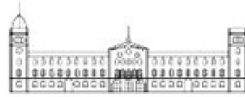

Undergraduate Thesis

MAJOR IN MATHEMATICS

Department of Mathematics

University of Barcelona

Isochrons (and applications to Neuroscience)

Manel Vila Vidal
manel@vilavidal.net

Advisor: Àlex Haro Provinciale
Barcelona, January the 29th, 2015

Acknowledgements

I would like to express my gratitude to Àlex, my advisor, with whom I have learnt a lot during the development of the research and the writing of the program. Thanks also for his encouragement to do things always better and for the coffees that have taken together.

I would also like to thank my father for his support and help, without which this project would have been difficult to accomplish. Thanks also to my mother, my brother and Clara for their understanding and for their constant support. Thanks Clara also for her interest in understanding what I was working on.

Contents

1 Introduction **3**

2 Mathematical analysis **4**

2.1 The quasi-Newton method 8

2.1.1 Substep 1: correction of (K, ω) 8

2.1.2 Substep 2: correction of (N, λ) 11

2.2 Theoretical background 12

2.3 Estimating the error 14

3 The reduced Hodgkin-Huxley model **19**

4 Numerical implementation **24**

4.1 The Taylor method for integrating ODEs 24

4.2 Discrete Fourier transform 26

4.3 The program 28

5 Results **30**

5.1 The reduced Hodgkin-Huxley model 30

5.2 The Rayleigh oscillator 32

6 Conclusions **36**

A The program **38**

Chapter 1

Introduction

Many biological systems exhibit a periodic behaviour. From a mathematical point of view they can be considered as systems moving along a stable limit cycle, that can be parametrised by its phase. The phase can be extended to the whole basin of attraction of the limit cycle via the asymptotic phase and the set of all points having the same asymptotic phase is called isochron. Isochrons were introduced in 1974 by Winfree [5] in order to understand the behaviour of an oscillatory system under a brief stimulus, namely, the phase advance or delay that the system would experience when sent away from the periodic orbit. This helps understanding, for example, the synchronisation in neural nets. Soon after Winfree's paper, Guckenheimer [6] showed that isochrones are in fact the leaves of the stable foliation of the stable manifold of a periodic orbit. Different techniques have been developed to compute the isochrons [1, 8].

An important part of this undergraduate thesis consists on understanding the mathematical concept underlying the idea of isochron. In chapter 2 a definition is given and we describe some properties of isochrons with the objective of being able to find an approximation to first order. We also formulate a functional equation for the parametrisation of the invariant cycle and the tangent vector to the isochrons and we show how it can be solved using a quasi-Newton method. Our arguments are based in [1], where the parametrisation of the whole isochron is found. With some transformations we can simplify our equations and easily solve them. We end the chapter by giving some hints on how to proof the convergence of the method using KAM arguments [2]. In chapter 3 we describe briefly a simplified version of the well-known Hodgkin-Huxley model for the neuron and try to get some insight on what isochrons can tell us about this model.

Another important part of this thesis has been writing a program in language C to implement the algorithm described in chapter 2 in order to be able to apply it to the reduced Hodgkin-Huxley model. The program is described in chapter 4 and the source code is appended. In chapter 5 the results obtained with the program are discussed.

Chapter 2

Mathematical analysis

Consider a differential equation in the plane

$$\dot{x} = X(x), \qquad x \in \mathbb{R}^2, \tag{2.1}$$

where X is analytic. Let $\varphi(t, x_0)$ be the associated flow, i.e.:

$$\begin{cases} \dfrac{d}{dt}\varphi(t, x_0) = X(\varphi(t, x_0)), \\ \varphi(0, x_0) = x_0. \end{cases} \tag{2.2}$$

We assume that X admits a stable hyperbolic limit cycle $\mathcal{K}$. Recall that a limit cycle is an isolated periodic orbit, i.e., a periodic orbit which is the ω-limit or α-limit set of a certain neighbourhood Ω around it. Furthermore, if the limit cycle is stable, it is the ω-limit of Ω, which is called the basin of attraction.

Our first goal is to find a parametrisation for $\mathcal{K}$, i.e. to find $\omega \in \mathbb{R}^+$ and some 1-periodic map:

$$\begin{aligned} K\colon \mathbb{T} = \mathbb{R}/\mathbb{Z} &\to \mathbb{R}^2 \\ \theta &\mapsto K(\theta) \end{aligned} \tag{2.3}$$

such that $\varphi(t, K(\theta)) = K(\theta + \omega t)$ and $K(\mathbb{T}) = \mathcal{K}$. Clearly if $T = 1/\omega$ is the period of the limit cycle $\varphi(T, K(\theta)) = K(\theta + \omega T) = K(\theta + 1) = K(\theta)$.

Proposition 2.1. *The map $K\colon \mathbb{T} \to \mathbb{R}^2$ and the number ω satisfy the following functional equation on K and ω:*

$$X\big(K(\theta)\big) - K'(\theta)\omega = 0 \tag{2.4}$$

Proof. It suffices to insert $\varphi(t, K(\theta)) = K(\theta + \omega t)$ into equation (2.2). □

Note that the solution of (2.4) is not unique. If (K, ω) is a solution of that equation, so is $(\tilde{K}, \omega)$, with $\tilde{K}(\theta) = K(\theta + \theta_0)$, for any $\theta_0 \in \mathbb{T}$. This phase shift, which allows

us to place the phase origin wherever we want in the orbit, is the only source of non-uniqueness for (2.4).

So far $\mathcal{K}$ has been parametrised in such a way that to each point of $\mathcal{K}$ a phase $\theta \in \mathbb{T}$ has been assigned. As $\mathcal{K}$ is a stable limit cycle we have that for each $x \in \Omega$

$$d(\varphi(t,x), \mathcal{K}) \xrightarrow{t\to\infty} 0. \tag{2.5}$$

It is quite natural to wonder what point $K(\theta) \in \mathcal{K}$ evolves under the flow $\varphi(t,\cdot)$ in such a way that

$$d(\varphi(t,x), \varphi(t, K(\theta))) = d(\varphi(t,x), K(\theta + \omega t)) \xrightarrow{t\to\infty} 0. \tag{2.6}$$

As pointed out in [6] it is possible to find a unique $\Theta\colon \Omega \to \mathbb{T}$ such that

$$d(\varphi(t,x), K(\Theta(x) + \omega t)) \xrightarrow{t\to\infty} 0, \tag{2.7}$$

Observe that Θ allows to extend the concept of phase out of the periodic orbit.

Definition 1. We say that a point $x \in \Omega$ is in asymptotic phase with a point $K(\theta) \in \mathcal{K}$ if the following holds

$$d(\varphi(t,x), K(\theta + \omega t)) \xrightarrow{t\to\infty} 0, \tag{2.8}$$

i.e., if $\Theta(x) = \theta$. In this case we say that x has asymptotic phase θ.

Definition 2. The set of points having the same asymptotic phase is called isochron:

$$S_\theta = \{x \in \Omega \colon \Theta(x) = \theta\}. \tag{2.9}$$

Note that in the case that $\mathcal{K}$ is unstable it suffices to reverse time in order to adapt the definitions given here. Note also that our definitions are also valid in the case of $n > 2$, where the isochrons are called isochronous sections and they are manifolds of dimension $n-1$.

The isochrons are the level sets of $\Theta(x)$ and they foliate the basin of attraction. In fact, soon after the isochrons were introduced by Winfree in 1974 [5], Guckenheimer showed [6] that they are the leaves of the stable manifold, that is $W^s(K(\theta))$, for each $\theta \in \mathbb{T}$. The isochrons are therefore tangent to the stable space $E^s(K(\theta))$, for each $\theta \in \mathbb{T}$. As a consequence we can obtain a linear approximation of the isochrons by finding the stable bundle of $\mathcal{K}$.

Proposition 2.2. *For each $\theta \in \mathbb{T}$, let $v_\theta \in T_{K(\theta)}S_\theta$, the tangent space of S_θ in the intersection with $\mathcal{K}$. Then v_θ is an eigenvector of $D\varphi(T, K(\theta))$:*

$$D\varphi(T, K(\theta))v_\theta = e^{\lambda T} v_\theta, \tag{2.10}$$

for some $\lambda \in \mathbb{R}_{<0}$. The other eigenvector of $D\varphi(T, K(\theta))$ is $K(\theta)$ with eigenvalue 1.

Proof. The first statement is a direct consequence of the observation made just before the proposition. We should strictly write $\lambda(\theta)$, but in the next proposition we will see that the stable vectors have the same eigenvalue everywhere. The sign of λ is in agreement with $\mathcal{K}$ being attractive. In the repulsive case we would have $\lambda > 0$.

Furthermore, it follows from the definition of isochron that v_θ and $K'(\theta)$ are transverse. Now consider $\varphi(T, K(\theta)) = K(\theta)$. Differentiating with respect to θ one obtains

$$D\varphi(T, K(\theta))K'(\theta) = K'(\theta), \tag{2.11}$$

which proves the second statement. □

Let $N(0)$ be the stable vector in $K(0)$ with

$$D\varphi(T, K(0))N(0) = e^{\lambda T} N(0). \tag{2.12}$$

In the next proposition we show that the stable vectors along $\mathcal{K}$ can be obtained by simply propagating $N(0)$ under the differential of the flux.

Proposition 2.3. *Let us define*

$$N(\theta) = e^{-\lambda T\theta} D\varphi(T\theta, K(0))N(0), \qquad \theta \in \mathbb{T}. \tag{2.13}$$

The following holds:

$$D\varphi(T, K(\theta))N(\theta) = e^{\lambda T} N(\theta). \tag{2.14}$$

Proof.

$$\begin{aligned}
D\varphi(T, K(\theta))N(\theta) =& e^{-\lambda T\theta} D\varphi(T, K(\theta))D\varphi(T\theta, K(0))N(0) \\
=& e^{-\lambda T\theta} D\varphi(T, \varphi(T\theta, K(0)))D\varphi(T\theta, K(0))N(0) \\
=& e^{-\lambda T\theta} D\varphi(T + T\theta, K(0))N(0) \\
=& e^{-\lambda T\theta} D\varphi(T\theta, \varphi(T, K(0)))D\varphi(T, K(0))N(0) \\
=& e^{-\lambda T\theta} D\varphi(T\theta, K(0))D\varphi(T, K(0))N(0) \\
=& e^{-\lambda T\theta} D\varphi(T\theta, K(0))e^{\lambda T} N(0) \\
=& e^{\lambda T} N(\theta)
\end{aligned} \tag{2.15}$$

□

Proposition 2.4. *The map $N\colon \mathbb{T} \to \mathbb{R}^2$ and the number λ satisfy the following functional equation on N and λ:*

$$DX\big(K(\theta)\big)N(\theta) = \lambda N(\theta) + \omega N'(\theta). \tag{2.16}$$

Proof. Let us recall that by differentiating (2.2) with respect to x_0 one obtains the variational equations:

$$\begin{cases} \frac{d}{dt} D\varphi(t, x_0) = DX(\varphi(t, x_0)) D\varphi(t, x_0), \\ D\varphi(0, x_0) = \mathrm{id}_2 \,. \end{cases} \tag{2.17}$$

If we differentiate $N(\theta) = e^{-\lambda T\theta} D\varphi(T\theta, K(0))N(0)$ with respect to θ:

$$\begin{aligned} N'(\theta) =& -\lambda T e^{-\lambda T\theta} D\varphi(T\theta, K(0))N(0)+ \\ & + T e^{-\lambda T\theta} \frac{d}{dt} D\varphi(T\theta, K(0))N(0) = \\ =& -\lambda T N(\theta)+ \\ & + T e^{-\lambda T\theta} DX(\varphi(T\theta, K(0)))D\varphi(T\theta, K(0))N(0) = \\ =& -\lambda T N(\theta) + T DX(K(\theta))N(\theta), \end{aligned} \tag{2.18}$$

which yields the desired result after multiplying by ω. □

Note that the solutions of (2.16) are not unique. If (N, λ) is a solution of that equation, so is $(\tilde{N}, \lambda)$, with $\tilde{N}(\theta) = bN(\theta)$, for any $b \in \mathbb{R} \setminus \{0\}$. This rescaling factor is the only source of non-uniqueness for (2.16).

Let us now sum up what we have done so far. Two approaches for obtaining (K, ω) and (N, λ) have arisen. The first one consists on determining first a point in $\mathcal{K}$, which we will name $K(0)$. Letting $K(0)$ evolve under the flux one can obtain the whole limit cycle and the period T, i.e., the number such that $\varphi(T, K(0)) = K(0)$. Additionally, one can obtain $N(0)$ and λ by solving the eigenvalue problem $D\varphi(T, K(0))N(0) = e^{\lambda T} N(0)$ and then propagate $N(0)$ under the differential of the flux in order to obtain $N(\theta)$.

For the second approach consider the functional equations (2.4) and (2.16), which must be thought of as equations for the mappings K and N and for the real numbers ω and λ. In the rest of the chapter we will show how they can be solved by means of a Newton-like method, provided that a good initial approximation is known. In section 2.1 we will design the method to be used in an informal way. In section 2.2 we will introduce some norms and give some basic properties that will allow us to give some estimate of the errors after one Newton step is performed. The error estimates will be given in section 2.3 and we will see that they are quadratically small. In this section we will also give a sketch of the proof of the convergence of the method. The kind of reasoning that we will do is very similar to that carried out in [1], where a parametrisation of the whole isochron is found by solving a functional equation. Our arguments are also very standard in KAM theory, and we will therefore follow very closely [2], where typical KAM problems are presented. However, note that "small divisors", a central piece in KAM theory, do not appear here.

2.1 The quasi-Newton method

Our objective is to find K, ω, N and λ that satisfy the functional equations derived in the previous section, namely:

$$\begin{aligned} & X\big(K(\theta)\big) - K'(\theta)\omega = 0 \\ & DX\big(K(\theta)\big)N(\theta) - N'(\theta)\omega - N(\theta)\lambda = 0 \end{aligned} \tag{2.19}$$

Suppose, however, that we have found K, ω, N and λ which are only an approximation, so that

$$\begin{aligned} & X\big(K(\theta)\big) - K'(\theta)\omega = E_K(\theta) \\ & DX\big(K(\theta)\big)N(\theta) - N'(\theta)\omega - N(\theta)\lambda = E_N(\theta) \end{aligned} \tag{2.20}$$

In this section we will describe how to refine this solutions using a quasi-Newton method.

2.1.1 Substep 1: correction of (K, ω)

We could in fact improve our solution (K, ω) alone without making use of N. However it proves useful to improve K and N together as seen below.

The functional equation to solve for K and ω is

$$X\big(K(\theta)\big) - K'(\theta)\omega = 0. \tag{2.21}$$

Given an approximate solution (K, ω) of (2.21) such that

$$X\big(K(\theta)\big) - K'(\theta)\omega = E_K(\theta), \tag{2.22}$$

we are looking for $(\Delta K, \delta\omega)$ such that $(K + \Delta K, \omega + \delta\omega)$ eliminates the error $E_K(\theta)$ in the linear approximation:

Retaining only linear terms in the corrections:

$$X\big(K(\theta) + \Delta K(\theta)\big) \approx X\big(K(\theta)\big) + DX\big(K(\theta)\big)\Delta K(\theta), \tag{2.23}$$

and

$$\big(K'(\theta) + \Delta K'(\theta)\big)(\omega + \delta\omega) \approx K'(\theta)\omega + K'(\theta)\delta\omega + \Delta K'(\theta)\omega, \tag{2.24}$$

which, when inserted in (2.21) yield

$$\begin{aligned} 0 =& X\big(K(\theta) + \Delta K(\theta)\big) - \big(K'(\theta) + \Delta K'(\theta)\big)(\omega + \delta\omega) \approx \\ \approx& X\big(K(\theta)\big) + DX\big(K(\theta)\big)\Delta K(\theta) - K'(\theta)\omega - K'(\theta)\delta\omega - \Delta K'(\theta)\omega = \\ =& E_K(\theta) + DX\big(K(\theta)\big)\Delta K(\theta) - K'(\theta)\delta\omega - \Delta K'(\theta)\omega, \end{aligned} \tag{2.25}$$

Therefore the corrections must satisfy the following equation:

$$-E_K(\theta) = DX\big(K(\theta)\big)\Delta K(\theta) - K'(\theta)\delta\omega - \Delta K'(\theta)\omega. \tag{2.26}$$

However, we will not solve this equation directly. A change of variables along with some further approximations will allow us to simplify it. The simplified equation will be easily solved. This is what we call a quasi-Newton method.

Consider the following adapted frame[1]:

$$P(\theta) = \Big(L(\theta)\Big|N(\theta)\Big) = \Big(K'(\theta)\Big|N(\theta)\Big). \tag{2.27}$$

We may now rewrite E_K and ΔK in this frame:

$$\begin{aligned} \Delta K(\theta) &= P(\theta)\xi(\theta), \\ E_K(\theta) &= P(\theta)\eta(\theta), \end{aligned} \tag{2.28}$$

Inserting this expressions in (2.26) and multiplying by P^{-1} in order to rewrite it in the frame we obtain:

$$-\eta(\theta) = P^{-1}(\theta)DX\big(K(\theta)\big)P(\theta)\xi(\theta) - \begin{pmatrix}\delta\omega \\ 0\end{pmatrix} - P^{-1}(\theta)P'(\theta)\xi(\theta)\omega - \xi'(\theta)\omega, \tag{2.29}$$

which needs to be solved for ξ and $\delta\omega$.

In the first term on the right-hand side of the last equation we can do the following approximation

$$\begin{aligned} P^{-1}(\theta)DX\big(K(\theta)\big)P(\theta)\xi(\theta) =& P^{-1}(\theta)\Big(DX\big(K(\theta)\big)K'(\theta)\Big|DX\big(K(\theta)\big)N(\theta)\Big)\xi(\theta) = \\ =& P^{-1}(\theta)\Big(K''(\theta)\omega + E'_K(\theta)\Big|N'(\theta)\omega + N(\theta)\lambda + E_N(\theta)\Big)\xi(\theta) \approx \\ \approx& P^{-1}(\theta)\Big(K''(\theta)\omega\Big|N'(\theta)\omega + N(\theta)\lambda\Big)\xi(\theta) = \\ =& P^{-1}(\theta)\Big(K''(\theta)\Big|N'(\theta)\Big)\xi(\theta)\omega + P^{-1}(\theta)\Big(0\Big|N(\theta)\Big)\xi(\theta)\lambda = \\ =& P^{-1}(\theta)P'(\theta)\xi(\theta)\omega + P^{-1}(\theta)\Big(0\Big|N(\theta)\Big)\xi(\theta)\lambda = \\ =& P^{-1}(\theta)P'(\theta)\xi(\theta)\omega + \begin{pmatrix}0 & 0 \\ 0 & \lambda\end{pmatrix}\xi(\theta), \end{aligned} \tag{2.30}$$

[1] Note that any vector V satisfying $\langle K', V\rangle \neq 0$ could have been chosen for the second component and there would be no need to introduce N. However we select N since this highly simplifies our calculations. It is clear that the exact K' and N are transverse, but it may not be the case with the approximations. We assume that the non-degeneracy condition is satisfied if the initial approximation is close enough to the real solution.

where we have made use of (2.20) and assumed that, since $E'_K(\theta)$ is controlled by $E_K(\theta)$, and $\xi(\theta)$ is of the same order of smallness as $E_K(\theta)$, $E'_K(\theta)\xi(\theta)$ and $E_N(\theta)\xi(\theta)$ are quadratically small and can be ignored in the linear approximation.

Inserting (2.30) into (2.29) we obtain the cohomological equation

$$-\eta(\theta) = \begin{pmatrix} 0 & 0 \\ 0 & \lambda \end{pmatrix} \xi(\theta) - \begin{pmatrix} \delta\omega \\ 0 \end{pmatrix} - \xi'(\theta)\omega. \tag{2.31}$$

This equation can be readily split into two uncoupled differential equations in the L and N components and we realise that the change of variables introduced has allowed us to reduce the initial equation to a much simpler one up to quadratically small terms,

$$\begin{cases} \eta^L(\theta) = \omega\partial_\theta\xi^L(\theta) + \delta\omega, \\ \eta^N(\theta) = \omega\partial_\theta\xi^N(\theta) - \lambda\xi^N(\theta), \end{cases} \tag{2.32}$$

that can be solved by considering Fourier series expansions for $\xi^L(\theta)$, $\xi^N(\theta)$, $\eta^L(\theta)$ and $\eta^N(\theta)$:

For the first equation we have:

$$\sum_{k\in\mathbb{Z}} \eta_k^L e^{2\pi ik\theta} = \delta\omega + \sum_{k\in\mathbb{Z}} 2\pi ik\omega\xi_k^L e^{2\pi ik\theta}, \tag{2.33}$$

which yields the following solution

$$\delta\omega = \eta_0^L, \qquad \xi_k^L = \frac{\eta_k^L}{2\pi ik\omega}, \quad \text{for } k \neq 0. \tag{2.34}$$

Observe that there is a free parameter, namely,

$$\xi_0^L = \langle \xi^L \rangle = \int_0^1 \xi^L(\theta)\ \mathrm{d}\theta. \tag{2.35}$$

This parameter can be adjusted at each step so that there is no shift in the initial parametrisation phase, so that $\Delta K(0) = 0$, or we can take it to be zero at each step and adjust it at the end if needed.

The second equation

$$\sum_{k\in\mathbb{Z}} \eta_k^N e^{2\pi ik\theta} = \sum_{k\in\mathbb{Z}} (2\pi ik\omega - \lambda)\xi_k^N e^{2\pi ik\theta}, \tag{2.36}$$

has the following solution

$$\xi_k^N = \frac{\eta_k^N}{2\pi ik\omega - \lambda}. \tag{2.37}$$

2.1.2 Substep 2: correction of (N, λ)

Once (K, ω) has been improved to $(\tilde{K}, \tilde{\omega}) = (K + \Delta K, \omega + \delta\omega)$, a similar reasoning can be made to refine (N, λ). We start the correction by redefining the errors at the mid-step:

$$X\big(\tilde{K}(\theta)\big) - \tilde{K}'(\theta)\tilde{\omega} = \tilde{E}_K(\theta) \tag{2.38}$$

$$DX\big(\tilde{K}(\theta)\big)N(\theta) - N'(\theta)\tilde{\omega} - N(\theta)\lambda = \tilde{E}_N(\theta) \tag{2.39}$$

We proceed as before introducing the corrected solution $(N + \Delta N, \lambda + \delta\lambda)$ in the equation for N and λ,

$$DX\big(\tilde{K}(\theta)\big)N(\theta) - N'(\theta)\tilde{\omega} - N(\theta)\lambda = 0, \tag{2.40}$$

and retaining only linear terms in the corrections:

$$\begin{aligned} DX\big(\tilde{K}(\theta)\big)\big(N(\theta) + \Delta N(\theta)\big) &= DX\big(\tilde{K}(\theta)\big)N(\theta) + DX\big(\tilde{K}(\theta)\big)\Delta N(\theta), \\ \big(N'(\theta) + \Delta N'(\theta)\big)\tilde{\omega} &= N'(\theta)\tilde{\omega} + \Delta N'(\theta)\tilde{\omega}, \\ \big(N(\theta) + \Delta N(\theta)\big)(\lambda + \delta\lambda) &\approx N(\theta)\lambda + N(\theta)\delta\lambda + \Delta N(\theta)\lambda, \end{aligned} \tag{2.41}$$

which yields:

$$-\tilde{E}_N(\theta) = DX\big(\tilde{K}(\theta)\big)\Delta N(\theta) - N(\theta)\delta\lambda - \Delta N'(\theta)\tilde{\omega} - \Delta N(\theta)\lambda. \tag{2.42}$$

We can now rewrite E_N and ΔN in the frame $\tilde{P}(\theta) = \Big(\tilde{L}(\theta)\Big|N(\theta)\Big) = \Big(\tilde{K}'(\theta)\Big|N(\theta)\Big)$:

$$\begin{aligned} \Delta N(\theta) &= \tilde{P}(\theta)\nu(\theta), \\ \tilde{E}_N(\theta) &= \tilde{P}(\theta)\zeta(\theta), \end{aligned} \tag{2.43}$$

Inserting this expressions in (2.42) and multiplying by $\tilde{P}^{-1}$ in order to rewrite it in the frame we obtain:

$$-\zeta(\theta) = \tilde{P}^{-1}(\theta)DX\big(\tilde{K}(\theta)\big)\tilde{P}(\theta)\nu(\theta) - \begin{pmatrix} 0 \\ \delta\lambda \end{pmatrix} - \tilde{P}^{-1}(\theta)\tilde{P}'(\theta)\nu(\theta)\tilde{\omega} - \nu'(\theta)\tilde{\omega} - \nu(\theta)\lambda, \tag{2.44}$$

which needs to be solved for ν and $\delta\lambda$.

As before some further approximations can be made:

$$\begin{aligned}
\tilde{P}^{-1}(\theta)DX\big(\tilde{K}(\theta)\big)\tilde{P}(\theta)\nu(\theta) =& \tilde{P}^{-1}(\theta)\Big(DX\big(\tilde{K}(\theta)\big)\tilde{K}'(\theta)\Big|DX\big(\tilde{K}(\theta)\big)N(\theta)\Big)\nu(\theta) = \\
=& \tilde{P}^{-1}(\theta)\Big(\tilde{K}''(\theta)\tilde{\omega} + \tilde{E}_K'(\theta)\Big|N'(\theta)\tilde{\omega} + N(\theta)\lambda + \tilde{E}_N(\theta)\Big)\nu(\theta) \approx \\
\approx& \tilde{P}^{-1}(\theta)\Big(\tilde{K}''(\theta)\tilde{\omega}\Big|N'(\theta)\tilde{\omega} + N(\theta)\lambda\Big)\nu(\theta) = \\
=& \tilde{P}^{-1}(\theta)\Big(\tilde{K}''(\theta)\Big|N'(\theta)\Big)\nu(\theta)\tilde{\omega} + \tilde{P}^{-1}(\theta)\Big(0\Big|N(\theta)\Big)\nu(\theta)\lambda = \\
=& \tilde{P}^{-1}(\theta)\tilde{P}'(\theta)\nu(\theta)\tilde{\omega} + \tilde{P}^{-1}(\theta)\Big(0\Big|N(\theta)\Big)\nu(\theta)\lambda = \\
=& \tilde{P}^{-1}(\theta)\tilde{P}'(\theta)\nu(\theta)\tilde{\omega} + \begin{pmatrix} 0 & 0 \\ 0 & \lambda \end{pmatrix}\nu(\theta),
\end{aligned} \tag{2.45}$$

where we have dropped again $\tilde{E}_K'(\theta)\nu(\theta)$ and $\tilde{E}_N(\theta)\nu(\theta)$.

Inserting (2.45) into (2.44) we obtain the cohomological equation for the correction of N and λ:

$$-\zeta(\theta) = \begin{pmatrix} 0 & 0 \\ 0 & \lambda \end{pmatrix}\nu(\theta) - \begin{pmatrix} 0 \\ \delta\lambda \end{pmatrix} - \nu'(\theta)\tilde{\omega} - \nu(\theta)\lambda, \tag{2.46}$$

which gives two differential equations of the same kind as obtained in the previous section:

$$\begin{cases} \zeta^L(\theta) = \tilde{\omega}\partial_\theta\nu^L(\theta) + \lambda\nu^L(\theta), \\ \zeta^N(\theta) = \tilde{\omega}\partial_\theta\nu^N(\theta) + \delta\lambda, \end{cases} \tag{2.47}$$

From the first equation we can obtain the Fourier coefficients of $\nu^L(\theta)$:

$$\nu_k^L = \frac{\zeta_k^L}{2\pi i k\tilde{\omega} + \lambda}, \tag{2.48}$$

and from the second those of $\nu^N(\theta)$ and $\delta\lambda$:

$$\delta\lambda = \zeta_0^N, \qquad \nu_k^N = \frac{\zeta_k^N}{2\pi i k\tilde{\omega}}, \quad \text{for } k \neq 0. \tag{2.49}$$

The coefficient $\nu_0^N = \langle\nu^N\rangle$ is now free and it determines the normalisation condition upon N. We can set $\nu_0^N = 0$ at each step and rescale vectors N at the end of the Newton method if needed.

2.2 Theoretical background

As we want to estimate the errors we need to be able to measure functions in some function space. For the following arguments we need to consider that all the functions that have appeared so far can be analytically extended to a complex neighbourhood.

Consider a complex strip of width $\rho > 0$:

$$\mathbb{T}_\rho = \{\theta \in \mathbb{C}/\mathbb{Z} \mid |\operatorname{Im}(\theta)| < \rho\}. \tag{2.50}$$

Definition 3. Given a holomorphic periodic function $f : \mathbb{T}_\rho \to \mathbb{C}$ we define the following norm

$$\|f\|_\rho = \sup_{\theta \in \mathbb{T}_\rho} |f(\theta)|. \tag{2.51}$$

The set of functions with finite $\|\cdot\|_\rho$ norm form a Banach space with this norm.

For vector-valued functions we will use the maximum norm. Given a periodic function $f = (f_1, f_2) : \mathbb{T}_\rho \to \mathbb{C}^2$ we extend the norm in the following way

$$\|f\|_\rho = \max(\|f_1\|_\rho, \|f_2\|_\rho). \tag{2.52}$$

For matrices of holomorphic functions on $\mathbb{T}_\rho$ the corresponding induced norm will be used. Given a 2×2 matrix $A = (f|g)$ we define:

$$\|A\|_\rho = \max(\|f_1\|_\rho + \|g_1\|_\rho, \|f_2\|_\rho + \|g_2\|_\rho). \tag{2.53}$$

Proposition 2.5. *Let f be an holomorphic function on $\mathbb{T}_\rho$. For any $\delta > 0$ the derivative of $f(\theta)$, $f'(\theta)$, is holomorphic on $\mathbb{T}_{\rho-\delta}$ and the following holds:*

$$\|f'\|_{\rho-\delta} \leq \frac{1}{\delta} \|f\|_\rho . \tag{2.54}$$

Proof. For every $\theta \in \mathbb{T}_\rho$ consider the closed disk D_r contained in $\mathbb{T}_\rho$. It is a typical result of complex analysis that

$$|f'(\theta)| \leq \frac{|f(\theta)|}{r}. \tag{2.55}$$

The largest r that one can choose for every $\theta \in \mathbb{T}_{\rho-\delta}$ is δ. Taking supremums at both sides yields the desired result. If a vector-valued function $f = (f_1, f_2)$ is considered:

$$\|f'\|_{\rho-\delta} = \max(\|f_1\|_{\rho-\delta}, \|f_2\|_{\rho-\delta}) \leq \frac{1}{\delta} \max(\|f_1'\|_\delta, \|f_2'\|_\rho) = \frac{1}{\delta} \|f\|_\rho . \tag{2.56}$$

□

Proposition 2.6. *Let X be an analytic vector field in a domain $U \subset \mathbb{C}^2$. Let $K \colon \mathbb{T}_\rho \to \mathbb{C}^2$ be such that*

$$d(K(\mathbb{T}_\rho), \mathbb{C}^2 - U) \geq \kappa > 0. \tag{2.57}$$

Then the following hold:

- *$X \circ K$ is holomorphic on $\mathbb{T}_\rho$.*

- *For* $\Delta K\colon \mathbb{T}_\rho \to \mathbb{C}^2$ *with* $\|\Delta K\|_\rho$ *sufficiently small (so that* $d((K+\Delta K)(\mathbb{T}_\rho), \mathbb{C}^2 - U) > 0$*), we have*

$$\|X \circ (K + \Delta K) - X \circ K - (DX \circ K)\Delta K\|_\rho \leq C \|\Delta K\|_\rho^2 . \tag{2.58}$$

Proof. It follows from Taylor's theorem:

$$\begin{aligned} X\big(K(\theta) + \Delta K(\theta)\big) &= X\big(K(\theta)\big) + DX\big(K(\theta)\big)\Delta K(\theta) + \\ &+ \int_0^1 (1-t) DX^2\big(K(\theta) + t\Delta K(\theta)\big)[\Delta K(\theta), \Delta K(\theta)]t \end{aligned} \tag{2.59}$$

Taking supremums at both sides yields the desired result with $C = \frac{1}{2}\sup_{x\in U}\|D^2X\|$. □

Proposition 2.7. *Let* $f\colon \mathbb{T}_\rho \to \mathbb{C}$ *be written as a Fourier series*

$$f(\theta) = \sum_{k\in\mathbb{Z}} f_k e^{2\pi i k\theta}. \tag{2.60}$$

Then for all $0 < \tilde\rho < \rho$ *the following holds:*

$$|f_k| \leq e^{-2\pi|k|\tilde\rho} \|f\|_\rho \leq \|f\|_\rho \tag{2.61}$$

Proof. For $k > 0$ the path of integration can be shifted downwards:

$$\begin{aligned} f_k = \int_{\mathbb{T}} f(\theta) e^{-2\pi i k\theta} d\theta &= \int_0^{-\tilde\rho} f(0 + is) e^{-2\pi i k(0+is)} ds \\ &+ \int_{\mathbb{T}} f(\theta - i\tilde\rho) e^{-2\pi i k(\theta - i\tilde\rho)} d\theta + \int_{-\tilde\rho}^0 f(1 + is) e^{-2\pi i k(1+is)} ds. \end{aligned} \tag{2.62}$$

As $f(0 + is) = f(1 + is)$ we have that

$$f_k = \int_{\mathbb{T}} f(\theta - i\tilde\rho) e^{-2\pi i k(\theta - i\tilde\rho)} d\theta = e^{-2\pi i k\tilde\rho} \int_{\mathbb{T}} f(\theta - i\tilde\rho) e^{-2\pi i k\theta} d\theta, \tag{2.63}$$

so that

$$|f_k| \leq e^{-2\pi k\tilde\rho} \int_{\mathbb{T}} \|f\|_\rho d\theta = e^{-2\pi k\tilde\rho} \|f\|_\rho . \tag{2.64}$$

For $k < 0$ shift the path of integration upwards. For $k = 0$ the path along the real line gives the desired result. □

2.3 Estimating the error

So far we have presented the quasi-Newton method in an informal way by simply dropping some terms that we have considered to be quadratically small. We will here

give an estimate of the error after one step and see that it is bounded by the square of the error before performing the step. When developing the method we have highlighted the approximately equal symbol $\approx$ in order to keep track of all the terms that have been ignored.

We will denote the errors after the first half step with $\tilde{E}$ and the errors after the whole quasi-Newton step has been performed as $\bar{E}$.

Proposition 2.8. *Assume that X is analytic in some domain $U \subset \mathbb{C}^2$. Let $K\colon \mathbb{T}_\rho \to U$ be such that*

$$d(K(\mathbb{T}_\rho), \mathbb{C}^2 - U) \geq \kappa > 0 \tag{2.65}$$

and let $N\colon \mathbb{T}_\rho \to \mathbb{C}^2$. Let $\epsilon \geq 0$ be such that

$$\begin{aligned} \|E_K\|_\rho &= \|X \circ K - \omega K'\|_\rho \leq \epsilon \\ \|E_N\|_\rho &= \|(DX \circ K)N - \omega N' - \lambda N\|_\rho \leq \epsilon \end{aligned} \tag{2.66}$$

and let $\delta > 0$ be such that it satisfies a certain relation

$$\delta^{-1} C\epsilon < 1, \tag{2.67}$$

for a certain constant C depending on κ, $|\omega|$, $|\lambda|$, $\sup_{x\in U} \|D^2 X\|$, $\|P\|_\rho$, $\|P^{-1}\|_\rho$, where $P = (K'|N)$. Then the errors $\bar{E}_N$, $\bar{E}_K$ for the improved solutions $(K + \Delta K, \omega + \delta\omega)$ and $(N + \Delta N, \lambda + \delta\lambda)$ obtained after the quasi-Newton step satisfy

$$\begin{aligned} \left\|\bar{E}_N\right\|_{\rho-\delta} &\leq C\delta^{-1}\epsilon^2, \\ \left\|\bar{E}_K\right\|_{\rho-\delta} &\leq C\delta^{-1}\epsilon^2, \end{aligned} \tag{2.68}$$

where the constant C appearing in the conclusions is different than that appearing in the hypothesis, as is common practice in KAM arguments.

Only a sketch of the proof will be given:

Let us first consider substep 1. Using proposition 2.7 and the explicit expressions for the corrections in the frame derived in the last section we can give the following bounds:

$$\begin{aligned} &|\delta\omega| \leq \|\eta\|_\rho \leq \left\|P^{-1}\right\|_\rho \|E_K\|_\rho \leq C\epsilon, \\ &\|\xi^L\|_\rho \leq \frac{1}{\omega}\|\eta^L\|_\rho \leq \left(\frac{1}{\omega} + \frac{1}{\lambda}\right)\|\eta^L\|_\rho, \\ &\|\xi^N\|_\rho \leq \left(\frac{1}{\omega} + \frac{1}{\lambda}\right)\|\eta^N\|_\rho, \\ &\|\xi\|_\rho \leq \left(\frac{1}{\omega} + \frac{1}{\lambda}\right)\|\eta\|_\rho \leq \left(\frac{1}{\omega} + \frac{1}{\lambda}\right)\|P^{-1}\|_\rho \|E_K\|_\rho \leq C\epsilon \end{aligned} \tag{2.69}$$

From these expressions and using also proposition 2.5 we can obtain the following bounds:

$$
\begin{aligned}
&\|E'_K\|_{\rho-\delta} \leq \frac{C}{\delta}\|E_K\|_\rho \leq \frac{C}{\delta}\epsilon, \\
&\|\Delta K\|_\rho \leq \|P\|_\rho \|\xi\|_\rho \leq C\epsilon, \\
&\|\Delta K'\|_{\rho-\delta} \leq \frac{C}{\delta}\|\Delta K\|_\rho \leq \frac{C}{\delta}\epsilon.
\end{aligned}
\tag{2.70}
$$

Now if we go back to substep 1 of the Newton method, where the correction of K and ω is considered we can add and substract terms to obtain the following expression:

$$
\begin{aligned}
\tilde{E}_K(\theta) =& X\big(K(\theta)+\Delta K(\theta)\big) - (K'(\theta)+\Delta K'(\theta))(\omega+\delta\omega) = \\
=& X\big(K(\theta)+\Delta K(\theta)\big) - X\big(K(\theta)\big) - DX\big(K(\theta)\big)\Delta K(\theta) + \\
&+ X\big(K(\theta)\big) + DX\big(K(\theta)\big)\Delta K(\theta) - K'(\theta)\omega - \Delta K'(\theta)\omega - K'(\theta)\delta\omega - \Delta K'(\theta)\delta\omega = \\
=& X\big(K(\theta)+\Delta K(\theta)\big) - X\big(K(\theta)\big) - DX\big(K(\theta)\big)\Delta K(\theta) + \\
&+ E_K(\theta) + DX\big(K(\theta)\big)\Delta K(\theta) - \Delta K'(\theta)\omega - K'(\theta)\delta\omega - \\
&- \Delta K'(\theta)\delta\omega = \\
=& X\big(K(\theta)+\Delta K(\theta)\big) - X\big(K(\theta)\big) - DX\big(K(\theta)\big)\Delta K(\theta) + \\
&+ E_K(\theta) - E_K(\theta) + P^{-1}(\theta)\Big(E'_K(\theta)\Big|E_N(\theta)\Big)\xi(\theta) \\
&- \Delta K'(\theta)\delta\omega = \\
=& X\big(K(\theta)+\Delta K(\theta)\big) - X\big(K(\theta)\big) - DX\big(K(\theta)\big)\Delta K(\theta) + \\
&+ P^{-1}(\theta)\Big(E'_K(\theta)\Big|E_N(\theta)\Big)\xi(\theta) \\
&- \Delta K'(\theta)\delta\omega = \\
=& A_1 + A_2 + A_3,
\end{aligned}
\tag{2.71}
$$

where we denote each line in the last expression by A_1, A_2, A_3.

Making use of the bounds that we have just computed we can estimate each line as follows:

$$
\begin{aligned}
\|A_1\|_\rho &\leq C\|\Delta K\|_\rho^2 \leq C\epsilon^2, \\
\|A_2\|_{\rho-\delta} &\leq \left\|P^{-1}\right\|_{\rho-\delta}\left\|E'_K\xi^L + E_N\xi^N\right\|_{\rho-\delta} \leq \\
&\leq \left\|P^{-1}\right\|_{\rho-\delta}\Big(\|E'_K\|_{\rho-\delta}\left\|\xi^L\right\|_{\rho-\delta} + \|E_N\|_{\rho-\delta}\left\|\xi^N\right\|_{\rho-\delta}\Big) \leq \\
&\leq \frac{C}{\delta}\epsilon^2 + C\epsilon^2, \\
\|A_3\|_{\rho-\delta} &\leq \|\Delta K'\|_{\rho-\delta}|\delta\omega| \leq \frac{C}{\delta}\epsilon^2.
\end{aligned}
\tag{2.72}
$$

Note that for the first estimate the proposition 2.6 has been used.

We can see that each term is bounded by the square of the initial error as desired.

A similar reasoning can be applied for the mid-step error for the functional equation on N:

$$\begin{aligned} \tilde{E}_N(\theta) =& DX\big(K(\theta) + \Delta K(\theta)\big)N(\theta) - N'(\theta)(\omega + \delta\omega) - N(\theta)\lambda = \\ =& DX\big(K(\theta) + \Delta K(\theta)\big)N(\theta) - DX\big(K(\theta)\big)N(\theta) + \\ & + DX\big(K(\theta)\big)N(\theta) - N'(\theta)\omega - N(\theta)\lambda - \\ & - N'(\theta)\delta\omega = \\ =& B_1 + E_N(\theta) + B_2, \end{aligned} \tag{2.73}$$

where we use a similar notation as before.

Using Taylor in a similar way as in proposition 2.6 one can find

$$\|B_1\|_\rho \le C \|\Delta K\|_\rho \|N\|_\rho \le C\epsilon, \tag{2.74}$$

and using proposition 2.5:

$$\|B_2\|_{\rho-\delta} \le \|N'\|_{\rho-\delta} |\delta\omega| \le \frac{C}{\delta}\epsilon, \tag{2.75}$$

so that $\tilde{E}_N$, the mid-step error for N, is of the order of E_N.

The same kind of bounds can be found for the correction of N and λ, but they will not be specified here, since it is out of the scope of this undergraduate thesis.

However, a key point must be noted. To control the error in the second substep we must be able to control the inverse of $\tilde{P}$. This can be done using Neumann series. Consider the following expression:

$$\tilde{P}^{-1} = (\tilde{P} - P + P)^{-1} = P^{-1}(I + (\tilde{P} - P)P^{-1})^{-1} = P^{-1}(I - X + X^2 - \dots), \tag{2.76}$$

where $X = P^{-1}(\tilde{P} - P)$.

The matrix $\tilde{P}$ is invertible if P is invertible and $\|X\|_\rho < 1$. And we can write

$$\|X\|_\rho = \left\|\tilde{P} - P\right\|_\rho \cdot \left\|P^{-1}\right\|_\rho < 1 \tag{2.77}$$

From this we can clearly see that the worse conditioned P is, the closer $\tilde{P}$ needs to be to P, which means that the error in the initial approximation needs to be smaller. In fact from (2.76) we can write the following bound

$$\left\|\tilde{P}^{-1}\right\|_\rho \le \|P\|_\rho \frac{1}{1 - \|X\|_\rho}, \tag{2.78}$$

which would allow us to obtain a specific condition that could be included in the constant C appearing in the hypothesis. We would like to repeat now that the greater

C is, the smaller ϵ can be, i.e., the better the initial condition passed to the algorithm has to be, for it to converge quadratically.

So far we have shown that the error after the iterative step is bounded by the square of the error before the step. However one must carefully observe that the functions after the step are in a slightly smaller domain and that there is some constant δ^{-1} that determines this loss of analyticity appearing in the bounds. A balance must be satisfied between taking a very small δ (where the constants blow up) and keeping the constants small (where we finally end up with no domain). This kind of problems are very usual in KAM theory and they are solved in the following way. The following sequence is considered for each step $n \geq 1$

$$\delta_n = \frac{1}{4}\delta_0 2^{-n}, \tag{2.79}$$

where δ_0 is the global analyticity loss. If the initial approximation is good enough it can be shown that the hypothesis of proposition 2.8 is satisfied at each step and the error at step n can be estimated as

$$\epsilon_n \leq C(\delta_0 2^{-n-1})^{-1}\epsilon_{n-1}^2 \leq \cdots \leq (C\delta_0^{-1}2^2\epsilon_0)^{2n}. \tag{2.80}$$

If $(C\delta_0^{-1}2^2\epsilon_0) < 1$ (which would appear as a hypothesis in the theorem stating the convergence of the method), ϵ_n decreases superexponentially, while δ_n decreases only exponentially. This is the key point in understanding why the hypothesis of our proposition is satisfied again after the Newton step, provided that the initial approximation is good enough.

Chapter 3

The reduced Hodgkin-Huxley model

The electrical activity in neurons can be understood in terms of four ions travelling through its membrane: sodium (Na^+), potassium (K^+), calcium (Ca^{2+}) and chloride (Cl^-). When the cell membrane is at rest, there is a high concentration of Na^+ and Cl^- in the extracellular medium, while the intracellular medium is rich in K^+ and negatively charged molecules.

The different concentrations of these ions inside and outside the cell membrane creates electrochemical gradients that are responsible for the electrical activity of the cells. The cell membrane does not allow ions to pass through it, but it has some ion-specific channels that can open and allow the flow of ions.

In 1952 Hodgkin and Huxley proposed a mathematical model [10] of the squid giant axon, which is one of the most important models in computational neuroscience. It must be said, however, that this model has the difficulty of involving four variables, and several simplifications that capture the essence of the dynamics of the neuron with only two variables have been proposed. The simplified model that we will consider in this thesis can be described with the following equations:

$$\begin{aligned} C\dot{V} &= I - g_{Na} m_\infty(V)(V - V_{Na}) - g_K n(V - V_K) - g_L(V - V_L), \\ \dot{n} &= n_\infty(V) - n, \end{aligned} \tag{3.1}$$

with

$$\begin{aligned} m_\infty(V) &= \frac{1}{1 + \exp(-(V - V_{max,m})/k_m)}, \\ n_\infty(V) &= \frac{1}{1 + \exp(-(V - V_{max,n})/k_n)}. \end{aligned} \tag{3.2}$$

This model is called the $I_{Na,p} + I_K$-model and is said to describe a fast persistent sodium current and a slower potassium current. In order to understand briefly the

meaning of these equations, consider the first equation in (3.1) written in the following way:

$$I = C\dot{V} + g_{Na}m_{\infty}(V)(V - V_{Na}) + g_K n(V - V_K) + g_L(V - V_L) \tag{3.3}$$

This equation is a simple application of Kirchoff's law to the neuron membrane. I represents the total current crossing the membrane, which may be due to synaptic current, axial current, tangential current along the membrane surface or current injected artificially with an electrode. On the right hand side of the last equality, we have $C\dot{V}$, which amounts for the capacitative current of the membrane, i.e., for the accumulation of ions at each side of the membrane when the membrane potential is changing. The third terms is called leakage current and is due to the fact that the membrane is never fully impermeable. Finally the two middle terms correspond to the flow of sodium and potassium ions with conductances $g_{Na}m_{\infty}(V)$ and $g_K n$, respectively. As we have said before the membrane has several ion channels that can open and close. g_{Na} and g_K can be interpreted as the maximum conductances, when all the channels are open, and $m_{\infty}(V)$ and n as the fraction of channels that are open at a given time. As we can see the fraction of open channels is voltage dependent in both cases, but time-dependent only in the second one.

As in [1] and [8] we use the following values for the parameters: $C_m = 1$, $g_{Na} = 20$, $V_{Na} = 60$, $g_K = 10$, $V_K = -90$, $g_L = 8$, $V_L = -80$, $V_{max,m} = -20$, $k_m = 15$, $V_{max,n} = -25$ and $k_n = 5$. The parameter I can be studied as a bifurcation parameter.

For $I = 0$ the system has no periodic orbits as shown in figure 3.1. A fixed point is localised at about $(V, n) \approx (-30, 0.4)$. The other two equilibria are localised at about $n = 0$ for $V \approx -55, -65$ and they are joined by two heteroclinic orbits. The neuron is expected to be in the stable equilibrium. If a small positive perturbation in the potential is applied, the neuron will go back to the stable equilibrium through a short path close to the short heteroclinic orbit. However, if this perturbation is such that the V-nullcline is crossed, then the neuron will go for a long loop close to the long heteroclinic orbit before returning to the starting point. In this case an action potential is elicited. Systems exhibiting this feature are called excitable media.

As the value of the parameter I is increased the stable and unstable equilibria connected by the two heteroclinic orbits become closer and closer until they fuse at $I = 4.51$, as shown in figure 3.2. At this point a bifurcation called Saddle-Node on an invariant circle (SNIC) happens.

For larger values of I, a limit cycle appears as shown in figure 3.3 and the membrane voltage exhibits a periodic behaviour with periodic spiking as seen in figure 3.4.

For the case $I = 10$ we show in figure 3.5 the isochrons defined in the last chapter computed for 40 evenly distributed phases. Let us recall that the concept of isochron allowed us to extend the idea of phase of oscillation to the basin of attraction of a

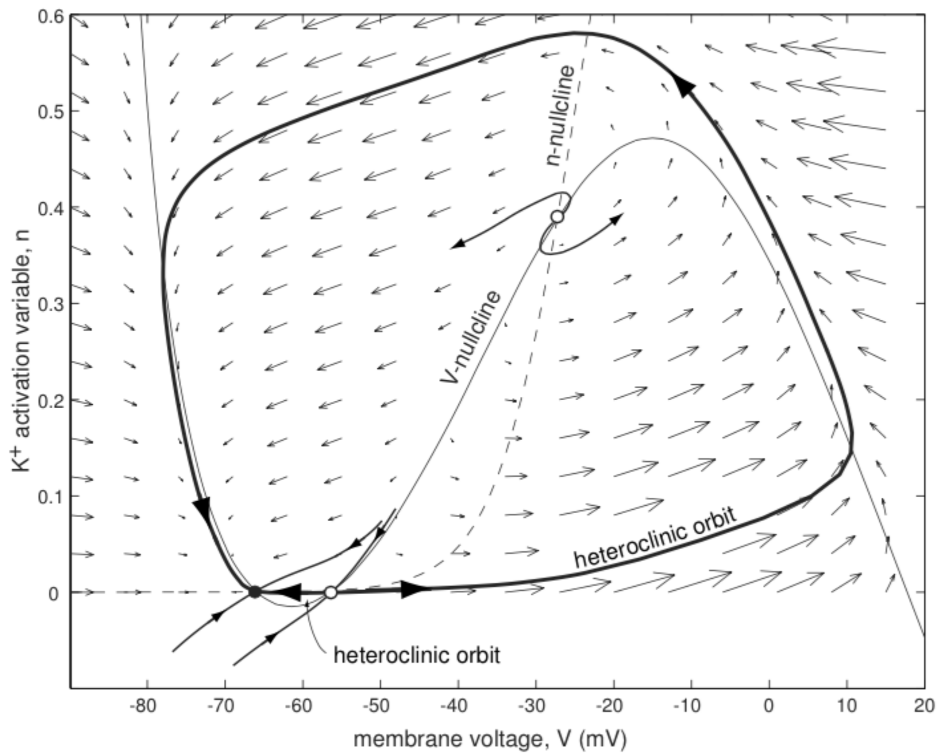

Figure 3.1: $I_{Na,p} + I_K$-model. No periodic orbits for $I = 0$. Source: [7]

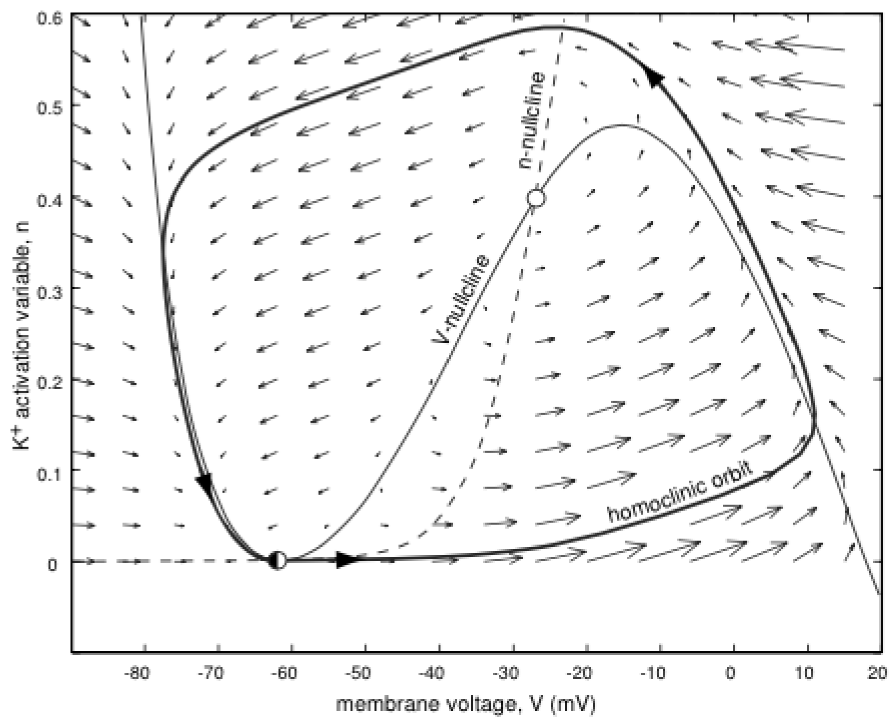

Figure 3.2: $I_{Na,p} + I_K$-model. SNIC bifurcation for $I = 4.51$. Source: [7]

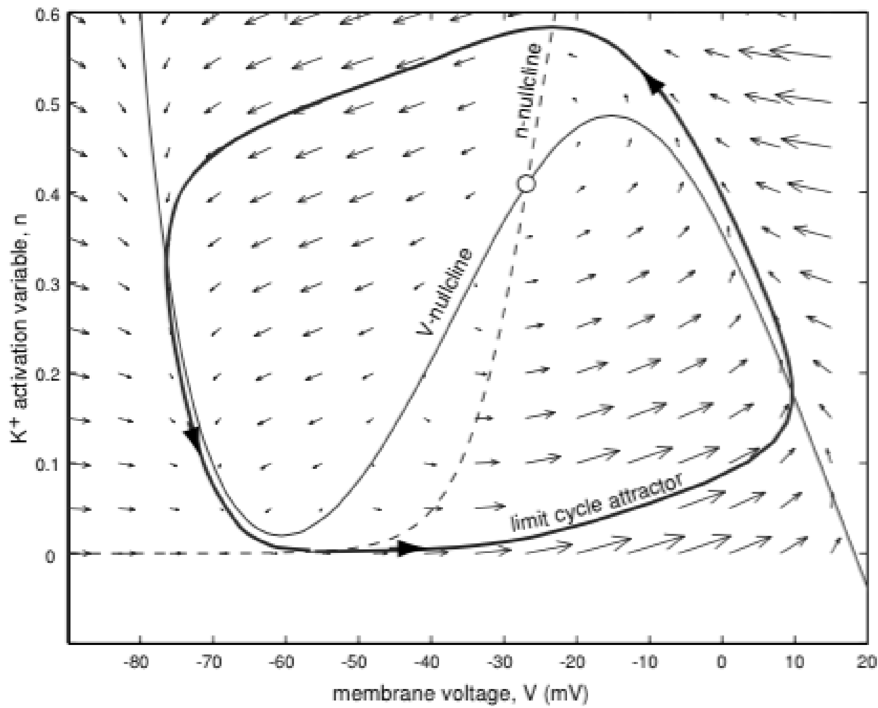

Figure 3.3: $I_{Na,p} + I_K$-model. Limit cycle attractor for $I = 10$. Source: [7]

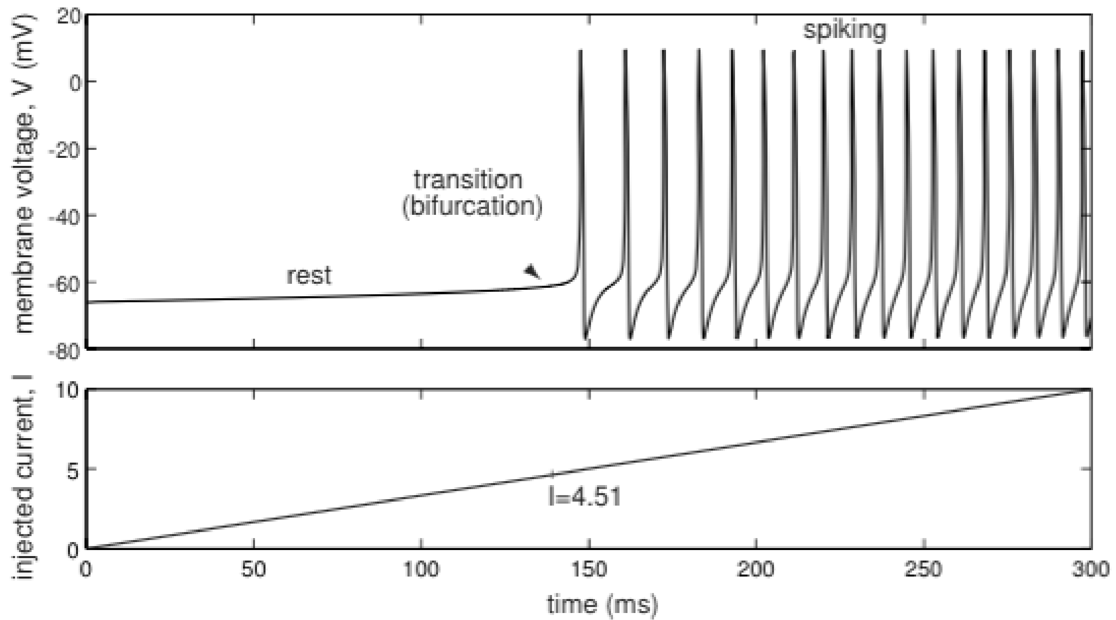

Figure 3.4: $I_{Na,p}+I_K$-model. Transition from resting state to periodic spiking. Source: [7]

limit cycle. Consider a neuron whose dynamics can be qualitatively described with figure 3.5. The neuron is periodically spiking and travelling along the limit cycle. As

we did before ($I = 0$) we can ask what happens if a small perturbation in the voltage occurs at a certain time. The neuron abandons the limit cycle and evolves in such a way that it will clearly go back to it, but it may have experienced a phase advance or delay. Isochrons allow us to determine quantitatively the phase shift experienced by the neuron. If a perturbation occurs when the neuron is at a phase θ_i and the neuron falls on an isochron with phase θ_f we know that when the neuron goes back to the limit cycle it will have suffered a phase shift:

$$\Delta\theta = \theta_f - \theta_i. \tag{3.4}$$

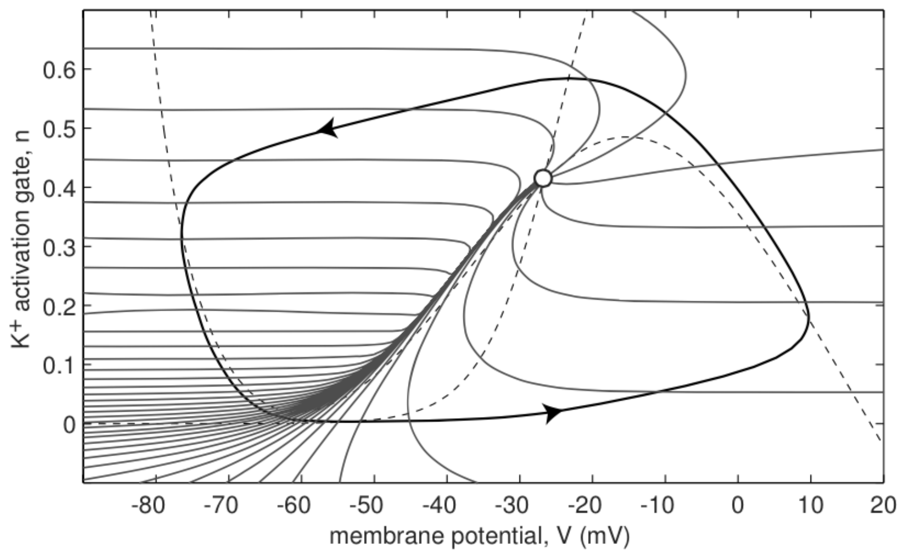

Figure 3.5: $I_{Na,p} + I_K$-model. Isochrons of the limit cycle attractor in figure 3.3. Source: [7]

Understanding the concept of isochron and being able to compute them plays a central role in computational neuroscience since it allows to understand how two coupled neurons behave depending on their intrinsic dynamics, how a net of synchronous neurons will react in front of a perturbation or how one desynchronised neuron can resynchronise again in front of the stimulus passed by the others.

Isochrons offer another kind information. Looking again at figure 3.5 we can see that isochrons accumulate at a certain part of the limit cycle, while they are spaciously distributed in other parts. As the isochrons are plotted for evenly distributed values of θ, which corresponds to evenly distributed values of t, we can easily see where the neuron goes faster and slower in the periodic orbit. It must be noted that the points where the trajectory slows down are those where the bifurcation occurred. In this case very strong slow-fast dynamics are observed.

Chapter 4

Numerical implementation

We select the model described in the last chapter (equations 3.1) and write a program in language C that is able to find a parametrisation of the periodic orbit $K(\theta)$ and find the vectors $N(\theta)$ through integration and that refines the initial solutions using the quasi-Newton method described in chapter 2. The source code is appended at the end of this undergraduate thesis, and the reader is invited to have a look at it while reading this chapter in order to follow the explanations given here.

4.1 The Taylor method for integrating ODEs

Our program needs to integrate the flux and the variational equations. For the integration of differential equations we use the Taylor method.

Consider the following Cauchy problem:

$$\begin{aligned} x'(t) &= f(t, x(t)), \\ x(t_0) &= x_0, \end{aligned} \tag{4.1}$$

The Taylor method is based on the possibility of performing a Taylor expansion of the function $x(t)$ around t_0, for $h \in \mathbb{R}$:

$$x(t_0 + h) = x(t_0) + x'(t_0)h + \frac{1}{2!}x''(t_0)h^2 + \dots \frac{1}{p!}x^{(p)}(t_0)h^p + \dots, \tag{4.2}$$

Taking a certain step h we can approximate $x(t_0+h)$ by the truncated Taylor series up to order p[1] and we call this approximation x_1:

$$x_1 = x(t_0) + x'(t_0)h + \frac{1}{2!}x''(t_0)h^2 + \dots \frac{1}{p!}x^{(p)}(t_0)h^p, \tag{4.3}$$

Now we can consider the new Cauchy problem with initial condition $x(t_1) = x_1$, where we use the notation $t_1 = t_0 + h$. The same procedure can be used in order to

[1] If $p = 1$ the method is know as the Euler method

obtain an approximation $x_2 \approx x(t_2)$, for $t_2 = t_1 + h$. The process can be iterated as long as needed, so that for each pair t_i, x_i a new x_{i+1} can be obtained for $t_{i+1} = t_i + h$.

Consider equation (4.3). At each step of the integration we need to compute the values of the derivatives $x^{(j)}(t_0)$ up to order p. The first derivative is given directly by the field: $x'(t_0) = f(t_0, x_0)$. For higher order derivatives we can differentiate the first equation in (4.1):

$$x''(t) = f_t(t, x(t)) + f_x(t, x(t))x'(t), \tag{4.4}$$

and so on. However, as we consider derivatives of higher order, the expressions that we get become more and more complicated and it becomes difficult to compute the derivatives of f and also to evaluate such long expressions numerically.

This difficulty however can be overcome using automatic differentiation, a recursive procedure which allows to compute easily the derivatives of a given function at a given point. This works only for a special class of functions, those that can be obtained by sum, product, quotient, and composition of elementary functions (polynomials, trigonometric functions, real powers, exponentials and logarithms).

Consider a function $a\colon \mathbb{R} \to \mathbb{R}$, which is assumed to be smooth, and which can be written as $a(t) = F(b(t), c(t))$, and assume furthermore that we know $b^{(j)}(t)$ and $c^{(j)}(t)$ up to order n at a given t_0. If F is one of the functions specified before the rules of automatic differentiation [4] allow us to write $a^{(j)}(t)$ in t_0 up to order n as a function of the already know derivatives of b and c in t_0. These rules can be applied recursively in $x'(t) = f(t, x(t))$ to obtain the derivatives of $x(t)$ in t_0 up to any order desired.

In our program we use the software package provided by Jorba and Zou [4], which, given some differential equations, is able to generate some routines of integration, which are in the header file `taylor.h` and have the structure:

```
int taylor_step_name(long double *ti, long double *x, int dir, int
    step_ctl, double log10abserr, double log10relerr, long double
  *endtime, long double *ht, int *order)
```

The arguments `ti` and `x` correspond to the initial conditions t_i and x_i and are overwritten with the new conditions t_{i+1} and x_{i+1} after one step of the integration.

This routines implement also optional stepsize and order control, which means that a different step h and different order p may be used for each step. By setting `step_ctl` to 1, the stepsize and order control are performed and the integrator tries to keep either the absolute or relative errors below the values given by the user. The decimal logarithm of the absolute and relative accuracy is passed to the function with `log10abserr` and `log10abserr`, respectively. At the end of the step, `ht` gives the time step used and `order` gives the degree used in the Taylor expansion. The flag `dir` can be either 1 for forward integration, or -1 for backward integration. Additionally, one can set a value for `endtime`, so that the size of the last step is adapted in order to fit

the required end time. The output of the function is 0, unless `ti=endtime`. In this case the function returns 1.

On the other hand, it is possible to set `step_ctl` to 0. In this case no stepsize or order control are performed and the stepsize and order to be used are defined by the user through `ht` and `order`, while the other flags are ignored.

4.2 Discrete Fourier transform

As we will be dealing with periodic functions and we will need to solve some equations involving their Fourier coefficients it essential to be able to find them. Consider $f\colon \mathbb{T}_\rho \to \mathbb{R}$:

$$f(\theta) = \sum_{k=-\infty}^{\infty} \tilde{f}_k e^{2\pi i k\theta}. \tag{4.5}$$

As we have seen in proposition 2.7 the coefficients are expected to decay exponentially with $|k|$ and therefore the Fourier series can be truncated at some point:

$$f(\theta) \approx \sum_{k=-N/2}^{N/2-1} \tilde{f}_k e^{2\pi i k\theta}. \tag{4.6}$$

With the objective of finding such an approximation we will make use of the Discrete Fourier Transform (DFT). We discretise $\mathbb{T}_\rho$ by taking N values $\{\theta_j = j/N\}_{j=0}^{N-1}$ equally spaced on the interval, and we evaluate the function f at these points in order to obtain a set of values:

$$\{f_j = f(\theta_j)\}_{j=0}^{N-1} \tag{4.7}$$

The DFT transforms this set of N values into an N periodic set of complex values

$$\tilde{f}_k = \sum_{j=0}^{N-1} f_j e^{-2\pi i k\theta_j}, \qquad k \in \mathbb{Z}, \tag{4.8}$$

that satisfy $\tilde{f}_j = \tilde{f}_{j(\mathrm{mod} N)}$. Those indexed by $0, \dots, N$ are the coefficients of the interpolating trigonometric polynomial up to a factor $1/N$:

$$P(\theta) = \sum_{j=0}^{N-1} \frac{1}{N} \tilde{f}_j e^{2\pi i j\theta}, \tag{4.9}$$

that satisfies $P(\theta_j) = f_j$.

The implementation of the Discrete Fourier Transform will be done using the Fast Fourier Transform algorithms found in the library FFTW, included in the code with the header `fftw3.h`. These algorithms are able to perform the transform in $O(NlogN)$ operations (specially if N is a power of 2).

In our code one can see that there is a bunch of functions related to Fourier transforms. The technical requirements of the routines in the FFTW library have been hidden under the functions `void createfourier()` and `void destroyfourier()` for the sake of clarity, and they can be thought of as allocating and freeing memory when working with pointers. The functions in `fftw3.h` perform the DFT on an array of numbers. However, as we are working with arrays of two-dimensional vectors we have also created our own functions

```
void forwardfourier(long double complex **IN, long double complex
    **OUT);
void backwardfourier(long double complex **IN, long double complex
    **OUT);
```

that work on arrays of vectors.

The function `forwardfourier` reads an array `IN[2][Ngrid]`, performs the DFT on each array of components and divides the result by `Ngrid` before writing it on `OUT[2][Ngrid]`, so that we obtain directly the coefficients of the interpolating trigonometric polynomial. The function `backwardfourier` performs a backward DFT, which simply consists in evaluating the interpolating polynomial at the values $\theta_j = j/N$, for $j = 1, \dots N$, so that no further rescaling is needed.

As already said when we do the forward Fourier transform we obtain the coefficients of the interpolating polynomial (4.9), that correspond, up to a shift in the indexes, to the coefficients of the truncated Fourier series (4.6). A function `int ind(int i)` has been also created, which reads an integer `i=-N/2,...,N/2-1` and shifts it to the range `0,...,N-1`. When working with vectors in the Fourier space this function will be used.

The function `void derivatefourier(long double complex **K_, long double complex **DK_)` is used for differentiating truncated Fourier series (4.6):

$$f(\theta) = \sum_{k=-N/2}^{N/2-1} 2\pi i k \tilde{f}_k e^{2\pi i k \theta}. \tag{4.10}$$

The function `int ind(int i)` turns out to be very useful in this case.

Still an important observation is to be made. As our functions are real-valued, the Fourier coefficient for a certain k is the complex conjugate of that for $-k$. However, when truncating the series and working with even N, we can see that the so-called Nyquist term, $\tilde{f}_{-N/2}$, is not compensated by a complex conjugate, since $\tilde{f}_{N/2}$ is taken to be 0. This means that we are no longer working with real objects. This could be compensated by artificially adding a term $\tilde{f}_{N/2}$ to compensate for this effect. This is something we don't do, because we don't need to evaluate the Fourier series out of the grid. However, this problem naturally arises when differentiating. In this case we set the Nyquist term of the derivative to be zero as suggested in [9].

4.3 The program

In the appendix, the source code is commented. However, some features are still worth mentioning.

As already seen, our program is written using precision `long double`. At the beginning of the code one can see that the values of the parameters are specified. The values chosen are those that already appeared in the previous chapter, and we choose the bifurcation paramater I to be 10 mV. The absolute and relative tolerances that we used for the integrator and for other purposes are 10^{-16}.

Several functions are defined: `void matrixvector2(long double M[2][2], long double *v)` and `void invmatrixvector2(long double M[2][2], long double *v)`, which apply the matrix `M` (and its inverse, respectively) to the vector `v`.

For the computation of the periodic orbit we first choose a Pincaré section Σ at $V = -40$ (see figure 3.5) and give initial conditions `x[0]=-40; x[1]= 0.2`. This is passed to the function `long double poincaremap(long double *x)`, which reads `x[2]`, applies the Poincaré map $p\colon \Sigma \to \Sigma$ to this point and returns the so-called return time. The internal structure of the function `poincaremap` is divided in two parts. In the first one it lets the point evolve until the Poincaré section is overpassed. Let x_0 be the last point before the Pincaré section is reached and h the step of the last integration, then we know that

$$\begin{aligned} &\Pi_1(\varphi(0,x_0)) = \Pi_1(x_0) < -40, \\ &\Pi_1(\varphi(h,x_0)) > -40, \end{aligned} \tag{4.11}$$

where $\Pi_1 : \mathbb{R}^2 \to \mathbb{R}$ is the projection on the x axis, which in this case corresponds to the variable V. Then we are interested in solving the equation

$$\Pi_1(\varphi(\tau,x_0)) + 40 = 0, \tag{4.12}$$

for the variable τ. This is done in the second part of the function `poincaremap` by means of a Newton method.

The function `poincaremap` is applied recursively until the point in the periodic orbit is found and the return value of the function gives the period of the oscillation, which is stored in `T`.

We start with `Ngrid=64` and store in `K[][0]`[2] the point of the orbit obtained in the Poincaré section. Using `taylor_step_inapik` we let this point evolve in time in order to obtain `Ngrid` samples on the periodic orbit evenly distributed in time, i.e, $K(\theta_j)$, for $\theta_j = j/N$. Then the Fourier transform is performed and the coefficients are stored

[2]In our program `K[i][j]` means the i-th component of the j-th sample in (4.7), i.e., `j` goes from `0` to `Ngrid`, and for each `j`, we can have `K[0][j]` and `K[1][j]`.

in `K_[2][Ngrid]`. The tail of the Fourier series is checked in order to determine if a larger number of samples is needed. In that case we choose `Ngrid*=2` and start again.

With all this information the error function for the functional equation in K and ω is evaluated and the results are stored in `EK`.

Then we proceed to compute the stable vector N. The first step is to find $N(0)$. In order to do this we integrate the variational equations backwards for a whole period so that we obtain:

$$D\varphi(-T, K(0)). \tag{4.13}$$

This is obtained using another integration routine: `taylor_step_inapikvariational(&t, vareqn, -1, 1, log10tol, log10rtol, &tau, &h, &order)`, where the initial conditions are `vareqn[0]=K[0][0]` and `vareqn[1]=K[1][0]` for the point in the orbit $K(0)$ and `vareqn[3]=1.`, `vareqn[4]=0.`, `vareqn[5]=0.` and `vareqn[6]=1.` for the values of the matrix $D\varphi$ at $K(0)$. As the integration is backwards and the orbit is unstable, we integrate at time intervals of `-T/Ngrid` and force the point to stay in the orbit by setting manually `vareqn[0]=K[0][Ngrid-j]` and `vareqn[1]=K[1][Ngrid-j]`, for the `j` which corresponds at every step.

The matrix $D\varphi(-T, K(0))$ is stored in `M` and the eigenvalues and eigenvectors are obtained. From these we can find $N(0)$ and λ. And by propagating this vector (backwards again) as discussed in proposition 2.3 we can obtain $N(\theta_j)$ for $\theta_j = j/N$. The Fourier transform is performed and the error function is evaluated.

Finally, with this initial approximations in hand, the Newton method described in chapter 2 is applied.

Chapter 5

Results

5.1 The reduced Hodgkin-Huxley model

Our program is applied to the reduced Hudgkin-Huxley model described in chapter 3.1. We start with the case $I = 10$, in which 8192 samples are needed for the Fourier series. We obtain a period $T = 7.0735$ and $\lambda = -3.9110$. With the initial approximation found by integration the error in the functional equation for K is of the order of 10^{-11}, but for N it is of the order of 10^{-7}. In figure 5.1 we plot the limit cycle obtained and the the linear approximation of 64 isochrons. The slow-fast dynamics is clearly observable.

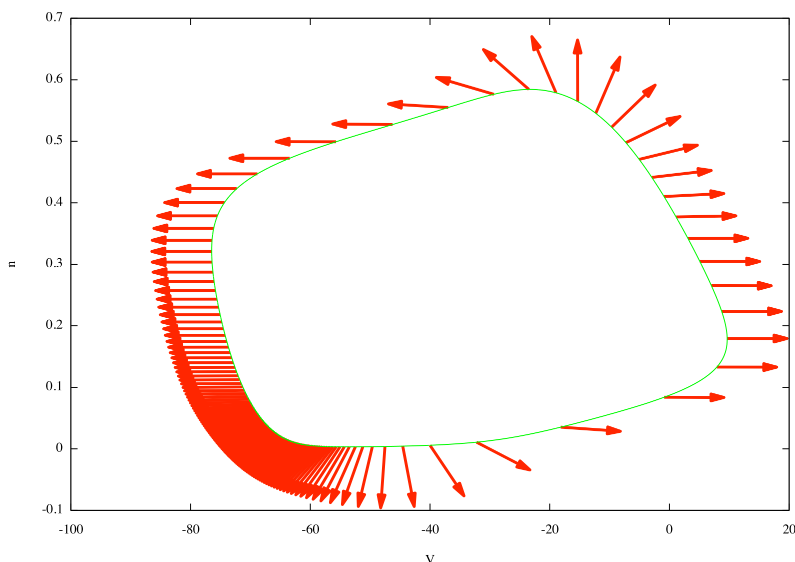

Figure 5.1: $I_{Na,p} + I_K$-model. $I = 10$. Limit cycle and 64 stable vectors. The stable vectors have been rescaled since they have extremely different length scales.

We expected the Newton method designed in chapter 2 to improve the solutions found, but to our surprise the method failed to find better solutions. Instead after ten

iterations the error of both K and N blows up to orders of 10^3, which is clearly unacceptable. We understand this anomalous behaviour in terms of numerical problems. On one hand when the differential of the flux is integrated for one period and the two eigenvalues are found, we see that one of them is 1, while the other is of the order of 10^{-12}. Compare this difference with the 10^{-16} tolerance that we chose for our program. We are in front of a very attractive limit cycle, which simplifies the calculation of $K(\theta)$, but makes it really difficult to find $N(\theta)$ satisfactorily. Yet another problem is to be considered. In figure 5.2 we plot for each θ the angle between the limit cycle and the isochron crossing it at $K(\theta)$, i.e., the angle between $N(\theta)$ and $K'(\theta)$.

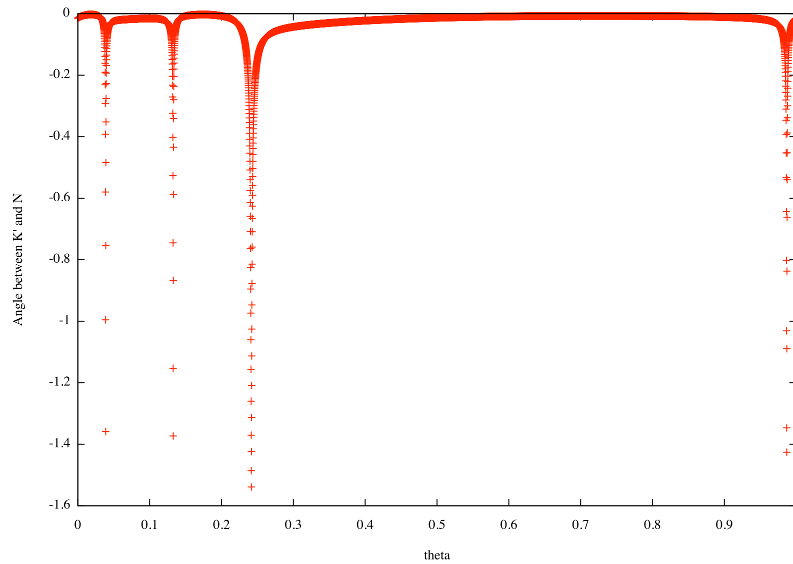

Figure 5.2: $I_{Na,p} + I_K$-model. $I = 10$. Angle between the orbit and the vector N for each θ. Note that $\Pi_1(K(\theta = 0)) = -40$.

It is clear that, apart from being far from continuous, even if so many samples have been considered, the angle is very close to 0 for most of the values of θ. Let us recall at this point that for the Newton method we need to transform our equations to the frame $P = \left(K' \middle| N\right)$, and it is clear that K' and N must satisfy the non-degeneracy condition $\langle K', N \rangle \neq 0$. The further they are from this condition the better our initial approximations have to be for our method to converge. The numerical instability is intrinsic to the model we have selected, which presents stiff dynamics, but we hope that it could be overcome using extended precision arithmetic.

We have also run our program for $I = 100$, which is a bit further from the SNIC bifurcation. In this case we obtain $T = 2.7405$ and $\lambda = -5.4031$. The initial approximation for K and N is also found with an error similar to that of $I = 10$, and the Newton method also fails to improve the solutions, even though the error grows a bit slowlier. A plot of the periodic orbit is shown in figure 5.3. Observe how in this case

the vectors N are more evenly distributed. This is due to the fact that, as already mentioned, the SNIC bifurcation is much further than at $I = 10$.

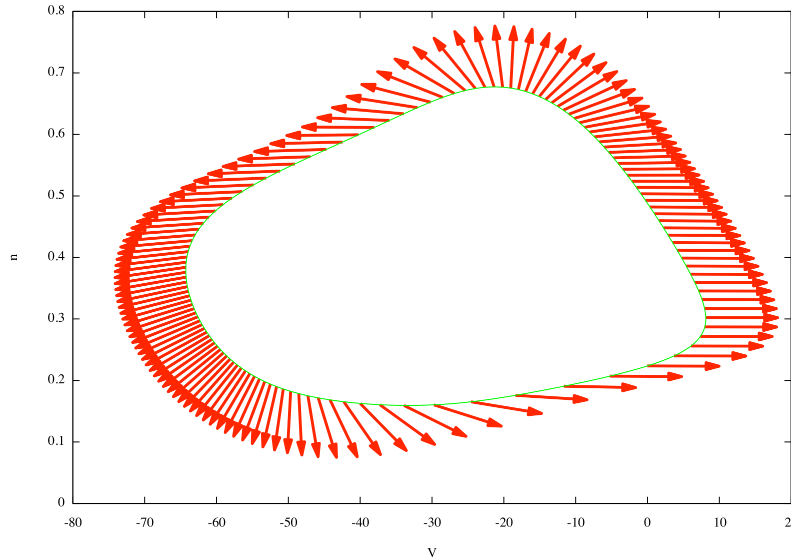

Figure 5.3: $I_{Na,p} + I_K$-model. $I = 100$. Limit cycle and 64 stable vectors. The stable vectors have been rescaled since they have extremely different length scales.

We have still considered the case $I = 190$, close to a Hopf bifurcation that occurs at around $I = 214$. In this case $T = 1.3055$ and $\lambda = -0.4639$, and we obtain an error of about 10^{-12} in the initial approximation of K and about 10^{-11} for N. Note than in this case the stable eigenvalue of the $D\varphi(T, K(0))$ is $e^{\lambda T} = 0.5457$, which is not even one order of magnitude below 0. In this case the Newton method can be applied as many times as desired and the error does not grow, nor does it decrease. In figure 5.4 the linear approximation of 64 isochrons is shown.

5.2 The Rayleigh oscillator

We have seen in the previous section that our program can effectively compute an initial approximation for K, N, T and λ for several values of I in the $I_{Na,p}+I_K$-model, but the Newton method that we have designed did not work properly. Therefore, we consider in this section a much simpler model in order to show the convergence of our method. The Rayleigh oscillator has equations:

$$\begin{aligned} \dot{x} &= -y + \mu(x - x^3), \\ \dot{y} &= x. \end{aligned} \tag{5.1}$$

It is known to have an unstable focus at $(0, 0)$. For $\mu = 1$ it also has a stable limit cycle, for which we have found $T = 6.6632$ and $\lambda = -1.059$. In this case we have

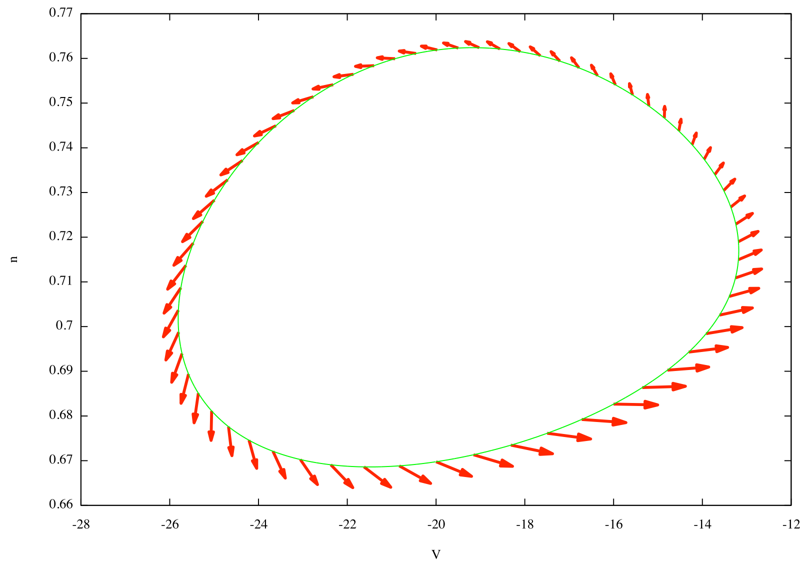

Figure 5.4: $I_{Na,p} + I_K$-model. $I = 190$. Limit cycle and 64 stable vectors.

needed $N = 2048$, and initial approximations of K and N are found with errors of 10^{-14} and 10^{-13}, respectively. The Newton method is able to reduce these errors to 10^{-16} in both cases after only two steps. A plot of the periodic orbit with 32 stable vectors is shown in figure 5.5

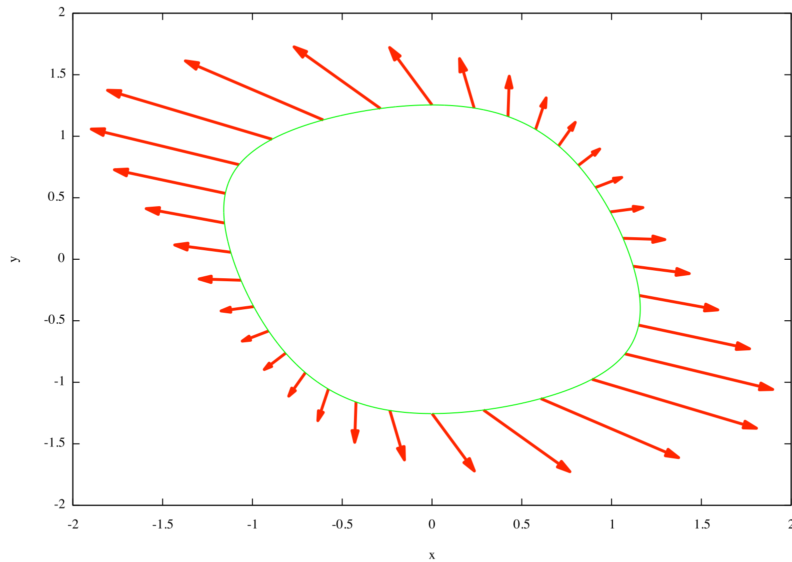

Figure 5.5: Rayleigh oscillator. $\mu = 1$. Limit cycle and 32 stable vectors.

Just for comparison with the $I_{Na,p} + I_K$-model and figure 5.2 we show in figure 5.6 the angle between K' and N in the Rayleigh oscillator. This is a much better conditioned problem and it is easy to understand why our method performs efficiently here.

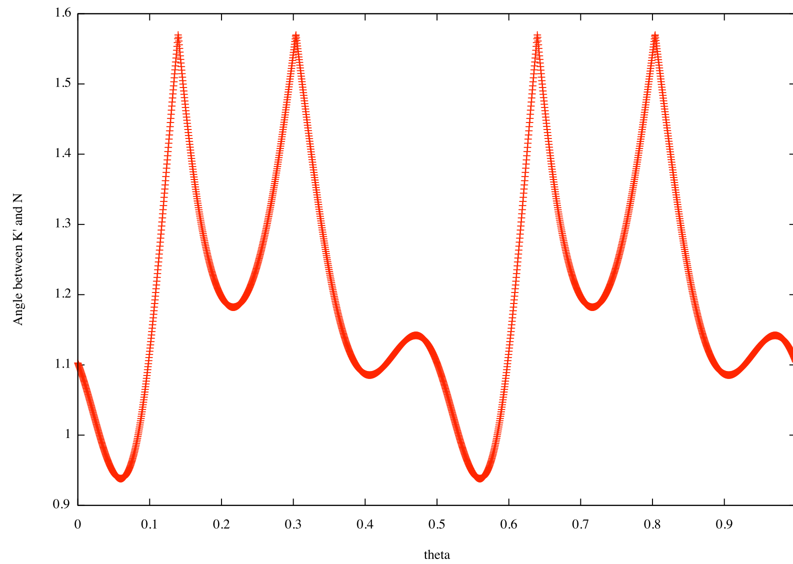

Figure 5.6: Rayleigh oscillator. $\mu = 1$. Angle between the orbit and the vector N for each θ. In this case $\Pi_1(K(\theta = 0)) = 0$

The Newton method that we designed allows also to do continuation with respect to parameters. For the case $\mu = 1.2$ we can take the values of K, T, N and λ that we obtained for $\mu = 1$ as an initial approximation and use the Newton method to improve them. No integration is needed in this case. For $\mu = 1.2$ and using this procedure the initial error is 10^{-2}, which is rapidly reduced to 10^{-15} after 20 iterations of the method. We obtain $T = 6.8212$ and $\lambda = -1.2997$. We plot the limit cycle and 32 stable vectors in figure 5.7. The same is done for $\mu = 1.6$ with an initial error of almost 1, which after 20 steps is reduced to 10^{-13}. We obtain $T = 7.1966$ and $\lambda = -1.8180$. The limit cycle and 32 stable vectors are shown in figure 5.8.

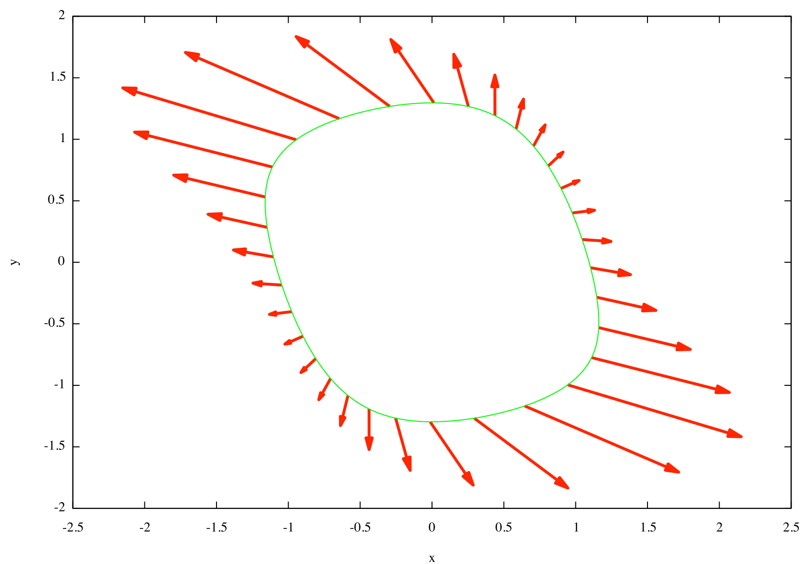

Figure 5.7: Rayleigh oscillator. $\mu = 1.2$. Limit cycle and 32 stable vectors obtained by improving the solutions found for $\mu = 1$.

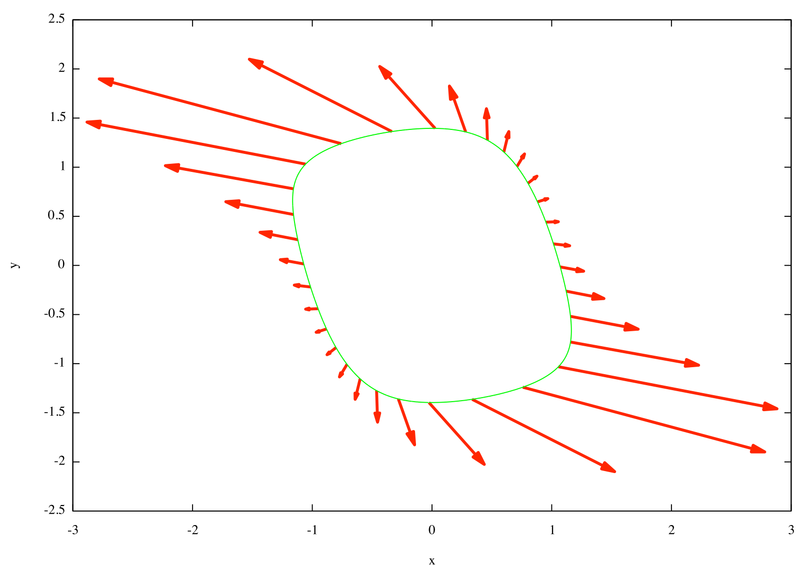

Figure 5.8: Rayleigh oscillator. $\mu = 1.6$. Limit cycle and 32 stable vectors obtained by improving the solutions found for $\mu = 1$.

Chapter 6

Conclusions

Some new techniques and concepts have been learnt throughout the development of this thesis. With an example we have seen how a biological system can be satisfactorily modelled with mathematical tools and how this can help to understand many properties of such a system.

Although the algorithm designed in chapter 2 did not work as expected for the model that we selected, we have also learnt many numerical techniques for programming that turn out to be very useful in front of real problems, that cannot usually be described by means of an analytical expression.

At the same time, we have also seen the close relationship between the designed algorithms and the numerical implementation, so that we can understand under which circumstances a numerical algorithm will fail.

Three topics remain open for the author as further work. The first one is to get a better understanding of the KAM theory and its implications. The second one is to think how to improve the designed algorithms in order for them to work in the first model considered here. The third one is to read more about how to effectively apply the concept of isochron to coupled oscillators and the implications that this has in neural systems.

Bibliography

[1] G. Huguet and R. de la Llave, *Computation of limit cycles and their isochrons; fast algorithms and their convergence*, SIAM J. Appl. Dyn. Syst., 12 (2004), pp. 1763-1802.

[2] R. de la Llave, *A tutorial on KAM theory*, in Smooth Ergodic Theory and Its Applications (Seattle, WA, 1999), AMS, Providence, RI, 2001, pp. 175-192.

[3] L. V. Ahlfors, *Complex Analysis: An Introduction to the Theory of Analytic Functions of One Complex Variable*, 3rd ed., International Series in Pure and Applied Mathematics, McGraw-Hill, New York, 1978.

[4] À. Jorba and M. Zou, *A software package for the numerical integration of ODEs by means of high-order Taylor methods*, Experiment. Math., 14 (2005), pp. 99-117.

[5] A. T. Winfree, *Patterns of phase compromise in biological cycles*, J. Math. Biol., 1 (1974/75), pp. 73-95.

[6] J. Guckenheimer, *Isochrons and phaseless sets*, J. Math. Biol., 1 (1974/75), pp. 259-273.

[7] E. M. Izhikevich, *Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting*, Comput. Neurosci., MIT Press, Cambridge, MA, 2007.

[8] A. Guillamon and G. Huguet, *A computational and geometric approach to phase resetting curves and surfaces*, SIAM J. Appl. Dyn. Syst., 8 (2009), pp. 1005-1042.

[9] S. G. Johnson, *Notes on FFT-based differentiation*, MIT Appl. Math., (2011)

[10] A. L. Hodgkin and A. F. Huxley, *A quantitative description of membrane current and its applications to conduction and excitation in nerve*, J. Physiol., 117 (1952), pp. 500-544

Appendix A

The program

```c
1  #include "taylor.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <math.h>
5  #include <complex.h>
6  // If <complex.h> comes before <fftw3.h>, then fftwl_complex is defined to be the
       native complex type and you can manipulate it with ordinary arithmetic
7  #include <fftw3.h>
8
9  long double current=10.;
10
11 long double C=1.0, gNa=20., VNa=60., gK=10., VK=-90., gL=8., VL=-80., Vmaxm=-20., km
       =15., Vmaxn=-25., kn=5.;
12
13 double tol=1e-16;
14 double log10tol;
15 double rtol=1e-16;
16 double log10rtol;
17
18 void field(long double *x, long double *f);
19 void dfield(long double *x, long double df[2][2]);
20 void matrixvector2(long double M[2][2], long double *v);
21 void invmatrixvector2(long double M[2][2], long double *v);
22
23 void checkpoint();
24 void space(int n);
25 long double poincaremap(long double *x);
26
27 // Fourier things
28 int Ngrid=64;
29 fftwl_complex *in, *out;
30 fftwl_plan planback, planfor;
31 void createfourier();
32 void destroyfourier();
33 void forwardfourier(long double complex **IN, long double complex **OUT);
34 void backwardfourier(long double complex **IN, long double complex **OUT);
35 void derivatefourier(long double complex **K_, long double complex **DK_);
36 int ind(int i);
37
38 void qNewton_K_step(long double *lambda, long double *omega, long double complex **K,
        long double complex **N, long double complex **DK, long double complex **EK);
```

```
39  void qNewton_N_step(long double *lambda, long double *omega, long double complex **K,
        long double complex **N, long double complex **DK, long double complex **DN, long
        double complex **EN);
40
41  int main(void){
42    log10tol=log10(tol);
43    log10rtol=log10(rtol);
44
45    space(2);
46    printf("------------------------------------------------------------------------------\n");
47    printf("------------------------------------------------------------------------------\n");
48    printf("------------------------ I N A P I K  M O D E L ------------------------\n");
49    printf("------------------------------------------------------------------------------\n");
50    printf("------------------------------------------------------------------------------\n");
51    space(2);
52
53
54    printf("Looking for K and T\n");
55    printf("------------------------------\n");
56    space(1);
57
58    long double t=0, h=0;
59    long double x[2];
60    long double aux;
61    int order=20;
62    int finished;
63    // Initial values are given on the Poincare section for V=-40
64    x[0]=-40;
65    x[1]=0.2;
66
67    // The Poincare map is applied until the fixed point is found up to a tolerance tol
68    do{
69      aux=x[1];
70      t=poincaremap(x);
71    }while(fabs(x[1]-aux)>tol);
72    long double T=t;
73    long double omega=1./T;
74    long double orbp[2];
75    orbp[0]=x[0];
76    orbp[1]=x[1];
77
78    printf("T=%20.12Lf\n", T);
79    space(1);
80
81    // Find Fourier coefficients using fftwl3
82    t=0;
83    long double complex *K[2], *K_[2];
84    long double tail[2];
85    tail[0]=-1.;
86
87    // We start with Ngrid=64. After doing the FFT, the tail is checked in order to
         determine if a larger number of samples is needed.
88    do{
89      if(tail[0]!=-1.){
90        Ngrid*=2;
91        destroyfourier();
92        free(K[0]);
93        free(K[1]);
```

```
94          free(K_[0]);
95          free(K_[1]);
96        }
97        createfourier();
98        // In K we will store the samples values, in K_ the Fourier coefficients of K(\
              theta)
99       K[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
100      K[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
101       K_[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
102       K_[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
103
104       // We integrate the flux at intervals of T/Ngrid in order to obtain the values of
              K(\theta)
105       t=0;
106       long double tau=0;
107       for(int i=0; i<Ngrid; i++){
108         K[0][i]=x[0];
109         K[1][i]=x[1];
110         tau+=T/(long double)Ngrid;
111         do{
112           finished=taylor_step_inapik(&t, x, 1, 1, log10tol, log10rtol, &tau, &h, &
                  order);
113         }while(finished!=1);
114       }
115
116       forwardfourier(K,K_);
117
118       // The norm of the tail is found. Use of the function ind for a reindexing is
              made since we are working Fourier coefficients
119       for(int i=0; i<2; i++){
120         tail[i]=cabs(K_[i][ind(-Ngrid/2)]);
121         for(int j=Ngrid/4; j<Ngrid/2; j++){
122             tail[i]+=cabs(K_[i][ind(j)])+cabs(K_[i][ind(-j)]);
123         }
124       }
125       // An average of the tail is computed
126       tail[0]=tail[0]/(long double)(Ngrid/2);
127       tail[1]=tail[1]/(long double)(Ngrid/2);
128     }while(tail[0]>tol || tail[1]>tol);
129     printf("Ngrid=%d\n", Ngrid);
130
131     // We store in DK_ the coefficients of the derivative of K(\theta) and in DK the
            values of this derivative evaluated at each \theta
132     long double complex *DK[2], *DK_[2];
133    DK[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
134    DK[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
135     DK_[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
136     DK_[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
137
138     derivatefourier(K_,DK_);
139     backwardfourier(DK_,DK);
140
141     // Find the error function evaluating the functional equation
142     long double complex *EK[2];
143    EK[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
144    EK[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
145
146     long double norm;
```

```
147      long double maxEK;
148      maxEK=0;
149
150      for(int j=0; j<Ngrid; j++){
151         x[0]=K[0][j];
152         x[1]=K[1][j];
153         long double f[2];
154         field(x,f);
155         EK[0][j]=f[0]-DK[0][j]*omega;
156         EK[1][j]=f[1]-DK[1][j]*omega;
157         norm=hypotl(EK[0][j],EK[1][j]);
158         if(norm>maxEK){
159              maxEK=norm;
160         }
161      }
162         space(1);
163         printf("maxEK=%.2Le\n", maxEK);
164
165      space(2);
166
167
168      printf("Looking for N and lambda\n");
169      printf("---------------------------\n");
170      space(1);
171
172      long double complex *N[2], *N_[2];
173      N[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
174      N[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
175      N_[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
176      N_[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
177
178      long double vareqn[6];
179      long double lambda=0;
180      long double taux;
181      long double vaux[2];
182      long double V[2];
183      long double M[2][2];
184
185      // Integrate backwards variational equations together with the flux for one period.
186      vareqn[0]=K[0][0];
187      vareqn[1]=K[1][0];
188      vareqn[2]=1.;
189      vareqn[3]=0.;
190      vareqn[4]=0.;
191      vareqn[5]=1.;
192      t=0;
193      tau=0;
194         for (int j=1; j<Ngrid+1; j++){
195              tau-=T/(long double)Ngrid;
196              do{
197                   finished=taylor_step_inapikvariational(&t, vareqn, -1, 1, log10tol,
                          log10rtol, &tau, &h, &order);
198              }while(finished!=1);
199              // The integration is performed at intervals of T/Ngrid and after each step
                   we force the point to stay in the orbit since we are integrating backwards
                    and the problem is very unstable
200              vareqn[0]=K[0][Ngrid-j];
201              vareqn[1]=K[1][Ngrid-j];
```

```
202         }
203
204       M[0][0]=vareqn[2];
205       M[0][1]=vareqn[3];
206       M[1][0]=vareqn[4];
207       M[1][1]=vareqn[5];
208
209       // Find the eigenvalues and eigenvectors of the matrix M, which is the differential
              of the flux after one cycle has been completed.
210
211       long double det=M[0][0]*M[1][1]-M[1][0]*M[0][1];
212       long double tr=M[0][0]+M[1][1];
213
214       long double eigval1=tr/2+sqrt(tr*tr/4-det);
215       long double eigval2=tr/2-sqrt(tr*tr/4-det);
216
217       if(eigval1<eigval2){
218         aux=eigval1;
219         eigval1=eigval2;
220         eigval2=aux;
221       }
222
223       // eigval2 must be 1 and eigval1 must be e^{lambda*T}
224       printf("eigval1=%20.12Lf\n", eigval1);
225       printf("eigval2=%20.12Lf\n", eigval2);
226
227       lambda=-log(eigval1)*omega;
228
229       vaux[0]=M[1][0];
230       vaux[1]=eigval1-M[0][0];
231       norm=sqrt(vaux[0]*vaux[0]+vaux[1]*vaux[1]);
232       vaux[0]/=norm;
233       vaux[1]/=norm;
234
235       printf("%20.12Lf    %20.12Lf\n", vaux[0], vaux[1]);
236
237       // We store the stable vector in N(theta=0)
238       N[0][0]=vaux[0];
239       N[1][0]=vaux[1];
240
241       printf("lambda=%20.12Lf\n", lambda);
242       space(1);
243
244
245       // Propagate N(0) backwards under the differential of the flux to find N(theta)
246       vareqn[0]=K[0][0];
247       vareqn[1]=K[1][0];
248       vareqn[2]=1.;
249       vareqn[3]=0.;
250       vareqn[4]=0.;
251       vareqn[5]=1.;
252       t=0;
253       tau=0;
254       long double maxnorm=1;
255       for (int j=1; j<Ngrid; j++){
256         tau-=T/(long double)Ngrid;
257         do{
258           finished=taylor_step_inapikvariational(&t, vareqn, -1, 1, log10tol, log10rtol,
```

```
            &tau, &h, &order);
259        }while(finished!=1);
260        vareqn[0]=K[0][Ngrid-j];
261        vareqn[1]=K[1][Ngrid-j];
262
263       M[0][0]= vareqn[2];
264       M[0][1]= vareqn[3];
265       M[1][0]= vareqn[4];
266       M[1][1]= vareqn[5];
267
268        vaux[0]=N[0][0];
269        vaux[1]=N[1][0];
270        matrixvector2(M,vaux);
271
272        // We apply the differential of the flux to N(0) to obtain N(theta), since we
             know that isochrons go to isochrons under the flux
273       N[0][Ngrid-j]=vaux[0]*exp(-lambda*tau);
274       N[1][Ngrid-j]=vaux[1]*exp(-lambda*tau);
275
276          norm=hypotl(N[0][Ngrid-j],N[1][Ngrid-j]);
277          if(norm>maxnorm){
278               maxnorm=norm;
279          }
280
281     }
282
283     // Check that with one more step we obtain N(0) again
284     tau-=T/(long double)Ngrid;
285     do{
286        finished=taylor_step_inapikvariational(&t, vareqn, -1, 1, log10tol, log10rtol, &
             tau, &h, &order);
287     }while(finished!=1);
288
289     M[0][0]= vareqn[2];
290     M[0][1]= vareqn[3];
291     M[1][0]= vareqn[4];
292     M[1][1]= vareqn[5];
293
294     vaux[0]=N[0][0];
295     vaux[1]=N[1][0];
296     matrixvector2(M,vaux);
297     vaux[0]*=exp(-lambda*tau);
298     vaux[1]*=exp(-lambda*tau);
299     printf("%20.12Lf    %20.12Lf\n", creall(N[0][0]), creall(N[1][0]));
300     printf("%20.12Lf    %20.12Lf\n", vaux[0], vaux[1]);
301
302
303     // Normalise all the vectors N by the maximum norm
304     for(int j=0; j<Ngrid; j++){
305           N[0][j]/=maxnorm;
306           N[1][j]/=maxnorm;
307     }
308
309
310     forwardfourier(N,N_);
311
312     // As before we store in DN_ the Fourier coefficients of the derivative of N and in
           DN the values of this derivative at each theta
```

```
313   long double complex *DN[2], *DN_[2];
314   DN[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
315   DN[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
316   DN_[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
317   DN_[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
318
319   derivatefourier(N_,DN_);
320   backwardfourier(DN_,DN);
321
322   // Find the error function evaluating the functional equation for N and lambda
323   long double complex *EN[2];
324   EN[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
325   EN[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
326
327   long double maxEN;
328   maxEN=0;
329
330   for(int j=0; j<Ngrid; j++){
331     long double df[2][2];
332     x[0]=K[0][j];
333     x[1]=K[1][j];
334     V[0]=N[0][j];
335     V[1]=N[1][j];
336     dfield(x,df);
337     matrixvector2(df,V);
338     EN[0][j]=V[0]-lambda*N[0][j]-DN[0][j]*omega;
339     EN[1][j]=V[1]-lambda*N[1][j]-DN[1][j]*omega;
340     norm=hypotl(EN[0][j],EN[1][j]);
341     if(norm>maxEN){
342         maxEN=norm;
343     }
344
345   }
346   space(2);
347   printf("maxEN=%.2Le\n", maxEN);
348
349   space(2);
350
351
352   printf("Refining␣solutions␣for␣K,␣N,␣omega,␣lambda\n");
353   printf("---------------------------------------------------\n");
354   space(1);
355
356   int d=0;
357   do{
358         // The substep for K and omega is applied
359         qNewton_K_step(&lambda,&omega,K,N,DK,EK);
360         // The values are refreshed
361         T=1./omega;
362         printf("%20.12Lf\n", creall(K[0][0]));
363         forwardfourier(K,K_);
364         derivatefourier(K_,DK_);
365         backwardfourier(DK_,DK);
366         // The error function EK is refreshed
367         maxEK=0;
368         for(int j=0; j<Ngrid; j++){
369              x[0]=K[0][j];
370              x[1]=K[1][j];
```

```
371                 long double f[2];
372                 field(x,f);
373                 EK[0][j]=f[0]-DK[0][j]*omega;
374                 EK[1][j]=f[1]-DK[1][j]*omega;
375                 norm=hypotl(EK[0][j],EK[1][j]);
376                 if(norm>maxEK){
377                     maxEK=norm;
378                 }
379             }
380             printf("maxEK=%.2Le\n", maxEK);
381             // The errors function EN is refreshed to the mid-value
382             for(int j=0; j<Ngrid; j++){
383                  long double df[2][2];
384                  x[0]=K[0][j];
385                  x[1]=K[1][j];
386                  V[0]=N[0][j];
387                  V[1]=N[1][j];
388                  dfield(x,df);
389                  matrixvector2(df,V);
390                  EN[0][j]=V[0]-lambda*N[0][j]-DN[0][j]*omega;
391                  EN[1][j]=V[1]-lambda*N[1][j]-DN[1][j]*omega;
392              }
393              // The substep for N and lambda is applied
394              qNewton_N_step(&lambda,&omega,K,N,DK,DN,EN);
395              // The values are refreshed
396              printf("%20.12Lf\n", lambda);
397              forwardfourier(N,N_);
398              derivatefourier(N_,DN_);
399              backwardfourier(DN_,DN);
400              // The error function EN is refreshed
401              maxEN=0;
402              for(int j=0; j<Ngrid; j++){
403                  long double df[2][2];
404                  x[0]=K[0][j];
405                  x[1]=K[1][j];
406                  V[0]=N[0][j];
407                  V[1]=N[1][j];
408                  dfield(x,df);
409                  matrixvector2(df,V);
410                  EN[0][j]=V[0]-lambda*N[0][j]-DN[0][j]*omega;
411                  EN[1][j]=V[1]-lambda*N[1][j]-DN[1][j]*omega;
412                  norm=hypotl(EN[0][j],EN[1][j]);
413                  if(norm>maxEN){
414                      maxEN=norm;
415                  }
416              }
417              printf("maxEN=%.2Le\n", maxEN);
418              d++;
419         }while(d<10);
420
421
422
423
424     destroyfourier();
425     free(K[0]);
426     free(K[1]);
427     free(K_[0]);
428     free(K_[1]);
```

```
429    free(DK[0]);
430    free(DK[1]);
431    free(DK_[0]);
432    free(DK_[1]);
433    free(EK[0]);
434    free(EK[1]);
435
436    free(N[0]);
437    free(N[1]);
438    free(N_[0]);
439    free(N_[1]);
440    free(DN[0]);
441    free(DN[1]);
442    free(DN_[0]);
443    free(DN_[1]);
444    free(EN[0]);
445    free(EN[1]);
446
447    space(2);
448
449    return 0;
450  }
451
452
453
454  void checkpoint(){
455    static int countercheck=1;
456    printf("THIS IS CHECKPOINT NUMBER %d\n", countercheck);
457    countercheck++;
458  }
459
460
461
462  void space(int n){
463    for(int i=0; i<n; i++){
464      printf("\n");
465    }
466  }
467
468
469
470  // The field evaluated at vector x and stored in vector f
471  void field(long double *x, long double *f){
472      long double minfi=1.+exp(-(x[0]-Vmaxm)/km);
473      long double minf=1./minfi;
474      long double ninfi=1.+exp(-(x[0]-Vmaxn)/kn);
475      long double ninf=1./ninfi;
476      f[0]=(current-gNa*minf*(x[0]-VNa)-gK*x[1]*(x[0]-VK)-gL*(x[0]-VL))/C;
477      f[1]=ninf-x[1];
478  }
479
480
481
482  // The differential of the field is evaluated at vector x and stored in matrix df
483  void dfield(long double *x, long double df[2][2]){
484    long double minfi=1.+exp(-(x[0]-Vmaxm)/km);
485    long double minf=1./minfi;
486    long double ninfi=1.+exp(-(x[0]-Vmaxn)/kn);
```

```
487   long double ninf=1./ninfi;
488   long double dminf_dV=(minf-minf*minf)/km;
489   long double dninf_dV=(ninf-ninf*ninf)/kn;
490   df[0][0]=(-gNa*minf-gNa*(x[0]-VNa)*dminf_dV-gK*x[1]-gL)/C;
491   df[1][0]=(-gK*(x[0]-VK))/C;
492   df[0][1]=dninf_dV;
493   df[1][1]=-1.;
494 }
495
496
497
498 // Computes the result of applying M to v and stores the result in v
499 void matrixvector2(long double M[2][2], long double *v){
500   long double vaux[2];
501   vaux[0]=v[0];
502   vaux[1]=v[1];
503   v[0]=vaux[0]*M[0][0]+vaux[1]*M[1][0];
504   v[1]=vaux[0]*M[0][1]+vaux[1]*M[1][1];
505 }
506
507
508
509 // Applies the inverse of matrix M to v and stores the result in v
510 void invmatrixvector2(long double M[2][2], long double *v){
511   long double det=M[0][0]*M[1][1]-M[0][1]*M[1][0];
512   long double vaux[2];
513   vaux[0]=v[0];
514   vaux[1]=v[1];
515   v[0]=vaux[0]*M[1][1]-vaux[1]*M[1][0];
516   v[1]=-vaux[0]*M[0][1]+vaux[1]*M[0][0];
517   v[0]/=det;
518   v[1]/=det;
519 }
520
521
522
523 // If x is in the Poincare section at V=-40 applies to it the Poincare map, if it is
      not it lets it evolve under the flux until the Poincare section is reached anyway
524 long double poincaremap(long double *x){
525   long double tau=0;
526   long double h=0;
527   int order=20;
528   long double taux;
529   long double xaux[2];
530
531   // Right semicycle if needed
532   while(x[0]>=-40.-tol){
533     taylor_step_inapik(&tau, x, 1,1, log10tol, log10rtol, NULL, &h, &order);
534   }
535
536   // It approaches the Poincare section and stops with xaux right before and x right
        after
537   do{
538     xaux[0]=x[0];
539     xaux[1]=x[1];
540     taux=tau;
541     taylor_step_inapik(&tau, x, 1,1, log10tol, log10rtol, NULL, &h, &order);
542   }while(x[0]<-40);
```

```
543
544   x[0]=xaux[0];
545   x[1]=xaux[1];
546   tau=taux;
547
548   // Newton to obtain tau and the return point to the Poincare section.
549   long double delta=0;
550   long double t=h;
551   long double **a;
552   do{
553     taux=0;
554     xaux[0]=x[0];
555     xaux[1]=x[1];
556     taylor_step_inapik(&taux, xaux, 1, 0, log10tol, log10rtol, NULL, &t, &order);
557     /* a=taylor_coeficients_inapik (t, x, order) gives a two dim array. The rows are
           the taylor coefficients of order i of the state variables */
558     /* a[0][0] and a[1][0] are x[0] and x[1] */
559     /* a[0][1] and a[1][1] are the components of the field in (a[0][0],a[1][0]) */
560     /* a[][2] gives the second order coefficients, which are the derivatives times 2!
            */
561     /* ... */
562     a=taylor_coefficients_inapik(0,xaux,1);
563     delta=(xaux[0]-(-40.))/(a[0][1]);
564     t=t-delta;
565     // Ensure that t is < dstep (the last step taken for which the taylor step was
           precise enough
566     if(fabs(t)>fabs(h)){
567       printf("\nLittle problem: t>last step: t-h=%20.12Le\n\n", t-h);
568     }
569   }while(fabs(xaux[0]-(-40))>tol/100.);
570   x[0]=xaux[0];
571   x[1]=xaux[1];
572   tau=tau+t;
573
574   return tau;
575 }
576
577
578
579 void createfourier(){
580   in=(fftwl_complex *) fftwl_malloc(sizeof(fftwl_complex)*Ngrid);
581   out=(fftwl_complex *) fftwl_malloc(sizeof(fftwl_complex)*Ngrid);
582   planfor=fftwl_plan_dft_1d(Ngrid,in,out,FFTW_FORWARD,FFTW_ESTIMATE);
583   planback=fftwl_plan_dft_1d(Ngrid,in,out,FFTW_BACKWARD,FFTW_ESTIMATE);
584 }
585
586
587
588 void destroyfourier(){
589   fftwl_destroy_plan(planback);
590   fftwl_destroy_plan(planfor);
591   fftwl_free(in);
592   fftwl_free(out);
593 }
594
595
596
597 // Computation of the forward Fourier transform of the vector array IN. The result is
```

```
      stored in OUT. Note that the result is divided by Ngrid
598 void forwardfourier(long double complex **IN, long double complex **OUT){
599   int i;
600   for(i=0; i<Ngrid; i++){
601     in[i]=IN[0][i];
602   }
603   fftwl_execute(planfor);
604   for(i=0; i<Ngrid; i++){
605     OUT[0][i]=out[i]/Ngrid;
606   }
607
608   for(i=0; i<Ngrid; i++){
609     in[i]=IN[1][i];
610   }
611   fftwl_execute(planfor);
612   for(i=0; i<Ngrid; i++){
613     OUT[1][i]=out[i]/Ngrid;
614   }
615 }
616
617
618
619 // Computation of the backward Fourier transform of the vector array IN. The result
      is stored in OUT
620 void backwardfourier(long double complex **IN, long double complex **OUT){
621   int i;
622   for(i=0; i<Ngrid; i++){
623     in[i]=IN[0][i];
624   }
625   fftwl_execute(planback);
626   for(i=0; i<Ngrid; i++){
627     OUT[0][i]=out[i];
628   }
629   for(i=0; i<Ngrid; i++){
630     in[i]=IN[1][i];
631   }
632   fftwl_execute(planback);
633   for(i=0; i<Ngrid; i++){
634     OUT[1][i]=out[i];
635   }
636 }
637
638
639
640 // Derivation of the function K(theta). We have the Fourier coefficients of this
      function in K_ and this function stores the coefficients of the derivative in DK_.
      Note that the Nyquist coefficient is set to 0.
641 void derivatefourier(long double complex **K_, long double complex **DK_){
642   DK_[0][ind(-Ngrid/2)]=0.*K_[0][ind(-Ngrid/2)];
643   DK_[1][ind(-Ngrid/2)]=0.*K_[1][ind(-Ngrid/2)];
644   for(int j=-Ngrid/2; j<Ngrid/2; j++){
645     DK_[0][ind(j)]=2.*M_PI*j*I*K_[0][ind(j)];
646     DK_[1][ind(j)]=2.*M_PI*j*I*K_[1][ind(j)];
647   }
648 }
649
650
651
```

```
652 // Reindexing. Given an index i between -N/2 and N/2-1 (Fourier coefficients) and it
        returns the corresponding index for the arrays given by the forward Fourier
        transform by FFTW
653 int ind(int i){
654   if(i>=0 && i<Ngrid/2){
655     return i;
656   }else if(i<0 && i>-Ngrid/2-1){
657     return i+Ngrid;
658   }else{
659     printf("The␣index␣is␣out␣of␣the␣boundary\n");
660     exit(-1);
661   }
662 }
663
664
665
666 // quasiNewton substep for K and omega
667 void qNewton_K_step(long double *lambda, long double *omega, long double complex **K,
         long double complex **N, long double complex **DK, long double complex **EK){
668     long double vaux[2];
669     long double M[2][2];
670     long double complex *frame[2][2];
671     frame[0][0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
672     frame[0][1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
673     frame[1][0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
674     frame[1][1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
675     long double complex *EKframe[2];
676     EKframe[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
677     EKframe[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
678     long double complex *EKframe_[2];
679     EKframe_[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
680     EKframe_[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
681     long double complex *DeltaKframe[2];
682     DeltaKframe[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
683     DeltaKframe[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
684     long double complex *DeltaKframe_[2];
685     DeltaKframe_[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid)
         ;
686     DeltaKframe_[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid)
         ;
687
688     long double deltaomega;
689     long double complex *DeltaK[2];
690     DeltaK[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
691     DeltaK[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
692
693
694
695
696     for(int j=0; j<Ngrid; j++){
697
698         // Store the frame P for each value of theta. P=(K'|N)
699         frame[0][0][j]=DK[0][j];
700         frame[0][1][j]=DK[1][j];
701         frame[1][0][j]=N[0][j];
702         frame[1][1][j]=N[1][j];
703        M[0][0]=frame[0][0][j];
704        M[0][1]=frame[0][1][j];
```

```
705            M[1][0]=frame[1][0][j];
706            M[1][1]=frame[1][1][j];
707
708            vaux[0]=EK[0][j];
709            vaux[1]=EK[1][j];
710            // Apply P(-1) to the error function in order to find the error in the frame.
                   Index 0 corresponds to the correction in the tangent direction, while
                  index 1 corresponds to the correction in the direction of the stable
                  vector
711            invmatrixvector2(M,vaux);
712            EKframe[0][j]=vaux[0];
713            EKframe[1][j]=vaux[1];
714        }
715
716        // Fourier coefficients of the error in the frame
717        forwardfourier(EKframe,EKframe_);
718
719        // Find delta omega and the corrections of the Fourier coefficients in the frame
               according to the formulas
720        deltaomega=EKframe_[0][ind(0)];
721        for(int j=-Ngrid/2; j<Ngrid/2; j++){
722            if(j!=0){
723                DeltaKframe_[0][ind(j)]=EKframe_[0][ind(j)]/(2*M_PI*I*j*(*omega));
724            }else{
725                DeltaKframe_[0][ind(j)]=0;
726            }
727            DeltaKframe_[1][ind(j)]=EKframe_[1][ind(j)]/(2*M_PI*I*j*(*omega)-(*lambda));
728        }
729
730        // Obtain the correction in the frame at each theta from the correction of the
               Fourier coefficients via an inverse Fourier transform
731        backwardfourier(DeltaKframe_,DeltaKframe);
732
733        for(int j=0; j<Ngrid; j++){
734            M[0][0]=frame[0][0][j];
735            M[0][1]=frame[0][1][j];
736            M[1][0]=frame[1][0][j];
737            M[1][1]=frame[1][1][j];
738            vaux[0]=DeltaKframe[0][j];
739            vaux[1]=DeltaKframe[1][j];
740            // Go back from the frame to the original coordinate system
741            matrixvector2(M,vaux);
742            DeltaK[0][j]=vaux[0];
743            DeltaK[1][j]=vaux[1];
744        }
745
746        // Refresh values for K, omega
747        for(int j=0; j<Ngrid; j++){
748            K[0][j]+=DeltaK[0][j];
749            K[1][j]+=DeltaK[1][j];
750        }
751        (*omega)+=deltaomega;
752
753
754        free(frame[0][0]);
755        free(frame[0][1]);
756        free(frame[1][0]);
757        free(frame[1][1]);
```

```
758     free(EKframe[0]);
759     free(EKframe[1]);
760     free(EKframe_[0]);
761     free(EKframe_[1]);
762     free(DeltaKframe[0]);
763     free(DeltaKframe[1]);
764     free(DeltaKframe_[0]);
765     free(DeltaKframe_[1]);
766 
767     free(DeltaK[0]);
768     free(DeltaK[1]);
769 }
770 
771 
772 
773 // quasiNewton substep for N and lambda (similar as before)
774 void qNewton_N_step(long double *lambda, long double *omega, long double complex **K,
        long double complex **N, long double complex **DK, long double complex **DN, long
        double complex **EN){
775     long double vaux[2];
776     long double M[2][2];
777     long double complex *frame[2][2];
778     frame[0][0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
779     frame[0][1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
780     frame[1][0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
781     frame[1][1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
782     long double complex *ENframe[2];
783     ENframe[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
784     ENframe[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
785     long double complex *ENframe_[2];
786     ENframe_[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
787     ENframe_[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
788     long double complex *DeltaNframe[2];
789     DeltaNframe[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
790     DeltaNframe[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
791     long double complex *DeltaNframe_[2];
792     DeltaNframe_[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid)
            ;
793     DeltaNframe_[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid)
            ;
794 
795     long double deltalambda;
796     long double complex *DeltaN[2];
797     DeltaN[0]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
798     DeltaN[1]=(long double complex *) malloc(sizeof(long double complex)*Ngrid);
799 
800 
801     for(int j=0; j<Ngrid; j++){
802 
803         frame[0][0][j]=DK[0][j];
804         frame[0][1][j]=DK[1][j];
805         frame[1][0][j]=N[0][j];
806         frame[1][1][j]=N[1][j];
807         M[0][0]= frame[0][0][j];
808         M[0][1]= frame[0][1][j];
809         M[1][0]= frame[1][0][j];
810         M[1][1]= frame[1][1][j];
811 
```

```
812             vaux[0]=EN[0][j];
813             vaux[1]=EN[1][j];
814             invmatrixvector2(M,vaux);
815             ENframe[0][j]=vaux[0];
816             ENframe[1][j]=vaux[1];
817         }
818
819         forwardfourier(ENframe,ENframe_);
820
821
822         for(int j=-Ngrid/2; j<Ngrid/2; j++){
823             DeltaNframe_[0][ind(j)]=ENframe_[0][ind(j)]/((*lambda)+2*M_PI*I*j*(*omega));
824             if(j!=0){
825                 DeltaNframe_[1][ind(j)]=ENframe_[1][ind(j)]/(2*M_PI*I*j*(*omega));
826             }else{
827                 deltalambda=ENframe_[1][ind(0)];
828                 DeltaNframe_[1][ind(j)]=0.;
829             }
830         }
831
832         backwardfourier(DeltaNframe_,DeltaNframe);
833
834         for(int j=0; j<Ngrid; j++){
835             M[0][0]=frame[0][0][j];
836             M[0][1]=frame[0][1][j];
837             M[1][0]=frame[1][0][j];
838             M[1][1]=frame[1][1][j];
839             vaux[0]=DeltaNframe[0][j];
840             vaux[1]=DeltaNframe[1][j];
841             matrixvector2(M,vaux);
842             DeltaN[0][j]=vaux[0];
843             DeltaN[1][j]=vaux[1];
844         }
845
846         // Refresh values for N, lambda
847         for(int j=0; j<Ngrid; j++){
848             N[0][j]+=DeltaN[0][j];
849             N[1][j]+=DeltaN[1][j];
850         }
851         (*lambda)+=deltalambda;
852
853
854         free(frame[0][0]);
855         free(frame[0][1]);
856         free(frame[1][0]);
857         free(frame[1][1]);
858         free(ENframe[0]);
859         free(ENframe[1]);
860         free(ENframe_[0]);
861         free(ENframe_[1]);
862         free(DeltaNframe[0]);
863         free(DeltaNframe[1]);
864         free(DeltaNframe_[0]);
865         free(DeltaNframe_[1]);
866
867         free(DeltaN[0]);
868         free(DeltaN[1]);
869     }
```