BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

BACHELOR'S THESIS

Brno, 2020 Radek Polášek

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní obor **Angličtina v elektrotechnice a informatice**

Ústav jazyků

Student: Radek Polášek ***ID:*** 203160
Ročník: 3 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Různé přístupy k vizualizaci 2D a 3D objektů v počítačové grafice

POKYNY PRO VYPRACOVÁNÍ:

Semestrální práce se bude zabývat různými technikami používanými při vizualizaci 2D a 3D objektů. V úvodní části se autor/ka uvede rozdíly mezi vektorovou a bitmapovou grafikou a v krátkosti popíše pojmy jako např. Bézierovy křivky nebo vhodné parametrické křivky a funkce používané v inženýrské praxi. Úvodní část by bylo vhodné doplnit o krátký historický přehled. Poté se autor/ka zaměří na popis technik používaných pro vykreslování 2D a 3D funkcí ve specializovaném softwaru – nejlépe v matematickém softwaru jako např. Maple, MATLAB nebo Mathematica. V případě zájmu může autor/ka také popsat způsoby modelování objektů v CAD systémech. Semestrální práce by měla laikovi poskytnout základní představu o technikách a procesech, které způsobí, že se 2D a 3D objekty (zejména ty, které jsou popsané, resp. popsatelné matematickými funkcemi) vykreslí na monitoru počítače.

DOPORUČENÁ LITERATURA:

Dokumentace k softwaru Maple / MATLAB / Mathematica a literatura zabývající se v obecných rysech počítačovou grafikou.

Termín zadání: 6.2.2020 ***Termín odevzdání:*** 12.6.2020

Vedoucí práce: Mgr. Petra Langerová

doc. PhDr. Milena Krhutová, Ph.D.
předseda oborové rady

BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF FOREIGN LANGUAGES
ÚSTAV JAZYKŮ

VARIOUS APPROACHES TO VISUALIZATION OF 2D AND 3D OBJECTS IN COMPUTER GRAPHICS
RŮZNÉ PŘÍSTUPY K VIZUALIZACI 2D A 3D OBJEKTŮ V POČÍTAČOVÉ GRAFICE

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR AUTOR PRÁCE	Radek Polášek
SUPERVISOR VEDOUCÍ PRÁCE	Mgr. Petra Langerová

BRNO 2020

Abstract

This bachelor thesis aims to broaden people's knowledge of the visualization processes used in modern computer software to construct two and three-dimensional objects. First, some of the essential definitions and concepts of functions and their parameters are explained, which are then utilized in following chapters of this thesis to further explain additional processes that allow these functions to be graphically represented in a system of coordinates. The next chapter is focused on direct examples from specific computer software and what the user should pay attention to in order to graphically interpret functions in a correct manner. Additionally, practical example of more advanced object visualization is included. Lastly, differences between individual visualization programs are also explained.

Keywords

Function, Visualization, graphics, variable, graph, fractal, Matlab, Maple

Abstrakt

Cílem této bakalářské práce je rozšířit povědomí lidí v okruhu procesů, které jsou využívány moderními počítačovými programy, k vizualizaci 2D a 3D objektů. Postupně jsou vysvětleny některé nejzákladnější definice a koncepty funkcí a jejich parametrů, které jsou poté v následujících kapitolách této práce využity k vysvětlení pokročilejších procesů, které umožňují grafickou reprezentaci těchto funkcí v souřadnicových systémech. Další kapitola se pak zaobírá přímo příklady z konkrétních počítačových softwarů a tím, na co by uživatel měl brát zřetel, aby dosáhl úspěšné grafické reprezentace funkcí. Dodatečně je také demonstrován proces vizualizace poněkud komplikovanějšího objektu. Na závěr práce jsou nastíněny rozdíly mezi jednotlivými vizualizačními programy.

Klíčová slova

Funkce, vizualizace, grafika, proměnná, graf, fraktál, Matlab, Maple

POLÁŠEK, Radek. *Různé přístupy k vizualizaci 2D a 3D objektů v počítačové grafice.* Brno, 2020. Dostupné také z: https://www.vutbr.cz/studenti/zav-prace/detail/127165. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav jazyků. Vedoucí práce Petra Langerová.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Mgr. Petra Langerová for guiding me during the process of writing my thesis. I would also like to thank doc. RNDr. Michal Novák, Ph.D. for his thorough and patient guidance concerning the mathematical aspects of this thesis.

TABLE OF CONTENTS

1 Introduction

Modern mathematical and programming software is becoming increasingly widespread as more and more people engage with at least some form of it on daily basis. Whether it is part of their job description or academic study, they often may have very little understanding of how the processes that take place in the background of the software they are using. Processes that make it possible for programs to output wide spectrum of images and the structure of these processes will be among the main topics of this thesis.

In the first part we will go over the basic principles of computer graphics and most importantly mathematical functions that serve as the backbone of all the visualization in computer software. We will discuss the individual methods of turning these mathematical functions into graphs and images and describe some rather interesting phenomena of the visualization process. After the basics are discussed, our main focus for the rest of this thesis will be specialized visualization software. It is this very software that serves as an entry point into the world of visualization for vast majority of students and even businesses. And as such, specialized software, namely Matlab and Maple, will be the main focus of the second part of this thesis. We shall go over the basic principles we have already learned and demonstrate them in the selected visualization software with attention to the most common errors and phenomena that usually occur during the processes. This will be done for both Matlab and Maple for the purpose of highlighting the main similarities and the most striking differences. As there is an undeniable competition between the various available computer programmes for visualization, every one of them has its own unique approach to specific procedures while also sharing a considerable amount of similarities in other procedures. These differences and similarities will be the main point of the last part of this thesis.

2 Raster and vector graphics

Before we delve into the process of visualization, it is important to discuss the most important sections of this topic to improve our understanding of several key concepts. The first step is to distinguish between vector and raster graphics.

2.1 Raster Graphics

One of the two fundamental ways to graphically represent information in computer graphics is the raster method. This way of representation utilizes a network of points that are called pixels. These pixels have each their own x and y coordinate in a square grid along with specific colour value. By connecting these pixels we are then able to construct the entirety of the image. Quality of image generated by this method greatly depends on its resolution or rather on its pixels per centimetre (ppcm) value. This value tells us how many pixels there are in a centimetre; therefore, higher ppcm value equals higher quality of an image. Perhaps the biggest advantage of this method is how easy it is to edit a raster image and the realistic look it offers. Raster image files must contain all the information that is necessary to create the image (colour, position of all the pixels, etc.), and because of that, the size of these files can scale up quickly with higher resolutions and sizes.

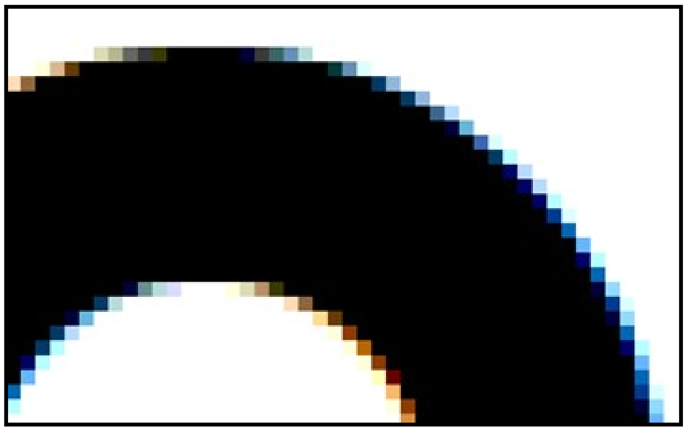


Figure 1: Example of a resized raster image

The most common formats for storing raster graphics include jpeg, png, bmp and gif. While jpeg is widely used, it does not support transparent parts and has lossy compression which can affect the quality with each compression of the image. If we are looking for formats with lossless compression, meaning formats of data that "*can be decompressed to exactly its original value*" (Mahoney, 2010, pg.1), we can use .gif, .bmp or .png, but even these have their own disadvantages.

2.2 Vector graphics

The second way of representation is vector graphics. Unlike raster, it does not utilize a square grid of pixels but rather complex mathematical equations. These equations define the shape and size of various lines (or polygons) along with their colour, orientation, line thickness, curvature (depending on their anchor points) and other values. The main advantage of this method over raster is the preservation of quality if we decide to change the size of the image. Thanks to the mathematical equations that form the curves, rescaling the image is achieved by a new calculation of these equations and the potential increase in size without any noticeable damage on its quality is therefore unlimited. Apart from higher quality, vector image files also tend to be smaller in size. This is due to them only containing the mathematical formulas that are essential for rendering of the image.

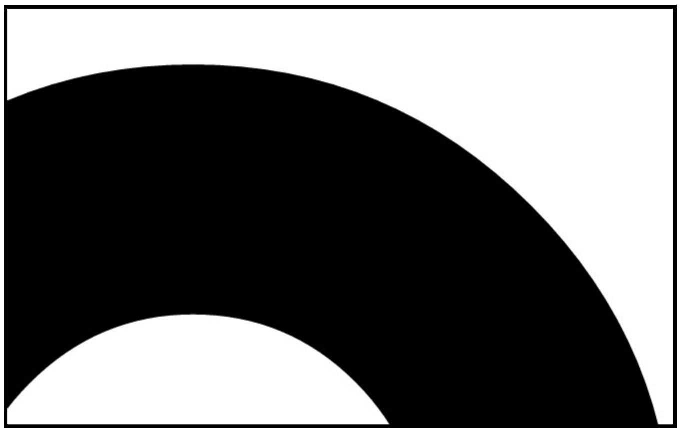


Figure 2: Example of a resized vector image

Most common format to use for vector graphics is .svg. It has the benefit that it can be edited and scripted in similar way to HTML while also having lossless compression with significant reduction in file size.

2.3 PDF

One of the more prominent file formats in modern computer graphics is the portable document format (pdf). The reason behind its widespread usage is primarily the fact that it preserves all of the formatting of a file, regardless of the reader’s operating system or what program is being used to read the file. We should know that pdf format supports the usage of both vector and raster graphics in its files. Exporting file formats into pdf can be realized by attempting to print the document and selecting a pdf printer as the designated device.

With the two main differences in graphics and formatting of files explained, we can now begin exploring different functions and processes that make the projection of two and three dimensional objects on our devices possible.

3 History

To get a good understanding of how computers turn series of numbers into lines and curves, we first need to understand some of the building blocks of this process. The first one being functions.

From the very beginning of advanced mathematics in ancient Mesopotamia and Greece, people were working with basic principles of functions at that time, but there was no notion of defined variable or explicitly defined functions. The notion of function did not arise until much later in late seventeenth century, when it was first properly defined. It is believed to be first used by Gottfried Wilhelm von Leibniz (1646-1716), German mathematician, physicist and philosopher. It is him and Sir Isaac Newton who are attributed with the creation of calculus and therefore mathematical analysis and its study of functions. This may well be the origin of modern concept of a function, but it was not until nineteenth century that the modern meaning of function was properly defined by Johann Peter Gustav Lejeune Dirichlet (1805-1859), who in his work on proving convergence conditions in Fourier series states*: "to any **x** there corresponds a single finite **y**"* (Dirichlet ,1829).

4 Function

Now that we know about the origin of functions, we can start delving into how they operate and how to utilize them in computer graphics. We can interpret values of a function in the form of table with two columns, one column being inputs of the function and the other being its outputs (See table 1).

Input	Output
1	3
2	6
3	9

Table 1: Representation of function as a table

From what we see, the conclusion to this function could be that output is always three times the input. The mathematical transcription of this function would then be:

$$f(x) = 3 \cdot x \tag{1}$$

Where "$\boldsymbol{f}$" represents the name of the function, "$\boldsymbol{x}$" is a parameter (input) of our function and the part after equal sign tells us what the function does with our parameter or parameters. Now we can order the input and output into pairs of (x, *f(x)*). This will allow us to construct a graphical representation of our function, a graph. If we obtain several of these pairs, we can start placing them as a set of dots into a coordinate system. By plotting one of the coordinates (x) on the x-axis and the second coordinate (*f(x)*) on the y-axis, we construct a point. Plotting more or all values from the function domain yields us a graph of the function (See Fig. 3).

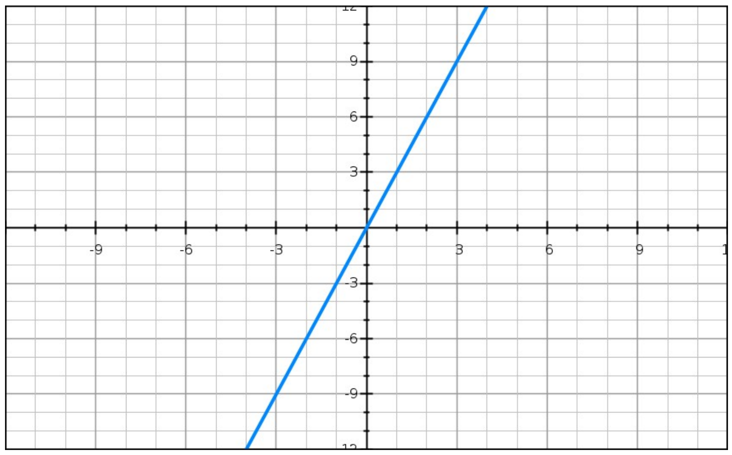


Figure 3: Graphical representation of a function

However, the table is not sufficient if we want to define a function properly. That is because every function is determined by three properties: domain, codomain and by the rule of assigning elements of the codomain (or rather range) to elements of the domain. This tells us what relationship there is between *x and f(x)* and can be achieved in several ways, first of which is simply listing the different values of a function in a table (similar to table 1). Another and for us the most important way is by using a formula. We have already utilized this way of representation previously while delving into the basics of how functions operate. The third way in which a function can sometimes be represented is a graph of that specific function from which we can obtain its values.

4.1 Domain

Every function has its domain, which represents a certain set of values that can be substituted for the *x* variable of that function, and will output the *f(x)* value. For example if we consider a function:

$$f(x) = \frac{1}{x} \tag{2}$$

Then *x* cannot be zero because the function would not output a real value. We can therefore say that the domain of this function is: $D(f) = \{x \in \mathbf{R} : \mathbf{x} \neq 0\}$. It is also important to note that every input value must be assigned to a unique output value; otherwise the function would not know which output value to provide us with if two were assigned to a single input.

4.2 Range and codomain

While codomain is the set of all possible values that the function can output, range values of a function are a subset of the codomain and represent a set of all the actual values that the function outputs. We can use the previously used function of $f(x) = 3 \cdot x$. Graph of this function (See Fig. 3) helps us to see that its range is not bounded and consists of all real numbers, therefore it can be written down as: H(f) = **R**. If we were to change the transcription for this function to $f(x) = |3 \cdot x|$, then the range of values would change to: H(f) = $\langle 0,\infty)$, since the output can no longer be a negative value. In other words, domain defines the numbers on the input of a function while range determines the output numbers.

4.3 Implicit functions

So far, we have only discussed functions with explicitly given variable, so called "explicit functions", i.e. functions of the form ***f(x) = 0***. The second way in which a function can be given is called "implicit". An example of such function can be the equation of circle with centre in the origin and with radius of one.

$$x^2 + y^2 - 1 = 0 \tag{3}$$

We can see that the variable is not given independently in this case but rather in a form of ***F(x, y) = 0***. It is possible to convert implicit functions into their explicit forms, but it is often not recommended as the resulting function is usually too complex and can branch into multiple results. If we were to transform our circle function, the resulting explicit format would look like this:

$$f(x) = \pm\sqrt{1 - x^2} \tag{4}$$

Note that there are now two branches of this function: positive and negative which is one of the reasons, why it is not recommended to convert between implicit and explicit form of more complicated functions.

4.4 Function of more than one variable

It is also possible to have a function with multitude of input parameters. For our example we can take a look at the power function:

$$f(x, y) = x^y \tag{5}$$

Function in this example takes the two input values and gives us the result which is x to the power of y. If we wanted to represent this function with a table, it would have three columns instead of just two. In other words: we can interpret functions of more than one variable (n variables) as a number that we assign to a set of n numbers. While functions of one variable plot their graphs into two dimensional planes, when dealing with functions of more variables that require two inputs and produce a single output, we need to pay attention to the fact that the graph of that function will now be a surface in at least three dimensional space as opposed to two dimensional plane. We can, of course, project graph of that function into just two dimensional plane. This attribute is quite important for different computer programs for graphical visualization as there are different commands to correctly display different functions depending on the environment. A good example of a situation where the display environment would make a tremendous difference is the multivariable function of:

$$f(x, y) = \sin(\mathrm{x}) \cdot \cos(\mathrm{y}) \tag{6}$$

If we choose to display this function in three dimensional space, we would get a different graphical representation than if we were to use the contour graph variant which utilizes a horizontal cross-section of the graph and assigns different colour values (for example) to individual data points depending on their output value (see Fig. 4).

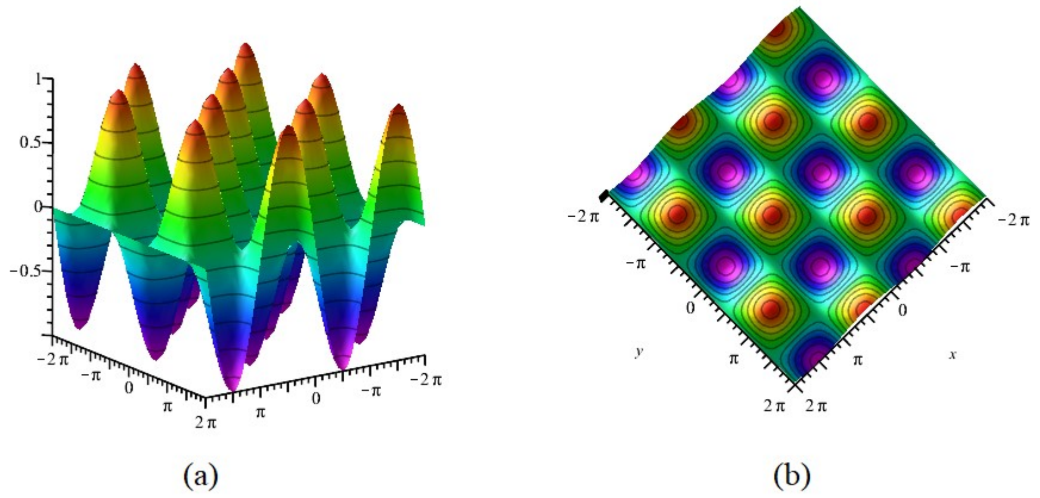


Figure 4: Three dimensional plot (a), Contour plot (b)

We will now have to distinguish between several types of function mapping. These are vital as they tell us in which category of numbers is the function operating.

4.5 Real function of one real variable

We can approach this type as a simple function in which every real number $x \in X$ (x belongs to group X which is also domain of that specific function) is associated with one real number $y \in Y$. Therefore (if both X and Y consist of only real numbers) we can denote this type of mapping as $f\colon \boldsymbol{R} \to \boldsymbol{R}$. This notation tells us that both domain (X) and codomain (Y) of the function are composed of real numbers.

4.6 Complex function of one real variable

Similar to previous type of function mapping, this one represents functions that map variables from a set of real numbers **R** into a set of complex numbers **C**, denoted as $f\colon \boldsymbol{R} \to \boldsymbol{C}$.

4.7 Complex function of complex variable

This type of function represents a case in which both D(f) and H(f) of a function are complex numbers, therefore: $f\colon \boldsymbol{C} \to \boldsymbol{C}$ and for an arbitrary $z \in \boldsymbol{C}$ ($z = x + jy$), we can say that $w = f(z) = u(x, y) + jv(x, y)$, where u and v are two real functions of two real variables.

It is important to distinguish between the different types of function mapping. Computer software utilizes the different types to correctly draw graphs and eliminate potential errors that could be caused by using the wrong one.

5 Coordinate systems

Now that we learned it is possible to project results of a function into a coordinate system, we need to know about the different types that we can utilize. We shall start with perhaps the most commonly used one in modern mathematics, the Cartesian coordinate system.

5.1 Cartesian coordinate system

This system breaks space into four quadrants separated by two perpendicular axes, horizontal X-axis and vertical Y-axis. Intersection of these two axes is known as the point of origin and has the value of 0. We calculate the location of an object by assigning respective x and y coordinates, depending on the distance from the point of origin alongside the two axes.

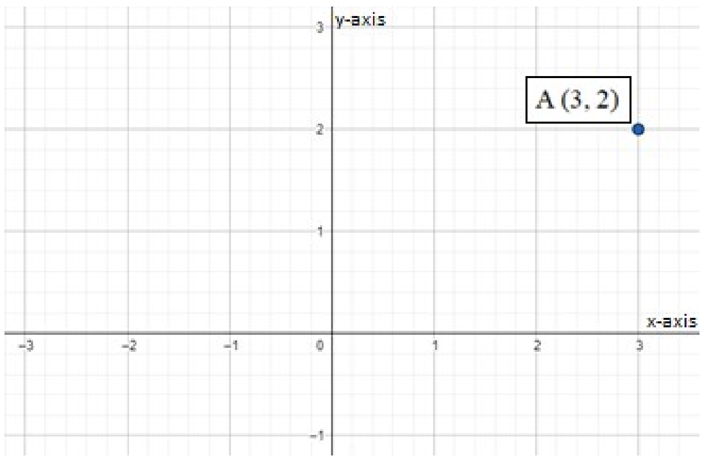


Figure 5: Cartesian coordinate system

For example an object A with *x* coordinate of 3 and *y* coordinate of 2 will be positioned in the first quadrant because both of its coordinates have a positive value. The object can then be rewritten as: A (3, 2) and will be projected as in the previous figure (See Fig. 5). When we work in 3D environment, Cartesian coordinate system is adjusted by adding the third axis, the z-axis which is placed perpendicular to both x and y axes and intersects with the point of origin. Objects in the system are now denoted by one additional value: z (depth) and can be described by this transcription: A (3, 2, 4).

Another detail we should pay attention to is the scale of generated graphs. While in our example identical for both x and y axes, it is by no means a rule. Certain programs may automatically adjust the scale of axes to better display the entirety of the graph. This information does not affect just the Cartesian coordinate system, but the other systems too.

5.2 Polar coordinate system

Another option for projecting functions into graphical environment is to utilize a system of different radii and angles, also known as the polar coordinate system. It uses a radius value r and an angle that is denoted by φ or ϑ. Polar grid is constructed from 2π rad or 360° (Depending on whether we are working with radians or degrees) angle around a point of origin that is denoted by intersection of two perpendicular axes.

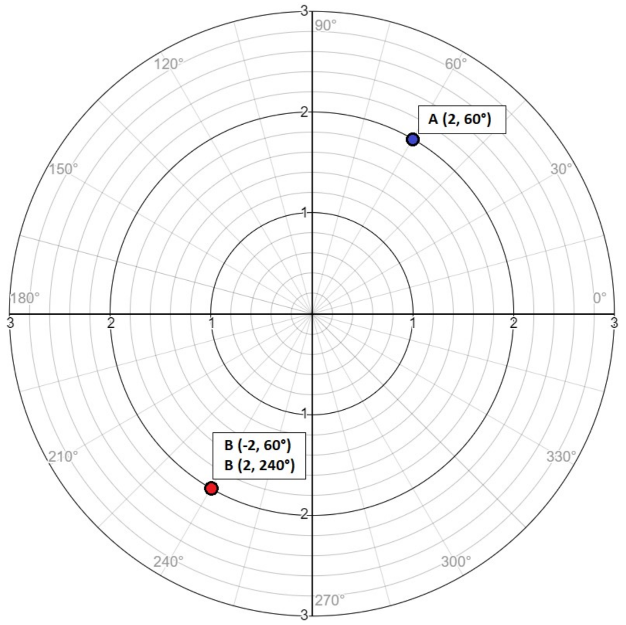


Figure 6: Polar coordinate system

If we are given an object with $r = 2$ and $\vartheta = 60°$, it would project as A (2, 60°). We can also be given negative r value in which case we simply add 180° (or $\pi\, rad$) to the value of ϑ to accommodate for the negative value of r (see Fig. 6). It is of course possible to convert a set of polar coordinates to their Cartesian counterpart. If we consider an object A (r, φ) in polar coordinate system, we can easily obtain x and y Cartesian coordinates using these equations:

$$x = r \cdot \cos\varphi \tag{7}$$
$$y = r \cdot \sin\varphi \tag{8}$$

5.3 Spherical coordinate system

Spherical coordinates can be utilized to denote objects in three dimensional space using distance to the object from point of origin that is held between three axes (each one perpendicular to the other two), angle ϑ ($\vartheta \in \langle 0;2\pi\rangle$) which is the deviation r from the z axis and angle φ ($\varphi \in \langle 0;\pi\rangle$) that denotes deflection from the x axis. As we can see, this system is very similar to the polar coordinate system in its structure (see Fig. 5).

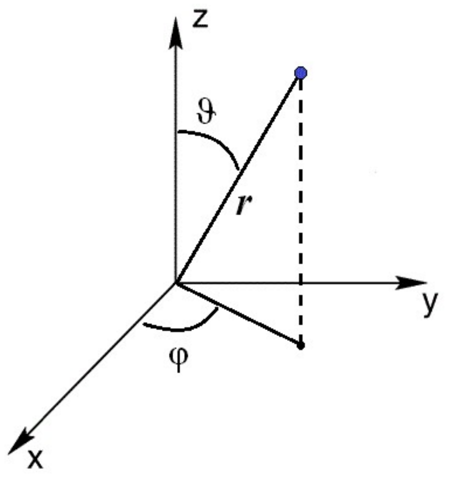


Figure 7: Spherical coordinate system

It is of course possible to convert between spherical and Cartesian coordinates. To do so, we will be using a set of three equations to obtain x, y and z Cartesian coordinates:

$$\begin{aligned} x &= r \cdot \cos\varphi \ \cdot \sin\vartheta \\ y &= r \cdot \sin\varphi \ \cdot \sin\vartheta \\ z &= r \cdot \cos\ \varphi \end{aligned} \tag{9}$$

There are various different examples of coordinate systems besides these three, but their main application is usually outside the field of mathematics and it is therefore unimportant to delve deeper into their definition. Most prominent examples of such systems would include cylindrical (which is utilized mainly for computing electromagnetic fields) or logarithmic-polar coordinate system. Cartesian system remains among the most used coordinate systems and will be our primary subject through this thesis.

More about fundamentals of functions can be found in *Calculus* (SPIVAK, 1994) or in *Matematika 2* (SVOBODA, VÍTOVEC, 2014)

6 Tools of visualization

With all the basic principles of graphics and functions explained, it is now time to move onto the essentials of the main procedure that turns typed in data into lines and curves. Before we move to comparison between various computer programs and their approaches to visualization, we can define some of the more basic functions that create these processes.

6.1 Interpolation

If we are given a set of data points of a function defined by a table (for example), they do not accurately represent how the complete function would look. Without a way to connect these data points, we would not be able to correctly identify what values are in between our given points. We could utilize the process of linear interpolation and connect these points with straight lines, but this approach introduces large amount of error. To interpret these values in a more efficient way, we can use the process of polynomial interpolation which connects the given data points with the use of polynomials. Depending on how many data points we are working with, the correct degree of polynomial has to be used. When we only have two data points available, we will be using a first degree polynomial which has the form of a line. Polynomial will have the form of quadratic parabola if it is of second degree over three points, while degree 3 and up means that polynomial will be a cubic curve over 4 or more points (see Fig. 8).

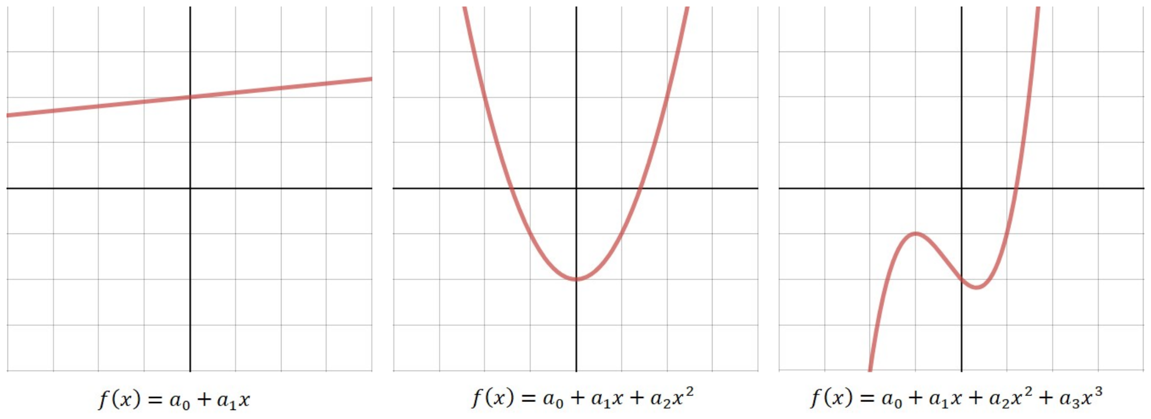


Figure 8: Examples of possible polynomials

If we are working with larger amount of data points, this approach can then be summed up into the following equation for *n+1* of points:

$$f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n \tag{10}$$

More information about the process of interpolation can be found in the lectures on *Data Fitting* (KUTZ, 2006) or in the *WolframMathworld Web resources* (WEISSTEIN, 2019)

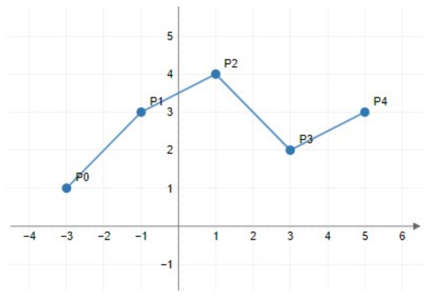
6.2 Spline interpolation

With the usage of polynomials, the spline interpolation connects data points in a graph of function. There are several variants of spline interpolation and we will be mainly focusing on cubic spline interpolation as it provides the most accurate approximation out of these methods, superior to that of linear or quadratic interpolation in particular if the function has more abrupt and unpredictable changes in its behaviour (see fig. 9). Mathematical distinction between specific types of the spline interpolation can be demonstrated on the equation for polynomial. If we were to consider a degree 1 polynomial denoted by the following equation:

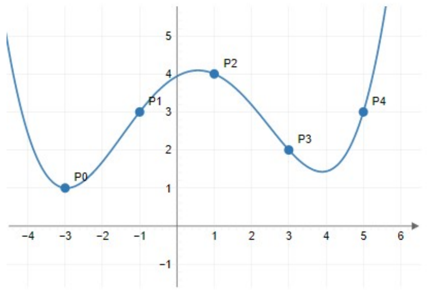
$$f(x) = a_0 + a_1x \tag{11}$$

Then the spline interpolation method will be linear and individual points in the coordinate systems will be connected by straight lines. If we move our polynomial up one degree, we arrive at quadratic spline interpolation and degree 3 polynomial will then be used for cubic spline interpolation.

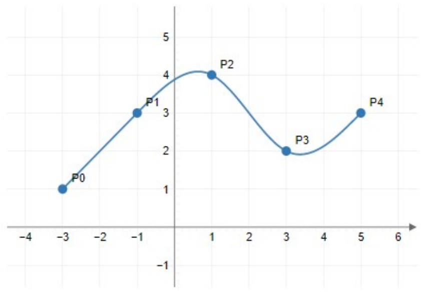
x	$f(x)$
-3	1
-1	3
1	4
3	2
5	3



Linear spline interpolation



Quadratic spline interpolation



Cubic spline interpolation

Figure 9: Different types of interpolation

6.3 Bezier curves

A special category of construction curves utilizing a set of control points are called the Bezier curves. Depending on the number of control points we can again distinguish between several types of these curves. Using *n* to denote the number of control points gives us the necessary information to distinguish between linear, quadratic and cubic Bezier curves. The control points can then be further divided into end points and controlling points. Two of the end points denote starting and end position of the curve and the remaining points control the shape of the curve. Another thing to notice is that although all the control points usually do not lie on the curve itself, all of them are confined inside a space known as convex hull, which is the smallest possible area containing all of the given points.

6.4 Fractals

When we mention snowflakes, spiral shells or ferns, there is no striking similarity between these objects at the first sight, but if we examine them closer, we find out they are all in fact fractals. Fractals represent a unique shape composed of seemingly infinite number of nearly exact shapes. To better understand this description, we will use the Mandelbrot set as an example. This fractal is aptly named after mathematician Benoit B. Mandelbrot, who stood behind the first mathematical description of a fractal and later computer visualization.

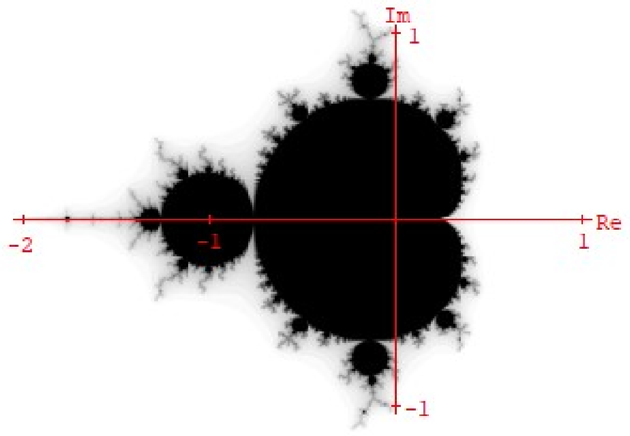


Figure 10: Mandelbrot set (Bourke, 2002) over real and imaginary coordinates

If we were to magnify any of the parts of this fractal, we would see infinitely expanding area similar in shape to that of a complete Mandelbrot set. This is due to the infinitely expanding length of its perimeter. This fractal image can then be generated by the following equation:

$$z_{n+1} = z_n^2 + c \tag{12}$$

where $z \in \mathbf{C}$ and $c \in \mathbf{C}$. If we were to apply this equation to each and every pixel (representing all the different c values), in a coordinate system plane, repeatedly and the z_n gets exponentially bigger, then the value c does not belong into the Mandelbrot set. If it, however, remains small, then that pixel is part of the set. If we then mark all of the points that do belong into the Mandelbrot set, we get a precise map of it (see Fig. 10).

Perhaps more easily demonstrated example of a fractal is the infinitely folding fractal also known as the Heighway Dragon Curve, which was discovered in 1966 by two physicists: J. Heighway and his colleague W. Harter (Tabachnikov, 2014). This particular curve represents a process of possibly infinite amount of iterations that lead to the creation of the fractal. The whole process begins with a single line. This line is then transformed during the first iteration into two connected lines holding a 90° angle above the original line, with their outer ends in place of the original line's ends (This can be demonstrated by folding a straightened rope into two equal parts and then creating a 90° angle between those parts). In the next iteration, we would again create an angle above every straight line with an alternating pattern of right and left. This process is then repeated in each of the following iterations and with each one of them, we can see the fractal more and more clearly:

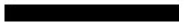
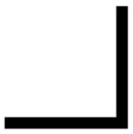
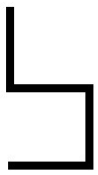
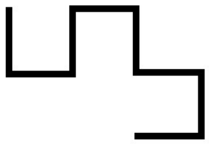
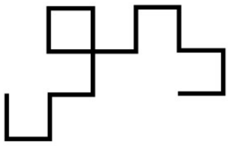
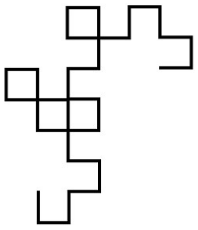
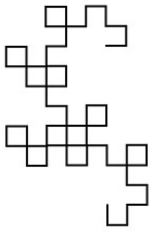
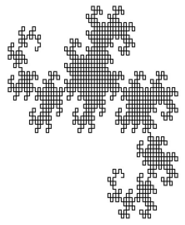
			
0	1	2	3
			
4	5	6	12

Table 2: Different iterations of the *Heighway Dragon* creation process

If we take a closer look at this process, we can identify that it is identical to unfolding a stacked strip of a paper, where with each step of the unfolding the previous shape is repeated at a 90° angle from the previous step. In ideal conditions, this process can be repeated indefinitely and what is rather interesting to observe is that the distance between the two ends of the curve always remains the same, no matter the iteration.

7 Visualization in specific software (Maple)

There is a broad range of different software that can simulate visualization processes by converting commands with functions into their respective representations. Each one of them utilizes slightly different set of commands to do so. In our example of how these types of software operate with functions we are firstly going to use Maple, mathematics-based software developed by Maplesoft™. Important part of each software that can be used to visualize functions is the form in which we feed the program commands. It is important to uphold the exact structure of individual commands; otherwise the software will not visualize the correct representation of the desired function or will simply not visualize it at all.

Maple allows us to use multiple visualization commands to graphically depict functions depending on type of the function and environment in which we want the function to be represented. We will begin with a command that allows us to depict a function in a two dimensional plane: plot.

```
plot(f, x)
```

As we can see, this command consists of several parts. The command itself and the parameters in brackets, where f symbolizes the function that we want to visualize and x is the independent variable of the function. Consider the first function from the section about functions (1). If we wanted to depict that function with domain of real numbers ranging from -10 to 10, using the plot command, then the command would have the following form:

```
plot(3*x, x=-10..10)
```

We can notice that although we did not specify the range of the *y* axis, the program automatically dealt with that problem and assigned a scale to it. This is one of the command parameters that do not have to be specified in order for the command to work properly (see Fig. 11a). There are, however, parameters and methods of visualization that have to be typed into the command window in order to obtain the correct version of what we are trying to visualize. We can demonstrate this problem on the following example where we try to use the same command to visualize a graph of complex function:

```
plot(3*x + 2*I, x = -10 .. 10)
```

In this equation, the letter **I** represents the imaginary unit of a complex number, while **x** is the real part of this number. Immediately after we run this command, Maple does not display the expected graph, but instead an error message with the following text: "*Warning, unable to evaluate the function to numeric values in the region; complex values were detected*". This is caused by the fact that the plot command serves the purpose of creating a two-dimensional plot of real values, so if those are not provided, the software has a list of errors to reach for if it finds any major inconsistency in the code such as this one. With the knowledge that Maple makes use of different commands for different types of graphs, we can venture into its documentation and find out what commands are used to display them. Programs such as Maple often include broad range of different plot commands, each specifically focused on narrow spectrum of functions In the next part we will go over some of the more prominent examples.

7.1 Different types of plot commands in Maple

If we want to construct graph of a function, it is crucial to first know some of the graph's attributes. Whether we are projecting the function into two-dimensional plane or three-dimensional space, whether the function operates with real numbers or complex numbers, in which way is the function denoted and what coordinate systems do we want to use. All of these attributes determine which of the different commands are available to be used. If we want to visualize a real function of a single variable, we can use the previously discussed "plot" command. If we are for example working with implicit function (See chapter 4.1), our command of choice will be the following:

```
implicitplot(f, x, y, options)
```

Similar to the basic plot function, the implicit variation requires the same parameters of function, domain and range. In addition the **options** parameter allows us to influence the visual outcome in additional ways such as scaling, using different coordinate system (see Fig. 11b), changing colours of various parts of the graph and naming the individual function plots. Now if we move out of the real numbers category and transition into visualization of complex number functions, we will need to use a different command to generate the graph:

```
complexplot(f, t=a..b, options)
```

This command is used to create a two-dimensional plot of a complex function **f**, in which the real part of the complex number is associated with the x-axis and imaginary part with the y-axis. The parameter **t** is the parameter of that function over the range of **a** to **b**.

So far all of these commands operate within the Cartesian coordinate system by default. We could of course change that by utilizing one of the coordinate options in parameter part of the different commands, or we could simply use commands that directly serve this purpose. As an example we can use the command "**polarplot**" which allows us to directly generate graphs in polar coordinate system.

```
polarplot(f, φ=a..b, options)
```

As in the previous commands, **f** is the function we want to plot, this time in the form of r(φ). This entry then produces a curve defined by [r(φ), φ], where r stands for radius and φ is the angle. The range argument is optional and if it is not given, then the default range will be from 0 to 2π (See Fig. 11c).

We can now start discussing the commands for graphs of functions in three-dimensional space. It is vital to know if we want our function visualized in 2D or in 3D environment because the resulting graphs vastly differ from one another and the interpretation could therefore not be what we were looking for. If we have a function that is to be projected into a three-dimensional environment, we need to reach for a different type of plot command:

```
plot3d(f, x=a..b, y=c..d)
```

Where the parameter **f** represents the expression that we are visualizing with variables of **x** and **y**, while the two additional parameters represent the range over which the graph is constructed. If we decide to not provide the range parameters, default values of -2π to 2π will be chosen for trigonometric functions and -10 to 10 for other entered functions (See Fig. 11d).

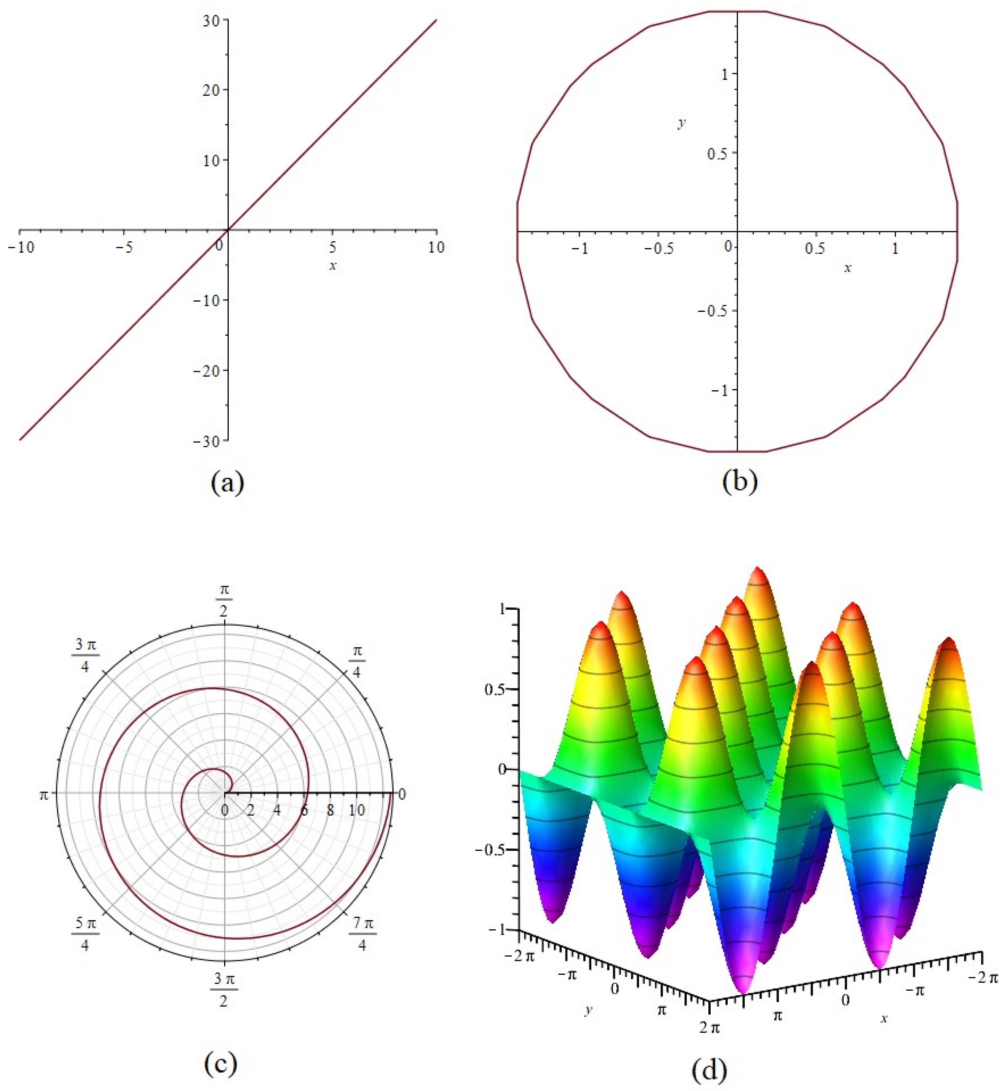


Figure 11: Resulting graphs of different Maple plot commands

(a) $\text{plot}(3\text{x}, \text{x} = -10\,..\,10)$ (13)

(b) $\text{implicitplot}(\text{x}^2 + \text{y}^2 = 2, \text{x} = -2\,..\,2, \text{y} = -2\,..\,2)$ (14)

(c) $\text{polarplot}(\vartheta, \vartheta = 0\,..\,4\text{Pi})$ (15)

(d) $\text{plot3d}(\sin(\text{x}) \cdot \cos(\text{y}), \text{x} = -2\text{Pi}\,..\,2\text{Pi}, \text{y} = -2\text{Pi}\,..\,2\text{Pi});$ (16)

Even though all of these commands are structured in a largely similar way, every one of them has its own set of requirements that need to be fulfilled. As we have already learned before, if any of these requirements are not met, the program will generate an error, warning us about a fault in the code. We will go over some of these error messages in the next part.

7.2 Examples of error messages in Maple

Since we should now be familiar with the basic usage of commands in maple, we can discuss what discrepancies in these commands cause specific error messages to occur. In the table below you will find examples of the most common ways in which commands are entered incorrectly along with their corresponding error message and the correct form of that command. The second column contains a brief explanation of the error message.

`plot(3*x`
`Error, unable to match delimiters`
 	
`plot(3*x)`
	One of the basic syntax errors warns us that the delimiters are not entered in correct pair form. Our example can be corrected by ending the bracket

`complexplot(3*x + 2*i, x = -10 .. 10)`
`Warning, expecting only range variable x in expressions [Re(3*x+2*i), Im(3*x+2*i)] to be plotted but found name i`
 	
`complexplot(3*x + 2*I, x = -10 .. 10)`
	When working with complex numbers, it is important to uphold the correct form for denoting the imaginary part (Maple uses uppercase **I**, not lowercase i)

`plot(3*x + 2*I, x = 2 .. 3)`
`Warning, unable to evaluate the function to numeric values in the region; complex values were detected`
 	
`Complexplot(3*x + 2*I, x = 2 .. 3)`
	We can immediately notice that although we are trying to plot a complex function, we are using the wrong command to do so. This problem can be fixed by using the correct type of plot function

`plot(3*x, y = 2 .. 3)`
`Warning, expecting only range variable y in expression 3*x to be plotted but found name x`
 	
`plot(3*x, x = 2 .. 3)`
	Error that occurs when we try to plot a function with variables that had not yet been defined instead of one that has been defined (x instead of y in this example)

Table 3: Examples of different plot-related error messages in Maple 2019.2

Although Maple and similar software tends to warn us if we are attempting an incorrect operation, there are some cases in which we must be at least partly familiar with the correct outcome of a function to identify if the program is displaying the correct solution. One of the best examples of this problem could be discontinuous functions, in other words: functions that are not represented by a continuous unbroken curve. If we were to plot a function like this, there would more than probably be large deviations around the discontinuity points of that function or so a connection between two points of discontinuity (See Fig. 12a).

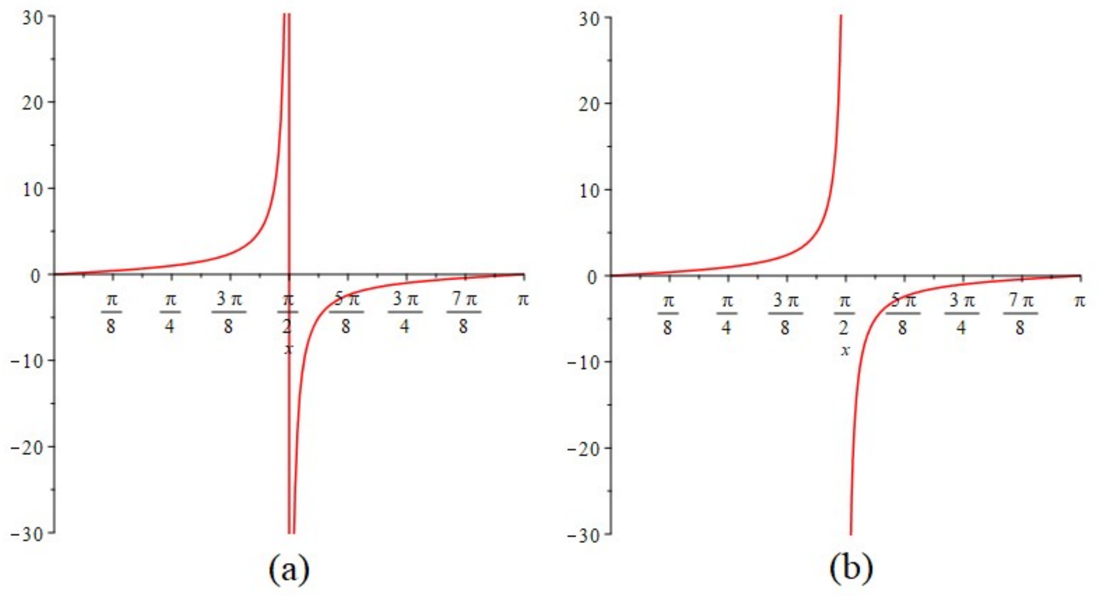


Figure 12: Software representation of tan(x) **(a)** and real tan(x) **(b)**

As we can see from the figure above, software does not correctly acknowledge the discontinuity and connects the two points with a line. There is of course a way to prevent this from happening with some additional parameters when entering the command, namely then entering the parameter "discont" and setting its value to **true**:

```
plot(tan(x), x = 0 .. Pi, discont = true)
```

This alteration of the command is then what allows us to arrive at the actual form of the tangent function (See Fig. 12b). Additionally, if we take another look at the circle graph from previous page (See Fig. 11b), we can notice that although the function is that of a circle, the resulting image is quite angular. This property is influenced by the grid parameter of plot command and is tied to interpolation (See chapter 6.1). The grid parameter can be entered in the following manner:

```
implicitplot(x^2 + y^2 = 2, x = -2 .. 2, y = -2 .. 2, grid = [g, g])
```

The values of **g** determine the density of the interpolation grid. With increasing values of **g** we obtain an object that more closely resembles a perfect circle. Another example of possible outcomes could be the problem of two-dimensional or three-dimensional environment. If we were to project the following function of: **x^2 + y^2 = 20**, into two-dimensional plane, the outcome can highly different than if we were to construct its graph in three-dimensional space.

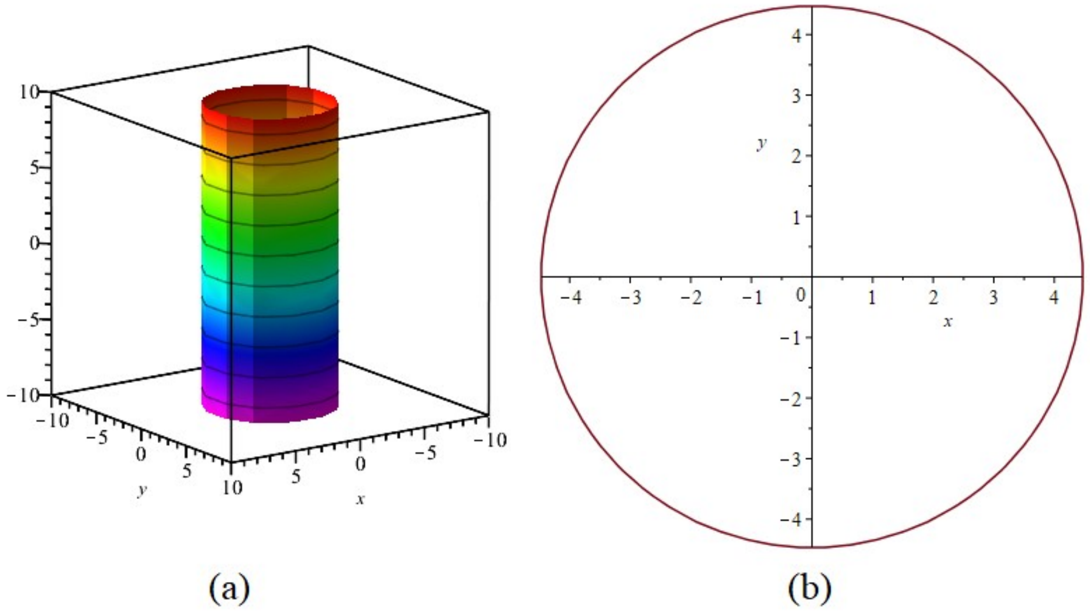


Figure 13: Function with the same transcription in space **(a)** and in 2D plane **(b)**.

As we can see from the figure above, if we decide to use a command to project this function into three-dimensional space, the resulting graph is that resembling of a cylinder, whereas using a different command and projecting it into two-dimensional plane yields us a graph of a circle. Since there is nothing inherently wrong with both ways in which this function can be visualized, it is up to us to choose the right procedure to suit our needs such as domain and range. All of the different command examples used above were retrieved from the official *Maple 2019.2 Help System* which allows users to find additional information about all the commands and options that Maple contains.

7.3 Other available software and differences

Where previously we utilized Maple to learn about the format of basic visualization commands, we will now draw from different software to help us compare between them and to understand their varying approaches to similar tasks. For the purpose of this thesis, we will be mainly comparing features and command libraries of Maple and Matlab, as they are both widely known and used. Before diving into specific procedures, it is important to shortly introduce the software.

8 Matlab

Since we have already explained the basic principles of how visualization software operates with commands containing our input, we can use that knowledge to compare between different variants of visualization in the form of different software that we use.

According to its creator, Matlab is *"a full-featured technical computing Environment"* (Moler C., Mathworks®, 2004). It is a product of *the Mathworks, Inc*. with widespread use through different academic institutions and business ventures. One of the many features this software provides is the ability to calculate input data, draw out visual representations of the data and manipulate the results. Apart from all these features, Matlab's working environment operates and presents itself in ways rather similar to those of programming languages rather than acting like an ordinary computing program. With such a wide variety of uses spanning across computation of mathematical problems, analysis and visualization of data, programming, and modelling & simulation, Matlab is arguably one of the most versatile technical environments on today's market. Once again we will first go through the most basic of visualization commands to demonstrate how Matlab converts numbers into graphical data.

8.1 Matlab's approach to visualization

Matlab operates as a computing language that is able to manage programming, mathematical computations and, most importantly, visualization of the computations. As a computing language it contains significant amount of similarities when compared to different available programming languages and as one of the most widely used computing languages it is also a good sample to demonstrate larger amount of some of the basic principles that these programs operate on. We can begin by perhaps the most significant principle: indexing. Indexing is the foundation of a large variety of commands that operate with arrays or matrices. We can demonstrate indexing on an example of a matrix below:

$$\boldsymbol{A} = [\mathbf{1\ 2\ 3\ 4};\ \mathbf{5\ 6\ 7\ 8};\ \mathbf{9\ 10\ 11\ 12}] \tag{17}$$

Using this command we defined matrix A consisting of three rows and four columns. Each row end is marked by a semicolon and its length depends on the quantity of numbers in-between two semicolons. After creation, the matrix A has the following form:

$$\boldsymbol{A} = \begin{bmatrix} \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} \\ \mathbf{5} & \mathbf{6} & \mathbf{7} & \mathbf{8} \\ \mathbf{9} & \mathbf{10} & \mathbf{11} & \mathbf{12} \end{bmatrix} \tag{18}$$

Now if we wanted to work with only few selected numbers of this matrix, we need to know their positions. This is where indexing takes over. Utilizing Cartesian coordinate system (Chapter 5.1), we are able to assign a coordinate pair to each of the matrix's positions. The only thing we need to do is determine our starting position which (in the case of matrices) is the left-most coordinate in the first row. This position will therefore be marked as (1, 1), since it marks the first row and the first column. If we wanted to include number on this position in any further equations or in different commands, we simply need to know its position indexes. For example, if we needed to add the value in the second row and the second column to a variable *v*, following command sequence would be the solution:

$$\begin{aligned} &\boldsymbol{v} = \mathbf{1} \\ &\boldsymbol{r} = \boldsymbol{v} + \boldsymbol{A}(\mathbf{2}, \mathbf{2}) \end{aligned} \tag{19}$$

Using this command, we took the number with indexes of (2, 2) from our matrix ***A*** and after adding its value to the previously defined variable, we saved the new result into ***r***. Good thing to notice here is how the index numbers operate. We can imagine that the number in first row and the first column position is our point of origin as it would be in the Cartesian coordinate system. Its index numbers however begin from 1 instead of zero as they would in the point of origin or in vast majority of programming languages. This is one of the most prominent distinctions between fundamentals of Matlab and different programming languages while at the same time a huge similarity with other computing languages such as Maple. This technique is called 1-based indexing (as opposed to 0-based indexing) and is implemented mainly because of how matrix operations work. If we apply this knowledge on our previous equation, we can safely compute the result saved in variable ***r*** which would, in this case, be **7**.

Our primary focus in most of the forthcoming examples shall be graphs, so it would be useful to know how to provide them with parameters. In Matlab, we are capable of generating and saving entire arrays of numbers into a single variable. Consider variable ***x*** to which we are going to generate the domain for our graph. If we want the domain to span from -2π to 2π, the easiest way is to use this three parameter command:

```
x = -2*pi:pi/100:2*pi;
```
(20)

We can see the three parameters (separated by colons) in the following order: minimum, step size and maximum. With the help of this command, we have now managed to insert around 400 values into the variable ***x***, effectively constructing our domain. The next step is to define the range. For the purpose of our example, our goal will be to construct a graph of the cosines function. Utilizing the following command, we are able to define ***y*** as cosine values of variable ***x***:

```
y = cos(x);
```
(21)

With both these variables set up, our requirements for construction of a graph are fulfilled. We can do so with the following command:

```
plot(x,y)
```
(22)

This is the most basic form of a plot function. It utilizes our two previously defined arrays of numbers and constructs a 2D line plot, which would in our case be the graph of cosine function ranging from -2π to 2π:

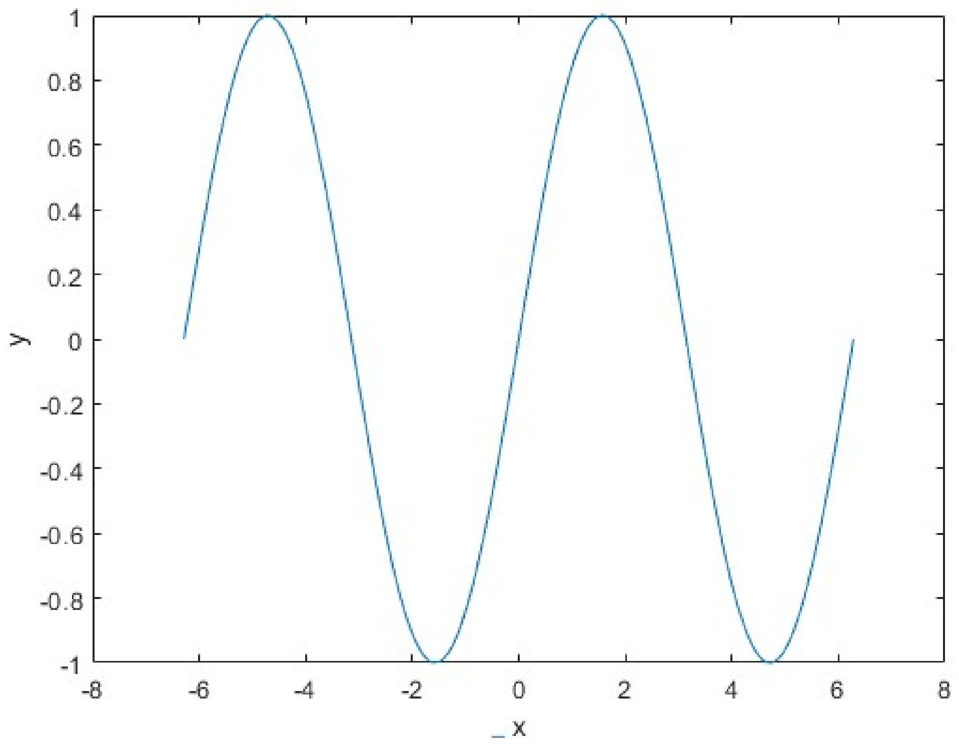


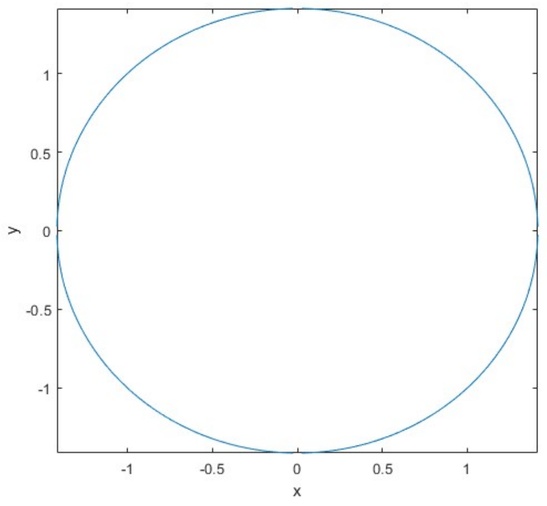
Figure 14: 2D line plot of a cosine function in Matlab

At this point we can start noticing differences between similar computing software. For example, the basic plot command works differently in each of the programmes. If we take a look back, in Maple, the plot command consists of variables and their definitions directly inside of the parameters part whereas Matlab requires both variables to be previously defined in order for the plot function to work correctly. Were we to try and use the same procedure we used in Maple in order to construct graphs in Matlab, we would face an error message. If we are working with Matlab, this is important to remember as any variable that we would like to include in our computations must be properly defined beforehand, as all variables are stored in an in-advance reserved portion of Matlab's memory.

In the next part we can discuss how Matlab treats implicitly constructed functions and their visualization in form of graphs. Similar to Maple, in Matlab we too are required to use a different function to project an implicitly constructed function. Not only that, but we also need to define the function's variables (as we have learned in the previous part) before we can use them in any subsequent commands. For our example we will again attempt to create a graph, this time that of an implicit function of a circle. The difference between the different approaches of Maple and Matlab can be spotted at first glance of the following command sequence:

```
i = @(x,y) x.^2 + y.^2 – 2;
fimpicit(i)
```
(23)

Notice how in the first step, we define the function and save it into the variable ***i***. We can then use this new variable ***i*** to call our previously defined function. The second step of the sequence is the function for constructing graphs of implicit functions defined by the variable in the parameters bracket which is, in our case, the variable ***i***. Several other parameters are to be noted as well. Firstly, we are utilizing the function handle (**@**) before specifying the individual parameters. "*A function handle is a MATLAB® data type that stores an association to a function. Indirectly calling a function enables you to invoke the function regardless of where you call it from"* (Mathworks®, 2020). Without the function handle we would not be able to define the function in this manner and would need to resort to more complicated function definition. The other important aspect to keep in mind is that when computing with ***x*** and ***y*** we are effectively no longer operating with scalars and therefore there is a need to employ an element-wise multiplication. This can be done by adding a period after both ***x*** and ***y*** variable, where the period marks that the operation after it is to be performed on an element-wise basis. The main purpose of this approach is to execute the operation between each element of the first variable and each element of the second variable. The one condition being that both variables must be the same size or be at least compatible. If they are not the same size, but are compatible, then they implicitly expand to match each other in size (Mathworks®, 2020).



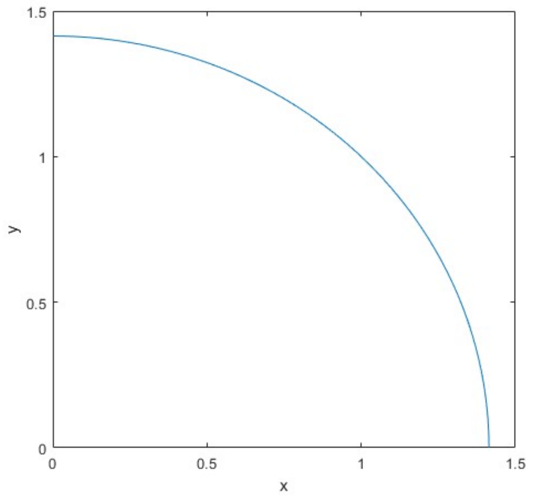


Figure 15: Implicit graphs in Matlab

If we take a look at the resultant graph (See Fig. 15), we see that although we did not specify any range over which we would like the software to scale the axes, Matlab made this action automatically. It is entirely possible however, to enter this parameter manually and choose which portion of function to inspect. This is done by including a vector of individual range parameters after the function variable parameter in the command structure:

```
fimplicit(i,[0 1.5 0 1.5])
```
(24)

Not only are we able to change the displayed range, but additional parameters also allow us to shift colours, line patterns and properties and much more. We can also choose to project our graph into a different coordinate system the same way we did it in Maple with one exception and that is defining the two variables of radius ***r*** and angle $\boldsymbol{\vartheta}$ beforehand. The resulting graph is visually identical to that of Maple (See chapter 7.1) :

```
theta = 0:0.01:4*pi;
r = theta;
polarplot(theta, r)
```
(25)

Shifting away from real numbers, we can attempt to plot a function of complex numbers.

Operations with complex numbers and their visualization are handled in a similar way in Matlab as they would be in Maple or any different software. We have previously learned that in Maple, we are able to plot the real part on x-axis against the imaginary part on y-axis. Similar procedure can be followed in Matlab and we would achieve the same exact results. If we consider two arrays of complex numbers: ***a*** and ***b***; that we want to use to construct our graph, we can plot each one of them individually without a problem.If we were, however, to plot multiple sets of complex coordinates, distinction needs to be made between their real and imaginary parts as they can no longer be called as a single variable in which they are stored. We can demonstrate this process on the following examples:

```
Plot(a)
                                        (26)
Plot(real(a),imag(a))
```

If our goal was to create a graph from an array of complex values, either of these two commands would deliver the appropriate results. As we can see in the second command example, we are utilizing the commands **real()** and **imag()** in our arguments. These commands take the complex array and produce only the real or the imaginary values of each complex number as their return values. This becomes important if more than one array of complex values is to be visualized. If we were to simply use the variables in which they are stored as arguments for the **plot()** function, Matlab would create a graph, but only of real values of the first array against the real values of the second array while simultaneously producing a warning message to warn us about the fact that both sets of imaginary values are being ignored and not projected. This is where we need to utilize the **real()** and **imag()** functions in the following form:

```
Plot(real(a),imag(a),real(b),imag(b))        (27)
```

With this approach, both complex arrays are projected into the graph with their real values on the x-axis and imaginary values on the y-axis. The entire process can further be adjusted by inserting the return values of both **real()** and **imag()** commands into new variables and utilizing those variables rather than the commands themselves.

So far both software choices seem to operate in a very similar manner in terms of command structure. Both have a command portion, portion for parameters and both can enter additional parameters to influence the graphic side of a visualized function. Now we are going to explore the different options we are presented with while using the Matlab's tools to visualize functions in three-dimensional environment. Our command of choice will be dependent on whether we want to project a curve or a surface. Let's begin by introducing Matlab's three-dimensional environment with construction of lines.

Our first task is to define a set of values over which the function will be projected. We can define a new variable ***r*** to accommodate these values:

r = -2*pi:0.1:2*pi (28)

We now have an array of 126 values saved inside ***r*** which means that our next step will be defining the individual variables of our function. This can be accomplished by a series of the following commands:

```
x = sin(r)
y = cos(r)
z = sin(r).*cos(r)                    (29)
plot3(x, y, z)
```

The last line consists of the command "**plot3**" which plots our line-connected coordinates in a three-dimensional space (See Fig. 16). This is, however, only one of the possible approaches. If our goal is to project a surface instead of a curve in three-dimensional space, we need to use an entirely different command and also define all of the data and save it into variables beforehand. This will make the process more intricate than we might be used to from Maple for example, but it is a necessary step required my Matlab.

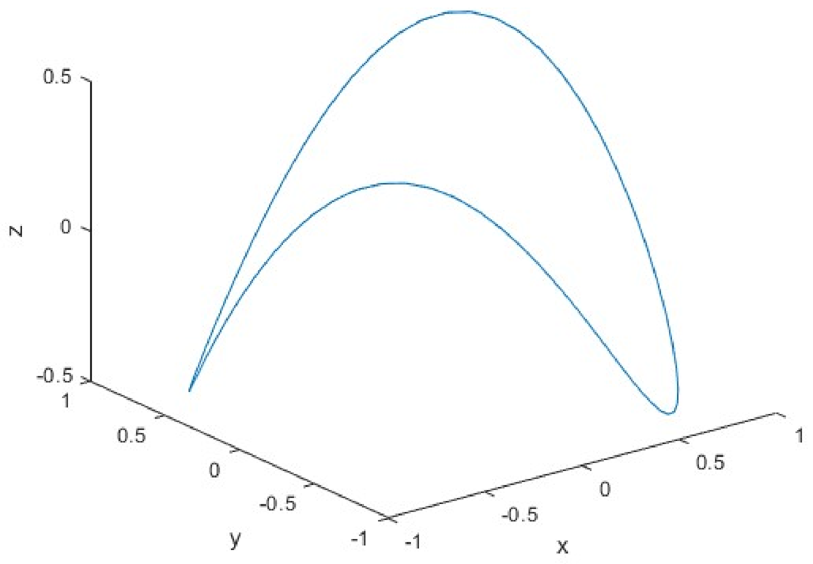


Figure 16: Example of a resulting graph of a plot3() command in Matlab

To make a graph of a surface in Matlab, we firstly need to create something called **meshgrid**. To achieve this, we need two sets of coordinates to serve as a base for our meshgrid. We can begin by defining these two variables:

x = -2*pi:0.1:2*pi

y = -2*pi:0.1:2*pi (30)

The next step is to create a two-dimensional grid coordinates using the defined ***x*** and ***y*** arrays:

[X, Y] = meshgrid(x, y) (31)

At this point of the process we have successfully created two matrices containing grid coordinates based on previously defined vector arrays. The only step that is left is to introduce a function into the process and we can now use the new ***X*** and ***Y*** coordinates as parameters of this function:

Z = sin(X).* cos(Y) (32)

After this step, all the required parameters are now defined and we are now presented with another choice. We can either project these parameters into a two-dimensional space with the usage of the **contour()** command (See Fig. 17a) or use the command **surf()** to visualize what we have been working on in three-dimensional space (See Fig. 17b). Whatever the choice, we will be using X, Y and Z as the three input parameters in both cases. "*The function plots the values in matrix Z as heights above a grid in the* x-y *plane defined by X and Y. The color of the surface varies according to the heights specified by Z.*" (Mathworks®, 2020).

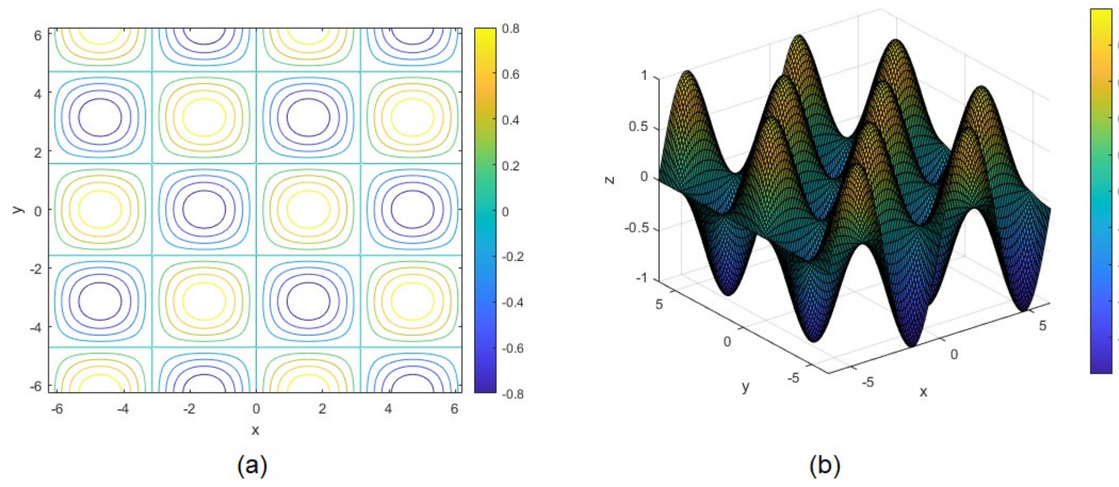
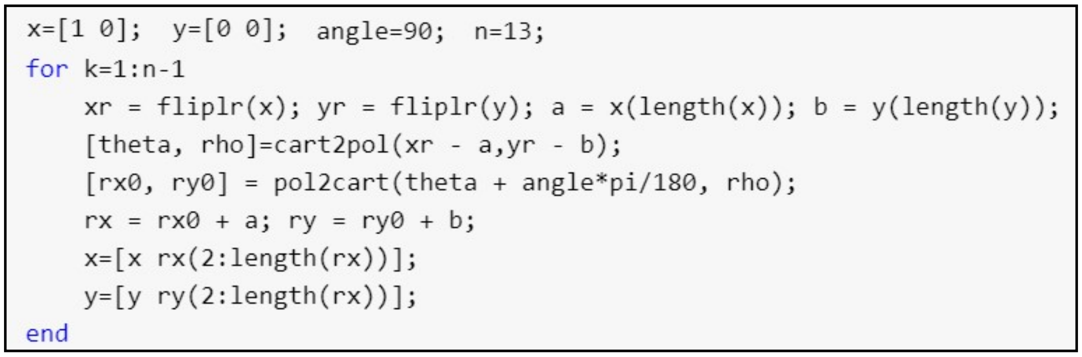


Figure 17: graphs of contour() **(a)** and surf() **(b)** commands in Matlab

As we can see, the **surf()** command yields us fairly equal results to those of Maple's **plot3d()** command. The uttermost difference between the approaches of these two programs is undoubtedly the process which Matlab introduces. Where in Maple we can fit the entirety of the command structure into a single line, making it more compact and easier to navigate within the single command, Matlab's method is that of individually defining each parameter as its own variable. This fact can sway our view both ways. The approach in Matlab is considerably more demanding and can introduce unwanted syntax errors if we are not familiar with Matlab's commands and working environment or careful enough, but by the same measure it also makes our working space more organized and each of the parameters is more accessible and easier to modify, should we want to. All the commands used in this chapter are retrievable from and can be further examined in the official Mathworks® Help Center.

8.2 Construction of an advanced object in Matlab

Since we have already gone over the most important Matlab commands and their usage, we can now better describe a more complicated process of visualization. For this purpose we are going to follow a process of a fractal creation (see chapter 6.4), namely that of the Heighway Dragon curve, and analyze it step by step. The following code was sourced from Mathworks' community file exchange and was originally uploaded by Mr. Joseph Kirk.



```
x=[1 0];  y=[0 0];  angle=90;  n=13;
for k=1:n-1
    xr = fliplr(x); yr = fliplr(y); a = x(length(x)); b = y(length(y));
    [theta, rho]=cart2pol(xr - a,yr - b);
    [rx0, ry0] = pol2cart(theta + angle*pi/180, rho);
    rx = rx0 + a; ry = ry0 + b;
    x=[x rx(2:length(rx))];
    y=[y ry(2:length(rx))];
end
```

Figure 18: Code for generating the Heighway Dragon fractal (Kirk, 2006)

First step, as in any more intricate Matlab visualization procedure, is to define some essential variables. In our case the **x** and **y** vectors that provide the sets of x and y coordinates for the line of the fractal:

$$\begin{aligned} \mathbf{x} &= \mathbf{[1\ 0]} \\ \mathbf{y} &= \mathbf{[0\ 0]} \end{aligned} \tag{33}$$

The other two important parameters to define at the beginning of the entire process are the angle which determines by how man degrees will each iteration be rotated, and the number of the iteration of which we want the fractal to be constructed. The iteration number is saved into the variable ***n*** and angle in degrees into the variable ***angle***:

$$\begin{aligned} \mathbf{angle} &= \mathbf{90} \\ \mathbf{n} &= \mathbf{13} \end{aligned} \tag{34}$$

With the essential variables defined, we can delve into the process by which the fractal is constructed. First step is to encapsulate the entirety of the following command sequence into a loop, in our case the "**for**" loop:

for k=1:n-1 (35)

This means each of the commands following this one will repeat a set amount of times (in our case until we reach one less iteration than the set amount ***n***). The next line of commands deals mainly with saving data into new variables that will be used later:

xr = fliplr(x); yr = fliplr(y)

a = x(length(x)); b = y(length(y)) (36)

The first line makes use of the **fliplr()** command which is used to flip the order of an array from left to right the variables **x** and ***y*** are used as an argument for this command and the output data is saved into alternatively named variables ***xr*** and ***yr***. The second line introduces two new variables (***a*** and ***b***) and simultaneously defines them. The command **length()** (which returns a number of elements contained in an array) is used as an index for the variable ***x*** (see chapter 8.1). This means that we count the elements in **x** then take the element on the position of that value and save it into the variable ***a***. The same process is followed for ***b***, but with values of ***y*** instead of ***x***.

[theta, rho]=cart2pol(xr - a,yr - b) (37)

Next, in the line above, the variables ***a*** and ***b*** are subtracted from the flipped arrays ***xr*** and ***yr*** the results are then transformed into arrays of polar coordinates (see chapter 5.2) with the help of the **cart2pol()** command. The next part accommodates for the rotation of the ongoing iteration:

[rx0, ry0] = pol2cart(theta + angle*pi/180, rho) (38)

As we need to shift the structure of the fractal by a set amount of degrees, the value of the previously defined variable ***angle*** is computed with the converted polar coordinate array ***theta*** and both polar coordinate arrays are then again converted to arrays of Cartesian coordinates with the help of the **pol2cart()** command. The newly converted arrays are saved into ***rx0*** and ***ry0*** variables.

```
rx = rx0 + a; ry = ry0 + b;
```
(39)

We create yet another set of variables and save the previously converted variables with the beforehand calculated array sizes (***a*** and ***b***) into them. The final step is to update the original arrays ***x*** and ***y*** that we started with to reflect this iteration's progress. This is achieved by the following set of commands.

```
x = [x rx(2:length(rx))];
y = [y ry(2:length(rx))];
                   ;
```
(40)

Selected elements (from second to the last one) of the ***rx*** array are added to the elements of the original array ***x*** and the newly created array then overwrites the previous array of ***x***. Same is done for ***y***, this time with ***ry*** and ***y*** elements. This entire process is then repeated again and again, until the **for** loop ends as mentioned in the beginning. All that is left to do is to enter the correct visualization command which would in our case be the standard **plot()** with ***x*** and ***y*** as its two parameters:

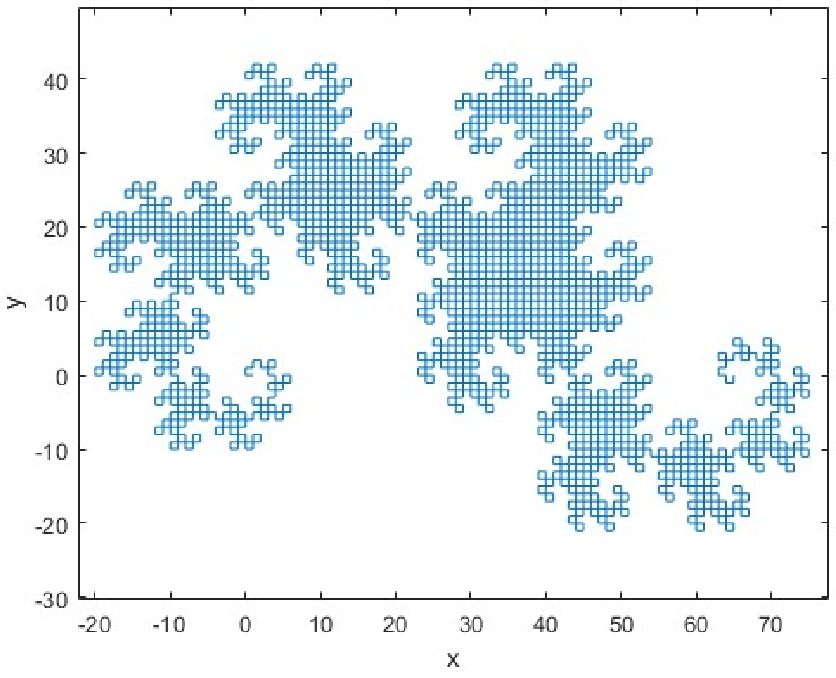


Figure 19: Resultant representation of 13th iteration of the Heighway Dragon fractal

8.3 Comparison between Matlab and Maple

This segment serves as a summary of different functions both programs offer and features that might be present in one but missing in the other. We can begin by assessing the software's working environment.

Regardless of the type, any graph we will visualize in Maple will be displayed directly in the working environment together with our code. While this approach is quite straightforward, it can also cause minor problems if our working environment is cluttered with heaps of information, displaying our graphical results into it might make it substantially more difficult to navigate through the data. Another disadvantage of this approach is the limited accessibility of different tools that would let us modify the final portrayal of the graph. This issue is not present in Matlab as every graph we visualize, via all the different types of plot functions, is opened in a separate window that also contains a handful of useful tools (as does Maple's working environment) that allow us to enrich the graph with additional information or visual aids such as axe labels, height levels, tools to change the environment around the graph itself and many more. All these graph enrichments can also be implemented as additional parameters directly inside the various functions. This approach is more direct but can also make the commands seem cluttered with non-vital text.

But perhaps the most apparent distinction between the two is the Matlab's emphasis of saving data (parameters) into individual variables. Where Maple's primary functionality is listing of all the parameters directly inside the function in question, Matlab goes the way of saving individual parameters as their own entities. Both programs can of course apply the opposite approaches, but that might not work correctly every time or not be compatible with some more intricate commands.

To make sure the approach we have chosen at any given moment is the right one, both Maple and Matlab have fairly extensive Help centres at their disposal. These Help centres contain additional information about different commands, processes and even specific parameters for individual commands. It is strongly encouraged to utilize these Help centres as much as possible every time an uncertainty arises through our work process, as they offer first hand explanations behind vast majority of the program's functions.

9 Conclusion

This thesis is focused on the procedures in computer software that make it possible for functions to be projected as their graphical representation on a computer. After we go through the first chapters describing the principles of computer graphics and the fundamental distinction between raster and vector graphics, the primary focus is then on introduction into the scope of functions, their different attributes and their behaviour under specific circumstances in different environment. Further, the focus shifts more to the different ways of operating with the visualization process.

In the second part of the thesis, mathematical software called Maple is utilized to demonstrate how software of this kind can be useful in our effort to create graphs of function. We learn about how different commands are constructed and what mistakes to avoid in order for them to function properly and that while similar software is usually quite intuitive and helps its user achieve the desired results, it is still important to follow certain procedures in order for the program to fully act according to our expectations. Afterwards we utilize different visualization software – Matlab by Mathworks® to further explain some of the processes and most importantly to make a comparison between different types of software available to the public. Whilst we see that there are many differences between how each of them operates and treats individual tasks, the innate mechanics of these programs share tremendous amount of similarities and it is therefore quite easy to navigate them once we learn the most differences such as the syntax and process of combining commands.

Software tools like Maple, Matlab, Mathematica and many others are nowadays widely used in academic and professional environment and it is therefore very beneficial for wider public to become more acquainted with at least the basics of the processes that stand behind the operation of such software.

10 Rozšířený český abstrakt

Cílem této bakalářské práce je seznámení čtenáře nejrůznějšími přístupy počítačové vizualizace objektů ve dvourozměrném i trojrozměrném prostředí. Jelikož jakákoliv iterace takovéhoto procesu vyžaduje určité znalosti nejrůznějších problematik matematiky, tvoří jejich vysvětlení první část této práce.

Před přikročením k základním stavebním blokům matematické stránky vizualizace jsou také vysvětleny rozdíly mezi dvěma základními typy grafiky, s kterými se setkáváme v běžném počítačovém rozhraní. Rozdíly v jejich realizaci a jejich použití v praxi jsou obsahem první tematické kapitoly. Dalším krokem k porozumění počítačové vizualizace je chápání matematických funkci. Funkce tvoří základními stavební jednotky jakékoliv navazující operace probírané v této práci a je tedy důležité se důkladně obeznámit s jejich konstrukcí a vlastnostmi. Pozornost je věnována také historii moderní definice funkce, bez které by nebylo možno zacházet se složitějšími matematickými úlohami, jak jsme dnes již zvyklí. V úvodu kapitoly o funkcích je čtenáři představen jednoduchý koncept fungování funkce. Mimo to jsou také probrány dodatečné vlastnosti, jež tvoří podmínky správného zobrazení dané funkce ve formě grafu, jako například definiční obor, obor hodnot. Následující část práce je věnována jednotlivým typům funkcí, jež jsou určeny specifickými pravidly týkající se veličin, s kterými funkce operuje. Rovněž se čtenáři seznámí s funkcemi implicitními, které na rozdíl od standardních funkcí nemají svou funkční veličinu zadanou explicitní formou, a také je představen typ funkcí, ve kterých figuruje více než jedna proměnná. Po představení těchto typů funkcí přichází na řadu další možná kategorizace funkcí, tentokrát závislá na tom, z jaké množiny čísel do funkce přiřazujeme. Všechny tyto kategorizace funkcí nám lépe pomáhají porozumět jednotlivým principům jejich konstrukce a slouží proto jako pomyslný vstupní bod k praktikám vizualizace.

Než je možno přistoupit k práci v jednotlivých počítačových programech, je nutno porozumět procesům zobrazování jako takovým a přesně s touto problematikou se pojí další kapitola této práce. Souřadnicové systémy tvoří pomyslný základ zobrazovacího prostředí a to ať již v počítačové grafice, či při zobrazování na prostý papír. Znalost správného prostředí pro zobrazení dané funkce určuje způsob jejího předpisu a v první řadě také formu souřadnic pro takovou funkci. Přesto, že v běžné praxi se nejčastěji setkáváme pouze s jedním souřadnicovým systémem, znalost a chápání ostatních jsou klíčovými vlastnostmi při práci v počítačové grafice, jelikož některé operace nelze realizovat bez nutné změny souřadnicového systému a s tím spjatého převodu hodnot souřadnic.

Dalším kritickým faktorem vizualizace objektů je samotný proces vykreslování obrazců. Za vytvořením křivky grafu může stát jeden z procesů popsaných v následující kapitole této bakalářské práce. Nejprve je pozornost věnovaná konstrukci pomocí polynomů. Čtenář se dozvídá, co to polynomy jsou, a jak jejich řády ovlivňují vzhled konstruované křivky. Dalším a tím převážně v praxi využívaným procesem je interpolace souřadnicových bodů za pomocí splajnů. Takováto interpolace rovněž zahrnuje několik podkategorií, které je možno využít v nejrůznějších situacích, kdy tou nejjednodušší metodou je prosté spojování sousedních bodů grafu za pomocí přímek. V neposlední řadě je také nastíněn proces generování fraktálů, který rovněž souvisí s vykreslováním dat v grafu, jak je možno demonstrovat na Mandelbrotově množině a později i v demonstraci v prostředí jednoho z vizualizačních softwarů.

Po seznámení čtenáře se základními principy funkcí a vizualizace přichází na řadu obeznámení se specifickými postupy vizualizačních softwarů. V první kapitole této části práce je čtenáři představen matematický vizualizační software Maple. S využitím tohoto softwaru je poté čtenáři představeno využití funkcí ve vizualizačním softwaru. Pozornost je věnována především konstrukci grafů všech typu již předem zmíněných funkcí čtenář má možnost dozvědět se, jak jsou taková konstrukční pravidla přenesena přímo do softwaru. Jelikož se jedná o vizualizační software, je nanejvýš důležité znát a dodržovat správnou formu syntaxe ve všech příkazech, které programu zadáváme. Tomuto tématu se věnuje krátká kapitola, ve které jsou popsány nejen nejčastější chyby syntaxe ze strany uživatele, na které je třeba brát zřetel, ale také nežádoucí artefakty, jež mohou při procesu zobrazování za specifických podmínek vzniknout a často negativně ovlivnit kýžený výsledek.

V další kapitole je čtenáři představen Matlab, velice rozšířený software nejen pro matematickou vizualizaci. Struktura fungování Matlabu pro účel vizualizace se od již představeného softwaru Maple poněkud liší, zejména protože Matlab dokáže plnit spoustu jiných funkcí mimo pole vizualizace. Je proto důležité si nejprve představit tyto nové metody a postupy, na kterých je fungování programu Matlab postaveno. Nejvýraznější změnou celého procesu zadávání příkazů může být pro uživatele například definování každé nové veličiny a její ukládání do specifických proměnných, před jakýmkoliv jiným použitím této nadefinované proměnné v dalších příkazech. Vizualizační postupy, které tak například Maple dokázal zvládnout prostým zadáním jednoho příkazu, se v Matlabu mohou značně protáhnout, jelikož je třeba předem definovat veškeré komponenty tohoto procesu. Na první pohled by se tak mohlo zdát, že Maple nabízí jednodušší řešení, avšak při podrobnějším zkoumání a porovnávání dalších funkcí obou programů zjišťujeme, že postupy Matlabu přinášejí své vlastní výhody.

Po shrnutí vizualizačních postupů Matlabu je věnována pozornost také praktické ukázce konstrukce kódu k vykreslení již dříve zmíněného fenoménu – fraktálu. V této části jsou již dříve popsané postupy a taktiky uplatněny v praktické ukázce vizualizace v prostředí programu Matlab a čtenáři je tak co možná nejpřesněji přiblížen vizualizační proces poněkud komplikovanějšího objektu.

Závěrem práce je porovnání obou vizualizačních softwarů s ohledem na běžný proces zobrazování. Z uvedených příkladů a popisu vyplývá zejména velká podobnost mezi oběma programy a do jisté míry i značná univerzalita příkazů pro vizualizaci jednotlivých druhů funkcí. Ne ve všem se však programy shodují a je tak zřejmé i podstatné množství rozdílů v jejich fungování. Tyto rozdíly však ve značném množství spadají spíše pod kategorii syntaxe a fundamentálních operací daného programu, než samotného procesu vizualizace. Důležité je také zmínit, že jak pro Maple, tak Matlab je online k dispozici oficiální knihovna veškerých jejich vizualizačních příkazů a postupů, která dobře poslouží nejen začínajícím uživatelům tohoto softwaru.

Klíčová slova

Funkce, vizualizace, grafika, proměnná, graf, fraktál, Matlab, Maple

11 References

ADOBE*, Portable Document Format* [online]. 2019. Retrieved from: https://acrobat.adobe.com/ie/en/acrobat/about-adobe-pdf.html

BOURKE, Paul. *The Mandelbrot at a Glance* [online]. 2002. Retrieved from: http://paulbourke.net/fractals/mandelbrot/

BOURNE, Murray. *Domain and Range of functions* [online]. 2019. Retrieved from: https://www.intmath.com/functions-and-graphs/2a-domain-and-range.php

BRONSHTEĬN, I. N. *Handbook of mathematics*. 4th ed. New York: Springer, c2004. ISBN 978-3540434917.

Data fitting: Polynomial Fitting and Splines (2016) YouTube video, added by Data4Bio [Online]. Avaliable at: https://youtu.be/BqZXS3n75l0

EVES, Howard. *Foundations and Fundamental Concepts of Mathematics*. 3rd edition. Dover Publications, 1997. ISBN 978-0486696096.

FOLEY, James D. *Computer graphics: principles and practice*. 2nd ed. in C. Reading, Mass.: Addison-Wesley, 1995. ISBN 978-0201848403.

KIRK, J. (2020). *Dragon Curve (aka Jurassic Park Fractal)* Retrieved from: https://www.mathworks.com/matlabcentral/fileexchange/11069-dragon-curve-aka-jurassic-park-fractal, MATLAB Central File Exchange. Retrieved June 1, 2020.

KAUFMAN, Arie. *Rendering, visualization, and rasterization hardware*. New York: Springer-Verlag, c1993. ISBN 978-354-0567-875.

KOPKA, Helmut a Patrick W. DALY. *LATEX: kompletní průvodce*. Brno: Computer Press, 2004. ISBN 80-722-6973-9.

MAHONEY, Matt. *Data Compression Explained* [online]. Ocarina Networks, 2010. Can be retrieved from: http://nishi.dreamhosters.com/u/dce2010-02-26.pdf

Maple Online Help, *complexplot*. Toronto: Maplesoft, a division of Waterloo Maple Inc., 2019. From: https://www.maplesoft.com/support/help/maple/view.aspx?path=plots%2Fcomplexplot

Maple Online Help, *implicitplot*. Toronto: Maplesoft, a division of Waterloo Maple Inc., 2019. From: https://www.maplesoft.com/support/help/Maple/view.aspx?path=plots/implicitplot

Maple Online Help, *plot*. Toronto: Maplesoft, a division of Waterloo Maple Inc., 2019. From:: https://www.maplesoft.com/support/help/Maple/view.aspx?path=plot

Maple Online Help, *plot3d*. Toronto: Maplesoft, a division of Waterloo Maple Inc., 2019. From: https://www.maplesoft.com/support/help/Maple/view.aspx?path=plot3d&term=plot3d

Maple Online Help, *Plotting Discontinuous Functions*. Toronto: Maplesoft, a division of Waterloo Maple Inc., 2019. From: https://www.maplesoft.com/support/help/Maple/view.aspx?path=plot/discont

Maple Online Help, *polarplot*. Toronto: Maplesoft, a division of Waterloo Maple Inc., 2019. From: https://www.maplesoft.com/support/help/Maple/view.aspx?path=plots/polarplot

MathWorks, (2020). *Help Center: cart2pol (R2020a)*. Retrieved June 1, 2020 from: https://uk.mathworks.com/help/matlab/ref/cart2pol.html

MathWorks, (2020). *Help Center: Create Function Handle (R2020a)*. Retrieved March 22, 2020 from: https://uk.mathworks.com/help/matlab/matlab_prog/creating-a-function-handle.html

MathWorks, (2020). *Help Center: fliplr (R2020a)*. Retrieved June 1, 2020 from: https://uk.mathworks.com/help/matlab/ref/fliplr.html

MathWorks, (2020). *Help Center: Meshgrid (R2020a)*. Retrieved March 23, 2020 from: https://uk.mathworks.com/help/matlab/ref/meshgrid.html

MathWorks, (2020). *Help Center: Plot Imaginary and Complex Data (R2020a)*. Retrieved March 22, 2020 from: https://www.mathworks.com/help/matlab/creating_plots/plot-imaginary-and-complex-data.html

MathWorks, (2020). *Help Center: pol2cart (R2020a)*. Retrieved June 1, 2020 from: https://uk.mathworks.com/help/matlab/ref/pol2cart.html

MathWorks, (2020). *Help Center: Power, .^ (R2020a)*. Retrieved March 22, 2020 from: https://uk.mathworks.com/help/matlab/ref/power.html

MOLER, Cleve. *The Origins of MATLAB*: Mathworks 2004. From: https://uk.mathworks.com/company/newsletters/articles/the-origins-of-matlab.html

ROSE, Michael. *Explainer: what are fractals?* [online]. 2012. Retrieved from: https://theconversation.com/explainer-what-are-fractals-10865

SHANNON, Alexander. Fractals, Compression and Contraction Mapping. *Eureka*. 2012, (62), 32-37.

SPIVAK, Michael. *Calculus*. 3rd edition. Houston: Publish or Perish, 1994. ISBN 0-914098-89-6.

STOVER, Christopher and WEISSTEIN, Eric. *Cartesian Coordinates*. From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/CartesianCoordinates.html

STOVER, Christopher and WEISSTEIN, Eric. *Polar coordinates*. From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/PolarCoordinates.html

SVOBODA, Zdeněk a Jiří VÍTOVEC. *Matematika 2* [online]. Brno, 2014. Retrieved December 10, 2019

TABACHNIKOV, S. *Dragon Curves Revisited, Mathematical Intelligencer*, 36, No. 1 (2014). Retrieved from: http://www.personal.psu.edu/sot2/prints/DragonCurves.pdf

WEISSTEIN, Eric W. *Bézier Curve*. From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/BezierCurve.html

WEISSTEIN, Eric W. *Spherical Coordinates*. From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/SphericalCoordinates.html

What's the Difference Between Raster rand Vector? (2019) Retrieved from: https://www.psprint.com/resources/difference-between-raster-vector/

ŽÁRA, Jiří, B. BENEŠ, J. SOCHOR a P. FEKEL. *Moderní počítačová grafika*. 2., přeprac. a rozš. vyd. Brno: Computer Press, 2004. ISBN 80-251-0454-0.

12 List of figures

13 List of tables