Global Surfaces of Section and the Hopf Fibration

Heidelberg University

Anna Ziegler

17 September 2021

Contents

Chapter 1

Introduction

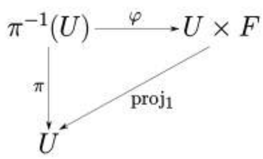
This bachelor project builds on the paper 'A Symplectic Dynamics Proof of the Degree-Genus Formula' by Peter Albers, Hansjörg Geiges and Kai Zehmisch [1]. Our focus will be on the first part of the paper, which deals with classifying global surfaces of section for the Hopf fibration. We will start by covering the basic mathematical concepts to then visualize the Hopf fibration using the programming language Processing, and finally identify and visualize various d-sections.

1.1 Fibre Bundles

1.1.1 What's a Fibre Bundle?

First, let us investigate what a fibre bundle is. Because, unlike its name, the Hopf fibration is not only a fibration, but a fibre bundle, which is more constrained.

Definition 1. A **fibre bundle** is a structure (E, B, π, F). Here E,B,F are topological spaces and $\pi : E \to B$ is a continuous surjection. E is called the total space, B the base space and F the fibre. π is the bundle projection satisfying the local triviality condition: For every $x \in B$, there is an open neighbourhood $U \subset B$ of x such that there is a homeomorphism $\varphi : \pi^{-1}(U) \to U \times F$ so that the following diagram should commute:



That means that a fibre bundle is a space that locally looks like a product, but globally it can look differently. Thus, a Cartesian Product of two spaces is the easiest example of a fibre bundle, since it also globally is a product. A more sophisticated example is the Möbius Strip: locally it looks like a Cartesian Product of a circle with a line, but it has a global twist. Both are depicted in figure 1.1.

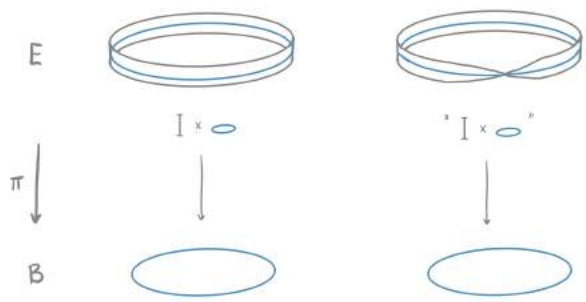


Figure 1.1: The Cartesian Product and the Möbius strip

1.1.2 The Hopf Fibration

For the Hopf Fibration, we have the total space being $E = S^3 \subset \mathbb{C}^2$. Interestingly, we can obtain that sphere by a twisted product of S^1 with S^2.

Definition 2. The **Hopf Fibration** is a fibre bundle given by: (S^3, S^2, H, S^1) with the Hopf Map H defined as

$$H : S^3 \subset \mathbb{C}^2 \to \mathbb{C} \cup \{\infty\} \cong S^2$$

$$(z_1, z_2) \mapsto \frac{z_1}{z_2}$$

or, alternatively:

$$H : S^3 \subset \mathbb{C}^2 \to \mathbb{C}P^1$$

$$(z_1, z_2) \mapsto z_1 : z_2$$

In this thesis, we will refer to the Hopf Map as the first of the two definitions, and identify $\mathbb{C} \cup \{\infty\}$ with the Riemann Sphere as defined below.

Definition 3. The **Riemann Sphere** is the compactification of the complex plane, which is achieved by adding the point ∞ to the complex plane. The mapping consists of the identity on $\mathbb{C}$ and the second map on $(\mathbb{C} \cup \infty) \setminus \{0\}$

$$z \mapsto \begin{cases} \frac{1}{z} & \text{for } z \in \mathbb{C} \\ 0 & \text{for } z = \infty \end{cases}$$

1.1.3 The Hopf Flow as a Reeb Flow

Another way of obtaining the Hopf Fibration is using the Reeb flow of the standard contact form α_{st} on $S^3 \subset \mathbb{R}^4$ given by

$$\alpha_{st} = (x_1 dy_1 - y_1 dx_1 + x_2 dy_2 - y_2 dx_2)|_{TS^3}$$

Definition 4. In a contact Manifold, given a contact-1-form α, the **Reeb Vector-field** R is the vector field satisfying

$$R \in ker(d\alpha)$$

$$\alpha(R) = 1$$

The Reeb vector field of α_{st} is thus given by:

$$R_{st} = x_1\partial_{y_1} - y_1\partial_{x_1} + x_2\partial_{y_2} - y_2\partial_{x_2}$$

The Hopf Flow being the Reeb flow of α_{st} implies lemma 5. For more information about contact forms, Reeb vector fields and everything related, see [2].

LEMMA 5. The **Hopf Flow** is defined as $\Phi_R^t(p) = e^{it} \cdot p$

Intuitively, this means that both copies of $\mathbb{C}$ get rotated simultaneously. The orbits of the Hopf flow are the fibres of the Hopf fibration. Using the Hopf map from definition 2, we can verify this claim, since points on the same orbit map to the same point on the Riemann sphere:

$$H(\Phi_R^t(z_1, z_2)) = H((e^{it}z_1, e^{it}z_2)) = \frac{e^{it} \cdot z_1}{e^{it} \cdot z_2} = \frac{z_1}{z_2} = H((z_1, z_2))$$

REMARK 6. An interesting property of the Hopf fibration is that any two Hopf fibres are linked by a so called Hopf link. This means, they are linked like two parts of a chain, as you can see in figure (1.2).

Figure 1.2: The Hopf link

1.1.4 Projecting the Hopf Fibration into $\mathbb{R}^3$

To get an intuition about how the Hopf fibration behaves, it is useful to project it into three dimensions. This can be done using the stereographic projection. Fortunately,

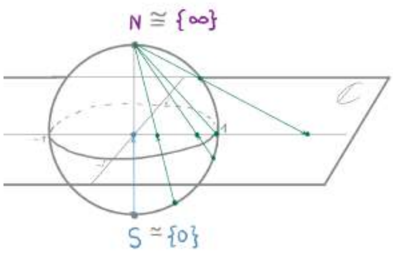


Figure 1.3: The Riemann Sphere

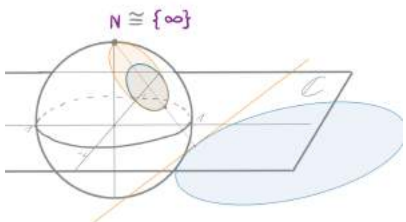


Figure 1.4: The circle-preserving 2D stereographic projection

this map is circle-preserving, which means that circles will be mapped onto circles. This property is very handy for our purpose since we want to look at the S^1-fibres of the Hopf fibration. Just like in the 2D-case (figure 1.4), the only circles that are not mapped to circles are those which go through the north pole of the projection.

They are mapped to straight lines, which we can interpret as 'infinitely big' circles. In the diagram below, you can see how the projection works: On the left you can see $S^3 \subset \mathbb{C}^2$. The Hopf map projects fibres of S^3 to points on the Riemann Sphere on the top right. Whereas, the stereographic projection maps S^3 to $\mathbb{R}^3 \cup \infty$, which you can see on the bottom right.

Let us go through the diagram, to get a closer look at how the projection works.

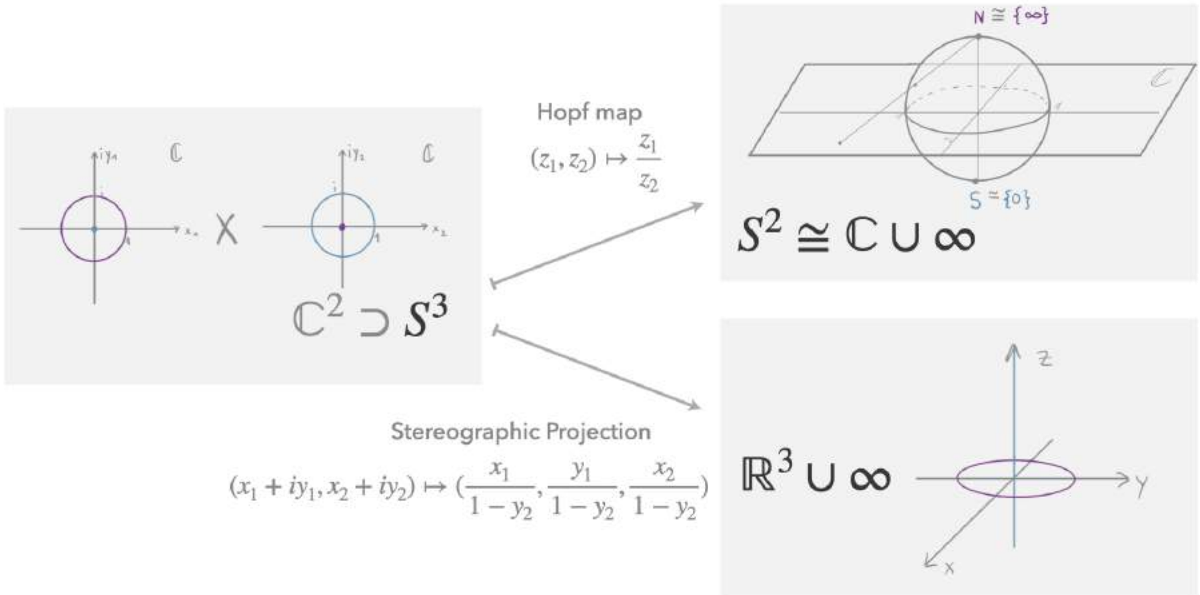


Figure 1.5: Projecting the Hopf Fibration into 3 Dimensions

We start by choosing a point on the Riemann Sphere, in this case the south pole, indicated in blue, and the north pole drawn in pink. Now we want to 'reverse' the Hopf map, to find the fibre that is projected onto these points by the map. For the north pole we compute:

$$\frac{z_1}{z_2} = \infty \implies z_2 = 0, |z_1| = 1$$

And analogously for the south pole:

$$\frac{z_1}{z_2} = 0 \implies z_1 = 0, |z_2| = 1$$

Thus, the north pole fibre is given by:

$$C_2 = \{(e^{i\theta}, 0) : \theta \in \mathbb{R}/2\pi\mathbb{Z}\}$$

and the south pole fibre respectively:

$$C_1 = \{(0, e^{i\theta}) : \theta \in \mathbb{R}/2\pi\mathbb{Z}\}$$

The reason for the names C_1 and C_2 will become clear in 1.2.1. Keep in mind that the sum of the absolute values of z_1 and z_2 have to equal 1, since we want to stay on S^3. You can see the fibres in S^3 in diagram 1.5 indicated in the matching colours. Having identified the fibres in $\mathbb{C}^2$, we use the stereographic projection σ with north pole $N_\sigma = (0, 0, 0, 1)$ to map them to $\mathbb{R}^3 \cup \infty$.

Definition 7. The **stereographic projection** σ with north pole $N_\sigma = (0,0,0,1)$ is defined as:

$$\sigma : S^3 \subset \mathbb{C}^2 \to \mathbb{R}^3 \cup \infty$$
$$(x_1 + iy_1, x_2 + iy_2) \mapsto (\frac{x_1}{1-y_2}, \frac{y_1}{1-y_2}, \frac{x_2}{1-y_2})$$

Using this projection, we obtain for $\theta \in \mathbb{R}/2\pi\mathbb{Z}$:

$$\sigma(C_1) = (\frac{0}{1-sin(\theta)}, \frac{0}{1-sin(\theta)}, \frac{cos(\theta)}{1-sin(\theta)}) = (0, 0, \frac{cos(\theta)}{1-sin(\theta)})$$

$$\sigma(C_2) = (\frac{cos(\theta)}{1}, \frac{sin(\theta)}{1}, \frac{0}{1}) = (cos(\theta), sin(\theta), 0)$$

Thus, we see that the two fibres we chose are projected onto the z-axis and onto a unit circle in the x-y-plane respectively. We can see this illustrated in figure 1.5: C_2 intersects the N_σ and thus gets projected onto a line.

This concludes the process. We will implement what we have learned in our code and choose the points on the Riemann Sphere systematically to study the behaviour of the Hopf fibration.

1.2 D-Sections

A valuable tool for studying the Hopf fibration is the concept of global surfaces of section, or d-sections. Intuitively speaking, we are searching for a surface bounded by a fibre or a union of fibres that will intersect all other fibres in our bundle exactly d times.

DEFINITION 8. A **global surface of section**, or d-section, for the flow of a non-singular vector field X on a three-manifold M is an embedded compact surface $\Sigma \subset M$ that has to meet three conditions:
i) the boundary $\partial\Sigma(M)$ is a union of orbits
ii) the interior $Int(\Sigma)$ is transverse to X
iii) the orbit of X through any point in $M \backslash \partial\Sigma$ intersects $Int(\Sigma)$ in exactly d points (in forward and backward time)
(from [1])

1.2.1 A Disc-Like 1-Section

Now we want to look at an example of a disc-like 1-section for the Hopf flow.
Before we go on, we will introduce the concept of the soul of a solid torus.

DEFINITION 9. Given a Torus $T = S^1 \times D^2$, the **soul** C_T **of V** is given by $C_T = S^1 \times \{0\}$

That means that the soul is the circle in the 'middle' of the torus as depicted in figure 1.6.

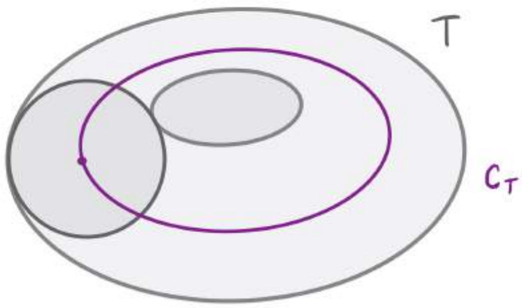


Figure 1.6: Soul C_T of a torus T

REMARK 10. Notice that we can build up a solid torus using a union of nested tori. We define them as $T_r = S^1 \times r * D^2$ with $0 < r \leq 1$. To obtain a solid torus, we need to add the soul to this union of tori:

$$T = \cup_{0<r\leq 1} T_r \cup C_T$$

LEMMA 11. S^3 can be obtained by glueing two solid tori together.

Proof. We define two tori $V_1, V_2 \subset S^3$ given by

$$V_1 := \{|z_1| \leq \tfrac{1}{\sqrt{2}}\} = \{|z_2| \geq \tfrac{1}{\sqrt{2}}\}$$

$$V_2 := \{|z_2| \leq \tfrac{1}{\sqrt{2}}\} = \{|z_1| \geq \tfrac{1}{\sqrt{2}}\}$$

The identification with $S^1 \times D^2$ is given by:

$$V_1 := \{(z, \sqrt{1-|z|^2 e^{i\theta}}) \ : \ |z| \leq \frac{\sqrt{2}}{2}, \theta \in \mathbb{R}\backslash 2\pi\mathbb{Z}\}$$

$$V_2 := \{(\sqrt{1-|z|^2 e^{i\theta}}, z) \ : \ |z| \leq \frac{\sqrt{2}}{2}, \theta \in \mathbb{R}\backslash 2\pi\mathbb{Z}\}$$

Clearly, the union of these two shapes form the whole of S^3. The souls C_1 of V_1 and C_2 of V_2 are given by the fibres mapping to the respective poles of the Riemann Sphere by the Hopf map. They are indicated in diagram 1.5 in pink and blue.

$$C_1 = \{(0, e^{i\theta}) : \theta \in \mathbb{R}/2\pi\mathbb{Z}\}$$

$$C_2 = \{(e^{i\theta}, 0) : \theta \in \mathbb{R}/2\pi\mathbb{Z}\}$$

In picture 1.7 you can see $\mathbb{R}^3$, with the two projected fibres in it. The sketch is

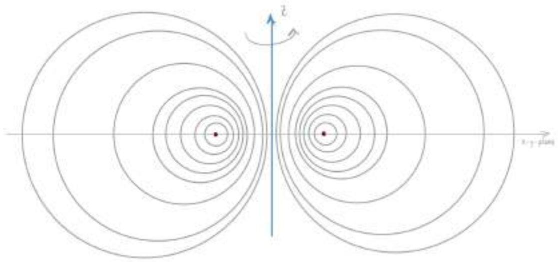


Figure 1.7: Two fibres of the Hopf Fibration being mapped into $\mathbb{R}^3$

symmetric under rotation, so the two blue points actually represent a circle and the grey circles indicate tori. Note that the total space $\mathbb{R}^3 \cup \{\infty\} \cong S^3$ without the two fibres can be obtained by a union of nested tori, like the ones indicated in grey. As we investigated in Remark 10, a solid torus can be obtained by a union of its soul and nested tori around this soul. This is precisely what we construct now: We take the blue soul C_1 and the pink soul C_2 and unite them with nested tori around them to obtain V_1 and V_2. This is illustrated in figure 1.8.

Although the blue fibre maps to a line in $\mathbb{R}^3$ using the stereographic projection, it is a circle in $\mathbb{C}^2$. If we chose a different north pole in the stereographic projection, for example $\tilde{N} = (1, 0, 0, 0)$, then we would get the same result, but with switched colours. In this case the pink fibre would be infinitely long and the blue fibre would be the circle in the x-y-plane. This shows the distortions to be only an artifact of the projection.

From this we can conclude that S^3 can be viewed as two solid tori glued together. S^3 is obtained by gluing the meridian of one torus onto the longitude of the other. □

Example 12. The disc-like surface Σ_N bounded by C_2 is a 1-section for the Hopf flow.

$$\Sigma_N = \{(re^{i\theta}, \sqrt{1-r^2}) : r \in [0,1], \theta \in \mathbb{R}/2\pi\mathbb{Z}\} \subset S^3$$

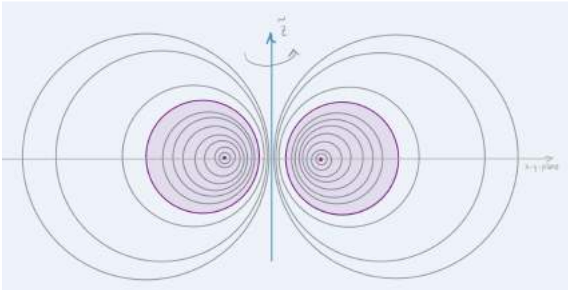


Figure 1.8: Mapping of S^3 with the two solid tori, V_1 indicated in blue, V_2 indicated in pink

Proof. Since the boundary is given by C_2, we can clearly see that it consists of one fibre, proving (i) in definition 8. As we discussed in remark 6, we know that each two Hopf fibres are linked by a Hopf link. Thus, we can infer that all other fibres also interlink with C_2 in a Hopf link. This implies that they must intersect $Int(\Sigma_N)$ exactly once, satisfying (ii) and (iii). In other words, it forms a 1-section for the Hopf fibration.
We provide a more rigorous proof in lemma 13. □

Chapter 2

Visualizing the Hopf Fibration

Now we want to implement the above in computer code in order to visualise the Hopf Fibration. The programming language of our choice is Processing, a visually oriented language based on Java.

NOTATION. To refer to the number stored in the variable 'Amount' we use the notation n_{Amount}. Variables will be indicated in italic: *Variable*.
To indicate the size of an $m \times k$ array with name ArrayName, we will use the notation $ArrayName[m][k]$.

2.0.1 Libraries

We import the library QScript [3], which includes a class for complex numbers and 3-dimensional vectors. For the points in $\mathbb{C}^2$ we include a custom class called CxComplex (Appendix B). All functions related to the projection of the Hopf fibration you can find in Appendix C. Furthermore, all functions related to d-sections are collected in Appendix D. We use the class Shapes3D [4] for the tube object, which we use to show the Hopf flow. In Appendix E you find everything related to the rotations of S^3. For the graphical functions that organize our window and create the graphical user interface, see Appendix F. Another class that we use for this purpose is called HScrollbar [5] (Appendix G), it provides the scrollbars. Finally, the class rec() [6] (Appendix H) is included to be able to record videos directly from the code.

2.1 Choosing Points on the Riemann Sphere

We implement various methods of choosing points on the Riemann Sphere. Firstly,

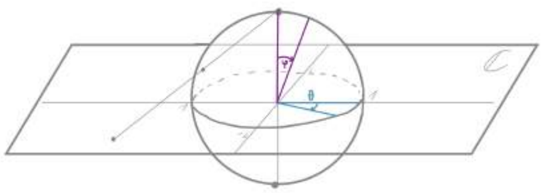


Figure 2.1: Angles of the Riemann Sphere, φ indicated in pink, θ in blue

we include functions for varying φ or θ for a given contrary angle. These input

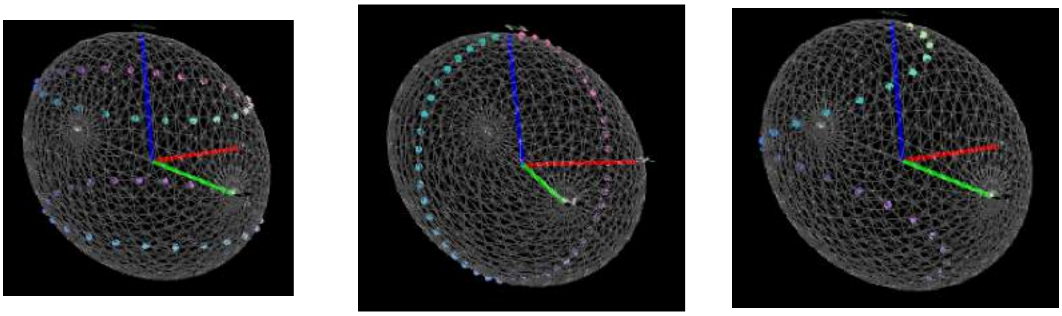


Figure 2.2: Choosing points on the sphere using `VaryTheta()`, `VaryPhi()` and `Spiral()`

angles are changeable with scrollbars. Additionally, we include a 'Spiral'- Function that generates points in a spiral on the Riemann Sphere. This function also has a scrollbar which can be used to rotate the spiral around the z-axis of the Riemann Sphere, e.g. to vary θ.

2.2 The Projection

For this process, we will follow diagram 1.5. We start at the Riemann sphere, follow the Hopf Map back to $S^3 \subset \mathbb{C}^2$ and project this fibre to $\mathbb{R}^3 \cup \infty$ using the stereographic projection. We plot $n_{noCircles}$ different fibres, and this value will be changeable in real-time. The detail of the plot depends on *noPoints*, a variable that sets the number of points we compute per circle.

Notation. We will structure this part using the functions in the code. Their syntax is encoded as `Outputtype FunctionName(Inputtype)`.

`CxComplex findFiberPoint(Complex)`

In the first step, we take a complex number z and find one point on the corresponding fibre in S^3. This can be interpreted as a 'reverse Hopf map', but instead of giving us the whole preimage - that is, the whole fibre - we get only one point on it. To achieve this, we set the imaginary part of the second complex coordinate to 0. The computation of the remaining coordinates then looks as follows.
We take: $z = (z_1, z_2) \in \mathbb{C}$, $p = (p_1, p_2) = (x_1, y_1, x_2, y_2) \in \mathbb{C} \times \mathbb{C}$
Set $y_2 = 0$. We know that

$$\frac{p_1}{p_2} = z$$

and also

$$(x_1)^2 + (y_1)^2 + (x_2)^2 + (y_2)^2 = 1$$

since $p \in S^3$. Thus we can infer:

$$x_2 = \sqrt{\frac{1}{1 + (z_1)^2 + (z_2)^2}}$$

$$x_1 = z_1 \cdot x_2$$

$$y_1 = z_2 \cdot x_2$$

The function returns this point in $\mathbb{C}^2$.

`CxComplex getNewPoint(CxComplex)`

Now, using the point $p \in \mathbb{C}^2$ that we obtained in the previous step, we follow the Hopf flow to acquire a new point on the fibre.
As a reminder, the Hopf Flow for $p \in \mathbb{C}^2$ is defined as:

$$\Phi_p^t = e^{it} \cdot p$$

The function returns Φ_p^t for $t = \frac{2\pi}{noPoints}$, where *noPoints* defines how many points per circle we want to compute. We use this function recursively to obtain $n_{noPoints}$ equidistant points on the fibre.

`Vector projectPoint(CxComplex)`

Having obtained points on the fibre we want to visualize, we now use the stereographic projection σ to map them to $\mathbb{R}^3 \cup \infty$.

$$\sigma : (x_1, y_1, x_2, y_2) \mapsto (\frac{x_1}{1-y_2}, \frac{y_1}{1-y_2}, \frac{x_2}{1-y_2})$$

This function returns the points' coordinates in $\mathbb{R}^3$. There is a distortion of the circles that can appear especially when the circles get larger and for small *noPoints*. This is explained by the fact that the projection does not preserve the equidistance of the points - most points get mapped close to the origin, and thus only to one side of the circle.

`void fillArray()`

This function combines all of the above to create one array holding all the points in $\mathbb{R}^3$ that we need to visualize the fibres. First, we choose $n_{noCircles}$ points on the Riemann Sphere. Then it creates an array of points in $\mathbb{C}^2$ and fills it with one point on each fibre using `findFibrePoint()`. Next it fills up the array *circlePoints[noCircles][noPoints]* like this:
Let us call the points on the Riemann Sphere $z_1, z_2, ..., z_{noCircles}$ and `findFibrePoint(`z_i`)` $= c_i$, and we shorten the functions' names to `project()` and `getNew()`. Then the matrix is filled in the following manner.

	1	*2*	*3*	...	*noPoints*
1	project(c_1)	project(getNew(c_1))	project(getNew(getNew(c_1))	...	project(getNew(...getNew(c_1)...)
2	project(c_2)	project(getNew(c_2))	project(getNew(getNew(c_2))	...	project(getNew(...getNew(c_2)...)
3	project(c_3)	project(getNew(c_3))	project(getNew(getNew(c_3))	...	project(getNew(...getNew(c_3)...)
...	...	...	...	...	...
noCircles	project($c_{noCircles}$)	project(getNew($c_{noCircles}$))	project(getNew(getNew($c_{noCircles}$))	...	project(getNew(...getNew($c_{noCircles}$)...)

Table 2.1: `fillArray()` filling up *circlePoints[noCircles][noPoints]*

`drawColCircle()`

For visualizing the projected fibres, we use the array that we received using `fillArray()`. The image is obtained by plotting an arc through the points in each row.
Additionally, we colour the circle and the corresponding point on the Riemann Sphere in the same shade, so we can identify the connections easily. The colour scheme for the j-th circle, whose points are stored in *circlePoints[j][0], circlePoints[j][1],...,circlePoints 1]* looks as follows.

```
R = circlePoints[j][0].x*105+150,
G = circlePoints[j][0].y*85+150,
B = circlePoints[j][0].z*70+160,
```

The RGB values we compute build on the x,y and z coordinates of the first projected point of the circle. Because of the black background, we add an integer, so the resulting colours are bright enough to be visible. Additionally, we multiply the values by a factor, to change how much the colour varies with the changing coordinate. Since these are the points with $y_2 = 0$ and $x_2 \geq 0$ (due to the function `findFibrePoint()`), we find that the projected x,y and z-values will lie on the unit sphere in $\mathbb{R}^3$, resulting in $x \in [-1, 1]$, $y \in [-1, 1]$, $z \in [0, 1]$. With this knowledge, we infer that the RGB values resulting from this scheme will be $R \in [45, 255]$, $G \in [65, 235]$ and $B \in [160, 250]$. As you can see in figure 2.2 and in the following chapters, the colours are well visible and vary enough to point out the connection between the points on the Riemann Sphere and the fibres.

Chapter 3

Visualizing d-Sections

First, we want to focus on visualizing disc-like 1-sections, and later also 2-sections. In both cases, we will first find a specific d-section and then create various other d-sections by using rotations on S^3 and the Hopf flow. To implement this, we add functions to our Processing sketch collected in `D-Section library` (Appendix D).

3.1 Disc-Like 1-Sections

To visualize disc-like 1-sections, we first investigate the 1-sections Σ_N and Σ_S . Then we will implement Σ_N.

3.1.1 The Disc-Like North Pole 1-Section

This surface is called the north pole 1-section because its boundary is given by the fibre C_2. As we investigated before (figure 1.5), this circle gets mapped to the north pole of the Riemann Sphere by the Hopf map. To put this into formulas:

$$C_2 = \{(e^{i\theta}, 0) : \theta \in \mathbb{R}/2\pi\mathbb{Z}\}$$

$$\Sigma_N = \{(re^{i\theta}, \sqrt{1-r^2}) : r \leq 1, \theta \in \mathbb{R}/2\pi\mathbb{Z}\}$$

Lemma 13. Σ_N is a 1-section for the Hopf fibration.

Proof. We can obtain Σ_N as a meridional disc $D_{\frac{\sqrt{2}}{2}}$ in V_1 glued together with a helicoidal surface A in V_2. Firstly, we want to show that the meridional disc in V_1 intersects all Hopf fibres in that solid torus exactly once. The Hopf fibres in V_1 lie on the nested tori T_r as defined in Remark 9. They are rotating once around the longitude λ and once around the meridian μ of the torus that they lay on. To get an intuition, we depict one of these tori in figure 3.1 as a rectangle whose opposite edges are identified. The Hopf fibres are then given by the pink fibre h and by all possible shifted fibres, like the one indicated in blue. Since all Hopf fibres rotate once around the meridian and longitude at the same time, they intersect a meridional disc exactly once. An example of a meridional disc intersecting the torus in figure 3.1 is given as the left vertical edge of the rectangle. We can see clearly that this disc intersects each fibre exactly once, and that this is also true if we move the vertical line horizontally. Consequently, every meridional disc has this property. Thus, the

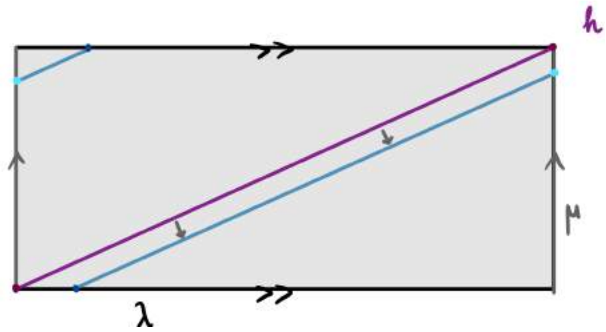


Figure 3.1: The Hopf fibres on an unfolded torus

meridional disc $D_{\frac{\sqrt{2}}{2}} = \{re^{i\theta} : r \leq \frac{\sqrt{2}}{2}, \theta \in \mathbb{R}/2\pi\mathbb{Z}\}$ intersects each Hopf fibre in V_1 exactly once. The boundary of this disc is given by $\mu_1 = \lambda_2$. We can embed it into V_1 as follows:

$$D_{\frac{\sqrt{2}}{2}} \to V_1 = D^2 \times S^1$$
$$re^{i\theta} \mapsto (re^{i\theta}, \sqrt{1-r^2})$$

Next, we proof that $A \in V_2$ is also intersecting all Hopf fibres in that solid torus exactly once. We define the helicoidal surface A with oriented boundary $C_2 \cup -(h - \mu_2) = C_2 \cup \lambda_2$. First we cut V_2 open at a meridonal disc and then we apply a Dehn Twist, so that the Hopf fibres in this torus correspond to $h = S^1 \times \{*\}$. The whole process is depicted in 3.2. λ_2 is now twisted once around the torus. Since the Hopf fibres in V_2 correspond to straight lines now, we can clearly see that they all intersect the surface exactly once. A can be described as $A = \{(re^{i\theta}, \theta) : r \in [0,1], \theta \in [0, 2\pi]\}$,

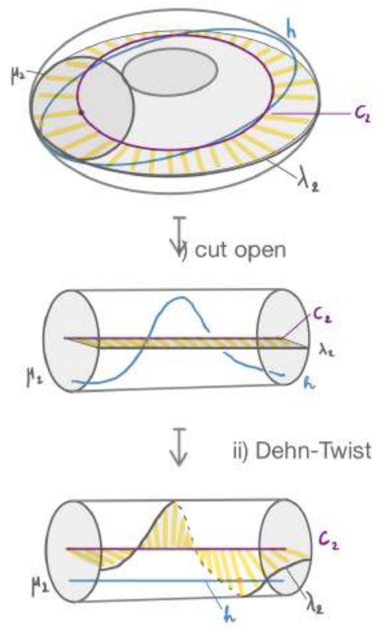


Figure 3.2: The yellow Helicoidal surface A in V_2

as you can see in figure 3.3. We embed A into V_2 as follows:

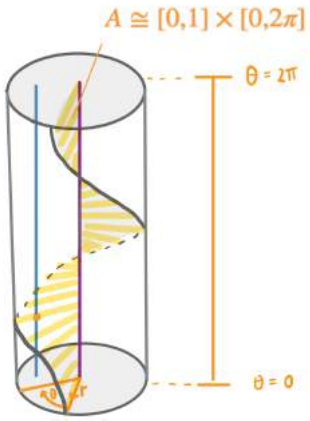


Figure 3.3: Embedding A into V_2

$$A = [0,1] \times [0,2\pi] \to V_2 = S^1 \times D^2$$
$$(r,\theta) \mapsto (\sqrt{1-\frac{1}{2}r^2}e^{i\theta}, \frac{\sqrt{2}}{2}r)$$

Hence these two surfaces glued together intersect each Hopf fibre in $S^3 = V_1 \cup V_2$ exactly once, proving (ii) and (iii) in definition 8. That means we only need to show (i) still. We glue the two surfaces together at λ_2, so that the glued surface has C_2 as a boundary, proving (i). This concludes our proof. □

3.1.2 The Disc-Like South Pole 1-Section

Similarly to the last section, we also investigate the special case of the disc-like 1-section bounded by the fibre $C_1 = \{(0, e^{it}) : t \in \mathbb{R}/2\pi\mathbb{Z}\}$. Since C_1 maps to the south pole of the Riemann Sphere by the Hopf map (the blue fibre in figure 1.5), we call this 1-section the south pole 1-section. Using the stereographic projection, the fibre gets mapped onto the z-axis in $\mathbb{R}^3$. Thus, the 1-section it "surrounds" is a half-plane. We include a special function `void drawSouthernDSectionBoundary()` for this case, in which we chose one half-plane for the visualization (see figure 3.4).

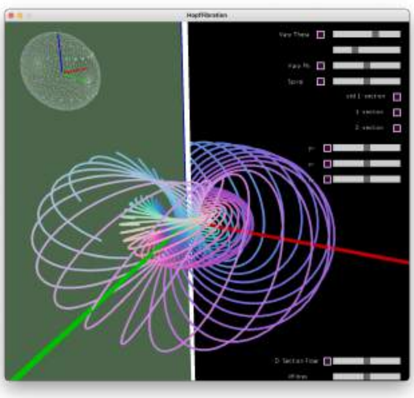


Figure 3.4: The South Pole 1-Section Σ_S

3.1.3 Implementing the North Pole 1-Section

For visualizing Σ_N, we first need to initialize a grid of points on the 1-section. We implement two different methods for this task.
The first method is using `CxComplex[][] getDSectionGrid(int noColumns, int noRows, CxComplex p)`. In figure 3.5 you can see how we set up the grid. First, we choose one point on the boundary fibre, p, as a starting point. This point will be stored at [0][0] in our array. Since the boundary is one fibre, we can add the other points on the boundary fibre using the Hopf flow (Lemma 5). We vary our angle from $\alpha = 0$ to $\alpha = \frac{3}{4}\pi$ to fill the first row of the array with equidistant points, pink in the picture. Just in the same way, the interval $\alpha \in [\frac{3}{4}\pi, \pi]$ will fill the last column of the matrix with the blue points, and $\alpha \in [\pi, \frac{7}{4}\pi]$ provides us with the last row indicated in pink, and the points on the rest of the boundary will fill in the first column.

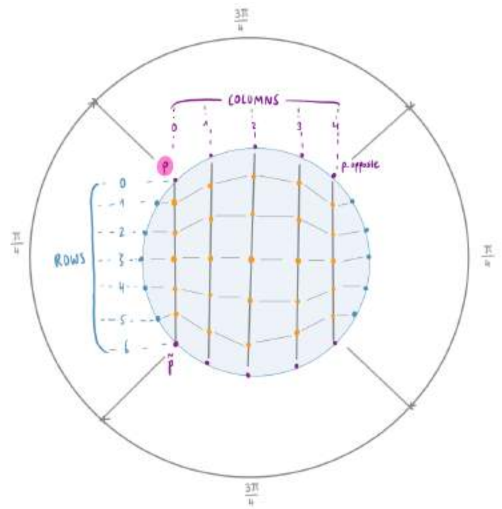


Figure 3.5: Setting up a Grid of the d-Section

After filling in the boundary, we compute the points in the interior of the surface. In figure 3.5 we find them coloured orange. Going through the columns, we fill in equidistant points on the line between the first and last entry. Additionally, we normalize the points in such a way that they lie again on our d-section. Since the boundary points all lay on C_2, the second complex coordinate is always 0, and that is also true for the points on the line between them. Therefore the function `normalize2ndCoordinate()` normalizes the point to be of length 1 (and thus to be in S^3) by adapting the second complex coordinate.

After creating this grid-array, we use `void displayGrid(CxComplex[][] grid)` to plot lines connecting the rows and columns. You can see the results in figure 3.6.

Looking at the results that this grid yields, we notice that the way to obtain it is not very elegant and that the surface also does not look as symmetric as we expected. We suspect the problem to be in the normalization of the points. But instead of solving this issue, we decide to implement a second, more elegant way to obtain the grid. We call it circular grid. Instead of aiming for equidistance, we now aim for better visualization of the flowing grid. Since the fibres closer to the origin seem to flow away faster than the fibres closer to the boundary of the disc-like 1-section when we apply the Hopf flow in 3.4, it makes sense to have a higher

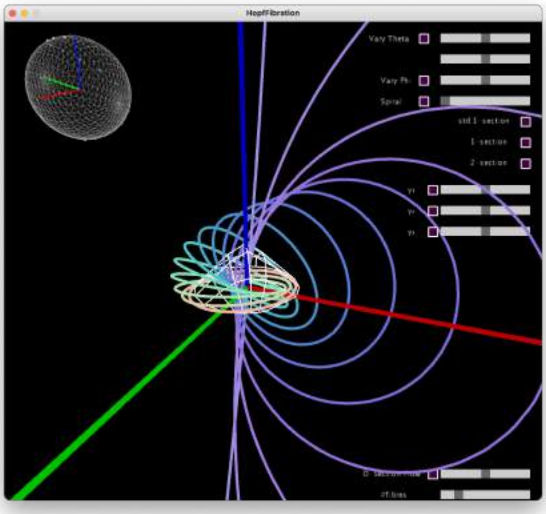


Figure 3.6: Visualizing the d-Section using a 6x6 grid

density of points closer to the origin. This is taken into account in the second way of obtaining a grid, implemented in `getDSectionGridCircular()`. Here we vary r and θ to obtain points on concentric circles. You can see a sketch of the process in figure 3.7.

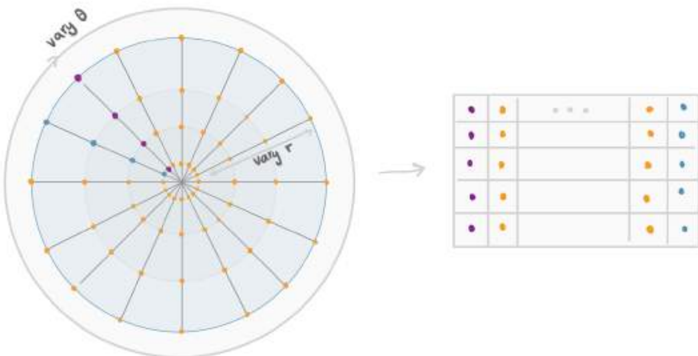


Figure 3.7: Setting up the circular grid, on the right you can see how the points are stored in the array

The results that this grid yields are more aesthetic, as you can see in 3.8.

Additionaly, we include a function called `void displayGrids(CxComplex[][] V_1, CxComplex[][] V_2)` which visualizes the intuition in 3.1.1 of glueing two surfaces together to obtain a d-section, colouring the two parts in different colours. You can see the results in figure 3.9.

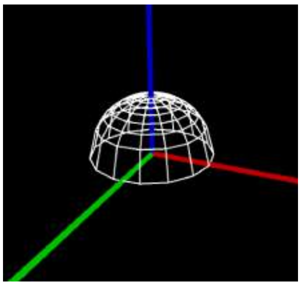


Figure 3.8: Visualizing Σ_N using a circular grid

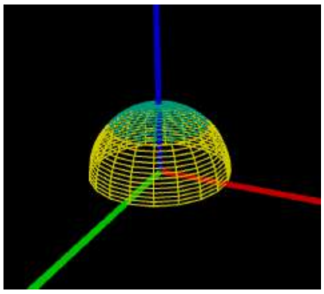


Figure 3.9: Visualizing Σ_N as a glueing of two surfaces

3.2 An Annular 2-Section

Now we construct a 2-section for the Hopf flow.

LEMMA 14. We can obtain a 2-section A for the Hopf fibration by glueing two helicoidal annuli $A_1 \subset V_1$ and $A_2 \subset V_2$ together.

Proof. Let us define the helicoidal annulus A_1 in V_1 with boundary $\partial A_1 = C_1 \cup -(h - 2\mu_1)$ with $Int(A_1)$. Intuitively, this helicoidal annulus is a surface connecting two Hopf-linked circles. To show that A_1 intersects each Hopf fibre in V_1 exactly twice, we use a similar approach as in Lemma 13. As you can see in figure 3.10, we again cut the torus open. As we can see, the Hopf fibre h and the boundary

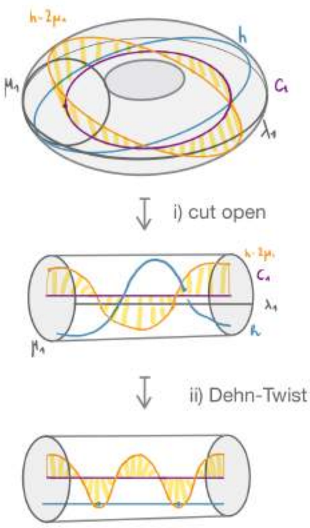


Figure 3.10: heliocoidal annulus A_1 in V_1

$-(h - 2\mu_1)$ are both twisted around the torus, but in different directions. When we now apply the Dehn-twist, we obtain straightened Hopf fibres and the orange boundary is twisted around the torus twice. This shows clearly that this surface intersects the Hopf fibres in V_1 exactly twice.
Analogously, we have a helicoidal annulus A_2 in V_2 with $\partial A_2 = C_2 \cup -(h - \mu_2)$. For

this annulus, we can go through the same procedure to prove that it intersects the Hopf fibres in V_2 exactly twice. And since

$$h - 2\mu_1 = \lambda_1 - \mu_1 = -(\lambda_2 - \mu_2) = -(h - 2\mu_2)$$

we can glue A_1 to A_2 at $-(h - 2\mu_1) = h - 2\mu_2$. The boundary of A is then given by $\partial A = C_1 \cup C_2$, which is clearly a union of orbits, proving (i) in definition 8. And since the two parts intersect each Hopf fibre in their solid torus exactly twice, the union will intersect all Hopf fibres exactly twice, satisfying (ii) and (iii). Thus we can conclude that A forms a 2-section for the Hopf flow. □

3.2.1 Implementing the Annular 2-Section

In the visualization of this 2-section, we colour the two parts A_1 and A_2 differently, to get a better intuition about the two tori. The functions `CxComplex[][] getV_1part_2Section(float varyR, float varyTheta)` and `CxComplex[][] getV_2part_2Section(float varyR, float varyTheta)` obtain the two parts of the grid. For this purpose, we vary θ and r in these expressions:

$$A_1 = (\frac{\sqrt{2}}{2} \cdot re^{-i\theta}, \sqrt{1 - \frac{1}{2}r^2} \cdot e^{i\theta})$$

$$A_2 = (\sqrt{1 - \frac{1}{2}r^2} \cdot e^{i\theta}, \frac{\sqrt{2}}{2} \cdot re^{-i\theta})$$

The grids we obtain here are visualized using the function `void displayGrids()` that plots the grid with distinct colours for the two parts. The standard colour scheme is yellow for A_1 and green for A_2, as you can see in figure 3.11.

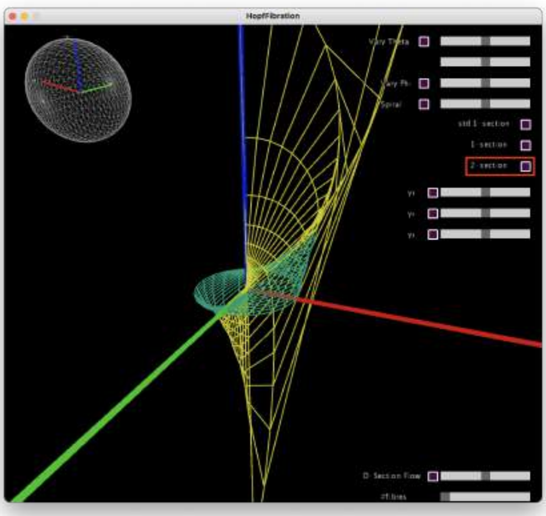


Figure 3.11: Implementation of the 2-section in two different colours

3.3 Rotating d-Sections

To study the d-sections of the Hopf fibration, we implemented two d-sections. Now we will use Matrices $M \in SU(2) \subset SO(4)$ to rotate them around the 3-sphere to create numerous d-sections. We use the matrix group SU(2) since it is compatible with the complex structure of our $\mathbb{C}^2$ space. All the necessary functions can be found in Appendix E.

LEMMA 15. For a d-section Σ of the Hopf flow, the surface $\Sigma_{rot} = A \cdot \Sigma$ with $A \in SU(2)$ is also a d-section for the Hopf flow.

Proof. We define the Hopf flow $\Phi_R^t(p)$ as in lemma 5 and the matrix $A \in SU(2)$. First, we note that any matrix $A \in SU(2)$ has the following property:
A is commutative with the complex multiplication and scalar multiplication, thus

$$A \cdot e^{it} = A \cdot (cos(t) + isin(t)) = (cos(t) + isin(t)) \cdot A = e^{it} \cdot A \quad \text{I)}$$

Now, we prove the first statement of the definition. Let us write down the orbit like this: For $x \in S^3$ the orbit is given as $\Phi_R^t(x) = e^{it} \cdot x$ for $t \in [0, 2\pi]$. The boundary of Σ_{rot} using $y \in \partial\Sigma$ is thus given by:

$$A\Phi_R^t(y) = A \cdot e^{it} \cdot y \overset{\text{I)}}{=} e^{it} \cdot A \cdot y = e^{it} \cdot (Ay) \text{ with } Ay \in S^3$$

Thus, the boundary of Σ_{rot} is given by an orbit of the Hopf flow. Now, we want to look at properties (ii) and (iii). For the d-section Σ these properties hold. For the rotated surface $A \cdot \Sigma = \Sigma_{rot}$, the properties (ii) and (iii) hold for the rotated flow $A \cdot \Phi_R^t$. But again, since

$$A \cdot \Phi_R^t = A \cdot e^{it} \overset{\text{I)}}{=} e^{it} \cdot A = \Phi_R^t \cdot A$$

the rotated flow is the Hopf flow. Thus, we can conclude that Σ_{rot} is indeed a d-section for the Hopf flow. □

To implement this way of obtaining new d-sections, we first need to choose various matrices $M_i \in SU(2)$ that are of the following form:

$$SU(2) = \{\begin{pmatrix} \alpha & -\overline{\beta} \\ \beta & \overline{\alpha} \end{pmatrix} | \alpha, \beta \in \mathbb{C}, |\alpha|^2 + |\beta|^2 = 1\}$$

This complex matrix can be converted to a real matrix as follows. Given a complex 2-by-2 matrix

$$\begin{pmatrix} a_{11} + ib_{11} & a_{12} + ib_{12} \\ a_{21} + ib_{21} & a_{22} + ib_{22} \end{pmatrix}$$

we derive the real matrix

$$\begin{pmatrix} a_{11} & -b_{11} & a_{12} & -b_{12} \\ b_{11} & a_{11} & b_{12} & a_{12} \\ a_{21} & -b_{21} & a_{22} & -b_{22} \\ b_{21} & a_{21} & b_{22} & a_{22} \end{pmatrix}$$

We include the functions `PMatrix3D giveRotationMatrix1(float angle_1)`, `PMatrix3D giveRotationMatrix2(float angle_2)` and

`PMatrix3D giveRotationMatrix3(float angle_3)`. Each function returns a rotation matrix M_i that is obtained in the following ways:
Correponding to $M_1 : \alpha = e^{i\gamma_1}, \beta = 0$
Correponding to $M_2 : \alpha = \beta = \frac{1}{\sqrt{2}} e^{i\gamma_2}$
Correponding to $M_3 : \alpha = isin(\gamma_3), \beta = cos(\gamma_3)$
Yielding the real matrices

$$M_1 = \begin{pmatrix} cos(\gamma_1) & -sin(\gamma_1) & 0 & 0 \\ sin(\gamma_1) & cos(\gamma_1) & 0 & 0 \\ 0 & 0 & cos(\gamma_1) & sin(\gamma_1) \\ 0 & 0 & -sin(\gamma_1) & cos(\gamma_1) \end{pmatrix}$$

$$M_2 = \frac{1}{\sqrt{2}} \cdot \begin{pmatrix} 0 & 0 & -cos(\gamma_2) & -sin(\gamma_2) \\ 0 & 0 & sin(\gamma_2) & -cos(\gamma_2) \\ cos(\gamma_2) & -sin(\gamma_2) & 0 & 0 \\ sin(\gamma_2) & cos(\gamma_2) & 0 & 0 \end{pmatrix}$$

$$M_3 = \begin{pmatrix} 0 & -sin(\gamma_3) & -cos(\gamma_3) & 0 \\ sin(\gamma_3) & 0 & 0 & -cos(\gamma_3) \\ cos(\gamma_3) & 0 & 0 & sin(\gamma_3) \\ 0 & cos(\gamma_3) & -sin(\gamma_3) & 0 \end{pmatrix}$$

Applying these matrices, we can rotate our d-section grid using the function `CxComplex[][] rotateDSection(CxComplex[][] grid, float angle, int kindOfRotation)`. The kind of rotation $i \in \{1, 2, 3\}$ determines which M_i will be used as a rotation matrix. The function returns a new grid by multiplying it with a rotation matrix. With these functions, we can now rotate our d-sections on the sphere to other d-sections we want to investigate. We also include a slightly different colour scheme which is used to visualize the rotated grid of the two-coloured d-sections. In this scheme, the yellow part is indicated in orange and the green part in blue (figure 3.12).

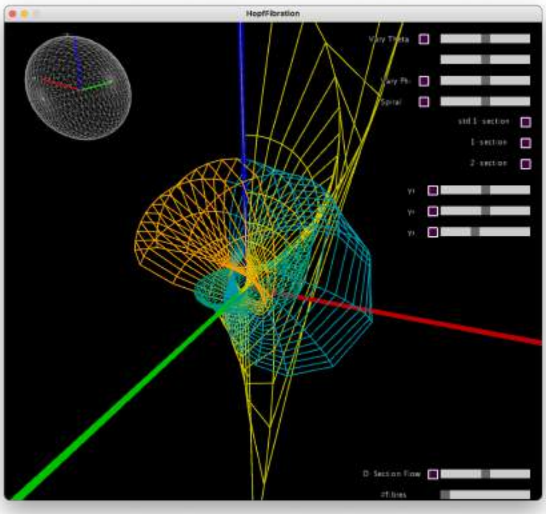


Figure 3.12: The rotated 2-section using M_3

3.4 Letting D-Sections Flow

LEMMA 16. Using a d-section Σ of a non-singular vector field X, we can obtain a second d-section Σ_{flow} for X by letting it flow with the flow of the vector field.

Proof. Firstly, if we let the surface go with the flow of the vector field, it is still compact. Furthermore, the boundary of Σ - being a union of orbits - will stay the same, since the flow moves along the orbits. Moreover, the interior will still be transverse to X, since it gets moved along the orbits of the points. Also, the orbits will intersect $Int(\Sigma_{flow})$ in exactly d points, since the d points in which $Int(\Sigma)$ intersects all orbits simply move along them. Thus we can conclude that Σ_{flow} is a d-section for the vector field X. □

3.4.1 The Flowing Disc-like North Pole 1-Section

We focus on this specific d-section to investigate in formulas how it flows with the Hopf flow. The 1-section is given by:

$$\Sigma_N = \{(re^{i\theta}, \sqrt{1-r^2}) : 0 \leq r \leq 1, \theta \in \mathbb{R}/2\pi\mathbb{Z}\}$$

And the Hopf flow is defined as:

$$\Phi_R^t(p) = e^{it} \cdot p$$

Now we take a point p in the interior of the 1-section: $p = (r \cdot e^{i\theta}, \sqrt{1-r^2})$ with fixed $\theta \in [0, 2\pi)$ and $0 \leq r < 1$.

$$\Phi_R^t(p) = e^{it} \cdot (r \cdot e^{i\theta}, \sqrt{1-r}) = (r \cdot e^{i\theta}e^{it}, \sqrt{1-r}e^{it})$$

When we project $\Phi_R^t(p)$ to $\mathbb{R}^3$ using the stereographic projection σ we introduced before, we find:

$$\sigma(\Phi_R^t(p)) = \sigma(r \cdot e^{i\theta}e^{it}, \sqrt{1-r}e^{it})$$

$$= \frac{1}{1-\sqrt{1-r}sin(t)}(rcos(\theta)cos(t), rsin(\theta)sin(t), \sqrt{1-r}cos(t))$$

Since all orbits are circles, we should flow back to the original Σ_N when we go with the flow for $t = 2\pi$. Interestingly, the point $\hat{N} = (0, 0, 1, 0) \in \Sigma_N$ lies on C_1, which maps onto the z-axis. That means, to flow once around its fibre, this point has to go through the north pole of σ. As a result, the projected point has to pass ∞.

$$\Phi_R^t(\hat{N}) = e^{it} \cdot (0, 1) = (0, e^{it})$$

Applying the stereographic projection yields:

$$\sigma(\Phi_R^t(\hat{N})) = \sigma((0, e^{it})) = (0, 0, \frac{cos(t)}{1 - sin(t)})$$

implying:

$$\sigma(\Phi_R^0(\hat{N})) = (0, 0, 1)$$

$$lim_{t \nearrow \frac{\pi}{2}} \sigma(\Phi_R^t(\hat{N})) = lim_{t \nearrow \frac{\pi}{2}}(0, 0, \frac{cos(t)}{1 - sin(t)}) \longrightarrow +\infty$$

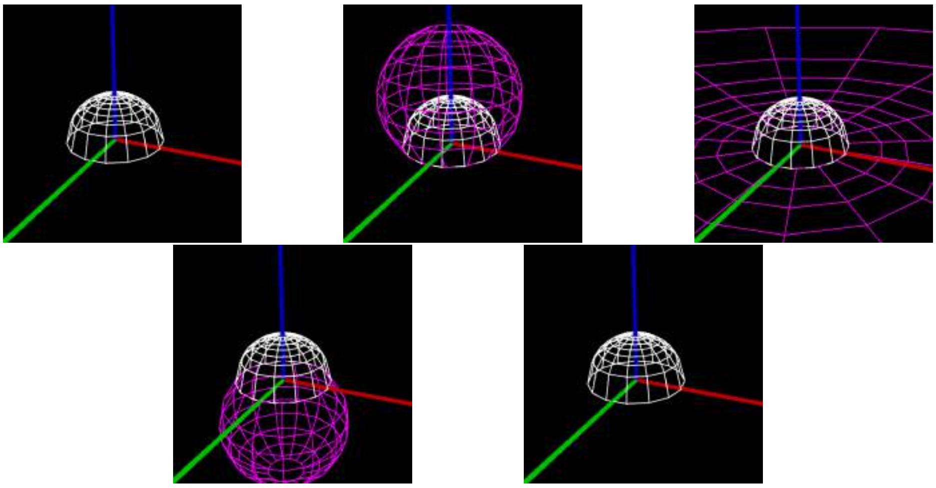


Figure 3.13: The pink grid visualizes Σ_N (indicated in white) flowing with the Hopf flow Φ_R^t for $t = 0$, $t = \frac{\pi}{4}$, $t = \frac{\pi}{2}$ $t = \frac{3\pi}{4}$ and $t = 2\pi$

$$lim_{t \searrow \frac{\pi}{2}} \sigma(\Phi_R^t(\hat{N})) = lim_{t \nearrow \frac{\pi}{2}} (0, 0, \frac{cos(t)}{1 - sin(t)}) \longrightarrow -\infty$$

$$\sigma(\Phi_R^{\pi}(\hat{N})) = (0, 0, -1)$$

$$\sigma(\Phi_R^{2\pi}(\hat{N})) = (0, 0, 1)$$

The flip from $+\infty$ to $-\infty$ is nicely visible in our implementation, as you can see in figure 3.13.

3.4.2 Letting the Grid Flow

To implement this feature, we include a scrollbar to enable the user to change t in real-time. Using this value, the function `CxComplex[][] letGridFlow(CxComplex[][] grid, float t)` goes through the grid and lets each point go with the Hopf flow.

We also implement a second way of visualizing the flow by drawing lines along the flow of the grid. Therefore, we can not only visualize the new position but also the movement of the grid. To achieve this, we implement so-called tubes provided by the library Shapes3D [4] for each point on the grid.

Since this procedure is computationally costly, we first compute the tubes in the `setup()` function. This function runs before the application starts. Using the function `Tube[][] setupTubes (CxComplex[][] grid, float t)`, we set up tubes of various lengths, that we only have to refer to later. When the application is running, we use `drawTubeCoord()` to display the tubes. These functions are not included in the final state of the code, because of their computational effort and the cost of crowding most of the screen. But if the user wants to enable them, it is possible to un-comment the needed lines (Appendix A, lines 91, 160, 202).

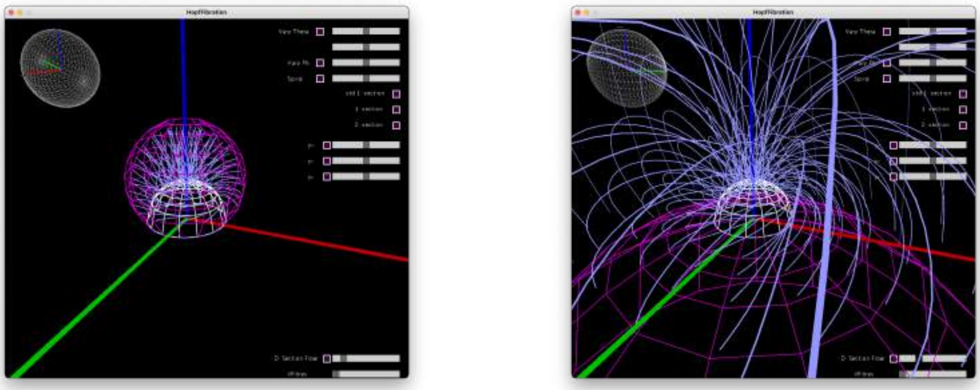


Figure 3.14: Tubes

Chapter 4

The Graphical User Interface

For a more intuitive and real-time changeable exploration of the Hopf fibration, we include a graphical user interface in the programme. All related functions are included in Appendix F. The GUI consists of an overlay that visualizes the points on the Riemann sphere and numerous scrollbars and buttons to change the way we obtain the points on the sphere and to enable the visualization of d-sections. We also enable the user to explore different d-sections using rotational matrices and the Hopf flow.

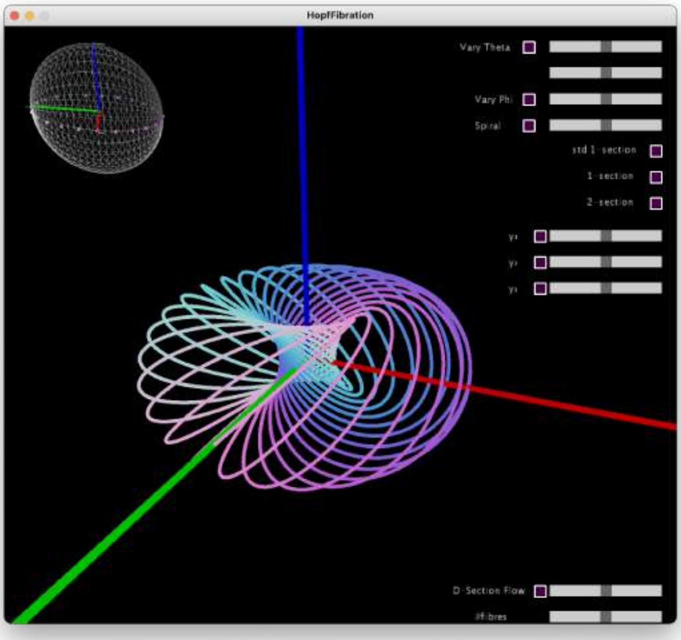


Figure 4.1: Overview of the GUI

4.1 The Riemann Sphere

The sphere in the top left corner visualizes the points on the Riemann Sphere that we choose. We colour them like the projected fibres, to make the connection visible.

Clicking on the sphere, it will start or stop rotating, so we can always get a good perspective at the points.

4.2 Scrollbars and Buttons

We use the class HScrollbars [5] (Appendix G) to set up the scrollbars for varying the different variables.
We go through the scrollbars and buttons from top to bottom. In the top right corner, we find the scrollbars to change the way the points on the Riemann Sphere are obtained. Here, the functions `VaryTheta()`, `VaryPhi()` and `Spiral()` are covered, as discussed in 2.1. With a click on the button, we enable the function, and with the scrollbars we can change the input angle. In the special case of `VaryTheta()`, there are two scrollbars, which means we can give two different input angles if we wish to. If we want to choose only one angle, we can click on one scrollbar and then hover over the second while we move our mouse to change the value. The second scrollbar will then follow the movement as well. Below that, there are three buttons to enable the visualization of d-sections. The first one is the standard Σ_N, the second one is Σ_N coloured in two different colours, and the third one is the annular 2-section. The three scrollbars tagged with γ_i, $i \in 1, 2, 3$ rotate the d-sections with the rotational matrices M_i, as discussed in 3.3. To enable a rotation, we click on the button on the left side of the scrollbar. Then we are able to change the input angle. On the bottom of the window, there are two more scrollbars. The first one is to let the grid flow with the Hopf flow. Again, a click on the button enables it. This feature can also be used additionally to a rotation. That means we can first rotate the d-section and then let this new d-section flow with the Hopf flow. The last scrollbar is used for changing the number of circles that are plotted. To reset the rotations or flow, we can click on the d-section button again.

Camera Modes

Using the 'UP' button on the keyboard, we can switch between two camera modes: The first one is static, and the second one is rotating your view with the horizontal movement of your mouse.

Recording the Screen

It is possible to record a video of your exploration, therefore it is only necessary to un-comment lines 2,3 and 268 in the main code (Hopffibration.pde). Then a video of the window will be recorded from the start of the application until you stop it using the key 'Q' or Processings stop-button. This file saves to where your 'HopfFibration'-folder is located.

Conclusion

For exploring the Hopf Fibration on your device, you can download Processing at *https://processing.org/download*. Next, add the attached folder 'HopfFibration' to your processing directory and 'QScript', 'Shapes3D', 'video' and 'VideoExport' to the library folder. Alternatively, you can also download the code from https://github.com/JeBentMooi/HopfFibration.git and import the libraries manually. Using the Processing application, you can open the project 'HopfFibration' and press the play button. Unfortunately, it is not possible to export the code as a standalone application, due to problems with the 3D renderer.

To conclude this thesis, we have gained a basic understanding of the mathematical concepts around the Hopf Fibration. Furthermore, we developed code to visualise the implications of its mathematical properties. Using the program, it is now possible to explore the Hopf Fibration and numerous of its d-sections to develop intuitions about its behaviour. This tool will aid future inquiries into the properties of the Hopf Fibration. Such as the study of more complex d-sections, like the pair of pants 1-section, as investigated in [1]. Also, using the method of the annular 2-section, it is possible to construct annular d-sections for any $d \in \mathbb{N}$. We look forward to any creative application of the program in the future, both in educational as well as research contexts.

Appendix A

Appendix: Code

The code can also be found at: https://github.com/JeBentMooi/HopfFibration.git

```
//library for recording the screen
import processing.video.*;
import com.hamoid.*;

//library for extrusions: Shapes 3D
import shapes3d.*;
import shapes3d.contour.*;
import shapes3d.org.apache.commons.math.*;
import shapes3d.org.apache.commons.math.geometry.*;
import shapes3d.path.*;
import shapes3d.utils.*;

//library for complex numbers: QScript
import org.qscript.*;
import org.qscript.editor.*;
import org.qscript.errors.*;
import org.qscript.events.*;
import org.qscript.eventsonfire.*;
import org.qscript.operator.*;




// ---------------------------------------------------------------------------------
// ---------------------------------------------------------------------------------

  Complex i = new Complex(0,1); //i

  //SETUP HOPF FIBRES
  int noPoints = 150; //how many points will be used to draw 1 circle; must be >=3
  int noCircles = 30; //how many circles do you want to plot? - can be changed with scrollbar
  Vector[][] circlePoints;
  Complex[] startingPoints;

  //SETUP SCROLLBARS & BUTTONS
```

```
HScrollbar s_VaryPhi, s_VaryTheta, s_VaryTheta2, s_Spiral;
HScrollbar s_gamma_1, s_gamma_2, s_gamma_3, s_gamma_4;
HScrollbar s_Flow, s_noCircles;
boolean VaryThetaMode = true;
boolean VaryPhiMode = false;
boolean SpiralMode = false;
boolean RotationMode = true;
boolean zoomMode = false;
float rot = 0;
int camMode = 0;
boolean Mode1sectStd = false;
boolean Mode1sect = false;
boolean Mode2sect = false;

//SETUP D SECTION GRIDS
int varyR = 8;
int varyTheta = 15;
CxComplex[][] circularGrid = new CxComplex[varyR][varyTheta];
CxComplex[][] circularGrid_flow = new CxComplex[varyR][varyTheta
  ];
CxComplex[][] circularGrid_rot = new CxComplex[varyR][varyTheta];
PVector[][] tubes;

//2.2.1
CxComplex[][] V_1grid_1 = getV_1part(13,20);
CxComplex[][] V_2grid_1 = getV_2part(13,20);
CxComplex[][] V_1grid_1_rot = getV_1part(13,20);
CxComplex[][] V_2grid_1_rot = getV_2part(13,20);
CxComplex[][] V_1grid_1_flow = getV_1part(13,20);
CxComplex[][] V_2grid_1_flow = getV_2part(13,20);


//2.2.2
CxComplex[][] V_1grid_2 = getV_1part_2Section(13,20);
CxComplex[][] V_2grid_2 = getV_2part_2Section(13,20);
CxComplex[][] V_1grid_2_rot = getV_1part_2Section(13,20);
CxComplex[][] V_2grid_2_rot = getV_2part_2Section(13,20);
CxComplex[][] V_1grid_2_flow = getV_1part_2Section(13,20);
CxComplex[][] V_2grid_2_flow = getV_2part_2Section(13,20);

//rotation modes & flow mode
boolean rotMode1 = false;
boolean rotMode2 = false;
boolean rotMode3 = false;
boolean flowMode = false;

void setup() {
  size(900, 800, P3D);
  frameRate(10); //fix-bug-thing, do not delete
  setupScrollbars();
  //grid
  circularGrid = getDSectionGridCircular(varyR, varyTheta);
  circularGrid_flow = getDSectionGridCircular(varyR, varyTheta);
  circularGrid_rot = getDSectionGridCircular(varyR, varyTheta);

  //SETUP TUBES
  //tubes = setupTubes(circularGrid, 2*PI);
}
```

```


void draw(){
  background(0);

  //CHOOSE CAMERAMODE
  if(camMode%3 == 1){
    camera(mouseX*2, height/2, (height/2) / tan(PI/6), width/2, height/2, 0, 0, 1, 0); //camera which rotates objects with MouseX
  } else if(camMode%3 == 0){
    camera(width/2, height/2, (height/2) / tan(PI/6), width/2, height/2, 0, 0, 1, 0); //centered camera
  }

  //UPDATE NoCircles
  noCircles = 2*(int)scrollbarValue(s_noCircles, 50); //always even number
  //get everything in order to plot
  circlePoints = new Vector[noCircles][noPoints];
  //setup array of points of interest in C, whose fibres we want to find
  startingPoints = new Complex[noCircles];


  //SETUP STARTING POINTS
  if (VaryThetaMode == true){
  addPointsVaryTheta(noCircles/2, scrollbarValue(s_VaryTheta, PI));
  addPointsVaryTheta(noCircles/2,noCircles/2, scrollbarValue(s_VaryTheta2, PI));
  } else if(VaryPhiMode == true){
  addPointsVaryPhi(noCircles, scrollbarValue(s_VaryPhi, PI));
  } else if (SpiralMode == true){
  addPointsSpiral(noCircles);
  }

  //SETUP COORDINATE SYSTEM
  centerCoordinatesystem();
  drawAxes(300);
  nameAxes(300);

  //DRAW D SECTION
  if (Mode1sectStd == true){ //Std grids
    if(rotMode1 == true){
      circularGrid_rot = rotateDSection(circularGrid, scrollbarValue(s_gamma_1,2*PI),1);
      displayGrid(circularGrid_rot, true, 250,250,0);
      displayGrid(circularGrid, true);
      if (flowMode == true){
        circularGrid_flow = letGridFlow(circularGrid_rot, scrollbarValue(s_Flow,2*PI));
        displayGrid(circularGrid_flow, true, 200,0,200);
      }
    } else if (rotMode2 == true){
      circularGrid_rot = rotateDSection(circularGrid, scrollbarValue(s_gamma_2,2*PI),2);
      displayGrid(circularGrid_rot, true, 250,250,0);
      displayGrid(circularGrid, true);
```

```
      if (flowMode == true){
        circularGrid_flow = letGridFlow(circularGrid_rot, scrollbarValue(s_Flow,2*PI));
        displayGrid(circularGrid_flow, true, 200,0,200);
      }
    } else if (rotMode3 == true){
      circularGrid_rot = rotateDSection(circularGrid, scrollbarValue(s_gamma_3,2*PI),3);
      displayGrid(circularGrid_rot, true, 250,250,0);
      displayGrid(circularGrid, true);
      if (flowMode == true){
        circularGrid_flow = letGridFlow(circularGrid_rot, scrollbarValue(s_Flow,2*PI));
        displayGrid(circularGrid_flow, true, 200,0,200);
      }
    } else { //no rotation mode == true
      displayGrid(circularGrid, true);
      if (flowMode == true){
        circularGrid_flow = letGridFlow(circularGrid, scrollbarValue(s_Flow,2*PI));
        displayGrid(circularGrid_flow, true, 200,0,200);
        //TUBES
        //drawTubeCoord(tubes, scrollbarValue(s_Flow, 50));
      }
    }

  } else if (Mode1sect == true){ //2.2.1 - glue meridonal disc to annulus
    if(rotMode1 == true){
      V_1grid_1_rot = rotateDSection(V_1grid_1, scrollbarValue(s_gamma_1,2*PI),1);
      V_2grid_1_rot = rotateDSection(V_2grid_1, scrollbarValue(s_gamma_1,2*PI),1);
      displayGridsColoured(V_1grid_1_rot, V_2grid_1_rot);
      displayGrids(V_1grid_1, V_2grid_1);
      if (flowMode == true){
        V_1grid_1_flow = letGridFlow(V_1grid_1_rot, scrollbarValue(s_Flow,2*PI));
        V_2grid_1_flow = letGridFlow(V_2grid_1_rot, scrollbarValue(s_Flow,2*PI));
        displayGridsColoured(V_1grid_1_flow,V_2grid_1_flow);
      }
    } else if(rotMode2 == true){
      V_1grid_1_rot = rotateDSection(V_1grid_1, scrollbarValue(s_gamma_2,2*PI),2);
      V_2grid_1_rot = rotateDSection(V_2grid_1, scrollbarValue(s_gamma_2,2*PI),2);
      displayGridsColoured(V_1grid_1_rot, V_2grid_1_rot);
      displayGrids(V_1grid_1, V_2grid_1);
      if (flowMode == true){
        V_1grid_1_flow = letGridFlow(V_1grid_1_rot, scrollbarValue(s_Flow,2*PI));
        V_2grid_1_flow = letGridFlow(V_2grid_1_rot, scrollbarValue(s_Flow,2*PI));
        displayGridsColoured(V_1grid_1_flow,V_2grid_1_flow);
      }
    } else if(rotMode3 == true){
```

```
      V_1grid_1_rot = rotateDSection(V_1grid_1, scrollbarValue(s_gamma_3,2*PI),3);
      V_2grid_1_rot = rotateDSection(V_2grid_1, scrollbarValue(s_gamma_3,2*PI), 3);
      displayGridsColoured(V_1grid_1_rot, V_2grid_1_rot);
      displayGrids(V_1grid_1, V_2grid_1);
      if (flowMode == true){
        V_1grid_1_flow = letGridFlow(V_1grid_1_rot, scrollbarValue(s_Flow,2*PI));
        V_2grid_1_flow = letGridFlow(V_2grid_1_rot, scrollbarValue(s_Flow,2*PI));
        displayGridsColoured(V_1grid_1_flow,V_2grid_1_flow);
      }
    } else {
      displayGrids(V_1grid_1, V_2grid_1);
      if (flowMode == true){
        V_1grid_1_flow = letGridFlow(V_1grid_1, scrollbarValue(s_Flow,2*PI));
        V_2grid_1_flow = letGridFlow(V_2grid_1, scrollbarValue(s_Flow,2*PI));
        displayGridsColoured(V_1grid_1_flow,V_2grid_1_flow);
      //TUBES
      //drawTubeCoord(tubes, scrollbarValue(s_Flow, 50));
      }
    }

  } else if(Mode2sect == true){ //2.2.2 - glue two annuli
    if(rotMode1 == true){
      V_1grid_2_rot = rotateDSection(V_1grid_2, scrollbarValue(s_gamma_1,2*PI),1);
      V_2grid_2_rot = rotateDSection(V_2grid_2, scrollbarValue(s_gamma_1,2*PI),1);
      displayGridsColoured(V_1grid_2_rot, V_2grid_2_rot);
      displayGrids(V_1grid_2, V_2grid_2);
      if(flowMode == true){
        V_1grid_2_flow = letGridFlow(V_1grid_2_rot, scrollbarValue(s_Flow,2*PI));
        V_2grid_2_flow = letGridFlow(V_2grid_2_rot, scrollbarValue(s_Flow,2*PI));
        displayGridsColoured(V_1grid_2_flow,V_2grid_2_flow);
      }
    } else if(rotMode2 == true){
      V_1grid_2_rot = rotateDSection(V_1grid_2, scrollbarValue(s_gamma_2,2*PI),2);
      V_2grid_2_rot = rotateDSection(V_2grid_2, scrollbarValue(s_gamma_2,2*PI),2);
      displayGridsColoured(V_1grid_2_rot, V_2grid_2_rot);
      displayGrids(V_1grid_2, V_2grid_2);
      if(flowMode == true){
        V_1grid_2_flow = letGridFlow(V_1grid_2_rot, scrollbarValue(s_Flow,2*PI));
        V_2grid_2_flow = letGridFlow(V_2grid_2_rot, scrollbarValue(s_Flow,2*PI));
        displayGridsColoured(V_1grid_2_flow,V_2grid_2_flow);
      }
    } else if(rotMode3 == true){
      V_1grid_2_rot = rotateDSection(V_1grid_2, scrollbarValue(s_gamma_3,2*PI),3);
```

```
      V_2grid_2_rot = rotateDSection(V_2grid_2, scrollbarValue(s_gamma_3,2*PI),3);
      displayGridsColoured(V_1grid_2_rot, V_2grid_2_rot);
      displayGrids(V_1grid_2, V_2grid_2);
      if(flowMode == true){
        V_1grid_2_flow = letGridFlow(V_1grid_2_rot, scrollbarValue(s_Flow,2*PI));
        V_2grid_2_flow = letGridFlow(V_2grid_2_rot, scrollbarValue(s_Flow,2*PI));
        displayGridsColoured(V_1grid_2_flow,V_2grid_2_flow);
      }
    } else {
      displayGrids(V_1grid_2, V_2grid_2);
      if(flowMode == true){
        V_1grid_2_flow = letGridFlow(V_1grid_2, scrollbarValue(s_Flow,2*PI));
        V_2grid_2_flow = letGridFlow(V_2grid_2, scrollbarValue(s_Flow,2*PI));
        displayGridsColoured(V_1grid_2_flow,V_2grid_2_flow);
      }
    }
 } else {  }

  //DRAW FIBRES
  fillArray(); //compute
  drawColCircle(); //draw fibres

  //  - GUI -
  //DRAW SPHERE IN CORNER
  camera(); //back to normal camera settings for the overlay
  centerCoordinatesystemOverlay();
  if(RotationMode == true){
    rot = getRotation(rot);
  }
  rotateSphere(rot);
  drawAxes(80,3);
  drawSphere();
  drawPointsOnSphere(startingPoints);

  //DRAW SLIDERS
  camera(); //back to normal camera settings for second overlay
  updateScrollbars();
  displayScrollbars();
  drawButtons();

  //enable to record screen:
  rec();
}

void mousePressed() {
  if (overVaryThetaButton()==true) {
    VaryThetaMode = true;
    VaryPhiMode = false;
    SpiralMode = false;
  }
  if (overVaryPhiButton()==true) {
    VaryThetaMode = false;
    VaryPhiMode = true;
```

```
    SpiralMode = false;
  }
  if(overSpiralButton() ==true){
    VaryThetaMode = false;
    VaryPhiMode = false;
    SpiralMode = true;
  }
  if(overSphere()==true && RotationMode == true){
    RotationMode = false;
  } else if(overSphere()==true && RotationMode == false){
    RotationMode = true;
  }
  if (over1sectionStd()==true) {
    Mode1sectStd = true;
    Mode1sect = false;
    Mode2sect = false;

    rotMode1 = false;
    rotMode2 = false;
    rotMode3 = false;
    flowMode = false;
  }
  if (over1section()==true) {
    Mode1sectStd = false;
    Mode1sect = true;
    Mode2sect = false;

    rotMode1 = false;
    rotMode2 = false;
    rotMode3 = false;
    flowMode = false;
  }
  if (over2section()==true) {
    Mode1sectStd = false;
    Mode1sect = false;
    Mode2sect = true;

    rotMode1 = false;
    rotMode2 = false;
    rotMode3 = false;
    flowMode = false;
  }
  if (overGamma1()==true) {
    rotMode1 = true;
    rotMode2 = false;
    rotMode3 = false;
  }
  if (overGamma2()==true) {
    rotMode1 = false;
    rotMode2 = true;
    rotMode3 = false;
  }
  if (overGamma3()==true) {
    rotMode1 = false;
    rotMode2 = false;
    rotMode3 = true;
  }
  if (overFlow()==true) {
```

```
      flowMode = true;
  }
}

void keyPressed(){
  if (key == CODED) {
    if (keyCode == UP) {
      camMode++;
    }
  } else if (key == 'q') {
    videoExport.endMovie();
    exit();
  }
}
```

Appendix B

Class CxComplex

I coded this class to have an object for the 4-dimensional points in $\mathbb{C} \times \mathbb{C}$

```
   class CxComplex { //Complex x Complex

Complex z_1;
Complex z_2;

//__CONSTRUCTOR__

  CxComplex(){
  this.z_1 = new Complex();
  this.z_1 = new Complex();
  }

  CxComplex(CxComplex z){
    this.z_1 = z.z_1;
    this.z_2 = z.z_2;
  }

  CxComplex(Complex x, Complex y) {
   this.z_1 = x;
   this.z_2 = y;
  }

  CxComplex(double x_1, double y_1, double x_2, double y_2) {
   this.z_1 = new Complex(x_1, y_1);
   this.z_2 = new Complex(x_2, y_2);
  }

  CxComplex(float x_1, float y_1, float x_2, float y_2) {
   this.z_1 = new Complex((double)x_1, (double)y_1);
   this.z_2 = new Complex((double)x_2, (double)y_2);
  }

//__FUNCTIONS__
double x_1(){
return this.z_1.real;
}
double y_1(){
return this.z_1.imag;
}
double x_2(){
return this.z_2.real;
```

```
}
double y_2(){
return this.z_2.imag;
}

CxComplex normalize(){
float len = sqrt(pow((float)this.x_1(),2)+pow((float)this.y_1(),2)+pow((float)this.x_2(),2)+pow((float)this.y_2(),2));
float x_1 = (float)this.x_1() / len;
float y_1 = (float)this.y_1() / len;
float x_2 = (float)this.x_2() / len;
float y_2= (float)this.y_2() / len;
return new CxComplex(x_1,y_1,x_2,y_2);
}

CxComplex goWithFlow(float t){ //goes with the Hopf flow
  Complex new_z_1 = this.z_1.mult(Complex.exp(i.mult(t)));
  Complex new_z_2 = this.z_2.mult(Complex.exp(i.mult(t)));
  return new CxComplex(new_z_1,new_z_2);
}

 CxComplex normalize2ndCoordinate(){
   float len = sqrt(pow((float)this.x_1(),2)+pow((float)this.y_1(),2)+pow((float)this.x_2(),2)+pow((float)this.y_2(),2)); //length of vector
   float x_1 = (float)this.x_1();
   float y_1 = (float)this.y_1();
   float x_2 = 1-len;
   float y_2= (float)this.y_2();
   return new CxComplex(x_1,y_1,x_2,y_2);
 }

 CxComplex applyRotMatrix(PMatrix3D rot){
   float a = rot.multX((float)this.z_1.real, (float)this.z_1.imag, (float)this.z_2.real, (float)this.z_2.imag);
   float b =rot.multY((float)this.z_1.real, (float)this.z_1.imag, (float)this.z_2.real, (float)this.z_2.imag);
   float c =rot.multZ((float)this.z_1.real, (float)this.z_1.imag, (float)this.z_2.real, (float)this.z_2.imag);
   float d =rot.multW((float)this.z_1.real, (float)this.z_1.imag, (float)this.z_2.real, (float)this.z_2.imag);

   return new CxComplex(a,b,c,d);
 }

}
//__more functions__

CxComplex subtract(CxComplex x, CxComplex y){
  Complex a = x.z_1.sub(y.z_1);
  Complex b = x.z_2.sub(y.z_2);
   return new CxComplex(a,b);
 }

CxComplex mult(double scalar, CxComplex z){
  Complex a = z.z_1;
  Complex b = z.z_2;
  a = new Complex(a.real*scalar, a.imag*scalar);
```

```
  b = new Complex(b.real*scalar, b.imag*scalar);
return new CxComplex(a,b);
}

CxComplex add(CxComplex x, CxComplex y){
  Complex a = x.z_1;
  Complex b = x.z_2;
  Complex c = y.z_1;
  Complex d = y.z_2;

  a = a.add(c);
  b = b.add(d);
  return new CxComplex(a,b);
}
```

Appendix C

Mathematics Library

```
  //__MATHEMATICAL ALGORITHM FUNCTIONS__

void fillArray(){ //gives #noPoints points on the circle in R3, (evenly spaced on the circle in C2)
  CxComplex[] PointsInC2 = new CxComplex[noCircles]; //array of points in C2
  for(int i=0; i<noCircles; i++){
    PointsInC2[i] = findFibrePoint(startingPoints[i]); //fill that array up:
  }
  for (int j=0; j<noCircles; j++){
    circlePoints[j][0]= projectPoint(PointsInC2[j]); //put starting points' projections in first entry
    for(int i=1; i< noPoints; i++){
      circlePoints[j][i] = projectPoint(getNewPoint(PointsInC2[j]));
      PointsInC2[j] = getNewPoint(PointsInC2[j]);
    }
  }
}

CxComplex findFibrePoint(Complex p){ //put in complex number p and get one point on circle in S^3
//  (x_1,y_2,x_2,y_2) is coordinate in C^2
//  trick: set y_2 =0.
float x_1;
float x_2;
float y_1;
x_2 = sqrt(1/(1+ pow((float)p.real,2)+ pow((float)p.imag,2)));
x_1 = (float)p.real*x_2;
y_1 = (float)p.imag*x_2;
CxComplex p_Fibre = new CxComplex(x_1,y_1,x_2,0);
return p_Fibre;
}

CxComplex getNewPoint (CxComplex p){ //generate new point on same Fibre
  //go fourth of orbit to find next point
  Complex t= new Complex(2*PI/noPoints);
  Complex z_1_second = p.z_1.mult(Complex.exp(i.mult(t)));
  Complex z_2_second = p.z_2.mult(Complex.exp(i.mult(t)));

```

```
  CxComplex p_2 = new CxComplex(z_1_second, z_2_second);
  return p_2;
}

CxComplex getNewPoint (CxComplex p, int noPoints){ //generate new point on same Fibre
  //go fourth of orbit to find next point
  Complex t = new Complex(2*PI/noPoints);
  Complex z_1_second = p.z_1.mult(Complex.exp(i.mult(t)));
  Complex z_2_second = p.z_2.mult(Complex.exp(i.mult(t)));

  CxComplex p_2 = new CxComplex(z_1_second, z_2_second);
  return p_2;
}

Vector goWithFlowAndProject(CxComplex p, float t){
  //generate new point on same Fibre //go fourth of orbit to find next point
  Complex z_1_second = p.z_1.mult(Complex.exp(i.mult(t)));
  Complex z_2_second = p.z_2.mult(Complex.exp(i.mult(t)));
  CxComplex p_2 = new CxComplex(z_1_second, z_2_second);
  Vector v = projectPoint(p_2);
  //println(v.x, v.y, v.z);
  return v;
}

Vector projectPoint(CxComplex p){//put in point in 4D get it projected into R^3 via stereographic projection with N=(0,0,0,1)
                                 //for projecting p, p', M
Vector p_projected = new Vector(p.z_1.real/(1-p.z_2.imag),p.z_1.imag/(1-p.z_2.imag),p.z_2.real/(1-p.z_2.imag));
return p_projected;
}

Complex SphericalToComplex(float theta, float phi){ //converts spherical coordinates to cartesian and projects them down
                                                    //theta is angle from x-axis in x/y plane, phi is from z-axis in z/y plane
  //convert from spherical to cartesian
  double x = sin(phi)*cos(theta);
  double y = sin(phi)*sin(theta);
  double z =cos(phi);
  //stereographicProjection
  double Re = x/(1-z);
  double Im = y/(1-z);
  return new Complex(Re, Im);
}

Vector ComplexToCartesian(Complex cplx){ //converts complex number into cartesian coordinate on the riemann sphere
  //stereographic projection
  float x=2*(float)cplx.real /(1+pow((float)cplx.real,2)+pow((float)cplx.imag,2));
  float y=2*(float)cplx.imag /(1+pow((float)cplx.real,2)+pow((float)cplx.imag,2));
  double z=(-1+pow((float)cplx.real,2)+pow((float)cplx.imag,2))/(1+pow((float)cplx.real,2)+pow((float)cplx.imag,2));
  return new Vector(x,y,z);
```

```
}

float distance(Vector vector_1, Vector vector_2){
  return (float)(Vector.sub(vector_1,vector_2)).mag();
}

void drawColCircle(){ //draw coloured circle
  strokeWeight(0.05);
  noFill();
  curveTightness(0.5);
  for(int j=0; j<noCircles; j++){
    beginShape();
    stroke((float)circlePoints[j][0].x*105+150,(float)circlePoints[j][0].y*85+150,(float)circlePoints[j][0].z*70+160); // colorscheme 01
    for(int i=0; i<noPoints-1; i++){
      if(i>0 && distance(circlePoints[j][i],circlePoints[j][i-1])>noPoints/2+10){//distance too big - end shape and start new shape for rest of circle.
      endShape();
      beginShape();
      } else {
      curveVertex((float)circlePoints[j][i].x,(float)circlePoints[j][i].y,(float)circlePoints[j][i].z);
      }
    }
    curveVertex((float)circlePoints[j][0].x,(float)circlePoints[j][0].y,(float)circlePoints[j][0].z); //1
    curveVertex((float)circlePoints[j][1].x,(float)circlePoints[j][1].y,(float)circlePoints[j][1].z); //2
    curveVertex((float)circlePoints[j][2].x,(float)circlePoints[j][2].y,(float)circlePoints[j][2].z); //3 - need those three for anchor point in beginning and end
    endShape();
  }
}

//__STARTING POINTS ON RIEMANN SPHERE__

void addPointsVaryTheta(int amount, float phi){ //look how theta changes fibres, constant phi
  for(int j=0; j<amount; j++){
    startingPoints[j]=SphericalToComplex(j*2*PI/amount, phi);
  }
}

void addPointsVaryTheta(int array_entry, int amount, float phi){ // gives #amount equidistant points on horizontal circle on S2, phi gives "height" in circle.
                                                             //starts in array_entry-th entry of the array
  for(int j=0; j<amount; j++){
    startingPoints[array_entry+j]=SphericalToComplex(j*2*PI/amount, phi);
  }
}

void addPointsVaryThetaAndPhi(int amount, float alpha){ //gives "
```

```
    vertical" circle
  for(int j=0; j<amount; j++){
     startingPoints[j]=SphericalToComplex(alpha*sin(j*PI/amount), alpha*cos(j*PI/amount));
  }
}

void addFewPointsVaryTheta(int amountHeights, float[] phi){//makes equidistant points on #amountHeights different Heights
  for(int j=0;j<amountHeights; j++){
  addPointsVaryTheta((noCircles/amountHeights)*j, noCircles/amountHeights, phi[j]);
  }
}

void addPointsVaryPhi(int amount, float theta){ //look how phi changes fibres, constant theta
  for(int j=0; j<amount; j++){
     startingPoints[j]=SphericalToComplex(theta, j*(2*PI/amount));
  }
}

void addPointsVaryPhi(int array_entry, int amount, float theta){ //look how phi changes fibres, constant theta
  for(int j=0; j<amount; j++){
     startingPoints[array_entry+j]=SphericalToComplex(theta, j*(2*PI/amount));
  }
}

void addFewPointsVaryPhi(int amountHeights, float[] theta){
  for(int j=0;j<amountHeights; j++){
  addPointsVaryPhi((noCircles/amountHeights)*j, noCircles/amountHeights, theta[j]);
  }
}

void addPointsSpiral(int amount){ //gives #amount points in a spiral around the sphere
  for(int j=0; j<amount; j++){
     startingPoints[j]=SphericalToComplex(j*(2*PI/amount)+scrollbarValue(s_Spiral, 2*PI), j*(PI/amount));
  }
}
```

Appendix D

D-Section Library

```
int dSectionNoPoints = 20;

void SetupDisplaySettingsDSection(){
  strokeWeight(0.02);
  fill(150,200,150,130);
  stroke(255);
}

//__SOUTHERN 1-SECTION__

void drawSouthernDSectionBoundary(){
  SetupDisplaySettingsDSection();
  pushMatrix();
  rotateX(PI/2);
  rotateZ(PI);
  rect(0,-200,400,400);
  popMatrix();
}

//__GRID 1-SECTION__

CxComplex[][] setupDSectionGrid(CxComplex[][] grid, int noCol, int noRow, CxComplex z){
  CxComplex p = new CxComplex(z.normalize());
  grid = getDSectionGrid(noCol, noRow, p);
  return grid;
}

CxComplex[][] getDSectionGrid(int noColumns, int noRows, CxComplex p){
  CxComplex[][] grid= new CxComplex[noColumns][noRows];
  //fill the first & last row of the grid, [0][0]=p and [0][noRows-1]=p_opposite
  for(int i=0; i<noColumns; i++){
   grid[i][0] = p.goWithFlow(3*PI*i/(4*noColumns));
   grid[noColumns-1-i][noRows-1] = p.goWithFlow(PI + 3*PI*i/(4*noColumns));
  }
  //fill the first & last column of the grid
  CxComplex startHereFirst = grid[0][0].goWithFlow(7*PI/4); //start at p_tilde
  CxComplex startHereLast = grid[0][0].goWithFlow(3*PI/4); //start
```

```
   at p_opposite
  for(int i=0; i<noRows-1; i++){
   grid[0][noRows-1-i] = startHereFirst.goWithFlow(i*PI/(4*noRows))
   ; //first col
   grid[noRows-1][i]= startHereLast.goWithFlow(i*PI/(4*noRows)); //
   last col
  }
  //fill the middle column for column
  for(int i = 1; i < noColumns-1; i++){ //go through cols
    for(double j = 1; j < noRows-1; j++){ //go through rows
      double rows = noRows;
      double multi = j/rows;
      CxComplex diff = new CxComplex(subtract(grid[i][noRows-1],
   grid[i][0]));
      diff = add(grid[i][0],mult(multi, diff));
      grid[i][(int)j] = diff.normalize2ndCoordinate();
    }
  }
  return grid;
}

CxComplex[][] getDSectionGridCircular(float varyR, float varyTheta)
   { //gives us north pole d-section
  CxComplex[][] circularGrid = new CxComplex[(int)varyR][(int)
   varyTheta];
  for(int i=0; i<varyR; i++){
    for(int j=0; j<varyTheta; j++){
      Complex Im = new Complex(0,1); //i
      Complex r = new Complex(i/varyR-1);
      Complex Theta = new Complex(j*2*PI/varyTheta-1);
      Complex c_2 = new Complex(Complex.sqrt(Complex.sub(1,Complex.
   pow(r,2))));
      Complex c_1 = new Complex(r.mult(Complex.exp(Im.mult(Theta)))
   );
      circularGrid[i][j] = new CxComplex(c_1, c_2);
    }
  }
  return circularGrid;
}


CxComplex[][] letGridFlow(CxComplex[][] grid, float t){
  CxComplex[][] newGrid = new CxComplex[grid.length][grid[0].length
   ];
  for(int i = 0; i < grid.length; i++){
    for(int j = 0; j < grid[0].length; j++){
    newGrid[i][j] = grid[i][j].goWithFlow(t);
    }
  }
  return newGrid;
}

void displayGrid(CxComplex[][] grid){
  strokeWeight(0.02);
  noFill();
  stroke(255);
  //make lines connecting each column
  for(int i = 0; i < grid[0].length; i++){ //go through rows
```

```
    for(int j = 1; j < grid.length; j++){ //go through cols
      Vector x = new Vector(projectPoint(grid[j-1][i]));
      Vector y = new Vector(projectPoint(grid[j][i]));
      line((float)x.x,(float)x.y,(float)x.z,(float)y.x,(float)y.y,(float)y.z);
    }
  }
  //make lines connecting each row
  for(int i = 0; i < grid.length; i++){ //go through cols
    for(int j = 1; j < grid[0].length; j++){ //go through rows
      Vector x = new Vector(projectPoint(grid[i][j-1]));
      Vector y = new Vector(projectPoint(grid[i][j]));
      line((float)x.x,(float)x.y,(float)x.z,(float)y.x,(float)y.y,(float)y.z);
    }
  }
}

void displayGrid(CxComplex[][] grid, boolean circular){
  strokeWeight(0.02);
  noFill();
  stroke(52,165,218);
  //make lines connecting each column
  for(int i = 0; i < grid[0].length; i++){ //go through rows
    for(int j = 1; j < grid.length; j++){ //go through cols
      Vector x = new Vector(projectPoint(grid[j-1][i]));
      Vector y = new Vector(projectPoint(grid[j][i]));
      line((float)x.x,(float)x.y,(float)x.z,(float)y.x,(float)y.y,(float)y.z);
    }
  }
  //make lines connecting each row
  for(int i = 0; i < grid.length; i++){ //go through cols
    for(int j = 1; j < grid[0].length; j++){ //go through rows
      Vector x = new Vector(projectPoint(grid[i][j-1]));
      Vector y = new Vector(projectPoint(grid[i][j]));
      line((float)x.x,(float)x.y,(float)x.z,(float)y.x,(float)y.y,(float)y.z);
    }
  }
  if (circular == true){ //if circular grid, then connect first and last column
    for(int i= 0; i<grid.length; i++){
      Vector x = new Vector(projectPoint(grid[i][0]));
      Vector y = new Vector(projectPoint(grid[i][grid[0].length-1]));
      line((float)x.x,(float)x.y,(float)x.z,(float)y.x,(float)y.y,(float)y.z);
    }
  }
}

void displayGrid(CxComplex[][] grid, int r, int g, int b){
  strokeWeight(0.02);
  noFill();
  stroke(r, g, b);
  //make lines connecting each column
  for(int i = 0; i < grid[0].length; i++){ //go through rows
```

```
    for(int j = 1; j < grid.length; j++){ //go through cols
     Vector x = new Vector(projectPoint(grid[j-1][i]));
     Vector y = new Vector(projectPoint(grid[j][i]));
     line((float)x.x,(float)x.y,(float)x.z,(float)y.x,(float)y.y,(float)y.z);
    }
  }
  //make lines connecting each row
  for(int i = 0; i < grid.length; i++){ //go through cols
    for(int j = 1; j < grid[0].length; j++){ //go through rows
     Vector x = new Vector(projectPoint(grid[i][j-1]));
     Vector y = new Vector(projectPoint(grid[i][j]));
     line((float)x.x,(float)x.y,(float)x.z,(float)y.x,(float)y.y,(float)y.z);
    }
  }
}

void displayGrid(CxComplex[][] grid, boolean circular, int r, int g, int b){
  strokeWeight(0.02);
  noFill();
  stroke(r, g, b);
  //make lines connecting each column
  for(int i = 0; i < grid[0].length; i++){ //go through rows
    for(int j = 1; j < grid.length; j++){ //go through cols
     Vector x = new Vector(projectPoint(grid[j-1][i]));
     Vector y = new Vector(projectPoint(grid[j][i]));
     line((float)x.x,(float)x.y,(float)x.z,(float)y.x,(float)y.y,(float)y.z);
    }
 }
  //make lines connecting each row
  for(int i = 0; i < grid.length; i++){ //go through cols
    for(int j = 1; j < grid[0].length; j++){ //go through rows
     Vector x = new Vector(projectPoint(grid[i][j-1]));
     Vector y = new Vector(projectPoint(grid[i][j]));
     line((float)x.x,(float)x.y,(float)x.z,(float)y.x,(float)y.y,(float)y.z);
    }
  }
  if (circular == true){ //if circular grid, then connect first and last column
    for(int i= 0; i<grid.length; i++){
      Vector x = new Vector(projectPoint(grid[i][0]));
      Vector y = new Vector(projectPoint(grid[i][grid[0].length-1]));
      line((float)x.x,(float)x.y,(float)x.z,(float)y.x,(float)y.y,(float)y.z);
    }
  }
}

//__A HELICOIDAL ANNULUS__

//set up V_1 and V_2 parts seperately:
CxComplex[][] getV_1part(float varyR, float varyTheta){
  CxComplex[][]V_1_circularGrid = new CxComplex[(int)varyR][(int)
```

```
    varyTheta];
  for(int k=0; k<varyR; k++){
    for(int j=0; j<varyTheta; j++){
      Complex Im = new Complex(0,1); //i
      Complex r = new Complex(k/(varyR-1));
      println("r", k, j, " : ", r.real, r.imag);
      Complex Theta = new Complex(j*2*PI/varyTheta-1);
      Complex c_1 = new Complex(r.mult(sqrt(2)/2));
      Complex c_2 = new Complex(Complex.sqrt(Complex.sub(1,Complex.pow(r,2).mult(0.5))).mult(Complex.exp(Im.mult(Theta))));
      V_1_circularGrid[k][j] = new CxComplex(c_2, c_1);
    }
  }
  return V_1_circularGrid;
}

CxComplex[][] getV_2part(float varyR, float varyTheta){
  CxComplex[][]V_2_circularGrid = new CxComplex[(int)varyR][(int)varyTheta];
  for(int k=0; k<varyR; k++){
    for(int j=0; j<varyTheta; j++){
      Complex Im = new Complex(0,1); //i
      Complex r = new Complex(k*(sqrt(2)/2)/(varyR-1));
      println("r", k, j, " : ", r.real, r.imag);
      Complex Theta = new Complex(j*2*PI/varyTheta-1);
      Complex c_1 = new Complex(Complex.sqrt(Complex.sub(1,Complex.pow(r,2))));
      Complex c_2 = new Complex(r.mult(Complex.exp(Im.mult(Theta))));
      V_2_circularGrid[k][j] = new CxComplex(c_2, c_1);
    }
  }
  return V_2_circularGrid;
}

//__AN ANNULAR 2-SECTION__

CxComplex[][] getV_1part2(float varyR, float varyTheta){
  CxComplex[][]V_1_circularGrid = new CxComplex[(int)varyR][(int)varyTheta];
  for(int k=0; k<varyR; k++){
    for(int j=0; j<varyTheta; j++){
      Complex Im = new Complex(0,1); //i
      Complex MinIm = new Complex(0,-1);
      Complex r = new Complex(k/(varyR-1));
      println("r", k, j, " : ", r.real, r.imag);
      Complex Theta = new Complex(j*2*PI/varyTheta-1);
      Complex c_1 = new Complex(r.mult(sqrt(2)/2).mult(Complex.exp(MinIm.mult(Theta))));
      Complex c_2 = new Complex(Complex.sqrt(Complex.sub(1,Complex.pow(r,2).mult(0.5))).mult(Complex.exp(Im.mult(Theta))));
      V_1_circularGrid[k][j] = new CxComplex(c_1, c_2);
    }
  }
  return V_1_circularGrid;
}

CxComplex[][] getV_2part2(float varyR, float varyTheta){
```

```
  CxComplex[][]V_2_circularGrid = new CxComplex[(int)varyR][(int)varyTheta];
  for(int k=0; k<varyR; k++){
    for(int j=0; j<varyTheta; j++){
      Complex Im = new Complex(0,1); //i
      Complex MinIm = new Complex(0,-1);
      Complex r = new Complex(k*(sqrt(2)/2)/(varyR-1));
      println("r", k, j, " : ", r.real, r.imag);
      Complex Theta = new Complex(j*2*PI/varyTheta-1);
      Complex c_1 = new Complex(Complex.sqrt(Complex.sub(1,Complex.pow(r,2))).mult(Complex.exp(MinIm.mult(Theta))));
      Complex c_2 = new Complex(r.mult(Complex.exp(Im.mult(Theta))));
      V_2_circularGrid[k][j] = new CxComplex(c_1, c_2);
    }
  }
  return V_2_circularGrid;
}

// for the last 2 sections we need this function to visualize the grids:

void displayGrids(CxComplex[][] V_1, CxComplex[][] V_2){
  displayGrid(V_2, true, 0, 150, 120);
  displayGrid(V_1, true, 205, 200, 0);
}

void displayGridsColoured(CxComplex[][] V_1, CxComplex[][] V_2){
  displayGrid(V_2, true, 0, 150, 170);
  displayGrid(V_1, true, 240, 160, 5);
}

//__TUBES__

PVector[][] setupTubes(CxComplex[][] grid, float t){// t is the time that it will flow in the Hopf flow
  int noTotalCoord = 50; //noCoord is how many coordinates will be passed into the curve
  //-----------------------
  //get an array of vectors out of that array of points from that grid
  PVector[][] tubeVectors =new PVector[grid.length * grid[0].length][noTotalCoord];
  for(int i=0; i<grid.length; i++){//go through cols of grid
    for(int j=0; j<grid[0].length; j++){//go through rows of grid
      for(int k=0; k<noTotalCoord; k++){ //go through coordinates
      tubeVectors[i*grid[0].length +j][k] = new PVector((float)goWithFlowAndProject(grid[i][j], k*t/noTotalCoord).x, (float)goWithFlowAndProject(grid[i][j],k*t/noTotalCoord).y, (float)goWithFlowAndProject(grid[i][j],k*t/noTotalCoord).z);
      }
    }
  }
  return tubeVectors;
}

void drawTubeCoord(PVector[][]tubeCoord, float NumCoord){ // NumCoord tells us how many coordinates we should draw
```

```
  float radius = 0.01; //radius of cross section
    for (int j=0; j<tubeCoord.length; j++){
      PVector[] coordinates;
       if(NumCoord >3){
        coordinates = new PVector[(int)NumCoord+1];
          for(int i=0; i<(int)NumCoord+1; i++){
            coordinates[i]=tubeCoord[j][i];
          }
        BSpline3D path = new BSpline3D(coordinates,20); //create
   path for these coordinates
        Oval oval = new Oval(radius, 10); //create cross section
        Tube tube = new Tube(path,oval); //create tube
        tube.drawMode(S3D.SOLID);
        tube.fill(color(150,150,255));
        tube.draw(getGraphics());
       }
    }
}

void drawTubes(Tube[][] tubes){ //draws all tubes completely
  for(int i=0; i<tubes.length; i++){//go through cols
    for(int j=0; j<tubes[0].length; j++){//go through rows
      tubes[i][j].drawMode(S3D.SOLID);
      tubes[i][j].fill(color(150,150,255));
      tubes[i][j].draw(getGraphics());
    }
  }
}
```

Appendix E

Rotation Library

the implementation of the rotation matrices in 3.2.4????

```
  //Here you will find everything about using the SO4 Matrices
  for rotating the d-section around.

PMatrix3D giveRotationMatrix1(float angle_1){
  PMatrix3D rot_1 = new PMatrix3D(cos(angle_1),-sin(angle_1),0,0,
                                  sin(angle_1),cos(angle_1),0,0,
                                  0,0,cos(angle_1),sin(angle_1),
                                  0,0,-sin(angle_1),cos(angle_1));
  return rot_1;
}

PMatrix3D giveRotationMatrix2(float angle_2){


  PMatrix3D rot_2 = new PMatrix3D(0,0,-cos(angle_2),-sin(angle_2),
                                  0,0,sin(angle_2),-cos(angle_2),
                                  cos(angle_2),-sin(angle_2),0,0 ,
                                  sin(angle_2),cos(angle_2),0,0);

  return rot_2;
}
PMatrix3D giveRotationMatrix3(float angle_3){
  PMatrix3D rot_3 = new PMatrix3D(0,-sin(angle_3),-cos(angle_3),0,
                                  sin(angle_3),0,0,-cos(angle_3),
                                  cos(angle_3),0,0,sin(angle_3),
                                  0,cos(angle_3),-sin(angle_3),0 );


  return rot_3;
}

CxComplex[][] rotateDSection(CxComplex[][] grid, float angle, int
  kindOfRotation){
  CxComplex[][] rotGrid = new CxComplex[grid.length][grid[0].length
  ];
  PMatrix3D rot;
  if(kindOfRotation == 1){
    rot = new PMatrix3D(giveRotationMatrix1(angle));
  } else if (kindOfRotation == 2){
    rot = new PMatrix3D(giveRotationMatrix2(angle));
  } else { //(kindOfRotation ==3) or faulty int
```

```
    rot = new PMatrix3D(giveRotationMatrix3(angle));
  }
  for(int i=0; i<grid.length; i++){
    for(int j=0; j<grid[0].length; j++){
      rotGrid[i][j] = grid[i][j].applyRotMatrix(rot);
    }
  }
  return rotGrid;
}
```

Appendix F

Graphics library

```
//__GRAPHICS FUNCTIONS__
//__SETUP COORDINATE SYSTEM__
void centerCoordinatesystem(){
  translate(width/2-50, height/2+50, -100);
  scale(100);
  rotateX(3*PI/8);
  rotateZ(PI/8);

void drawAxes(float size){
  //X  - red
  stroke(192,0,0);
  strokeWeight(0.07);
  line(0,0,0,size,0,0);
  //length indicator
  //line(10,0,-20,10,0,20);

  //Y - green
  stroke(0,192,0);
  line(0,0,0,0,size,0);
  //Z - blue
  stroke(0,0,192);
  line(0,0,0,0,0,size);
}

void drawAxes(float size, float weight){
  //X  - red
  stroke(192,0,0);
  strokeWeight(weight);
  line(0,0,0,size,0,0);
  //length indicator
  //line(10,0,-20,10,0,20);
  //Y - green
  stroke(0,192,0);
  line(0,0,0,0,size,0);
  //Z - blue
  stroke(0,0,192);
  line(0,0,0,0,0,size);
}

void nameAxes(float size){
  textSize(20);
  fill(200);
```

```
  text("x",size+2,0,0);
  text("y",0,size+2,0);
  text("z",0,0,size+2);
}

// overlay coordinate system
void centerCoordinatesystemOverlay(){
  translate(width/7, height/7,0);
  rotateX(3*PI/10);
  rotateY(PI/20);
  rotateZ(PI/10);
}

//__GUI__

int sphere_size = 80;

void drawSphere(){
  strokeWeight(1);
  stroke(255, 70);
  noFill();
  sphere(sphere_size);
  text("N",0,0,83);
  text("x",83,0,0);
  text("y",0,83,0);
}

float getRotation(float rot){
  float currentRot = rot+PI/100;
  return currentRot;
}

void rotateSphere(float rot){
  rotateZ(rot);
}

void drawPointsOnSphere(Complex[]points){
  Vector p = new Vector();
  for (int i=0; i<points.length; i++){
    //get coordinates on sphere
    p = (ComplexToCartesian(points[i]));
    //draw them
    strokeWeight(5);
    stroke((float)projectPoint(findFibrePoint(points[i])).x
   *105+150,(float)projectPoint(findFibrePoint(points[i])).y
   *85+150,(float)projectPoint(findFibrePoint(points[i])).z*70+160)
   ; //colourscheme 01
    point(80*(float)p.x, 80*(float)p.y, 80*(float)p.z);
  }
}

//function that checks if mouse is over sphere
boolean overSphere(){
  if(mouseX<sphere_size+width/7 && mouseY<sphere_size + height/7){
  return true;
  } else {
  return false;
  }
```

```
}

//__SCROLLBARS & BUTTONS__
int width_scrollbars = 150;
int height_scrollbars = 15;
int space = 20; //space to left & top of screen

float x_scrollbars;
float y_VaryTheta = space;
float y_VaryPhi = space*2+height_scrollbars;
float y_Spiral = height-space-height_scrollbars;
float y_1sectionStd = space*5+height_scrollbars*4;
float y_1section = space*6+height_scrollbars*5;
float y_2section = space*7+height_scrollbars*6;
float y_gamma_1 = y_Spiral+7*space+5*height_scrollbars;
float y_gamma_2 = y_Spiral+8*space+6*height_scrollbars;
float y_gamma_3 = y_Spiral+9*space+7*height_scrollbars;
float y_gamma_4 = y_Spiral+10*space+8*height_scrollbars;

void setupScrollbars(){
  x_scrollbars = width-width_scrollbars-space;
  y_VaryTheta = space+height_scrollbars/2;
  y_VaryPhi = y_VaryTheta+2*space+2*height_scrollbars;
  y_Spiral = y_VaryTheta+3*space+3*height_scrollbars;

  s_VaryTheta = new HScrollbar(x_scrollbars, y_VaryTheta,
    width_scrollbars, height_scrollbars, 2);
  s_VaryTheta2 = new HScrollbar(x_scrollbars, y_VaryTheta+space+
    height_scrollbars, width_scrollbars, height_scrollbars, 2);
  s_VaryPhi = new HScrollbar(x_scrollbars, y_VaryPhi,
    width_scrollbars, height_scrollbars, 2);
  s_Spiral = new HScrollbar(x_scrollbars, y_Spiral,
    width_scrollbars, height_scrollbars, 2);

  s_gamma_1 = new HScrollbar(x_scrollbars, y_gamma_1,
    width_scrollbars, height_scrollbars, 2);
  s_gamma_2 = new HScrollbar(x_scrollbars, y_gamma_2,
    width_scrollbars, height_scrollbars, 2);
  s_gamma_3 = new HScrollbar(x_scrollbars, y_gamma_3,
    width_scrollbars, height_scrollbars, 2);

  s_Flow = new HScrollbar(x_scrollbars, height-space/2-space-
    height_scrollbars, width_scrollbars, height_scrollbars, 2);
  s_noCircles = new HScrollbar(x_scrollbars, height-space/2,
    width_scrollbars, height_scrollbars, 2);
  s_noCircles.spos = s_noCircles.xpos +30*width_scrollbars/100;
}

void updateScrollbars(){
  s_VaryTheta.update();
  s_VaryTheta2.update();
  s_VaryPhi.update();
  s_Spiral.update();
  s_gamma_1.update();
  s_gamma_2.update();
  s_gamma_3.update();
  s_Flow.update();
  s_noCircles.update();
```

```
}

void displayScrollbars(){
  s_VaryTheta.display();
  s_VaryTheta2.display();
  s_VaryPhi.display();
  s_Spiral.display();
  s_gamma_1.display();
  s_gamma_2.display();
  s_gamma_3.display();
  s_Flow.display();
  s_noCircles.display();
  //add text
  fill(200);
  textSize(2*space/3);
  text("#fibres", x_scrollbars - space*5,  height-space/4);
}

float scrollbarValue(HScrollbar bar, float maxValue){ //converts position of scrollbar to float between 0 & maxValue
return (bar.spos-bar.xpos)*maxValue/(bar.swidth-bar.sheight);
}

void drawButtons(){
  //draw buttons
  fill(70,0,70);
  strokeWeight(2);
  stroke(255);
  rect(x_scrollbars- height_scrollbars-space, y_VaryTheta- space/3, height_scrollbars, height_scrollbars); //varyTheta
  rect(x_scrollbars- height_scrollbars-space, y_VaryPhi- space/3, height_scrollbars, height_scrollbars); //VaryPhi
  rect(x_scrollbars- height_scrollbars-space, y_Spiral- space/3, height_scrollbars, height_scrollbars); //Spiral
  rect(width-space-height_scrollbars,y_1sectionStd, height_scrollbars, height_scrollbars); //1section
  rect(width-space-height_scrollbars,y_1section, height_scrollbars, height_scrollbars); //1section
  rect(width-space-height_scrollbars,y_2section, height_scrollbars, height_scrollbars); //2section

  rect(x_scrollbars - space, y_gamma_1 - space/3, height_scrollbars, height_scrollbars); //gamma_1
  rect(x_scrollbars - space, y_gamma_2- space/3, height_scrollbars, height_scrollbars); //gamma_2
  rect(x_scrollbars - space, y_gamma_3- space/3, height_scrollbars, height_scrollbars); //gamma_3

  rect(x_scrollbars - space, height-space/2-4*space/3-height_scrollbars, height_scrollbars, height_scrollbars);//flow

  //add text
  fill(200);
  textSize(2*space/3);
  text("Vary Theta", x_scrollbars - space*6, y_VaryTheta + space/3);
  text("Vary Phi", x_scrollbars - space*5, y_VaryPhi+ space/3);
  text("Spiral", x_scrollbars - space*5, y_Spiral+ space/3);
```

```
  text("std 1-section",width-space-height_scrollbars*4-space*3, y_1sectionStd+space/2);
  text("1-section",width-space-height_scrollbars*4-space*2, y_1section+space/2);
  text("2-section",width-space-height_scrollbars*4-space*2, y_2section+space/2);
  text("\u03B3\u2081", x_scrollbars - space*2 - height_scrollbars, y_gamma_1+space/3);
  text("\u03B3\u2082", x_scrollbars - space*2- height_scrollbars, y_gamma_2+space/3);
  text("\u03B3\u2083", x_scrollbars - space*2- height_scrollbars, y_gamma_3+space/3);
  text("D-Section Flow", x_scrollbars - space*6.5,  height-space/2-3*space/4-height_scrollbars);
}

//__functions to check if mouse is over buttons__
boolean overVaryThetaButton()  {
  if (mouseX >= x_scrollbars-height_scrollbars-space && mouseX <= x_scrollbars-space &&
      mouseY >= y_VaryTheta && mouseY <= y_VaryTheta+height_scrollbars) {
    return true;
  } else {
    return false;
  }
}

boolean overVaryPhiButton()  {
  if (mouseX >= x_scrollbars-height_scrollbars-space && mouseX <= x_scrollbars-space &&
      mouseY >= y_VaryPhi && mouseY <= y_VaryPhi+height_scrollbars) {
    return true;
  } else {
    return false;
  }
}

boolean overSpiralButton()  {
  if (mouseX >= x_scrollbars-height_scrollbars-space && mouseX <= x_scrollbars-space &&
      mouseY >= y_Spiral && mouseY <= y_Spiral+height_scrollbars) {
    return true;
  } else {
    return false;
  }
}

boolean over1section()  {
  if (mouseX >= width-space-height_scrollbars && mouseX <= width-space &&
      mouseY >= y_1section && mouseY <= y_1section+height_scrollbars) {
    return true;
  } else {
    return false;
  }
```

```
}

boolean over2section()  {
  if (mouseX >= width-space-height_scrollbars && mouseX <= width-
    space &&
      mouseY >= y_2section && mouseY <= y_2section+
    height_scrollbars) {
    return true;
  } else {
    return false;
  }
}

boolean over1sectionStd()  {
  if (mouseX >= width-space-height_scrollbars && mouseX <= width-
    space &&
      mouseY >= y_1sectionStd && mouseY <= y_1sectionStd+
    height_scrollbars) {
    return true;
  } else {
    return false;
  }
}

boolean overGamma1(){
  if (mouseX >= x_scrollbars - space && mouseX <= x_scrollbars -
    space+height_scrollbars &&
      mouseY >= y_gamma_1 - space/3 && mouseY <= x_scrollbars -
    space+height_scrollbars) {
    return true;
  } else {
    return false;
  }
}

boolean overGamma2(){
  if (mouseX >= x_scrollbars - space && mouseX <= x_scrollbars -
    space+height_scrollbars &&
      mouseY >= y_gamma_2 - space/3 && mouseY <= x_scrollbars -
    space+height_scrollbars) {
    return true;
  } else {
    return false;
  }
}

boolean overGamma3(){
  if (mouseX >= x_scrollbars - space && mouseX <= x_scrollbars -
    space+height_scrollbars &&
      mouseY >= y_gamma_3 - space/3 && mouseY <= x_scrollbars -
    space+height_scrollbars) {
    return true;
  } else {
    return false;
  }
}

boolean overFlow(){
```

```
  if (mouseX >= x_scrollbars - space && mouseX <= x_scrollbars -
   space +height_scrollbars &&
      mouseY >= height-space/2-4*space/3-height_scrollbars  &&
   mouseY <= height-space/2-4*space/3-height_scrollbars+
   height_scrollbars) {
    return true;
  } else {
    return false;
  }
}
```

Appendix G

Class HScrollbar

```
    class HScrollbar {
  int swidth, sheight;    // width and height of bar
  float xpos, ypos;       // x and y position of bar
  float spos, newspos;    // x position of slider
  float sposMin, sposMax; // max and min values of slider
  int loose;              // how loose/heavy
  boolean over;           // is the mouse over the slider?
  boolean locked;
  float ratio;

HScrollbar (float xp, float yp, int sw, int sh, int l) {
    swidth = sw;
    sheight = sh;
    int widthtoheight = sw - sh;
    ratio = (float)sw / (float)widthtoheight;
    xpos = xp;
    ypos = yp-sheight/2;
    spos = xpos + swidth/2 - sheight/2;
    newspos = spos;
    sposMin = xpos;
    sposMax = xpos + swidth - sheight;
    loose = l;
}

  void update() {
    if (overEvent()) {
      over = true;
    } else {
      over = false;
    }
    if (mousePressed && over) {
      locked = true;
    }
    if (!mousePressed) {
      locked = false;
    }
    if (locked) {
      newspos = constrain(mouseX-sheight/2, sposMin, sposMax);
    }
    if (abs(newspos - spos) > 1) {
      spos = spos + (newspos-spos)/loose;
    }
  }
```

```
  float constrain(float val, float minv, float maxv) {
    return min(max(val, minv), maxv);
  }

  boolean overEvent() {
    if (mouseX > xpos && mouseX < xpos+swidth &&
       mouseY > ypos && mouseY < ypos+sheight) {
      return true;
    } else {
      return false;
    }
  }

  void display() {
    noStroke();
    fill(204);
    rect(xpos, ypos, swidth, sheight);
    if (over || locked) {
      fill(0, 0, 0);
    } else {
      fill(102, 102, 102);
    }
    rect(spos, ypos, sheight, sheight);
  }

  float getPos() {
    // Convert spos to be values between
    // 0 and the total width of the scrollbar
    return spos * ratio;
  }
}
```

Appendix H

Rec Library

This library for recording the screen is from Tim Rodenbroeker [6].

```
//For video Export add rec(); at the very end of draw!
final String sketchname = getClass().getName();

import com.hamoid.*;
VideoExport videoExport;

void rec(){
  if(frameCount == 1){
    videoExport = new VideoExport(this,"../"+sketchname+".mp4");
    videoExport.setFrameRate(30);
    videoExport.startMovie();
  }
  videoExport.saveFrame();
}
```

Bibliography

[1] Peter Albers; Hansjörg Geiges; Kai Zehmisch. "A Symplectic Dynamics Proof Of The Degree-Genus-Formula". In: (2019). arXiv:1905.03054.

[2] H. Geiges. "An Introduction to Contact Topology". In: *Cambridge Stud. Adv. Math. 109, Cambridge University Press, Cambridge* (2008).

[3] Peter Lager. *QScript*. http://www.lagers.org.uk/qscript/. Processing Library.

[4] Peter Lager. *Shapes3D*. http://www.lagers.org.uk/s3d4p/index.html. Processing Library.

[5] Processing Foundation. *HScrollbar*. https://processing.org/examples/scrollbar.html. Processing Library.

[6] Tim Rodenbroeker. *rec()*. https://timrodenbroeker.de/processing-tutorial-video-export/. Processing Library.

[7] J. Noble. "Programming Interactivity". In: (2012).

[8] Niles Johnson. *A visualization of the Hopf fibration*. https://nilesjohnson.net/hopf.html.

[9] Niles Johnson. *Talk: What is a fibration?* https://www.youtube.com/watch?v=QXDQsmL-8Us.

[10] Peter Albers. *lecture notes: Kontaktgeometrie*. Heidelberg University. (not published). 2020.