Differential Geometry

Lecture Notes

Dr. Silvio Fanzon

20 Sep 2023

Table of contents

Welcome

These are the Lecture Notes of **Differential Geometry 661955** for T1 2023/24 at the University of Hull. We will study curves and surfaces in $\mathbb{R}^3$. I will follow these lecture notes during the course. If you have any question or find any typo, please email me at

S.Fanzon@hull.ac.uk

Up to date information about the course, Tutorials and Homework will be published on the University of Hull **Canvas Website**

canvas.hull.ac.uk/courses/67594

and on the **Course Webpage** hosted on my website

silviofanzon.com/blog/2023/Differential-Geometry

Readings

The main textbook of the course is Pressley [5]. Other interesting readings are the books by do Carmo [2] and Bär [1]. I will assume some knowledge from Analysis and Linear Algebra. A good place to revise these topics are the books by Zorich [6, 7].

Visualization

It is important to visualize the geometrical objects and concepts we are going to talk about in this course. I will show basic Python code to plot curves and surfaces. This part of the course is **`not required`** for the final examination. If you want to have fun plotting with Pyhton, I recommend installation through Anaconda or Miniconda. The actual coding can then be done through Jupyter Notebook. Good references for scientific Python programming are [3, 4].
If you do not want to mess around with Python, you can still visualize pretty much everything we will do in this course using the excellent online 3D grapher tool CalcPlot3D. To understand how it works, please refer to the help manual or to the short video introduction.

! You are not expected to purchase any of the above books. These lecture notes will cover 100% of the topics you are expected to known in order to excel in the final exam.

1 Curves

We all have in mind examples of curves. These are, intuitively speaking, 1D objects in the 2D or 3D space. For example in two dimensions one could think of a straight line, a hyperbole or a circle. These can be all described by an equation in the x and y coordinates: respectively

$$y = 2x + 1\,, \quad y = e^x\,, \quad x^2 + y^2 = 1\,.$$

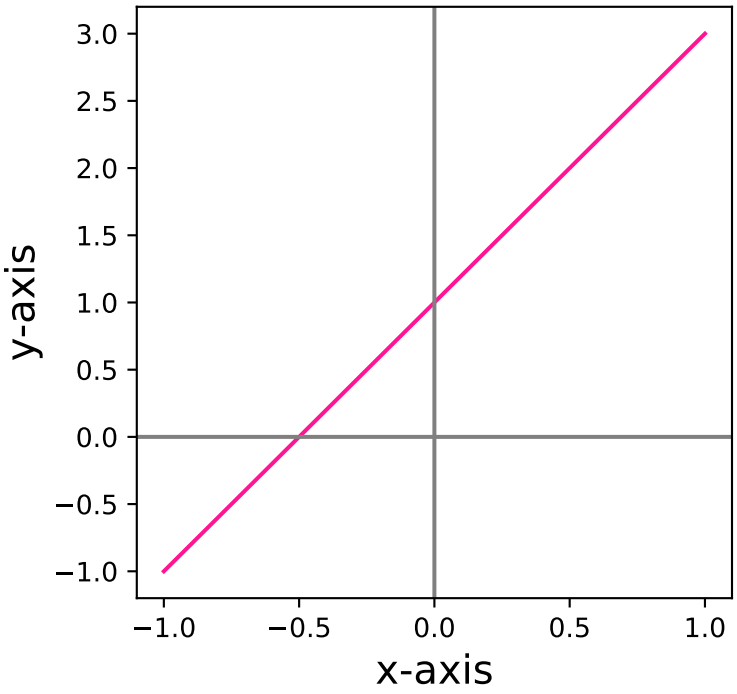


Figure 1.1: Plotting straight line $y = 2x + 1$

Goal

The aim of this course is to study curves by differentiating them.

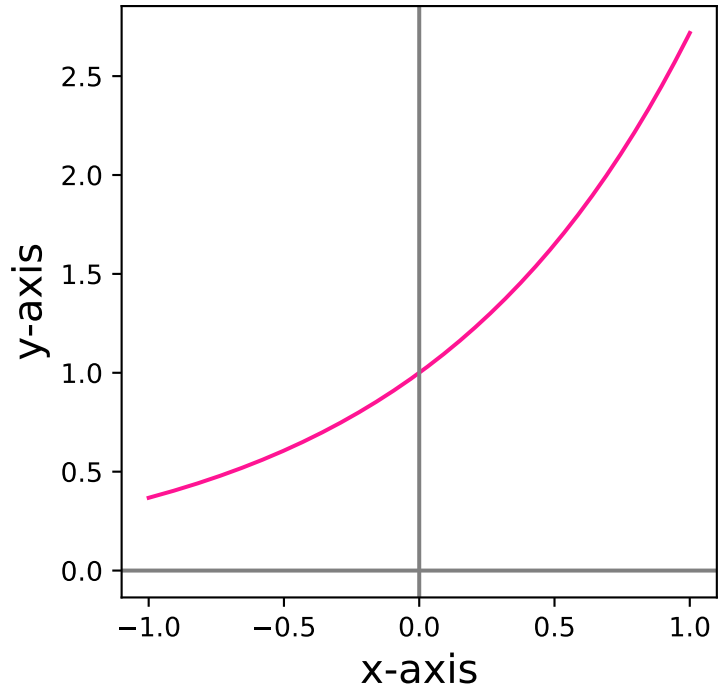


Figure 1.2: Plot of hyperbole $y = e^x$

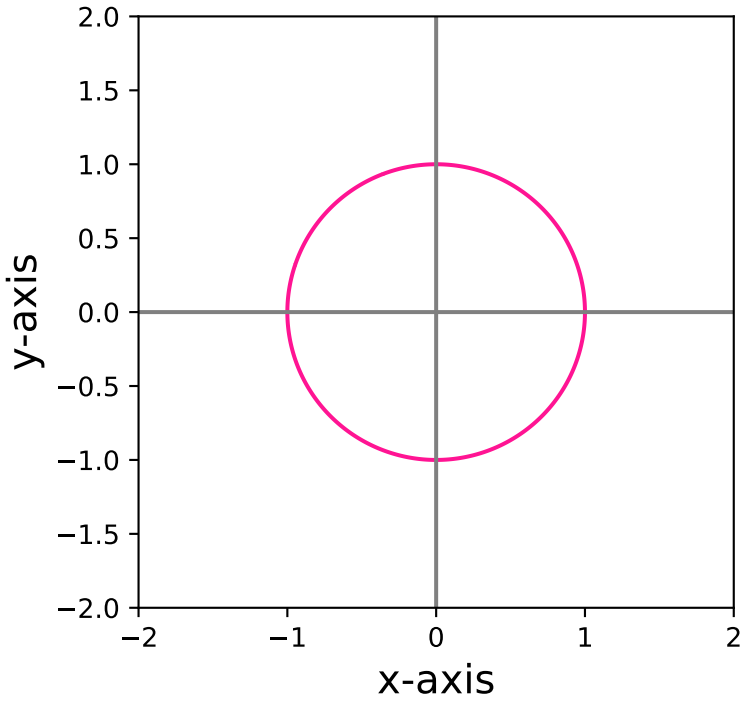


Figure 1.3: Plot of unit circle of equation $x^2 + y^2 = 1$

Question

In what sense do we differentiate the above curves?

It is clear that we need a way to mathematically describe the curves. One way of doing it is by means of Cartesian equations. This means that the curve is described as the set of points $(x, y) \in \mathbb{R}^2$ where the equation

$$f(x, y) = c\,,$$

is satisfied, where

$$f : \mathbb{R}^2 \to \mathbb{R}\,.$$

is some given function, and

$$c \in \mathbb{R}$$

some given value. In other words, the curve is identified with the subset of $\mathbb{R}^2$ given by

$$C = \{(x, y) \in \mathbb{R}^2 : \ f(x, y) = c\}\,.$$

For example, in the case of the straight line, we would have

$$f(x, y) = y - 2x\,, \quad c = 1\,.$$

while for the circle

$$f(x, y) = x^2 + y^2\ , c = 1\,.$$

But what about for example a helix in 3 dimensions? It would be more difficult to find an equation of the form

$$f(x, y, z) = 0$$

to describe such object.

Problem

We need a unified way to describe curves.

1.1 Parametrized curves

Rather than Cartesian equations, a more useful way of thinking about curves is viewing them as the *path traced out by a moving point*. If $\gamma(t)$ represents the position a point in $\mathbb{R}^n$ at time t, the whole curve can be identified by the function

$$\gamma : \mathbb{R} \to \mathbb{R}^n\,, \ \ \gamma = \gamma(t)\,.$$

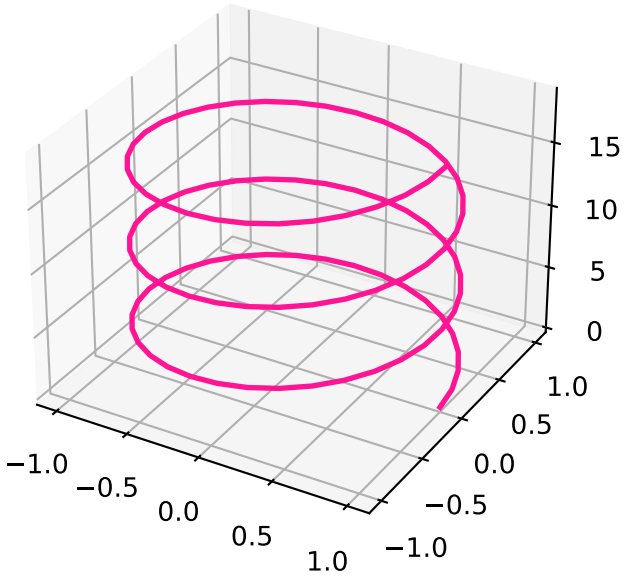


Figure 1.4: Plot of a 3D Helix

This motivates the following definition of **parametrized curve**, which will be our **main** definition of curve.

Definition 1.1: Parametrized curve

A **parametrized curve** in $\mathbb{R}^n$ is a function

$$\gamma : (a, b) \to \mathbb{R}^n .$$

where

$$-\infty \leq a < b \leq \infty .$$

A few remarks:

- The symbol (a, b) denotes an **open** interval
$$(a, b) = \{t \in \mathbb{R} : \ a < t < b\} .$$
- The requirement that
$$-\infty \leq a < b \leq \infty$$
means that the interval (a, b) is possibly unbounded.
- For each $t \in (a, b)$ the quantity $\gamma(t)$ is a vector in $\mathbb{R}^n$.

- The **components** of $\gamma(t)$ are denoted by

$$\gamma(t) = (\gamma_1(t), \dots, \gamma_n(t)),$$

where the components are functions

$$\gamma_i : (a, b) \to \mathbb{R},$$

for all $i = 1, \dots, n$.

1.2 Parametrizing cartesian curves

At the start we said that examples of curves in $\mathbb{R}^2$ were the straight line, the hyperbole and the circle, with equations

$$y = 2x + 1, \quad y = e^x, \quad x^2 + y^2 = 1.$$

We saw that these can be represented by Cartesian equations

$$f(x, y) = c$$

for some function $f : \mathbb{R}^2 \to \mathbb{R}$ and value $c \in \mathbb{R}$. Curves that can be represented in this way are called **level curves**. Let us give a precise definition.

Definition 1.2: Level curve

A **level curve** in $\mathbb{R}^n$ is a set $C \subset \mathbb{R}^n$ which can be described as

$$C = \{(x_1, \dots, x_n) \in \mathbb{R}^n : f(x_1, \dots, x_n) = c\}$$

for some given function

$$f : \mathbb{R}^n \to \mathbb{R}$$

and value

$$c \in \mathbb{R}.$$

We now want to represent level curves by means of parametrizations.

Definition 1.3

Suppose given a level curve $C \subset \mathbb{R}^n$. We say that a curve

$$\gamma : (a, b) \to \mathbb{R}^n$$

parametrizes C if

$$C = \{(\gamma_1(t), \dots, \gamma_n(t)) \ : \ t \in (a,b)\}\,.$$

Question

Can we **represent** the level curves we saw above by means of a parametrization γ?

The answer is YES, as shown in the following examples.

Example: Parametrizing the straight line

The straight line

$$y = 2x + 1$$

is a **level curve** with

$$C = \{(x,y) \in \mathbb{R}^2 \ : \ f(x,y) = c\}\,,$$

where

$$f(x,y) := y - 2x\,, \quad c := 1\,.$$

How do we represent C as a **parametrized curve** γ? We know that the curve is 2D, therefore we need to find a function

$$\gamma \ : (a,b) \to \mathbb{R}^2$$

with componenets

$$\gamma(t) = (\gamma_1(t), \gamma_2(t))\,.$$

The curve γ needs to be chosen so that it parametrizes the set C, in the sense that

$$C = \{(\gamma_1(t), \gamma_2(t)) \ : \ t \in (a,b)\}\,. \tag{1.1}$$

Thus we need to have

$$(x,y) = (\gamma_1, \gamma_2)\,. \tag{1.2}$$

How do we define such γ? Note that the points (x,y) in C satisfy

$$(x,y) \in C \iff y = 2x + 1\,.$$

Therefore, using (1.2), we have that

$$\gamma_1 = x\,, \quad \gamma_2 = y = 2x + 1$$

from which we deduce that γ must satisfy

$$\gamma_2(t) = 2\gamma_1(t) + 1 \tag{1.3}$$

for all $t \in (a, b)$. We can then choose

$$\gamma_1(t) := t\,,$$

and from (1.3) we deduce that

$$\gamma_2(t) = 2t + 1\,.$$

This choice of γ works:

$$\begin{aligned} C &= \{(x, 2x+1) \,:\, x \in \mathbb{R}\} && (1.4)\\ &= \{(t, 2t+1) \,:\, -\infty < t < \infty\} && (1.5)\\ &= \{(\gamma_1(t), \gamma_2(t)) \,:\, -\infty < t < \infty\}\,, && (1.6) \end{aligned}$$

where in the second line we just swapped the symbol x with the symbol t. In this case we have to choose the time interval as

$$(a, b) = (-\infty, \infty)\,.$$

In this way γ satisfies (1.1) and we have successfully parametrized the straight line C.

Remark 1.4: Parametrization is not unique

Let us consider again the straight line

$$C = \{(x, y) \in \mathbb{R}^2 \,:\, 2x + 1 = y\}\,.$$

We saw that $\gamma\colon (-\infty, \infty) \to \mathbb{R}^2$ defined by

$$\gamma(t) := (t, 2t + 1)$$

is a parametrization of C. But of course any γ satisfying

$$\gamma_2(t) = 2\gamma_1(t) + 1$$

would yield a parametrization of C. For example one could choose

$$\gamma_1(t) = 2t\,, \quad \gamma_2(t) = 2\gamma_1(t) + 1 = 4t + 1\,.$$

In general, any time rescaling would work: the curve γ defined by

$$\gamma_1(t) = nt\,, \quad \gamma_2(t) = 2\gamma_1(t) + 1 = 2nt + 1$$

parametrizes C for all $n \in \mathbb{N}$. Hence there are **infinitely many** parametrizations of C.

Example: Parametrizing the circle

The circle C is described by all the points $(x, y) \in \mathbb{R}^2$ such that

$$x^2 + y^2 = 1\,.$$

Therefore if we want to find a curve

$$\gamma = (\gamma_1, \gamma_2)$$

which parametrizes C, this has to satisfy

$$\gamma_1(t)^2 + \gamma_2(t)^2 = 1 \tag{1.7}$$

for all $t \in (a, b)$.
How to find such curve? We could proceed as in the previous example, and set

$$\gamma_1(t) := t\,.$$

Then (1.7) implies

$$\gamma_2(t) = \sqrt{1 - t^2}\,,$$

from which we also deduce that

$$-1 \leq t \leq 1$$

are the only admissible values of t. However this curve does not represent the full circle C, but only the upper half, as seen in the plot below.
Simlarly, another solution to (1.7) would be γ with

$$\gamma_1(t) = t\,, \quad \gamma_2(t) = -\sqrt{1 - t^2}\,,$$

for $t \in [-1, 1]$. However this choice does not parametrize the full circle C either, but only the bottom half, as seen in the plot below.
How to represent the whole circle? Recall the trigonometric identity

$$\cos(t)^2 + \sin(t)^2 = 1$$

for all $t \in \mathbb{R}$. This suggests to choose γ as

$$\gamma_1(t) := \cos(t)\,, \quad \gamma_2(t) := \sin(t)$$

for $t \in [0, 2\pi)$. This way γ satisfies (1.7), and actually parametrizes C, as shown below. Note the following:

- If we had chosen $t \in [0, 4\pi]$ then γ would have covered C twice.
- If we had chosen $t \in [0, \pi]$, then γ would have covered the upper semi-circle
- If we had chosen $t \in [\pi, 2\pi]$, then γ would have covered the lower semi-circle
- Similarly, we can choose $t \in [\pi/6, \pi/2]$ to cover just a portion of C, as shown below.

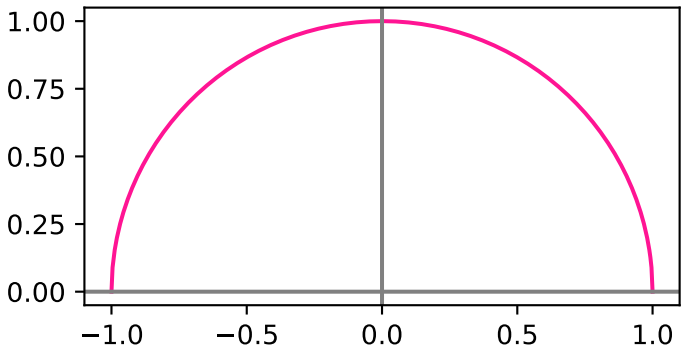


Figure 1.5: Upper semi-circle

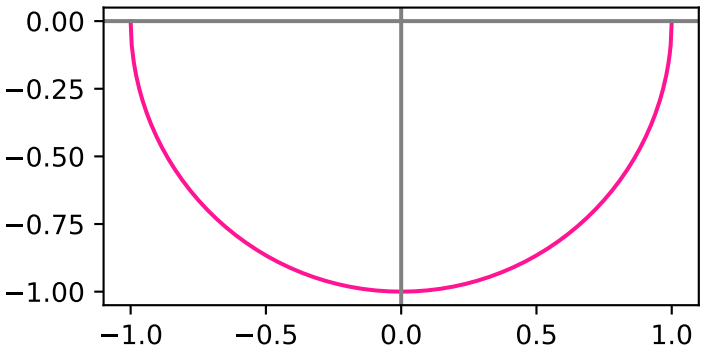


Figure 1.6: Lower semi-circle

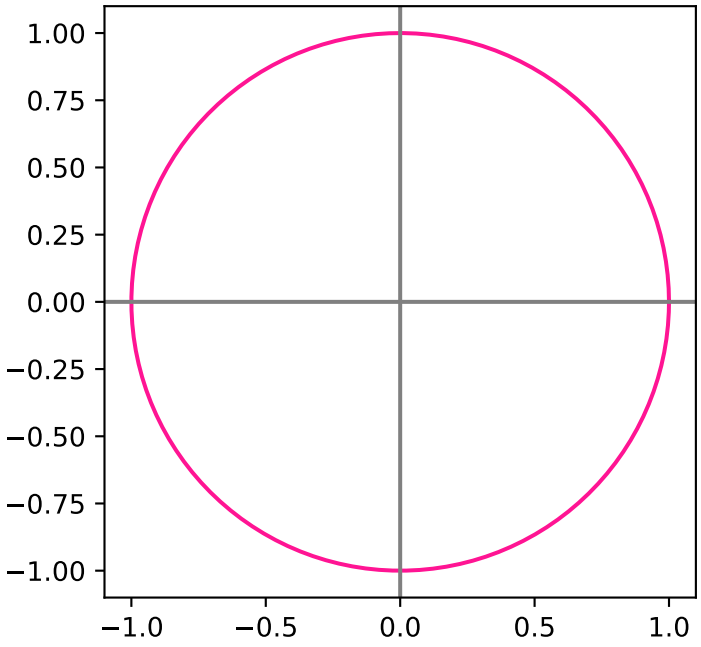


Figure 1.7: Lower semi-circle

Finally we are also able to give a mathematical description of the 3D Helix.

Example: Parametrizing the helix

The Helix plotted above can be parametrized by

$$\gamma : (-\infty, \infty) \to \mathbb{R}^3$$

defined by

$$\gamma_1(t) = \cos(t)\,,\ \gamma_2(t) = \sin(t)\,,\ \gamma_3(t) = t\,.$$

The above equations are in line with our intuition: the helix can be drawn by *tracing a circle while at the same time lifting the pencil.*

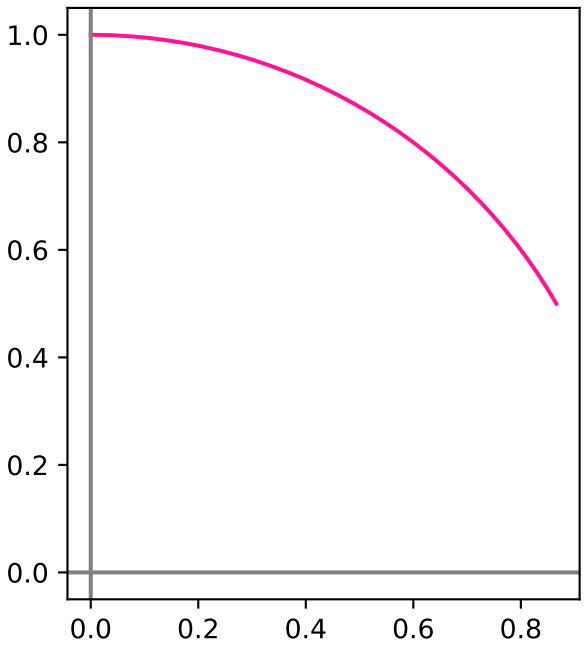


Figure 1.8: Plotting a portion of C

2 Curves in Python

2.1 Curves in 2D

Suppose we want to plot the parabola $y = t^2$ for t in the interval $[-3, 3]$. In our language, this is the two-dimensional curve

$$\gamma(t) = (t, t^2)\,, \quad t \in [-3, 3]\,.$$

The two Python libraries we use to plot γ are **numpy** and **matplotlib**. In short, **numpy** handles multi-dimensional arrays and matrices, and can perform high-level mathematical functions on them. For any question you may have about numpy, answers can be found in the searchable documentation available here. Instead **matplotlib** is a plotting library, with documentation here. Python libraries need to be imported every time you want to use them. In our case we will import:

```
import numpy as np
import matplotlib.pyplot as plt
```

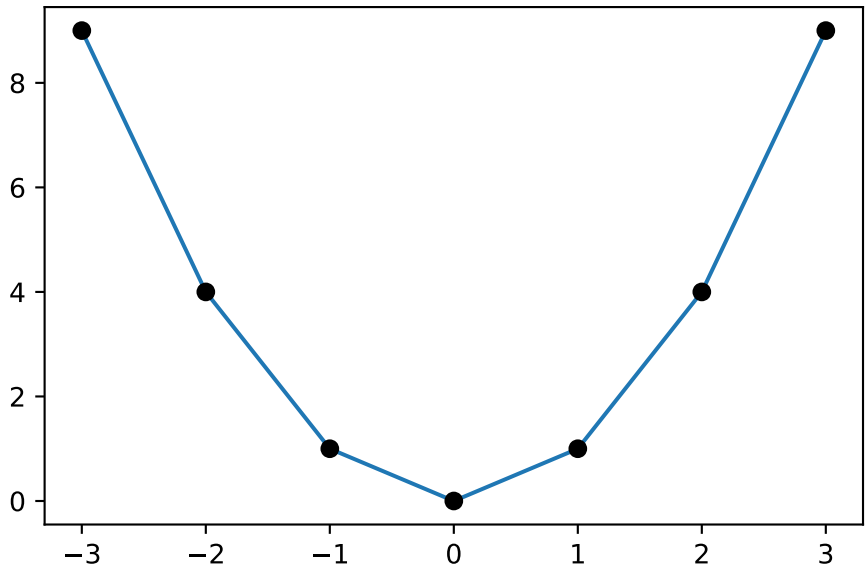
The above imports **numpy** and the module **pyplot** from **matplotlib**, and renames them to `np` and `plt`, respectively. These shorthands are standard in the literature, and they make code much more readable.
The function for plotting 2D graphs is called `plot(x,y)` and is contained in `plt`. As the syntax suggests, `plot` takes as arguments two arrays

$$x = [x_1, \dots, x_n]\,, \quad y = [y_1, \dots, y_n]\,.$$

As output it produces a graph which is the linear interpolation of the points (x_i, y_i) in $\mathbb{R}^2$, that is, consecutive points (x_i, y_i) and (x_{i+1}, y_{i+1}) are connected by a segment. Using `plot`, we can graph the curve $\gamma(t) = (t, t^2)$ like so:

```
# Code for plotting gamma

import numpy as np
import matplotlib.pyplot as plt

# Generating array t
```

```
t = np.array([-3,-2,-1,0,1,2,3])

# Computing array f
f = t**2

# Plotting the curve
plt.plot(t,f)

# Plotting dots
plt.plot(t,f,"ko")

# Showing the plot
plt.show()
```



Let us comment the above code. The variable `t` is a numpy array containing the ordered values

$$t = [-3, -2, -1, 0, 1, 2, 3]\,. \tag{2.1}$$

This array is then squared entry-by-entry via the operation $t**2$ and saved in the new numpy array `f`, that is,

$$f = [9, 4, 1, 0, 1, 4, 9]\,.$$

The arrays `t` and `f` are then passed to `plot(t,f)`, which produces the above linear interpolation, with `t` on the *x-axis* and `f` on the *y-axis*. The command `plot(t,f,'ko')` instead plots a black dot at each point (t_i, f_i). The latter is clearly not needed to obtain a plot, and it was only included to highlight the fact that `plot` is actually producing a linear interpolation between points. Finally `plt.show()` displays the figure in the user window[1].
Of course one can refine the plot so that it resembles the continuous curve $\gamma(t) = (t, t^2)$ that we all have in mind. This is achieved by generating a numpy array `t` with a finer stepsize, invoking the function `np.linspace(a,b,n)`. Such call will return a numpy array which contains `n` evenly spaced points, starts at `a`, and ends in `b`. For example `np.linspace(-3,3,7)` returns our original array `t` at 2.1, as shown below

```
# Displaying output of np.linspace

import numpy as np

# Generates array t by dividing interval
# (-3,3) in 7 parts
t = np.linspace(-3,3, 7)

# Prints array t
print("t =", t)
```

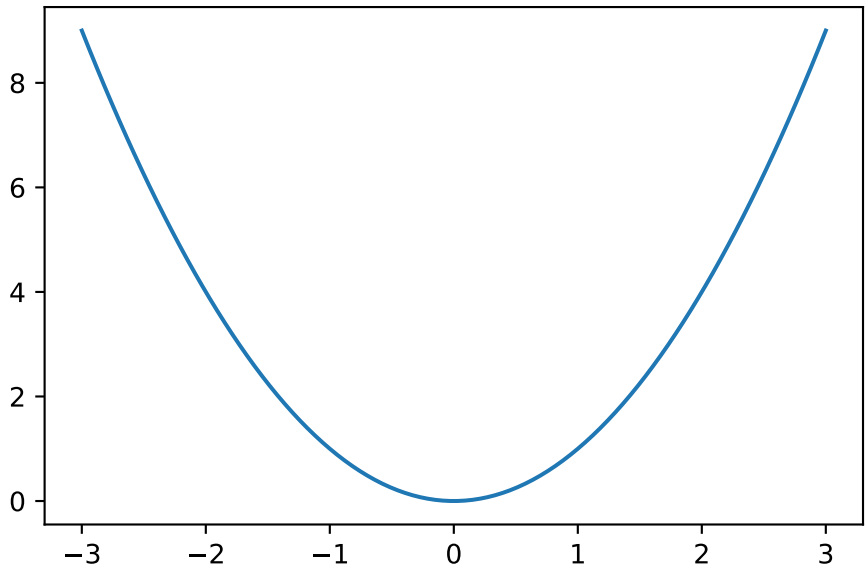
```
t = [-3. -2. -1.  0.  1.  2.  3.]
```

In order to have a more refined plot of γ, we just need to increase n.

```
# Plotting gamma with finer step-size

import numpy as np
import matplotlib.pyplot as plt

# Generates array t by dividing interval
# (-3,3) in 100 parts
t = np.linspace(-3,3, 100)

# Computes f
f = t**2

# Plotting
```

[1]The command `plt.show()` can be omitted if working in Jupyter Notebook, as it is called by default.

```
plt.plot(t,f)
plt.show()
```



We now want to plot a parametric curve $\gamma\colon (a,b) \to \mathbb{R}^2$ with

$$\gamma(t) = (x(t), y(t))\,.$$

Clearly we need to modify the above code. The variable `t` will still be a numpy array produced by `linspace`. We then need to introduce the arrays `x` and `y` which ecode the first and second components of γ, respectively.

```
import numpy as np
import matplotlib.pyplot as plt

# Divides time interval (a,b) in n parts
# and saves output to numpy array t
t = np.linspace(a, b, n)

# Computes gamma from given functions x(y) and y(t)
x = x(t)
y = y(t)
```

```
# Plots the curve
plt.plot(x,y)

# Shows the plot
plt.show()
```

We use the above code to plot the 2D curve known as the Fermat's spiral

$$\gamma(t) = (\sqrt{t}\cos(t), \sqrt{t}\sin(t)) \quad \text{for} \quad t \in [0, 50]\,. \tag{2.2}$$

```
# Plotting Fermat's spiral

import numpy as np
import matplotlib.pyplot as plt

# Divides time interval (0,50) in 500 parts
t = np.linspace(0, 50, 500)

# Computes Fermat's Spiral
x = np.sqrt(t) * np.cos(t)
y = np.sqrt(t) * np.sin(t)

# Plots the Spiral
plt.plot(x,y)
plt.show()
```

Before displaying the output of the above code, a few comments are in order. The array `t` has size 500, due to the behavior of `linspace`. You can also fact check this information by printing `np.size(t)`, which is the numpy function that returns the size of an array. We then use the numpy function `np.sqrt` to compute the square root of the array `t`. The outcome is still an array with the same size of `t`, that is,

$$t = [t_1, \dots, t_n] \quad \Longrightarrow \quad \sqrt{t} = [\sqrt{t_1}, \dots, \sqrt{t_n}]\,.$$

Similary, the call `np.cos(t)` returns the array

$$\cos(t) = [\cos(t_1), \dots, \cos(t_n)]\,.$$

The two arrays `np.sqrt(t)` and `np.cos(t)` are then multiplied, term-by-term, and saved in the array `x`. The array `y` is computed similarly. The command `plt.plot(x,y)` then yields the graph of the Fermat's spiral:

The above plots can be styled a bit. For example we can give a title to the plot, label the axes, plot the spiral by means of green dots, and add a plot legend, as coded below:

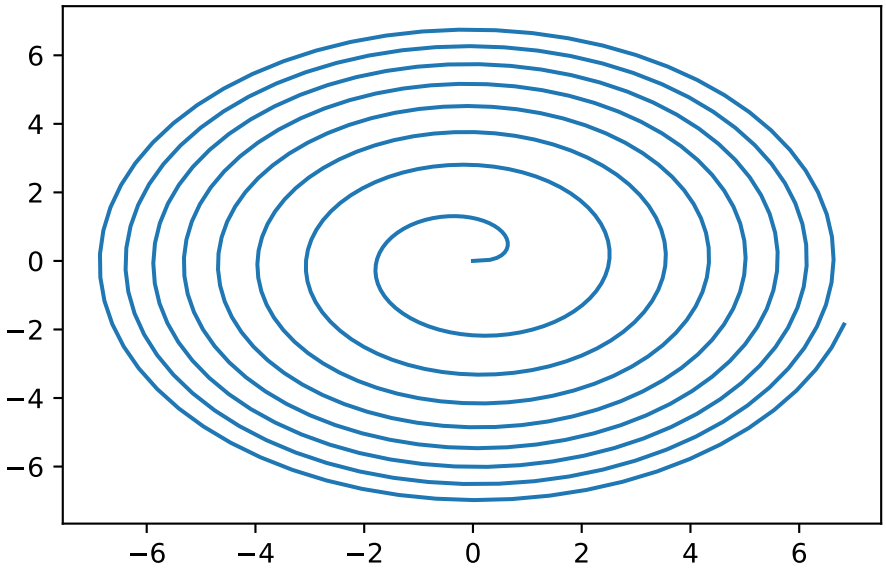


Figure 2.1: Fermat's spiral

```
# Adding some style

import numpy as np
import matplotlib.pyplot as plt

# Computing Spiral
t = np.linspace(0, 50, 500)
x = np.sqrt(t) * np.cos(t)
y = np.sqrt(t) * np.sin(t)

# Generating figure
plt.figure(1, figsize = (4,4))

# Plotting the Spiral with some options
plt.plot(x, y, '--', color = 'deeppink', linewidth = 1.5, label =
 ↪  'Spiral')

# Adding grid
plt.grid(True, color = 'lightgray')

# Adding title
plt.title("Fermat's spiral for t between 0 and 50")
```

```
# Adding axes labels
plt.xlabel("x-axis", fontsize = 15)
plt.ylabel("y-axis", fontsize = 15)

# Showing plot legend
plt.legend()

# Show the plot
plt.show()
```

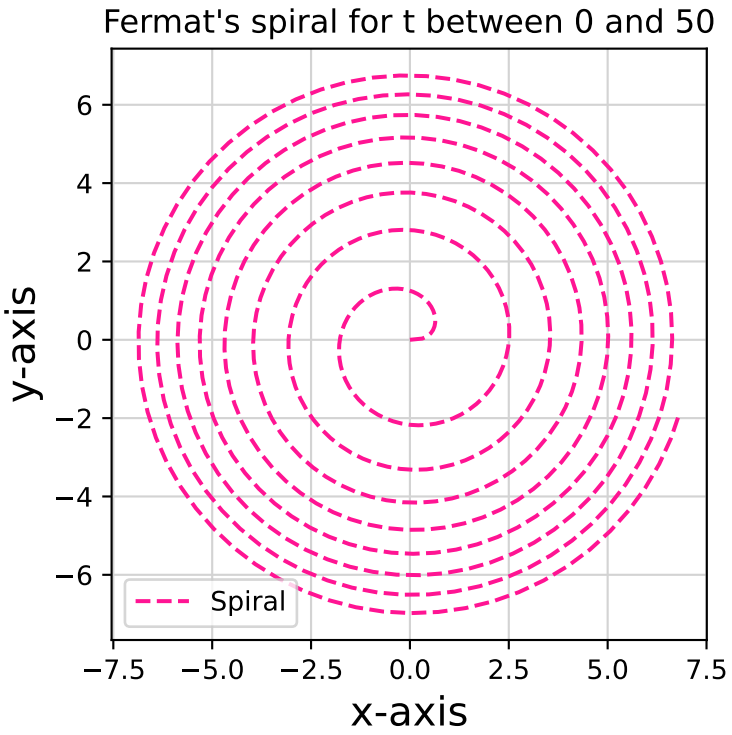


Figure 2.2: Adding a bit of style

Let us go over the novel part of the above code:

- `plt.figure()`: This command generates a figure object. If you are planning on plotting just one figure at a time, then this command is optional: a figure object is generated implicitly when calling `plt.plot`. Otherwise, if working with `n` figures, you need to generate a figure object with `plt.figure(i)` for each `i` between `1` and `n`. The number `i` uniquely

identifies the i-th figure: whenever you call `plt.figure(i)`, Python knows that the next commands will refer to the i-th figure. In our case we only have one figure, so we have used the identifier `1`. The second argument `figsize = (a,b)` in `plt.figure()` specifies the size of `figure 1` in inches. In this case we generated a figure 4 x 4 inches.

- `plt.plot`: This is plotting the arrays `x` and `y`, as usual. However we are adding a few aestethic touches: the curve is plotted in *dashed* style with `--`, in *deep pink* color and with a line width of 1.5. Finally this plot is labelled *Spiral*.
- `plt.grid`: This enables a grid in *light gray* color.
- `plt.title`: This gives a title to the figure, displayed on top.
- `plt.xlabel` and `plt.ylabel`: These assign labels to the axes, with font size 15 points.
- `plt.legend()`: This plots the legend, with all the labels assigned in the `plt.plot` call. In this case the only label is *Spiral*.

Matplotlib styles

There are countless plot types and options you can specify in **matplotlib**, see for example the Matplotlib Gallery. Of course there is no need to remember every single command: a quick Google search can do wonders.

Generating arrays

There are several ways of generating evenly spaced arrays in Python. For example the function `np.arange(a,b,s)` returns an array with values within the half-open interval $[a, b)$, with spacing between values given by `s`. For example

```
import numpy as np

t = np.arange(0,1, 0.2)
print("t =",t)
```

```
t = [0.  0.2 0.4 0.6 0.8]
```

2.2 Implicit curves 2D

A curve γ in $\mathbb{R}^2$ can also be defined as the set of points $(x, y) \in \mathbb{R}^2$ satisfying

$$f(x, y) = 0$$

for some given $f\colon \mathbb{R}^2 \to \mathbb{R}$. For example let us plot the curve γ implicitly defined by

$$f(x, y) = (3x^2 - y^2)^2\ y^2 - (x^2 + y^2)^4$$

for $-1 \leq x, y \leq 1$. First, we need a way to generate a grid in $\mathbb{R}^2$ so that we can evaluate f on such grid. To illustrate how to do this, let us generate a grid of spacing 1 in the 2D square $[0,4]^2$. The goal is to obtain the 5 x 5 matrix of coordinates

$$A = \begin{pmatrix} (0,0) & (1,0) & (2,0) & (3,0) & (4,0) \\ (0,1) & (1,1) & (2,1) & (3,1) & (4,1) \\ (0,2) & (1,2) & (2,2) & (2,3) & (2,4) \\ (0,3) & (1,3) & (2,3) & (3,3) & (3,4) \\ (0,4) & (1,4) & (2,4) & (3,4) & (4,4) \end{pmatrix}$$

which corresponds to the grid of points

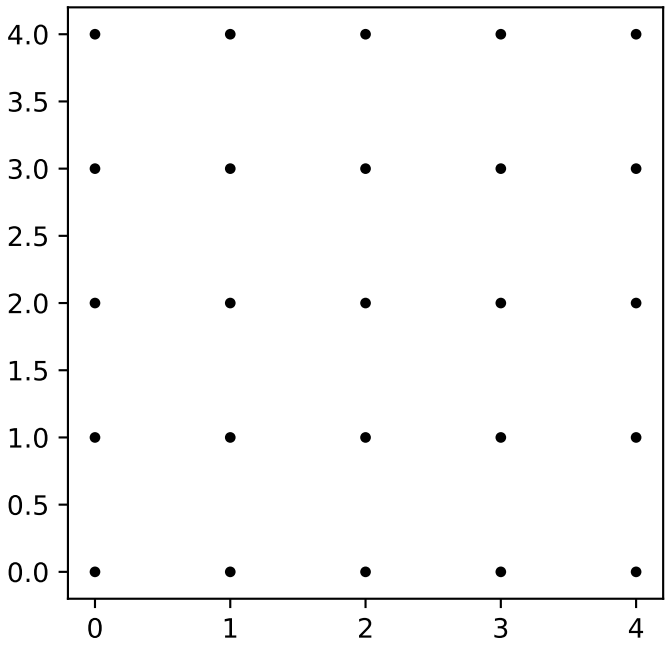


Figure 2.3: The 5 x 5 grid corresponding to the matrix A

To achieve this, first generate `x` and `y` coordinates using

```
x = np.linspace(0, 4, 5)
y = np.linspace(0, 4, 5)
```

This generates coordinates

$$x = [0,1,2,3,4]\,, \quad y = [0,1,2,3,4]\,.$$

We then need to obtain two matrices X and Y: one for the x coordinates in A, and one for the y coordinates in A. This can be achieved with the code

```
X[0,0] = 0
X[0,1] = 1
X[0,2] = 2
X[0,3] = 3
X[0,4] = 4
X[1,0] = 0
X[1,1] = 1
...
x[4,3] = 3
x[4,4] = 4
```

and similarly for Y. The output would be the two matrices X and Y

$$X = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 & 4 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \end{pmatrix}$$

If now we plot X against Y via the command

```
plt.plot(X, Y, 'k.')
```

we obtain Figure 2.3. In the above command the style `'k.'` represents black dots. This procedure would be impossible with large vectors. Thankfully there is a function in numpy doing exactly what we need: `np.meshgrid`.

```
# Demonstrating np.meshgrid

import numpy as np

# Generating x and y coordinates
xlist = np.linspace(0, 4, 5)
ylist = np.linspace(0, 4, 5)

# Generating grid X, Y
X, Y = np.meshgrid(xlist, ylist)
```

```
# Printing the matrices X and Y
# np.array2string is only needed to align outputs
print('X =', np.array2string(X, prefix='X= '))
print('\n')
print('Y =', np.array2string(Y, prefix='Y= '))
```

```
X = [[0. 1. 2. 3. 4.]
     [0. 1. 2. 3. 4.]
     [0. 1. 2. 3. 4.]
     [0. 1. 2. 3. 4.]
     [0. 1. 2. 3. 4.]]


Y = [[0. 0. 0. 0. 0.]
     [1. 1. 1. 1. 1.]
     [2. 2. 2. 2. 2.]
     [3. 3. 3. 3. 3.]
     [4. 4. 4. 4. 4.]]
```

Now that we have our grid, we can evaluate the function f on it. This is simply done with the command

```
Z =((3*(X**2) - Y**2)**2)*(Y**2) - (X**2 + Y**2)**4
```

This will return the matrix Z containing the values $f(x_i, y_i)$ for all (x_i, y_i) in the grid $[X, Y]$. We are now interested in plotting the points in the grid $[X, Y]$ for which Z is zero. This is achieved with the command

```
plt.contour(X, Y, Z, [0])
```

Putting the above observations together, we have the code for plotting the curve $f = 0$ for $-1 \leq x, y \leq 1$.

```
# Plotting f=0

import numpy as np
import matplotlib.pyplot as plt

# Generates coordinates and grid
xlist = np.linspace(-1, 1, 5000)
```

```
ylist = np.linspace(-1, 1, 5000)
X, Y = np.meshgrid(xlist, ylist)

# Computes f
Z =((3*(X**2) - Y**2)**2)*(Y**2) - (X**2 + Y**2)**4

# Creates figure object
plt.figure(figsize = (4,4))

# Plots level set Z = 0
plt.contour(X, Y, Z, [0])

# Set axes labels
plt.xlabel("x-axis", fontsize = 15)
plt.ylabel("y-axis", fontsize = 15)

# Shows plot
plt.show()
```

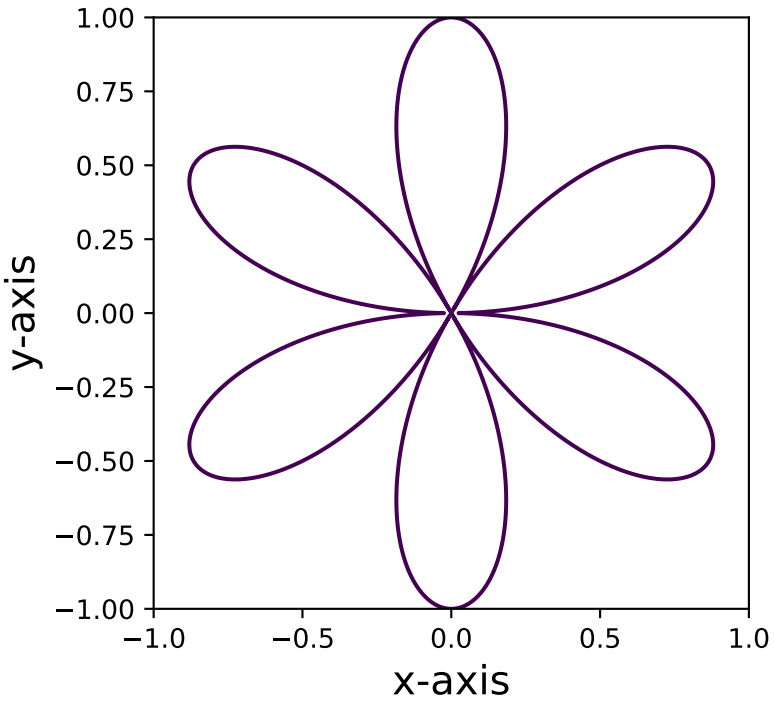


Figure 2.4: Plot of the curve defined by f=0

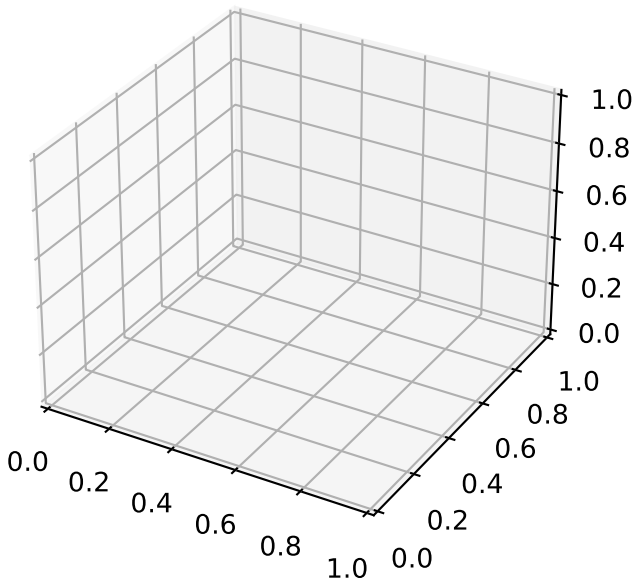
2.3 Curves in 3D

Plotting in 3D with matplotlib requires the `mplot3d` toolkit, see here for documentation. Therefore our first lines will always be

```
# Packages for 3D plots

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
```

We can now generate empty 3D axes

```
# Generates and plots empty 3D axes

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

# Creates figure object
fig = plt.figure(figsize = (4,4))

# Creates 3D axes object
ax = plt.axes(projection = '3d')

# Shows the plot
plt.show()
```



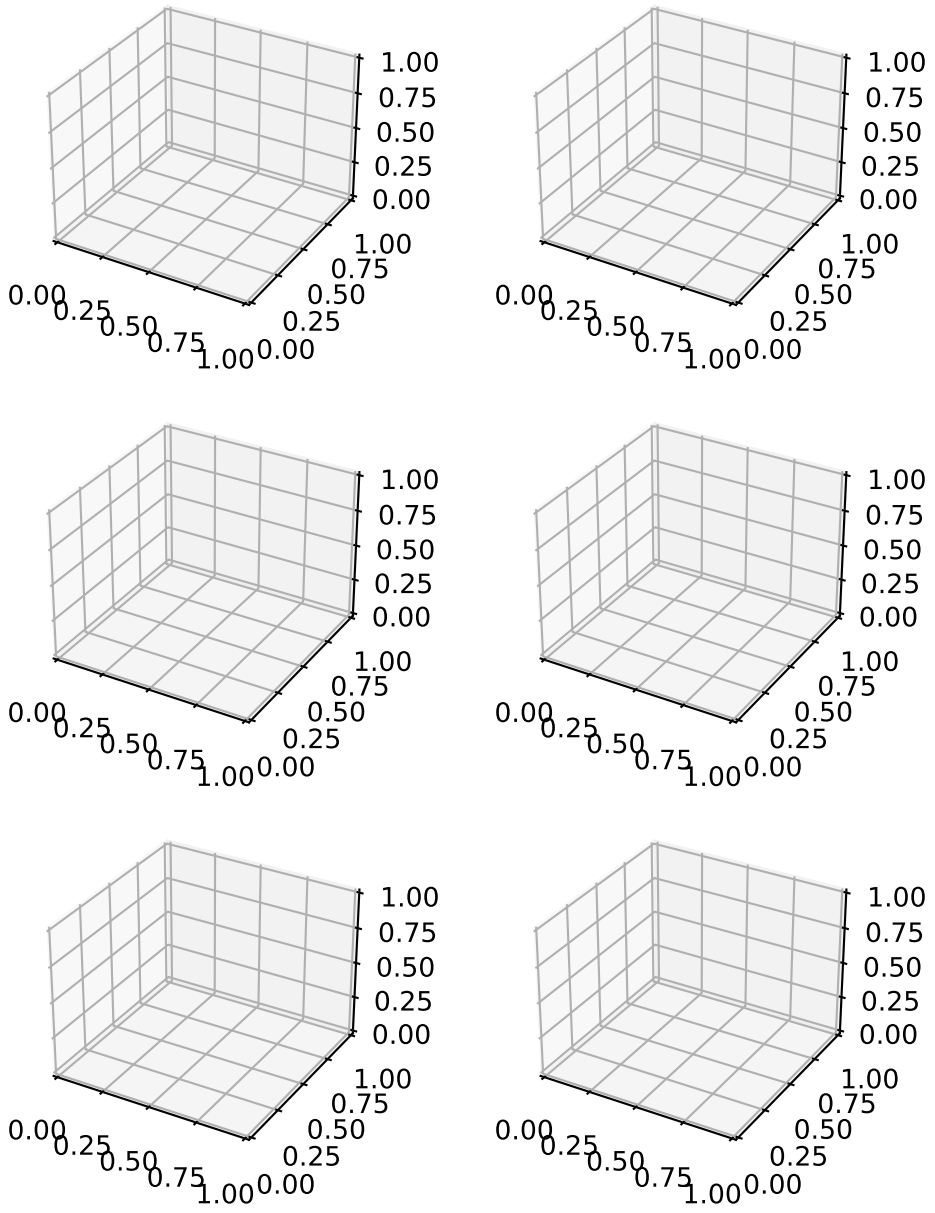
In the above code `fig` is a figure object, while `ax` is an axes object. In practice, the figure object contains the axes objects, and the actual plot information will be contained in axes. If you want multiple plots in the figure container, you should use the command

```
ax = fig.add_subplot(nrows = m, ncols = n, pos = k)
```

This generates an axes object `ax` in position `k` with respect to a `m x n` grid of plots in the container figure. For example we can create a 3 x 2 grid of empty 3D axes as follows

```
# Generates 3 x 2 empty 3D axes

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

# Creates container figure object
fig = plt.figure(figsize = (6,8))

# Creates 6 empty 3D axes objects
ax1 = fig.add_subplot(3, 2, 1, projection = '3d')
ax2 = fig.add_subplot(3, 2, 2, projection = '3d')
ax3 = fig.add_subplot(3, 2, 3, projection = '3d')
```

```
ax4 = fig.add_subplot(3, 2, 4, projection = '3d')
ax5 = fig.add_subplot(3, 2, 5, projection = '3d')
ax6 = fig.add_subplot(3, 2, 6, projection = '3d')

# Shows the plot
plt.show()
```



We are now ready to plot a 3D parametric curve $\gamma\colon (a,b) \to \mathbb{R}^3$ of the form

$$\gamma(t) = (x(t), y(t), z(t))$$

with the code

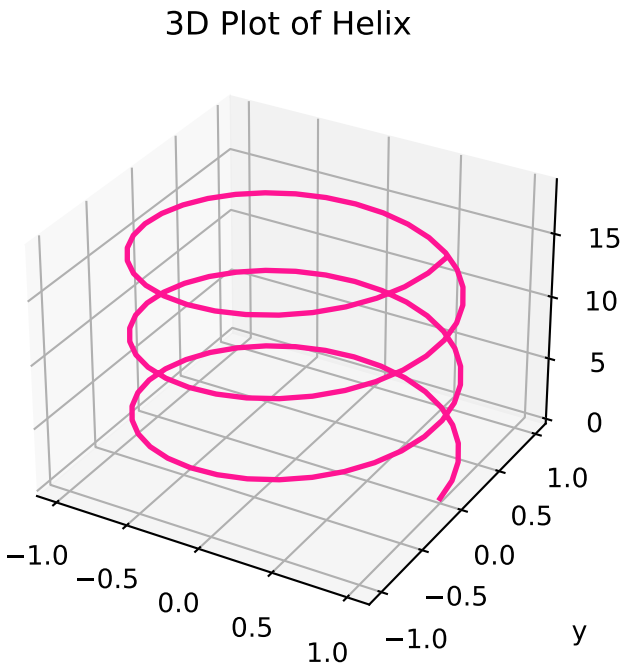
```
# Code to plot 3D curve

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

# Generates figure and 3D axes
fig = plt.figure(figsize = (size1,size2))
ax = plt.axes(projection = '3d')

# Plots grid
ax.grid(True)

# Divides time interval (a,b)
# into n parts and saves them in array t
t = np.linspace(a, b, n)

# Computes the curve gamma on array t
# for given functions x(t), y(t), z(t)
x = x(t)
y = y(t)
z = z(t)

# Plots gamma
ax.plot3D(x, y, z)

# Setting title for plot
ax.set_title('3D Plot of gamma')

# Setting axes labels
ax.set_xlabel('x', labelpad = 'p')
ax.set_ylabel('y', labelpad = 'p')
ax.set_zlabel('z', labelpad = 'p')

# Shows the plot
plt.show()
```

For example we can use the above code to plot the Helix

$$x(t) = \cos(t)\,, \quad y(t) = \sin(t)\,, \quad z(t) = t \tag{2.3}$$

for $t \in [0, 6\pi]$.

```
# Plotting 3D Helix

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

# Generates figure and 3D axes
fig = plt.figure(figsize = (4,4))
ax = plt.axes(projection = '3d')

# Plots grid
ax.grid(True)

# Divides time interval (0,6pi) in 100 parts
t = np.linspace(0, 6*np.pi, 100)

# Computes Helix
x = np.cos(t)
y = np.sin(t)
z = t

# Plots Helix - We added some styling
ax.plot3D(x, y, z, color = "deeppink", linewidth = 2)

# Setting title for plot
ax.set_title('3D Plot of Helix')

# Setting axes labels
ax.set_xlabel('x', labelpad = 20)
ax.set_ylabel('y', labelpad = 20)
ax.set_zlabel('z', labelpad = 20)

# Shows the plot
plt.show()
```



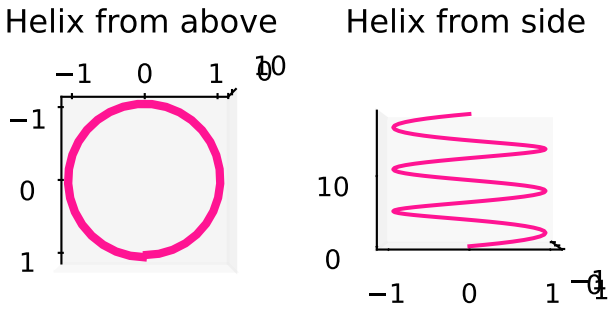
We can also change the viewing angle for a 3D plot store in `ax`. This is done via

```
ax.view_init(elev = e, azim = a)
```

which displays the 3D axes with an elevation angle `elev` of `e` degrees and an azimuthal angle `azim` of `a` degrees. In other words, the 3D plot will be rotated by `e` degrees above the xy-plane and by `a` degrees around the z-axis. For example, let us plot the helix with 2 viewing angles. Note that we generate 2 sets of axes with the `add_subplot` command discussed above.

```
# Plotting 3D Helix

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

# Generates figure object
fig = plt.figure(figsize = (4,4))

# Generates 2 sets of 3D axes
ax1 = fig.add_subplot(1, 2, 1, projection = '3d')
ax2 = fig.add_subplot(1, 2, 2, projection = '3d')
```

```
# We will not show a grid this time
ax1.grid(False)
ax2.grid(False)

# Divides time interval (0,6pi) in 100 parts
t = np.linspace(0, 6*np.pi, 100)

# Computes Helix
x = np.cos(t)
y = np.sin(t)
z = t

# Plots Helix on both axes
ax1.plot3D(x, y, z, color = "deeppink", linewidth = 1.5)
ax2.plot3D(x, y, z, color = "deeppink", linewidth = 1.5)

# Setting title for plots
ax1.set_title('Helix from above')
ax2.set_title('Helix from side')

# Changing viewing angle of ax1
# View from above has elev = 90 and azim = 0
ax1.view_init(elev = 90, azim = 0)

# Changing viewing angle of ax2
# View from side has elev = 0 and azim = 0
ax2.view_init(elev = 0, azim = 0)

# Shows the plot
plt.show()
```



2.4 Interactive plots

`Matplotlib` produces beautiful static plots; however it lacks built in interactivity. For this reason I would also like to show you how to plot curves with `Plotly`, a very popular Python graphic library which has built in interactivity. Documentation for `Plotly` and lots of examples can be found here.

2.4.1 2D Plots

Say we want to plot the 2D curve $\gamma\colon (a,b) \to \mathbb{R}^2$ parametrized by

$$\gamma(t) = (x(t), y(t))\,.$$

The `Plotly` module needed is called `graph_objects`, usually imported as `go`. The function for line plots is called `Scatter`. For documentation and examples see link. The code for plotting γ is as follows.

```
# Plotting gamma 2D

# Import libraries
import numpy as np
import plotly.graph_objects as go

# Compute times grid by dividing (a,b) in
# n equal parts
t = np.linspace(a, b, n)

# Compute the parametric curve gamma
# for given functions x(t) and y(t)
```

```
x = x(t)
y = y(t)

# Create empty figure object and saves
# it in the variable "fig"
fig = go.Figure()

# Create the line plot object
data = go.Scatter(x = x, y = y, mode = 'lines', name = 'gamma')

# Add "data" plot to the figure "fig"
fig.add_trace(data)

# Display the figure
fig.show()
```

Some comments about the functions called above:

- `go.Figure`: generates an empty Plotly figure
- `go.Scatter`: generates the actual plot. By default a scatter plot is produced. To obtain linear interpolation of the points, set `mode = 'lines'`. You can also label the plot with `name = "string"`
- `add_trace`: adds a plot to a figure
- `show`: displays a figure

As an example, let us plot the Fermat's Spiral defined at 2.2. Compared to the above code, we also add a bit of styling.

```
# Plotting Fermat's Spiral

# Import libraries
import numpy as np
import plotly.graph_objects as go

# Compute times grid by dividing (0,50) in
# 500 equal parts
t = np.linspace(0, 50, 500)

# Computes Fermat's Spiral
x = np.sqrt(t) * np.cos(t)
y = np.sqrt(t) * np.sin(t)
```

```
# Create empty figure object and saves
# it in the variable "fig"
fig = go.Figure()

# Create the line plot object
data = go.Scatter(x = x, y = y, mode = 'lines', name = 'gamma')

# Add "data" plot to the figure "fig"
fig.add_trace(data)

# Here we start with the styling options
# First we set a figure title
fig.update_layout(title_text = "Plotting Fermat's Spiral with Plotly")

# Adjust figure size
fig.update_layout(autosize = False, width = 600, height = 600)

# Change background canvas color
fig.update_layout(paper_bgcolor = "snow")

# Axes styling: adding title and ticks positions
fig.update_layout(
xaxis=dict(
        title_text="X-axis Title",
        titlefont=dict(size=20),
        tickvals=[-6,-4,-2,0,2,4,6],
        ),

yaxis=dict(
        title_text="Y-axis Title",
        titlefont=dict(size=20),
        tickvals=[-6,-4,-2,0,2,4,6],
        )
)

# Display the figure
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

The above code generates an image that cannot be rendered in pdf. To see the output, please click here for the digital version of these notes. Note that the style customizations could be listed in a single call of the function `update_layout`. There are also pretty buit-in themes available, see here. The layout can be specified with the command

```
fig.update_layout(template = template_name)
```

where `template_name` can be `"plotly"`, `"plotly_white"`, `"plotly_dark"`, `"ggplot2"`, `"seaborn"`, `"simple_white"`.

2.4.2 3D Plots

We now want to plot a 3D curve $\gamma\colon (a,b) \to \mathbb{R}^3$ parametrized by

$$\gamma(t) = (x(t), y(t), z(t))\,.$$

Again we use the `Plotly` module `graph_objects`, imported as `go`. The function for 3D line plots is called `Scatter3d`, and documentation and examples can be found at link. The code for plotting γ is as follows.

```
# Plotting gamma 3D

# Import libraries
import numpy as np
import plotly.graph_objects as go

# Compute times grid by dividing (a,b) in
# n equal parts
t = np.linspace(a, b, n)

# Compute the parametric curve gamma
# for given functions x(t), y(t), z(t)
x = x(t)
y = y(t)
z = z(t)

# Create empty figure object and saves
# it in the variable "fig"
fig = go.Figure()
```

```
# Create the line plot object
data = go.Scatter3d(x = x, y = y, z = z, mode = 'lines', name = 'gamma')

# Add "data" plot to the figure "fig"
fig.add_trace(data)

# Display the figure
fig.show()
```

The functions `go.Figure`, `add_trace` and `show` appearing above are described in the previous Section. The new addition is `go.Scatter3d`, which generates a 3D scatter plot of the points stored in the array `[x,y,z]`. Setting `mode = 'lines'` results in a linear interpolation of such points. As before, the curve can be labeled by setting `name = "string"`.

As an example, we plot the 3D Helix defined at 2.3. We also add some styling. We can also use the same pre-defined templates described for `go.Scatter` in the previous section, see here for official documentation.

```
# Plotting 3D Helix

# Import libraries
import numpy as np
import plotly.graph_objects as go

# Divides time interval (0,6pi) in 100 parts
t = np.linspace(0, 6*np.pi, 100)

# Computes Helix
x = np.cos(t)
y = np.sin(t)
z = t

# Create empty figure object and saves
# it in the variable "fig"
fig = go.Figure()

# Create the line plot object
# We add options for the line width and color
data = go.Scatter3d(
    x = x, y = y, z = z,
```

```
    mode = 'lines', name = 'gamma',
    line = dict(width = 10, color = "darkblue")
    )

# Add "data" plot to the figure "fig"
fig.add_trace(data)

# Here we start with the styling options
# First we set a figure title
fig.update_layout(title_text = "Plotting 3D Helix with Plotly")

# Adjust figure size
fig.update_layout(
    autosize = False,
    width = 600,
    height = 600
    )

# Set pre-defined template
fig.update_layout(template = "seaborn")

# Options for curve line style


# Display the figure
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

The above code generates an image that cannot be rendered in pdf. To see the output, please click here for the digital version of these notes. Once again, the style customizations could be listed in a single call of the function `update_layout`.

3 Parametrized curves

Let us recall the definition of **parametrized curve**.

Definition 3.1: Parametrized curve

A **parametrized curve** in $\mathbb{R}^n$ is a function

$$\gamma : (a, b) \to \mathbb{R}^n .$$

where

$$(a, b) = \{t \in \mathbb{R} : \ a < t < b\} ,$$

with

$$-\infty \leq a < b \leq \infty .$$

The **components** of $\gamma(t) \in \mathbb{R}^n$ are denoted by

$$\gamma(t) = (\gamma_1(t), \dots , \gamma_n(t)) ,$$

where the components are functions

$$\gamma_i : (a, b) \to \mathbb{R} ,$$

for all $i = 1, \dots , n$.

As we already mentioned, the aim of the course is to study curves by **differentiating** them. Let us see what that means for curves.

Definition 3.2: Smooth functions

A scalar function $f : (a, b) \to \mathbb{R}$ is called **smooth** if the derivative

$$\frac{d^n f}{dt^n}$$

exists for all $n \geq 1$ and $t \in (a, b)$.

We will denote the first and second derivatives of f as follows:

$$\dot{f} := \frac{df}{dt}\,, \quad \ddot{f} := \frac{d^2 f}{dt^2}\,.$$

Example

The function $f(x) = x^4$ is smooth, with

$$\frac{df}{dt} = 4x^3, \ \frac{d^2 f}{dt^2} = 12x^2\,, \tag{3.1}$$

$$\frac{d^3 f}{dt^3} = 24x, \ \frac{d^4 f}{dt^4} = 24\,, \tag{3.2}$$

$$\frac{d^n f}{dt^n} = 0 \ \text{ for all } \ n \geq 5\,. \tag{3.3}$$

Other examples smooth functions are polynomials, as well as

$$f(t) = \cos(t), \ f(t) = \sin(t)\,, \ f(t) = e^t\,.$$

Definition 3.3

Let $\gamma : (a, b) \to \mathbb{R}^n$ with

$$\gamma(t) = (\gamma_1(t), \dots, \gamma_n(t))$$

be a parametrized curve. We say that γ is **smooth** if the components

$$\gamma_i : (a, b) \to \mathbb{R}$$

are smooth for all $i = 1, \dots, n$. The derivatives of γ are

$$\frac{d^k \gamma}{dt^k} := \left(\frac{d^k \gamma_1}{dt^k}, \dots, \frac{d^k \gamma_n}{dt^k} \right)$$

for all $k \in \mathbb{N}$. As a shorthand, we will denote the first derivative of γ as

$$\dot{\gamma} := \frac{d\gamma}{dt} = \left(\frac{d\gamma_1}{dt}, \dots, \frac{d\gamma_n}{dt} \right)$$

and the second by

$$\ddot{\gamma} := \frac{d^2\gamma}{dt^2} = \left(\frac{d^2\gamma_1}{dt^2}, \dots, \frac{d^2\gamma_n}{dt^2}\right) .$$

In Figure 3.1 we skectch a smooth and a non-smooth curve. Notice that the curve on the right is smooth, except for the point x.

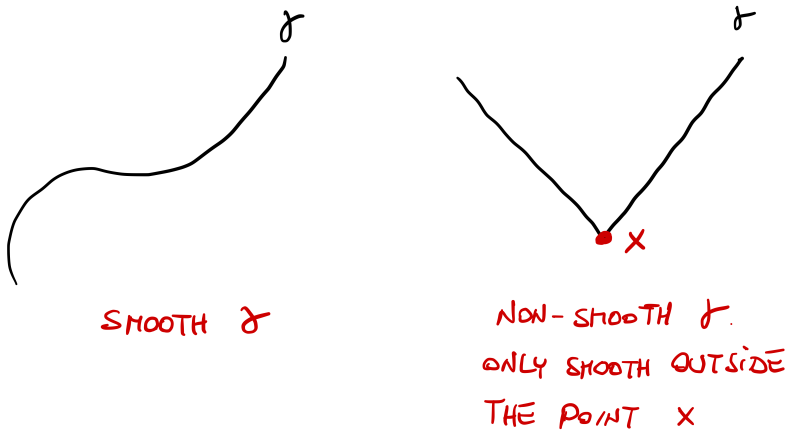


Figure 3.1: Example of smooth and non-smooth curves

We will work under the following assumption.

Assumption

All the parametrized curves in this lecture notes are assumed to be **smooth**.

Example

The circle

$$\gamma(t) = (\cos(t), \sin(t))$$

is a smooth parametrized curve, since both $\cos(t)$ and $\sin(t)$ are smooth functions. We have

$$\dot{\gamma} = (-\sin(t), \cos(t)) .$$

For example the derivative of γ at the point $(0, 1)$ is given by

$$\dot{\gamma}(\pi/2) = (-\sin(\pi/2), \cos(\pi/2)) = (-1, 0) .$$

The plot of the circle and the derivative vector at $(-1, 0)$ can be seen in Figure 3.2.

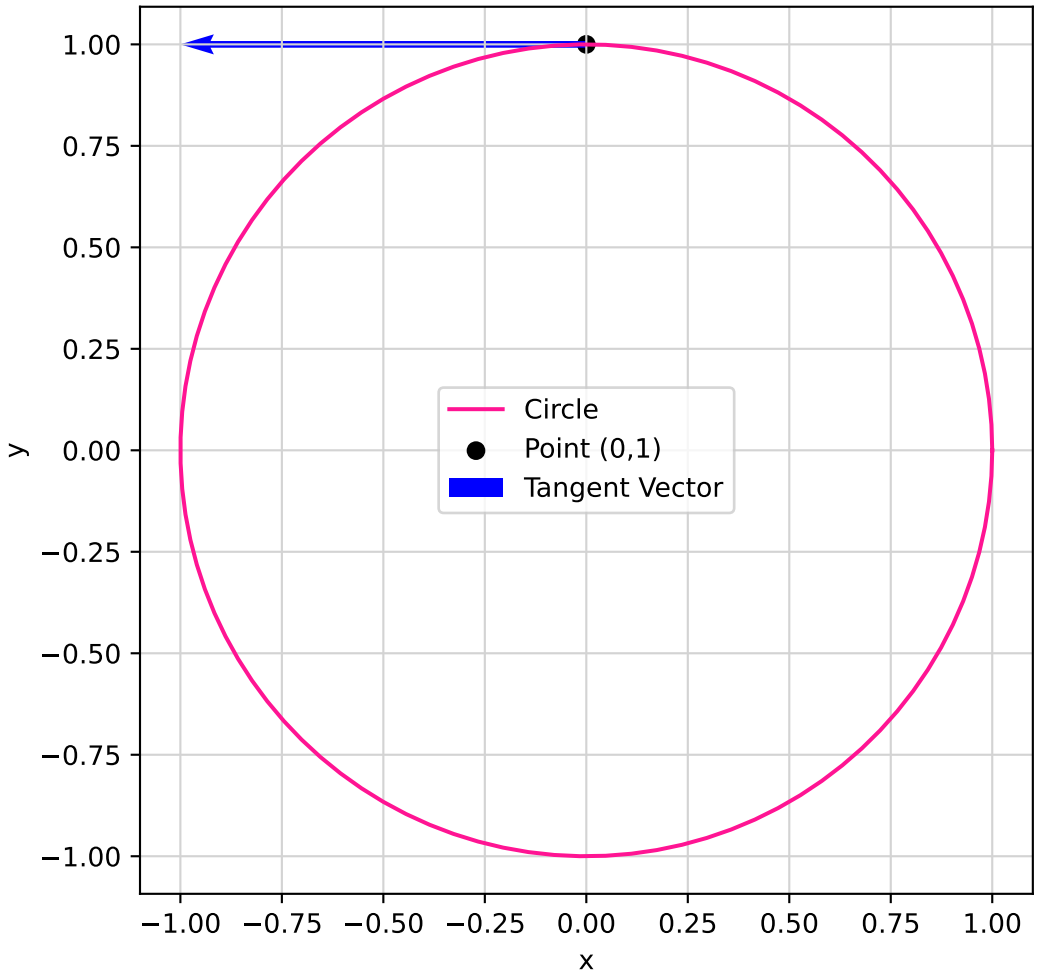


Figure 3.2: Plot of Circle and Tangent Vector at $(0, 1)$

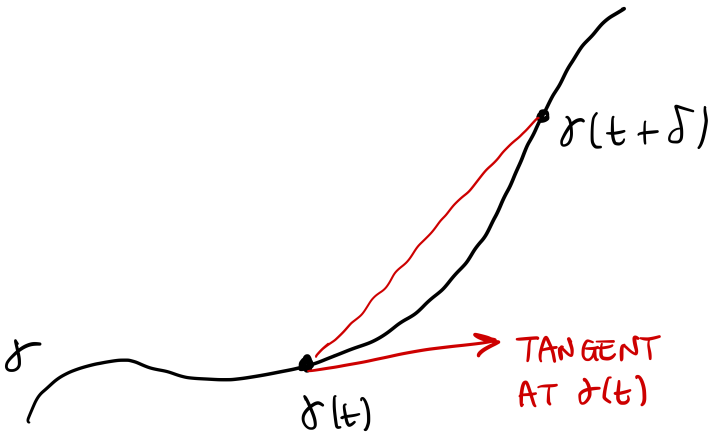


Figure 3.3: Approximating the tangent vector

4 Surfaces

Coming soon

5 Surfaces in Python

5.1 Plots with Matplotlib

I will take for granted all the commands explained in Chapter 2. Suppose we want to plot a surface S which is defined by the parametric equations

$$x = x(u, v)\,, \quad y = y(u, v)\,, \quad z = z(u, v)$$

for $u \in (a, b)$ and $v \in (c, d)$. This can be done via the function called `plot_surface` contained in the mplot3d Toolkit. This function works as follows: first we generate a mesh-grid $[U, V]$ from the coordinates (u, v) via the command

```
[U, V] = np.meshgrid(u, v)
```

Then we compute the parametric surface on the mesh

```
x = x (U, V)
y = y (U, V)
z = z (U, V)
```

Finally we can plot the surface with the command

```
plt.plot_surface(x, y, z)
```

The complete code looks as follows.

```
# Plotting surface S

# Importing numpy, matplotlib and mplot3d
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

# Generates figure object of size m x n
```

```
fig = plt.figure(figsize = (m,n))

# Generates 3D axes
ax = plt.axes(projection = '3d')

# Shows axes grid
ax.grid(True)

# Generates coordinates u and v
# by dividing the interval (a,b) in n parts
# and the interval (c,d) in m parts
u = np.linspace(a, b, m)
v = np.linspace(c, d, n)

# Generates grid [U,V] from the coordinates u, v
U, V = np.meshgrid(u, v)

# Computes S given the functions x, y, z
# on the grid [U,V]
x = x(U,V)
y = y(U,V)
z = z(U,V)

# Plots the surface S
ax.plot_surface(x, y, z)

# Setting plot title
ax.set_title('The surface S')

# Setting axes labels
ax.set_xlabel('x', labelpad=10)
ax.set_ylabel('y', labelpad=10)
ax.set_zlabel('z', labelpad=10)

# Setting viewing angle
ax.view_init(elev = e, azim = a)

# Showing the plot
plt.show()
```

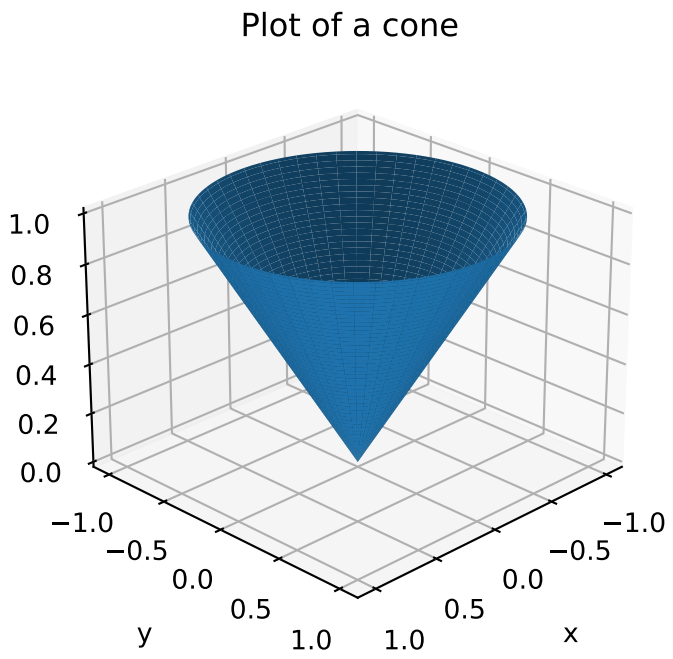
For example let us plot a cone described parametrically by:

$$x = u\cos(v)\,, \quad y = u\sin(v)\,, \quad z = u$$

for $u \in (0,1)$ and $v \in (0, 2\pi)$. We adapt the above code:

```
# Plotting a cone

# Importing numpy, matplotlib and mplot3d
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

# Generates figure object of size 4 x 4
fig = plt.figure(figsize = (4,4))

# Generates 3D axes
ax = plt.axes(projection = '3d')

# Shows axes grid
ax.grid(True)

# Generates coordinates u and v by dividing
# the intervals (0,1) and (0,2pi) in 100 parts
u = np.linspace(0, 1, 100)
v = np.linspace(0, 2*np.pi, 100)

# Generates grid [U,V] from the coordinates u, v
U, V = np.meshgrid(u, v)

# Computes the surface on grid [U,V]
x = U * np.cos(V)
y = U * np.sin(V)
z = U

# Plots the cone
ax.plot_surface(x, y, z)

# Setting plot title
ax.set_title('Plot of a cone')

# Setting axes labels
```

```
ax.set_xlabel('x', labelpad=10)
ax.set_ylabel('y', labelpad=10)
ax.set_zlabel('z', labelpad=10)

# Setting viewing angle
ax.view_init(elev = 25, azim = 45)

# Showing the plot
plt.show()
```



As discussed in Chapter 2, we can have multiple plots in the same figure. For example let us plot the torus viewed from 2 angles. The parametric equations are:

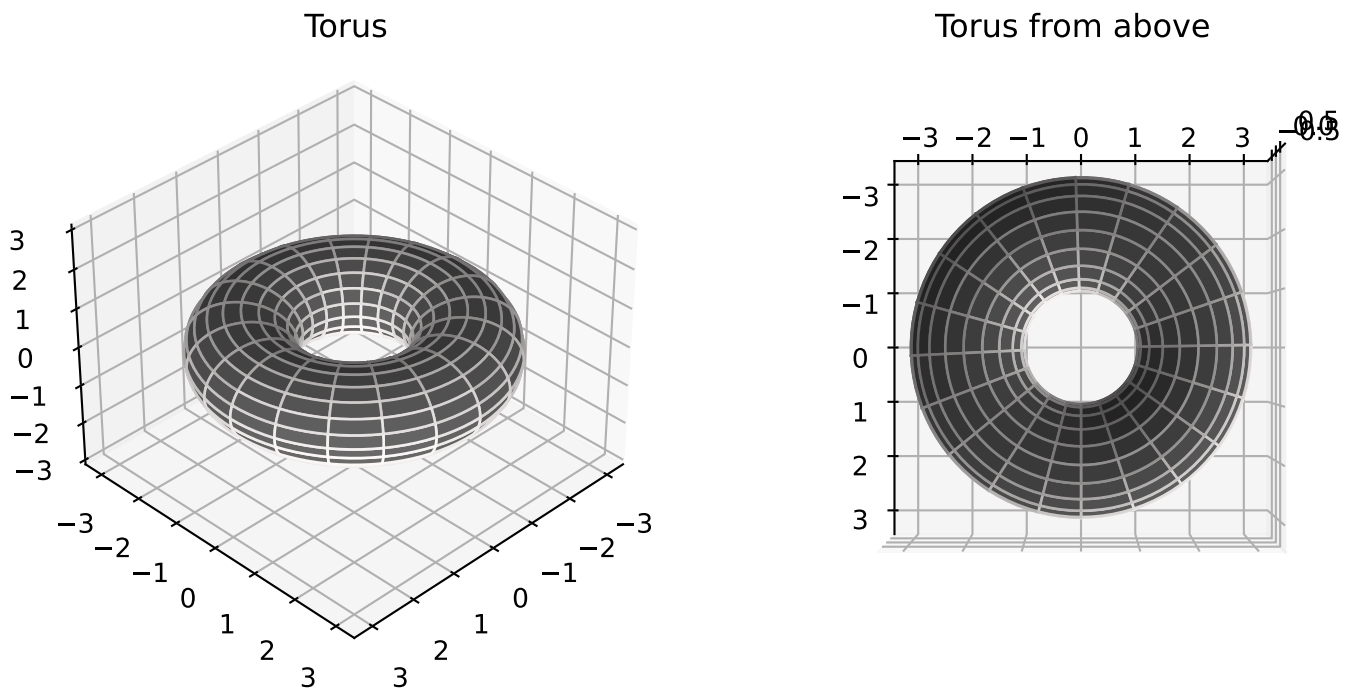
$$x = (R + r\cos(u))\cos(v)$$
$$y = (R + r\cos(u))\sin(v)$$
$$z = r\sin(u)$$

for $u, v \in (0, 2\pi)$ and with

- R distance from the center of the tube to the center of the torus
- r radius of the tube

```
# Plotting torus seen from 2 angles

# Importing numpy, matplotlib and mplot3d
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

# Generates figure object of size 9 x 5
fig = plt.figure(figsize = (9,5))

# Generates 2 sets of 3D axes
ax1 = fig.add_subplot(1, 2, 1, projection = '3d')
ax2 = fig.add_subplot(1, 2, 2, projection = '3d')

# Shows axes grid
ax1.grid(True)
ax2.grid(True)

# Generates coordinates u and v by dividing
# the interval (0,2pi) in 100 parts
u = np.linspace(0, 2*np.pi, 100)
v = np.linspace(0, 2*np.pi, 100)

# Generates grid [U,V] from the coordinates u, v
U, V = np.meshgrid(u, v)

# Computes the torus on grid [U,V]
# with radii r = 1 and R = 2
R = 2
r = 1

x = (R + r * np.cos(U)) * np.cos(V)
y = (R + r * np.cos(U)) * np.sin(V)
z = r * np.sin(U)

# Plots the torus on both axes
ax1.plot_surface(x, y, z, rstride = 5, cstride = 5, color = 'dimgray',
 ↪  edgecolors = 'snow')
```

```
ax2.plot_surface(x, y, z, rstride = 5, cstride = 5, color = 'dimgray',
 ↪  edgecolors = 'snow')

# Setting plot titles
ax1.set_title('Torus')
ax2.set_title('Torus from above')

# Setting range for z axis in ax1
ax1.set_zlim(-3,3)

# Setting viewing angles
ax1.view_init(elev = 35, azim = 45)
ax2.view_init(elev = 90, azim = 0)

# Showing the plot
plt.show()
```



Notice that we have added some customization to the `plot_surface` command. Namely, we have set the color of the figure with `color = 'dimgray'` and of the edges with `edgecolors =`

`'snow'`. Moreover the commands `rstride` and `cstride` set the number of *wires* you see in the plot. More precisely, they set by how much the data in the mesh $[U, V]$ is downsampled in each direction, where rstride sets the row direction, and cstride sets the column direction. On the torus this is a bit difficult to visualize, due to the fact that $[U, V]$ represents angular coordinates. To appreciate the effect, we can plot for example the paraboiloid

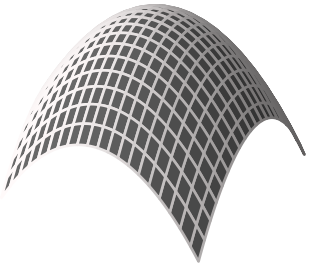
$$\begin{aligned} x &= u \\ y &= v \\ z &= -u^2 - v^2 \end{aligned}$$

for $u, v \in [-1, 1]$.

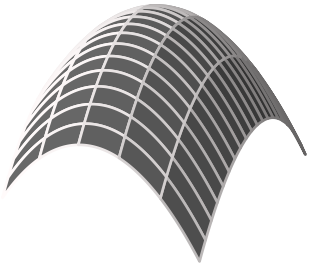
```
# Showing the effect of rstride and cstride

# Importing numpy, matplotlib and mplot3d
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

# Generates figure object of size 6 x 6
fig = plt.figure(figsize = (6,6))

# Generates 2 sets of 3D axes
ax1 = fig.add_subplot(2, 2, 1, projection = '3d')
ax2 = fig.add_subplot(2, 2, 2, projection = '3d')
ax3 = fig.add_subplot(2, 2, 3, projection = '3d')
ax4 = fig.add_subplot(2, 2, 4, projection = '3d')

# Generates coordinates u and v by dividing
# the interval (-1,1) in 100 parts
u = np.linspace(-1, 1, 100)
v = np.linspace(-1, 1, 100)

# Generates grid [U,V] from the coordinates u, v
U, V = np.meshgrid(u, v)

# Computes the paraboloid on grid [U,V]
x = U
y = V
z = - U**2 - V**2

# Plots the paraboloid on the 4 axes
```

```
# but with different stride settings
ax1.plot_surface(x, y, z, rstride = 5, cstride = 5, color = 'dimgray',
 ↪  edgecolors = 'snow')

ax2.plot_surface(x, y, z, rstride = 5, cstride = 20, color = 'dimgray',
 ↪  edgecolors = 'snow')

ax3.plot_surface(x, y, z, rstride = 20, cstride = 5, color = 'dimgray',
 ↪  edgecolors = 'snow')

ax4.plot_surface(x, y, z, rstride = 10, cstride = 10, color = 'dimgray',
 ↪  edgecolors = 'snow')

# Setting plot titles
ax1.set_title('rstride = 5, cstride = 5')
ax2.set_title('rstride = 5, cstride = 20')
ax3.set_title('rstride = 20, cstride = 5')
ax4.set_title('rstride = 10, cstride = 10')

# We do not plot axes, to get cleaner pictures
ax1.axis('off')
ax2.axis('off')
ax3.axis('off')
ax4.axis('off')

# Showing the plot
plt.show()
```

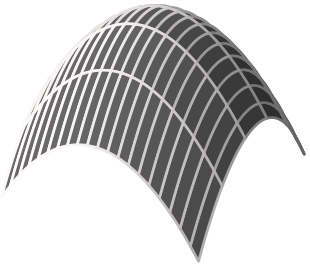
rstride = 5, cstride = 5



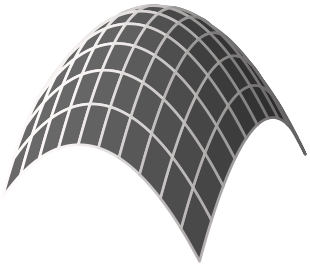
rstride = 5, cstride = 20



rstride = 20, cstride = 5



rstride = 10, cstride = 10



In this case our mesh is `100 x 100`, since `u` and `v` both have 100 components. Therefore setting `rstride` and `cstride` to 5 implies that each row and column of the mesh is sampled one time every 5 elements, for a total of

$$100/5 = 20$$

samples in each direction. This is why in the first picture you see a `20 x 20` grid. If instead one sets `rstride` and `cstride` to 10, then each row and column of the mesh is sampled one time every 10 elements, for a total of

$$100/10 = 10$$

samples in each direction. This is why in the fourth figure you see a `10x10` grid.

5.2 Plots with Plotly

As done in Section 2.4, we now see how to use `Plotly` to generate an interactive 3D plot of a surface. This can be done by means of functions contained in the `Plotly` module `graph_objects`, usually imported as `go`. Specifically, we will use the function `go.Surface`. The code will look similar to the one used to plot surfaces with `matplotlib`:

- generate meshgrid on which to compute the parametric surface,
- store such surface in the numpy array `[x,y,z]`,
- pass the array `[x,y,z]` to `go.Surface` to produce the plot.

The full code is below.

```
# Plotting a Torus with Plotly

# Import "numpy" and the "graph_objects" module from Plotly
import numpy as np
import plotly.graph_objects as go

# Generates coordinates u and v by dividing
# the interval (0,2pi) in 100 parts
u = np.linspace(0, 2*np.pi, 100)
v = np.linspace(0, 2*np.pi, 100)

# Generates grid [U,V] from the coordinates u, v
U, V = np.meshgrid(u, v)

# Computes the torus on grid [U,V]
# with radii r = 1 and R = 2
R = 2
r = 1

x = (R + r * np.cos(U)) * np.cos(V)
y = (R + r * np.cos(U)) * np.sin(V)
z = r * np.sin(U)

# Generate and empty figure object with Plotly
# and saves it to the variable called "fig"
fig = go.Figure()

# Plot the torus with go.Surface and store it
```

```
# in the variable "data". We also do now show the
# plot scale, and set the color map to "teal"
data = go.Surface(
    x = x , y = y, z = z,
    showscale = False,
    colorscale='teal'
    )

# Add the plot stored in "data" to the figure "fig"
# This is done with the command add_trace
fig.add_trace(data)

# Set the title of the figure in "fig"
fig.update_layout(title_text="Plotting a Torus with Plotly")

# Show the figure
fig.show()
```

```
Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): text/html
```

The above code generates an image that cannot be rendered in pdf. To see the output, see the link to the digital version of these notes. To further customize your plots, you can check out the documentation of `go.Surface` at this link. For example, note that we have set the colormap to `teal`: for all the pretty colorscales available in Plotly, see this page.

One could go even fancier and use the tri-surf plots in `Plotly`. This is done with the function `create_trisurf` contained in the module `figure_factory` of `Plotly`, usually imported as `ff`. The documentation can be found here. We also need to import the Python library `scipy`, which we use to generate a *Delaunay triangulation* for our plot. Let us for example plot the torus.

```
# Plotting Torus with tri-surf

# Importing libraries
import numpy as np
import plotly.figure_factory as ff
from scipy.spatial import Delaunay

# Generates coordinates u and v by dividing
```

```
# the interval (0,2pi) in 100 parts
u = np.linspace(0, 2*np.pi, 20)
v = np.linspace(0, 2*np.pi, 20)

# Generates grid [U,V] from the coordinates u, v
U, V = np.meshgrid(u, v)

# Collapse meshes to 1D array
# This is needed for create_trisurf
U = U.flatten()
V = V.flatten()

# Computes the torus on grid [U,V]
# with radii r = 1 and R = 2
R = 2
r = 1

x = (R + r * np.cos(U)) * np.cos(V)
y = (R + r * np.cos(U)) * np.sin(V)
z = r * np.sin(U)

# Generate Delaunay triangulation
points2D = np.vstack([U,V]).T
tri = Delaunay(points2D)
simplices = tri.simplices

# Plot the Torus
fig = ff.create_trisurf(
    x=x, y=y, z=z,
    colormap = "Portland",
    simplices=simplices,
    title="Torus with tri-surf",
    aspectratio=dict(x=1, y=1, z=0.3),
    show_colorbar = False
    )

# Adjust figure size
fig.update_layout(autosize = False, width = 700, height = 700)

# Show the figure
```

```
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Again, the above code generates an image that cannot be rendered in pdf. To see the output, see the link to the digital version of these notes.

License

Reuse

Citation

For attribution, please cite this work as:

> Fanzon, Silvio. (2023). *Lecture Notes on Differential Geometry*. https://www.silviofanzon.com/2023-Differential-Geometry-Notes/

BibTex citation:

```
@electronic{Danzon-Diff-Geom-2023,
    author = {Fanzon, Silvio},
    title = {Lecture Notes on Differential Geometry},
    url =
     ↪  {https://www.silviofanzon.com/2023-Differential-Geometry-Notes/},
    year = {2023}}
```

References

[1] C. Bär. *Elementary Differential Geometry*. Cambridge University Press, 2010.

[2] M. P. do Carmo. *Differential Geometry of Curves and Surfaces*. Second Edition. Dover Books on Mathematics, 2017.

[3] R. Johansson. *Numerical Python. Scientific Computing and Data Science Applications with Numpy, SciPy and Matplotlib*. Second Edition. Apress, 2019.

[4] Q. Kong, T. Siauw, and A. Bayen. *Python Programming and Numerical Methods*. Academic Press, 2020.

[5] A. Pressley. *Elementary Differential Geometry*. Second Edition. Springer, 2010.

[6] V. A. Zorich. *Mathematical Analysis I*. Second Edition. Springer, 2015.

[7] V. A. Zorich. *Mathematical Analysis II*. Second Edition. Springer, 2016.