# Comparing trigonometric interpolation against the Barycentric form of Lagrange interpolation: A battle of accuracy, stability and cost 

Department of Mathematics, Linköping University

## Beatrice Söderqvist

LiTH-MAT-EX-2022/13-SE

Credits: 16 hp
Level: G2
Supervisor: Andrew Ross Winters,
Department of Mathematics, Linköping University
Examiner: David Rule,
Department of Mathematics, Linköping University
Linköping: June 2022

## Abstract

This report analyzes and compares Barycentric Lagrange interpolation to Cardinal Trigonometric interpolation, with regards to computational cost and accuracy. It also covers some edge case scenarios which may interfere with the accuracy and stability. Later on, these two interpolation methods are applied on parameterized curves and surfaces, to compare and contrast differences with the standard one dimensional scenarios. The report also contains analysis of and comparison with regular Lagrange interpolation.

The report concludes that Barycentric Lagrange interpolation is generally speaking more computationally efficient, and that the inherent need for periodicity makes Cardinal Trigonometric interpolation less reliable in comparison. On the other hand, Barycentric Lagrange interpolation is difficult to implement for higher dimensional problems, and also relies heavily on Chebyshev spaced nodes, something which can cause issues in a practical application of interpolation. Given ideal scenarios, Cardinal Trigonometric interpolation is more accurate, and for periodic functions generally speaking better than Barycentric Lagrange interpolation. Regular Lagrange interpolation is found to be unviable due to the comparatively big computational cost.

## Keywords:

Interpolation, Big O-notation, Cardinal function, Lagrange Interpolation, Barycentric Lagrange Interpolation, Trigonometric Interpolation, Parametrization

## URL for electronic version:

http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-185999

## Sammanfattning

Rapporten analyserar och jämför Barycentrisk Lagrange-interpolation med Kardinalisk Trigonometrisk interpolation, m.a.p. beräkningstid och noggrannhet. Rapporten täcker även några scenarion där metoderna är mindre noggranna och/eller stabila. I senare delen av rapporten används interpolationsmetoderna på parametriserade kurvor och ytor, för att jämföra och undersöka skillnader mot det vanliga endimensionella scenariot. Rapporten innehåller också en undersökning av vanlig Lagrange-interpolation.

I slutsatsen finner rapporten att Barycentrisk Lagrange-interpolation generellt sett är mer beräkningseffektiv, och att behovet av periodicitet gör trigonometrisk interpolation mindre pålitlig. $\AA$ andra sidan så är Barycentisk Lagrangeinterpolation svårare att implementera för problem av högre dimensioner, och kräver ofta Chebyshev-noder för att fungera, något som kan orsaka problem i en praktisk applicering av interpolation. I ideala scenarion är Kardinalisk Trigonometrisk interpolation mer noggrann, och för periodiska funktioner generellt sätt bättre än Barycentrisk Lagrange-interpolation. Vanlig Lagrange-interpolation ses som olämplig att använda p.g.a. den höga tidskostnaden.

## Nyckelord:

Interpolation, Ordo, Kardinalfunktion, Lagrangeinterpolation, Barycentrisk Lagrangeinterpolation, Trigonometrisk interpolation, Parametrisering

## URL för elektronisk version:

http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-185999

## Acknowledgements

I want to thank everyone who has in some way helped me on the journey to and through this project, including classmates, teachers, friends and family. Mathematics is built on previous mathematical knowledge, and mathematicians are only built with the support of lots of other people. I want to give a special thanks to:

Ida $\AA$ kerholm and Kristoffer Tondel, for being great friends and colleagues over the years and providing help during the project with various technical aspects. I also want to give an extra thanks to Kristoffer who helped push me to finish the project before summer.

Emma Vališová, for being an amazing best friend, giving me a feeling of comfort in times of hardship, and also introducing me to the wonders of BibTex.

My dad who listened carefully every time I showed my progress with this project, and whose support in my creative endeavors has always meant so much to me.

My supervisor Andrew Ross Winters, for coming up with the project idea in the first place, letting me set my own schedule and pace with the project, giving feedback at quite frankly an alarming rate, and all around being a wonderful teacher and mentor.

Finally I want to give thanks to you for taking the time and effort to read this report. It is a labor of love and I really hope you enjoy it!

## Contents

1 Introduction ..... 1
1.1 Background ..... 1
1.2 Method and content ..... 2
2 Theoretical Background ..... 5
2.1 Polynomial interpolation ..... 5
2.1.1 The standard Lagrange technique ..... 6
2.1.2 The Barycentric form of Lagrange ..... 6
2.2 Trigonometric interpolation ..... 7
2.2.1 Using a cardinal basis ..... 8
2.2.2 Fast Fourier Transform (FFT) ..... 9
2.2.3 Periodic extensions ..... 10
2.3 Computational cost ..... 11
2.3.1 Barycentric Lagrange ..... 11
2.3.2 Trigonometric Interpolation with cardinal basis ..... 11
2.4 Error calculation ..... 12
2.4.1 The $L_{p}$ Norm ..... 12
2.4.2 Spectral Convergence ..... 13
2.5 Problems with Stability and Convergence ..... 14
2.5.1 The Runge Phenomenon ..... 14
2.5.2 The Gibbs phenomenon ..... 16
2.5.3 Outliars or broken data ..... 17
3 Numerical Implementation ..... 19
3.1 Comparing the computational cost ..... 20
3.2 Evaluating the error ..... 24
Beatrice Söderqvist, 2022. ..... ix
4 Higher dimensions ..... 31
4.1 An introduction to parametrization ..... 31
4.2 An implementation of parametrization ..... 32
4.3 The third dimension ..... 35
5 Comparison and Conclusion ..... 41
5.1 Final comparison ..... 41
5.2 Further studies and potential improvements ..... 43
A Code appendix ..... 47
A. 1 Functions ..... 47
A. 2 Computational Cost ..... 52
A. 3 Error and stability ..... 56
A. 4 Higher dimensions (functions and programs) ..... 59

## Chapter 1

## Introduction

To understand the problem at hand we first need to understand interpolation itself, what it is and when we use it.

### 1.1 Background

Interpolation means inferring data from preexisting data. Just like extrapolating, interpolation is about finding a pattern of any sort, and trying to estimate new data points that still follow that pattern. In theory interpolation allows us to create any extra data points needed, or remove any unwanted/irregular data from an otherwise functional data set.

There are several practical applications of interpolation. It may, for example, be used to add extra frames in video, to make slow motion shots smooth and visible. It can also be used to remove dead pixels in a digital image, or to assist in animating at a higher frame rate. The animation industry, in fact, has an entire job title for interpolating images, the Inbetweener, whose job is to take a few frames of animation and draw in (or interpolate) fluid movement. However, just as there are several different ways to draw and animate a character based on circumstance, there are several different kinds of interpolation that solve different needs.

Because there are many patterns you can assume that a data set is following (a line, a polynomial, a trigonometric function etc.), there are many different ways to interpolate a given data set. A good first example is linear interpolation. If the data set is $0,10,20,30,40,50$ most people are able to intuitively tell
that the points in between would likely be $5,15,25,35,45$, making the doubled set $0,5,10,15,20,25,30,35,40,45,50$. Linear interpolation is a good example of a method that is intuitive and cost efficient, but not all data is made of straight lines. Thus, the accuracy of linear interpolation is usually insufficient. For increased accuracy non-linear interpolation methods such as Lagrange and trigonometric interpolation become useful.

Lagrange interpolation is a kind of polynomial interpolation, so an interpolation where you assume the data follows, or can at least be modelled by, some sort of polynomial. The Barycentric form of Lagrange interpolation is a variant of Lagrange interpolation. Other forms of polynomial interpolation such as Newton interpolation will not be discussed in this report, for the sake of brevity.

Trigonometric interpolation is, as the name suggests, a kind of interpolation method that assumes the data can be represented as some combination of sine and cosine functions. This is very closely related to Fourier series, in particular the real valued sine-cosine form of Fourier series is almost identical to the definition of trigonometric interpolation (which will be discussed in chapter 2.2). The specific kind of trigonometric interpolation we will use in this report is trigonometric interpolation using a cardinal basis (discussed in 2.2.1).

There are other kinds of interpolation methods, but they mainly fall into two camps. Either they attempt to design a function that matches all of the points in a given data set with some sort of polynomial (like Lagrange does), or they work by truncating an infinite series on a finite set of points. The former are sometimes called "Nodal" methods, and the latter are sometimes called "Modal" methods. Splines are also an alternative, which we mention breifly in chapter 2.5.3.

Both the Barycentric form of Lagrange interpolation, and the cardinal form of trigonometric interpolation will be described in detail at the start of the theoretical discussion.

### 1.2 Method and content

In this report we will analyze the theoretical differences between the Barycentric form of Lagrange interpolation and cardinal trigonometric interpolation, examining computational cost and asymptotic computational complexity. After the theoretical discussion we will use MatLab to implement and compare these algorithms, using different data sets to compare accuracy, stability and compu-
tational cost, as well as their practical limitations. The reasoning behind using MatLab and not some other computer software is that MatLab easily handles matrix calculations, which is useful when working with large two-dimensional data sets.

After having verified our theoretical results using numerical computations, we move on to discussing interpolation on parameterized curves, which allows us to interpolate two and three dimensional curves/surfaces.

At the end of the report we will summarize the strengths and weaknesses of Barycentric Lagrange interpolation and cardinal trigonometric interpolation respectively. We will compare their accuracy, stability and cost, and then give some recommendations for which algorithm to use in which context.

## Chapter 2

## Theoretical Background

In this section we will cover the construction of various interpolation methods. We then compare them in purely theoretical terms according to how fast they are to construct/use, their stability and accuracy, and if there are any important edge cases/base assumptions made about the interpolated functions.

When interpolating, we work with discrete values at various different nodes. This is how information is collected in real life, we don't measure a continuous function of data, we may measure it at specific points in time, or at specific points in space (for example temperature tracking over a day or the measuring of a coastline respectively). This means that we have a vector of points we want to interpolate, with nodes $t_{j}$ and data $y_{j}$ at the nodes ( $j$ goes from 0 to $n$ in Lagrange interpolation and from $-n$ to $n$ in trigonometric interpolation, both will be motivated in their respective chapters).

### 2.1 Polynomial interpolation

When interpolating data, a polynomial is usually a good starting point. Taylor series have long been used to approximate functions, and they are all just polynomials with specifically chosen coefficients. Polynomials in general are easily defined, have continuous derivatives, and are mostly intuitive (for people invested in mathematics). In fact, linear interpolation is technically just interpolation with polynomials of the first degree.

In theory, if we have $n+1$ distinct data points, there exists a polynomial of at most degree $n$, such that the polynomial intersects all $n+1$ data points.

Furthermore, polynomial interpolation is unique [1], so the polynomial of at most degree $n$ is also the only possible interpolation polynomial (as an example, think of the uniqueness of a line given two points in the Euclidean plane). If the function data itself is a polynomial of degree $\leq n$, then the polynomial interpolation will be exact; since the function is a polynomial of degree $\leq n$ then it may also be seen as interpolating the $n+1$ data points, but because polynomial interpolation is unique, both the function and the interpolating function are the same polynomial.

### 2.1.1 The standard Lagrange technique

Lagrange interpolation starts with finding the cardinal basis

$$
l_{k}\left(t_{j}\right)= \begin{cases}1 & \text { if } j=k  \tag{2.1}\\ 0 & \text { otherwise }\end{cases}
$$

Any polynomial of degree $n$ may be expressed as $p(x)=\prod_{k=1}^{n} c\left(x-r_{k}\right)$ where $r_{k}$ are the roots to the polynomial and $c$ is some constant. If we want to create a cardinal basis we simply need to remove the root for $t_{k}\left(\right.$ so $\left.l_{k}\left(t_{k}\right) \neq 0\right)$ and then normalize the function (so $l_{k}\left(t_{k}\right)=1$ ). The end result [1] is:

$$
\begin{equation*}
l_{k}(x)=\prod_{i=0, i \neq k}^{n} \frac{\left(x-t_{i}\right)}{\left(t_{k}-t_{i}\right)} \tag{2.2}
\end{equation*}
$$

Now that we have all of the cardinal basis for all of the data points, we need only multiply each with $y_{k}$ and then sum it all together, and we have our interpolating polynomial:

$$
\begin{equation*}
p(x)=\sum_{k=0}^{n} y_{k} l_{k}(x) . \tag{2.3}
\end{equation*}
$$

### 2.1.2 The Barycentric form of Lagrange

While the Lagrange polynomial is useful in theory, it's not as good for computation and it has stability problems 4. For every data point, all of the cardinal functions $l_{k}$ must be evaluated at said point. The basis $l_{k}$ is a product of $n$ terms, so the total work is $O\left(n^{2}\right)$ for every data point [1]. We will now introduce the Barycentric form, which is both faster and more numerically stable (a detailed explanation of why can be found in chapter 2.3 and this chapter respectively).

Let's return to the construction of the cardinal basis. We define the polynomial $\phi(x)=\prod_{j=0}^{n}\left(x-t_{j}\right)$ and the barycentric weights:

$$
\begin{equation*}
w_{k}=\frac{1}{\prod_{j=0, j \neq k}^{n}\left(t_{k}-t_{j}\right)} . \tag{2.4}
\end{equation*}
$$

Now we may construct the cardinal basis as

$$
\begin{equation*}
l_{k}(x)=\phi(x) \frac{w_{k}}{x-t_{k}}, \tag{2.5}
\end{equation*}
$$

and thus the interpolating polynomial is

$$
\begin{equation*}
p(x)=\phi(x) \sum_{k=0}^{n} y_{k} \frac{w_{k}}{x-t_{k}} . \tag{2.6}
\end{equation*}
$$

However, if $p(x)$ is the constant function 1, then all of the function data is $y_{k}=1$ (as previously established, polynomial interpolation of a polynomial of degree 0 must be exact). So,

$$
\begin{equation*}
1=\phi(x) \sum_{k=0}^{n} \frac{w_{k}}{x-t_{k}} \tag{2.7}
\end{equation*}
$$

Because $\frac{p(x)}{1}=p(x)$ we may finally determine an alternate form for $p(x)$ which does not contain $\phi(x)$ :

$$
\begin{equation*}
p(x)=\frac{\sum_{k=0}^{n} y_{k} \frac{w_{k}}{x-t_{k}}}{\sum_{k=0}^{n} \frac{w_{k}}{x-t_{k}}} . \tag{2.8}
\end{equation*}
$$

This is the Barycentric form of Lagrange interpolation 4]. It is important to note that $w_{k}$ is calculated independently of $y_{k}$, thus independent of function data, so every $w_{k}$ need only be calculated once, even if we change the data. Barycentric Lagrange interpolation is also more numerically stable than regular Lagrange interpolation, but to save time we leave the explanation to Berrut and Trefethen and their publication named "Barycentric Lagrange Interpolation" 4].

### 2.2 Trigonometric interpolation

Up until now we have only discussed interpolation using polynomials up to degree $n$, but now we examine an alternative with a different basis function ansatz. First, we assume that the function or data set we want to interpolate is periodic. Without loss of generality, we further assume that one period can be represented within the interval $[-1,1]$. We know that $f(x)=f(x+2)$ for all $x$, and while
it's possible to interpolate this function using a polynomial, the more reasonable approach would be to use Trigonometric Interpolation. Trigonometric functions are periodic, and sums of periodic functions are also periodic, so using sums of trigonometric functions to approximate periodic functions is a reasonable idea, at least compared to polynomials (which are not periodic).

For trigonometric interpolation, we define our nodes as $t_{k}=\frac{2 k}{N}, k=-n, \ldots, n$ where $N=2 n+1$. We use an odd number of nodes, mainly because we want to include the node $t=0$ (this will be helpful in chapter 2.2.1. Also notice that unlike polynomial interpolation, the end points $t= \pm 1$ are not included.

We assume that every trigonometric interpolating polynomial may be written as

$$
\begin{equation*}
p(x)=\frac{a_{0}}{2}+\sum_{k=1}^{n}\left(a_{k} \cos (k \pi x)+b_{k} \sin (k \pi x)\right) \tag{2.9}
\end{equation*}
$$

where the goal is to find the constant $a_{0}$, and the other unknown coefficients $a_{1}, a_{2} \ldots a_{k}, b_{1}, b_{2} \ldots b_{k}$. Using Euler's formula $e^{i x}=\cos (x)+i \sin (x)$ we can condense the real valued function into the more elegant complex form:

$$
\begin{equation*}
p(x)=\sum_{k=-n}^{n} c_{k} e^{i k \pi x} \tag{2.10}
\end{equation*}
$$

where we instead need to find $c_{-n}, c_{-n+1} \ldots c_{0} \ldots c_{n-1}, c_{n}$. In both the real and the complex notation, we must determine the values of $2 n+1$ unknowns.

### 2.2.1 Using a cardinal basis

When performing trigonometric interpolation, we can take a similar approach to Lagrange. That is, we create a cardinal basis for each node and then multiply each basis with the value of their respective nodal function values. In trigonometric interpolation (on equally spaced nodes), the cardinal basis we want to use is:
$\tau(x)=\frac{2}{N}\left(\frac{1}{2}+\cos (\pi x)+\cos (2 \pi x)+\cos (3 \pi x)+\ldots+\cos (n \pi x)\right)=\frac{\sin (N \pi x / 2)}{N \sin (\pi x / 2)}$
Notice that $\tau\left(t_{k}\right)=0$ for $k \neq 0$ and that $\tau\left(t_{0}\right)=\lim _{x \rightarrow 0} \tau(x)=1$.
On equally spaced nodes it is straightforward to define $\tau_{k}(x)=\tau\left(x-t_{k}\right)$ since this will ensure that $\tau_{k}\left(t_{k}\right)=\tau(0)=1$, and that all of the other nodes within the period equal zero. So now we have a set of functions with period 2 , and each
is 1 at one node and 0 at all the other nodes within each period. We now have a cardinal basis. With the cardinal basis for each node established, we use the same formula as in Lagrange interpolation (though with a different set of nodes) to get our final trigonometric interpolating function: $p(x)=\sum_{k=-n}^{n} y_{k} \tau_{k}$.

If we use an even number of nodes rather than an odd amount, all that changes is the denominator in the cardinal basis [1].

$$
\begin{equation*}
\tau(x)=\frac{\sin (N \pi x / 2)}{N \tan (\pi x / 2)} \tag{2.12}
\end{equation*}
$$

### 2.2.2 Fast Fourier Transform (FFT)

While the cardinal basis form of trigonometric interpolation is functional and fast, there exist other alternatives to trigonometric interpolation, the most popular of which is the Fast Fourier Transform. The details of the FFT algorithm is a topic beyond the scope of this paper, so this will mostly serve as an introduction to its computational cost and why it is useful to determine the interpolation coefficients.

As mentioned in Chapter 2.2, we may write the task of interpolation as the search for the complex valued unknowns $c_{-n}, c_{-n+1} \ldots c_{0} \ldots c_{n-1}, c_{n}$ in 2.10. Since we have $2 n+1$ unknowns and $2 n+1$ nodes, we may do this via a linear system of equations of size $(2 n+1) \times(2 n+1)$. Solving this system of equations by ordinary means without any special approach will give you a computational time of $O\left(n^{3}\right)$ (it would require some kind of LU factorization, and then forward/backward solves, but the point is that it is very computationally expensive). However, because of the structure in this system of equations and the underlying "nice" properties of sine/cosine basis functions, there are techniques that can be used to cut down on time, and eventually solve it within $O\left(n \log _{2}(n)\right)$. This way of solving the linear system of equations is called the Fast Fourier Transform. Rather than one single algorithm, it's a collective name for all the algorithms that may solve the system in $O\left(n \log _{2}(n)\right)$ [6].

There are many different kinds of FFT algorithms, Kopriva uses Temperton's self sorting, in place complex FFT [6], and MatLab uses FFTW (Fastest Fourier Transform in the West) which is a collection of various algorithms that all fall under the FFT umbrella [3]. FFTs are a subcategory of interpolation methods of their own, so we will not dig much deeper into them as it would take up the entire report.

### 2.2.3 Periodic extensions

Now, while we have made the assumption that the function we are interpolating is periodic, what happens if it is not? Trigonometric interpolation will still yield a periodic output, so what happens is that the interpolated function "pretends" that the function being interpolated is also periodic.

As an example: $f(x)=|x|$ is not a periodic function. However, if we use trigonometric interpolation on the interval $x=[-1,1]$, then the interpolating function will have a period of two, so if we evaluate it between $x=[1,3]$ we will get the same function just translated by two units. This applies infinitely in either direction, creating a periodic function (specifically a triangle wave), where there previously was the non-periodic $f(x)=|x|$


Figure 2.1: A periodic extension of $f(x)=|x|$.

While the end points in this example have the same value, $f(-1)=f(1)$, the interpolation algorithm will still interpolate regardless of this. So if instead of $f(x)=|x|$ we use $f(x)=x$, we now get a discontinuity every $x=1+2 n, n=$ $0, \pm 2, \pm 4 \ldots$ The same can applies for the periodicity of derivatives, the periodic extension can produce discontinuities there too if the end points do not match up. These kinds of discontinuities will be analyzed in 2.4.2.

It should be noted that while you can look outside the interpolated bounds for polynomial interpolation too, what you find is at best nothing interesting and at worst a nonsensical mess (unless we are interpolating a polynomial, in which case it is perfect).

### 2.3 Computational cost

We have now established our two competing methods: The Barycentric form of Lagrange interpolation and trigonometric interpolation using cardinal basis. We may now analyse how fast they are in terms of construction and evaluation.

It should be noted that the implementation used in chapter 3 is based on treating individual values of $x$ rather than creating functions/polynomials where we can simply insert $x$. This has the upside of being more intuitive and efficient, at the cost of always having to define how many nodes we want to evaluate rather than constructing a polynomial, and then choosing how many points to evaluate for it. Because of this implementation, we may consider $x$ to be an arbitrary real number when considering the computational cost, rather than a variable in a polynomial. Had we taken the variable approach, we would have used Horner's rule and Horner's method in this section [7], but the computational cost analysis is essentially the same regardless of implementation, just explained in different ways.

### 2.3.1 Barycentric Lagrange

As mentioned in the motivation for using the Barycentric form, the standard Lagrange interpolation method has a computational time of $O\left(n^{2}\right)$ to evaluate at a given point $x$. The Barycentric weights also take $O\left(n^{2}\right)$ operations to calculate ( $n+1$ weights consisting of products with $n+1$ nodes). However, the weights only depend on the interpolation nodes and not the data, i.e. only on $t_{k}$ and not $y_{k}$. This means that after we have selected the set of interpolation nodes, we only need to calculate the barycentric weights once. All in all we get that for any set of nodes (not function data), the computation time for the Barycentric form is $O\left(n^{2}\right)$ in their construction, but on subsequent runs evaluating the interpolant in Barycentric form costs just $O(n)$. This is because we have two sums of $n$ polynomials (so each is $O(n)$ ), and we then have a division between two polynomials, which as mentioned is with our implementation just division between two real numbers, so $O(1)$.

### 2.3.2 Trigonometric Interpolation with cardinal basis

The cardinal form of trigonometric interpolation as described in chapter 2.2.1 has a very good asymptotic computational complexity, since for each node we only need to calculate $\frac{\sin (N \pi x / 2)}{N \sin (\pi x / 2)}$ and then sum it all together. Each cardinal basis has a constant computational time, so our total sum will only be $O(n)$. However, while the time for each cardinal basis is constant, it does involve using
trigonometric functions, and while not slow, they are definitely much slower than simple multiplication and division. Thus, we can expect our cardinal trigonometric interpolation to do well with very large amounts of nodes, but with smaller or more medium sized calculations, the fact that the asymptotic computational complexity is $O(n)$ does not help as much.

Chapter 2.2 .2 talks about the motivation behind FFT, and as discussed there, the computational time is $O\left(n \log _{2}(n)\right)$.

### 2.4 Error calculation

Arguably one of the most important aspects of any interpolation method is its actual accuracy. In this chapter we will examine how close our interpolation functions fit the interpolated function, working under ideal scenarios. In the next chapter (chapter 2.5) we will then break down some edge cases where the error converges more slowly, or does not converge at all.

### 2.4.1 The $L_{p}$ Norm

To begin with, we need to establish what type of error we are measuring. There are many different ways to define this problem, but the one we will be using in this report is the $L_{p}$ norm (also called the p-norm). The $L_{p}$ norm for any given vector of numbers is defined as [7]:

$$
\begin{equation*}
\|\mathbf{x}\|_{p}:=\left(\sum_{i=1}^{n}\left|x_{i}\right|^{p}\right)^{1 / p} \tag{2.13}
\end{equation*}
$$

and when $p$ approaches infinity we get:

$$
\begin{equation*}
\|\mathbf{x}\|_{\mathrm{inf}}:=\max _{i}\left|x_{i}\right| \tag{2.14}
\end{equation*}
$$

The norms that we will be using are the 2-norm (also called the Euclidean norm), and the infinity-norm (also called the maximum norm). Because we are interested in the size of the error specifically, we will in the following chapter use the notation $\left\|u-I_{N} u\right\|$, where $u$ is the vector of values from the function being interpolated, and $I_{N}$ is the interpolation operation (so $I_{N} u$ are the values from the interpolation function).

### 2.4.2 Spectral Convergence

If we want to meaningfully analyse the interpolation error, Canuto \& Hussani [2] provide us with some helpful equations.

For trigonometric interpolation:

$$
\begin{equation*}
\left\|u-I_{N} u\right\|_{2} \leq C N^{-m}\left\|u^{(m)}\right\|_{2} \tag{2.15}
\end{equation*}
$$

For polynomial interpolation (though not with equally spaced nodes, see chapter 2.5.1):

$$
\begin{equation*}
\left\|u-I_{N} u\right\|_{2} \leq C N^{\frac{1}{2}-m}\|u\|_{H^{m}} \tag{2.16}
\end{equation*}
$$

Where $N$ is the number of nodes, $C>0$ is a constant, $m$ is the number of continuous derivatives the interpolated function possesses, and $u^{(m)}$ the $m$-th derivative of the function.

We will not attempt to prove these equations as true, and some parts like the Sobolev norm $\left\|u^{(m)}\right\|_{H^{m}}$ are beyond the scope of this paper, but the focus here is the relationship between the error, $N$ and $m$. Both equations contain the factor $N^{-m}$ or $N^{\frac{1}{2}-m}$. This means that if for example $m=1$, using twice the nodes for trigonometric interpolation results in half the error, but with $m=2$ twice the nodes means the error is one fourth and so on. Polynomial interpolation works in much the same way, but the error converges slightly slower due to the $\frac{1}{2}$ in the exponent. When $m$ approaches infinity we get an exponential decay, this decay is also called Spectral Convergence. This is directly related to how quick the Fourier coefficients converge, hence the use of the word "spectral".

We now see how periodic extensions (chapter 2.2.3) can become problematic. Even if the function itself has infinitely many derivatives that are smooth and bounded, and even if all of those derivatives are periodic, unless we choose the proper bounds, we will get a much slower convergence. Compare $\sin (x)$ over $x=[0, \pi]$ to $\sin (x)$ over $x=[0,2 \pi]$. In both cases, the edge values are equal to zero, but the derivative of $\sin (x)$ is $\cos (x)$, and $\cos (0) \neq \cos (\pi)$, meaning that the first alternative does not even have a continuous first derivative. The end result is that $\sin (x)$ over $x=[0,2 \pi]$ will converge spectrally ( $m=$ inf ), but $\sin (x)$ over $x=[0, \pi]$ will not.

### 2.5 Problems with Stability and Convergence

Even the best of interpolation methods may have issues, and these may not even be the best methods. This section will cover some of the problem scenarios relevant to both interpolation methods, and how well they handle them.

### 2.5.1 The Runge Phenomenon

When interpolating with any polynomial using equally spaced nodes (not just Lagrange polynomials), increasing the number of nodes does not necessarily improve the results. What occurs instead is that the polynomial starts to oscillate rapidly around the edges, this is known as the Runge phenomenon.


Figure 2.2: Polynomial interpolation of $f(x)=\frac{1}{1+25 x^{2}}$, using equally spaced nodes ( 10 to the left, 20 to the right). The Runge phenomenon is clearly visible, with wild swinging up and down at the edges.

As can be seen in the example, the polynomial approximates the function well towards the middle, but the closer we get to the edges the worse the approximation becomes. Crucially, this issue gets worse and worse the more nodes we add, so the pointwise error does not converge. The solution to this problem is to forgo equally spaced nodes altogether, and instead use a different distribution.

Since the problem appears at the edges of the function, we use a distribution of nodes that's more dense at the edges and less dense in the middle. Thus we arrive at the Chebyshev points of the second kind. The Chebyshev points of the second kind are defined as $t_{k}=-\cos \left(\frac{k \pi}{n}\right), k=0,1 \ldots n$


Figure 2.3: The distribution of Chebyshev nodes $n=10$. 5 ]

Using Chebyshev nodes we may finally achieve spectral convergence when using Barycentric Lagrange interpolation, as long as the function is sufficiently smooth (as discussed in chapter 2.4.2. Unless mentioned otherwise, we will use Chebyshev nodes for the Barycentric form of Lagrange interpolation, in the remainder of this report.


Figure 2.4: Polynomial interpolation of $f(x)=\frac{1}{1+25 x^{2}}$, using Chebyshev nodes (10 to the left, 20 to the right). The Runge phenomenon is no longer present.

The Runge phenomenon does not affect trigonometric interpolation functions [1]. However, they have a similar effect in a different problem called the Gibbs Phenomenon.

### 2.5.2 The Gibbs phenomenon

When using trigonometric interpolation, we can run into some problems if the function being interpolated is not continuous. Wherever there is a discontinuity, the interpolant can over- and undershoot the function or the periodic extensions (see 2.2.3). Importantly, this overshoot does not go away as we add more nodes. This is known as the Gibbs phenomenon.

The square wave is a great example of a periodic function that is not continuous, it has two jump discontinuities per period (one goes up, the other goes down). The square wave has many definitions as it is more about the look rather than the underlying math, but the definition we have used here is:

$$
f(x)= \begin{cases}1 & \text { if } \pi(n-1) \leq x<\pi n ; n= \pm 1, \pm 3, \pm 5 \ldots  \tag{2.17}\\ -1 & \text { otherwise }\end{cases}
$$



Figure 2.5: Trigonometric interpolation of the sign function, which when repeated periodically becomes a square wave with amplitude 1 (see 2.2.3). We use 10 nodes in the figure on the left, and 100 nodes in the figure on the right.

As we add more and more nodes, we notice that the height of the overshoot stays the same, however the width of the overshoot does decrease, meaning that the infinity norm will not decrease, but the 2 norm could decrease (see chapter 2.4), although we have no guarantee of the latter.

Generally speaking, whenever we interpolate a function that has a jump of size $\alpha$, the overshoot is $\alpha \cdot(0.08949 \ldots)$ in each direction, when the number of nodes approaches infinity (so the jump in our square wave made from a sign function is 2 , meaning the jump in our interpolating function is about $2 \cdot(1+0.8949 \ldots) \approx 2.36)$. We will not explain where 0.08949 comes from, but [2] goes into more detail.

### 2.5.3 Outliars or broken data

Using one interpolating polynomial to interpolate all the data simultaneously can be dangerous, since that assumes that there must exist some smooth polyno$\mathrm{mial} /$ trigonometric sum that fits the data. In most cases, something that looks similar to say a 2 nd degree polynomial, still doesn't have to follow a perfect trajectory. This can be due to the person collecting the data, the equipment being used, or simply due to some minor chaos that's common across various physics experiments.

Because both of our interpolation methods are "one function fits all", neither is equipped to deal with the issue of outliar data. An alternative to this could be for example the use of splines [7], where instead of 7 points creating one polynomial of degree 6 , you instead could use two polynomials of degree 3 , and then connect them together continuously. Another alternative (although it is not strictly speaking interpolation, but rather a form of curve fitting) is the use of orthogonal polynomials [1], such as the least squares method. Orthogonal polynomials allow for curve fitting which does not require perfect matches, which in turn can lead to a more well rounded function if the data slightly oscillates, or single data points don't follow the rest of the curve.

Neither of these alternatives will be discussed in detail in this report, but if the data being collected is unreliable, these methods are a good approach.

## Chapter 3

## Numerical Implementation

Now that we have laid out the theoretical groundwork, we move on to a numerical implementation of the two competing methods. We first analyze the computational cost, and then calculate the accuracy on various ideal and less than ideal scenarios.

As mentioned in chapter 2.3 , the way we have chosen to implement the interpolation methods is via calculating arrays of distinct values, so for each interpolation function the input is a set of nodes, the data at those nodes, and then which values for x we wish to evaluate. Barycentric Lagrange interpolation also requires weights as an input. The output is thus the evaluated function values corresponding to each x in the "evaluation node" list. For the sake of visual clarity, we will only ever use a set of equally spaced nodes as evaluation nodes, as it makes plotting easier, but you may use whatever evaluation nodes you wish, as they are simply distinct values for x .

Most of the code was written independently, but some help has been taken from Driscoll [1] and Kopriva [6]. All of the code that was used can be seen in the code appendix

### 3.1 Comparing the computational cost

For the computational cost, we are using the tic/toc functions in MatLab. This is to isolate only the cost of the functions we want to test, and to not count such things as creating the array of interpolated nodes, or plotting etc. All of the code is being run on a laptop with the CPU "AMD Ryzen 75800 HS", none of the code makes any use of a GPU or any other CPUs.


Figure 3.1: The computational cost of Lagrange interpolation. Each cross is the mean time of 10 runs of Lagrange_Array.m, and each evaluates 50 points on a randomly selected function.

The asymptotic computational complexity for Lagrange interpolation is $O\left(n^{2}\right)$ as expected, and as we will see later in this section, it is on another order of magnitude compared to Cardinal Trigonometric and Barycentric Lagrange interpolation. This is mainly due to the fact that the algorithm makes for some complicated lists, since 2.2 is not only a product, but a product excluding particular values (forcing us to make lists containing every value but one). Even at low numbers of nodes, this takes a lot of time.


Figure 3.2: The computational cost of constructing the barycentric weights. Each point on the plot is the mean time of 20 runs of Barycentric_Weights.m.


Figure 3.3: The computational cost of evaluating the Barycentric form of the Lagrange polynomial (2.8), given a set of already defined weights. Each point on the plot is the mean time of 10 runs of Barycentric_Lagrange_Array.m, where crosses belong to the function evaluating 2000 points, and circles belong to the function evaluating 1000 points.

Barycentric Lagrange interpolation is both significantly faster than regular Lagrange interpolation, and has a better asymptotic computational complexity (since with weights, the cost is linear). The linearity of Barycentric_Lagrange_Array.m also makes it easier to illustrate how this particular implementation handles evaluating the function. As can be seen in fig. 3.3, evaluating twice the number of nodes takes about twice as long, which is expected since all that does is run the outer loop of the function twice as long, nothing else changes. Thus we have a linear relationship between the number of points we want to evaluate, and the amount of time it takes to evaluate the polynomial.


Figure 3.4: The computational cost of constructing the entire barycentric polynomial. Barycentric_Lagrange_Array.m is used to evaluate 1000 nodes, and is shown as crosses on the plot. Barycentric_Weights.m is shown as circles. Both plots are averages of 10 runs of their respective algorithm.

As can be seen in this image, somewhere around 1500 nodes, the barycentric weights start to overtake the rest of the construction in terms of computational cost. However since we are evaluating 1000 nodes, it is slightly unrealistic to construct the polynomial with 1500 nodes. Because of the asymptotic computational complexity, there will exist some scenario where the number of nodes being evaluated are higher than the amount used for construction, and where the barycentric weights take more time, but for most practical purposes we can think of the barycentric weights as the more cost efficient algorithm. It may be $O\left(n^{2}\right)$, but because it is a very simple algorithm it is still faster for most realistic sets of nodes.


Figure 3.5: A comparison between Cardinal Trigonometric and Barycentric Lagrange (with preestablished weights). Each circle/cross on the plot is the mean time of 20 runs, and each function evaluates 200 points. Barycentric Lagrange is shown as circles, Cardinal Trigonometric as crosses.


Figure 3.6: A comparison between Cardinal Trigonometric and Barycentric Lagrange (with the time of weight calculation included). Every circle/cross on the plot is the mean time of 20 runs, and each function evaluates 500 points. Barycentric Lagrange is shown as circles and Cardinal Trigonometric as crosses.

As can be seen here, while $O(n)$ is great and all, the Barycentric polynomial is significantly faster to calculate compared to the trigonometric one. On top of that, the Barycentric weights are very fast and have a low impact on even relatively large sets of nodes. If we were to frame it in more theoretical terms, we can expand the costs and frame them as such: Cardinal Trigonometric has a cost of $|O(n)| \leq C_{1} n+C_{0}$, the Barycentric form of Lagrange has a cost of $|O(n)| \leq D_{2} n^{2}+D_{1} n+D_{0}$. Since $D_{2} \neq 0, D_{2} n^{2}$ will eventually dominate, but since $C_{1} \gg D_{1}$, the Cardinal Trigonometric will take longer for even a relatively large number of nodes.

### 3.2 Evaluating the error

As established in the theory section, we will mostly make use of the 2-norm to determine accuracy, as it is a fairly rounded way to measure it, and since we also have a nice theoretical framework around spectral accuracy that uses the 2-norm.

Each time we measure the error we use 100 evaluation nodes (evaluation nodes, not construction nodes), since the number of evaluation nodes itself should have no real implication towards the error (just the computation time). However, if there are some strange results when attempting to reproduce these scenarios using the code below, it can be useful to increase the number of evaluation nodes. This is because a low number of evaluation nodes may pick the places we evaluate the error in an uneven or biased way (the higher the number of evaluation nodes, the less the error function ought to oscillate).

An important thing to note in this chapter is that we are using loglog and semilog scales on the plots, depending on the results. If we attain spectral convergence it shows up as a straight line on a plot where only the y-axis is a log scale (the error), but if we do not have spectral convergence ( $m=\inf$ in 2.4.2 we use a loglog scale because the error is decreasing as by some sort of power function, and will thus show up as a straight line on the loglog scale. This also means that in our implementation, when we increase the number of nodes to be interpolated upon, for the semilog scale we simply add a number to it " Nconst $=$ Nconst +1 ", but when we use the loglog scale we instead multiply it by a number "Nconst $=N$ const $* 2$ ". If we would not do the latter, plotting the loglog scale would take a significantly longer amount of time.


Figure 3.7: The two norm error of Barycentric Lagrange interpolation on the function $f(x)=e^{\sin (\pi x)-2 \cos (\pi x)}$.


Figure 3.8: The two norm error of Cardinal Trigonometric interpolation on the function $f(x)=e^{\sin (\pi x)-2 \cos (\pi x)}$.

In an ideal scenario with a periodic and smooth function, both Cardinal Trigonometric interpolation and Barycentric Lagrange interpolation attain spectral accuracy. As expected, while both have spectral accuracy, Cardinal Trigonometric interpolation is faster for periodic functions. Since spectral accuracy makes the error decrease exponentially, it shows up as a (mostly) straight line on a semi log scale.


Figure 3.9: Cardinal Trigonometric of a square wave 2.17) (seen on the left), and a triangle wave given by $|x|$ (seen on the right). The function $|x|$ is interpolated on the interval $x=[-1,1]$, meaning the triangle wave has an amplitude of 1 and period of 2 .

This example shows the difference between a discontinuous function (the square wave), and a continuous function with a discontinuous first derivative (the triangle wave).

The error for the square wave appears to bottom out around $10^{4}$ nodes, although it is hard to tell if this is due to poor implementation or not, as it is a fairly high number of nodes (and thus may have introduced some numerical error in highly oscillating square wave interpolation).


Figure 3.10: The two norm error of Barycentric Lagrange interpolation on the Runge function, $f(x)=\frac{1}{1+25 x^{2}}$.


Figure 3.11: The two norm error of Cardinal Trigonometric interpolation on the Runge function, $f(x)=\frac{1}{1+25 x^{2}}$.

Barycentric Lagrange interpolation has spectral convergence with Chebyshev nodes. With Cardinal Trigonometric interpolation however, we run into problems. Even though the function in figure 3.11 is smooth, the periodic extension is not. The bounds chosen for this interpolation were $x=[-1,1]$, but the Runge function has a derivative with no repeated values. Thus it is impossible to get anything better than a periodic extension that is continuous but without a continuous derivative, and so Cardinal Trigonometric interpolation is limited in the same ways it would be when interpolating for example $f(x)=|x|$. It is interpolating a continuous function but with a discontinuous derivative.


Figure 3.12: The two norm error of Barycentric Lagrange interpolation on the Runge function, $f(x)=\frac{1}{1+25 x^{2}}$. This time we use equally spaced nodes instead of Chebyshev nodes of the second kind.

The Runge phenomenon makes the error go up instead of down, until it reaches a point where the interpolation function oscillates so violently the program can not keep up with the measurements.


Figure 3.13: Barycentric Lagrange interpolation of a square wave, using 1280 Chebyshev nodes of the second kind.

When calculating the error of functions without spectral convergence ( $m \neq$ $i n f$ ), we were unable to do many comparisons with Barycentric Lagrange. This is due to the numerical implementation being somewhat unstable, since the barycentric weights 2.4 are calculated beforehand, and if we are not careful they have a tendency to get either really big or really small. If they get too big/too small, MatLab is unable to recognize the difference between the weights and infinity/zero, and thus the implementation just breaks.

## Chapter 4

## Higher dimensions

Up until now, we have limited the scope of interpolation to discrete data points collected over some axis. However, the main ideas and algorithms of interpolation are applicable much more generally to any set of data points, such as two dimensional objects (circles, squares, images of cats etc.) and three dimensional objects (spheres, cubes, images of cats etc.). A lot of the same problems with stability also persist into higher dimensions, as parametrization rephrases the problem as several one dimensional problems. This chapter is dedicated to these higher dimensional scenarios (though we will not go into the fourth dimension and beyond, as it is very hard to draw).

### 4.1 An introduction to parametrization

While functions are typically represented in the 2D-plane with a relationship between $x$ and $y$ (for example, $y=x^{3}-6 x^{2}+2 x$ ), this is a bit lacking for most 2D-shapes. A circle for example can be represented as $\left(x-x_{c}\right)^{2}+\left(y-y_{c}\right)^{2}=r^{2}$ where $\left(x_{c}, y_{c}\right)$ is the center point and $r$ is the radius of the circle. However, things get increasingly complex and hard to define with spirals, loops etc. A more intuitive approach is to use parametrization, where instead of defining a relationship between $x$ and $y$, we define a separate variable $\theta$ and let it "wander" across the plane. In our circle example, this means we define $x=\cos (\theta)$ and $y=\sin (\theta)$, and let $\theta$ "wander" between $[0,2 \pi]$.

One major upside of parametrization is that we do not need to consider the whole when interpolating, we simply need to interpolate the $x$ and $y$ values separately. For two dimensional curves, this means interpolating $x$ with nodes given by $\theta$, and $y$ with nodes given by $\theta$. The numerical implementation can be found in the code appendix.

### 4.2 An implementation of parametrization

This section will mostly feature some visual examples of interpolation on parameterized curves. We are not concerned with computing cost, and accuracy is more so shown via visual examples of the curves, and not via exact measurement (as in chapter 3.2.

The different kinds of curves showcased are used to highlight certain phenomena, so we will make use of mostly curves with infinite periodic derivatives.


Figure 4.1: Barycentric Lagrange Interpolation of a Limaçon on 6, 8, 12 and 24 Chebyshev nodes of the second kind. Already after 12 nodes we get a surprisingly accurate interpolation.


Figure 4.2: Cardinal Trigonometric interpolation of the same Limaçon with 6 and 24 nodes respectively. Because it is made of $\cos ()$ and $\sin ()$ functions, trigonometric interpolation gets a perfect approximation almost immediately.

We saw with the examples in chapter 3 that trigonometric interpolation can quickly get a near perfect fit for any function that is periodic, infinitely differentiable, and with periodic derivatives. With parameterized curves, the result is no different. Barycentric Lagrange interpolation also performs well with Chebyshev nodes, but takes slightly longer to converge, as expected.


Figure 4.3: Interpolation of a spiral. Top left is Barycentric Lagrange interpolation with 20 nodes, and the three following images are Cardinal Trigonometric interpolation on 20,100 and 2000 nodes.

Figure 4.3 shows the Gibbs phenomenon on a parameterized curve. Barycentric Lagrange performs well with Chebyshev spaced nodes, but Cardinal Trigonometric interpolation struggles due to the implied discontinuity in the $x$ axis (see chapter 2.2 .3 for why this is discontinuity is only implied and thus not relevant to Lagrange). Even at high amounts of nodes, the Gibbs phenomenon causes a lot of oscillation.


Figure 4.4: Barycentric Lagrange interpolation with equally spaced nodes, on the function $x=1.2 \cos (\theta), y=\cos (\theta) \sin (\theta)^{3}$, with 30,60 and 90 nodes respectively. The image on the bottom right is a zoomed out version of the 90 node interpolation.

As can be seen in figure 4.4, the Runge phenomenon is just as present and just as destructive as before. The implementation used makes it look like there are very jagged corners when the function fails, but in reality the function is actually smooth, we are just not evaluating enough points. In theory we could evaluate more, but since all that would accomplish is making an already failed attempt look a bit smoother, we do not increase the number of evaluation nodes.


Figure 4.5: A cat interpolated with Cardinal Trigonometric interpolation.

As long as we have a set of nodes that are equally spaced and in some sense loop around, we may use Cardinal Trigonometric interpolation. Barycentric Lagrange interpolation could be applied to any curve that has Chebyshev spaced nodes, but these nodes are of course much harder to construct by hand.

### 4.3 The third dimension

Reaching into the third dimension requires a bit more work than the second dimension. In order to parameterize a single three dimensional curve, we only need one variable, but to parameterize a three dimensional surface, we need two. For this purpose we have constructed new interpolating functions, but in broad strokes they still function the same. We are still interpolating one dimension at a time, now just with two parameters instead of one.

With that said, the focus of this section is more on the results rather than the underlying theory, for the sake of brevity. We did not have time to construct Barycentric Lagrange interpolation with two parameters, so instead the comparison here is between regular Lagrange interpolation and Cardinal Trigonometric interpolation.


Figure 4.6: Regular Lagrange interpolation of a Möbius strip.

Since regular Lagrange is a fairly slow algorithm we use a comparatively low number of nodes. While the Runge phenomenon is still possible, because we limit ourselves to so few nodes we are less likely to encounter the phenomenon (or at least visually see the effects of it).


Figure 4.7: Cardinal Trigonometric interpolation of a single sphere


Figure 4.8: Cardinal Trigonometric interpolation of two spheres at once (the formula for a sphere, but with $v=[-\pi, \pi])$. Note that although we use two spheres as interpolation nodes, we still only evaluate values between $-\pi / 2$ and $\pi / 2$ ("veval" in the code).

While initially a sphere might seem as very periodic and smooth, in reality the way we construct it in three dimensions is not. The standard way of computing a sphere is with:

$$
f(u, v)=\left\{\begin{array}{l}
x=\cos (u) \cos (v)  \tag{4.1}\\
y=\sin (u) \cos (v) \quad \text { where } u=[0,2 \pi], v=[-\pi / 2, \pi / 2] \\
z=\sin (v)
\end{array}\right.
$$

However, if we pay attention to $z$, we notice that the end points $\sin (-\pi / 2)=$ $-1 \neq \sin (\pi / 2)=1$ meaning that although the sphere looks round, there is a discontinuity in the z-axis (and while $x$ and $y$ are periodic, they are limited to a non-periodic derivative with regards to $v$ ). Since we know that $\sin (x)$ has a period of $2 \pi$, we may increase the bounds of $v$ to $v=[-\pi, \pi]$ to create a fully periodic function with infinite periodic derivatives. With these bounds, v "wanders" up and back down again, meaning we are interpolating two spheres rather than one. The end result is that one sphere is affected by Gibbs phenomenon, but two spheres are not.

Gibbs phenomenon is particularly problematic in three dimensions, since it can apply to any number of surfaces which look periodic but are not, like for example the Möbius strip.


Figure 4.9: Cardinal Trigonometric Interpolation of the function in figure 4.6

## Chapter 5

## Comparison and Conclusion

This chapter is a quick summary of the comparisons between the two interpolation methods, and some concluding thoughts on when to use which method. At the end is also a section giving some insight into topics which could be analyzed in more detail.

### 5.1 Final comparison

The Barycentric form of Lagrange interpolation (2.8) is better than the regular form (2.3) in almost every way. It is faster to construct, more stable [4], and not overly complicated to implement. While it is harder to implement, in the one dimensional case it does not require much extra thought. For the three dimensional surfaces however, it is a bit more complicated, as evidenced by the fact we did not have time to implement it in this project. In the choice between regular and Barycentric Lagrange interpolation, the Barycentric variant ought to be used as often as possible. Regular Lagrange interpolation should only be used for educational purposes or if the Barycentric variant is too complicated to implement.

With regards to computational cost, Barycentric Lagrange interpolation is the fastest for most practical scenarios. Cardinal Trigonometric interpolation may have the better asymptotic computational complexity of $O(n)$, but because it makes use of $\sin ()$ and $\cos ()$ functions, it takes a lot longer for a machine to process, compared to multiplication and division. There exists a theoretical point where Cardinal Trigonometric interpolation is faster than Barycentric interpolation, but it is well beyond 2000 nodes and thus not likely to be found.

If the Barycentric weights are already given, Barycentric Lagrange interpolation is guaranteed to be faster than Cardinal Trigonometric interpolation. Both Barycentric Lagrange and Cardinal Trigonometric are orders of magnitude faster than Lagrange interpolation.

In terms of accuracy, Cardinal Trigonometric interpolation is better than Barycentric Lagrange by a small margin, if they are both working under ideal conditions. However, both of these come with some pretty big asterisks, as they require very specific sets of nodes. Barycentric Lagrange interpolation requires Chebyshev nodes of the second kind to not encounter the Runge phenomenon, and Cardinal Trigonometric interpolation needs equally spaced nodes in such a way that the function being interpolated maintains periodicity. While the former is problematic with practical data collection (most data is collected at equal intervals), the latter is problematic with how the functions themselves are structured, as the need for periodicity means Cardinal Trigonometric interpolation essentially interpolates discontinuous functions if the function is not periodic. If the data for some reason is unreliable or contains outliar points (a practical example being something like dead pixles), neither method is all that functional as they both assume the data being collected is mostly accurate.

When it comes to higher dimensional interpolation, and especially 3D surfaces, the issue of how we get our nodes becomes increasingly relevant. As seen with the two spheres, even node sets that seem like they would work do not. Because the Gibbs phenomenon is much more noticeable compared to the Runge phenomenon, at least with the low amounts of nodes we used, Lagrange is generally speaking more accurate and more reliable.

With higher dimensions the issue of computational cost also becomes increasingly relevant. Lagrange interpolation is already a fairly slow algorithm, so combine that with the fact that we are interpolating a mesh of points, and we run into some very slow calculations. Even with 10 in $u$ and 10 in $v$ we have a matrix of 100 separate points, which for Cardinal Trigonometric is no problem but for regular Lagrange takes quite some time. In terms of computational cost, regular Lagrange is significantly worse than the other options in three dimensional problems. Ironically, we did not have time to implement Barycentric Lagrange interpolation into three dimensions, as it does not build on the same cardinal function idea and thus requires a different approach.

Generally speaking, if you know the data is periodic, use Cardinal Trigonometric interpolation, if you know it is not periodic or are unsure, try Barycentric Lagrange interpolation, even if the data is equally spaced.

### 5.2 Further studies and potential improvements

While we tried our best to create as many comparisons as possible between the two interpolation methods, there were some things we did not have the time to implement. As discussed in chapter 3.2, the Barycentric weights while theoretically stable, have a tendency to get too big or small for any numerical implementation. Finding a solution to that issue would help with comparing Barycentric Lagrange to Cardinal Trigonometric interpolation in scenarios where $m \neq i n f$ according to 2.4.2.

Another thing not implemented was Barycentric Lagrange on three dimensional surfaces, since it is not based on cardinal bases and thus needs some other approach compared to regular Lagrange. This is mostly due to a lack of time and effort on our part, the project was nearing completion and we had to focus on other things. As Barycentric Lagrange and regular Lagrange both produce the same polynomial, it would be mostly for the sake of decreasing the computational cost, but an implementation would be welcome regardless.

Of course, there are also many other interpolation methods that ought to be analyzed. A comparison with the Fast Fourier Transform in particular would be helpful due to its popularity, but things like newton polynomials and even splines would also be interesting to compare.

Most of the sources used in this report talk about other aspects of interpolation to some extent, especially [1], 2], [4] and [6]. These are excellent books and articles if you wish to learn more about interpolation, and explore various methods and scenarios.

## Bibliography

[1] Richard J. Braun and Tobin A. Driscoll. Fundamentals of Numerical Computation. Society of Applied and Industrial Mathematics, 2017.
[2] Alfio Quarteroni Claudio Canuto M. Yousuff Hussaini and Thomas A. Zang. Spectral Methods: Fundamentals in Single Domains. Springer Science \& Business Media, 2007.
[3] Matteo Frigo and Steven G. Johnson. Fastest Fourier Transform in the West.
[4] Lloyd N. Trefethen Jean-Paul Berrut Barycentric Lagrange Interpolation*. Society of Applied and Industrial Mathematics, 2004.
[5] Steven G. Johnson. Chebyshev-nodes-by-projection.
[6] David A. Kopriva. Implementing Spectral Methods for Partial Differential Equations. Springer, 2009.
[7] Linde Wittmeyer-Koch Lars Eldén`Numeriska Beräkningar analys och illustrationer med MATLAB. Studentlitteratur AB, Lund, 2001.

## Appendix A

## Code appendix

The appendix is split into several parts, with the first being all of the custom made functions, and the others being the various uses of those functions in different examples.

## A. 1 Functions

```
Affine_Transform_Eq.m
function [out_nodes,transformations] = Affine_Transform_Eq(x)
    % Transforms a set of EQUALLY SPACED nodes into an array
    % suitable for trigonometric interpolation
    % Store the affine transformations for future reference
    transformations = zeros(1,2);
    out_nodes = x;
    amount = length(x);
    size = x(length(x))-x(1);
    first_node = x(1);
    % Translation
    out_nodes = out_nodes - size/2 - first_node;
    transformations(1) = - size/2 - first_node;
    % Scaling
    out_nodes = out_nodes .* (2/size) .* ((amount-1)/amount);
    transformations(2) = (2/size) * ((amount-1)/amount);
end
```


## Barycentric_Lagrange_Array.m

```
function out_array = Barycentric_Lagrange_Array (x_j, y, w, x)
    % This is the Barycentric form of Lagrange interpolation
    % The output has been changed to an array of discrete numbers
    % rather than an anonymous function
    n = length(x_j)-1;
    out_array = zeros(1,length(x));
    % The outer loop simply cycles through every value of x,
    % instead of creating an anonymous function we simply
    % calculate distinct points one after another
    for i = 1:length(x)
        sum_top = 0;
        sum_bot = 0;
        for k = 0:n
            % We need to manually avoid a division by zero
            % While this isn't exactly elegant, it
            % does solve the issue, though it should be
            % remembered that we are now inable to get a lower
            % error than 10--15
            if x(i) == x_j(k+1)
                    x(i) = x(i) + 0.000000000000001; % 10~-15
                    % Note that 10~-16 is so small, MatLab recognizes
                    % it as zero too
            end
            % Calculate the sum for each side of the fraction...
            sum_top = sum_top + y(k+1) * w (k+1)/(x (i)-x_j(k+1));
            sum_bot = sum_bot + w(k+1)/(x(i)-x_j(k+1));
        end
        % ...and at the end divide the two.
        out_array(i) = sum_top / sum_bot;
    end
end
```


## Barycentric_Weights.m

```
function w = Barycentric_Weights(X)
    % This function produces the barycentric weights from the
    % input nodes. Remember, only nodes determine the weights,
    % not the data at the nodes
    w = ones(length(X),1);
    for J = 2:length(w)
        for K = 1:J-1
            W}(\textrm{K})=\textrm{w}(\textrm{K})*(\textrm{X}(\textrm{K})-\textrm{X}(\textrm{J}))
            W}(\textrm{J})=\textrm{W}(\textrm{J})*(X(J)-X(K))
        end
    end
    W = 1.// w;
```


## Cardinal_Trigonometric_Array.m

```
function out_arr = Cardinal_Trigonometric_Array(nodes,y,x)
    % This is the cardinal basis approach to trig. interpolation
    % The output has been changed to an array of discrete numbers
    % rather than an anonymous function
    % Ensure that the nodes are 2k/N spaced, and remember to do
    % the same affine transformations on both sets of nodes
    [nodes,transforms] = Affine_Transform_Eq(nodes);
    x = (x+transforms(1))*transforms(2);
    N = length(nodes);
    out_arr = zeros(1,length(x));
    % The outer loop simply cycles through every value of x,
    % instead of creating an anonymous function we simply
    % calculate distinct points one after another
    for i = 1:length(x)
        % The sum for a single evaluated point
        sum = 0;
        for k = 1:N
            t_k = nodes(k);
            y_k = y(k);
            if not(x(i)==t_k)
                if rem(N,2)==1
                        tau_top = y_k * sin(N * pi *(x(i)-t_k)*0.5);
                        tau_bot = N * sin(pi*(x(i)-t_k)*0.5);
                        tau = tau_top / tau_bot;
                    else
                        tau_top = y_k * sin(N * pi *(x(i)-t_k)*0.5);
                        tau_bot = N * tan(pi*(x(i)-t_k)*0.5);
                        tau = tau_top / tau_bot;
                end
                else
                    tau = y_k;
            end
            sum = sum + tau;
        end
        out_arr(i) = sum;
    end
end
```

```
    Cheb_Spaced_Nodes.m
function out_nodes = Cheb_Spaced_Nodes(amount,st,en)
    % Creates an array of nodes that are Chebyshev spaced
    % If the start is bigger than the end, we flip them around
    % This function always returns an array that goes from the
    % smallest to the biggest number
    if st > en
        oldst = st;
        st = en;
        en = oldst;
    end
    n = amount-1;
    out_nodes = ones(1, amount);
    size = abs(st - en) / 2;
    for k = 0:n
        out_nodes (k+1) = - cos(k * pi / n);
    end
    out_nodes = (out_nodes * size) + size + st;
end
```


## Eq_Spaced_Nodes.m

```
function out_nodes = Eq_Spaced_Nodes(amount,st,en)
    % Creates an array of equally spaced nodes
    % IMPORTANT: Use Trig_Spaced_Nodes instead for trigonometric
    % interpolation
    % If the start is bigger than the end, we flip them around
    % This function always returns an array that goes from the
    % smallest to the biggest number
    if st > en
        oldst = st;
        st = en;
        en = oldst;
    end
    size = abs(st - en);
    out_nodes = ones(1, amount);
    for i = 1:amount
        out_nodes(i) = i;
    end
    out_nodes = ((out_nodes-1) * size / (amount-1)) + st;
end
```


## Trig_Spaced_Nodes.m

```
function out_nodes = Trig_Spaced_Nodes(amount,st,en)
    % While trigonometric interpolation does require equally
    % spaced nodes, they also have to be such that the end points
    % are *not* included. For example if we want to interpolate
    % the interval -2 to 2 with five points, we need the points
    % -1.6, -0.8, 0, 0.8, 1.6
    % So we specifically want to ensure that the distance between
    % the first and last is also equal, if we were to do a
    % periodic extension
    if st > en
        oldst = st;
        st = en;
        en = oldst;
    end
    % This section is the only change from Eq_Spaced_Nodes.m
    r_size = abs(st - en);
    st = st + 0.5*r_size/amount;
    en = en - 0.5*r_size/amount;
    size = abs(st - en);
    out_nodes = ones(1,amount);
    for i = 1:amount
        out_nodes(i) = i;
    end
    out_nodes = ((out_nodes-1) * size / (amount-1)) + st;
```

end

```
    Lagrange _Array.m
function out_arr = Lagrange_Array(nodes, data, intNodes)
    % This is the basic form of lagrange interpolation
    % The output has been changed to an array of discrete numbers
    % rather than an anonymous function
    t = nodes;
    x = intNodes;
    y = data;
    n = length(t)-1;
    out_arr = zeros(1,length(intNodes));
    % The outer loop simply cycles through every value of x,
    % instead of creating an anonymous function we simply
    % calculate distinct points one after another
    for i = 1:length(intNodes)
        sum = 0;
        for k = 0:n
            % First we calculate the cardinal basis
            nt_k = [0:k-1 k+1:n];
            ell_k= prod(x(i)-t(nt_k+1))/prod}(t(k+1)-t(nt_k+1))
            % Then we make one big sum for the final polynomial
            y_val = y(k+1);
            sum = sum + y_val * ell_k;
        end
        out_arr(i) = sum;
    end
end
```


## A. 2 Computational Cost

## CC_Lagrange.m

```
% This file is used to test the speed of standard Lagrange.
Nconst = 10;
Neval = 50; % This value is set very low since Lagrange is SLOW
iterations = 50;
Nadd = 10; % By how much we increase the nodes each iteration
Nlist = zeros(1,iterations);
meanlist = zeros(1,iterations);
xeval = Eq_Spaced_Nodes(Neval, -2,2);
for k = 1:iterations
    x = Cheb_Spaced_Nodes(Nconst, -2,2);
    y = rand(1,Nconst);
```

```
    % We run the time test a few times and then take the average,
    % to help remove any noise from the calculation
    timearr = zeros(1,10);
    for i = 1:10
        tic
        LP = Lagrange_Array(x,y,xeval);
    timearr(i) = toc;
    end
    meanlist(k) = mean(timearr);
    Nlist(k) = Nconst;
    Nconst = Nconst + Nadd;
end
plot(Nlist,meanlist,'+');
xlabel("Amount of nodes");
ylabel("Mean time (seconds)");
set(gca,'fontsize',12);
```


## CC BL.m

```
% This file runs three time trials one after another, first it
% does only the barycentric weights, then only the barycentric
% "polynomial", and then finally both at the same time (the way
% it would be used if we had to always update the weights). The
% final run is therefore *not* just an addition between the
% results of the first two runs, but in theory it should be
% exactly that
Nstart = 100; % The starting amount of nodes for each algorithm
iterations = 200; % How many times we increase the node size
Nadd = 20; % By how much we increase the nodes each iteration
MeanAmount = 20; % How many times to re-run tests for stability
Neval = 2000; % How many points to evaluate
%___BARYCENTRIC WEIGHTS___
N = Nstart;
meanlist = zeros(1,iterations);
Nlist = zeros(1,iterations);
for k = 1:iterations
    x = Cheb_Spaced_Nodes (N, -2,2);
    y = rand (1,N);
    timearr = zeros(1,MeanAmount);
    for i = 1:MeanAmount
        tic
        w = Barycentric_Weights(x);
        timearr(i) = toc;
    end
    meanlist(k) = mean(timearr);
```

```
    Nlist(k) = N;
    N = N + Nadd;
end
hold off
plot(Nlist,meanlist,'o');
xlabel("Amount of nodes");
ylabel("Mean time (seconds)");
set(gca,'fontsize',12);
hold on
%___BARYCENTRIC_POLYNOMIAL___
Nconst = Nstart;
Nlist = zeros(1,iterations);
meanlist = zeros(1,iterations);
xeval = Eq_Spaced_Nodes(Neval,-2,2);
for k = 1:iterations
    x = Cheb_Spaced_Nodes(Nconst, -2,2);
    y = rand(1,Nconst);
    w = Barycentric_Weights(x);
    timearr = zeros(1,MeanAmount);
    for i = 1:MeanAmount
        tic
        BLP_arr = Barycentric_Lagrange_Array(x,y,w,xeval);
        timearr(i) = toc;
    end
    meanlist(k) = mean(timearr);
    Nlist(k) = Nconst;
    Nconst = Nconst + Nadd;
end
plot(Nlist,meanlist,'+');
%___BARYCENTRIC COMPLETE___
N = Nstart;
meanlist = zeros(1,iterations);
Nlist = zeros(1,iterations);
for k = 1:iterations
    x = Cheb_Spaced_Nodes(N,-2,2);
    y = rand(1,N);
    timearr = zeros(1,MeanAmount);
    for i = 1:MeanAmount
            tic
```

```
        w = Barycentric_Weights(x);
        BLP_arr = Barycentric_Lagrange_Array(x,y,w,xeval);
        timearr(i) = toc;
    end
    meanlist(k) = mean(timearr);
    Nlist(k) = N;
    N = N + Nadd;
end
plot(Nlist,meanlist,'.');
```

    CC_CT.m
    \% This file tests the speed of Cardinal trigonometric
\% interpolation, and then the speed of barycentric lagrange
\% (without predetermined weights)
Nstart $=10$; \% The starting amount of nodes for each algorithm
iterations $=100$; \% How many times we increase the node size
Nadd = 10; \% By how much we increase the nodes each iteration
MeanAmount $=20$; \% How many times to re-run tests for stability
Neval = 200; \% How many points to evaluate
\% ___CARDINAL_POLYNOMIAL__
$\mathrm{N}=$ Nstart;
meanlist $=$ zeros(1,iterations);
Nlist = zeros(1,iterations);
xeval = Eq_Spaced_Nodes (Neval,-2,2);
for $k=1: i t e r a t i o n s$
$\mathrm{x}=$ Eq_Spaced_Nodes $^{(N,-2,2)}$;
$\mathrm{y}=\mathrm{rand}(1, \mathrm{~N})$;
timearr = zeros(1, MeanAmount);
for $i=1$ : MeanAmount
tic
CTP_arr = Cardinal_Trigonometric_Array(x,y,xeval);
timearr (i) = toc;
end
meanlist(k) = mean(timearr);
Nlist(k) = N;
$\mathrm{N}=\mathrm{N}+\mathrm{Nadd}$;
end
hold off
plot(Nlist,meanlist,' + ' ) ;
xlabel("Amount of nodes");
ylabel("Mean time (seconds)");
set (gca,'fontsize', 12);
hold on

```
%___BARYCENTRIC_POLYNOMIAL___
Nconst = Nstart;
Nlist = zeros(1,iterations);
meanlist = zeros(1,iterations);
xeval = Eq_Spaced_Nodes(Neval, -2,2);
for k = 1:iterations
    x = Cheb_Spaced_Nodes(Nconst, - 2, 2);
    y = rand(1,Nconst);
    w = Barycentric_Weights(x);
    timearr = zeros(1,MeanAmount);
    for i = 1:MeanAmount
        tic
        %w = Barycentric_Weights(x);
        BLP_arr = Barycentric_Lagrange_Array(x,y,w,xeval);
        timearr(i) = toc;
        end
        meanlist(k) = mean(timearr);
        Nlist(k) = Nconst;
        Nconst = Nconst + Nadd;
end
plot(Nlist,meanlist,'o');
```


## A. 3 Error and stability

## Stability_Gibbs.m

```
% ___An example of the Gibbs phenomenon___
Nconst = 100;
Neval = 8000;
% Define the nodes (x) and the data (y)
x = Eq_Spaced_Nodes(Nconst,-pi,pi);
y = ones(1,Nconst);
xeval = Eq_Spaced_Nodes(Neval, -3*pi,3*pi);
% Do a loop instead of the sign(x) to ensure that there is no y=0
for i=1:Nconst
    if i<=Nconst/2
        y(i)=-1;
    end
end
f = @(x) sign(x);
% ____CALCULATION
TP = Cardinal_Trigonometric_Array(x,y,xeval);
```

```
% ___GRAPHICS__
border = [x(1)*4,x(length(x))*4];
hold off
plot(x,y,'o');
hold on
fplot(f,border);
plot(xeval,TP,'b');
xlabel("x");
ylabel("f(x)");
axis([-3*pi, 3*pi, -1.5, 1.5]);
set(gca,'fontsize',14);
```


## Stability _Runge.m

```
% ___An example of the Runge phenomenon__-
Nconst = 400;
Neval = 1000;
% Define the nodes (x) and the data (y)
x_eq = Eq_Spaced_Nodes(Nconst,-2,2);
x_ch = Cheb_Spaced_Nodes(Nconst, -2,2);
y_eq = 1./(1 + 25 * x_eq.^2);
y_ch = 1./(1 + 25 * x_ch. - 2);
xeval = Eq_Spaced_Nodes(Neval,-2,2);
% ___CALCULATION
w_eq = Barycentric_Weights(x_eq);
BLP_eq = Barycentric_Lagrange_Array(x_eq,y_eq,w_eq, xeval);
w_ch = Barycentric_Weights(x_ch);
BLP_ch = Barycentric_Lagrange_Array(x_ch,y_ch,w_ch,xeval);
% ___GRAPHICS___
border = [x_eq(1), x_eq(length(x_eq))];
hold off
plot(x_ch,y_ch,'o');
hold on
plot(x_eq,y_eq,'o');
plot(xeval, BLP_eq,'r',Linestyle=' --');
plot(xeval, BLP_ch,'b',Linestyle='--');
xlabel("x");
ylabel("f(x)");
set(gca,'fontsize',14);
```

```
    Error_Calculation_BL.m
% Error Calculation for Barycentric Lagrange interpolation
hold off
Nconst = 10;
Neval = 8000;
iterations = 140;
err_list = zeros(1,iterations);
node_list = zeros(1,iterations);
x_eval = Eq_Spaced_Nodes(Neval,-pi,pi);
for i = 1:iterations
    x = Cheb_Spaced_Nodes(Nconst,-pi,pi);
    x_eval = Eq_Spaced_Nodes(Neval,-pi,pi);
    y = exp(sin(pi.*x)-2.*\operatorname{cos}(pi*x));
    y_eval = exp(sin(pi.*x_eval)-2*cos(pi.*x_eval));
    w = Barycentric_Weights(x);
    BLP_arr = Barycentric_Lagrange_Array(x,y,w,x_eval);
    err_list(i) = norm(BLP_arr-y_eval,2);
    node_list(i) = Nconst;
    % Depending on the problem, either add an integer to Nconst
    % or multiply Nconst with an integer
    % Nconst = Nconst * 2;
    Nconst = Nconst + 2;
end
hold off
%loglog(node_list, err_list,'-')
semilogy(node_list, err_list,'-')
xlabel("x");
ylabel("f(x)");
set(gca,'fontsize',14);
```


## Error_Calculation_CT.m

```
% Error calculation for Cardinal Trigonometric interpolation
```

hold off
Nconst = 10;
Neval = 1000;
iterations = 40;
err_list = zeros(1,iterations);
node_list = zeros(1,iterations);
for i = 1:iterations
$\mathrm{x}=\mathrm{Tr} \mathrm{g}_{-}$Spaced_Nodes (Nconst, $-1,1$ ) ;
x_eval = Eq_Spaced_Nodes(Neval,-1,1);
$\mathrm{y}=\exp (\sin (\mathrm{pi} \cdot * \mathrm{x})-2 \cdot * \cos (\mathrm{pi} * \mathrm{x}))$;

```
    y_eval = exp(sin(pi.*x_eval) - 2* cos(pi.*x_eval));
    TP_arr = Cardinal_Trigonometric_Array(x,y,x_eval);
    err_list(i) = norm(TP_arr-y_eval,2);
    node_list(i) = Nconst;
    % Depending on the problem, either add an integer to Nconst
    % or multiply Nconst with an integer
    % Nconst = Nconst * 2;
    Nconst = Nconst+1;
end
hold off
%loglog(node_list,err_list,' _')
semilogy(node_list, err_list,' -')
xlabel("nodes");
ylabel("L2 error");
set(gca,'fontsize',14);
```


## A. 4 Higher dimensions (functions and programs)

```
Lagrange_Array_2D.m
function out_arr = Lagrange_Array_2D(u, v, data, u_ev, v_ev)
    % This is the basic form of a 2D Lagrange interpolation
    n = length(u)-1;
    m = length(v)-1;
    out_arr = zeros(length(u_ev),length(v_ev));
    % Outer sums are for evaluation
    for k = 1:length(u_ev)
        for r = 1:length(v_ev)
            sum = 0;
            % Inner sums are the actual interpolations
            for i = 0:n
                for j = 0:m
                    % First we calculate the cardinal bases
                    nt_i = [0:i-1 i+1:n];
                    ell_i_top = prod(u_ev(k)-u(nt_i+1));
                    ell_i_bot = prod(u(i+1)-u(nt_i+1));
                ell_i = ell_i_top/ell_i_bot;
                    mt_j = [0:j-1 j+1:m];
                ell_j_top = prod(v_ev(r)-v(mt_j+1));
                ell_j_bot = prod(v(j+1)-v(mt_j+1));
                ell_j = ell_j_top/ell_j_bot;
                % Then we make a big sum for the final polynomial
```

```
                    y_val = data(i+1,j+1);
                    sum = sum + y_val * ell_i * ell_j;
                    end
            end
            out_arr(k,r) = sum;
        end
    end
end
```


## Cardinal_Trigonometric_Array_2D.m

function out_arr = Cardinal_Trigonometric_Array_2D (u, v, d, uev, vev)
\% This function does Cardinal Trigonometric interpolation
\% with two parameters. The same principles used in
\% Lagrange_Array_2D apply here.
[u,utransforms] = Affine_Transform_Eq(u);
uev = (uev+utransforms (1)) *utransforms (2);
[v, vtransforms] = Affine_Transform_Eq(v);
vev $=$ (vev+vtransforms (1)) *vtransforms (2);
$\mathrm{n}=$ length (u);
$\mathrm{m}=$ length (v) ;
out_arr = zeros(length(uev), length(vev));
\% Outer sums are for evaluation
for $k=1: l e n g t h(u e v)$
for $r=1: l e n g t h(v e v)$
sum $=0$;
\% Inner sums are the actual interpolations
for $i=1: n$
for $j=1: m$
if $\operatorname{not}(u(i)==u e v(k))$
if $r e m(n, 2)==1$
tau_top $=\sin (\mathrm{n} * \mathrm{pi} *(\mathrm{uev}(\mathrm{k})-\mathrm{u}(\mathrm{i})) * 0.5)$;
tau_bot $=\mathrm{n} * \sin (\mathrm{pi} *(\mathrm{uev}(\mathrm{k})-\mathrm{u}(\mathrm{i})) * 0.5)$;
tau_u = tau_top / tau_bot;
else
tau_top $=\sin (\mathrm{n} * \mathrm{pi} *(\mathrm{uev}(\mathrm{k})-\mathrm{u}(\mathrm{i})) * 0.5)$;
tau_bot $=\mathrm{n} * \tan (\mathrm{pi} *(\mathrm{uev}(\mathrm{k})-\mathrm{u}(\mathrm{i})) * 0.5)$;
tau_u = tau_top / tau_bot;
end
else
tau_u = $1 ;$
end
if $\operatorname{not}(v(j)==\operatorname{vev}(r))$
if $r e m(m, 2)==1$
tau_top $=\sin (m * \operatorname{pi} *(v e v(r)-v(j)) * 0.5) ;$
tau_bot $=m * \sin (p i *(v e v(r)-v(j)) * 0.5) ;$
tau_v = tau_top / tau_bot;
else

```
                tau_top = sin(m * pi *(vev(r)-v(j))*0.5);
                    tau_bot = m * tan(pi*(vev(r)-v(j))*0.5);
                    tau_v = tau_top / tau_bot;
                        end
                else
                    tau_v = 1;
                    end
                        % Then we make a big sum for the final polynomial
                y_val = d(i,j);
                    sum = sum + y_val * tau_u * tau_v;
                    end
            end
                out_arr(k,r) = sum;
        end
    end
end
```

Interpolating_Cats.m

```
cat_pts =[20.89 -26.67
19.99 -29.56
16.93 -28.66
14.4 -26.31
12.42 -23.07
12.06 -20
10.08 -16.94
7.19 -17.3
4.67 -20.55
2.32 -25.23
6.65 -26.13
7.37 -28.84
2.68 -28.66
-2.54 -28.66
-8.13 -28.66
-13.54 -28.84
-15.88 -27.04
-21.11 -28.3
-26.7-30.46
-30.13 -33.17
-29.77 -36.05
-28.32 -39.11
-32.11 - 37.67
-34.09 -35.33
-33.73 -30.64
-30.85 -28.3
-27.06 - 26.67
-23.28 -25.05
-19.85 -22.53
-17.51 -19.64
-16.61 -16.22
```

```
32 -16.07 -12.97
33-15.16 -9.91
4-14.26 -7.02
-12.46 -3.96
-9.57 0.01
-6.33 3.25
-2.18 5.42
2.32 6.68
7.37 7.40
10.44 8.84
10.08 12.99
8.81 17.67
8.09 21.82
9.54 23.98
12.78 21.82
16.39 20.56
20.35 23.08
22.52 25.61
24.32 22.72
24.5 18.94
25.94 15.51
25.4 11.18
22.7 10.64
23.42 6.86
23.96 3.07
23.24 -0.71
20.89 -3.6
17.83-7.02
16.39-11.17
15.85 -14.6
16.21 -19.28
17.83 -23.61
20.89 -26.67];
n = length(cat_pts);
Nconst = n;
Neval = 1000;
disp(Nconst)
theta = Eq_Spaced_Nodes(Nconst,0, 2*pi);
teval = Eq_Spaced_Nodes(Neval,0,2*pi);
x = cat_pts(:,1);
y = cat_pts(:,2);
TP_x = Cardinal_Trigonometric_Array(theta,x,teval);
TP_y = Cardinal_Trigonometric_Array(theta,y,teval);
hold off
plot(x,y,'o')
hold on
```

```
plot(TP_x,TP_y,'r')
xlabel("x");
ylabel("y");
set(gca,'fontsize',14);
```


## Multi Dim Limacon.m

```
% This is an example of the limacon, a sort of circular thing
% that can intersect itself.
% The code is easily modified to create a spiral instead,
% or to make an example of the runge phenomenon.
Nconst = 5;
Neval = 1000;
theta = zeros(2,Nconst);
% First we create all of the theta arrays
theta(1,:) = Trig_Spaced_Nodes(Nconst,0, 2*pi);
theta(2,:) = Cheb_Spaced_Nodes(Nconst,0, 2*pi);
teval = Eq_Spaced_Nodes(Neval,0,2*pi);
% Then we use theta to create x and y
x = (2.2+5.5*\operatorname{cos}(theta)).*\operatorname{cos}(theta);
y = (2.2+5.5*\operatorname{cos(theta)).*sin(theta);}
w_t = Barycentric_Weights(theta(2,:));
% And last we construct two sets of interpolation arrays, one for
% each coordinate
TP_x = Cardinal_Trigonometric_Array(theta(1,:),x(1,:),teval);
TP_y = Cardinal_Trigonometric_Array(theta(1,:),y(1,:),teval);
ch_BLP_x = Barycentric_Lagrange_Array(theta(2,:), x(2,:),w_t,teval);
ch_BLP_y = Barycentric_Lagrange_Array(theta(2,:),y(2,:),w_t,teval);
hold off
plot(x(2,:),y(2,:),'O')
hold on
plot(x(1,:),y(1,:),'+')
plot(x(1,:),y(1,:),'+')
plot(TP_x,TP_y,'r')
plot(ch_BLP_x,ch_BLP_y,'b')
```


## Multi Dim 3D.m

```
% Parametric equations for a Mobius strip at the origin:
    X = (1 + V/2.*\operatorname{cos}(U/2) ).*\operatorname{cos}(u)
    Y = (1 + V/2.*\operatorname{cos}(U/2) ).*sin(u)
    Z = V/2.*sin(u/2)
    on the intervals 0 <= u <= 2*pi and - < <= v = 1.
    The parametric equations are periodic in u and not in v.
% 
% Currently the file is using the function for two spheres:
```

```
% X = cos(U).*\operatorname{cos(V);}
% Y = sin(U).*\operatorname{cos}(V);
% Z = sin(V);
%
% on the intervals 0 <= u <= 2*pi and -pi <= v = pi.
N = 8;
M = 8;
u = Trig_Spaced_Nodes(N + 1, 0, 2*pi);
% Change this interval to -pi/2, pi/2 for a single sphere instead
v = Trig_Spaced_Nodes(M + 1, -pi, pi);
ueval = Eq_Spaced_Nodes(100, 0, 2*pi);
veval = Eq_Spaced_Nodes(100, -pi/2, pi/2);
[U,V] = meshgrid(u,v);
U = U';
V = V';
% Create gridded data for the different interpolations
X = cos(U).*\operatorname{cos(V);}
Y = sin(U).*\operatorname{cos(V);}
Z = sin(V);
% Plot the original curve
surf(X,Y,Z)
title("Double Sphere: original data")
figure
% Interpolate
P_x = Lagrange_Array_2D(u, v, X, ueval, veval);
P_y = Cardinal_Trigonometric_Array_2D(u, v, Y, ueval, veval);
P_z = Cardinal_Trigonometric_Array_2D(u, v, Z, ueval, veval);
% Plot the interpolation
surf(P_x, P_y, P_z)
axis([-1,1,-1,1,-1,1])
title("Double Sphere: interpolated data")
```


## Copyright

The publishers will keep this document online on the Internet - or its possible replacement - from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/her own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

## Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.
(C) 2022, Beatrice Söderqvist

