Software Project, Approximation and Simulation of Stochastic Nerve Axon Equations

Christian Pangerl

Contents

Chapter 1

Abstract

This paper focuses on the implementation of numerical approximations to the stochastic Nagumo equation, a random ordinary differential equation for phase-adaption and the stochastic Hodgkin-Huxley model. For each of these issues, a brief introduction and a subsequent detailed presentation of the specific source code is provided. Besides the accurate numerical implementations, a further objective was, to some extend, the development of a framework that is easily extendible to different numerical solvers as well as numerical SPDEs.

Chapter 2

Introduction

Stochastic Nagumo and Hodgkin-Huxley equations are examples of stochastic reaction-diffusion equations, or more generally stochastic partial differential equations (SPDEs). These are partial differential equations (PDEs) forced by a Wiener process. For a rigorous introduction into this matter see eg. [3]. We want to give here, however, a brief numerical presentation of this topic, which is taken from [1, p. 436–440].

2.1 Q-Wiener Processes

Throughout this chapter, we assume a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ with filtration $(\mathcal{F}_t)_{t\geq 0}$. Furthermore, separable Hilbert spaces are denoted by U or H.

Definition 2.1.1. A bounded linear operator $Q \in L(U)$, which is nonnegative, symmetric and trace-class is called *covariance operator*.

Definition 2.1.2. A U-valued stochastic process $(W(t))_{t\geq 0}$ is called *Q-Wiener process*, if

i) $W(0) = 0$ a.s.,

ii) $t \mapsto W(t)$ is a continuous function $\mathbb{R}_+ \to U$, for a.e. $\omega \in \Omega$,

iii) $W(t)$ is $\mathcal{F}_t$-adapted and $W(t) - W(s)$ is independent of $\mathcal{F}_s$ for $s < t$, and

iv) $W(t) - W(s) \sim N(0, (t-s)Q)$ for every $0 \leq s \leq t$.

In the following, we will assume existence of Q-Wiener processes. As particular example set $U = L^2(0, a)$ for $a > 0$. We would like to derive sufficient conditions for which an U-valued Q-Wiener process assumes values in the Sobolev space $H_0^r(0, a)$. For this purpose, let $A : \mathcal{D}(A) \subset U \to U$ be the unbounded linear operator given by the Laplacian with homogeneous Dirichlet boundary conditions, $Au = -\partial_{xx}^2 u$, with $\mathcal{D}(A) = H^2(0, a) \cap H_0^1(0, a)$. Then, $\mathcal{D}(A^{r/2})$ equals $H_0^r(0, a)$. Furthermore, A has eigenfunctions $\phi_j(x) =$

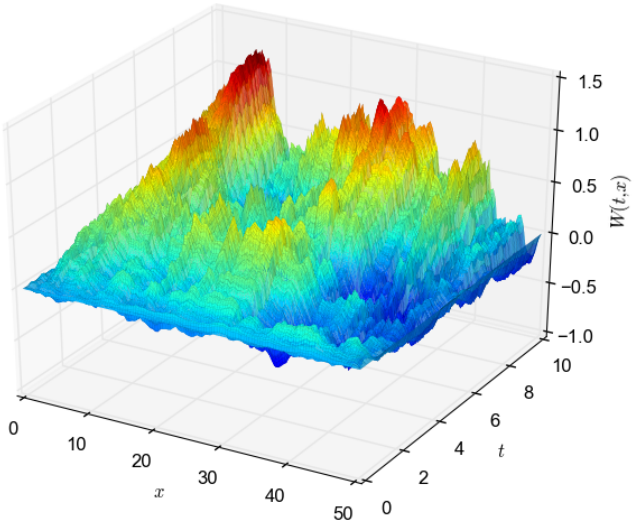

Figure 2.1: Wiener process for $r = 0.5$.

$\sqrt{2/a}\sin(j\pi x)$, which form an orthonormal basis in U. The eigenvalues λ_j to these eigenfunctions scale like $|\lambda_j| \propto j^2$. Consequently, a Q-Wiener process takes values in $H_0^r(0,a)$, if the associated covariance operator Q has eigenfunctions ϕ_j to eigenvalues $q_j = |j|^{-(2r+1+\varepsilon)}$, for an arbitrary $\varepsilon > 0$.

2.2 Approximation of Q-Wiener Processes

Let $J \in \mathbb{N}$, set $h = \frac{a}{J}$ and consider the grid points $x_k = kh$, for $k = 1, \ldots, J-1$. In addition, fix $T > 0$, $N \in \mathbb{N}$ and let $\Delta t = \frac{T}{N}$, for $n = 0, \ldots, N$. A $H_0^r(0,a)$-valued Q-Wiener process, where Q has eigenfunctions and associated eigenvalues as above, is approximated via the truncated Karhunen-Loève expansion

$$W^{J-1}(t_{n+1}, x_k) - W^{J-1}(t_n, x_k) = \sum_{j=1}^{J-1} b_j \sin\left(\frac{\pi jk}{J}\right) \xi_j^n, \tag{2.1}$$

for $k = 1, \ldots, J-1$, $n = 0, \ldots, N-1$, $b_j := \sqrt{2q_j \Delta t/a}$ and ξ_j^n, independent and identically $N(0,1)$-distributed samples. The sum (2.1) is efficiently computed by the discrete sinus transform.

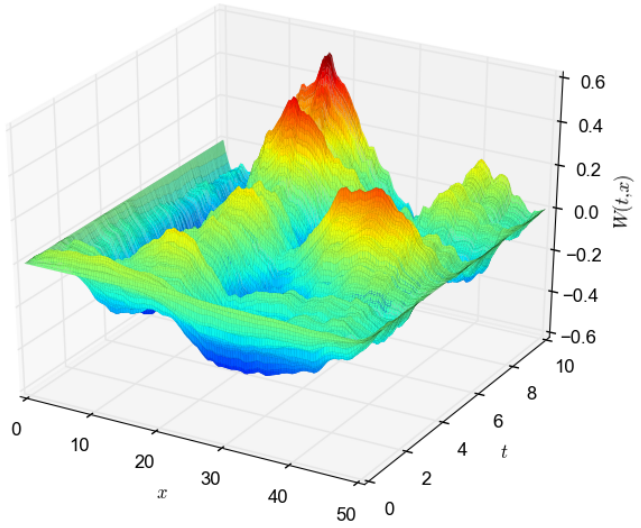

Figure 2.2: Wiener process for $r = 1$.

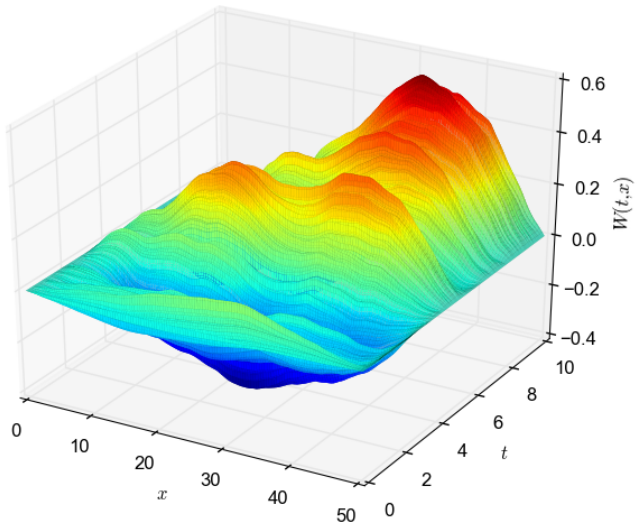

Figure 2.3: Wiener process for $r = 1.5$.

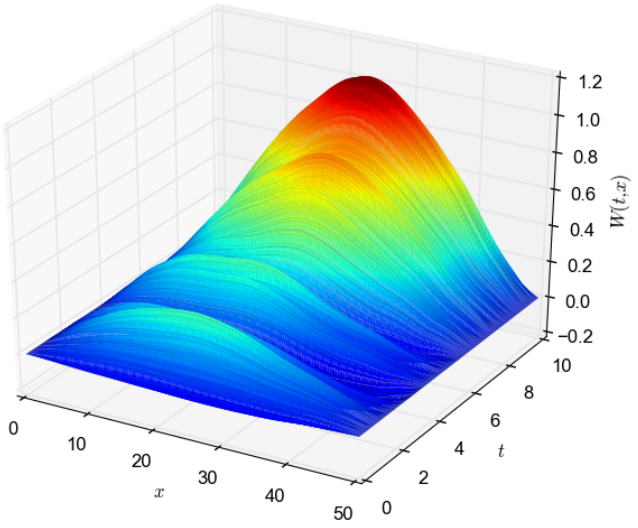

Figure 2.4: Wiener process for $r = 2$.

2.3 Implementation

The main functionality for running simulations is listed below. It basically implements (2.1).

Listing 2.1: `Q_WP_simulation.py`

```
import numpy as np
import scipy.fftpack as spfft


def coeff_b_j(a,J,r,epsilon,delta_t):
    '''
    ###################### PURPOSE: ##############################
    Composes coefficients b_j for simulation of a Q-Wiener
    process by virtue of a discrete sine transform (DST)

    ###################### INPUT: ################################
    a       : interval (0,a) as domain,
    J       : (J-1) is the number of grid points in (0,a),
              boundary {0,a} excluded
              CONVENTION: J must be a power of 2, in order to
```

```
            facilitate fast computation using DST
    r, epsilon    : regularity parameters
    delta_t       : time step

    ###################### OUTPUT: ################################
    b_j              : numpy-array of J-1 components

    ################################################################
    '''
    q_j = 1./(np.arange(1, J, dtype=float)**((2*r + 1 + epsilon)/2.))
    b_j = np.sqrt(2./a * delta_t) * q_j
    return b_j


def simulate_Q_WP_increments(a,J,r,epsilon,delta_t,M):
    '''
    ######################## INPUT: ##################################
    a, J, r, epsilon, delta_t     : as in method coeff_b_j
    M                             : number of time steps

    ######################## OUTPUT: #################################
    delta_W : matrix of dimensions J+1 x M+1, each column contains
              an approximation to an INCREMENT of the Q-Wiener
              process at specific time steps
              REMARKS: row 0, and row J are filled with zeros for
                       Dirichlet boundary conditions;
                       column 0 is filled with zeros for W(t=0)

    ################################################################
    '''
    b_j = coeff_b_j(a,J,r,epsilon,delta_t)
    # b_j has J-1 components

    delta_W = np.zeros((J+1,M+1))
    delta_W[1:J,1:] = np.random.randn(J-1,M)
    delta_W[1:J,1:] = delta_W[1:J,1:] * np.atleast_2d(b_j).transpose()

    # applying DST-1 (type=1) to each column (axis=0) of delta_W[1:J,1:]
    delta_W[1:J,1:] = 0.5 * spfft.dst(delta_W[1:J,1:], type=1, axis=0)
    return delta_W


def simulate_Q_WP(a,J,r,epsilon,delta_t,M):
    '''
```

```
    ##################### INPUT: ###################################
    as for simulate_Q_WP_increments

    ################### OUTPUT: ###################################
    W  : matrix of dimensions J+1 x M+1, each column contains an
         approximation to the Q-Wiener process at specific time
         steps
         REMARKS:
         row 0, and row J are filled with zeros for Dirichlet
         boundary conditions; column 0 is filled with zeros for
         W(t=0)

    ################################################################
    '''
    W = simulate_Q_WP_increments(a, J, r, epsilon, delta_t, M)

    # summing along each line of delta_W[1:J,1:] yields the values of the
    # Q-Wiener process at each time step
    W[1:J,1:] = np.cumsum(W[1:J,1:], axis=1)
    return W
```

For visualization purposes, we have

Listing 2.2: Q_WP_visualization.py

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from pylab import*


def visualize_Q_WP(X_J,t,W):
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    X_J, t = np.meshgrid(X_J, t)
    surf = ax.plot_surface(X_J, t, W,
                           rstride=1, cstride=1,
                           cmap=cm.jet, linewidth=0.02)

    ax.set_xlabel('$x$')
    ax.set_ylabel('$t$')
    ax.set_zlabel('$W(t,x)$')

    plt.show()
```

Finally, the main-program for linking simulation and subsequent visualization.

Listing 2.3: Q_WP_main.py

```
import numpy as np
from Q_WP_simulation import simulate_Q_WP
from Q_WP_visualization import visualize_Q_WP


if __name__ == "__main__":

    T = 10.
    N = 1000

    t = np.linspace(0, T, N+1)
    delta_t = np.abs(t[1]-t[0])

    a = 50.
    J = 128
    X_J = np.linspace(0,a,J+1)

    r = 2.
    epsilon = 0.001

    # simulation
    W = simulate_Q_WP(a, J, r, epsilon, delta_t, N)

    # visualization
    visualize_Q_WP(X_J, t, np.transpose(W))
```

Chapter 3

Implementation of Stochastic Nagumo Equations

This chapter is to approximate the stochastic Nagumo equation, in order to simulate stochastic travelling wave solutions. For the readers convenience, we begin by recalling some of the results from [2, Chapter 5], where a discussion of the numerical results can be found as well. The main objective is to describe the implementation.

3.1 Stochastic Nagumo Equations

We are concerned with a particular version of stochastic Nagumo equations for parameters $\nu, b > 0$, $\alpha \in (0, 1]$ and $\bar{\nu} := \nu(1 + \alpha^2 b)$, given by

$$dv(t) = \left[\bar{\nu}\partial_{xx}^2 v(t) + bf(v(t))\right] dt + \sigma(v(t))\, d\beta(t), \tag{3.1}$$

on $(t, x) \in \mathbb{R}_+ \times \mathbb{R}$, with initial condition $v(0) = v_0 \in L^2(\mathbb{R})$. The function $f : \mathbb{R} \to \mathbb{R}$ is of the form

$$f(v) = v(1 - v)(v - a), \quad a \in (0, 1),$$

and $\sigma : \mathbb{R} \to \mathbb{R}$ is defined by

$$\sigma(v) = \begin{cases} \alpha b\, v(1 - v), & \text{for } v \in [0, 1] \\ 0, & \text{else.} \end{cases}$$

The stochastic process $(\beta(t))_{t\geq 0}$ denotes a one-dimensional Brownian motion w.r.t. a filtration $(\mathcal{F}_t)_{t\geq 0}$ on the probability space $(\Omega, \mathcal{F}, \mathbb{P})$.

Equation (3.1) exhibits an explicitly given stochastic travelling wave

$$\hat{v}(t, x) := v^{TW}\left(x + ct + \sqrt{2b\nu}\alpha\, \beta(t)\right), \tag{3.2}$$

which is a real-valued stochastic process $(\hat{v}(t,x))_{t\geq 0}$, for every $x \in \mathbb{R}$, where $c = \sqrt{2b\nu}\left(\frac{1}{2} - a\right)$ is the deterministic part of the wave speed. Furthermore,

$$v^{TW}(x) = \frac{1}{1+e^{-kx}}, \quad k = \sqrt{\frac{b}{2\nu}}, \tag{3.3}$$

denotes the wave profile of a travelling wave solution to the deterministic version of equation (3.1). As we are solely interested in approximations and simulations, we refer to [2] for a more detailed investigation.

3.2 Finite Difference Approximation

The following is a summary of [2, Chapter 5]. Therefore, we use centered finite differences for spatial and a semi-implicit Euler-Maruyama method for temporal approximation of equation (3.1).

3.2.1 Spatial Discretization

We consider approximations of (3.1) on an interval $[0, L]$, for $L > 0$. Let $x_j := jh$, $j = 0, \ldots, J$, for $J \in \mathbb{N}$ and $h := \frac{L}{J}$. The finite difference approximation to solutions $(v(t))_{t\geq 0}$ of (3.1), for any $t \geq 0$ is given by

$$d\mathbf{v}_J(t) = \left[\bar{\nu} A^N \mathbf{v}_J(t) + b\mathbf{f}(\mathbf{v}_J(t))\right] dt + \sigma(\mathbf{v}_J(t))\, d\beta(t), \tag{3.4}$$

with $\mathbf{v}_J(0) := [v_0(x_0), \ldots, v_0(x_J)]^{\mathrm{T}} \in \mathbb{R}^{J+1}$, and

$$A^N := -\frac{1}{h^2}\begin{pmatrix} 2 & -2 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & \ddots & \ddots & -1 \\ & & & -2 & 2 \end{pmatrix} \in \mathbb{R}^{(J+1)\times(J+1)}, \tag{3.5}$$

being the centered finite difference approximation to the Laplacian in (3.1) with homogeneous Neumann boundary conditions. As outlined in [2], equations (3.1), as well as 3.4) have unique solutions to any respective initial value. The coefficients in (3.1), should be understood as,

$$\mathbf{f}(\mathbf{v}(t)) := [f(v(t,x_0)), \ldots, f(v(t,x_J))]^{\mathrm{T}},$$

and

$$\sigma(\mathbf{v}(t)) := [\sigma(v(t,x_0)), \ldots, \sigma(v(t,x_J))]^{\mathrm{T}},$$

for every $t \geq 0$.

3.2.2 Temporal Discretization

For $T > 0$ set $\Delta t := \frac{T}{N}$, $N \in \mathbb{N}$, and approximate the solution $\mathbf{v}_J(t_n)$ of (3.4) at $t_n = n\Delta t$ for $n = 1, \ldots, N$, by

$$\mathbf{v}_{J,n} := \left[I - \Delta t\, \bar{\nu} A^N\right]^{-1} \left(\mathbf{v}_{J,n-1} + \Delta t\, b\, \mathbf{f}(\mathbf{v}_{J,n-1}) + \sqrt{\Delta t}\, \sigma(\mathbf{v}_{J,n-1})\xi_{n-1}\right), \tag{3.6}$$

where $\mathbf{v}_{J,0} = \mathbf{v}_J(0)$ with $(\xi_n)_{n=1,\ldots,N}$, independent and identically $N(0,1)$-distributed random variables.

For a detailed discussion of simulation results see [2].

3.3 Implementation

3.3.1 Numerical Realization of the stochastic Nagumo Equation

This subsection is concerned with presenting a concise approach for implementing the stochastic Nagumo equation in particular and stochastic reaction-diffusion equations in general.

In this regard, the class **SPDE** serves as base class for numerical versions of stochastic reaction-diffusion equations. A finite difference implementation is provided by **SPDE_FD**. Nevertheless, the code is designed to support simple extendability to e.g. Galerkin based methods. Furthermore, the first and foremost objective was to develop a flexible enough framework for simulating SPDEs using reasonable conventions. This is done by treating e.g. SPDEs or spatial and temporal grids as instances of corresponding classes. This has the advantage of providing common interfaces to functional implementations of the numerical schemes in question (see eg. Subsection 3.3.2).

Listing 3.1: `classes_for_spde_handling.py`

```
import classes_for_grid_handling as grh
from fcts_helpers import composeA

class SPDE(object):
    '''
    ################# PURPOSE: ########################
    Base class for numerical versions of stochastic
    reaction diffusion equations

    ################# ATTRIBUTES: #####################
    '''
    # DIFFUSION
    # diffusion parameter
    diff_param = None
```

```
15
    # boundary condition
17  BC = None

19  # discrete Laplacian
    A = None
21
    # REACTION TERM
23  f = None

25  # NOISE
    multiplicative_noise = True
27
    # stochastic diffusion coefficient
29  sigma = None

31  # Wiener process
    wp = None
33
    # INITIAL VALUE
35  init_value = None

37  def __init__(self,diff_param,f,sigma,
                 init_value,A=None,BC=None):
39      # set diffusion
        self.diff_param = diff_param
41      self.BC = BC
        self.A = A
43
        # set reaction term
45      self.f = f

47      # set noise
        self.sigma = sigma
49
        # initial value
51      self.init_value = init_value


53
class SPDE_FD(SPDE):
55  '''
    ################### PURPOSE: ######################
57  Sub-class of class SPDE implementing a finite
    difference approximation
```

```
59
    ################### ATTRIBUTES: ####################
61  '''
    def __init__(self,diff_param,f,sigma,init_value,BC,
63                 L,J,T,N,spatial_two_sided):
        # GRID
65      self.st_grid = grh.SpatioTemporalGrid(L,
                                              spatial_two_sided,
67                                            J,T,N)
        # LAPLACIAN
69      # with boundary conditions BC
        if diff_param != None:
71          A = composeA(self.st_grid.spatial_grid.X_J.shape[0],BC)
        else:
73          A = None

75      # set general attributes of SPDE
        SPDE.__init__(self,diff_param,f,sigma,init_value,A,BC)
77


79 ##########################################################################
   ######## Base-Classes for Drift/Diffusion/Noise wrappers #######
81 ##########################################################################
   class DriftCoeff:
83      def __init__(self,drift):
            self.drift = drift
85


87 class DiffusionCoeff:
        def __init__(self,diffusion):
89          self.diffusion = diffusion


91
   class Noise(object):
93      '''
        ##################### PURPOSE: ############################
95      Common Base-Class for AddNoise and MultNoise classes

97      ###########################################################
        '''
99      def __init__(self):
            pass
101

        def eval_with_wp(self):
```

```
103         pass


105
class AddNoise(Noise):
107     '''
        ##################### PURPOSE: ##########################
109     Base-class for additive noise; implements an efficient
        evaluation of additive noise with a realization of a
111     Wiener process wp

113     ########################################################
        '''
115     def __init__(self,sigma,B):
            self.sigma = sigma
117         self.B = B

119     def eval_with_wp(self,wp):
            pass
121


123 class MultNoise(Noise):
        '''
125     ##################### PURPOSE: ############################
        Base-class for multiplicative noise; implements an efficient
127     evaluation of multiplicative noise with a realization of a
        Wiener process wp
129
        ########################################################
131     '''
        def __init__(self,sigma):
133         self.sigma = sigma

135     def eval_with_wp(self,v,wp):
            pass
```

The handling of spatial and temporal grids for the numerical implementation is provided by the class **SpatioTemporalGrid**, which is used consistently in this project.

Listing 3.2: `classes_for_grid_handling.py`

```
import numpy as np
2

class SpatioTemporalGrid:
4     def __init__(self,L,spatial_two_sided,J,T,N):
```

```
6         # set spatial grid
          self.spatial_grid = SpatialGrid(L, J, spatial_two_sided)
8
          # set temporal grid
10        self.temporal_grid = TemporalGrid(T, N)


12
  class TemporalGrid:
14    def __init__(self,T,N):
          self.set_temporal_grid(T, N)
16
      def set_temporal_grid(self,T,N):
18
          # array_like
20        # length-(N+1) array of evenly spaced grid points
          self.t = np.linspace(0,T,N+1)
22
          # time step
24        self.delta_t = np.abs(self.t[1]-self.t[0])

26        self.number_of_steps = N
          self.temp_domain = (0,T)
28


30 class SpatialGrid:
      def __init__(self,L,J,spatial_two_sided):
32        self.set_spatial_grid(L,J,spatial_two_sided)

34    def set_spatial_grid(self,L,J,spatial_two_sided):
          self.spatial_two_sided = spatial_two_sided
36        if self.spatial_two_sided == True:
              # array_like
38            # length-(J+1) array of evenly spaced grid points
              self.X_J = np.linspace(-L,L,J+1)
40
              # tuple; start-, end point of domain
42            self.spatial_domain = (-L,L)
          else:
44            self.X_J = np.linspace(0,L,J+1)
              self.spatial_domain = (0,L)
46
          # mesh size
48        self.h = np.abs(self.X_J[1] - self.X_J[0])
```

```
        self.J = J
```

A concrete realization of the stochastic Nagumo equation and corresponding initial values is given by

Listing 3.3: `Nagumo_model.py`

```
import numpy as np
import classes_for_spde_handling as spde


class SPDE_FD_Nagumo(spde.SPDE_FD):
    '''
    ##################### PURPOSE: ##########################
    Sub-class of class SPDE_FD for finite differences
    approximation of stochastic Nagumo's equation

    ##################### ATTRIBUTES: #######################
    '''
    # REACTION TERM: parameters
    a = None
    b = None

    # NOISE AMPLITUDE
    alpha = None

    # DIFFUSION: parameters
    nu = None
    nu_bar = None

    # WAVE SPEEDS
    c_det = None
    c_stoch = None

    def __init__(self,nu,a,b,alpha,v_0,BC,
                 L,J,T,N,spatial_two_sided):

        # REACTION TERM: parameters
        self.a = a
        self.b = b

        # NOISE AMPLITUDE
        self.alpha = alpha

        # DIFFUSION: parameters
```

```
39          self.nu = nu
            self.nu_bar = (nu*(1+b*alpha**2))
41
            # WAVE SPEEDS
43          # deterministic case
            self.c_det = (np.sqrt(2.*self.b*self.nu_bar)
45                        *(0.5-self.a))

47          # stochastic case
            self.c_stoch = (np.sqrt(2.*self.b*self.nu)
49                          * (0.5-self.a))

51          # Base-class: SPDE_FD
            # PARAMETERS
53          f = NagumoDriftCoeff(self.a, self.b)
            sigma = NagumoDiffusionCoeff(self.alpha * self.b)
55
            spde.SPDE_FD.__init__(self, self.nu_bar,
57                                  f, sigma, v_0, BC,
                                    L, J, T, N,
59                                  spatial_two_sided)

61          # Wiener process
            self.wp = np.random.randn(self.st_grid.temporal_grid.t.shape[0],)
63

65 ##########################################################################
   ############ Drift-/Diffusion-Coefficients ###############
67 ##########################################################################
   class NagumoDriftCoeff:
69      def __init__(self,a,b):
            self.a = a
71          self.b = b

73      def eval(self,v,add_param=None):
            return self.b*v*(1.-v)*(v-self.a)
75

77 class NagumoDiffusionCoeff:
        def __init__(self,sigma_0):
79          self.sigma_0 = sigma_0

81      def eval(self,v,add_param=None):
            v[v<0.] = 0.
```

```
83          v[v>1.] = 0.
            return self.sigma_0*v*(1.-v)
85
        def eval_with_wp(self,v,wp,add_param=None):
87          return self.eval(v,add_param) * wp

89
    ##########################################################################
91  ################### Initial Values #######################################
    ##########################################################################
93  class NagumoInitialValueTW:
        def __init__(self,perturbation,location):
95          self.perturbation = perturbation
            self.location = location
97
        def eval(self,v,add_param=None):
99          return 1. /(1. + np.exp((-self.perturbation
                                     * (v - self.location) / np.sqrt(2))))
101


103 class NagumoInitialValuePulse:
        def __init__(self,perturbation,location):
105         self.perturbation = perturbation
            self.location = location
107
        def eval(self,v,add_param=None):
109         return np.exp(-(self.perturbation * (v - self.location)**2.))

111
    class NagumoInitialValueKink:
113     def __init__(self,c_1,c_2):
            self.c_1 = c_1
115         self.c_2 = c_2

117     def eval(self,v,add_param=None):
            a = -np.ones((v.shape[0],))
119         a[np.abs(v) <= self.c_2] = -self.c_1
            a[np.abs(v) > self.c_2] = self.c_1
121         a[np.abs(v) > 3. * self.c_2] = 0.
            return a
123         #return self.c_1 * np.sin(v)
```

3.3.2 Finite Difference Scheme

Here, we list the implementation of the above semi-implicit Euler-Maruyama scheme. Again, the general applicability of the following methods is emphasized.

Listing 3.4: `fcts_semi_implicit_em.py`

```
import numpy as np
import scipy.sparse as sparse
import scipy.sparse.linalg as splinalg


def compose_iteration_matrix_fd(A,J,diff_param,delta_t,h):
    # iteration matrix
    EE = sparse.eye(J) + diff_param * delta_t * A/h/h
    return EE


def fd_si_em_single_step(spde,v_n,xi,EE,
                         add_param_f=None,
                         add_param_sigma=None):
    '''
    ##################### PURPOSE: ###################################
    Performs one step in a semi-implicit Euler-Maruyama
    finite-difference scheme

    ##################### INPUT: #####################################
    spde          : instance of a class implementing a numerical
                    SPDE
    v_n           : current iteration vector
    xi            : random sample
    EE            : Iteration matrix
    add_param_f   : length-N tuple containing all other relevant
                    data for evaluating coefficient function f in
                    spde
    add_param_sigma  : length-M tuple containing all relevant
                       data for evaluating coefficient function
                       sigma in spde

    ##################### OUTPUT: ####################################
    v_n         : subsequent iteration vector

    ##################################################################
    '''
```

```
    delta_t = spde.st_grid.temporal_grid.delta_t
    f_n = spde.f.eval(v_n,add_param_f)
    if spde.multiplicative_noise:
        sigma_n_W = spde.sigma.eval_with_wp(v_n,xi,add_param_sigma)
    else:
        sigma_n_W = spde.sigma.eval_with_wp(xi)
    v_n = (splinalg.spsolve(EE, v_n + delta_t * f_n
                            + np.sqrt(delta_t) * sigma_n_W))
    return v_n


def fd_si_em_simul(spde):
    '''
    ##################### PURPOSE: ###################################
    Generates one sample path for a SPDE using finite differences
    (in space) and a semi-implicit Euler-Maruyama scheme (in time)

    ##################### INPUT: #####################################
    spde      : instance of a class implementing a numerical SPDE

    ##################### OUTPUT: ####################################
    st_grid   : instance of class SpatioTemporalGrid
    v_Jn      : matrix, columns approximate one sample
                [v(t_n,x_0),...,v(t_n,x_J)]^T

    ##################################################################
    '''

    # spatial and temporal grid
    st_grid = spde.st_grid
    temporal_grid = st_grid.temporal_grid
    spatial_grid = st_grid.spatial_grid

    l = temporal_grid.t.shape[0]
    J = spatial_grid.X_J.shape[0]

    delta_t = temporal_grid.delta_t
    h = spatial_grid.h

    # set parameters
    A = spde.A
    diff_param = spde.diff_param
    BC = spde.BC

```

```
    # Wiener process
83  xi = spde.wp

85  # compose iteration matrix
    EE = compose_iteration_matrix_fd(A,J,diff_param,delta_t,h,BC)
87
    # initialize
89  v_Jn = np.zeros((J,l))
    v_Jn[:,0] = spde.init_value.eval(spatial_grid.X_J)
91  for n in xrange(l-1):
        v_Jn[:,n+1] = fd_si_em_single_step(spde,
93                                         v_Jn[:,n],
                                           xi[n], EE)
95  return st_grid, v_Jn
```

Along with the following helper methods. The function **composeA** composes the finite difference approximation of the Laplacian 3.5. To facilitate fast computations, the sparse-module of scipy is used. Moreover, **generate_bm_sample_path** generates a single sample path of a one dimensional Brownian motion and **compute_exp_cov** computes the discrete covariance matrix to the covariance function $\exp(|x-y|/l)$ for $l > 0$.

Listing 3.5: `fcts_helpers.py`

```
import numpy as np
2 from scipy import sparse
import scipy.linalg as LA
4
from classes_for_grid_handling import TemporalGrid
6
def composeA(J,BC):
8   '''
    ###################### INPUT: ########################
10  J       : #of spatial grid points
    BC      : boundary conditions
12            'd' - homogeneous Dirichlet;
              'n' - homogeneous Neumann
14
    ##################### OUTPUT: ########################
16  A       : sparse, J-1 x J-1-matrix,
              if BC=='d', or J x J-matrix, if BC == 'n'
18
    ######################################################
20  '''
    A = sparse.diags([2.*np.ones((J, )),-np.ones((J-1, )),
```

```
22                      -np.ones((J-1, ))], [0,-1,1], format='csc')
    if BC == 'n':
24          A[0,1] = -2.
            A[-1,-2] = -2.
26      elif BC == 'd':
            A = A[1:-1,1:-1]
28      else:
            raise ValueError("Neither Neumann, nor Dirichlet boundary conditions ")
30      return A


32
def generate_bm_sample_path(T,N):
34          temp_grid = TemporalGrid(T,N)
            dW = np.random.randn(temp_grid.t.shape[0],)
36          dW[0] = 0
            dW = np.sqrt(temp_grid.delta_t) * dW
38          return temp_grid, np.cumsum(dW)


40
# compute covariance exp(-abs(x)/l)
42 def compute_exp_cov(h,J,l):
    r = np.arange(J,dtype=float)
44  toep = LA.toeplitz(r)
    return (np.exp(-np.abs(toep)*h/l))
```

A collection of minor helper functions specific to the Nagumo equation are given by

Listing 3.6: `Nagumo_helpers.py`

```
1 import numpy as np


3
def nagumo_compute_wave_speed(grid,v_Jn):
5   # indices of first appearance of an element >0.5 per column
    space_ind = np.argmax(v_Jn > 0.5, axis = 0)
7   x = grid.spatial_grid.X_J[space_ind]
    dx = np.abs(x[1:] - x[:-1])
9   c_n = dx / grid.temporal_grid.delta_t
    return c_n
11


13 def nagumo_potential_change_rate(grid,v_Jn):
    # compute column-index corresponding to x=0
15  if grid.spatial_grid.spatial_two_sided == True:
```

```
        zero_point = np.ceil(grid.spatial_grid.X_J.shape[0] / 2.)
    else:
        zero_point = 0

    # compute wave speed
    delta_s = v_Jn[zero_point,1:] - v_Jn[zero_point,:-1]
    c = np.zeros(grid.temporal_grid.t.shape[0]); c[0] = 0.
    c[1:] = grid.temporal_grid.delta_t * delta_s
    return c


def nagumo_signal_fixed_location(grid,v_Jn):
    # compute column-index corresponding to x=0
    if grid.spatial_grid.spatial_two_sided == True:
        zero_point = np.ceil(grid.spatial_grid.X_J.shape[0] / 2.)
    else:
        zero_point = 0

    s = v_Jn[zero_point,:]
    return s

def nagumo_signal_fixed_time(grid,v_Jn,n):
    s = v_Jn[:,n]
    return s
```

3.3.3 Visualization

The following functions compute images based on simulations provided by the previous methods.

Listing 3.7: Nagumo_visualization.py

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

from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from pylab import *


def nagumo_visualize_2D(grid,v_Jn):
    plt.pcolormesh(grid.spatial_grid.X_J,grid.temporal_grid.t,v_Jn)
    plt.xlabel(r'$x$')
    plt.ylabel(r'$t$')
    plt.colorbar()
```

```
    plt.show()

def nagumo_visualize_3D(grid,v_Jn):
    X_J, t = np.meshgrid(grid.spatial_grid.X_J, grid.temporal_grid.t)
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    surf = ax.plot_surface(X_J, t, v_Jn, rstride=1,
                           cstride=1, cmap=cm.jet, linewidth=0.02)

    ax.set_xlabel(r'$x$')
    ax.set_ylabel(r'$t$')
    ax.set_zlabel(r'$v(t,x)$')
    ax.set_zlim(0, 1.5)
    #fig.colorbar(surf, shrink=0.5, aspect=5)
    plt.show()

def nagumo_visualize_potential_change_rate(grid,c):
    plt.plot(grid.temporal_grid.t, c)
    plt.xlabel(r'$t$')
    plt.ylabel(r'temporal rate of change in potential at x = 0')
    plt.show()

def nagumo_visualize_signal_fixed_location(grid,s):
    plt.plot(grid.temporal_grid.t, s)
    plt.xlabel(r'$t$')
    plt.ylabel('potential at x = 0')
    plt.show()

def nagumo_visualize_signal_fixed_time(grid,s):
    plt.plot(grid.spatial_grid.X_J, s)
    plt.xlabel(r'$x$')
    plt.ylabel(r'solution $v(t_n,x)$')
    plt.show()

def nagumo_visualize_wave_speed(temp_grid,c_rel,
                                c_sample,c_mean,c_std,M):
    # set up plot
    fig = plt.figure()

    # single sample path
    tlim = temp_grid.temp_domain

    max1 = np.abs(c_sample - c_rel).max()
    max2 = np.abs(c_mean - c_rel).max()
```

```
    max12 = max(max1, max2)

    # plot single sample path, c_sample
    ax1 = plt.axes(xlim=tlim,
                   ylim=(c_rel - 1. * max12, c_rel + 2. * max12))
    line1, = ax1.plot([], [], c='r',
                      label=r'$sample\, path$')
    line1.set_data(temp_grid.t, c_sample)

    # plot c_mean
    ax2 = plt.axes(xlim=tlim,
                   ylim=(c_rel - 1. * max12, c_rel + 2. * max12))

    line2, = ax2.plot([], [], c='k',
                      label=r'$mean\, of\,'+str(M)+'\, samples$')
    line2.set_data(temp_grid.t, c_mean)

    plt.fill_between(temp_grid.t, c_mean-2*c_std, c_mean+2*c_std,
                     color='b', alpha=0.1)

    ax2.set_xlabel('$t$')
    ax2.set_ylabel('$C(t)/t$')
    ax2.axhline(c_rel, c='k', ls=':', label=r'$c_{rel}=c_{det}-c_{stoch}$')
    ax2.legend(prop=dict(size=12))

    plt.show()

def nagumo_visualize_mean_crel(alpha,mean,c_rel,M,xlim,ylim):
    # set up plot
    fig = plt.figure()

    # alpha
    alpha_lim = xlim

    # c_rel
    c_rel_lim = ylim

    # plot true c_rel
    ax1 = plt.axes(xlim=alpha_lim, ylim=c_rel_lim)
    line1, = ax1.plot([], [], c='r',
                      label=r'$c_{rel}(\alpha)$')
    line1.set_data(alpha, c_rel)

    # plot mean
```

```
    ax2 = plt.axes(xlim=alpha_lim, ylim=c_rel_lim)
    line2, = ax2.plot([], [], c='k',
                      label=r'$mean\, of\,'+str(M)+'\, samples$')
    line2.set_data(alpha, mean)

    ax2.set_xlabel('$\alpha$')
    ax2.set_ylabel('$c_{rel}(\alpha), \, mean$')
    ax2.legend(prop=dict(size=12))

    plt.show()
```

The animations are computed by

Listing 3.8: Nagumo_animation.py

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.animation import FuncAnimation


class NagumoAnimation(FuncAnimation):

    def __init__(self,spde_nagumo,v_Jn,wave_speed,plot_y_range):

        # set Nagumo equation
        self.spde_nagumo = spde_nagumo

        # set grid
        self.grid = spde_nagumo.st_grid

        # set numerical solution
        self.v_Jn = v_Jn

        # set wave speeds
        self.wave_speed = wave_speed

        # set plotting ranges
        self.plot_y_range = np.asarray(plot_y_range)

        self.xlim = self.grid.spatial_grid.spatial_domain
        self.tlim = self.grid.temporal_grid.temp_domain

        self.yc_min = self.plot_y_range[0,0]
        self.yc_max = self.plot_y_range[0,1]
```

```
32          self.yv_min = self.plot_y_range[1,0]
            self.yv_max = self.plot_y_range[1,1]
34
            # y-ranges
36          self.c_range = (self.yc_min -5. * (self.yc_max - self.yc_min),
                            self.yc_max + 20. * (self.yc_max - self.yc_min))
38          self.v_range = (self.yv_min, self.yv_max)

40          # set figure
            self.fig = plt.figure()
42
            # set upper sub-plot (for wave speed)
44          self.ax1 = self.fig.add_subplot(211, xlim = self.tlim,
                                            ylim = self.c_range)
46          self.ax1.axhline(spde_nagumo.c_stoch, c='k', ls = ':',
                             label=r"$c=\sqrt{2b\nu}\left(\frac{1}{2} - a\right)$")
48          self.ax1.set_xlabel('$t$')
            self.ax1.set_ylabel('$c(t)$')
50
            # initial data
52          self.c_line, = self.ax1.plot([], [], c='r',
                                         label='$c(t)$')
54
            # set bottom sub-plot (for TW)
56          self.ax2 = self.fig.add_subplot(212, xlim = self.xlim,
                                            ylim = self.v_range)
58          self.ax2.set_xlabel('$x$')
            self.ax2.set_ylabel('$v(t,x)$')
60
            # initial data
62          self.wave_line, = self.ax2.plot([], [], c='r',
                                            label='$v(t,x)$')
64
            # set title
66          self.title = self.ax1.set_title("")

68          # set legend
            self.legend = self.ax1.legend(prop=dict(size=12))
70

72      # initiator; defines base frame for animation
        def init(self):
74          self.title.set_text("")
```

```
        self.c_line.set_data([], [])
        self.wave_line.set_data([], [])
        return (self.c_line,
                self.wave_line,
                self.title)


    # animator; updates animation sequentially
    # INPUT: n - frame number
    def animate(self,n):
        self.title.set_text("t = %.2f" % self.grid.temporal_grid.t[n])
        self.c_line.set_data(self.grid.temporal_grid.t[:n],
                             self.wave_speed[:n])
        self.wave_line.set_data(self.grid.spatial_grid.X_J,
                                self.v_Jn[:,n])
        return (self.c_line,
                self.wave_line,
                self.title)


    # run animation
    def run_save_animation(self):
        anim = animation.FuncAnimation(self.fig,
                                       self.animate,
                                       init_func = self.init,
                         frames = self.grid.temporal_grid.t.shape[0],
                         interval = 20, blit = True, repeat=False)

        anim.save('wave_animation_nagumo6.mp4',
                  fps = None,
                  extra_args=['-vcodec', 'libx264'])
        plt.show()
```

3.3.4 Main-Program

All previous class and method definitions are combined in

Listing 3.9: Nagumo_main.py

```
import numpy as np

import Nagumo_model as nm
import Nagumo_helpers as nh
import fcts_semi_implicit_em as nfd
```

```
7 import Nagumo_animation as na
  import Nagumo_visualization as nv
9

11 if __name__ == "__main__":

13     # PARAMETERS
       # model PARAMETERS
15     a = 0.1
       b = 1.
17     nu = 10.
       alpha = 0.2
19
       # SIMULATION PARAMETERS
21     #L = 100
       L = 75.## wave speed plot, 80
23     T = 50.
       J = 2000. #J = 50000.## wave speed plot
25     #J = 5000
       N = 600. #1000
27
       # two sided spatial domain
29     spatial_two_sided = True

31     # boundary condition
       BC = 'n'
33
       ##########################################################################
35     ##################### SETTING INITIAL VALUES ##############################
       ##########################################################################
37
       # perturbed TW
39     # pert = 0.05; for animation
       pert = 0.05; location_TW = 73.
41     pert_TW = nm.NagumoInitialValueTW(pert, location_TW)

43     # Gaussian pulse
       pert_pulse = 1e-2; location_pulse = 0.5
45     pulse = nm.NagumoInitialValuePulse(pert_pulse, location_pulse)

47     # Kink
       c_1 = 1; c_2 = 5.
49     kink = nm.NagumoInitialValueKink(c_1, c_2)
```

```
51  ##########################################################################
    ######################## SETTING NAGUMO EQUATION ########################
53  ##########################################################################

55  # INITIAL VALUE: perturbed TW
    spde_nagumo_TW = nm.SPDE_FD_Nagumo(nu,a,b,alpha,pert_TW,BC,
57                                     L,J,T,N,spatial_two_sided)

59  # INITIAL VALUE: Gaussian pulse
    spde_nagumo_pulse = nm.SPDE_FD_Nagumo(nu,a,b,alpha,pulse,BC,
61                                       L,J,T,N,spatial_two_sided)

63  # INITIAL VALUE: Kink
    spde_nagumo_kink = nm.SPDE_FD_Nagumo(nu,a,b,alpha,kink,BC,
65                                       L,J,T,N,spatial_two_sided)

67

    ##########################################################################
69  ############################ SIMULATIONS #################################
    ##########################################################################
71
    # simulation with perturbed TW
73  fd_grid, v_Jn = nfd.fd_si_em_simul(spde_nagumo_TW)

75  #simulation with Gaussian pulse
    #fd_grid, v_Jn = nfd.fd_si_em_simul(spde_nagumo_pulse)
77
    # simulation with kink
79  #fd_grid, v_Jn = nfd.fd_si_em_simul(spde_nagumo_kink)

81

    ##########################################################################
83  ########################## VISUALIZATION #################################
    ##########################################################################
85
    # visualization in 2D-plot
87  #nv.nagumo_visualize_2D(fd_grid, np.transpose(v_Jn))

89  # visualization in 3D-plot
    #nv.nagumo_visualize_3D(fd_grid, np.transpose(v_Jn))
91
    # visualize wave speed
93  #c = nh.nagumo_potential_change_rate(fd_grid, v_Jn)
    #nv.nagumo_visualize_potential_change_rate(fd_grid,c)
```

```
95
    # visualize potential
97  #s = nh.nagumo_signal_fixed_location(fd_grid, v_Jn)
    #nv.nagumo_visualize_signal_fixed_location(fd_grid, s)
99
    # plot solution
101 #s = nh.nagumo_signal_fixed_time(fd_grid, v_Jn, 70)
    #nv.nagumo_visualize_signal_fixed_time(fd_grid, s)
103
    # computing wave speed
105 #c_n = nh.nagumo_compute_wave_speed(fd_grid, v_Jn)

107 # animate wave
    #nagumo_animation = na.NagumoAnimation(spde_nagumo_TW, v_Jn, c_n,
109 #                                      [[0,spde_nagumo_TW.c_stoch],[0,1.5]]
    #                                      )
111 #nagumo_animation.run_save_animation()
```

Chapter 4

Numerical Investigation of the Wave Speed

In this chapter we discuss Implementations for the numerics in [2, Chapter 5] concerning the speed of different travelling waves in equation (3.1).

4.1 A Random ODE for Phase Adaption

For the reader's convenience, we recall the main issue and numerical discretization from [2]. Recall the stochastic process $(\hat{v}(t,x))_{t\geq 0}$, for $x \in \mathbb{R}$, defined by

$$\hat{v}(t,x) := v^{TW}\left(x + ct + \sqrt{2b\nu}\alpha\beta(t)\right), \tag{4.1}$$

for $c = \sqrt{2b\nu}\left(\frac{1}{2} - a\right)$ with parameters a, b and ν as above. As discussed in [2, Chapter 4, Proposition 4.1.1.], $\hat{v}$ may be viewed as an explicitly given stochastic travelling wave solution to (3.1). The deterministic version of Nagumo's equation, as well, exhibits a travelling wave solution $\bar{v}$ given by

$$\bar{v}(t,x) := \bar{v}^{TW}(x + \bar{c}t), \tag{4.2}$$

with

$$\bar{v}^{TW}(x) = \frac{1}{1 + e^{-\bar{k}x}}, \quad \bar{k} = \sqrt{\frac{b}{2\bar{\nu}}}, \tag{4.3}$$

and $\bar{c} = \sqrt{2b\bar{\nu}}(\frac{1}{2} - a) = c\sqrt{1 + \alpha^2 b}$. For an $m > 0$, we consider

$$\dot{C}(t,\omega) = m\langle \bar{v}_x(t, \cdot + C(t,\omega)), \hat{v}(t,\cdot,\omega) - \bar{v}(t, \cdot + C(t,\omega))\rangle_H, \tag{4.4}$$

$C(0) = 0$, or

$$\dot{C}(t,\omega) = m\langle \bar{v}_x(t, \cdot + C(t,\omega)), v^{TW}(\cdot + ct + \sqrt{2b\nu}\alpha\beta(t,\omega)) - \bar{v}(t, \cdot + C(t,\omega))\rangle_H, \tag{4.5}$$

$C(0) = 0$. By a substitution, we have

$$\bar{f}(C,t,\omega) := \int_{\mathbb{R}} \Big[v^{TW}(x + c_{rel}t + \sqrt{2b\nu}\alpha\beta(t,\omega) - C(t,\omega)) - \bar{v}^{TW}(x) \Big] \bar{v}_x^{TW}(x)\, dx, \quad (4.6)$$

with $c_{rel} := c - \bar{c}$, for every $t \geq 0$. Numerical solutions to this random ordinary differential equations are obtained by Heun's method outlined in the following.

4.2 Approximation

Let $T > 0$ and $N \in \mathbb{N}$, set $\Delta t := \frac{T}{N}$ and consider $t_n = n\Delta t$, $n = 0, \ldots, N$. Equation (4.5) is approximated by

$$\begin{aligned} \tilde{C}_{n+1} &= C_n + \Delta t\, \bar{f}(C_n, t_n, \omega), \\ C_{n+1} &= C_n + \frac{\Delta t}{2} \left(\bar{f}(C_n, t_n, \omega) + \bar{f}(\tilde{C}_{n+1}, t_{n+1}, \omega) \right), \end{aligned} \quad (4.7)$$

with $C_0 = 0$, for $\omega \in \Omega$ and $n = 0, \ldots, N-1$.

For the discussion of simulation results, we refer to [2].

4.3 Implementation

Here, we follow the same approach as in Chapter 3. Thus, the following listing provides the implementation of (4.5) as the class **WaveSpeedODE**.

Listing 4.1: `classes_for_wave_speed.py`

```
import numpy as np
from classes_for_grid_handling import SpatialGrid

class WaveSpeedODE(object):
    def __init__(self,
                 a,b,nu,alpha,
                 L,J,sp_two_sided=True):

        # set parameters
        self.a = a
        self.b = b
        self.nu = nu
        self.alpha = alpha
        self.nu_bar = (self.nu
                       * (1. + self.b * self.alpha**2))

```

```
        # WAVE SPEED of deterministic wave
        self.c_det = (np.sqrt(2. * self.b * self.nu)
                      * (0.5 - self.a))

        # WAVE SPEED of stochastic wave
        self.c_stoch = (np.sqrt(2. * self.b * self.nu_bar)
                        * (0.5 - self.a))

        # relative WAVE SPEED
        self.c_rel = self.c_det - self.c_stoch

        # WAVE PROFILES
        self.det_wp = TWProfil(self.nu, self.b)
        self.stoch_wp = TWProfil(self.nu_bar, self.b)

        # sample path
        self._sample_path = None

        # temporal grid
        self._temp_grid = None

        # spatial grid
        self.spatial_grid = SpatialGrid(L, J, sp_two_sided)

        # set initial value
        self.initial_value = 0.

        # set dimension
        self.dim = 1

    def _c_stoch_wave(self,t,n):
        drift = self.c_rel * t
        diff = (np.sqrt(2. * self.b * self.nu) * self.alpha
                * self.sample_path[n])
        return drift + diff

    def f(self,C,t,n):
        c = self._c_stoch_wave(t,n)
        v1 = self.det_wp.eval_tw(self.spatial_grid.X_J + c - C)
        v2 = self.stoch_wp.eval_tw_derivative(self.spatial_grid.X_J)
        v3 = self.stoch_wp.eval_tw(self.spatial_grid.X_J)

        v1 = v1 - v3

```

```
        d1 = np.dot(v1[1:], v2[1:])
        d2 = np.dot(v1[:-1], v2[:-1])
        return (d1 + d2) * self.spatial_grid.h / 2.

    def generate_bm_sample_path(self):
        dt = self.temp_grid.delta_t
        t = self.temp_grid.t
        dW = np.random.randn(t.shape[0],)
        dW[0] = 0
        dW = np.sqrt(dt) * dW
        return np.cumsum(dW)

    def generate_sample_path(self,process):
        if process == 'bm':
            W = self.generate_bm_sample_path()
        else:
            raise ValueError("Input needs to be 'bm'")
        return W

    def _get_sample_path(self):
        return self._sample_path

    def _set_sample_path(self, process):
        W = self.generate_sample_path(process)
        self._sample_path = W

    def _get_temp_grid(self):
        return self._temp_grid

    def _set_temp_grid(self, grid):
        self._temp_grid = grid

    sample_path = property(_get_sample_path, _set_sample_path)
    temp_grid = property(_get_temp_grid, _set_temp_grid)


class TWProfil:
    def __init__(self,nu,b):
        self.k = np.sqrt(b / (2. * nu))

    def eval_tw(self,x):
        return 1./(1 + np.exp(-self.k * x))

    def eval_tw_derivative(self,x):
```

```
        return self.k * self.eval_tw(x) * (1. - self.eval_tw(x))
```

The next listing contains an implementation of Heun's method (4.7) above, as well as implementations of the trapezium method, the explicit and implicit Euler methods. Nevertheless, only the former is applied to solving (4.5).

Listing 4.2: Nagumo_ode_solvers.py

```
import numpy as np
import scipy.optimize as optimize

from classes_for_grid_handling import TemporalGrid


##########################################################################
########################## HEUN'S METHOD #################################
##########################################################################
def heun_single_step(u_n,ode,delta_t,add_param=()):
    '''
    ############################ INPUT ############################
    add_param[0]     : t_n, current time step
    add_param[1]     : t_{n+1} subsequent time step
    add_param[2]     : n, index of time step

    ###############################################################
    '''
    u_tilde = u_n + delta_t * ode.f(u_n,add_param[0],add_param[2])
    u_n = (0.5 * u_n + 0.5 * (u_tilde + delta_t *
                              ode.f(u_tilde,add_param[1],add_param[2])))
    return u_n

def nagumo_solve_c_heun(ode,T,N,M):
    '''
    ############################ INPUT ############################
    ode              : instance of class WaveSpeedODE
    T                : float, final time
    N                : float or integer, number of time steps
    M                : integer, number of sample paths

    ###############################################################
    '''
    temp_grid = TemporalGrid(T,N)

    ode.temp_grid = temp_grid
```

```
37
    u = np.zeros((temp_grid.t.shape[0], M+2))
39  c = np.zeros((temp_grid.t.shape[0], M+2))
    u[0,:] = ode.initial_value
41  c[0,:] = ode.initial_value

43  for m in xrange(M):
        u_n = u[0,m]
45      ode.sample_path = 'bm'
        for n in xrange(temp_grid.t.shape[0] - 1):
47          u[n+1,m] = heun_single_step(u_n,ode,temp_grid.delta_t,
                                        (temp_grid.t[n],temp_grid.t[n+1],n))
49          u_n = u[n+1,m]
        c[1:,m] = u[1:,m] / temp_grid.t[1:]
51
    # compute row mean
53  c[1:,-2] = np.mean(c[1:,:M], axis=1)

55  # compute row standard deviation
    c[1:,-1] = np.std(c[1:,:M], axis=1)
57  return temp_grid, u, c


59
def nagumo_solve
61


63 ##############################################################################
   ########################## EXPLICIT EULER ####################################
65 ##############################################################################
def explicit_euler(ode,T,N):
67  temp_grid = TemporalGrid(T,N)
    u = np.zeros((ode.f.dim, temp_grid.t.shape[0]))
69
    u[:,0] = ode.initial_value
71  u_n = u[:,0]
    for n in xrange(N):
73      u[:,n+1] = u_n + temp_grid.delta_t * ode.f(u_n)
        u_n = u[:,n+1]
75  return temp_grid, np.squeeze(u)

77 def explicit_euler_single_step(u_n,ode,delta_t,t_n,n):
    u_n = u_n + delta_t * ode.f(u_n, t_n, n)
79  return u_n
```

```
81 def nagumo_solve_c_expl_euler(ode,T,N):

83     temp_grid = TemporalGrid(T,N)

85     ode.temp_grid = temp_grid
       ode.sample_path ='bm'
87
       u = np.zeros((ode.dim, temp_grid.t.shape[0]))
89
       u[:,0] = ode.initial_value
91     u_n = u[:,0]
       for n in xrange(N):
93         u[:,n+1] = explicit_euler_single_step(u_n, ode, temp_grid.delta_t,
                                                 temp_grid.t[n], n)
95         u_n = u[:,n+1]
       return temp_grid, np.squeeze(u)
97

99 ##########################################################################
   ########################## IMPLICIT EULER ################################
101 ##########################################################################
   def imp_f(y,x,delta_t,f):
103    return (y - x) - delta_t * f(y)

105 def implicit_euler(ode,T,N):
       temp_grid = TemporalGrid(T,N)
107    u = np.zeros((ode.dim, temp_grid.t.shape[0]))

109    u[:,0] = ode.initial_value
       u_n = u[:,0]
111    for n in xrange(N):
           f_iter = imp_f(x=u_n, delta_t=temp_grid.delta_t, f=ode.f)
113        u[:,n+1] = optimize.fsolve(f_iter, u_n)
           u_n = u[:,n+1]
115    return temp_grid, np.squeeze(u)

117 def imp_f_time_dep(y,x,delta_t,t,n,ode):
       (y-x)-delta_t * ode.f(y,t,n)
119
   def imp_euler_single_step(u_n,ode,dt,t_n,n):
121    f_iter = imp_f_time_dep
       u_n = optimize.fsolve(f_iter, u_n, (u_n,dt,t_n,n,ode))
123    return u_n
```

```
125 def nagumo_solve_c_imp_euler(ode,T,N):
        temp_grid = TemporalGrid(T,N)
127
        ode.temp_grid = temp_grid
129     ode.sample_path ='bm'

131     u = np.zeros((ode.dim, temp_grid.t.shape[0]))

133     u[:,0] = ode.initial_value
        u_n = u[:,0]
135     for n in xrange(N):
            u[:,n+1] = imp_euler_single_step(u_n,ode,temp_grid.delta_t,
137                                              temp_grid.t[n],n)
            u_n = u[:,n+1]
139     return temp_grid, np.squeeze(u)


141
    ##############################################################################
143 ############################ TRAPEZIUM METHOD ################################
    ##############################################################################
145 def trapezium_f(x,delta_t,f,y):
        return (y - x) - 0.5 * delta_t * (f(y) + f(x))
147
    def trapezium_euler(ode,T,N):
149     temp_grid = TemporalGrid(T,N)
        u = np.zeros((ode.dim, temp_grid.t.shape[0]))
151
        u[:,0] = ode.initial_value
153     u_n = u[:,0]
        for n in xrange(N):
155         f_iter = trapezium_f(x=u_n, delta_t=temp_grid.delta_t, f=ode.f)
            u[:,n+1] = optimize.fsolve(f_iter, u_n)
157         u_n = u[:,n+1]
        return temp_grid, np.squeeze(u)
```

Simulations are performed in

Listing 4.3: Nagumo_wave_speed_main.py

```
1 import numpy as np

3 import Nagumo_visualization as nv
  import classes_for_wave_speed as ws
5 import Nagumo_ode_solvers as ode_solve
```

```
7
if __name__ == "__main__":
9
    # PARAMETERS
11  # model PARAMETERS
    a = 0.01
13  b = 1.
    nu = 10.
15  alpha = 0.2

17  # simulation PARAMETERS
    L = 20.
19  J = 2000.
    T = 1000.
21  N = 10000.
    # tested parameter for plot
23  #N = 320000.
    #T = 20000.
25
    # two sided spatial domain
27  spatial_two_sided = True

29  # boundary condition
    BC = 'n'
31

33  ##########################################################################
    ##################### WAVE SPEED VISUALIZATION ###########################
35  ##########################################################################

37  # WAVE SPEED ODE
    wave_speed_ode = ws.WaveSpeedODE(a, b, nu, alpha, L, J,
39                                   spatial_two_sided)

41  print wave_speed_ode.c_rel

43  # number of samples
    M = 100
45
    ##########################################################################
47  ############ solve ODE for C using Heun's method #########################
    ##########################################################################
49  temp_grid, loc, c = ode_solve.nagumo_solve_c_heun(wave_speed_ode,
                                                      T, N, M)
```

```
51
    # visualize result for C
53  c_rel = wave_speed_ode.c_rel
    c_sample = c[:,0]
55  c_mean = c[:,-2]
    c_std = c[:,-1]
57  nv.nagumo_visualize_wave_speed(temp_grid, c_rel, c_sample,
                                   c_mean, c_std, M)
59
    ##########################################################################
61  ########## solve ODE for C using explicit Euler ##########################
    ##########################################################################
63  #temp_grid, C = nagumo_solve_c_expl_euler(wave_speed_ode, T, N)

65  # visualize result for C
    #nv.nagumo_wave_speed_visualize(temp_grid, C / temp_grid.t)
67

69  ##########################################################################
    ######### solve ODE for C using implicit Euler ###########################
71  ##########################################################################
    #temp_grid, C = nagumo_solve_c_imp_euler(wave_speed_ode, T, N)
73
    #visualize result for C
75  #nv.nnagumo_wave_speed_visualize(temp_grid, C )

77
    ##########################################################################
79  ################## solve for various alpha ###############################
    ##########################################################################
81  alpha = np.linspace(0, 1, 10)
```

Chapter 5

Implementation of Stochastic Hodgkin-Huxley Equations

In this Chapter, we present the implementation of stochastic Hodgkin-Huxley equations. Although, more subtle questions concerning the existence and uniqueness of solutions, as well as spatial and temporal approximations are largely omitted here, we will state the equations in concern.

5.1 Stochastic Hodgkin-Huxley Equations

Citing [2], we treat

$$\begin{aligned}
\tau dU(t) = &\Big[\lambda^2 AU(t) - \bar{g}_{Na} m(t)^3 h(t)\,(U(t) - E_{Na}) \\
&\qquad - \bar{g}_K n(t)^4\,(U(t) - E_K) - g_L\,(U(t) - E_L)\Big]\,dt + B\,dW(t), \\
dn(t) = &\Big[\alpha_n(U(t))(1 - n(t)) - \beta_n(U(t))n(t)\Big]\,dt \\
&\qquad + \sigma_n\,\mathbf{1}_{\{0\le n(t)\le 1\}} n(t)\,(1 - n(t))\,B_n\,dW_n(t), \\
dm(t) = &\Big[\alpha_m(U(t))(1 - m(t)) - \beta_m(U(t))m(t)\Big]\,dt \\
&\qquad + \sigma_m\,\mathbf{1}_{\{0\le m(t)\le 1\}} m(t)\,(1 - m(t))\,B_m\,dW_m(t), \\
dh(t) = &\Big[\alpha_h(U(t))(1 - h(t)) - \beta_h(U(t))h(t)\Big]\,dt \\
&\qquad + \sigma_h\,\mathbf{1}_{\{0\le h(t)\le 1\}} h(t)\,(1 - h(t))\,B_h\,dW_h(t), \qquad (5.1)
\end{aligned}$$

for W, W_n, W_m and W_h cylindrical Wiener processes on $H = L^2(\mathcal{O})$, and $\sigma_n, \sigma_m, \sigma_h \in [0,1]$ controlling the noise amplitude. Furthermore, $B, B_n, B_m, B_h \in L_2(H,V)$, with respective integral kernels $b, b_n, b_m, b_h \in H^1(\mathcal{O}\times\mathcal{O})$. The parameters τ and λ are time, respectively space constants. Finally, $\bar{g}_{Na}, \bar{g}_K, g_L > 0$ are membrane conductances, and $E_{Na}, E_K, E_L \in \mathbb{R}$ are resting potentials.

Using finite differences in space, as well as a semi-implicit Euler-Maruyama scheme for U and a theta-Euler-Maruyama method for n, m, h, a fully dis-

crete system can be obtained. For a detailed derivation of that matter, we refer the reader to [2].

5.2 Implementation

5.2.1 Numerical Realization of the Hodgkin-Huxley Equations

As in the previous section, we first describe an implementation of (5.1).

Listing 5.1: `HH_model.py`

```
import numpy as np

import classes_for_spde_handling as spde
import classes_for_grid_handling as grh

import HH_drift_diff_coeff as hhdd
import fcts_helpers as fhelp
import HH_noise as hhn


class HH_U(spde.SPDE_FD):
    '''
    ###################### PURPOSE: ##########################
    Implements numerical version of the equation for U in
    the stochastic Hodgkin–Huxley model

    ###################### Attributes: #######################
    '''
    def __init__(self,
                 param_U,
                 param_disc,
                 BC,
                 add_noise_type,
                 sigma,
                 U_0,
                 I,
                 add_noise_param=None):

        # SPDEFiniteDifference PARAMETERS
        f = hhdd.HH_U_DriftCoeff(param_U,I)
        spde.SPDE_FD.__init__(self,
                              param_U['diff_param'],
                              f,
```

```
                                None,
                                U_0,
                                BC,
                                param_disc['L'],
                                param_disc['J'],
                                param_disc['T'],
                                param_disc['N'],
                                param_disc['spatial_two_sided'])

        # set NOISE
        self.multiplicative_noise = False
        if add_noise_type == 1:
            self.sigma = hhn.AddNoise1(sigma,
                                       self.st_grid.spatial_grid.X_J.shape[0])
        elif add_noise_type == 2:
            Cov = fhelp.compute_exp_cov(self.st_grid.spatial_grid.h,
                                       self.st_grid.spatial_grid.X_J.shape[0],
                                       add_noise_param['l'])
            self.sigma = hhn.AddNoise2(sigma, Cov)
            print self.sigma.B
        elif add_noise_type == 3:
            pass

        # set Wiener process
        self.wp = np.random.randn(self.st_grid.spatial_grid.X_J.shape[0],
                                  self.st_grid.temporal_grid.t.shape[0])

class HH_X(spde.SPDE_FD):
    '''
    ###################### PURPOSE: ##########################
    Implements a numerical version of the equations for
    x=n,m,h in the stochastic Hodgkin—Huxley model

    ##########################################################
    '''
    def __init__(self,
                 param_x,
                 param_disc,
                 param_gen_env,
                 mult_noise_type,
                 sigma,
                 x_0):

        # SPDEFiniteDifference PARAMETERS
```

```
78          f = hhdd.HH_X_DriftCoeff(param_x,param_gen_env)
            spde.SPDE_FD.__init__(self,
80                                None,
                                  f,
82                                None, # sigma
                                  x_0,
84                                None,
                                  param_disc['L'],
86                                param_disc['J'],
                                  param_disc['T'],
88                                param_disc['N'],
                                  param_disc['spatial_two_sided'])
90
            if mult_noise_type == 1:
92              self.sigma = hhdd.HH_X_DiffusionCoeff(sigma,
                                      self.st_grid.spatial_grid.X_J.shape[0],
94                                    self.st_grid.spatial_grid.spatial_domain[1],
                                      param_disc['spatial_two_sided'])
96          elif mult_noise_type == 2:
                self.sigma = hhdd.HH_X_DiffusionCoeff_2(sigma)
98          elif mult_noise_type == 3:
                self.sigma = hhdd.HH_X_DiffusionCoeff_3(sigma,
100                                   self.st_grid.spatial_grid.h,
                                      self.st_grid.spatial_grid.X_J.shape[0],
102                                   0.05)

104         # set realization of discrete Wiener process
            self.wp = np.random.randn(self.st_grid.spatial_grid.X_J.shape[0],
106                                   self.st_grid.temporal_grid.t.shape[0])


108


110 class HH_Coupled:
        '''
112     ###################### PURPOSE: ##########################
        Implements numerical version of the coupled system of
114     equations for U and X=(n,m,h) in the stochastic Hodgkin-
        Huxley model
116
        ##########################################################
118     '''
        # Equation for U
120     U = None
```

```
122     # Equations for n,m,h
        n = None
124     m = None
        h = None
126
        # grid
128     st_grid = None

130     def __init__(self,
                     hh_param,
132                  add_noise_type,
                     mult_noise_type,
134                  U_0,
                     n_0,
136                  m_0,
                     h_0,
138                  I,
                     add_noise_param=None):
140
            # set global grid
142         pd = hh_param.param_disc
            self.st_grid = grh.SpatioTemporalGrid(pd['L'],
144                                               pd['spatial_two_sided'],
                                                  pd['J'],
146                                               pd['T'],
                                                  pd['N'])
148
            # set SPDEs
150         # Equation for U
            self.U = HH_U(hh_param.param_U,
152                       hh_param.param_disc,
                          hh_param.BC,
154                       add_noise_type,
                          hh_param.param_sigma['sigma_U'],
156                       U_0,
                          I,
158                       add_noise_param)

160         # Equations for X=(n,m,h)
            self.n = HH_X(hh_param.param_n,
162                       hh_param.param_disc,
                          hh_param.param_gen_env,
164                       mult_noise_type,
                          hh_param.param_sigma['sigma_n'],
```

```
166                         n_0)

168         self.m = HH_X(hh_param.param_m,
                        hh_param.param_disc,
170                     hh_param.param_gen_env,
                        mult_noise_type,
172                     hh_param.param_sigma['sigma_m'],
                        m_0)
174
        self.h = HH_X(hh_param.param_h,
176                     hh_param.param_disc,
                        hh_param.param_gen_env,
178                     mult_noise_type,
                        hh_param.param_sigma['sigma_h'],
180                     h_0)
```

Previous code referred to implementations of drift- and diffusion coefficient specific to (5.1), which are provided by

Listing 5.2: `HH_drift_diff_coeff.py`

```
import numpy as np
2
import fcts_helpers as fhelp
4


6 ##########################################################
  ########## Drift-Coefficients ######################
8 ##########################################################

10 ############ Equation for U ############################
  class HH_U_DriftCoeff:
12     def __init__(self,param_U,I):
           # PARAMETERS for U
14         self.gNa = param_U['gNa']
           self.gK = param_U['gK']
16         self.gL = param_U['gL']
           self.ENa = param_U['ENa']
18         self.EK = param_U['EK']
           self.EL = param_U['EL']
20
           # EXITATORY SIGNAL
22         self.I = I

24     def eval(self,U,(n,m,h,t,x)):
```

```
        f = -(self.gNa*(m**3)*h*(U-self.ENa)
26              +self.gK*(n**4)*(U-self.EK)
                +self.gL*(U-self.EL)) + self.I.eval(t,x)
28      return f

30 ############ Equations for x = n,m,h ##################
class HH_X_DriftCoeff:
32    def __init__(self,param_x,param_gen_env):

34        self.param_x = param_x
          self.param_gen_env = param_gen_env
36        self.TC = param_gen_env['TC']
          self.T_base = param_gen_env['T_base']
38        self.Q_10 = param_gen_env['Q_10']
          self.phi = self.Q_10**((self.TC-self.T_base)/10.)
40
          if self.param_x['type'] == 'h':
42            self.alpha_x = AlphaH(self.param_x)
              self.beta_x = BetaH(self.param_x)
44        else:
              self.alpha_x = AlphaX(self.param_x)
46            self.beta_x = BetaX(self.param_x)

48    def eval(self,x,U):
          dx = self.phi*(self.alpha_x.eval(U) * (1. - x)
50              -self.beta_x.eval(U) * x)
          return dx
52

54 ##########################################################
########## Diffusion-Coefficients ##################
56 ##########################################################
class HH_U_DiffusionCoeff:
58    '''
      ##################### PURPOSE: ##########################
60    Wrapper-class for particular version of additive or
      multiplicative noise
62
      ##########################################################
64    '''
      def __init__(self, noise):
66        # instance of AddNoise
          self.noise = noise
68
```

```
70 ##########################################################################
   ###### HH-diffusion coefficient paper implementation #######
72 ##########################################################################
   class HH_X_DiffusionCoeff:
74     def __init__(self,sigma_x,J,L,spatial_two_sided):
           self.sigma_x = sigma_x
76         self.J = J
           if spatial_two_sided:
78             L = 2. * L
           self.L = L
80
       def eval(self,x,add_param=None):
82         s1 = (self.sigma_x * np.sqrt(self.L)
                  / 24. / np.sqrt(self.J))
84         s2 = (self.sigma_x * np.sqrt(self.L)
                  / np.sqrt(2) / 12. / np.sqrt(self.J))
86
           v1 = x[1:-1] + x[:-2]
88         v2 = x[1:-1] - x[:-2]
           v3 = 12. * x[1:-1]**2 - 8. * x[1:-1]**3
90         x1 = (v3 - 3. * v1**2 + v1**3) / v2

92         v1 = x[1:-1] + x[2:]
           v2 = x[1:-1] - x[2:]
94         x2 = (v3 - 3. * v1**2 + v1**3) / v2

96         x[0] = s2 * ((8.*x[0]**3 - 12.*x[0]**2 +
                        3.*(x[0] + x[1])**2 - (x[0] + x[1])**3)
98                      / (x[1] - x[0]))
           x[-1] = s2 * ((-8.*x[-1]**3 + 12.* x[-1]**2
100                - 3.*(x[-2] + x[-1])**2 + (x[-1] + x[-2])**3)
                        / (x[-1] - x[-2]))
102        x[1:-1] = (x1 + x2) * s1
           return x
104
       def eval_with_wp(self,x,wp,add_param=None):
106        y = self.eval(x,add_param)
           return y * wp
108

110 ##########################################################################
   ##### HH-diffusion coefficient standard fd-implementation ######
112 ##########################################################################
```

```
class HH_X_DiffusionCoeff_2:
    def __init__(self,sigma_x):
        self.sigma_x = sigma_x

    def eval(self,v,add_param=None):
        v[v<0.] = 0.
        v[v>1.] = 0.
        return self.sigma_x*v*(1.-v)

    def eval_with_wp(self,x,wp,add_param=None):
        y = self.eval(x,add_param)
        return y * wp


##########################################################################
### HH-diffusion coefficient with exponential covariance-op###
##########################################################################
class HH_X_DiffusionCoeff_3:
    def __init__(self,sigma_x,h,J,l):
        self.sigma_x = sigma_x
        self.h  = h
        self.J = J
        self.l = l
        self.B = fhelp.compute_exp_cov(self.h, self.J, self.l)

    def eval(self,v,add_param=None):
        v[v<0.] = 0.
        v[v>1.] = 0.
        return self.sigma_x*v*(1.-v)

    def eval_with_wp(self,x,wp,add_param=None):
        y = self.eval(x,add_param)
        return y * np.dot(self.B,wp)


##########################################################################
########## alpha and beta-coefficient functions in HH ##########
##########################################################################
class AlphaX(object):
    def __init__(self,param_x):
        self.ax_1 = param_x['ax_1']
        self.ax_2 = param_x['ax_2']
        self.A_x = param_x['A_x']
```

```
158     def eval(self,U,add_param=None):
            alpha = (self.ax_1 * (U + self.A_x)
160                  / (1. - np.exp(-self.ax_2 * (U + self.A_x))))
            return alpha
162

164 class BetaX(object):
        def __init__(self,param_x):
166         self.bx_1 = param_x['bx_1']
            self.bx_2 = param_x['bx_2']
168         self.B_x = param_x['B_x']

170     def eval(self,U,add_param=None):
            beta = self.bx_1 * np.exp(-self.bx_2 * (U + self.B_x))
172         return beta


174
    class AlphaH(AlphaX):
176     def __init__(self,param_x):
            AlphaX.__init__(self,param_x)
178
        def eval(self,U,add_param=None):
180         alpha = (self.ax_1 * np.exp(-self.ax_2*(U+self.A_x)))
            return alpha
182

184 class BetaH(BetaX):
        def __init__(self,param_x):
186         BetaX.__init__(self,param_x)

188     def eval(self,U,add_param=None):
            beta = (self.bx_1 /(1.+np.exp(-self.bx_2*(U+self.B_x))))
190         return beta
```

The additive noise in equation (5.1) is given by

Listing 5.3: HH_noise.py

```
1 import numpy as np

3 import classes_for_spde_handling as spde

5

    class AddNoise2(spde.AddNoise):
```

```
    '''
    ##################### PURPOSE: ############################
    Sub-class of AddNoise; implements an efficient
    evaluation of additive noise with a Wiener process

    ###########################################################
    '''
    def __init__(self,sigma,B):
        spde.AddNoise.__init__(self,sigma,B)

    # overwrite corresponding method in Base-class AddNoise
    def eval_with_wp(self,wp):
        return self.sigma*np.dot(self.B, wp)


class AddNoise1(spde.AddNoise):
    '''
    ##################### PURPOSE: ############################
    Sub-class of AddNoise; as AddNoise0

    ##################### INPUT: ##############################
    sigma   : parameter controlling noise amplitude
    J       : number of spatial grid points

    ###########################################################
    '''
    def __init__(self,sigma,J):
        B = np.ones(J,)
        spde.AddNoise.__init__(self,sigma,B)

    def eval_with_wp(self,wp):
        return self.sigma * wp.sum() * self.B
```

As the Hodgkin-Huxley equations model nerve cells, we assume the system initially in equilibrium, before it is subject to an applied current. Functionality for computing the so called Nernst resting potentials and the excitatory signal is provided by

Listing 5.4: HH_initial_values.py

```
import numpy as np
from scipy.optimize import fsolve

import HH_drift_diff_coeff as hhd
```

```
6
##########################################################################
8 ######################## INITIAL VALUES ##################################
##########################################################################
10 class RestingStates(object):
    '''
12     ###################### PURPOSE: ########################
    Computes the resting potential of the squid's large axon,
14     which will serve as initial value for subsequent
    simulations
16
    ############################################################
18     '''
    def __init__(self,hh_param):
20         # PARAMETERS
        self.hh_param = hh_param
22
        self.param_U = hh_param.param_U
24         self.param_n = hh_param.param_n
        self.param_m = hh_param.param_m
26         self.param_h = hh_param.param_h

28         # alpha_n, beta_n
        self.alpha_n = hhd.AlphaX(self.param_n)
30         self.beta_n = hhd.BetaX(self.param_n)

32         # alpha_m, beta_m
        self.alpha_m = hhd.AlphaX(self.param_m)
34         self.beta_m = hhd.BetaX(self.param_m)

36         # alpha_h, beta_h
        self.alpha_h = hhd.AlphaH(self.param_h)
38         self.beta_h = hhd.BetaH(self.param_h)

40     def compute_resting_X(self,U):
        # EQUILIBRIUM for n
42         alpha1 = self.alpha_n.eval(U)
        beta1 = self.beta_n.eval(U)
44         n_eq = alpha1 / (alpha1 + beta1)

46         # EQUILIBRIUM for m
        alpha1 = self.alpha_m.eval(U)
48         beta1 = self.beta_m.eval(U)
        m_eq = alpha1 / (alpha1 + beta1)
```

```
50
        # EQUILIBRIUM for h
52      alpha1 = self.alpha_h.eval(U)
        beta1 = self.beta_h.eval(U)
54      h_eq = alpha1 / (alpha1 + beta1)
        return (n_eq,m_eq,h_eq)
56
    def compute_Nernst_U(self,ion):
58      R = self.hh_param.phys_constants['R']
        F = self.hh_param.phys_constants['F']
60      if ion == 'K':
            ion = self.hh_param.param_gen_env['param_K']
62      elif ion == 'Na':
            ion = self.hh_param.param_gen_env['param_Na']
64      elif ion == 'Cl':
            ion = self.hh_param.param_gen_env['param_Cl']
66      T = self.hh_param.param_gen_env['T']
        C_out = ion['C_out']
68      C_in = ion['C_in']
        z = ion['z']
70      return (R*T)/(z*F)*np.log(C_out/C_in)*10**3

72  def iter_func(self,U):
        (n_eq,m_eq,h_eq) = self.compute_resting_X(U)
74      return (self.param_U['gNa']*(m_eq**3)*h_eq*(U-self.param_U['ENa'])
               +self.param_U['gK']*(n_eq**4)*(U-self.param_U['EK'])
76             +self.param_U['gL']*(U-self.param_U['EL']))

78  def compute_resting_U(self):
        np_K = self.compute_Nernst_U('K')
80      rp = fsolve(self.iter_func, np_K)
        return rp
82

84 class HHInitialEquilibrium:
    '''
86  ############# REMARK: #################
    class for initial resting potential and
88  probabilities n,m,h

90  ########################################
    '''
92  def __init__(self,eq):
        self.eq = eq
```

```
94
    def eval(self,x,add_param=None):
96      return self.eq * np.ones(len(x),)

98
##########################################################################
100 ######################## EXCITATORY SIGNAL ##############################
##########################################################################
102 class ExcitatorySignal:
    def __init__(self,start,end,pert1,pert2,loc):
104     self.start = start
        self.end = end
106     self.pert1 = pert1
        self.pert2 = pert2
108     self.loc = loc

110 def eval(self,t,x,add_param=None):
        if (t <= self.start) or (t > self.end):
112         response = 0.
        else:
114         response = self.pert1/(1.+np.exp((-self.pert2
                    *(x-self.loc)/np.sqrt(2))))
116     return response
```

To complete the numerical realization of the Hodgkin-Huxley system, we rely on an appropriate set of parameters. Those are given by the following listing.

Listing 5.5: `HH_parameters.py`

```
1 # taken from Ermentrout, Terman,
  # Mathematical Foundations of Neuroscience,
3 # Chapter 1, p. 23f.

5 # PARAMETERS for membrane potential
  # UNITS: U [mV]; gX [mS/cm^3]; EX [mV]
7
  phys_constants = {
9                  'R':8.31451,
                   'F':96485.3,
11                 }

13 param_K = {
             'C_in':400.,
15           'C_out':20.,
```

```
            'z':1.
17          }

19  param_Na = {
             'C_in':50.,
21           'C_out':440.,
             'z':1.
23           }

25  param_Cl = {
             'C_in':40.,
27           'C_out':560.,
             'z':-2.
29           }

31  param_gen_env = {
                 'T':293.2,
33               'TC':20.,
                 'T_base':6.3,
35               'Q_10':3.,
                 'param_K':param_K,
37               'param_Na':param_Na,
                 'param_Cl':param_Cl
39               }

41  param_U = {
            'tau':1.,
43          'diff_param':0.1,
            'gNa':120., 'gK':36., 'gL':0.3,
45          'ENa':50., 'EK':-77., 'EL':-54.4
            }
47
    # PARAMETERS for opening probabilities
49  # Equation: n
    param_n = {'type':'n',
51          'ax_1':0.01, 'ax_2':0.1, 'A_x':55.,
            'bx_1':0.125, 'bx_2':1/80., 'B_x':65.
53          }

55  # Equation: m
    param_m = {'type':'m',
57          'ax_1':0.1, 'ax_2':1/10., 'A_x':40.,
            'bx_1':4., 'bx_2':1/18., 'B_x':65.
59          }
```

```
# Equation: h
param_h = {'type':'h',
           'ax_1':0.07, 'ax_2':1/20., 'A_x':65.,
           'bx_1':1., 'bx_2':1/10., 'B_x':35.
           }


##########################################################
################ HH-Parameter class ######################
##########################################################
class HHData:

    # PHYSICAL CONSTANTS
    phys_constants = phys_constants

    def __init__(self,
                 param_gen_env,
                 param_U,
                 param_n,
                 param_m,
                 param_h,
                 param_sigma,
                 param_disc,
                 BC):

        # ENVIRONMENTAL CONSTANTS
        self.param_gen_env = param_gen_env

        # MODEL parameters
        self.param_U = param_U
        self.param_n = param_n
        self.param_m = param_m
        self.param_h = param_h
        self.BC = BC

        # NOISE parameters
        self.param_sigma = param_sigma

        # DISCRETIZATION parameters
        self.param_disc = param_disc
```

5.2.2 Finite Difference Scheme

Here the implementation of the numerical approximation of (5.1) taken from [2] is presented.

Listing 5.6: `HH_fd2.py`

```
import numpy as np

import fcts_semi_implicit_em as si_em
import fcts_theta_em as tem


def fd_simul_hodgkin_huxley(spde_hh,theta):
    # Equation for U
    U = spde_hh.U

    # Equations for n,m,h
    n = spde_hh.n
    m = spde_hh.m
    h = spde_hh.h

    # spatial and temporal grid
    st_grid = spde_hh.st_grid
    temporal_grid = st_grid.temporal_grid
    spatial_grid = st_grid.spatial_grid

    l = temporal_grid.t.shape[0]
    J = spatial_grid.X_J.shape[0]

    delta_t = temporal_grid.delta_t
    delta_h = spatial_grid.h

    # parameters for U
    A = U.A
    diff_param = U.diff_param

    # Wiener processes
    xi_U = U.wp
    xi_n = n.wp
    xi_m = m.wp
    xi_h = h.wp

    # compose iteration matrix
    EE = si_em.compose_iteration_matrix_fd(A, J, diff_param,
```

```
                                        delta_t, delta_h)

    # initialize
    U_Jn = np.zeros((J,l))
    U_Jn[:,0] = U.init_value.eval(spatial_grid.X_J)
    #print U_Jn[:,0]

    n_Jn = np.zeros((J,l))
    m_Jn = np.zeros((J,l))
    h_Jn = np.zeros((J,l))
    n_Jn[:,0] = n.init_value.eval(spatial_grid.X_J)
    m_Jn[:,0] = m.init_value.eval(spatial_grid.X_J)
    h_Jn[:,0] = h.init_value.eval(spatial_grid.X_J)

    for i in xrange(l-1):
        print i
        # first: one step for U
        U_Jn[:,i+1] = si_em.fd_si_em_single_step(U, U_Jn[:,i],
                                xi_U[:,i], EE,
                                add_param_f=(n_Jn[:,i],m_Jn[:,i],h_Jn[:,i],
                                             temporal_grid.t[i],
                                             spatial_grid.X_J))
        # second: one step for each n,m,h in X
        #right hand sides
        rs_n = (n_Jn[:,i]+(1.-theta)*delta_t*n.f.eval(n_Jn[:,i],U_Jn[:,i])
            +np.sqrt(delta_t)*n.sigma.eval_with_wp(n_Jn[:,i],xi_n[:,i]))

        rs_m = (m_Jn[:,i]+(1.-theta)*delta_t*m.f.eval(m_Jn[:,i],U_Jn[:,i])
            +np.sqrt(delta_t)*m.sigma.eval_with_wp(m_Jn[:,i],xi_m[:,i]))

        rs_h = (h_Jn[:,i] + (1.-theta)*delta_t*h.f.eval(h_Jn[:,i],U_Jn[:,i])
            +np.sqrt(delta_t)*h.sigma.eval_with_wp(h_Jn[:,i],xi_h[:,i]))

        n_Jn[:,i+1] = tem.theta_em_single_step(n, n_Jn[:,i],
                            (theta,delta_t,rs_n,U_Jn[:,i+1]))
        m_Jn[:,i+1] = tem.theta_em_single_step(m, m_Jn[:,i],
                            (theta,delta_t,rs_m,U_Jn[:,i+1]))
        h_Jn[:,i+1] = tem.theta_em_single_step(h, h_Jn[:,i],
                            (theta,delta_t,rs_h,U_Jn[:,i+1]))

    return st_grid, U_Jn, n_Jn, m_Jn, h_Jn
```

The corresponding implementation of the theta-Euler-Maruyama method is given by the following listing. It also contains a working implementation of

a fully implicit scheme to (5.1), which was created for bug-fixing purposes.

Listing 5.7: `fcts_theta_em.py`

```
import numpy as np
import scipy.sparse as sparse
from scipy.optimize import fsolve

import HH_drift_diff_coeff as hhd


def fiter(x,add_param,f):
    '''
    ################### PURPOSE : ###############
    composes function call for fsolve

    ################### INPUT: ##################
    x               : variable
    add_param       : length-4 tuple
      add_param[0]  : theta
      add_param[1]  : delta_t
      add_param[2]  : right hand side
      add_param[3]  : tuple of additional
                      parameters for f
    f               : drift coefficient

    ##############################################
    '''
    return (x-add_param[0]*add_param[1]*f(x,add_param[3])
            -add_param[2])

def theta_em_single_step(spde,est,add_param):
    f = spde.f.eval
    y = fsolve(fiter, est, (add_param,f))
    return y


#########################################################################
################## complete implicit solver ##################
#########################################################################
def compose_it_matrix(A,J,diff_param,delta_t,h,BC):
    # iteration matrix
    EE = sparse.eye(J) + diff_param * delta_t * A/h/h
    return EE
```

```
42
class ImpEM(object):
44    '''
      ###################### PURPOSE: ##########################
46    Fully implicit scheme for discrete Hodgkin–Huxley system

48    ############################################################
      '''
50    def __init__(self,hh_param,delta_t,h,J,BC,A):
          # PARAMETERS
52        self.hh_param = hh_param

54        self.param_U = hh_param.param_U
          self.param_n = hh_param.param_n
56        self.param_m = hh_param.param_m
          self.param_h = hh_param.param_h
58
          self.A = A
60        self.J = J
          self.BC = BC
62
          self.delta_t = delta_t
64        self.h = h

66        self.EE = compose_it_matrix(self.A,
                                      self.J,
68                                    self.param_U['diff_param'],
                                      self.delta_t,
70                                    self.h,
                                      self.BC)
72
          # alpha_n, beta_n
74        self.alpha_n = hhd.AlphaX(self.param_n)
          self.beta_n = hhd.BetaX(self.param_n)
76
          # alpha_m, beta_m
78        self.alpha_m = hhd.AlphaX(self.param_m)
          self.beta_m = hhd.BetaX(self.param_m)
80
          # alpha_h, beta_h
82        self.alpha_h = hhd.AlphaH(self.param_h)
          self.beta_h = hhd.BetaH(self.param_h)
84
      def g_n(self,U,n,m,h):
```

```
86          return (n*(1.+(self.alpha_n.eval(U)+self.beta_n.eval(U))
                        *self.delta_t)-self.delta_t*self.alpha_n.eval(U))
88
    def g_m(self,U,n,m,h):
90          return (m*(1.+(self.alpha_m.eval(U)+self.beta_m.eval(U))
                        *self.delta_t)-self.delta_t*self.alpha_m.eval(U))
92
    def g_h(self,U,n,m,h):
94          return (h*(1.+(self.alpha_h.eval(U)+self.beta_h.eval(U))
                        *self.delta_t)-self.delta_t*self.alpha_h.eval(U))
96
    def f_U(self,U,n,m,h):
98          return (self.param_U['gNa'] * (m**3) * h * (U-self.param_U['ENa'])
                    + self.param_U['gK'] * (n**4) * (U-self.param_U['EK'])
100                 + self.param_U['gL'] * (U-self.param_U['EL']))

102 def f(self,U,n,m,h):
        a = np.dot(self.EE.toarray(),U)
104     b = self.delta_t * self.f_U(U,n,m,h)
        return a + b
106
    def f_iter(self,x,rs_U,rs_n,rs_m,rs_h):
108     split = np.array_split(x,4)
        U = split[0]
110     n = split[1]
        m = split[2]
112     h = split[3]

114     rs = np.hstack([rs_U,rs_n,rs_m,rs_h])

116     U1 = self.f(U,n,m,h)
        n1 = self.g_n(U, n, m, h)
118     m1 = self.g_m(U, n, m, h)
        h1 = self.g_h(U, n, m, h)
120
        ls = np.hstack([U1,n1,m1,h1])
122
        return rs - ls
124
    def imp_em_solve(self,init_guess,rs):
126     solution = fsolve(self.f_iter,init_guess,rs)
        split = np.array_split(solution,4)
128     U = split[0]
        n = split[1]
```

```
130         m = split[2]
            h = split[3]
132         return U,n,m,h
```

5.2.3 Visualization

The following functions compute images based on simulations provided by the previous methods.

Listing 5.8: HH_visualization.py

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

4 from mpl_toolkits.mplot3d import Axes3D
  from matplotlib import cm
6 from pylab import *

8 def HH_visualize_2D(grid,v_Jn):
      plt.pcolormesh(grid.spatial_grid.X_J,grid.temporal_grid.t,v_Jn)
10    plt.xlabel(r'$x$')
      plt.ylabel(r'$t$')
12    plt.colorbar()
      plt.show()
14
  def HH_visualize_signal_fixed_time(grid,s):
16    plt.plot(grid.spatial_grid.X_J, s)
      plt.xlabel(r'$x$')
18    plt.ylabel(r'solution $v(t_n,x)$')
      plt.show()
20
  def HH_visualize_signal_fixed_location(grid,s):
22    plt.plot(grid.temporal_grid.t, s)
      plt.xlabel(r'$t$')
24    plt.ylabel('potential at x = 0')
      plt.show()
```

Animations are created by

Listing 5.9: HH_animation.py

```
1 import numpy as np
  import nice_colors as nc
3 import matplotlib.pyplot as plt
  import matplotlib.animation as animation
5 from matplotlib.animation import FuncAnimation
```

```
7
class HHAnimation(FuncAnimation):
9
    def __init__(self,spde_hh,
11                 U_Jn,n_Jn,m_Jn,h_Jn,
                   plot_y_range):
13
        # set SPDE
15      self.spde_hh = spde_hh

17      # set grid
        self.grid = spde_hh.st_grid
19
        # resting potential
21      self.rp = self.spde_hh.U.init_value.eq

23      # set numerical solution for
        # potential
25      self.U_Jn = U_Jn

27      # ion channels, probabilities
        self.n_Jn = n_Jn
29      self.m_Jn = m_Jn
        self.h_Jn = h_Jn
31
        # set wave speeds
33      #self.wave_speed = wave_speed

35      # set plotting ranges
        self.plot_y_range = np.asarray(plot_y_range)
37
        # plotting range: x-spatial variable
39      self.xlim = self.grid.spatial_grid.spatial_domain
        #self.tlim = self.grid.temporal_grid.temp_domain
41
        # plotting range: y-membrane potential
43      self.yU_min = self.plot_y_range[1,0]
        self.yU_max = self.plot_y_range[1,1]
45
        # plotting range: y-ion channels
47      self.y_ion_channels_min = self.plot_y_range[0,0]
        self.y_ion_channels_max = self.plot_y_range[0,1]
49
```

```
        # y-ranges
51      self.U_range = (self.yU_min, self.yU_max)
        self.ion_channel_range = (self.y_ion_channels_min-0.2,
53                                self.y_ion_channels_max+1.)

55      # set figure
        self.fig = plt.figure()
57
        # set upper sub-plot (opening probabilities ion channels)
59      self.ax1 = self.fig.add_subplot(211, xlim = self.xlim,
                                        ylim = self.ion_channel_range)
61      #self.ax1.set_xlabel('$x$')
        self.ax1.set_ylabel('$n(t),\,m(t),\,h(t)$')
63
        # initial data
65      self.n_line, = self.ax1.plot([], [],
                                     c=tuple(nc.colors[:,4]),
67                                   label=r'$n$')
        self.m_line, = self.ax1.plot([], [],
69                                   c=tuple(nc.colors[:,3]),
                                     label=r'$m$')
71      self.h_line, = self.ax1.plot([], [],
                                     c=tuple(nc.colors[:,1]),
73                                   label=r'$h$')

75      self.legend1 = self.ax1.legend(prop=dict(size=12))

77      # set bottom sub-plot (membrane potential)
        self.ax2 = self.fig.add_subplot(212, xlim = self.xlim,
79                                      ylim = self.U_range)
        self.ax2.axhline(self.rp, c='k', ls = ':', label=r"$resting\,pot.$")
81      self.ax2.set_xlabel('$x,\,(mm)$')
        self.ax2.set_ylabel('$U(t,x),\,(mV)$')
83
        # initial data
85      self.U_line, = self.ax2.plot([], [], c='r')

87      # set title
        self.title = self.ax1.set_title("")
89
        # set legend
91      self.legend2 = self.ax2.legend(prop=dict(size=12))

93  # initiator; defines base frame for animation
```

```
    def init(self):
95      self.title.set_text("")
        self.n_line.set_data([], [])
97      self.m_line.set_data([], [])
        self.h_line.set_data([], [])
99      self.U_line.set_data([], [])
        return (self.n_line, self.m_line,
101             self.h_line, self.U_line,
                self.title)
103
    # animator; updates animation sequentially
105 # INPUT: n - frame number
    def animate(self,n):
107     self.title.set_text("t = %.2f ms" % self.grid.temporal_grid.t[n])
        self.n_line.set_data(self.grid.spatial_grid.X_J,
109                          self.n_Jn[:,n])
        self.m_line.set_data(self.grid.spatial_grid.X_J,
111                          self.m_Jn[:,n])
        self.h_line.set_data(self.grid.spatial_grid.X_J,
113                          self.h_Jn[:,n])
        self.U_line.set_data(self.grid.spatial_grid.X_J,
115                          self.U_Jn[:,n])
        return (self.n_line, self.m_line,
117             self.h_line, self.U_line,
                self.title)
119
    # run animation
121 def run_save_animation(self):
        anim = animation.FuncAnimation(self.fig,
123                                    self.animate,
                                       init_func = self.init,
125                                frames = self.grid.temporal_grid.t.shape[0],
                                   interval = 20, blit = True, repeat=False)
127
        anim.save('hh_det5.mp4', fps = None,
129               extra_args=['-vcodec', 'libx264'])
        plt.show()
```

5.2.4 Main-Program

Finally, all previous class and method definitions are combined in

Listing 5.10: HH_main.py

```
import numpy as np

import HH_animation as hha
import HH_parameters as hhp
import HH_initial_values as hhinit
import HH_visualization as hhv
import HH_model as hhm
import HH_fd2 as hhfd


if __name__=="__main__":

    ############## DISCRETIZATION parameters ################
    L = 10.
    T = 20. # duration of action potential (ms)
    J = 500.
    N = 500

    # two sided spatial domain
    spatial_two_sided = True

    param_disc = {
                  'L':L,
                  'T':T,
                  'J':J,
                  'N':N,
                  'spatial_two_sided':spatial_two_sided
                  }

    # boundary condition
    BC = 'n'

    ################ NOISE parameters ############################
    sigma_U = 0.
    sigma_n = 0.
    sigma_m = 0.
    sigma_h = 0.

    param_sigma = {
                   'sigma_U':sigma_U,
                   'sigma_n':sigma_n,
                   'sigma_m':sigma_m,
                   'sigma_h':sigma_h
                   }
```

```
45
    # set HHData object
47  hh_param = hhp.HHData(hhp.param_gen_env,
                          hhp.param_U,
49                        hhp.param_n,
                          hhp.param_m,
51                        hhp.param_h,
                          param_sigma,
53                        param_disc,
                          BC)
55
    ##########################################################################
57  ##################### SETTING INITIAL VALUES #############################
    ##########################################################################
59  rs = hhinit.RestingStates(hh_param)
    rp = rs.compute_resting_U()
61  (n_inf,m_inf,h_inf) = rs.compute_resting_X(rp)

63  U_0 = hhinit.HHInitialEquilibrium(rp)
    n_0 = hhinit.HHInitialEquilibrium(n_inf)
65  m_0 = hhinit.HHInitialEquilibrium(m_inf)
    h_0 = hhinit.HHInitialEquilibrium(h_inf)
67
    ##########################################################################
69  ##################### SETTING EXCITATORY SIGNAL ##########################
    ##########################################################################
71  pert1 = 50.
    pert2 = .5
73  loc = L

75  # time duration of excitatory shock (ms)
    signal_start = 0.2*T
77  signal_end = 0.6*T

79  I = hhinit.ExcitatorySignal(signal_start,
                                signal_end,
81                              pert1,
                                pert2,
83                              loc)

85  ##########################################################################
    ##################### SETTING ADDITIVE NOISE #############################
87  ##########################################################################
```

```
89  # ADDITIVE NOISE for U
    add_noise_type = 1
91
    add_noise_param = {
93                    'l':.01
                      }
95
    ##########################################################################
97  ################## SETTING MULTIPLICATIVE NOISE ##########################
    ##########################################################################
99  mult_noise_type = 2


101
    ##########################################################################
103 ###################### SETTING HH EQUATION ###############################
    ##########################################################################
105 spde_hh = hhm.HH_Coupled(hh_param,
                            add_noise_type,
107                         mult_noise_type,
                            U_0,
109                         n_0,
                            m_0,
111                         h_0,
                            I,
113                         add_noise_param)

115 ##########################################################################
    ###################### SIMULATIONS #######################################
117 ##########################################################################
    theta = 1.
119
    st_grid,U_Jn,n_Jn,m_Jn,h_Jn = hhfd.fd_simul_hodgkin_huxley(spde_hh,
121                                                            theta)

123 ##########################################################################
    ###################### VISUALIZATION #####################################
125 ##########################################################################
    hhv.HH_visualize_2D(st_grid, np.transpose(U_Jn))
127 #hhv.HH_visualize_signal_fixed_time(st_grid, U_Jn[:,50])
    #hhv.HH_visualize_signal_fixed_location(st_grid, U_Jn[0,:])
129
    hhv.HH_visualize_2D(st_grid, np.transpose(n_Jn))
131 hhv.HH_visualize_2D(st_grid, np.transpose(m_Jn))
    hhv.HH_visualize_2D(st_grid, np.transpose(h_Jn))
```

```
133
    ##########################################################################
135 ######################## ANIMATION ########################################
    ##########################################################################
137 plot_ranges = [[0.,1.],[-100.,70.]]

139 hh_animation = hha.HHAnimation(spde_hh,
                                 U_Jn,
141                              n_Jn,
                                 m_Jn,
143                              h_Jn,
                                 plot_ranges)
145 hh_animation.run_save_animation()
```

Chapter 6

Conclusion

In this paper we have provided implementations of import conductance based nerve axon equations. The particular organization of the source code aims at facilitating extendability by the use of reasonable conventions and class definitions. The validity of the implementations is confirmed, at least on an empirical level, by the generated figures and animations matching the expected outcomes.

Literature

Literature

[1] Gabriel J. Lord, Catherine E. Powell, and Tony Shardlow. *An Introduction to Computational Stochastic PDEs*. 32 Avenue of the Americas, New York, NY 10013-2473, USA: Cambridge University Press, 2014.

[2] Christian Pangerl. "Stability of Stochastic Travelling Waves in Stochastic Nagumo Equations". MA thesis. Universität Augsburg, 2015.

[3] Claudia Prévôt and Michael Röckner. *A concise course on stochastic partial differential equations*. Springer, 2007.

[4] Martin Sauer and Wilhelm Stannat. "Analysis and Approximation of Stochastic Nerve Axon Equations". In: (2014).

[5] Wilhelm Stannat. "Stability of Travelling Waves in Stochastic Nagumo Equations". In: (2013).