15. OPTIMIZATION

Optimization is the determination of the location of the extreme value of a function. This task often occurs in engineering practice, for example the location of the maximum displacement of a support structure, the position of a point with the smallest error when adjusting geodetic measurements, or the largest contamination when testing water quality.

During optimization, there is a target (or objective) function $f(\boldsymbol{x})$,where $\boldsymbol{x}$ is the vector of the independent variables: $\boldsymbol{x} = [x_1, x_2, \cdots x_n]$. During the optimization, we look for the place where the objective function has a minimum or a maximum value. There are many different methods to solve this problem, most of them were developed for the problem of minimizing the function.

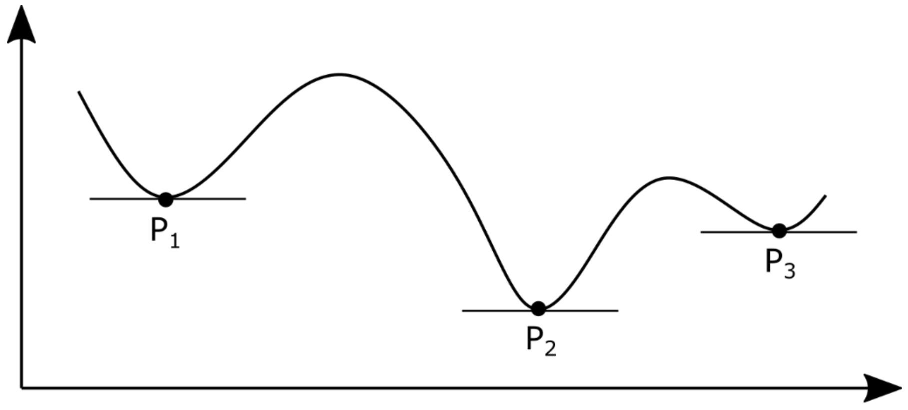
$$f(x) \to min.$$

If the extreme value to be determined is not the minimum but the maximum, we can still trace the problem of maximization back to the minimization by multiplying with (-1) times the original function: $max(f(x)) = min(-f(x))$.

The extreme value is always examined in a given interval or range. They can be within that range

- local minima/maxima or
- a global minimum/maximum.

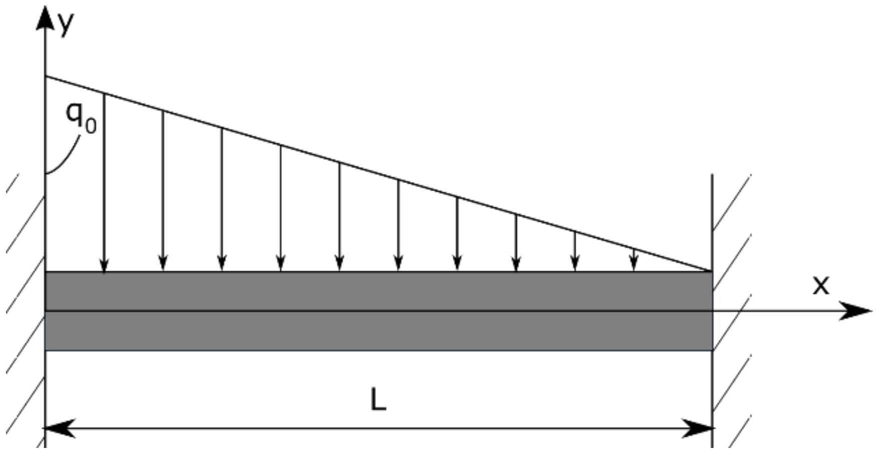
Local minima are the places where the function value in any small neighborhood of the point is greater than the function value at this point (points P_i in the figure). If there are several local minima in a domain with different function values, then the point corresponding to the smallest function value is the global minimum P_2 in the figure).



We can also talk about unconditional and conditional extreme value (or unconstrained and constrained optimization). In the case of constrained optimization, we search for the minimum of the function in such a way that the points must also satisfy some constraint or condition. There can be different cases, one or more conditions given by equations or inequalities, these constraints can be linear or non-linear. Different methods can be used in different cases (e.g. Lagrange method, penalty function method, Karush-Kuhn-Tucker conditions, linear programming). First, we learn about the methods developed for unconditional extreme value problems.

FINDING THE MINIMUM OF A UNIVARIATE FUNCTION

Now let's look at an example from elasticity theory, where we are looking for the location and value of the maximum deflection! We have a fixed I-beam with a linearly distributed load as shown in the figure.



The dimensions and loads of the beam are as follows:

- Length of the beam: $L = 3000\ mm$
- Moment of inertia: $I = 5.29 \cdot 10^7\ mm^4$
- Modulus of elasticity (Young's modulus): $E = 70\,000\ N/mm^2$
- Maximum value of distributed load: $q_0 = 15\ kN/m\ = 15\ N/mm$

The deflection of the beam is given by the following function::

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

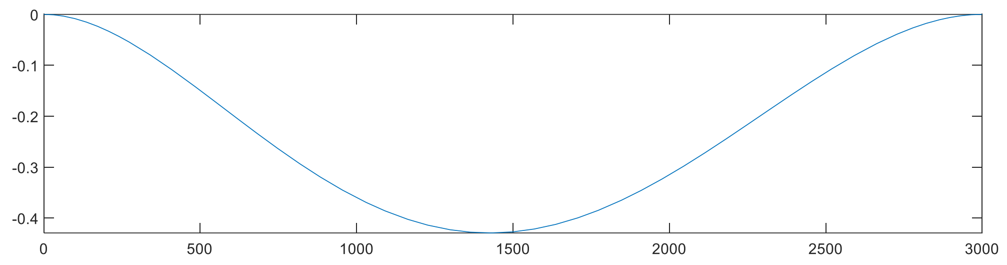
1) How much is the deflection (in mm) at 1 and 2 m?
2) Where will the deflection equal to 0.3 mm?
3) Where is the maximum (along the x-axis) of the deflection (in mm) and what is its value?

First, we define the variables!

```
> %% Deflection calculation
> format longG
> E = 70000; I = 5.29e7; q0 = 15; L = 3000; EI = E*I;
```

It is advisable to merge the variables E and I into the bending stiffness (EI) so that they are not confused with the Euler number 'e' (2.71...) and the imaginary unit 'i' ($\sqrt{-1}$). This could cause problems later with symbolic calculations. Next, we can define the function of the deflection. Our only variable is the x distance. We can visualize the function to get a better idea about the deflection.

```
> y = @(x) q0/(120*L*EI)*(x^5-5*L*x^4+7*L^2*x^3-3*L^3*x^2)
> % plot deflections
> figure(1); fplot(y,[0 3000])
```



1) How much is the deflection (in mm) at 1 and 2 m?

The answer to the first question can be given by a simple substitution. Pay attention to the fact that everything was defined in millimeters before, so here too the numbers must be replaced in mm!

```
> y(1000), y(2000) % deflection at 1 or at 2 m: -0.3601 and -0.3151 mm
```

2) Where will the deflection equal to 0.3 mm?

We can no longer solve this question by simple substitution, we have to solve the following non-linear equation:

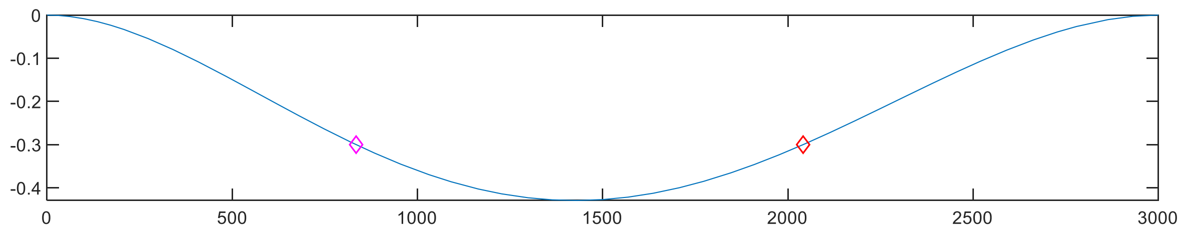
$$\frac{q_0}{120LEI}(x^5 - 5Lx^4 + 7L^2x^3 - 3L^3x^2) = -0.3$$

The deflection values mean negative coordinates! Let's rearrange the equation to zero and give the resulting equation a different name:

$$h(x) = \frac{q_0}{120LEI}(x^5 - 5Lx^4 + 7L^2x^3 - 3L^3x^2) + 0.3 = 0$$

We can solve this task by root finding! There will be two places where the value of the y coordinate will be exactly -0.3 mm, for these we can take an initial guess from the figure.

```
> % where will the deflection equal to 0.3 mm?
> h = @(x) y(x)+0.3
> h1 = fzero(h,1000) % 834.388
> h2 = fzero(h,2000) % 2040.970
> % check
> y(h1) % -0.3
> y(h2) % -0.3
> hold on; plot(h1,y(h1),'md',h2,y(h2),'rd')
```

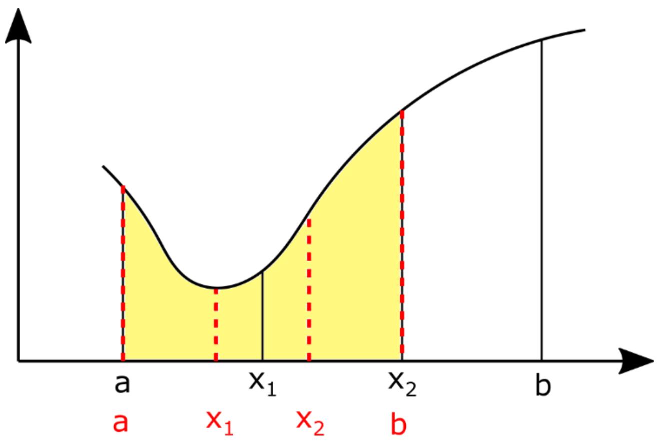


Before answering the third question, let's look at two methods for determining the minimum of a univariate function!

INTERVAL METHODS - TERNARY SEARCH ALGORITHM

The interval method is similar to the closed interval methods used to find the roots of a nonlinear equation. We have to specify a certain closed interval [a, b] that contains one minimum of the function (in other words, the function is unimodal in the interval), but instead of calculating only one function value inside the interval, we will use two points (x_1, x_2). Similar to the closed interval methods, the interval must be narrowed in some way until the solution is found. For this, let's examine the function values at two interior points (x_1, x_2)!

As the function only has one minimum in the interval, its value is monotonically decreasing until the minimum and then it is monotonically increasing. If the function value at x_1 is smaller than that at x_2, it means that the minimum is in the interval [a, x_2]. Similarly, if the function value at x_2 is smaller than the function value at x_1, then the minimum has to be in the interval [x_1, b]. Therefore, we can shrunk the interval and found our new a and b values. See the figure! We can find a new x_1 and x_2 in this new interval and continue the algorithm while the interval is bigger than some tolerance value or we have reached a certain number of iterations.



As long as the function is unimodal in the interval, the method will converge. The more interesting question is how to define the most efficient x_1 and x_2 values (that is, how to find the minimum with fewest calculations)? One approach is to distribute the points equally, x_1 over 1/3 and x_2 over 2/3 of the interval. This is called ternary search algorithm, which can be implemented in MATLAB with the following code.

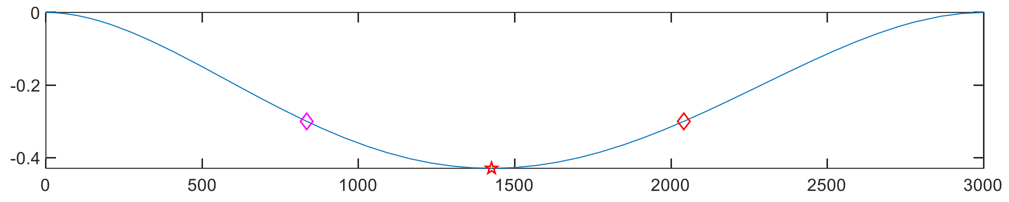
```
> function [x, i] = ternary(f, a, b, tol)
> i = 1;
>  x1 = a + 1/3*(b-a);
>  x2 = b - 1/3*(b-a);
>  while abs(x2-x1) > tol
>    if f(x1) < f(x2)
>      b = x2;
>    else
>      a = x1;
>    end
>    i = i+1;
>    x1 = a + 1/3*(b-a);
>    x2 = b - 1/3*(b-a);
>  end
>  x = (x1+x2)/2;
> end
```

3) Where is the maximum (along the x-axis) of the deflection (in mm) and what is its value?

Let's find the place of maximum deflection using the interval method! Although we talk about maximum deflection values, this task is actually a minimum search! If we observe the coordinate system in the first figure, we will see that the deflections mean negative displacement values, so the maximum deflection means the smallest y coordinate. The starting unimodal interval can be [1000, 2000] based on the figure!

```
> % interval method - ternary search
> [x1 i1] = ternary(y,1000,2000,1e-6)
> % x1 =  1.4259e+03; i1 =  50
> y1 = y(x1) % -0.4293
> hold on; plot(x1,y1,'rp')
```

So, in a total of 50 iterations, we found the minimum location (the location of the maximum deflection) at 1425.9 mm, and its value was -0.4293 mm.



GOLDEN-SECTION SEARCH[1]

There is, however, a more efficient method than selecting the third points! We can use the golden ratio. This ratio often occurs in nature and is also often used in the arts. Using the golden ratio, we can divide a section L into two parts (L=L1+L2) in such a way that the ratio of the larger section to the total length is the same as the ratio of the smaller section to the larger one.

$$R = \frac{L2}{L} = \frac{L1}{L2}$$

Let's express L1 and L2 as a function of R and L: $L2 = R \cdot L$; $L1 = L2 \cdot R = L \cdot R^2$. Substitute this into the equation $L = L1 + L2$:

$$L = L \cdot R^2 + R \cdot L$$

Dividing this by L and ordering to 0, we get the next formula:

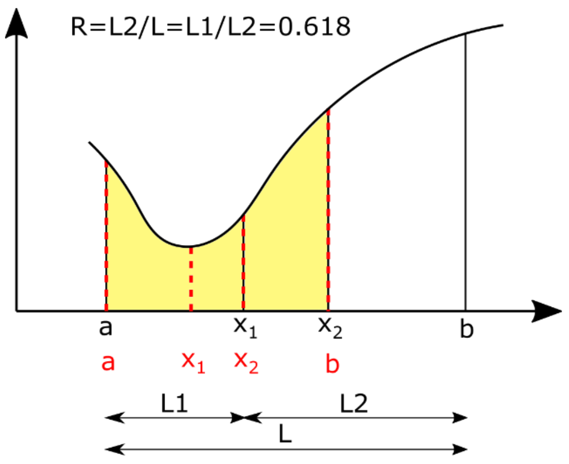
$$R^2 + R - 1 = 0$$

The only positive root of the quadratic equation will be the desired ratio, the golden ratio. This is how the smaller and larger section are proportional to each other, and the larger section to the total length.

$$R = \frac{\sqrt{5} - 1}{2} = 0.618$$

[1] Supplementary material for home study.

Let's see how we can use the golden ratio for a more efficient optimization by modifying the selection of interior points of the interval method!



Select the internal points symmetrically so that the two points are at a distance of 0.618·L from both ends of the section. In this case too, let's narrow the interval based on the function values of the internal points. So the location of the minimum will be in the interval between the two neighbors of the point with the smallest function value. The difference compared to the previous method is that, due to the properties of the golden section, one of the internal points of the new interval will be the same as one of the previous internal points. In the figure, the point x_2 of the new interval is the same as the point x_1 of the previous interval! This means that there is no need to recalculate the value of the function at this point, it is enough to perform the calculation at the other new internal point. This can be a significant time saver, especially for very complex functions. Let's see how we could implement this in Matlab (golden.m)!

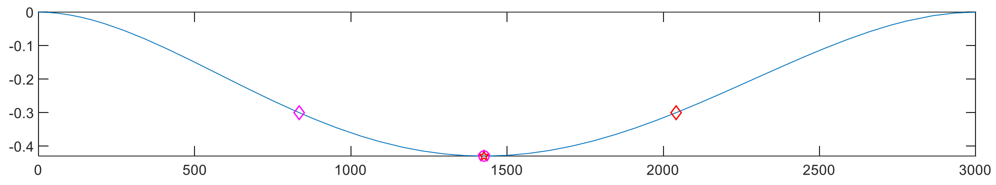
```
> function [x, i] = golden(f, a, b, tol)
> i = 1; R = (sqrt(5)-1)/2;
> x1 = b - R*(b-a);
> x2 = a + R*(b-a);
> f1 = f(x1); f2 = f(x2);
>
> while abs(x2-x1)>tol
>   if f1 < f2
>      b = x2;
>      x2 = x1; f2 = f1; % taken from previous iteration
>      x1 = b - R*(b-a);
>      f1 = f(x1);  % this 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;
> end
> x = (x1+x2)/2;
> end
```

In the first iteration, the function values at points x_1 and x_2 must be calculated, but after that it is enough to calculate only one function value at each iteration, the other can be taken from the previous iteration!

(Note: The interval/golden section method can also be solved with a recursive algorithm, see goldenRec.m file.)

Let's find and plot the place of maximum deflection with this method also!

```
> % solution with golden section method
> [x2 i2] = golden(y,1000,2000,1e-6)
> % x2 = 1.4259e+03; i2 = 42
> y2 = y(x2) % -0.4293
> plot(x2, y2, 'mo')
```



Now, instead of the previous 50 iterations, 42 iteration steps were enough for the solution. Moreover, the difference in terms of the number of function evaluations is much larger. When using the ternary search algorithm, 2 function values had to be calculated in each iteration, i.e. 50*2=100 function evaluations were performed. In the case of the golden ratio, 2 evaluations had to be performed in the first iteration, but after that only one evaluation was needed per iteration, i.e. the function value had to be calculated a total of 43 times instead of 100! For complex functions, this is a big advantage over ternary search.

NEWTON-METHOD

If calculating the derivative of the function is not a problem, we can also solve the extreme value search by finding the roots of the first derivative. Let's now apply Newton's method to find extreme values of a function! In contrast to the root finding of a function, instead of solving the equation $f(x) = 0$, we have to solve $f'(x) = 0$. Nevertheless, the same algorithm can be used now (see the previous **newton.m** file).

```
> function [x2, i] = newton(f, df, x0, delta, N)
>     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
```

Iteration formula for finding the roots of a function, solving $f(x) = 0$:

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

Newton's method can be used similarly to find the extreme values, solving $f'(x) = 0$ equation. I this case $f(x)$ should be replaced with $f'(x)$, and $f'(x)$ with $f''(x)$:

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

The advantage of Newton's method is that it can be used to find both minimum and maximum locations. The disadvantage is that it is necessary to calculate both the first and second derivatives! Let's solve the previous problem using Newton's method now! Let's choose one of the endpoints of the previous interval (2000) as a starting value!

The original equation is:

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

The first (or second) derivative of this equation is not too difficult to determine even without a computer, since it is a polynomial. The first derivative:

$$f(x) = y' = \frac{q_0}{120LEI}(5x^4 - 20Lx^3 + 21L^2x^2 - 6L^3x),$$

Of course, Matlab can also be used to determine the derivative with respect to *x*, symbolically. To compare Matlab’s soluton with the derivation without a computer, first convert the variables *EI, L, q_0,* and *x* to symbols, define the symbolic expression *ys* with them, and then calculate the derivatives symbolically.

```
>  %% Newton method
>  % Determination of derivatives
>  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)
>  dx=diff(ys,x)
>  % -(q0*(6*L^3*x - 21*L^2*x^2 + 20*L*x^3 - 5*x^4))/(120*EI*L)
>  ddx = diff(ys,x,2)
>  % -(q0*(6*L^3 - 42*L^2*x + 60*L*x^2 - 20*x^3))/(120*EI*L)
```

For later use, we will need functions instead of symbolic expressions! Let's use the **matlabFunction** command to transform the symbolic expressions!

```
>  % Let's transform the symbolic expression dx into a function!
>  dxf = matlabFunction(dx)
>  % @(EI,L,q0,x)(q0.*(L.*x.^3.*2.0e+1+L.^3.*x.*6.0-x.^4.*5.0-
   L.^2.*x.^2.*2.1e+1).*(-1.0./1.2e+2))./(EI.*L)
```

In the result, we see that a four-variable (*EI,L,q_0,x*) function has been created! This is due to overriding variables *EI,L,q_0* when defining them as symbolic. However, we need a univariable function. To do this, we need to re-enter the values of the *EI, L* and *q_0* variables, and then we can simply copy the obtained symbolic results after the beginning of the function definition using CTRL+C/CTRL+V.

```
>  % Let's transform the symbolic expressions into function!
>  E = 70000; I = 5.29e7; q0 = 15; L = 3000; EI = E*I;
>  dxf = @(x) -(q0*(6*L^3*x - 21*L^2*x^2 + 20*L*x^3 - 5*x^4))/(120*EI*L)
>  ddxf = @(x) -(q0*(6*L^3 - 42*L^2*x + 60*L*x^2 - 20*x^3))/(120*EI*L)
```

Also, with a small change, we can use the **matlabFunction** command here as well. For this, the variables *EI, L, q_0* must be redefined, and then their values must be substituted into the symbolic expressions with the **subs** command. With the help of the **subs** command, we can replace all the previously defined variables at once as numbers, or we can specify the value of any variable.

```
>  % Another solution with matlabFunction
>  E = 70000; I = 5.29e7; q0 = 15; L = 3000; EI = E*I;
>  dx = subs(dx),ddx = subs(ddx)
>  dxf = matlabFunction(dx)
> ddxf = matlabFunction(ddx)
```

Finally, the solution using Newton's method:

```
> % solution using Newton's method:
> [xn in] = newton(dxf, ddxf, 2000, 1e-6, 100)
> % xn = 1.4257e+03; in = 3
```

Now we have reached the solution in only 3 iterations. It can be seen that this method converges much faster, if it converges.

USING A MATLAB BUILT-IN FUNCTION (FMINSEARCH, FMINBND)

Of course, Matlab also has its own built-in function that can be used to search for a minimum, e.g. the **fminsearch** command. This uses the Nelder-Mead simplex method.

```
> % Matlab built-in function - fminsearch
> 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)
> xmin = fminsearch(y,2000) % 1.4259e+0
```

We can call the **fminsearch** function to return not only the location of the minimum, but also its value and other details.

```
> [x,fval,exitflag,output] = fminsearch(y,2000)
> % x = 1425.9; fval = -0.42935
> i = output.iterations % i = 25 - number of iteration
> n = output.funcCount % n = 50 - number of evaluation
```

Matlab also has a one-variable optimization command that requires a closed interval: **fminbnd**. You can call this function with an interval [a,b]: **fminbnd**(function,a,b).

BIVARIATE OPTIMIZATION

It is often necessary to determine extreme values, not only for one, but also for multi-variable tasks. This can be the determination of the location of the smallest or largest value of a surface, or even the maximum displacement of certain points of a spatial support in the x, y direction. But it is also possible to ideally choose road network junctions by minimizing distances. Even in the unconstrained multivariate case, there are several solution methods to choose from. We can use, for example, the multivariate Newton method, the gradient method, and the Nelder-Mead simplex method.

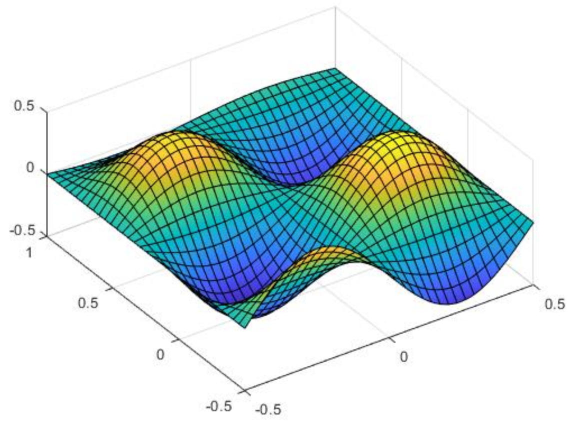
Let's first look at the determination of the extreme values of a surface given by a function! The surface can be defined with the following function:

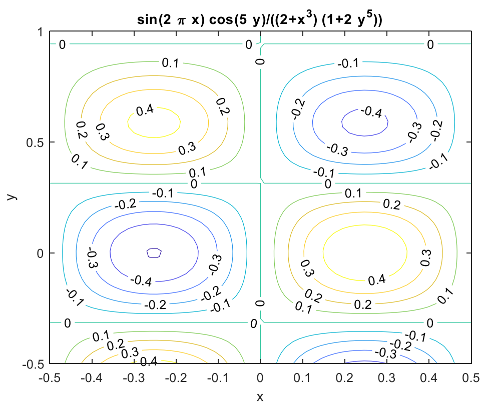
$$f(x,y) = \frac{\sin(2\cdot\pi\cdot x)\cdot\cos(5\cdot y)}{(2+x^3)\cdot(1+2\cdot y^5)}$$

The range where we look for extreme values is: $-0.5 \le x \le 0.5$; $-0.5 \le y \le 1$.

Let's define the surface and plot it in a spatial figure and also with contour lines, labeling the contour lines!

```
> %% Determination of local extreme values of a bivariate function
> clc; clear all; close all; format shortG;
> f = @(x,y) sin(2*pi*x).*cos(5*y)./((2+x.^3).*(1+2*y.^5))
> figure(1); fsurf(f,[-0.5 0.5 -0.5 1])
> figure(2); h1 = ezcontour(f,[-0.5 0.5 -0.5 1])
> set(h1,'ShowText','on')
```





We see that there are several local minima and maxima in the given range. How can we find them? We can use several methods, for example the Nelder-Mead method used by one of Matlab's built-in functions (**fminsearch**), or the multivariate Newton method, which is a generalization of the method used in the univariate case.

NELDER-MEAD SIMPLEX METHOD

The Nelder-Mead simplex method was originally published in 1965 (Nelder and Mead, 1965[2]). This is one of the most well-known methods for multivariate optimization without the use of derivatives, a so-called direct search method. It is a heuristic algorithm that is simple and easy to apply.

The algorithm is based on a simplex, which is a geometric shape consisting of *n+1* vertices in an *n*-dimensional space, a triangle in 2D, and a tetrahedron in 3D. The procedure does not require the calculation of gradients, only the determination of the values of the peaks. The procedure itself performs various geometric transformations on the simplex, e.g. mirroring, stretching, shrinking, expanding, contracting, in order to reduce the function values in the vertices.

During the method, we take a starting polyhedron (simplex), in the 2-dimensional case a triangle, and then using various transformations during the procedure (stretching, shrinking, mirroring) we change the position of the 3 points so that it always aligns with the shape of the surface until it shrinks to the vicinity of the minimum location. To understand the procedure, it is worth looking at the following animation as an example: https://en.wikipedia.org/wiki/File:Nelder-Mead_Himmelblau.gif.

SOLUTION USING THE NELDER-MEAD SIMPLEX METHOD IN MATLAB

Determine the locations of the local extreme value of the given surface using the Nelder-Mead simplex method! Matlab's built-in **fminsearch** procedure uses this

[2] J. A. Nelder, R. Mead, A Simplex Method for Function Minimization, The Computer Journal, Volume 7, Issue 4, January 1965, Pages 308–313, https://doi.org/10.1093/comjnl/7.4.308

algorithm for optimization. Note that this function can only be used for minimum search, not for maximum search (as expected from its name).

It is very important that, like other Matlab built-in functions, **fminsearch** can only work with vector variable functions, so the function must be vectorized in the first step. The contour map shows that there are 2 minima and 2 maxima in the examined range. We can easily choose a starting value for them from the figure. The initial values should be given in a column vector!

```
> F = @(v) f(v(1),v(2)); % conversion to vector variable
> % initial values for minimum locations:
> x01 = [-0.2;0]; x02 = [0.2;0.6];
> % initial values for maximum locations
> x03 = [0.2;0]; x04 = [-0.2;0.6];
```

First, determine the location and value of the 2 minima and plot them in the contour map as well!

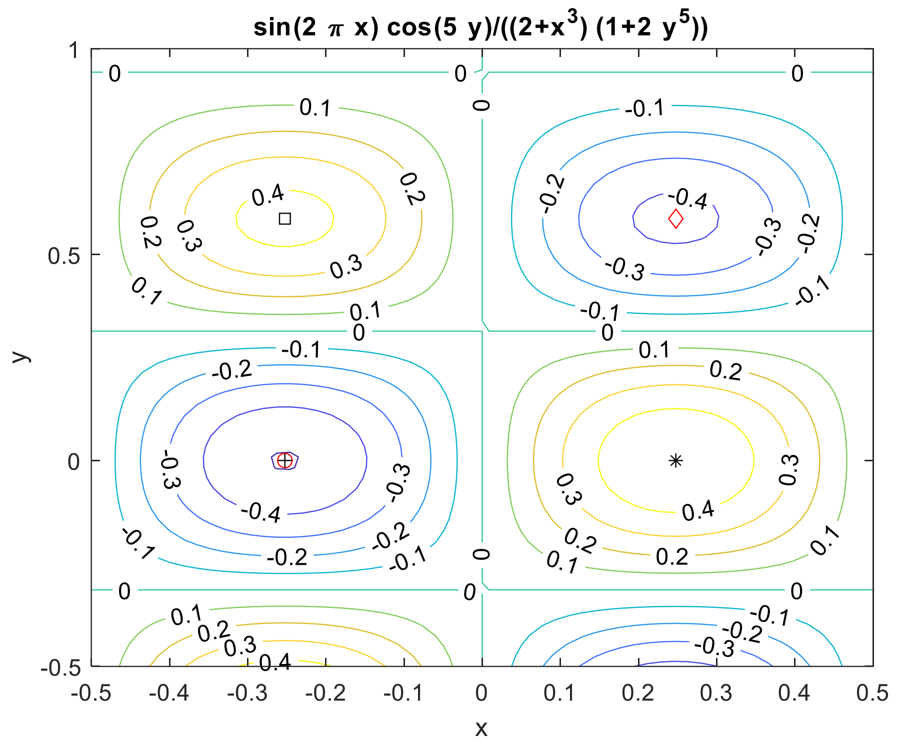
```
> % Determination of minimum places
> [sol1 f1] = fminsearch(F,x01) %  -0.25241   8.9111e-06;   -0.504
> [sol2 f2] = fminsearch(F,x02) %  0.24767      0.58715;   -0.42622
> hold on; plot(sol1(1),sol1(2),'ro'); plot(sol2(1),sol2(2),'rd')
```

Let's find the location and value of the maximum points also! First, let's see what happens if we give a starting value to our optimization algorithm near the maximum point!

```
> % Determination of maximum places
> [sol3 f3] = fminsearch(F,x03) %  -0.25241   1.7236e-05; -0.504
> plot(sol3(1),sol3(2),'k+');
```

The obtained value is the same as our first solution, which was a minimum location. **fminsearch** can only be used to find a minimum location. If we want to get a maximum location with it, then we have to modify the function definition and define -1 times the original function. That is, the problem of maximum search is transformed into minimum search. Of course, if we want to get the value of the function back at this point, then the maximum location must be substituted back into the original function.

```
> Fmax = @(v) -1*F(v) % Defining a function times -1
> [sol3 f3] = fminsearch(Fmax,x03) %  0.24765  -2.3376e-05; -0.49618
> plot(sol3(1),sol3(2),'k*');
> [sol4 f4] = fminsearch(Fmax,x04) %  0.24765  -2.3376e-05; -0.49618
> plot(sol4(1),sol4(2),'ks');
> % Maximum locations substituted back into the original function
> f3 = F(sol3) % 0.49618
> f4 = F(sol4) % 0.43293
```



MULTIVARIATE NEWTON-METHOD[3]

Of course, many other optimization methods can also be used. Another method could be the multivariate Newton method, which can be easily generalized from the univariate case. The advantage of Newton's method is that it can be used to determine both minimum and maximum locations, as it searches for the locations where the derivative will be zero, i.e. the tangent is horizontal. The disadvantage is that the first and second derivatives must also be determined, which is computationally demanding.

In the univariate case, the iteration formula of Newton's method for finding the extreme value was as follows:

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

This formula can be generalized to the multivariate case, but the gradient vector (∇f) of the multivariate function should be used instead of the first derivative, and the Hesse matrix (H) should be used instead of the second derivative. Here, too, the several variables must be specified in one (x) vector.

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

where the Hessian matrix is the matrix of the second partial derivatives of the function f(x), and the gradient vector is the vector of the first partial derivatives. For two variables f(x,y), the gradient vector and the Hesse matrix will be:

[3] Supplementary material for home study.

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

Previously, we used the Jacobian matrix (containing in a vector the first partial derivatives of the equations) for solving nonlinear equation systems. The Hessian matrix can also be generated with the Jacobian matrix calculated for the gradient vector of the function.

The function below (gradmulti.m) is the solution of the multivariate Newton method in Matlab, where the input is the gradient vector (grad), the Hessian matrix (hesse), an initial position (x0), tolerance value for the stopping condition (eps) and a maximum iteration number (nmax). Note that the original function F is not needed for the solution, only the Hessian matrix and the gradient vector.

In the output, x1 is the solution itself, i is the iteration number, and X contains the successive solutions.

```
> function [x i X] = gradmulti(grad, hesse, x0, eps, nmax)
>
> x1 = x0 - pinv(hesse(x0))*grad(x0);
> i=1;
> X=[x0 x1];
>
> while and(norm(x1 - x0) > eps, i < nmax)
>     x0 = x1;
>     x1 = x0 - pinv(hesse(x0))*grad(x0);
>     i = i + 1;
>     X = [X x1];
> end
> x = x1;
```

Among Matlab's built-in functions, we also find methods that perform gradient-based optimization, such as the **fminunc** function, which uses quasi-Newton minimization. In the case of the **fminunc** function, it is also possible to use the Newton method, if we specify the gradient vector and the Hessian matrix as optional inputs. For the exact method of calling the function, it is worth looking at the Matlab documentation.

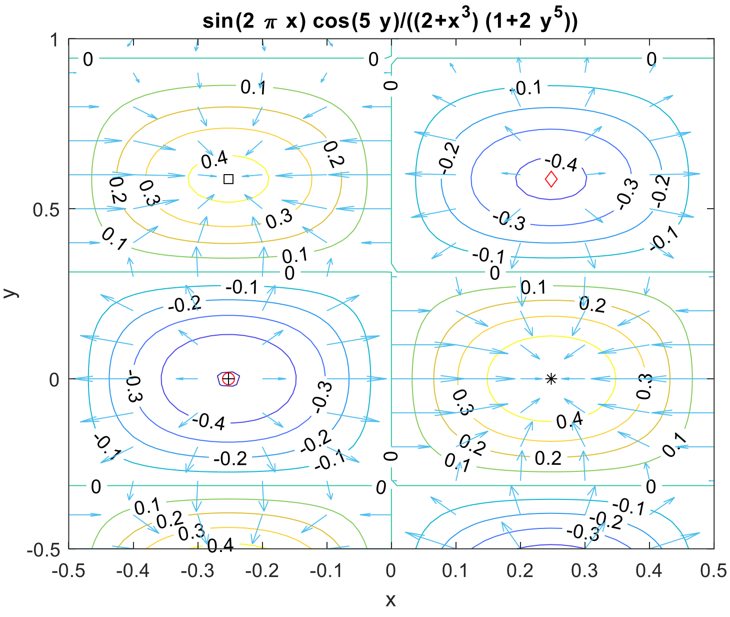
SOLUTION WITH MULTIVARIATE NEWTON-METHOD[4]

We have seen that in the multivariate case, the gradient vector and the Hessian matrix will be needed, instead of the first and second derivatives used in the univariate case. Let's generate these in Matlab. To create the gradient vector, we can use Matlab's **gradient** command, both numerically and symbolically. For a better visualization, we first plot the numerically produced gradient vectors! To do this, create a grid with the **meshgrid** command and calculate the gradient values in this grid, which can be plotted with the **quiver** command! (See the material for the numerical derivation exercise in

[4] Supplementary material for home study.

detail!) This is presented for illustration purposes only, the numerical gradients are not needed for the solution.

```
>   % Representation of a gradient vector numerically
>   [X,Y] = meshgrid(-0.5:0.1:0.5, -0.5:0.1:1);
>   Z = f(X,Y); % function values in the grid points
>   [px,py] = gradient(Z); % calculation of gradients numerically
>   quiver(X,Y,px,py)
```



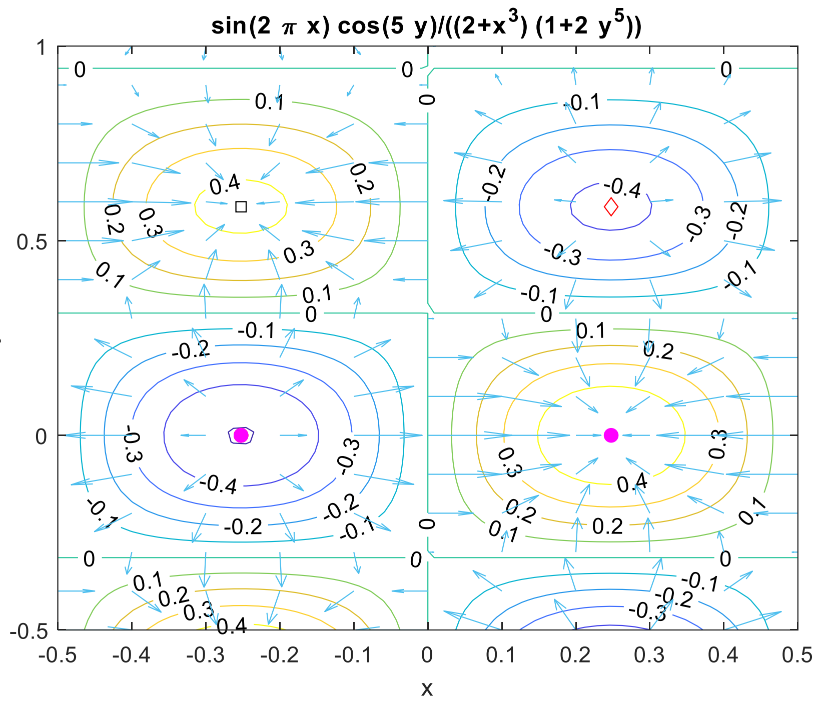
To apply the multivariate Newton method, we need the gradient vector and the Hessian matrix in the form of a vector variable function, not a numerical matrix. This can be determined by symbolic calculations by applying the **gradient** and **hessian** commands to the symbolic function *f*!

```
>   % Functions symbolically
>   syms x y;
>   fs = f(x,y) % (cos(5*y)*sin(2*pi*x))/((x^3 + 2)*(2*y^5 + 1))
>   G = gradient(fs) % Gradient vector symbolically
>   % (2*pi*cos(5*y)*cos(2*pi*x))/((x^3 + 2)*(2*y^5 + 1)) -
    (3*x^2*cos(5*y)*sin(2*pi*x))/((x^3 + 2)^2*(2*y^5 + 1))
>   % - (5*sin(5*y)*sin(2*pi*x))/((x^3 + 2)*(2*y^5 + 1)) -
    (10*y^4*cos(5*y)*sin(2*pi*x))/((x^3 + 2)*(2*y^5 + 1)^2)
>   H = hessian(fs) % Hessian matrix symbolically
>   % Convert H and G into vector variable functions
>   G = matlabFunction(G) % Convert symbolic expression G into function
>   H = matlabFunction(H) % Convert symbolic expression H into function
>   G = @(x) G(x(1),x(2)) % conversion to vector variable
>   H = @(x) H(x(1),x(2)) % conversion to vector variable
```

With the method, we can determine either minimum or maximum location. Let's use the initial values given earlier to find a minimum and a maximum location!

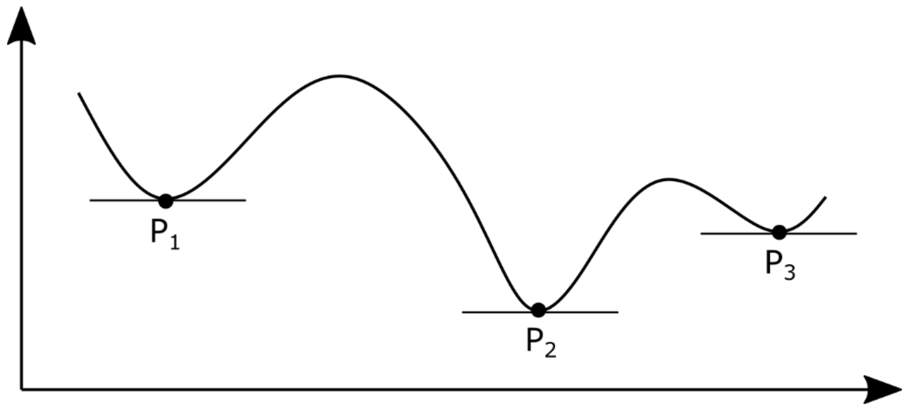
```
>   % Search for minimum and maximum location
>   [min1 i M1] = gradmulti(G,H,x01,1e-6,100) % one of the minimum
>   [max1 i M2] = gradmulti(G,H,x03,1e-6,100) % one of the maximum
>   plot([min1(1);max1(1)],[min1(2);max1(2)],'mo','MarkerFaceColor','m')
>   F(min1) % -0.504
>   F(max1) % 0.49618
```

The method converged very quickly, in 3 iterations we reached the minimum location within the desired accuracy.



GLOBAL OPTIMIZATION

As we have seen, several local minima can exist in a given range. The smallest of these will be the global minimum (P_2 in the figure). The methods described so far, such as the interval method, Newton's method, or the Nelder-Mead simplex method used by Matlab, require an initial value to be specified and usually "get stuck" at the nearest local minimum. In the case of several minima, they must be called with several initial values, and then by comparing them we can decide which is the global minimum. However, we can use a heuristic approach to approximate the global minimum without specifying an initial value, such as the genetic algorithm, which we deal with next.



GLOBAL OPTIMIZATION WITH GENETIC ALGORITHM[5]

Let the task be to determine the global minimum of a univariate function *f(x)* in a given interval $x \in [a,b]$, i.e.

$$\min_{x\in[a,b]} f(x)$$

Genetic algorithms are population-based special evolutionary algorithms, heuristic search techniques that can be used to search for an optimum. They borrowed their techniques and terms from evolutionary biology.

Genetic algorithms are generally characterized by the fact that the objective function to be optimized must satisfy relatively few constraints. For example, unlike traditional procedures, it does not have to be formulated in a closed form, linear, or derivable. The only assumption these procedures make is that a small change in the value of the function's parameters should result in a small change in the value of the function (so the "surface" of the function should not be completely random).

The objective function whose minimum location is sought can also be called a cost function or an energy function. Another possible name is the fitness function.

Genetic algorithms are implemented with computer simulations. The elements of the search space make up the individuals of the population, which can be crossed and mutated, so that new individuals can be created. During the operation of the genetic algorithm, on the one hand, it creates new individuals with the crossover and mutation operators, and on the other hand, it filters out individuals with a worse fitness function value and removes them from the population.

Steps of the procedure:

- Initialization

The easiest way to generate the initial population is randomly. The size of the population depends on the nature of the problem, but most often it consists of a few hundred or a few thousand individuals. Traditionally, individuals are evenly distributed in the search space.

- Selection

In each succeeding generation, a portion of the current population is selected for reproduction. It is usually based on fitness, where fitter individuals (according to the fitness function) are more likely to be selected. The fitness function measures the quality of the individual.

- Genetic operators

New individuals can be created from individuals by crossover (or recombination) and mutation, these are genetic operators, usually applied randomly.

- Stop condition

[5] The theoretical summary of genetic algorithms was prepared based on Sándor Laky's 2012 PhD thesis entitled Metaheuristic optimization in geodesy

Genetic algorithms usually run until a stopping condition is met. This can be the achievement of a certain number of generations, or if the fitness value of the best individual no longer improves significantly with each iteration.

We will not go into the details of genetic algorithms here, they help to solve many difficult programming tasks, but they do not guarantee that they will find the optimum. Since global minimization is relatively slow and requires many operations, the following strategy is appropriate:

- with a relatively small population and allowing few generations, we get close to the global minimum,

- then, starting from there, we refine it using some local optimization method

An illustrative example of how genetic algorithms work can also be found on the MATLAB YouTube channel: https://www.youtube.com/watch?v=1i8muvzZkPw

USING GENETIC ALGORITHM IN MATLAB

Let's determine the global minimum of the previous surface on the given domain using a genetic algorithm! We will need the **ga** (genetic algorithm) function for this, and the **gaoptimset** function for the appropriate settings.

With the help of **gaoptimset**, we can set the number of individuals of the initial population and the number of generations to be used. Let's now use a population of 200 elements and 20 generations, and set it to display the iteration steps as well! Let's also set the display format of the number to be longer, to more decimal places, so that the difference between several runs is visible!

```
> % initialization
> format longG;
> options =
  gaoptimset('Generations',20,'PopulationSize',200,'Display','iter');
```

The **ga** algorithm can be called by specifying several parameters, in general the command looks like this:

```
[x,fval] = ga(fitnessfcn,nvars,A,b,Aeq,beq,LB,UB,nonlcon,options)
```

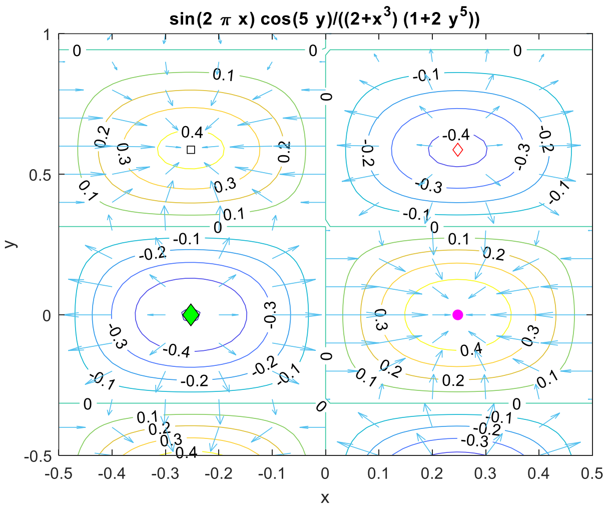
Among the inputs, *fitnessfcn* is the objective function to be optimized, *nvars* is the number of variables (in this case 2, since the task is bivariate optimization). *A,b,Aeq,beq* are parameters of linear constraints (inequalities and equations), which are not present in this case, if it is necessary to use them, they can significantly slow down the running time. *LB* (lower boundary), *UB* (upper boundary) contain the lower and upper boundaries of the investigated range in one vector. Nonlinear conditions could be specified in the *nonlcon* parameter. The last parameter is the *options*, where we can take into account the values specified in **gaoptimset**. Instead of the parameters we don't want to use, we have to enter an empty matrix, since Matlab expects the data in the specified order. It is important that Matlab, like with other built-in commands, expects a vector variable function! If the task is to optimize a bivariate function, it must first be converted into a vector variable function. We have already done this before, when we generated the function *F* from the function *f* to be able to use it with **fminsearch**.

```
> %% Solution with a genetic algorithm
> % [x,fval] = ga(fitnessfcn,nvars,A,b,Aeq,beq,LB,UB,nonlcon,options)
> A = []; b = []; Aeq = []; beq = []; % linear constraint
> nlc = []; % nonlinear constraint
> LB = [-0.5,-0.5]; % lower boundary
> UB = [0.5,1]; % upper boundary
> % approximate minimum location with genetic algorithm
> xga = ga(F,2,A,b,Aeq,beq,LB,UB,nlc,options)
> % xga = -0.252456968480699     -0.000708545893755685
> % fga =  -0.503991955761332
```

Let's run the algorithm several times! We will see that the solution will be slightly different each time, but it will not change in magnitude.

Refine the result using a local minimum search algorithm, use the minimum determined by the genetic algorithm as a starting value!

```
> % refine minimum with local optimization algorithm
> [xy_min zmin]=fminsearch(F,xga)
> % xy_min = -0.252380312148973     -7.22400617317352e-06
> % zmin =  -0.503995085159198
> plot(xy_min(1),xy_min(2),'kd','MarkerSize',10, 'MarkerFaceColor','g')
```



Let's compare the minimum value found with the genetic algorithm and the value refined with the local optimization algorithm!

```
> F(xy_min)<F(xga) % logical 1 -> true
```

NEW FUNCTIONS USED IN THE CHAPTER

subs	-	Substituting specific values into a symbolic variable
fminsearch	-	Finding the minimum of a single/multivariable function using the Nelder-Mead simplex method
fminunc	-	Unconditional extreme value search using quasi-Newton minimization.
fminbnd	-	Search for minimum location by specifying an interval
ga	-	Optimization using a genetic algorithm
gaoptimset	-	Initialization of genetic algorithm, specification of parameters