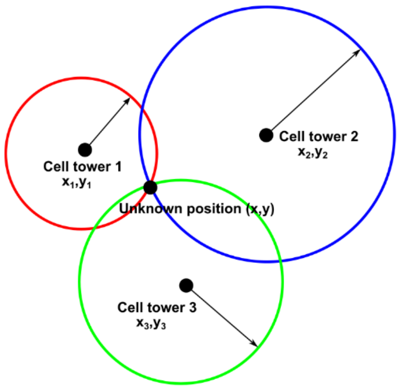
7. SOLUTION OF NONLINEAR SYSTEM OF EQUATIONS

Civil engineering problems often require the solution of nonlinear equations, for example when looking for the intersection of several nonlinear equations. Such examples are typically the various geodetic point determination tasks (arc intersection, intersection, resection etc.), but we also have to solve non-linear equation systems during the design of beam structures and retaining walls. Now let's look at a mobile phone positioning task where we measure the distances between the mobile phone and the surrounding cell phone towers. This is actually an arc intersection task.

When determining the position of mobile phones, the distance between the device and cell towers is measured. The distances define a circle around the cell towers/base stations (see figure). In case of 2 measured distances, the circles can be described by 2 quadratic equations, and their intersection gives the possible positions of the mobile phone. The equations can be given in the following implicit form, in the case of two known bases:



$$(x-x_1)^2+(y-y_1)^2=r_1^2$$
$$(x-x_2)^2+(y-y_2)^2=r_2^2$$

If at least 3 distances and the coordinates of the mobile towers are known, the x,y coordinates of the unknown location can be determined, but this will be an overdetermined task with 3 equations for 2 unknowns. In the case of 2 measured distances, we get 2 possible solutions for the position. We will now deal with the latter case, when we have 2 equations and 2 unknowns.

Solving a two-variable equation system consisting of two quadratic algebraic polynomials can also be traced back to finding the roots of a 4th degree polynomial, but now we will deal with the solution of the general nonlinear equation system.

VECTOR NOTATION OF A SYSTEM OF EQUATIONS

For the general solution of nonlinear systems of equations, let's introduce the vector notation for the equations and unknowns. In this way, it will be easier to refer to them, and the built-in functions of Matlab also need to be entered in this form.

We can usually find a solution to a system of equations when the number of unknowns is the same as the number of equations. Systems of equations are usually given in the following form:

$$f_1(x_1,x_2,x_3,\dots,x_n)=0$$
$$f_2(x_1,x_2,x_3,\dots,x_n)=0$$
$$f_3(x_1,x_2,x_3,\dots,x_n)=0$$
$$\vdots$$
$$f_n(x_1,x_2,x_3,\dots,x_n)=0$$

In the vector notation mode, the multiple variables are stored in a vector. Let this be the vector $\boldsymbol{x}$ whose elements are: $\boldsymbol{x} = (x_1, x_2, x_3, \dots, x_n\ )$.The equations are also stored in a vector ($\boldsymbol{f}$), whose elements are: $\boldsymbol{f} = (f_1, f_2, f_3, \dots, f_n\ )$. With this notation, we can simply describe the system of equations.

$$\boldsymbol{f}(\boldsymbol{x}) = 0$$

In the univariate case, we have to use one initial value (x_0). In the multivariable case, it is necessary to specify initial values for all variables, which are given in a vector $\boldsymbol{x_0}$. The more variables we have, the more difficult it is to determine a good initial value for each one. In practice, however, we can utilize the engineering knowledge of the given problem, and based on these, we can often give good approximate values even for multivariable cases. If we can plot the equations (for example in the case of two variables), we can also determine the starting values based on the figure

EXAMPLE FOR SYSTEM OF NONLINEAR EQUATIONS

Our task will be position determination with a mobile phone. In the example, let's assume that we only know the distance from 2 mobile towers, so we will get two possible locations for the position. The coordinates of each base station (cell tower) and the distance measurements are summarized in the table below (in kilometers).

Number of the cell tower (base)	**X coordinate** (x_i)	**Y coordinate** (y_i)	**Measured distances** (r_i)
1	1	1	5
2	10	8	8

The equations can be given in the following implicit form, where we look for the x,y coordinates:

$$(x-1)^2 + (y-1)^2 = 5^2$$

$$(x-10)^2 + (y-8)^2 = 8^2$$

As a first step, we rearrange the equations to zero:

$$(x-1)^2 + (y-1)^2 - 5^2 = 0$$

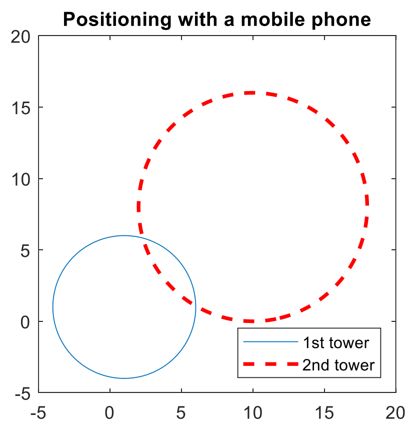
$$(x-10)^2 + (y-8)^2 - 8^2 = 0$$

These equations are not given in the usual explicit form: $y = f(x)$, but they are given in implicit form: $f(x, y) = 0$. There are different Matlab commands to plot functions in explicit form (**fplot**, **ezplot**) and curves in implicit form (**fimplicit**, **ezplot**). The **fplot** command can only be used to plot univariate functions given in explicit form $y = f(x)$, it cannot be used to display implicitly specified curves.

equation	*form*	*recommended Matlab command*	*other Matlab command (older Matlab versions and Octave also)*
$y = f(x)$	explicit form	**fplot**	**ezplot**
$f(x, y) = 0$	implicit form	**fimplicit**	**ezplot**

First, we plot the two curves with the **fimplicit** command. Colors, line type, etc. can be entered similarly to the **plot** or **fplot** commands.

```
> clear all; clc; close all;
> f1 =@(x,y) (x-1).^2 + (y-1).^2 -
  5^2;
> f2 =@(x,y) (x-10).^2 + (y-8).^2 -
  8^2;
> figure(1);
> g1=fimplicit(f1,[-5 20 -5 20]);
> hold on;
> fimplicit(f2,[-5 20 -5 20],...
                '--r','LineWidth',2);
> axis equal;
> legend('1st tower','2nd
  tower','Location','SE')
> title('Positioning with a mobile
  phone')
```



How can we find the points of intersection of the two equations given in implicit form ($f(x,y)=0$)? To plot these curves we define them as multivariate functions. In the univariate case ($f(x)=0$), we searched for the roots of the nonlinear function rearranged to zero, for example using the Newton's method. Newton's method can also be used in multivariable cases, with some modifications. Let's check this out!

MULTIVARIATE NEWTON'S METHOD

Let's take a detailed look at one of the most well-known methods, the multivariate Newton's method for solving systems of nonlinear equations, to better understand a possible way of solving these systems. The multivariate method can be generalized from the univariate case. In the univariate case, Newton's method could be derived from the linearization of the function:

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

This resulted in the iteration formula of Newton's method:

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

In the multivariate case, the iteration formula is very similar:

$$\boldsymbol{x}_{\mathbf{i+1}} = \boldsymbol{x_i} - \boldsymbol{J}(\boldsymbol{x_i})^{-1}\cdot \boldsymbol{f}(\boldsymbol{x_i})$$

where $\boldsymbol{f}(\boldsymbol{x_i})$ is the system of equations in a column vector, $\boldsymbol{x_i}$ are the values of the variables in a vector, and $\boldsymbol{J}(\boldsymbol{x_i})^{-1}$ instead of $1/f'(x_i)$ is the inverse of the n×n Jacobian matrix at $\boldsymbol{x_i}$. The elements of the Jacobian matrix are the partial derivatives of the equations:

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \frac{\partial f_2}{\partial x_3} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}$$

For example, the Jacobian matrix of the system of equations below is:

$$\begin{array}{c} 3x^2 + 2y + 1 = 0 \\ -x^3 - 5y + 2 = 0 \end{array} \rightarrow \; J(x) = \begin{bmatrix} 6x & 2 \\ -3x^2 & -5 \end{bmatrix}$$

The calculation of the multivariable iteration formula requires an inverse calculation instead of a simple reciprocal, which we mostly try to avoid because it is slow and numerically unstable and imprecise. How can we determine $\boldsymbol{x}_{\mathbf{i+1}}$ without inverse calculation? Let's rearrange the iteration equation in a form where we can use the matrix decompositions!

$$\boldsymbol{x}_{\mathbf{i+1}} - \boldsymbol{x}_{\boldsymbol{i}} = -\boldsymbol{J}(\boldsymbol{x}_{\boldsymbol{i}})^{-1} \cdot \boldsymbol{f}(\boldsymbol{x}_{\boldsymbol{i}})$$

Let’s introduce the notation $\boldsymbol{\Delta x} = \boldsymbol{x}_{\boldsymbol{i+1}} - \boldsymbol{x}_{\boldsymbol{i}}$ and multiply both sides by $\boldsymbol{J}(\boldsymbol{x}_{\boldsymbol{i}})$! In this way, we get a system of linear equations, which we can solve with matrix decomposition!

$$\boldsymbol{J}(\boldsymbol{x}_{\boldsymbol{i}}) \cdot \Delta\boldsymbol{x} = -\,\boldsymbol{f}(\boldsymbol{x}_{\boldsymbol{i}})$$

where $\boldsymbol{J}(\boldsymbol{x}_{\boldsymbol{i}})$ is the *nxn* Jacobian matrix at $\boldsymbol{x}_{\boldsymbol{i}}$, $-\boldsymbol{f}(\boldsymbol{x}_{\boldsymbol{i}})$ is an *nx1* vector. We have seen several effective methods for solving systems of linear equations, all of which used different matrix decompositions. Let's now use the solution of the form $x = A\backslash b$, which uses LU decomposition in case of square A matrix. Once the value of $\Delta\boldsymbol{x}$ is known, $\boldsymbol{x}_{\boldsymbol{i+1}}$ can be calculated.

The task to be solved in each iteration is therefore:

- $\boldsymbol{J}(\boldsymbol{x}_{\boldsymbol{i}}) \cdot \Delta\boldsymbol{x}_{\boldsymbol{i}} = -\,\boldsymbol{f}(\boldsymbol{x}_{\boldsymbol{i}})$ solving a system of linear equations for Δx_i
- Calculation of $\boldsymbol{x}_{\boldsymbol{i+1}} = \boldsymbol{x}_{\boldsymbol{i}} + \Delta\boldsymbol{x}_{\boldsymbol{i}}$ until $\boldsymbol{f}(\boldsymbol{x}) \approx 0$ or $\Delta\boldsymbol{x} \approx 0$ (until it is smaller than a given tolerance value).

MULTIVARIATE NEWTON’S METHOD IN MATLAB

Let's write a function that implements the multivariate Newton’a method! Let the function be called **newtonsys**. (Save the file in the newtonsys.m file!) The stopping criteria should now be that successive iteration solutions differ less than a specified tolerance. Also, the loop should stop when it has reached the specified maximum number of iterations. The input is the system of equations (*f*), the Jacobian matrix (*J*), the initial values (*x₀*), tolerance (ε) and maximum number of iterations. The function has two outputs, the solution and the number of iterations performed.

```
> function [x1, n] = newtonsys (f, J, x0, eps, nmax)
>     dx = J(x0)\-f(x0)); % first iteration
>     x1 = x0 + dx;       % first iteration
>     n = 1;
>     while norm(x1-x0)>eps && n<=nmax
>         x0 = x1;
>         dx = J(x0)\-f(x0);
>         x1 = x0 + dx;
>         n = n + 1;
>     end;
> end
```

SOLUTION WITH MULTIVARIATE NEWTON'S METHOD

a) Determination of the Jacobian matrix

As we have seen, the multivariable Newton's method requires, in addition to the original equations and the initial values, the partial derivatives of the equations for the Jacobian matrix. We can produce this with symbolic calculations. The partial derivatives can also be determined one by one with the **diff** command, but Matlab has a command to directly generate the Jacobian matrix (**jacobian**). To use the jacobian command, let's now define the system of equations symbolically!

```
>  %% Solution with multivariate Newton's method
>  syms x y
>  fs1 = (x-1).^2 + (y-1).^2 - 5^2
>  fs2 = (x-10).^2 + (y-8).^2 - 8^2
>  disp('Jacobian matrix')
>  js = jacobian([fs1; fs2])
>  % [  2*x - 2,  2*y - 2]
>  % [ 2*x - 20, 2*y - 16]
```

Let's define the Jacobian matrix as a function, instead of a symbolic expression! We can do this by copying each element to the appropriate places using CTRL+C/CTRL+V, or we can also use the **matlabFunction** command here, which generates a function from a symbolic expression (there may be cases where this does not work).

```
>  J=@(x,y) [2*x - 2,  2*y - 2; 2*x - 20, 2*y - 16] % or
>  J = matlabFunction(js)
```

b) Vectorization of the system of equations and the Jacobian matrix

For the solution, we have to use vector variables instead of x,y variables, where the unknowns are in the ***v*** vector (v_1=x és v_2=y) and the equations are in the ***f*** vector.

$$\boldsymbol{v} = \binom{x}{y}; \boldsymbol{F} = \binom{f_1}{f_2}$$

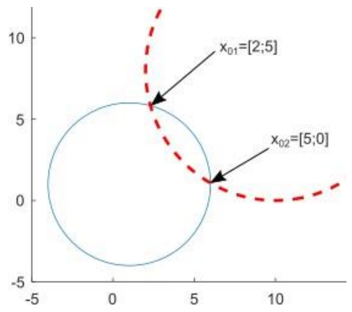
The same is required for the Jacobian matrix! Let's vectorize our equations and the Jacobian matrix!

```
>  % let's define f and J with vector variables
>  F = @(v) [f1(v(1),v(2)); f2(v(1),v(2))];
>  J = @(v) J(v(1),v(2));
```

c) Solution by Newton's method

Let's solve the problem with the defined **newtonsys** function. For this, the newtonsys.m file must be in the same directory where the current script file is saved. For the numerical solutions, as mentioned earlier, it is necessary to enter an initial value. The closer the initial value is to the actual solution, the faster the solution will be, the more likely the method will converge. Initial values for the x and y coordinates of the intersection point can be selected from the figure.

```
> % Solution with Newton’s method
> % initial values from the figure
> x01 = [2; 5] % first intersection
> x02 = [5; 0] % second intersection
> [x1 i1] = newtonsys (f, J, x01, 1e-6, 100); %
  1st solution
> [x2 i2] = newtonsys (f, J, x02, 1e-6, 100); %
  2nd solution
```

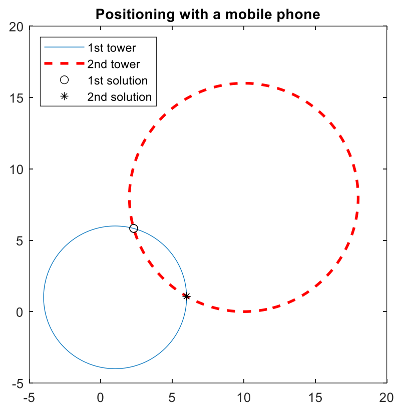


Write the results in a formatted format and plot them in the figure! As a check, substitute back the obtained results into the implicit equations to see if we really get 0 (in the case of multiple variables, the norm of the residual vector can be calculated as a check, i.e. the length of the deviation vector)!

```
> disp('Possible positions using Newton''s method:')
> fprintf('1st solution: %.4f,%.4f, number of iterations: %d\n',x1, i1)
> fprintf('2nd solution: %.4f,%.4f, number of iterations: %d\n',x2, i2)
> % check
> norm(f(x1)) % 1.4648e-14
> norm(f(x2)) % 0
> plot(x1(1),x1(2),'ko')
> plot(x2(1),x2(2),'k*')
> legend('1st tower','2nd tower',...
>     '1st solution', '2nd solution','Location','NW')
```

The solutions are:

```
Possible positions using Newton's method:
1st solution: 2.3005,5.8279, number of iterations: 5
2nd solution: 5.9995,1.0721, number of iterations: 5
```



The error is considered to be 0 within the numerical accuracy.

Let's look at the solution with another starting value (x=1 and y=1)!

```
> % choosing another starting value
> disp('Let''s enter another starting value!')
> x0 = [1; 1]
> [x3 i3] = newtonsys (f, J, x0, 1e-6, 100);
> fprintf('3rd solution: %.4f,%.4f, number of iteration: %d\n',x3, i3)
```

The solution in this case:

```
Warning: Matrix is singular to working precision.
Warning: Matrix is singular, close to singular or badly scaled. Results may
be inaccurate.
RCOND = NaN.
3rd solution: NaN,NaN, number of iteration: 2
```

What happened? Why don't we have a solution? (NaN = Not a Number)

Because the Jacobian matrix is singular. The geometric reason is that the initial value coincides with the coordinate of one of the cell towers, the center of the circle. It really does matter how we choose the starting value, and what method we use to solve the problem, because it is not certain that the solution will converge

SOLUTION NUMERICALLY WITH FSOLVE

Of course, Matlab also has built-in functions for solving systems of nonlinear equations. Let's first look at the **fsolve** command, which uses a numerical nonlinear system solver command. Several algorithms are built into **fsolve**, based on these, the program tries to find the optimal solution by minimizing the sums of squares of the remaining differences (in single-variable cases, it is better to use the significantly faster **fzero** command). Let's use the same initial values as before. The system of equations must also be given in vector form, but the Jacobian matrix is not required, it is optional (its specification can speed up the algorithm). In order to find the two different solutions, we need to run the **fsolve** command twice, once with initial values close to first solution, and once with initial values close to the second solution.

```
> % numerical solution - fsolve
> x1 = fsolve(f,x01) % x1 = [2.3005;  5.8279]
> x2 = fsolve(f,x02) % x2 = [5.9995;  1.0721]
```

Of course, we got the same results as before. For numerical verification, let's substitute the obtained results back into the original system of equations and check the errors (the deviations from 0).

```
> % check
> norm(f(x1)) % 7.4621e-12
> norm(f(x2)) % 7.0300e-08
```

The error within the numerical accuracy can be considered 0. When using **fsolve**, the desired accuracy can be specified using the **optimset** variable, similar to **fzero**, with the variables '**TolFun**' and '**TolX**'. With '**TolFun**' you can specify a tolerance for the function value, and with '**TolX**' you can specify a tolerance for the differences of the successive solutions. The default value for both is 10^{-6}. If we want to solve the task with 10^{-9} accuracy, then we can do it this way:

```
> %  solution with 10^-9 precision
> x1 = fsolve(f,x01,optimset('TolFun',1e-9))
```

The **fsolve** command can be called with multiple outputs, so that it also gives us the function values right away, without having to replace them back for checking, and by using **optimset** the individual iteration steps can also be displayed.

```
> % function values and iteration steps
> opt = optimset('TolFun',1e-9,'Display','iter');
> [X,FVAL] = fsolve(f,x01,opt)
```

The solution:

```
X = 2.3005
    5.8279
FVAL = 1.0e-11 *[0.5059  0.5485]
                                        Norm of        First-order   Trust-
region
 Iteration  Func-count     f(x)          step           optimality    radius
     0          3            145                           160              1
     1          6        1.77485       0.970584            12.4             1
     2          9      0.000981063     0.148822            0.284         2.43
     3         12      3.63457e-10    0.00367156        0.000173         2.43
     4         15      5.56836e-23   2.23746e-06        7.13e-11         2.43

Equation solved.
```

```
fsolve completed because the vector of function values is near zero
as measured by the value of the function tolerance, and
the problem appears regular as measured by the gradient.
```

Of course, the **fsolve** command can be used to find the roots of a univariate nonlinear equation, like **fzero**, but the latter is much faster and more efficient (but it cannot be used in multivariate cases). In a similar way, with the help of **fsolve**, we can solve not only algebraic polynomials, but also other systems of equations containing trigonometric, logarithmic, exponential etc. functions.

SOLUTION SYMBOLICALLY WITH SOLVE

The example we solved is an algebraic polynomial. Algebraic polynomials usually have a symbolic solution. In this case, all solutions can be obtained at the same time without entering initial values, just like in the univariate case. Here, the command is also the same, this is the **solve** command. The **solve** command, unlike **fsolve**, can only be used for algebraic polynomials.

To use **solve**, the equations must be entered symbolically, we have already done this during the generation of the Jacobi matrix (fs1 and fs2), let's use them here too! The result contains both solutions at the same time, in exact form, in the form of a symbolic structure (xs). The values of the actual variables can be queried by entering a dot after the name of the structure (xs). It is advisable to convert these into numbers with the **double** command.

```
>  xs = solve(fs1, fs2)
>  %    x: [2x1 sym]
>  %    y: [2x1 sym]
>  xs.x, xs.y % values of variables x,y symbolically
>  %  (77*39^(1/2))/260 + 83/20         69/20 - (99*39^(1/2))/260
>  %  83/20 - (77*39^(1/2))/260         (99*39^(1/2))/260 + 69/20
>  xs = [double(xs.x) double(xs.y)] % values of x,y numerically
>  %     5.9995    1.0721
>  %     2.3005    5.8279
```

Both **solve** and **fsolve** gave the same solution. In the case of **fsolve**, the system of equations first had to be rearranged to zero, and vector variables had to be used. The **fsolve** command had to be called as many times with different initial values as many solutions we had. In the case of **solve**, the equations had to be rearranged to zero, but the equations had to be entered symbolically. The latter provided all the solutions in one step without an initial value, but it can only be used for algebraic polynomials.

INTERSECTION OF A PARAMETRIC CURVE AND A FUNCTION

So far, we have not dealt with curves given in parametric form or with polar coordinates, although many curves cannot be written with traditional Cartesian (x,y) coordinates, only with polar coordinates or in parametric form (e.g. spiral). There are also curves that can be written with traditional Cartesian coordinates (x,y) and in a parametric form as well (e.g. a circle).

Equation of a circle with Cartesian coordinates:

$$x^2 + y^2 = a^2$$

Parametric equation of a circle:

$$x = a \cdot \cos(t)$$

$$y = a \cdot \sin(t)$$

Equation of a circle with polar coordinates:

$$r = a$$

Many interesting curves can be found collected, for example, on the following page: https://mathshistory.st-andrews.ac.uk/Curves/

Now let's solve a task in Matlab where we are looking for the intersection point of a parametric curve with a function. The parametric curve is given below.

$$X(t) = t \cdot \cos(2 \cdot \pi \cdot t)$$

$$Y(t) = t \cdot \sin(2 \cdot \pi \cdot t)$$

$$t \in [0,2]$$

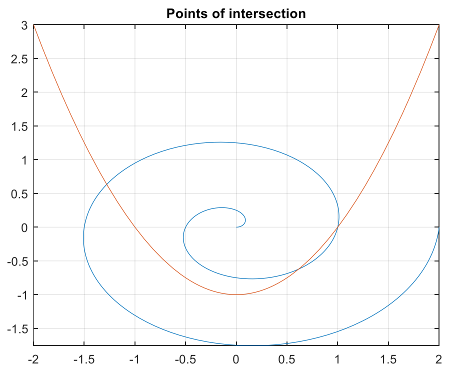
We look for the intersection of this with the following parabola:

$$y = x^2 - 1$$

Let's find all their intersections! To do this, we first plot the curves. A parametric curve can be represented in Matlab in the following way:

– fplot(funx,funy,tinterval)

```
> % Plot the curves
> clc; clear all; close all;
> % Define parametric curve
> xp = @(t) t.*cos(2*pi*t)
> yp = @(t) t.*sin(2*pi*t)
> % define a function to calculate coordinates using a parameter
> XY = @(t) [xp(t),yp(t)]
> figure(1); fplot(xp,yp,[0,2])
> % define parabola
> f = @(x) x.^2-1 % explicit form -> fplot
> hold on; fplot(f,[-2,2])
> grid on; title('Points of intersection')
```



The solution can be obtained in different ways. One solution is to substitute the X,Y values given by the parametric equations into the equation of the parabola and thereby bring the problem back to the solution of a one-variable nonlinear equation. First, let's rearrange the equation of the parabola to zero:

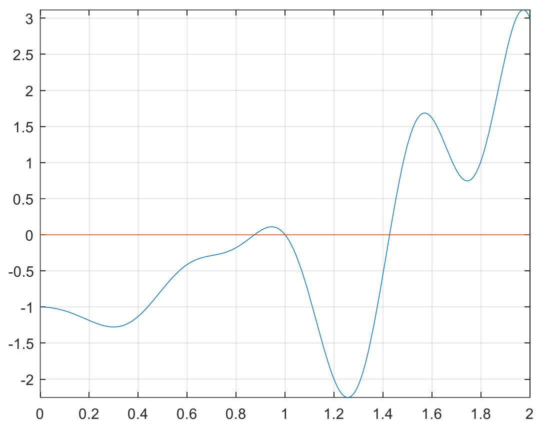
$$x^2 - 1 - y = 0$$

Substituting the parametric equations into the parabola, we get the following:

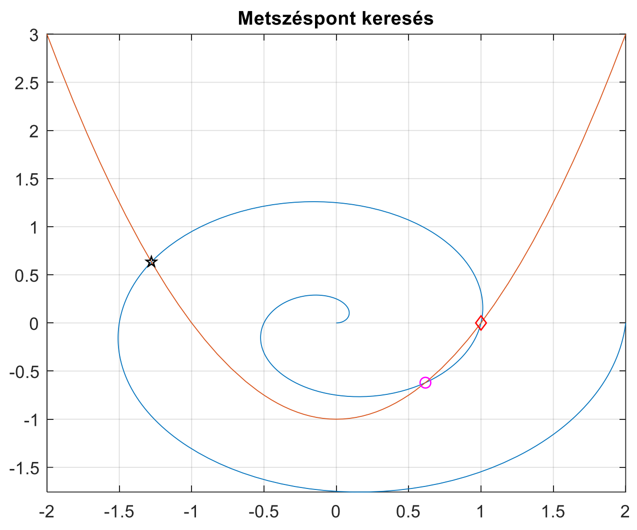
$$(t \cdot \cos(2 \cdot \pi \cdot t))^2 - 1 - (t \cdot \sin(2 \cdot \pi \cdot t)) = 0$$

With this, we reduced the problem to finding the roots of a one-variable nonlinear equation. Let's plot the new equation in another figure and take the initial values from here!

```
>  % Intersection search
>  % rearranged parabola equation to zero: x^2-1-y=0, with substitution
>  g = @(t) (t.*cos(2*pi*t)).^2-1-t.*sin(2*pi*t)
>  % or we can simply use the previously defined functions:
>  g = @(t) (xp(t)).^2-1-yp(t)
>  figure(2); fplot(g,[0,2]);
>  hold on; plot(xlim,[0,0]); grid on;
```



```
>  % initial values from the figure, solution
>  t1 = 0.9; t2 = 1.1; t3 = 1.4;
>  sol1 = fzero(g,t1) % 0.8744
>  sol2 = fzero(g,t2) % 1
>  sol3 = fzero(g,t3) % 1.4267
>
>  % Coordinates and plot of intersection points
>  M1 = XY(sol1) % 0.6159   -0.6207
>  M2 = XY(sol2) % 1.0000   -0.0000
>  M3 = XY(sol3) % -1.2782    0.6338
>  figure(1)
>  plot(M1(1),M1(2),'mo')
>  plot(M2(1),M2(2),'rd')
>  plot(M3(1),M3(2),'kp')
```



Another way to solve the problem is to solve it as a system of equations with 3 variables, where the three variables are x,y,t. When defining the system of equations, pay attention to the fact that the equations must be rearranged to 0 in the first step! In this case, we have to choose initial values for all three variables. For the variables x,y, the initial values can be read from the figure, for the parameter t, move the cursor in the first figure near the desired intersection over the spiral and use the approximate value displayed there by Matlab.

The system of equations rearranged to zero:

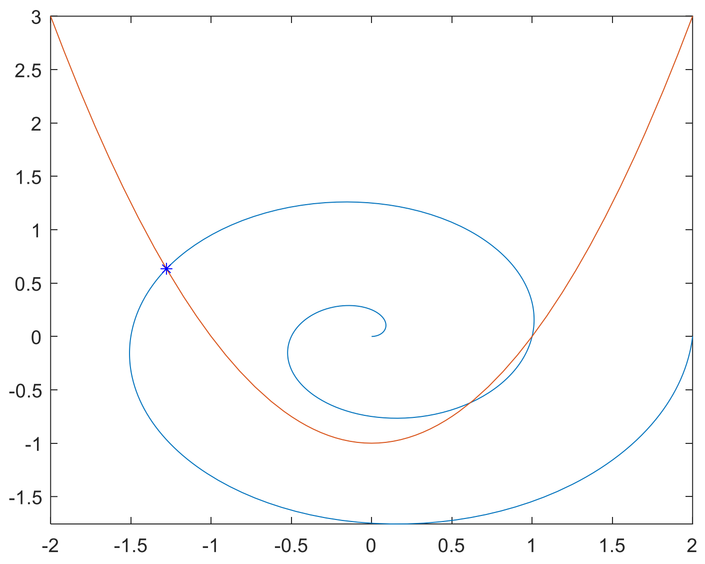
$$x - t \cdot \cos(2 \cdot \pi \cdot t) = 0$$

$$y - t \cdot \sin(2 \cdot \pi \cdot t) = 0$$

$$y - x^2 + 1 = 0$$

Let's find one of the intersection points using this method (initial values taken from the first figure):

```
>  % solution as a system of equations with 3 unknowns
>  F = @(x,y,t) [x-t.*cos(2*pi*t);
>                       y-t.*sin(2*pi*t);
>                       y-x.^2+1]
>   % another solution:
>   % F = @(x,y,t) [x-xp(t); y-yp(t); y-f(x)]
>   F = @(v) F(v(1),v(2),v(3))
>   v01 = [-1; 1; 1.5] % initial value from the figure
>   sol1 = fsolve(F,v01)
>  %        -1.2782
>  %       0.63383
>  %        1.4267
>  plot(sol1(1),sol1(2),'b*')
```



NEW FUNCTIONS USED IN THE CHAPTER

fimplicit	-	Plot implicit function f(x,y)=0
axis equal	-	use equal data unit lengths along each axis
jacobian	-	Calculation of the Jacobi matrix (partial derivatives of an equation)
fsolve	-	Solving nonlinear systems of equations numerically
solve	-	Solving symbolically a system of algebraic polynomials