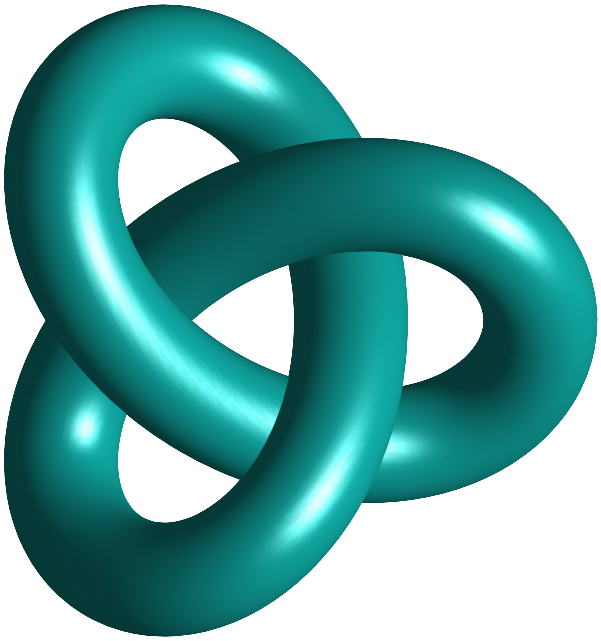


Trefoil knot

Jörg J. Buchholz

December 21, 2022

1. Introduction

In this paper we want to create the trefoil knot depicted in figure 1.1 in Matlab [1].

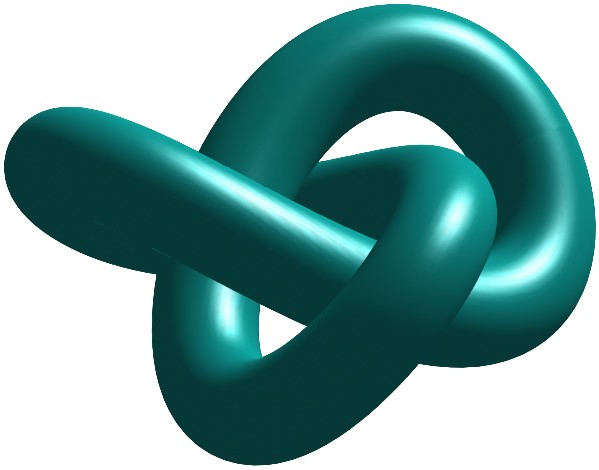


Figure 1.1.: A trefoil knot

Wikipedia [2] says:

> In knot theory, a branch of mathematics, the trefoil knot is the simplest example of a nontrivial knot. The trefoil can be obtained by joining together the two loose ends of a common overhand knot, resulting in a knotted loop.

2. Mathematical background

2.1. Three-dimensional space curve

The surface of the trefoil knot in figure 1.1 is just a radial expansion of the three-dimensional space curve illustrated in figure 2.1

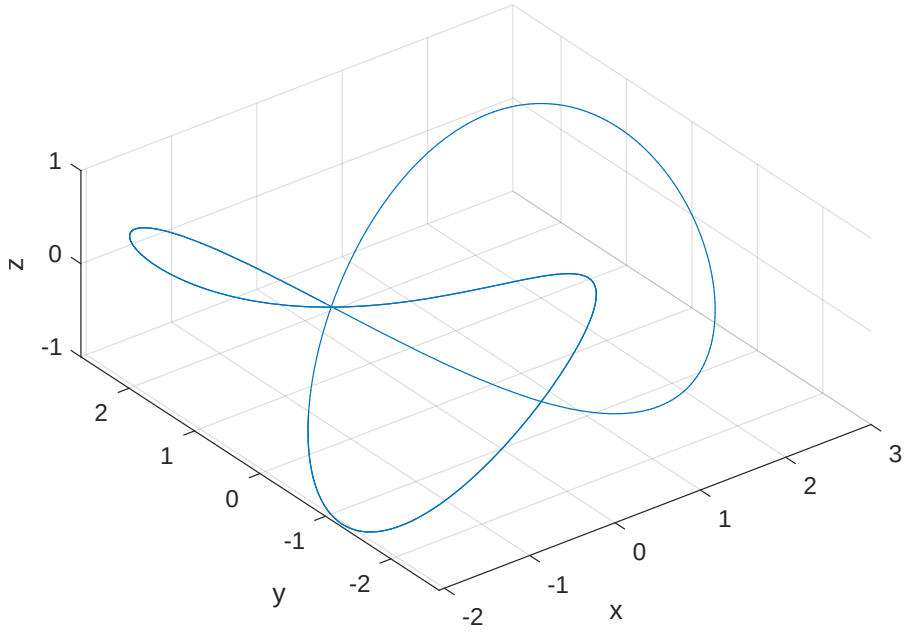


Figure 2.1.: 3D space curve

The space curve is defined in [2] by the parametric equation given in equation (2.1):

$$\boldsymbol{r}(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} = \begin{bmatrix} \cos(t) + 2\cos(2t) \\ \sin(t) - 2\sin(2t) \\ -\sin(3t) \end{bmatrix} \tag{2.1}$$

In order to understand how the space curve is created, we take a closer look at its three components. The first component is a sum of two cosine functions

$$x(t) = x_1(t) + x_2(t) = \cos(t) + 2\cos(2t)$$

of which the first one is a cosine with an amplitude of 1 and an angular frequency[1] of 1 (blue graph in figure 2.2 over one period: $t = 0 \dots 2\pi$)

$$x_1(t) = \cos(t) \tag{2.2}$$

while the second one is a greater, faster cosine with an amplitude of 2 and an angular frequency of 2 (red graph in figure 2.2):

$$x_2(t) = 2\cos(2t) \tag{2.3}$$

Therefore, we can construct the sum of equation (2.2) and equation (2.3) point-by-point for every t-value and come up with the w-shaped yellow graph in figure 2.2.

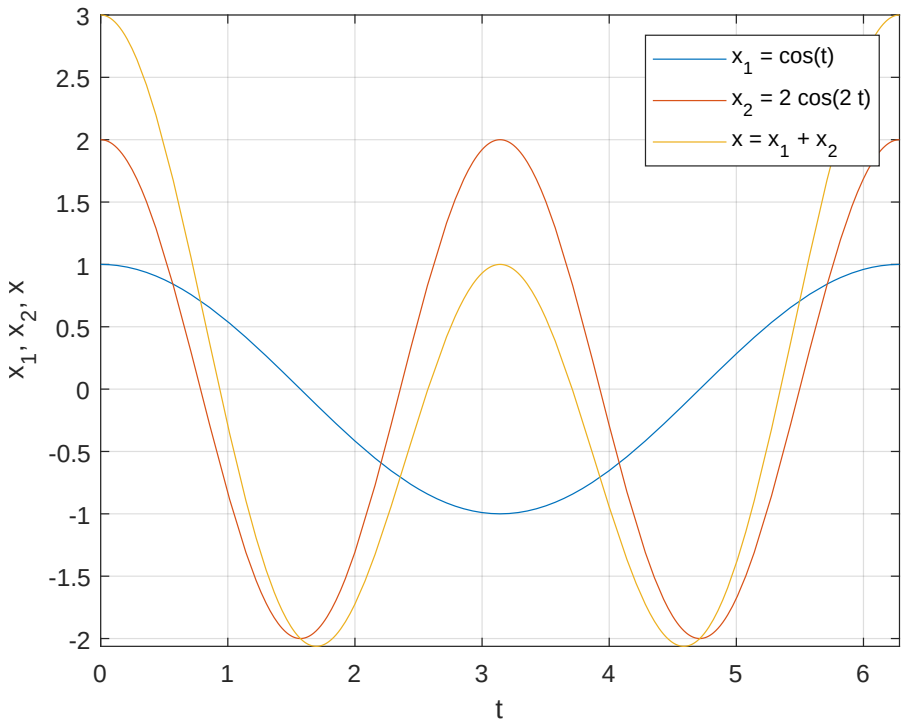


Figure 2.2.: $x(t) = x_1(t) + x_2(t) = \cos(t) + 2\cos(2t)$

Applying the same procedure to the second component of the 3D curve

$$y(t) = y_1(t) + y_2(t) = \sin(t) - 2\sin(2t)$$

we overlay a smaller, slower sine (blue, amplitude: 1, frequency: 1) with a greater, faster, negative sine (red, amplitude: -2, frequency: 2) in order to construct the yellow graph in figure 2.3.

[1] We deliberately omit the physical units of angular frequency and time.

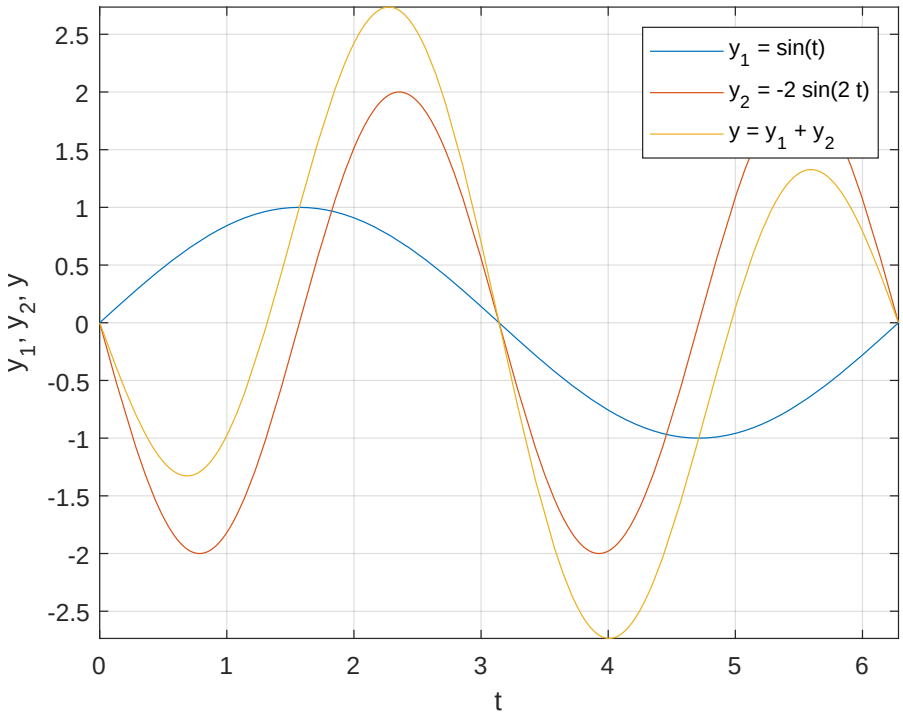


Figure 2.3.: $y(t) = y_1(t) + y_2(t) = \sin(t) - 2\sin(2t)$

In the next step, we draw the graphs of both components $x(t)$ and $y(t)$ in one diagram (figure 2.4)

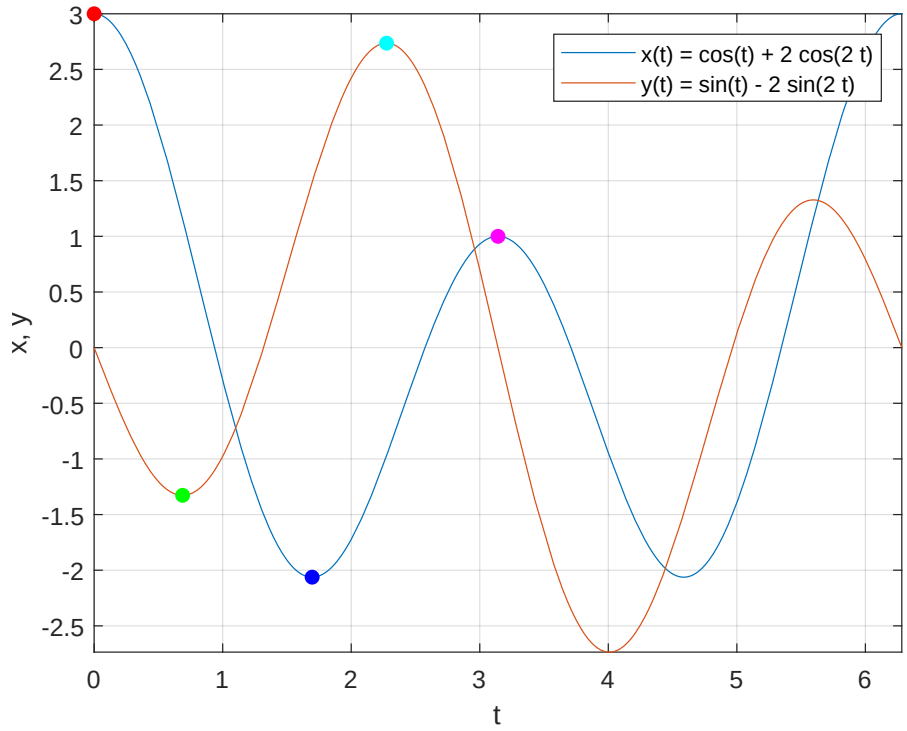


Figure 2.4.: $x(t)$ and $y(t)$

and construct the 2D parametric equation

$$\boldsymbol{r}_{2D}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix} = \begin{bmatrix} \cos(t) + 2\cos(2t) \\ \sin(t) - 2\sin(2t) \end{bmatrix}$$

as a generalized Lissajous curve in figure 2.5, where x is the abscissa, y is the ordinate, and t is the parameter of the curve.

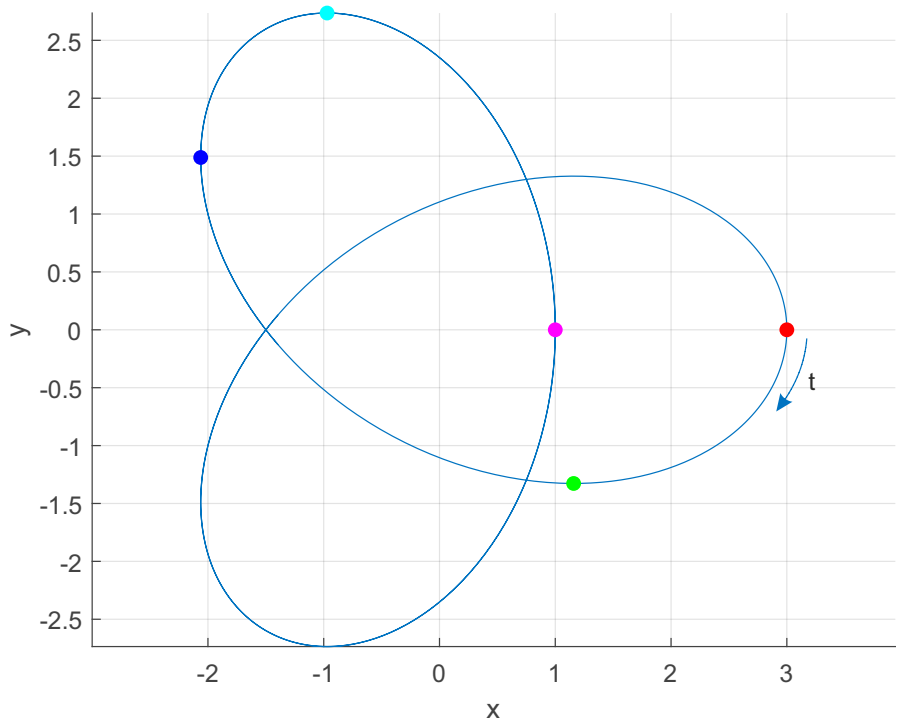


Figure 2.5.: Two-dimensional curve in parametric representation: $\boldsymbol{r}_{2D}(t)$

By comparing figure 2.5 to figure 2.4, we realize that $\boldsymbol{r}_{2D}(t)$

- starts (and ends) with the red dot ● at $\begin{bmatrix}3 & 0\end{bmatrix}$ for $t = 0$ (and $t = 2\pi$), reaches
- a local y-minimum with the green dot ●
- a global x-minimum with the blue dot ●
- a global y-maximum with the cyan dot ●
- a local x-maximum with the magenta dot ● at $\begin{bmatrix}1 & 0\end{bmatrix}$ for $t = \pi$
- ...

Finally, we use the third component of equation (2.1)

$$z(t) = -\sin(3t)$$

depicted in figure 2.6 to move every single point of figure 2.5 into the third dimension.

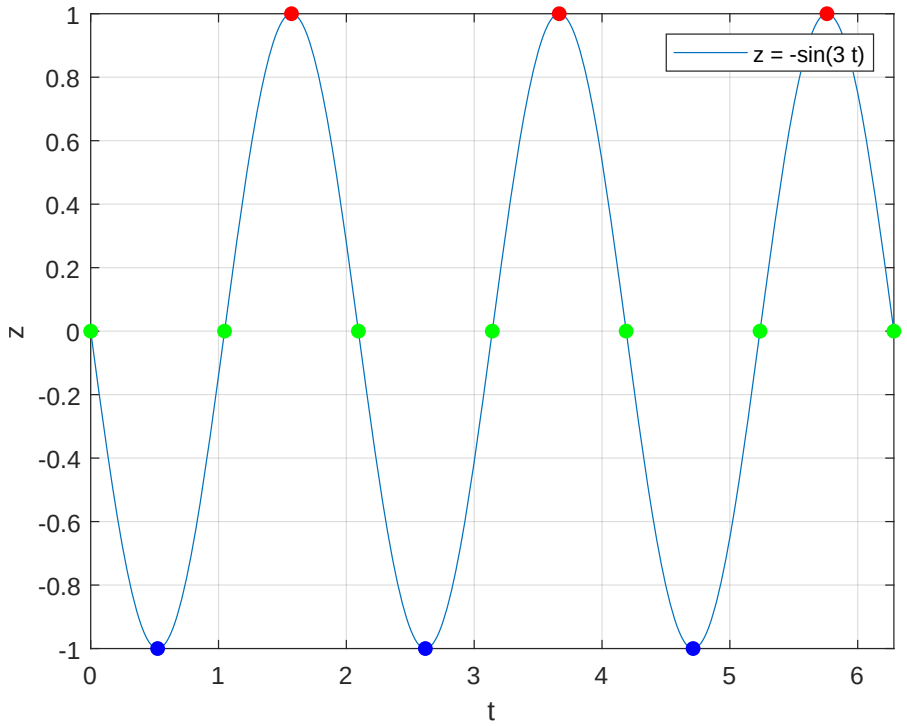


Figure 2.6.: $z(t) = -\sin(3t)$

All red points, $t = (4k + 3) \cdot \frac{\pi}{6}$, $k \in \mathbb{Z}$, defined in figure 2.6 will be raised up to a maximum height of $+1$, while all blue points, $t = (4k + 1) \cdot \frac{\pi}{6}$, will be pushed down to a minimum height of -1, and all green points, $t = k \cdot \frac{\pi}{3}$, remain at a height of 0.

We visualize figure 2.5 in the $z = 0$ plane[2] of the three-dimensional coordinate system in figure 2.7 and indicate that

- red points ● will be raised up
- blue points ● will be pushed down
- green points ● keep their height

[2]Note that all points in figure 2.7 still have a z-component of 0.

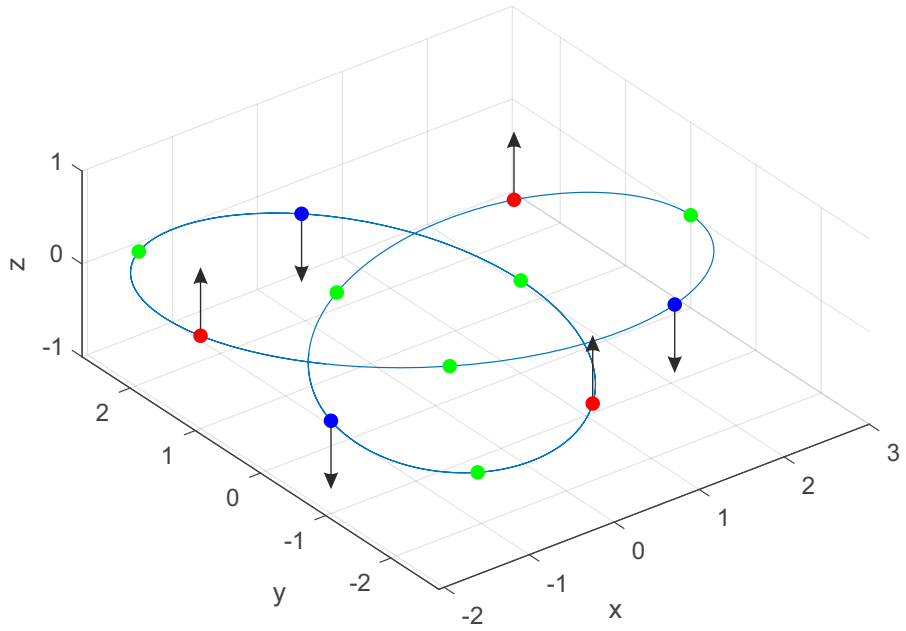


Figure 2.7.: 3D visualization of 2D curve with height motion indicators

And here comes the magic: Knowing that all red dots in figure 2.8 are positioned at the ceiling of the axes box at $z = +1$, all blue dots lie on the floor at $z = -1$, and all green dots (still) have a height of $z = 0$ makes it much easier to interpret the 3D curve in figure 2.8.

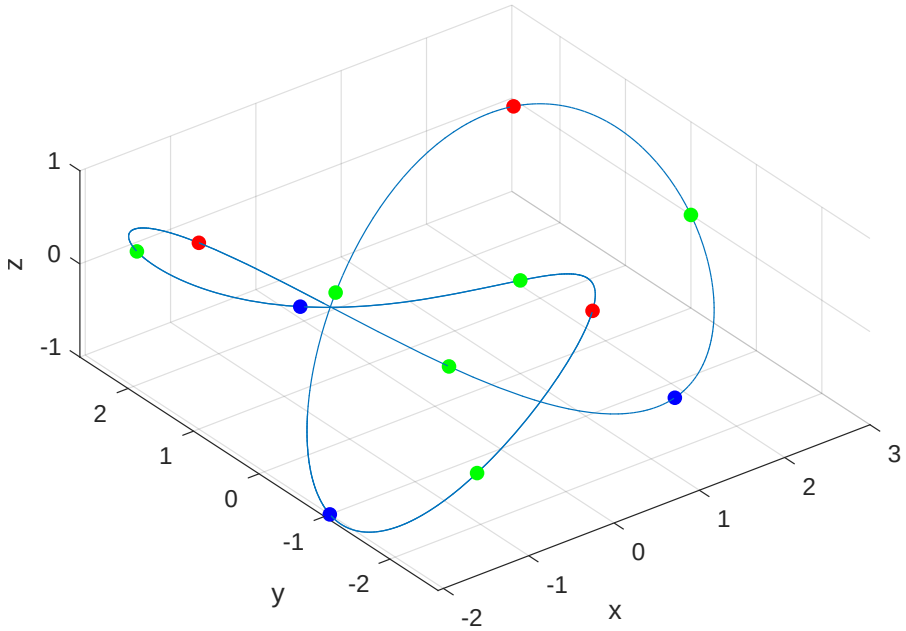


Figure 2.8.: 3D curve with lifted (red) and lowered (blue) points

2.2. Radial expansion

In order to expand the space curve of figure 2.1 into the trefoil object of figure 1.1, we create a tubular surface with a constant radius around the space curve, using the space curve as its center line (figure 2.9).

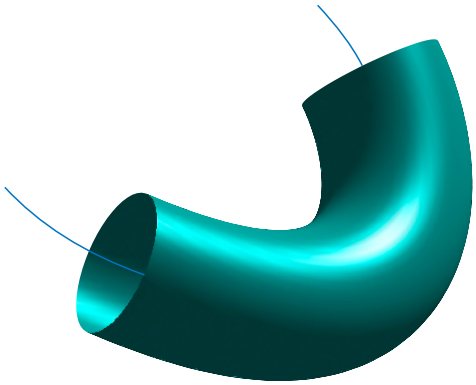
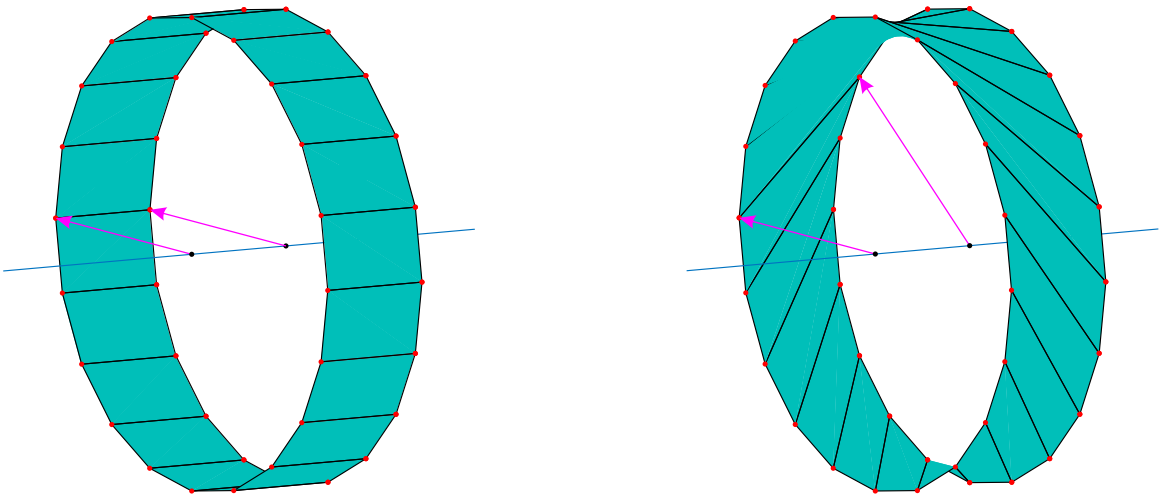


Figure 2.9.: Radial expansion of a space curve into a 3D object

It seems to be a natural choice to construct the tube of figure 2.9 by stringing together cylinders[3] with very small lengths (figure 2.10a).



(a) Undistorted cylinder (b) Distorted cylinder

Figure 2.10.: Related vertices should be as close as possible.

[3]To be precise, the elements of the tube are not really exact cylinders. Because of the curvature of the space curve, the terminal circles in figure 2.10a are not exactly parallel; this is very obvious in figure 2.14. On the other hand, as long as we use enough small tube elements, we can reasonably approximate them as cylinders.

2.2.1. Vertices and faces

In most 3D programs, a surface can be created by defining its vertices and its faces. As indicated in figure 2.10a we define the terminal circles by a sufficient number of vertices (red dots) and each "rectangular" face by four vertices.

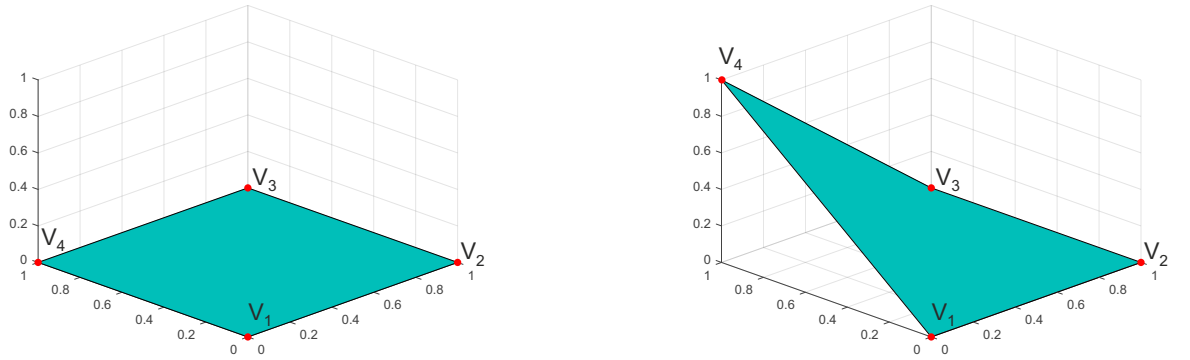
A plane is defined by three points. Therefore, graphic cards expect surfaces to be defined by triangles. Internally, Matlab does this triangulation, but also allows users to define faces with more than three vertices. This makes sense if all vertices are coplanar (i. e. if all vertices lie in one plane)

$$V_1 = \begin{bmatrix}0\\0\\0\end{bmatrix}, \quad V_2 = \begin{bmatrix}1\\0\\0\end{bmatrix}, \quad V_3 = \begin{bmatrix}1\\1\\0\end{bmatrix}, \quad V_4 = \begin{bmatrix}0\\1\\0\end{bmatrix}$$

as illustrated in figure 2.11a, but seems a little odd if not all vertices are coplanar:

$$V_1 = \begin{bmatrix}0\\0\\0\end{bmatrix}, \quad V_2 = \begin{bmatrix}1\\0\\0\end{bmatrix}, \quad V_3 = \begin{bmatrix}1\\1\\0\end{bmatrix}, \quad V_4 = \begin{bmatrix}0\\1\\1\end{bmatrix}$$

Matlab displays a strange three-dimensional "surface" without any crease in that case (figure 2.11b).



(a) Four coplanar vertices create a planar surface.

(b) If the four vertices are not coplanar, Matlab displays a creaseless nonplanar surface.

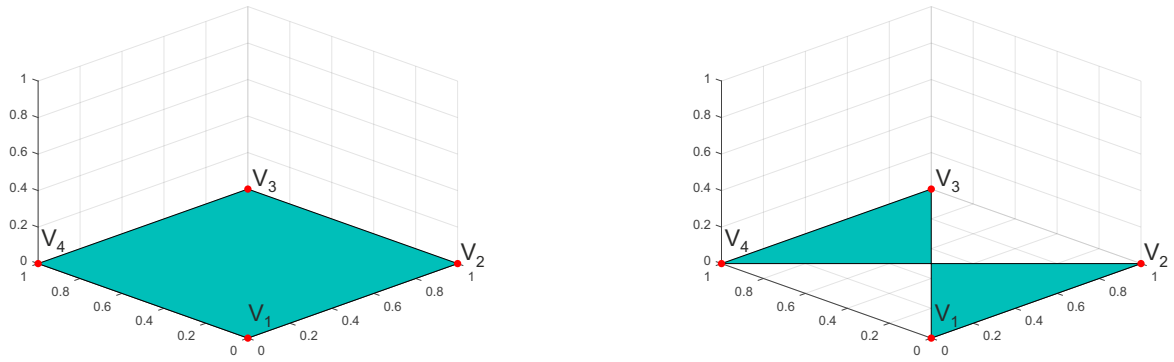
Figure 2.11.: In Matlab we can create planar and nonplanar surfaces.

Other 3D programs use the clockwise or anti-clockwise order of the three vertex indices in the face matrix to define the normal vector of the triangle. By that, a triangle can be visible from one side and invisible from the other side.

Matlab surfaces are automatically double-sided and are therefore always visible from both sides. It does not matter whether we define an anti-clockwise vertex order of

$\begin{bmatrix}1 & 2 & 3 & 4\end{bmatrix}$ or a clockwise vertex order of $\begin{bmatrix}4 & 3 & 2 & 1\end{bmatrix}$ or any other "continuous perimeter" order like $\begin{bmatrix}2 & 3 & 4 & 1\end{bmatrix}$, $\begin{bmatrix}3 & 4 & 1 & 2\end{bmatrix}$, ... They all lead to the correct surface in figure 2.12a.

But if we "jump across diagonals" using vertex indices orders like $\begin{bmatrix}1 & 2 & 4 & 3\end{bmatrix}$, $\begin{bmatrix}4 & 3 & 1 & 2\end{bmatrix}$, ... Matlab comes up with the strange "surface" in figure 2.12b.



(a) Correct vertex order $\begin{bmatrix}1 & 2 & 3 & 4\end{bmatrix}$ (b) Incorrect vertex order $\begin{bmatrix}1 & 2 & 4 & 3\end{bmatrix}$

Figure 2.12.: The vertex order in the face matrix is important.

By defining the four vertices that make up a "rectangle" in figure 2.10, we run into another problem: If we use arbitrary corresponding vertices to define the faces, we might end up with the "twisted parallelograms" of figure 2.10b and the "cylinder" becomes distorted. Therefore, it seems to be very important to chose the corresponding vertices as close to each other as possible. While this could be achieved by actually looking for the nearest partner vertex in the corresponding terminal circle, we will use a more general way in section 2.2.3.

2.2.2. Circle in parameter representation

We can create a unit circle around the origin by defining two orthonormal basis vectors $\boldsymbol{N}$ and $\boldsymbol{B}$ and use the angle θ as the parameter in a parametric circle representation (figure 2.13):

$$\boldsymbol{R} = \boldsymbol{N}\cos\theta + \boldsymbol{B}\sin\theta \tag{2.4}$$

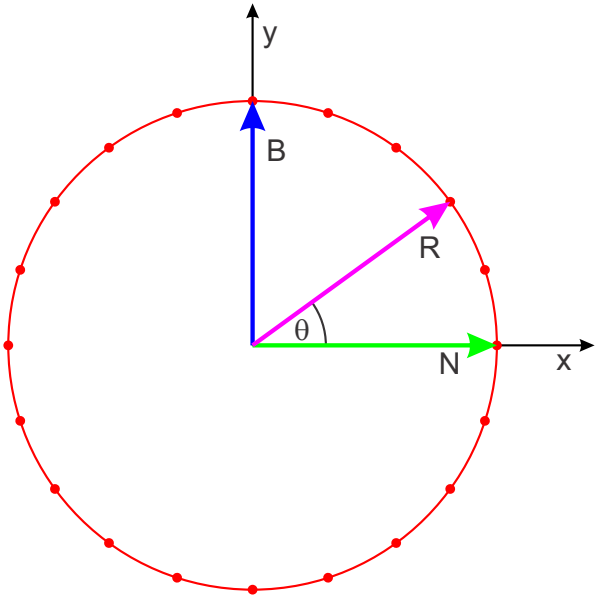


Figure 2.13.: Unit circle in parametric representation

To reach single vertices (red dots in figure 2.13) the parameter θ has to cycle through distinct values

$$\theta_i = \frac{2\pi}{n} i$$

where the integer index i with $\{i \in \mathbb{N}_0 \,|\, 0 \leq i < n\}$ addresses a single vertex and n is the number of vertices on the circle.

2.2.3. Moving trihedron

As displayed in figure 2.10, we have to make sure that the basis vectors defining two consecutive circles only change minimally.

A moving trihedron or moving frame provides a set of three orthogonal unit vectors,

- the tangent vector $\boldsymbol{T}$, representing the current direction of the curve,
- the normal vector $\boldsymbol{N}$, pointing towards the center of the current circle of curvature,
- the binormal vector $\boldsymbol{B}$, orthogonal to $\boldsymbol{T}$ and $\boldsymbol{N}$,

for every point of a space curve. As illustrated in figure 2.14, we can use $\boldsymbol{N}$ and $\boldsymbol{B}$ as orthonormal basis vectors to create unit circles containing the vertices of the tube around the curve.

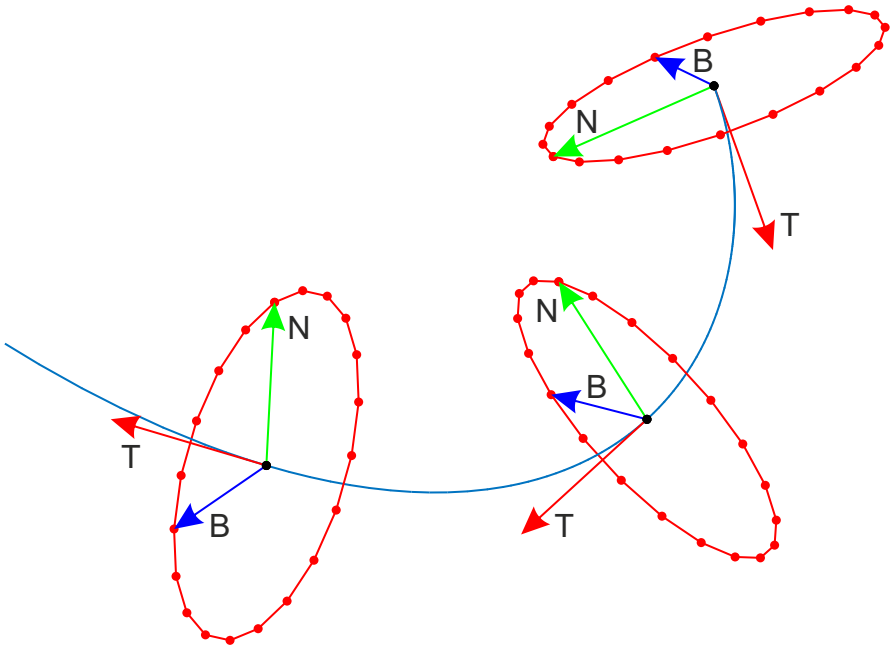


Figure 2.14.: Moving trihedron

In order to compute the trihedron, we define the following vectors:

$\boldsymbol{r}(t)$ is the position vector from the origin to a point on the space curve (figure 2.15).

$\dot{\boldsymbol{r}}(t)$ is the derivative of $\boldsymbol{r}(t)$ with respect to the parameter t:

$$\dot{\boldsymbol{r}}(t) = \frac{\mathrm{d}\boldsymbol{r}(t)}{\mathrm{d}t}$$

$\ddot{\boldsymbol{r}}(t)$ is the derivative[4] of $\dot{\boldsymbol{r}}(t)$ with respect to t:

$$\ddot{\boldsymbol{r}}(t) = \frac{\mathrm{d}\dot{\boldsymbol{r}}(t)}{\mathrm{d}t}$$

According to the Frenet-Serret formulas [3], the tangent vector $\boldsymbol{T}$ is just the normalized first derivative:

$$\boldsymbol{T} = \frac{\dot{\boldsymbol{r}}}{|\dot{\boldsymbol{r}}|} \tag{2.5}$$

The binormal vector $\boldsymbol{B}$ is the normalized cross product[5] of the first and the second derivative:

$$\boldsymbol{B} = \frac{\dot{\boldsymbol{r}} \times \ddot{\boldsymbol{r}}}{|\dot{\boldsymbol{r}} \times \ddot{\boldsymbol{r}}|} \tag{2.6}$$

[4] If we interpret the parameter t as time, $\dot{\boldsymbol{r}}(t)$ is the velocity and $\ddot{\boldsymbol{r}}(t)$ is the acceleration of the moving point $\boldsymbol{r}(t)$.

[5] Since the cross product of two vectors creates a new vector orthogonal to both vectors, $\boldsymbol{B}$ is orthogonal to both $\dot{\boldsymbol{r}}$ and $\ddot{\boldsymbol{r}}$, and since $\boldsymbol{T}$ is collinear to $\dot{\boldsymbol{r}}$ according to equation (2.5), $\boldsymbol{B}$ is also orthogonal to $\boldsymbol{T}$.

Finally, the normal vector $\boldsymbol{N}$ is the cross product[6] of the binormal vector and the tangent vector:

$$\boldsymbol{N} = \boldsymbol{B} \times \boldsymbol{T} \tag{2.7}$$

Using $\boldsymbol{N}$ and $\boldsymbol{B}$ as an orthonormal basis in figure 2.15, we can now define the position vector $\boldsymbol{r_c}(\theta)$ of a point on the unit circle via

$$\boldsymbol{r_c}(\theta) = \boldsymbol{r} + \boldsymbol{N}\cos\theta + \boldsymbol{B}\sin\theta \tag{2.8}$$

where $\boldsymbol{r}$ is the position vector of a point on the space curve defining the center of the circle and the angle θ is the parameter in the parameter representation of the circle (equation (2.4)).

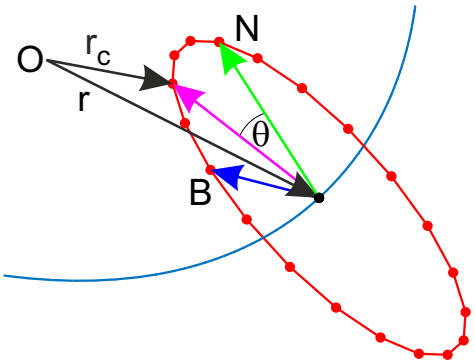


Figure 2.15.: Circle around a point of the space curve

[6] We do not have to normalize $\boldsymbol{N}$ because it equals the cross product of two orthogonal unit vectors $\boldsymbol{B}$ and $\boldsymbol{T}$. Since the norm $|\boldsymbol{N}|$ of the resulting cross product vector equals the area of the parallelogram set up by $\boldsymbol{B}$ and $\boldsymbol{T}$, and the parallelogram degenerates to a square with a side length of 1 in this case, the area of the square and thus the norm of the normal vector $\boldsymbol{N}$ equal 1 too. Note that the cross product of two unit vectors results in another unit vector only if both unit vectors are orthogonal to each other.

3. Matlab implementation

3.1. `trefoil_knot.m`

`trefoil_knot` is the main program that computes and displays the trefoil knot.

3.1.1. Analytical description of the space curve

While other tube generating functions take the points of a space curve and approximate the first and second derivatives numerically, we live in the luxury of having a mathematical description of the space curve (equation (2.1)) and can therefore use Matlab's Symbolic Math Toolbox to compute the derivatives analytically.

We declare a symbolic parameter

```
syms t
```

define the three components of the space curve according to equation (2.1)

```
x = cos (t) + 2 * cos (2 * t);
y = sin (t) - 2 * sin (2 * t);
z = -sin (3 * t);
```

and create the 3D position vector:

```
r = [x y z];
```

The Symbolic Math Toolbox makes it very easy to compute the first derivative (`dr`)

```
dr = diff (r);
```

$$\dot{\boldsymbol{r}}(t) = \begin{bmatrix} -\sin(t) + 4\sin(2t) \\ \cos(t) - 4\cos(2t) \\ -3\cos(3t) \end{bmatrix}$$

and the second derivative (`ddr`) of the symbolic position vector analytically:

```
ddr = diff (dr);
```

$$\ddot{\boldsymbol{r}}(t) = \begin{bmatrix} -\cos(t) - 8\cos(2t) \\ -\sin(t) + 8\sin(2t) \\ 9\sin(3t) \end{bmatrix}$$

3.1.2. Initialization

We define the number of circles (i. e. the number[1] of cylinders in the tube)

```
n_circles = 200;
```

the number of vertices per circle (figure 2.13)

```
n_vertices_per_circle = 50;
```

and the radius[2] of the tube:

```
radius = 0.6;
```

Using the number of circles, we define a vector of `n_circles` evenly spaced parameter values (`tt`) in the domain $t = 0 \dots 2\pi$

```
tt = linspace (0, 2 * pi, n_circles);
```

and compute the corresponding vectors of the position

```
rr = double (subs (r, t, tt'));
```

the first derivative

```
drr = double (subs (dr, t, tt'));
```

and the second derivative

```
ddrr = double (subs (ddr, t, tt'));
```

of the space curve at these points. From this point on, we will use numerical computations. Therefore, the `double` commands convert the symbolic vectors to numeric vectors.

In the next step, we have to define the vertices and faces of the tube according to section 2.2.1. We compute the total number of vertices from the number of circles and the number of vertices per circle

```
n_vertices = n_circles * n_vertices_per_circle;
```

and initialize the vertex matrix[3] with zeros:

```
vertices = zeros (n_vertices, 3);
```

The number of faces (figure 2.10a) takes into account that the first and the last circle are identical and that the first and the last vertex of a circle are identical:

```
n_faces = (n_circles - 1) * (n_vertices_per_circle - 1);
```

[1] Since the first ($t = 0$) and the last circle ($t = 2\pi$) are identical, there is one circle more than there are cylinders.

[2] The arbitrarily chosen radius of 0.6 leaves a bit of open space around the tube (figure 1.1).

[3] The vertex matrix has three columns for the x-, y- and z-components of each vertex.

For the initialization of the face matrix we consider the fact that each "rectangular" face consists of four vertices (figure 2.10a):

```
faces = zeros (n_faces, 4);
```

3.1.3. Vertex matrix creation

In a loop over all circles

```
for i_circle = 1 : n_circles
```

we call the function `circle3` (section 3.2) with the position (`rr`), the first (`drr`), and the second derivative (`ddrr`) of the current space curve point in order to compute the vertex matrix (`c`) of the current circle according to section 2.2.3:

```
  c = circle3 ( ...
    rr(i_circle, :), ...
    drr(i_circle, :)', ...
    ddrr(i_circle, :)', ...
    radius, n_vertices_per_circle);
```

Every *column* of the circle vertex matrix represents one vertex of the circle:

$$c = \begin{bmatrix} x_1 & x_2 & \cdots & x_{50} \\ y_1 & y_2 & \cdots & y_{50} \\ z_1 & z_2 & \cdots & z_{50} \end{bmatrix} \tag{3.1}$$

Matlab's general vertex matrix uses one vertex per *row*:

$$vertices = \begin{bmatrix} c_1^T \\ \hline\hline\hline c_2^T \\ \hline\hline\hline \vdots \\ \hline\hline\hline c_{200}^T \end{bmatrix} = \left[\begin{array}{ccc} c_{1x_1} & c_{1y_1} & c_{1z_1} \\ c_{1x_2} & c_{1y_2} & c_{1z_2} \\ \vdots & \vdots & \vdots \\ c_{1x_{50}} & c_{1y_{50}} & c_{1z_{50}} \\ \hline\hline\hline c_{2x_1} & c_{2y_1} & c_{2z_1} \\ c_{2x_2} & c_{2y_2} & c_{2z_2} \\ \vdots & \vdots & \vdots \\ c_{2x_{50}} & c_{2y_{50}} & c_{2z_{50}} \\ \hline\hline\hline \vdots & \vdots & \vdots \\ \hline\hline\hline c_{200x_1} & c_{200y_1} & c_{200z_1} \\ c_{200x_2} & c_{200y_2} & c_{200z_2} \\ \vdots & \vdots & \vdots \\ c_{200x_{50}} & c_{200y_{50}} & c_{200z_{50}} \end{array}\right] \tag{3.2}$$

In order to concatenate the vertices of all circles into one large matrix as in equation (3.2), we could use a simple concatenation statement like

```
vertices = [vertices; c'];
```

inside the loop together with a an empty initialization before the loop, but we have all internalized that it is bad practice to let arrays grow inside a loop without proper[4] memory preallocation. Instead, we copy each (transposed) circle into the corresponding sub-matrix of the vertex matrix directly by using appropriate indices:

```
  vertices( ...
    (i_circle - 1) * n_vertices_per_circle + 1 : ...
    i_circle * n_vertices_per_circle, :) = c';

end
```

3.1.4. Face matrix creation

Each row in Matlab's face matrix (equation (3.3)) defines the indices of the four vertices that make up one "rectangular" patch (section 2.2.1):

$$faces = \begin{bmatrix} f_{1v_1} & f_{1v_2} & f_{1v_3} & f_{1v_4} \\ f_{2v_1} & f_{2v_2} & f_{2v_3} & f_{2v_4} \\ \vdots & \vdots & \vdots & \vdots \\ f_{10000v_1} & f_{10000v_2} & f_{10000v_3} & f_{10000v_4} \end{bmatrix} \tag{3.3}$$

We initialize a face index counter

```
i_face = 1;
```

and start an outer loop over all circles and an inner loop over all vertices of the current circle:

```
for i_circle = 1 : n_circles - 1

  for i_vertex = 1 : n_vertices_per_circle - 1
```

Addressing the four vertex indices of a single face is a bit tricky since the first two vertices are part of one circle (`i_circle - 1`) and the last two vertices are part of another circle (`i_circle`) according to figure 2.10a.

Additionally, in order to avoid the problem of figure 2.12b, we have to take the correct order of the vertex indices into account. Therefore, the second and the third vertex in the following lines of code both use a `+ 1` index:

[4] Is there a proper way to preallocate memory for an array of known final size if the array grows inside a loop by concatenation?

```
    faces (i_face, :) = [ ...
      i_vertex + (i_circle - 1) * n_vertices_per_circle, ...
      i_vertex + (i_circle - 1) * n_vertices_per_circle + 1, ...
      i_vertex + (i_circle) * n_vertices_per_circle + 1, ...
      i_vertex + (i_circle) * n_vertices_per_circle, ...
      ];
```

Finally, we increment the face index counter inside the inner loop:

```
    i_face = i_face + 1;

  end

end
```

3.1.5. Patch me up

We are now ready to create the graphical representation of the trefoil knot of figure 1.1. We open a new figure

```
figure
```

and use the matrix of `vertices` from section 3.1.3 and the matrix of `faces` from section 3.1.4 to define the surface of the knot:

```
patch ( ...
  Faces = faces, ...
  Vertices = vertices, ...
  FaceColor = [0.0638, 0.7446, 0.7292], ...
  EdgeColor = 'none', ...
  );
```

The chosen `FaceColor` is the a nice dark cyan (actually, it is the 128^{th} element of the `parula colormap`). If we omitted `EdgeColor = 'none'`, we would see "rectangular" edges[5] as in figure 3.1.

[5] If we look very closely at figure 3.1 we can discover that Matlab indeed does a (non-perfect) triangulation of the "rectangular" patches if we ask it to export the figure into a pdf-document via `exportgraphics (gcf, 'test.pdf', 'ContentType', 'vector')`. In the original Matlab figure, the triangulation is not visible.

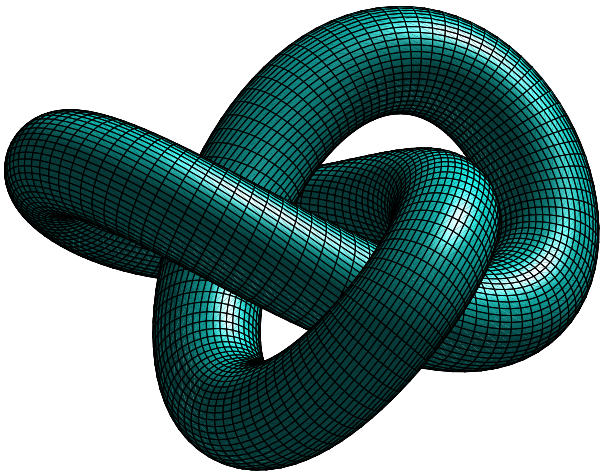


Figure 3.1.: Patch with edges (and visible triangulation)

Switching on a light

```
light
```

and asking Matlab to vary the light across the faces by using the recommended `gourand` lighting method for curved surfaces

```
lighting gouraud
```

makes the knot look a bit more photorealistic. We scale all axis equally

```
axis equal
```

switch off the axis system

```
axis off
```

use Matlab's standard view for three-dimensional objects

```
view (3)
```

and finally arrive at the smooth trefoil knot display of figure 1.1.

3.2. `circle3.m`

The `circle3` function

```
function circle_vertices = circle3 ( ...
  center, ...
  first_derivative, ...
  second_derivative, ...
  radius, ...
  n_vertices)
```

is called for every point of the space curve in section 3.1.3 in order to create a unit circle around the point as described in section 2.2.3. We supply the function with the current point (`center`), the current tangent (`first_derivative`), the current `second_derivative` pointing towards the center of the current circle of curvature, the `radius` of the circle to be created, and the number of vertices (`n_vertices`) of the polygonal chain approximating the circle.

For the sake of flexibility, we have outsource the computation of the normal vector and the binormal vector from the first and second derivative at the current point into the function `tnb` described in section 3.3:

```
[~, normal, binormal] = tnb ( ...
  first_derivative(:), ...
  second_derivative(:));
```

`tnb` returns the `t`angent, `n`ormal, and `b`inormal vectors of which we only use the normal and the binormal.

We define a vector holding the linearly spaced angles of the circle vertices

```
th = linspace (0, 2 * pi, n_vertices);
```

and compute the vertices of the circle according to equation (2.8) in a very compact and elegant way:

```
circle_vertices = ...
    center(:) + ...
    radius * (normal * cos (th) + binormal * sin (th));

end
```

The fact that the lines above actually work at all, is another sign of Matlab's Marvelous Matrix Manipulation Magic: Since `circle_vertices` has to be a $3 \times$ `n_vertices` matrix according to equation (3.1), we compute the outer (or dyadic) product of the column vector `normal` and the row vector `cos (th)` via a common matrix matrix product, which is nothing too special. But then, we add a column **vector** `center(:)` and a **matrix**, which definitely is quite strange from a mathematical point of view.

To understand why this vector matrix sum does not result in a dimensions mismatch error, we have to know that in R2016b, MathWorks expanded Matlab's arithmetic [4]. We can now e. g. add a column vector a and a row vector b:

```
syms a1 a2 b1 b2

a = [a1; a_2]
```

$$a = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}$$

```
b = [b1, b2]
```

$b = \begin{pmatrix} b_1 & b_2 \end{pmatrix}$

```
M = a + b
```

$$M = \begin{pmatrix} a_1 + b_1 & a_1 + b_2 \\ a_2 + b_1 & a_2 + b_2 \end{pmatrix}$$

This mathematically unusual "outer" (or "dyadic") sum results in a matrix M, the elements M_{ij} of which are the sums of the corresponding vector elements:

$$M_{ij} = a_i + b_j$$

3.3. tnb.m

As described in section 3.2, the `tnb` function

```
function [tangent, normal, binormal] = tnb (r_dot, r_dot_dot)
```

is called for each point of the space curve and computes the `tangent`, the `normal`, and the `binormal` vector of the curve at the current point from the first derivative (`r_dot`) and the second derivative (`r_dot_dot`) vector.

According to equation (2.5), the tangent vector is just the normalized first derivative:

```
tangent = r_dot / norm (r_dot);
```

As stated by equation (2.6), the binormal vector is the normalized cross product of the first and the second derivative:

```
binormal = cross (r_dot, r_dot_dot);
binormal = binormal / norm (binormal);
```

In accordance with equation (2.7), the normal vector is the cross product of the binormal vector and the tangent vector:

```
normal = cross (binormal, tangent);

end
```

Bibliography

[1] MathWorks. (2022) Matlab. [Online]. Available: https://de.mathworks.com/products/matlab.html

[2] Wikipedia. (2022) Trefoil knot. [Online]. Available: https://en.wikipedia.org/wiki/Trefoil_knot

[3] ——. (2022) Frenet-Serret formulas. [Online]. Available: https://en.wikipedia.org/wiki/Frenet-Serret_formulas

[4] L. Shure. (2016) MATLAB arithmetic expands in R2016. MathWorks. [Online]. Available: https://blogs.mathworks.com/loren/2016/10/24/matlab-arithmetic-expands-in-r2016b/

A. Null space

In section 2.2.3, we describe the creation of the moving trihedron from the first and the second derivative at each point of the space curve. In this appendix, we want to show why we cannot utilize Matlab's `null` command to compute the normal and the binormal vector from the tangent vector.

Matlab states:

> Use the `null` function to calculate orthonormal and rational basis vectors for the null space of a matrix. The null space of a matrix contains vectors $\boldsymbol{x}$ that satisfy $\boldsymbol{Ax} = \boldsymbol{0}$.

If we define an arbitrary vector `n`

```
n = [1 2 3]
```

we can compute its null space:

```
nn = null (n)

nn =

   -0.5345   -0.8018
    0.7745   -0.3382
   -0.3382    0.4927
```

The column vectors `n1` and `n2` of the null space

```
n1 = nn(:, 1)

n1 =

   -0.5345
    0.7745
   -0.3382
```

```
n2 = nn(:, 2)

n2 =

   -0.8018
   -0.3382
    0.4927
```

are unit vectors

```
norm (n1)

ans =

     1
```

```
norm (n2)

ans =

     1
```

and are orthogonal to each other and to the original vector:

```
dot (n1, n2)

ans =

  -1.6653e-16

dot (n, n1)

ans =

   3.3307e-16

dot (n, n2)

ans =

   2.2204e-16
```

These properties should qualify the null space vectors of the tangent vector to be used as normal and binormal vectors instead of the method of section 2.2.3 and section 3.3. And indeed, we can use the null space vectors for parts of the trefoil knot, but if we want to display a whole knot at once, we end up with figure A.1.

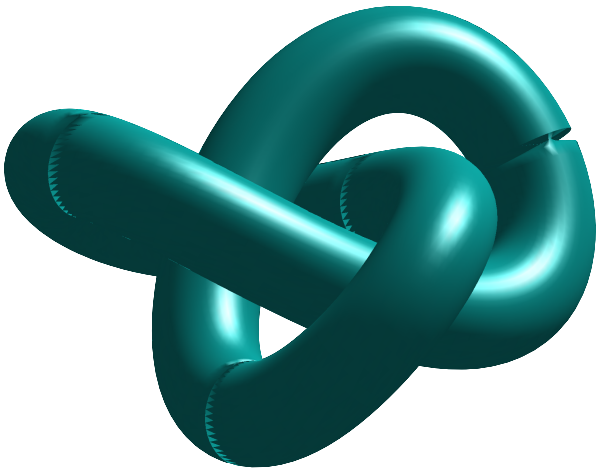


Figure A.1.: Trefoil knot using null space to compute normal and binormal vectors

There seem to be points of discontinuity at $t = k \cdot \frac{\pi}{2}$, $k \in \mathbb{Z}$, where the null space vectors "jump" and we run into the distorted cylinder problem demonstrated in figure 2.10b.

In detail[1] we can show that the null spaces for $t = +0$

```
null (double (subs (dr, t, eps)))

ans =

   -0.7071   -0.7071
    0.5000   -0.5000
   -0.5000    0.5000
```

and $t = -0$

```
null (double (subs (dr, t, -eps)))

ans =

    0.7071    0.7071
    0.5000   -0.5000
   -0.5000    0.5000
```

are similar but not identical at all.

If we plot the null space vectors for a few small negative and small positive t-values (figure A.2)

[1]Note, that `eps` is the *floating-point relative accuracy* in Matlab, which is a "very small number" ($\epsilon = 2.2204 \cdot 10^{-16}$).

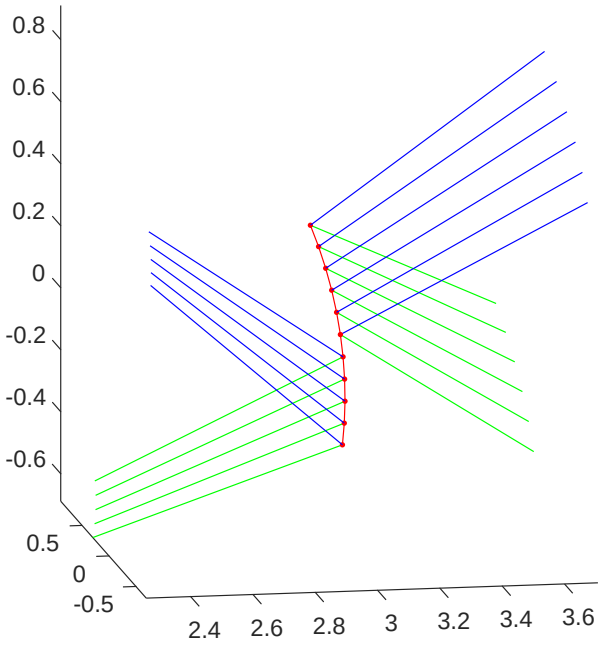


Figure A.2.: Null space vectors for small negative and small positive parameter values

it becomes obvious that the null space vectors show discontinuities at certain parameter values and that we cannot use them as continuous normal and binormal vectors for that reason.