y=0 on the interval [-50,150].
> hold on; plot([-50,150],[0,0])
```

We can find the roots of the third degree polynomial, e.g. with the **fzero** function. Based on the figure, there are 3 eigenvalues, so **fzero** must be called with 3 different starting values to find all the solutions. Initial values can be taken from the figure, where the function intersects the x-axis! Let these be x=120, 50, -40!




```
> % Solution with fzero
> L1 = fzero(eq1,120) % 106.7674
> L2 = fzero(eq1,50) %  44.6017
> L3 = fzero(eq1,-40) % -41.3691
```

The result are the three eigenvalues, the three principal stresses: $\sigma_1 = 106.7674, \sigma_2 = 44.6017,\ \sigma_3 = -41.36$. Finding the three roots required 3 function calls. However, there are special algorithms for polynomials that provide all solutions in one step and do not even need initial values. One of these is the **solve** command, which can be used with symbolic expressions. Let's use the **solve** command to solve the equation 'eq = - fo^3 + 110*fo^2 + 1500*fo – 197000'!

```
> sol1 = solve(eq)
> %  root(z^3 - 110*z^2 - 1500*z + 197000, z, 1)
> %  root(z^3 - 110*z^2 - 1500*z + 197000, z, 2)
> %  root(z^3 - 110*z^2 - 1500*z + 197000, z, 3)
> sol2 = real(double(sol1))
> %   -41.3691
> %    44.6017
> %   106.7674
```

As a result of the **solve** command we got symbolic expressions, not concrete numbers. We must convert them into numbers with the **double** command. However, the resulting numbers may contain numerically negligible complex parts that can be omitted with the **real** command. We can specify the whole process in one line and get all three solutions at once:

```
> sol = real(double(solve(eq)))
```

Another command that can be used for algebraic polynomials is **roots**, which solves the equation numerically. This also provides all the solutions in one step and there is no need to enter an initial value here either. Here, on the other hand, the coefficients of the polynomial must be collected in a vector starting from the term of the highest degree back to the constant term. Our equation is: $q = -{\sigma_e}^3 + 110\ {\sigma_e}^2 + 1500\ \sigma_e - 197000$. In such a simple case, we can also manually enter the vector of coefficients:

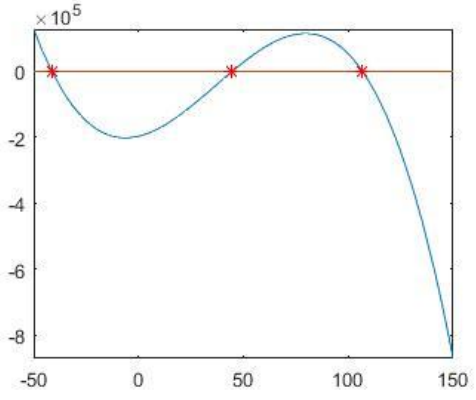
```
> % vector of coefficients
> c = [-1, 110, 1500, -197000]
```

Or we can use the **sym2poly** command, which extracts the coefficients from a symbolic polynomial:

```
> % other solution
> c = sym2poly(eq)
> % c = [-1  110 1500 -197000]
```

Let's solve it with the **roots** command and plot the results in the diagram!

```
> L = roots(c)
> %   106.7674
> %   -41.3691
> %    44.6017
> plot(L,eq1(L),'r*')
```



As we saw, there was no need to enter initial values here either, and we got all the roots at the same time. Of course, the directions of the principal axes could also be determined by substituting the principal stresses back into the original system of equations. However, we can also use the many built-in functions of Matlab, there is also a solution to the eigenvalue and eigenvector problem, as this is a very common task, the name of this command is: **eig**.

```
>  [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
```

Here we called the **eig** command with two outputs and got all the eigenvalues (in the diagonal of matrix D) and their corresponding eigenvectors (columns of matrix V) at the same time!

NEW FUNCTIONS USED IN THE CHAPTER

and(cond1, cond2), cond1 && cond2,	-	Logical AND
diff	-	Symbolic derivation
sym	-	Converting expressions into symbolic expressions
syms	-	Defining symbolic variables
fzero	-	Finding the roots of a univariate equation numerically
det	-	Determinant of a matrix
solve	-	Algebraic polynomial roots symbolically
roots	-	Algebraic polynomial roots numerically
double	-	Convert a number specified as a symbolic expression to a floating point number
real	-	Real part of a complex number
sym2poly	-	Extract vector of all numeric coefficients from symbolic polynomial
eig	-	Determination of matrix eigenvalues and eigenvectors