Computational Methods in Neural Engineering - Exercise

Robin Steiner (11778873)

January 3, 2024

1 ODE solving & RC circuit

1.1 ODE solving

1.1.1

The smaller the step size the better both Euler methods approximate the analytical solution

For $tDt > 0.05$ the forward Euler method becomes instable in this case.

The instability of the forward Euler method at high step sizes stems from the fact that this method can over/under-shoot the solution. If this happens it will do the opposite on the next step, resulting in an oscilation around the actual solution. In general this overshoot can appear, because the method assumes a constant derivative along the time step, which is not the case, this mistake becomes more problematic for large time steps

1.1.2

The following code solves and plots the ODE $y' = -20y$ using the built-in solver 'RK45'.

```python
def odefun(t, y): return -20 * y
sol = integrate.solve_ivp(odefun, [0, 1], [y0], method='RK45')

[...]
ax.plot(sol.t,sol.y[0],label='RK45')
[...]

```

The resulting plot can be seen in Figure 1.

1.1.3

Next the following ODE is solved using RK45 and the backward Euler method. Again those results are compared to the analytical solution which is also found using Python.

$$y'(t) = -3y(t) + 9t, \quad y(0) = 9$$

The analytical solution as well as the forward Euler approximation are given by:

$$y(t) = 3e^{-3t} + 3t$$

and:

$$y_{n+1} = \frac{y_n + 9 \cdot t_{n+1} \cdot \text{timeStep}[n+1]}{1 + 3 \cdot \text{timeStep}[n+1]}$$

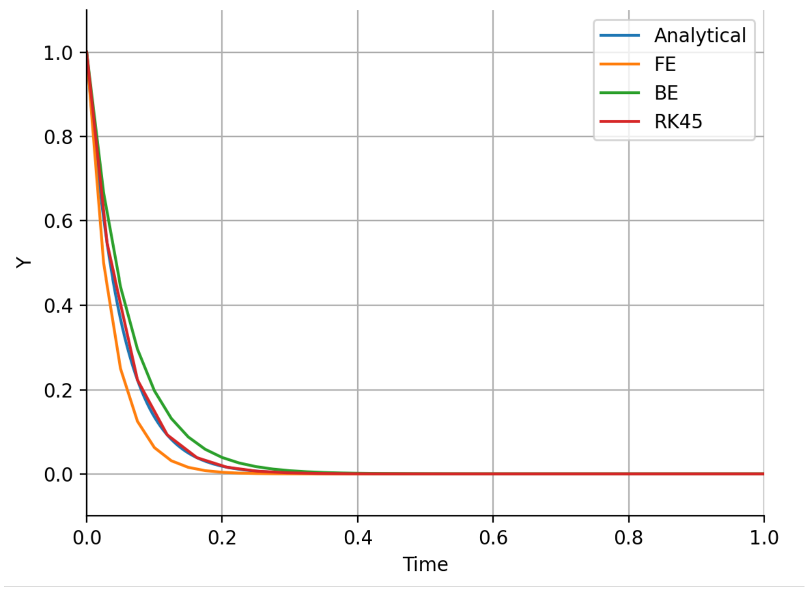

Figure 1: Plot of the solution for $y' = -20y$ using a forward and backward Euler method as well as ‘RK45’. Additionally, it shows the analytical solution

The code is given by:

```python
### ------------- ANALYTICAL SOLUTION -------------
t_sym = symbols('t')
y_sym = Function('y')
ode = Eq(y_sym(t_sym).diff(t_sym), -3*y_sym(t_sym) + 9*t_sym)
analytical_solution = dsolve(ode, y_sym(t_sym), ics={y_sym(0): y0})

### ------------- BACKWARD EULER ---------
yBEVec = np.zeros(timeSteps)  # Allocate memory
yBEVec[0] = y0  # Set initial condition
# Loop over time
for t in range(timeSteps-1):
    yBEVec[t+1] = (yBEVec[t] + 9*timeStep[t+1]*tDt) / (1 + 3*tDt)

### ------------- RK45 SOLVER ---------
def odefun(t, y): return -3 * y + 9 * t
sol = integrate.solve_ivp(odefun, [0, tStop], [y0], method='RK45')

```

The resulting plots are shown in Figure 2 and 3.

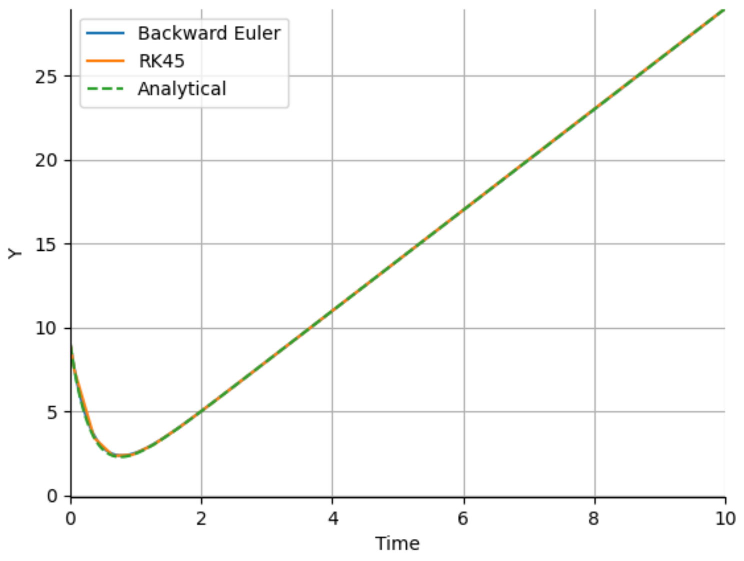

Figure 2: Plot of the solution for $y'(t) = -3y(t) + 9t$ using a backward Euler method as well as 'RK45'. Additionally, it shows the analytical solution

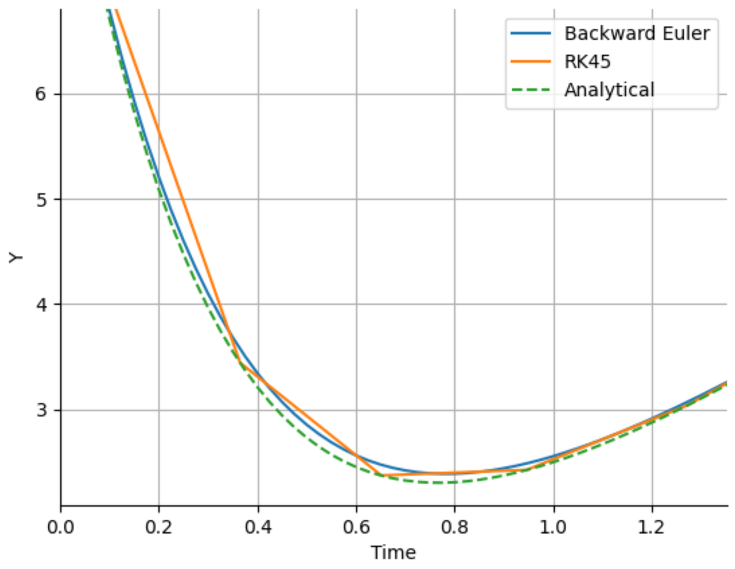

Figure 3: Zoomed in plot of the solution for $y'(t) = -3y(t) + 9t$ using a forward and backward Euler method as well as 'RK45'. Additionally, it shows the analytical solution

1.2 RC circuit / passive neuron

1.2.1

Looking at Figure 4 - 8 we can deduce the following:

- Increasing resistance (R) will lead to a higher maximal voltage.
- Increasing capacitance (C) will result in a slower charging and discharging of the capacitor.

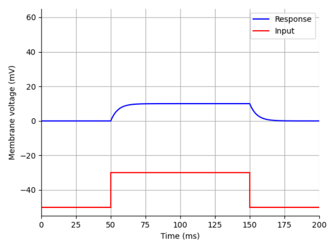

Figure 4: Response Plot for: $R = 10M\Omega, C = 0.5nF$

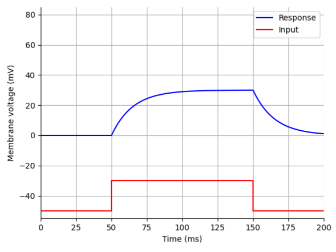

Figure 5: Response Plot for: $R = 30M\Omega, C = 0.5nF$

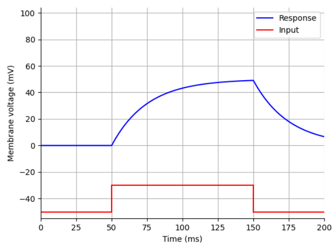

Figure 6: Response Plot for: $R = 50M\Omega, C = 0.5nF$

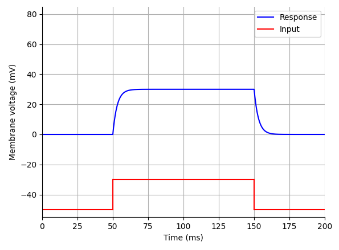

Figure 7: Response Plot for: $R = 30M\Omega, C = 0.1nF$

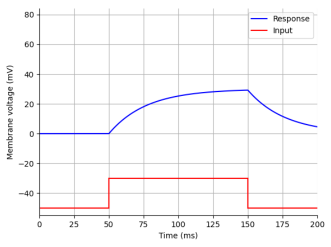

Figure 8: Response Plot for: $R = 30M\Omega, C = 0.9nF$

The time constant (τ) of an RC circuit is a measure of how quickly the circuit responds to changes. It is given by the product of resistance (R) and capacitance (C), i.e., $\tau = R * C$. More specifically it is the time it takes to charge the capacitor 63.2% of its final voltage in response to a step change in voltage.
To find the maximum voltage in response to a current step input, we can set $\frac{dV}{dt} = 0$, giving us the equation:

$$0 = -\frac{v}{R \cdot C} + \frac{I_{\text{Stim}}}{C}$$

Solving this for V gives us V_{max} as:

$$V_{max} = I_{\text{Stim}} \cdot R$$

The correctness of this can be easily checked by looking at the plots from Figure 4 - 8 (where $I_{Stim} = 1$)

Next we calculate how long it takes until the capacitor is charged up to 99% using. The voltage across a charging capacitor in an RC circuit is given by:

$$v(t) = V_0 \left(1 - e^{-\frac{t}{R \cdot C}}\right)$$

we can set $v(t)$ equal to that percentage of the final value (V_0) and solve for t:

$$0.99 \cdot V_0 = V_0 \left(1 - e^{-\frac{t}{R \cdot C}}\right)$$

$$0.99 = 1 - e^{-\frac{t}{R \cdot C}}$$

$$-\frac{t}{R \cdot C} = \ln(0.01)$$

Solve for t:

$$t = -R \cdot C \cdot \ln(0.01) \approx 4.605 \cdot R \cdot C$$

Therefore, we can use factor 5 to approximate the time it takes for a capacitor to be fully charged up:

$$t \approx 5 \cdot R \cdot C$$

1.2.2

The following code shows the implementation of the RK45 solver with the provided code as a base. The result is plotted together with the results of a forward and backward Euler solver and can be seen in Figure 9. The time step for the forward and backward Euler solver was set to 5ms to illustrate the differences.

Additionally, Figure 10 shows the dynamic time steps used by the RK45 solver.

```
[...]
### ------------- PARAMETERS ---------
solvers = ['BE', 'FE', 'RK45']
showTimeSteps = False
[...]
### ------------- SOLVING FOR ALL SOLVERS ---------
for solver in solvers:
        if solver != 'FE' and solver != 'BE':
            # Use scipy's solve_ivp to solve the ODE system for the built-in solver
            sol = solve_ivp(
                lambda t, v: ode_system(t, v, I, R, C, tDel, tDur, tDt, solver),
                [0, tStop],
                [0],  # Initial condition
                method=solver
            )
            # Extract the solution
            vVec_solvers[solver] = sol.y[0]
            timeSteps_solvers[solver] = sol.t
        else:
            # Solve the ODE system for Forward Euler and Backward Euler
            vVec = np.zeros(timeSteps)
            for t in range(0, timeSteps - 1):
                IStim = 0
                if t >= int(tDel / tDt) and t < int((tDel + tDur) / tDt):
                    IStim = I  # in nA

                if solver == 'FE':
                    vVec[t + 1] = vVec[t] + ((-vVec[t] / R + IStim) / C) * tDt
                elif solver == 'BE':
                    vVec[t + 1] = (vVec[t] + IStim * (tDt / C)) / (1 + tDt / (R * C))

            vVec_solvers[solver] = vVec
            timeSteps_solvers[solver] = timeStep
[...]
### ------------- PLOTTING ---------
for solver in solvers:
        plot, = ax.plot(timeSteps_solvers[solver], vVec_solvers[solver
], label=solver+' Nr. Timesteps='+str(timeSteps_solvers[solver].size))
            if showTimeSteps:
                for t_step in timeSteps_solvers[solver]:
                    ax.axvline(x=t_step, color=plot.get_color(), linestyle=
'--', linewidth=0.8)
    [...]
    ### ------------- ODE ---------
    def ode_system(t, v, I, R, C, tDel, tDur, tDt, solver):
        # Stimulus current
        IStim = 0
        if t >= tDel and t < tDel + tDur:
            IStim = I  # in nA

        # Compute change of v
        if solver == 'BE':
            return (v + IStim * tDt / C) / (1 + tDt / (R * C))
        else:
            return (-v / R + IStim) / C
    [...]

```

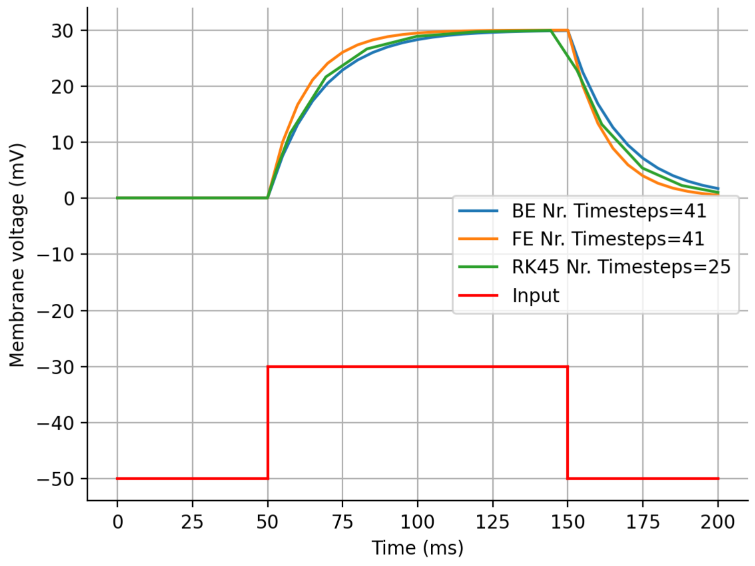

Figure 9: Plot of the response of an RC element calculated using a forward and backward Euler method as well as ‘RK45’

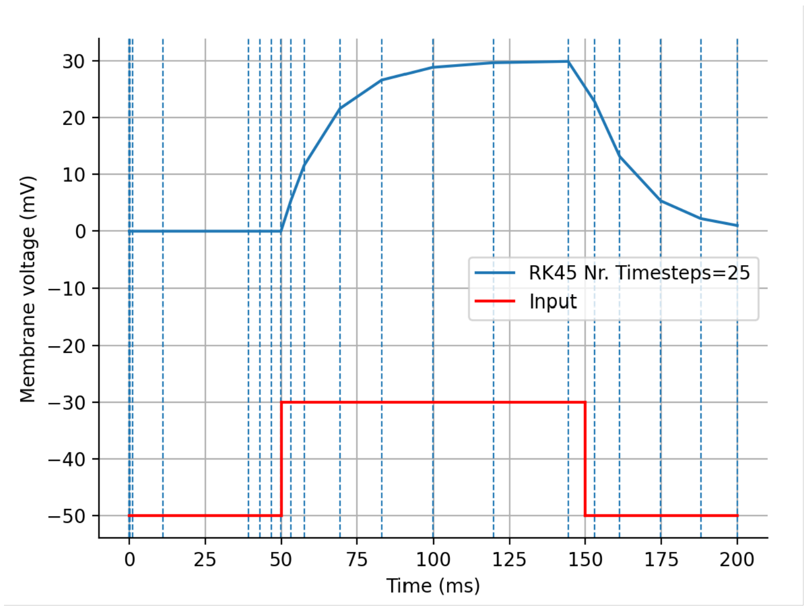

Figure 10: Plot of the response of an RC element calculated ‘RK45’ method. The colored vertical Lines indicate the time steps made for the calculation.

1.2.3

Using the following parameters generates the resulting plot shown in Figure 11. It gives a comparison between all the different available solvers.

```
    solvers = ['RK45', 'RK23', 'DOP853', 'Radau', 'BDF', 'LSODA']

```

Additionally, Figure 12 shows all the dynamic time steps of those solvers (color-coded). The number of steps are shown in the legend.

As we can see many solvers perform better in this case than the RK45 solver. The issue with the solution of RK45 is, that the dip happens too early (before the dip of the input signal).

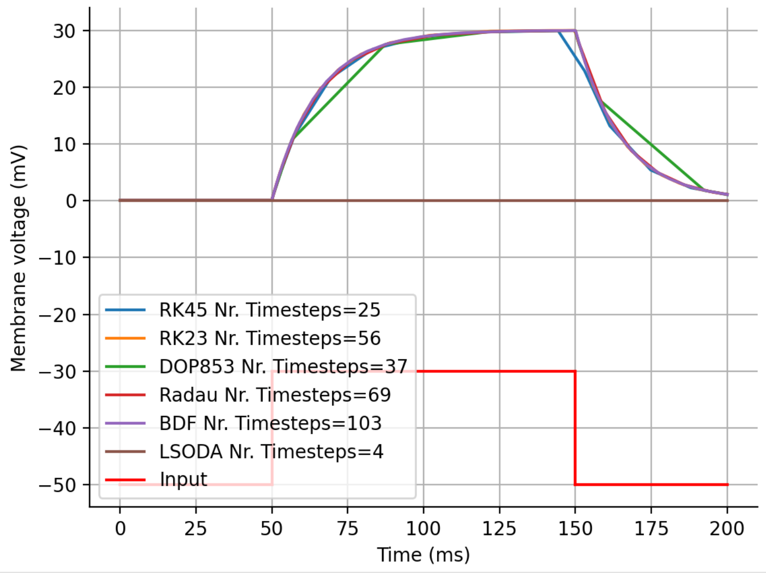

Figure 11: Plot of the response of an RC element calculated using a forward and backward Euler method as well as 'RK45'

Figure 13 shows a comparison between RK45 and RK23. In this specific case the RK23 solver actually performs better than the RK45 solver, since its closer to the real solution. It also uses more time steps though.

A second example of a better-performing solver is the Radau solver shown in Figure 14. It is intended to be used with stiff ODEs which our ODE can be classified as, at least to a degree. Radau however also uses almost three times the time steps compared to RK45.

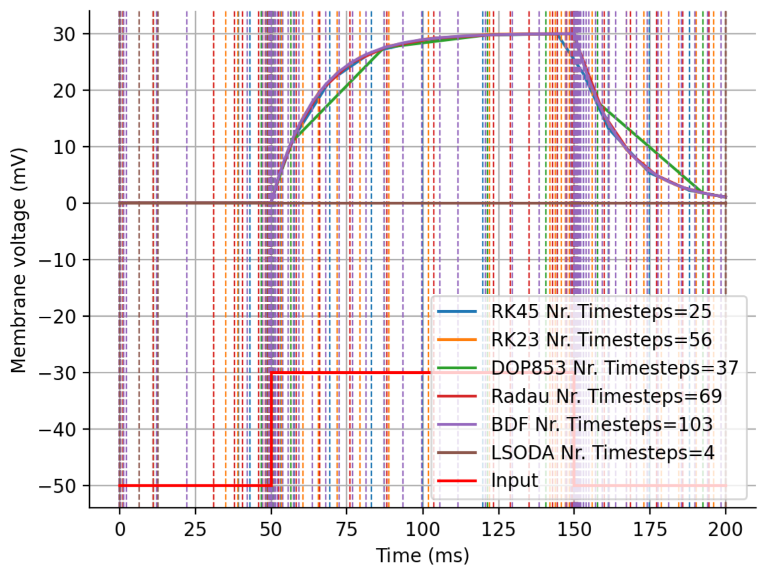

Figure 12: Plot of the response of an RC element calculated 'RK45' method. The colored vertical Lines indicate the time steps made for the calculation.

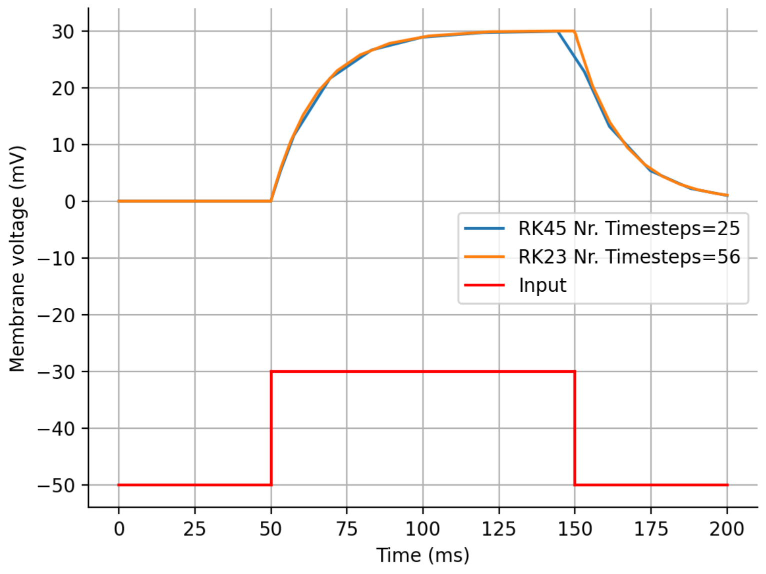

Figure 13: Plot of the response of an RC element calculated 'RK45' method. The colored vertical Lines indicate the time steps made for the calculation.

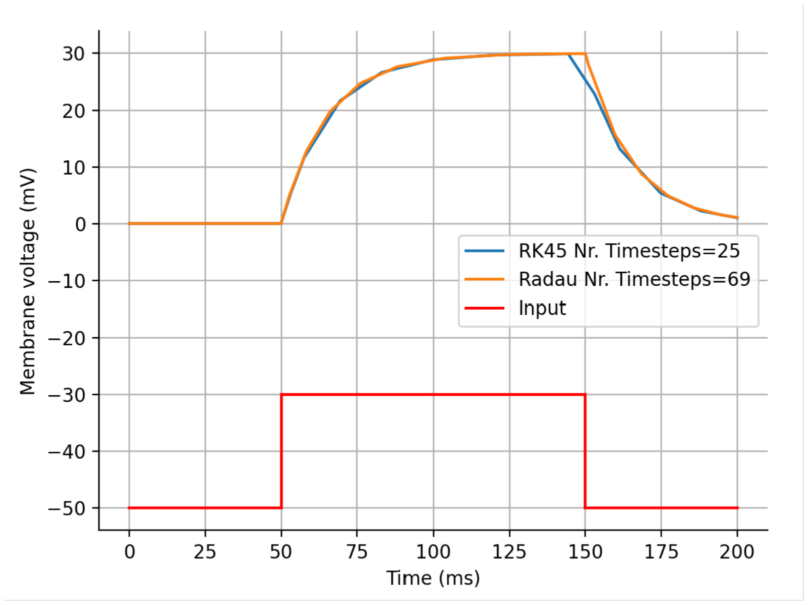

Figure 14: Plot of the response of an RC element calculated ‘RK45’ method. The colored vertical Lines indicate the time steps made for the calculation.

1.2.4

We can also try to 'tune' the RK45 solver in order to get better results. For this we can for example use the relative and absolute tolerances (r_{tol} and a_{tol}).
Figure 15 - 16 show a comparison between the results of the untuned and the tuned version of the RK45 solver with $r_{tol} = 5 \cdot 10^{-5}$ and $a_{tol} = 5 \cdot 10^{-7}$

As we can see, we can eliminate the early dip using those parameter values while still keeping a low amount of time steps. The lower we choose those parameter values the more time steps we will have and the more accurate are result will become.

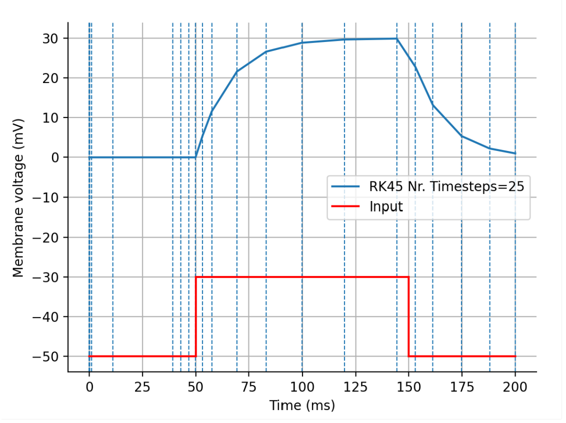

Figure 15: Response Plot of the RK45 solver with default parameters

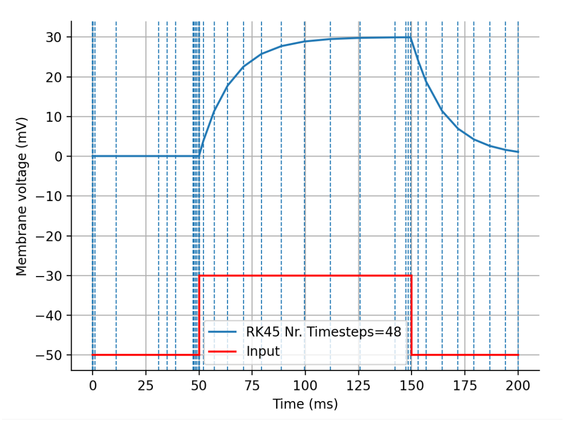

Figure 16: Response Plot of the RK45 solver with $r_{tol} = 5 \cdot 10^{-5}$ and $a_{tol} = 5 \cdot 10^{-7}$

2 Hodgkin-Huxley Model (single)

2.1

When varying the amplitude of the applied stimulus (I) for a 0.5 ms long pulse, we find, that the threshold current is at around $13.45uA/cm^2$ The resulting action potential can be seen in Figure 17.
When the stimulus amplitude is doubled ($I = 26.9\ \mu A/cm^2$), the action potential is initiated earlier. The resulting plot can be seen in Figure 18

2.2

As we can see in Figure 19 and 20, when increasing the time step to $tDt = 0.1ms$ and $I = 20\mu A/cm^2$ we still get a good solution with the Backward Euler Solver. The Forward Euler however produces immense oscillations, indicating that the time step is to big for this solver. The reason why this happens was discussed in a previous section.

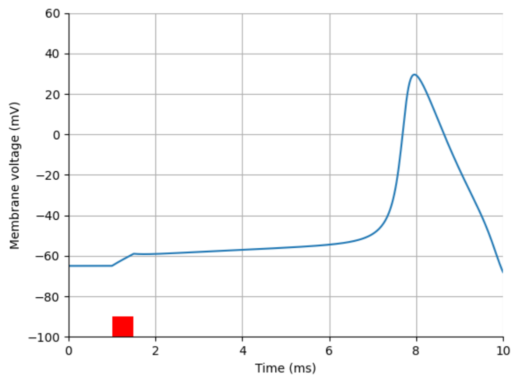

Figure 17: Membrane voltage over time for the spiking threshhold.

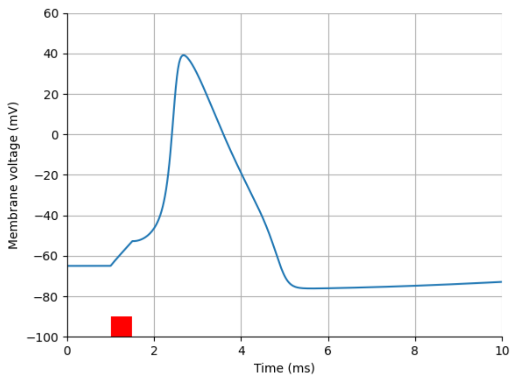

Figure 18: Membrane voltage over time for double the spiking threshhold.

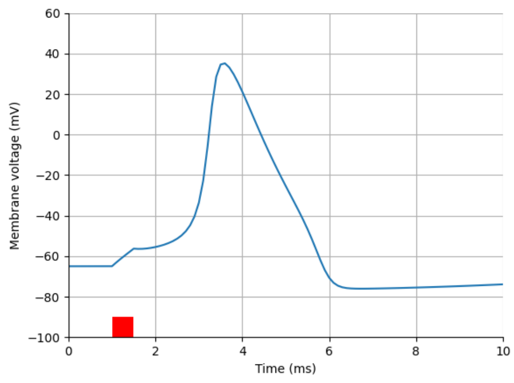

Figure 19: Action potential response solved with the Backward Euler Method

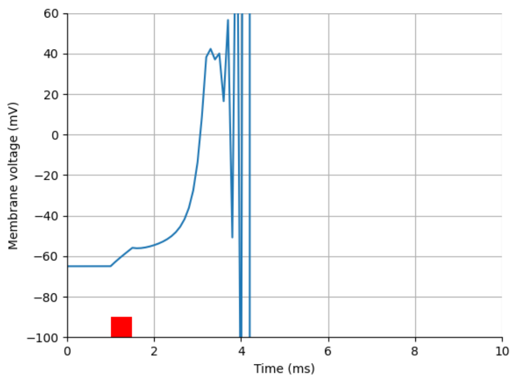

Figure 20: Action potential response solved with the Forward Euler Method

2.3

Figure 21 illustrates the behavior of the membrane voltage and the sodium and potassium current densities during the simulation. The code that produced this plot it given by:

```python
fig, ax1 = plt.subplots()
ax1.grid()
ax1.set_xlabel('Time (ms)')
ax1.set_ylabel('Membrane voltage (mV)', color='tab:blue')
ax1.plot(timeStep, vVec, label='Membrane Voltage', color='tab:blue')
ax1.tick_params(axis='y', labelcolor='tab:blue')

ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-
axis
ax2.set_ylabel('Current densities (uA/cm2)', color='tab:red')

# Plot sodium and potassium current densities with legends
ax2.plot(timeStep, gNa * mVec**3 * hVec * (vVec - eNa), '--', label='
Sodium Current (iNa)', color='tab:red')
ax2.plot(timeStep, gK * nVec**4 * (vVec - eK), '-.', label='Potassium
Current (iK)', color='tab:green')

ax2.tick_params(axis='y', labelcolor='tab:red')

# Add legend
lines, labels = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax2.legend(lines + lines2, labels + labels2, loc='upper right')

fig.tight_layout()  # ensure the shared x-axis labels are not slightly
cut off
plt.show()

```

The rise in the membrane voltage during the initial phase of the simulation is caused by the influx of sodium ions through voltage-gated sodium channels. This depolarization phase is a result of the activation of sodium channels (governed by variables m and h) and the subsequent increase in sodium current density. After reaching a peak, the membrane voltage starts to decline due to the inactivation of sodium channels and the activation of potassium channels. The potassium current density increases, leading to repolarization and the restoration of the resting membrane voltage.

2.4

As can be seen in Figure 22 and 23 higher temperatures generally lead to shorter action potential durations (widths) and decreased action potential amplitudes (heights). The peak tends to appear earlier for higher temperatures.
For temperatures above 15C we can no longer observe an action potential. This also means that with the previously chosen parameters we do not get am action potential at 37C. Still it has been shown many times be researcher, that the Hodgkin-Huxley model can be used for humans. For this we would have to adapt our parameters accordingly, though.

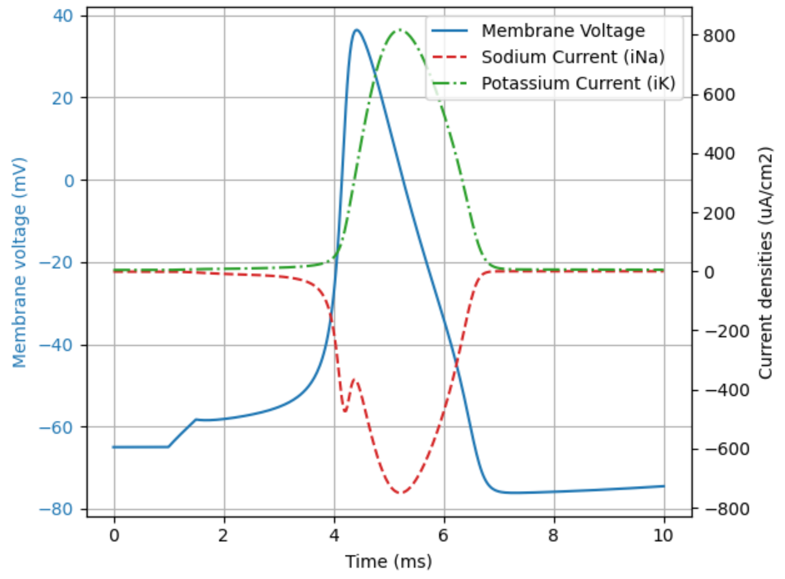

Figure 21: Membrane voltage and current densities over time. $(I = 15\mu A/cm^2,\ tDur = 0.5ms)$

2.5

The sodium current density (i_{Na}) in the Hodgkin & Huxley model is given by the equation:

$$i_{Na} = g_{Na} \cdot m^3 \cdot h \cdot (v - e_{Na})$$

where:

g_{Na} : Sodium channel maximum conductivity
m : Activation gating variable
h : Inactivation gating variable
v : Membrane voltage
e_{Na} : Sodium reversal/equilibrium potential

The conditions under which the sodium flux reverses its direction, leading to a repolarizing effect, occur when the membrane voltage exceeds the sodium reversal potential (e_{Na}).

The activation gating variable m responds to an increase in membrane voltage by increasing its value. The equation governing m in the model is:

$$\frac{dm}{dt} = \alpha_m \cdot (1 - m) - \beta_m \cdot m$$

where:

α_m : Rate of activation
β_m : Rate of deactivation

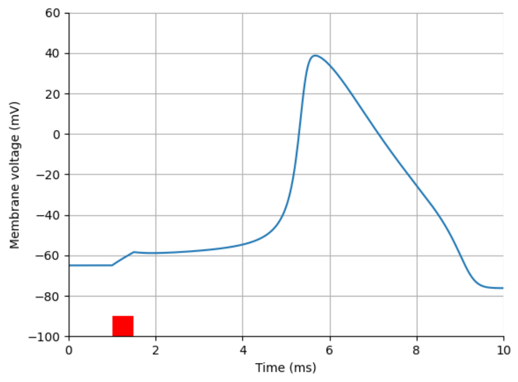

Figure 22: Action Potential at 2°C

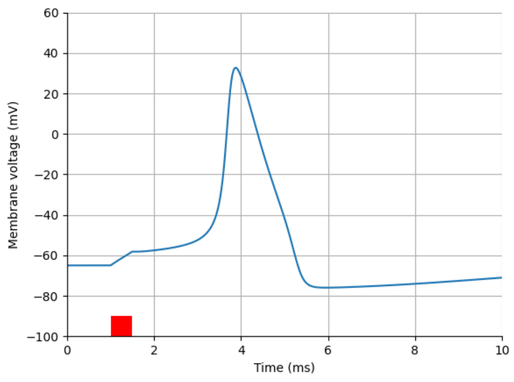

Figure 23: Action Potential at 10°C

An increase in membrane voltage generally leads to an increased activation of sodium channels, resulting in an increase in m and a higher probability of sodium channels being open.

The inactivation gating variable h responds to an increase in membrane voltage by decreasing its value. The equation governing h in the model is:

$$\frac{dh}{dt} = \alpha_h \cdot (1 - h) - \beta_h \cdot h$$

where:

$$\alpha_h : \text{Rate of inactivation}$$
$$\beta_h : \text{Rate of deinactivation}$$

An increase in membrane voltage typically leads to a decrease in the inactivation of sodium channels, resulting in a decrease in h and a lower probability of sodium channels being inactivated.

2.6

The following Python script computes the Strength-Duration (SD) curve for the single-compartment Hodgkin & Huxley model. The script uses a binary search algorithm to efficiently find the threshold for different pulse durations.

```python
    def binary_search_threshold(time_step, stim_duration, pulse_amplitude):
        low, high = 0, 1000  # Initial search range for threshold
        threshold = None

        while high - low > 1e-6:
            current_amplitude = (low + high) / 2
            _, v, _, _, _ = HH_single(I=current_amplitude, tDur=
    stim_duration)
            max_voltage = np.max(v)

            if max_voltage > 0:
                high = current_amplitude
                threshold = current_amplitude
            else:
                low = current_amplitude

        return threshold
    [...]

    # Parameters
    pulse_durations = np.logspace(-2, 2, 20)  # Pulse durations from 0.01 to 100 ms in 20 logarithmic steps
    stim_duration_long_pulse = 100  # Duration of the long pulse for rheobase calculation

    # Compute SD curve
    thresholds = []

    for duration in pulse_durations:
        threshold = binary_search_threshold(0.025, duration, 15)
        thresholds.append(threshold)
    [...]

    rheobase = binary_search_threshold(0.025, stim_duration_long_pulse, 15)
    double_rheobase = 2 * rheobase
    index_double_rheobase = np.argmin(np.abs(np.array(thresholds) - double_rheobase))
    chronaxie = pulse_durations[index_double_rheobase]

```

From the computed SD curve we get the following values for the Rheobase and Chronaxie:

- Rheobase: 2.22 uA/cm^2
- Chronaxie: 2.07 ms

2.7

The following Python script computes the spiking probability for amplitudes between 0 and 30 $\mu A/cm^2$. For each amplitude, the simulation is run 50 times, and a logistic curve is fitted to the simulated data.

```
def logistic_function(x, L, k, x0):
    return L / (1 + np.exp(-k * (x - x0)))

# Parameters
amplitudes = np.arange(0, 30, 1)  # Stimulus amplitudes from 0 to 30 A / cm  in 1  A / cm  steps
num_simulations = 50
tDur = 0.5

```

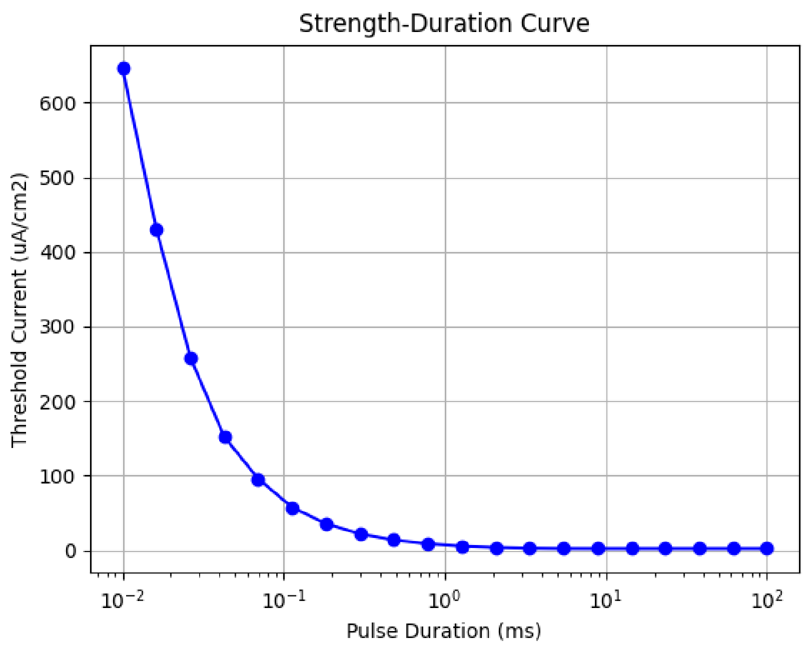

Figure 24: Strength-Duration Curve for the Hodgkin & Huxley Model.

```
# Function to run simulation and compute spiking probability
def run_simulation_and_compute_probability(amplitude):
    spiking_count = 0

    for _ in range(num_simulations):
        _, v, _, _, _ = HH_single(I=amplitude, tDur=tDur)
        if np.max(v) > 0:
            spiking_count += 1

    spiking_probability = spiking_count / num_simulations * 100
    return spiking_probability

# Compute spiking probabilities
spiking_probabilities = [run_simulation_and_compute_probability(
amplitude) for amplitude in amplitudes]

# Fit a logistic curve to the spiking probabilities
params, _ = curve_fit(logistic_function, amplitudes,
spiking_probabilities, p0=[100, 1, 15])

```

Additionally, a random factor is added to the total ionic current:

```
iIon = iNa+iK+iL + 20 * np.random.randn() # in uA/cm2
```

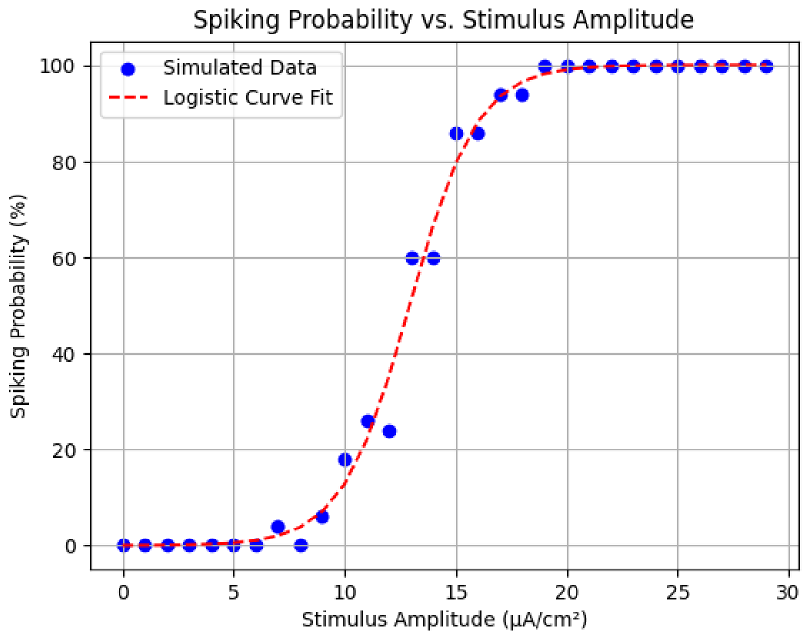

Figure 25: Spiking Probability vs. Stimulus Amplitude with Logistic Curve Fit.

The resulting plot can be seen in Figure 25 and the parameters of the fitted logistic curve are:

- Parameter L: 100.18
- Parameter k: 0.66
- Parameter x_0: 12.94

3 Multi-Compartment Hodgkin-Huxley Model

3.1

The membrane voltage was for a stimulation current of $I = -0.5$ plotted over the location of the fiber at a specific timestamp, which was chosen to be 0.1 ms after the stimulus onset. The resultant plot is shown in Figure 26.

The code that produces the plot is given by:

```
    specific_timestamp = int((tDel + 0.1) / tDt)
    fig, ax = plt.subplots()
    location = np.arange(-lComp * (nComp // 2), lComp * (nComp // 2) + 1,
    lComp)
    ax.plot(location, vMat[:, specific_timestamp], 'b')
    ax.set_xlabel('Location along fiber ( m )')
    ax.set_ylabel('Membrane Voltage (mV)')
    ax.set_title('Membrane Voltage Distribution at t = {:.2f} ms'.format(
    specific_timestamp * tDt))
    plt.show()

```

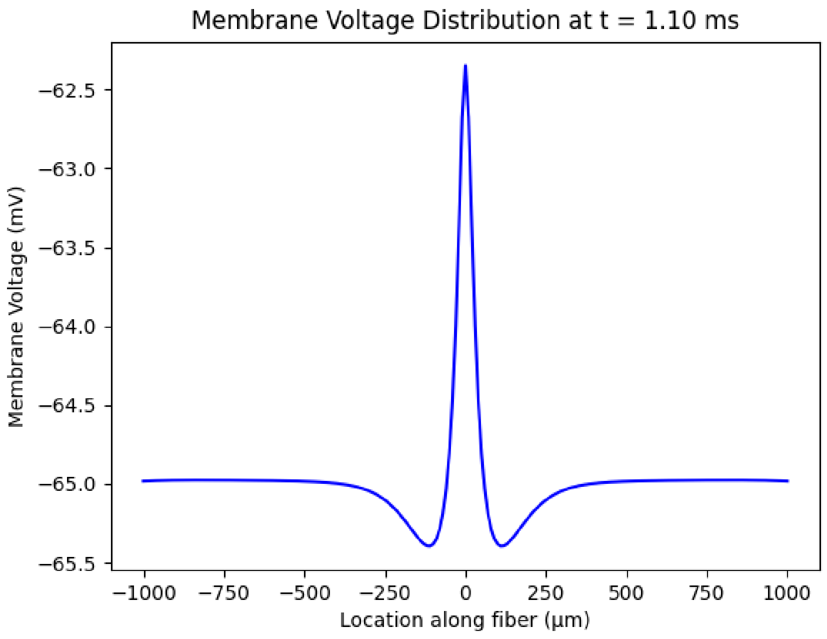

Figure 26: Membrane voltage distribution along the fiber at a specific time point after the onset of the cathodic stimulus.

When a cathodic stimulus is applied, the area immediately beneath the electrode becomes hyperpolarized due to the influx of negative charges. This creates a local potential difference between the stimulated area and adjacent regions. Adjacent regions experience a relative depolarization due to this, as positive charges flow towards the hyperpolarized area to balance the potential.

3.2

The activating function is calculated as the second spatial derivative of the extracellular potential and plotted using the following code:

```
    fig, ax = plt.subplots()
    location = np.arange(-lComp * (nComp // 2), lComp * (nComp // 2) + 1,
    lComp)
    ax.plot(location, np.gradient(np.gradient(potentials)), 'b')
    plt.title('Activating Function for Cathodic Pulse')
    plt.xlabel('Location along fiber ( m )')
    plt.ylabel('Activating Function (mV/ms)')
    plt.show()

```

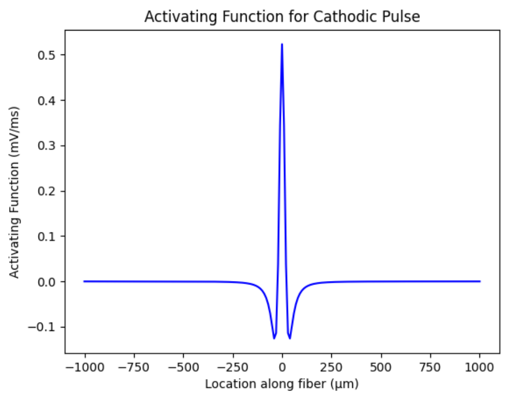

Figure 27: Activating functions for a cathodic pulse.

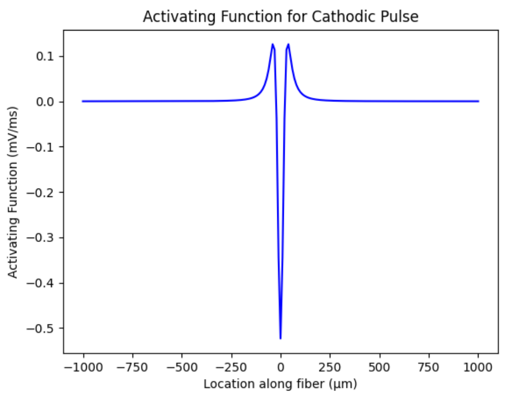

Figure 28: Activating functions for an anodic pulse

As can be seen in Figure 27 and 28, the activation function shows the regions that have high probability of hyperpolarization or depolarization in a given electric field. The cathodic and anodic activation functions are the inverse of each other, which is in alignment with the fact, that a positive stimulus can induce a hyperpolarization at the exact position where a negative stimulus can induce a depolarization.

3.3

Using the previous parameters of our model and altering the Stimulus amplitude, we find, that cathodic threshold to initiate an action potential from a point source is $I = -4\mu A$ and for a disk with $d = 50\mu m$ the threshold is $I = -2.5\mu A$. The action potential is initiated at the center, directly underneath the electrode and propagates outwards to the left and right from there. This can be seen in Figure 29 and 30.

3.4

When the intracellular resistivity increases, it becomes harder for the ionic currents to flow through the cytoplasm within the axon. This increased resistance slows down the rate at which the depolarization spreads down the axon. Decreasing the intracellular resistivity on the other hand increases the conduction velocity of an action potential.

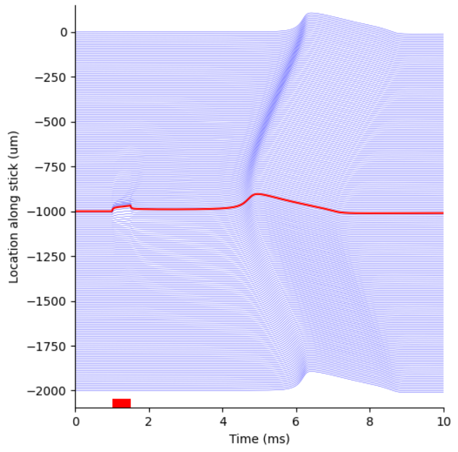

Figure 29: Propagating action potential initiated from a point electrode

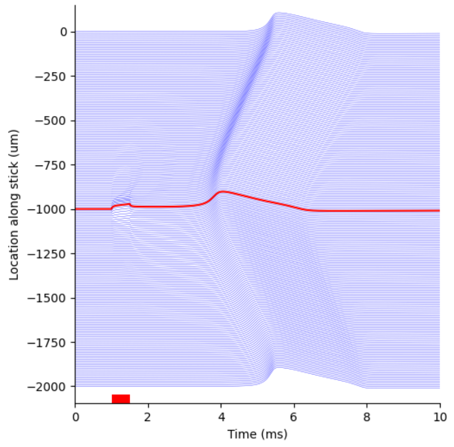

Figure 30: Propagating action potential initiated from a disk electrode

3.5

Despite initial hyperpolarization, it is still possible to trigger an action potential with an anodic pulse at higher amplitudes. This phenomenon is known as anodal break excitation. After the end of the anodic pulse, there can be a rebound effect. The membrane potential, having been hyperpolarized, returns to its resting state. This rebound can overshoot, leading to depolarization sufficient to reach the threshold for an action potential. For anodic stimulation, the action potential is typically not initiated directly under the electrode (where the hyperpolarization is greatest) but rather at a location adjacent to this area.

3.6

The following code was used to create the spiking activity plot that can be seen in Figure 31:

```python
def detect_spiking(vMat):
    threshold = -20  # Threshold voltage for spiking in mV
    return 1 if np.any(vMat > threshold) else 0

def HH_multi(cellX):

    [...]

    I = -5
    cellY = 25

    [...]

    return detect_spiking(vMat)
```

```python

if __name__ == '__main__':
    cellX_values = np.arange(-3000, 3001, 100)  # cellX from -3000 to 3000 in steps of 100  m
    spiking_activity = []

    for cellX in cellX_values:
        spiking = HH_multi(cellX)
        spiking_activity.append(spiking)

    # Plotting spiking activity vs. cellX coordinate
    plt.plot(cellX_values, spiking_activity, 'o-')
    plt.xlabel('Cell X Coordinate ( m )')
    plt.ylabel('Spiking Activity (0=No, 1=Yes)')
    plt.title('Spiking Activity vs. Cell X Coordinate')
    plt.grid(True)
    plt.show()

```

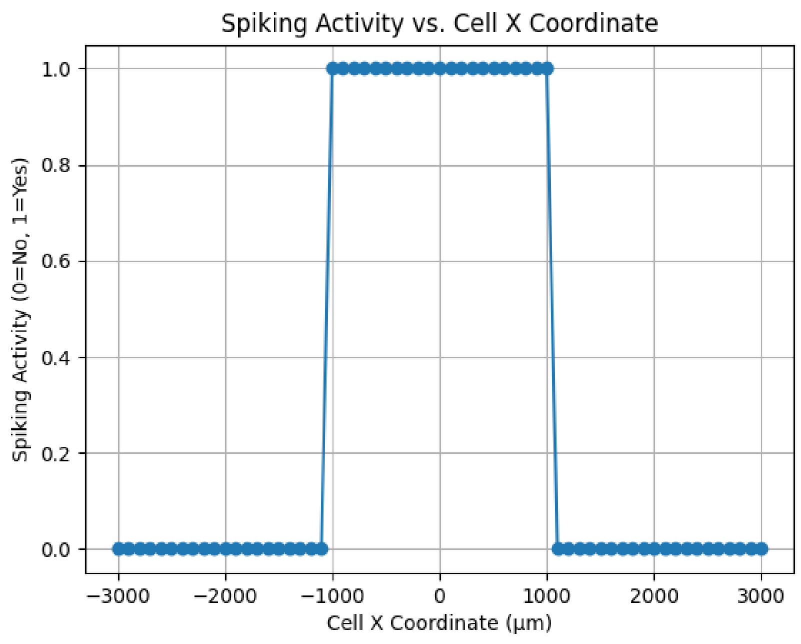

Figure 31: Spiking activity for a shifting electrode along the fiber

4 Finite Element Simulation

4.1

Figures 32-34 show the configuration built in `Agros2D` alongside the calculated mesh and e-Field. At first glance this e-Field seems correct.

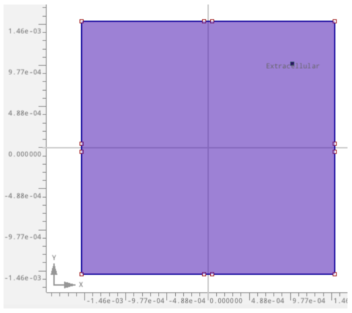

Figure 32: Electrode and Extracellular Configuration

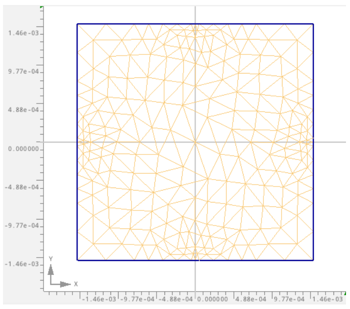

Figure 33: Generated Mesh for Finite Element Simulation

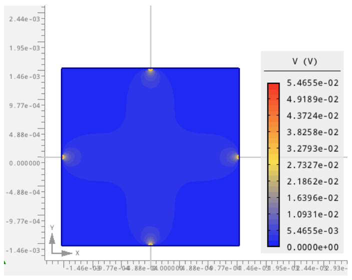

Figure 34: Calculated E-Field for the given Configuration

Looking at Figure 36 we see though, that especially for sharp changes in the electrical potential there are issues with the accuracy of this result.

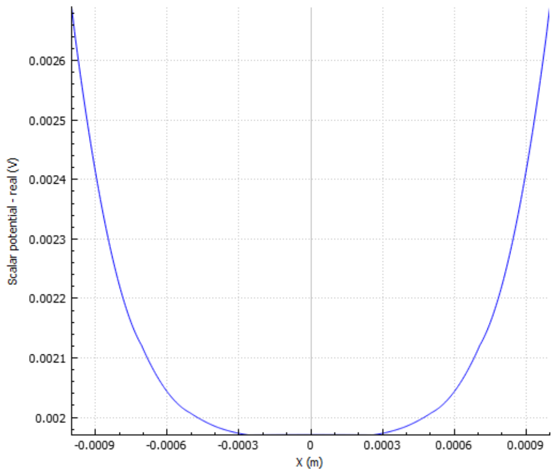

Figure 35: along the fiber in the center of the chamber

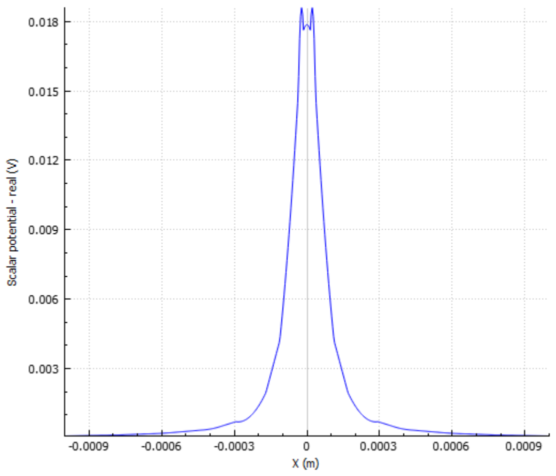

Figure 36: Electrical potential near the upper electrode

4.2

Next we increase the refinement iterations of the mesh to 2 and 3. The results can be seen in Figure 37-45. As we can see, the mesh gets more dense for each added iteration, also the results for the electrical potential are already a lot better for two iterations.

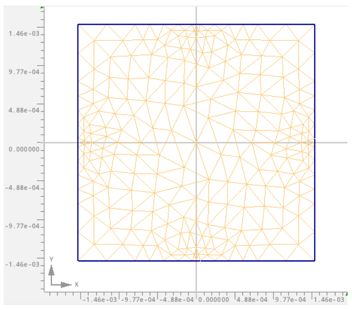
Figure 37: Generated Mesh for 1 refinement iteration

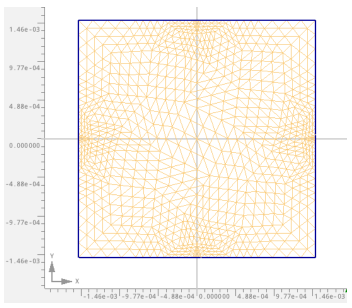
Figure 38: Generated Mesh for 2 refinement iterations

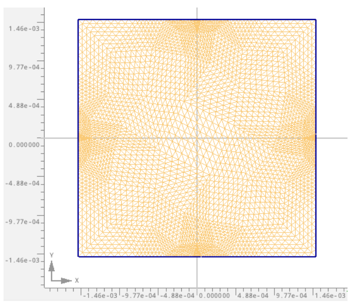
Figure 39: Generated Mesh for 2 refinement iterations

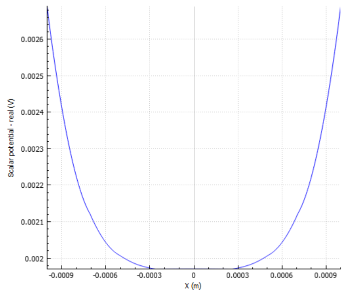
Figure 40: Electrical potential along the fiber for 1 refinement iteration

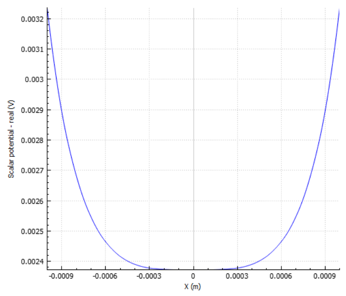
Figure 41: Electrical potential along the fiber for 2 refinement iterations

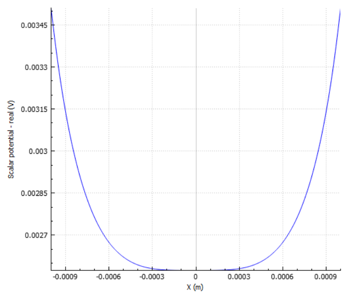
Figure 42: Electrical potential along the fiber for 3 refinement iterations

4.3

The following results show the anodic and cathodic spiking threshold for the electrode configuration above

1. **Direct Scaling**: Potentials imported in `HH_multi.py` were scaled by the stimulus amplitude I. The calculated thresholds for this method were as follows:

$$\text{Cathodic Threshold} : -50$$
$$\text{Anodic Threshold} : 180$$

2. **Electrode Boundary Condition Modification**: The stimulus current I was set to $1\mu A$, and the applied current density was directly modified in `Agros2D` by altering the electrode boundary condition. The calculated current densities and corresponding thresholds were as

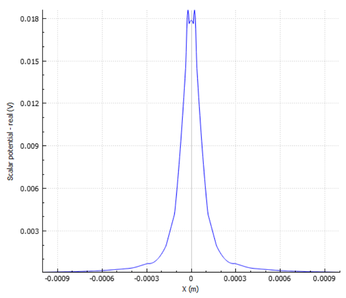

Figure 43: Electrical potential near the upper electrode for 1 refinement iteration

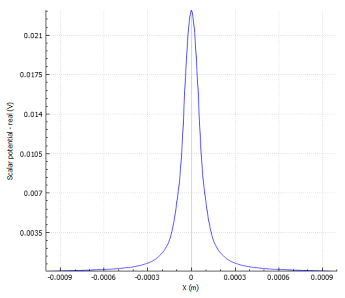

Figure 44: Electrical potential near the upper electrode for 2 refinement iterations

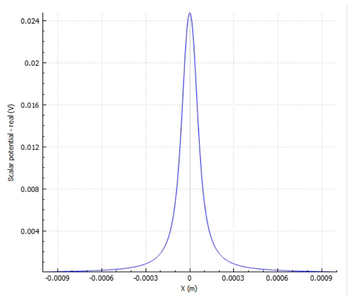

Figure 45: Electrical potential near the upper electrode for 3 refinement iterations

follows:

$$\text{Cathodic Threshold} : -50 \text{ (Current Density} = -6369.43\mu A/mm^2)$$
$$\text{Anodic Threshold} : 180 \text{ (Current Density} = 22929.94\mu A/mm^2)$$

As we can see both methods yield the exact same results, which was to be expected since the scaling of the current is of linear nature. This outcome is expected since the action potentials scale linearly with stimulus amplitude.

4.4

The following shows the spiking threshold for different electrode configurations depicted in Figure 46-48:

1. **Four electrodes**:

$$\text{Cathodic Threshold} : -50$$
$$\text{Anodic Threshold} : 180$$

2. **Horizontally aligned electrodes**:

$$\text{Cathodic Threshold} : -40$$
$$\text{Anodic Threshold} : 120$$

3. **Vertically aligned electrodes**:

$$\text{Cathodic Threshold} : -200$$
$$\text{Anodic Threshold} : 100$$

Figures 49-54 depict the location where the action potential initiates for each given configuration and for an anodic and a cathodic impulse. Those result seem correct, considering the proximity of the electrodes as well as, the fact, that anodic impulses tend to induce action potentials further away from the electrode, whereas cathodic impulses stimulate directly next to them.

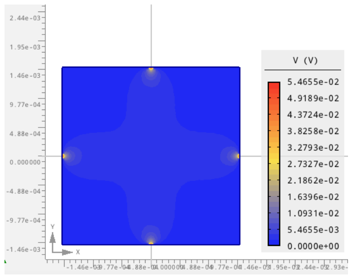

Figure 46: Calculated E-Field for four electrodes

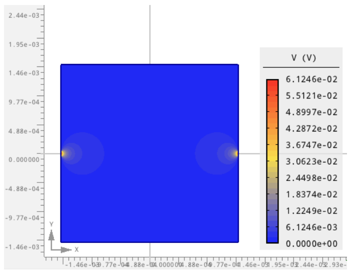

Figure 47: Calculated E-Field for two horizontally aligned electrodes

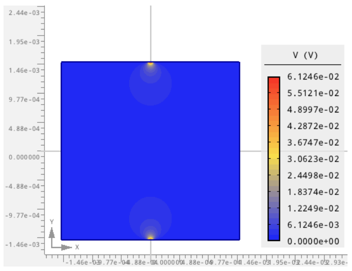

Figure 48: Calculated E-Field for two vertically aligned electrodes

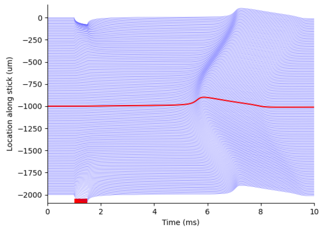

Figure 49: Anodic action potential initialization location for four electrodes

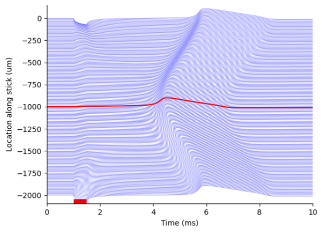

Figure 50: Anodic action potential initialization location for two horizontally aligned electrodes

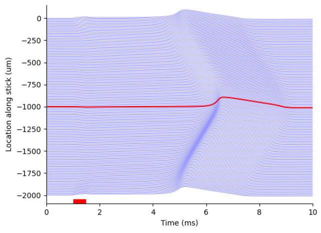

Figure 51: Anodic action potential initialization location for two vertically aligned electrodes

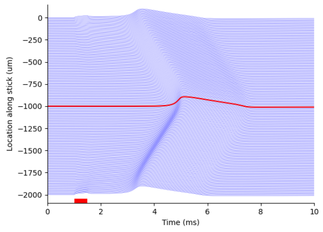

Figure 52: Cathodic action potential initialization location for four electrodes

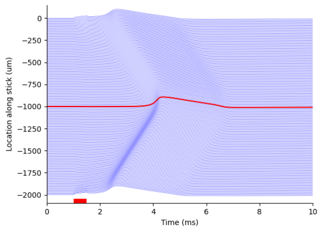

Figure 53: Cathodic action potential initialization location for two horizontally aligned electrodes

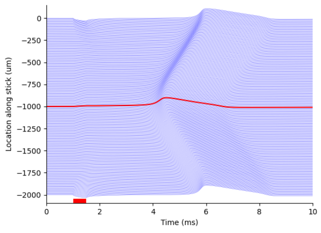

Figure 54: Cathodic action potential initialization location for two vertically aligned electrodes