Tutor/s

Dr. Miguel Gonzalez Perez
Departament de Química Física

Grau de Química

Mathematical techniques and Python programming for solving chemical problems.
Técnicas matemáticas y programación en Python para la resolución de problemas químicos.

Treball Final de Grau

Cristian Manica Georgiev
January 2026

Quiero agradecer primero a mi madre y a mi padre, por hacer posible mis estudios universitarios, respetando mi decisión de estudiar química.
Agradezco todo el apoyo de todas las personas que han hecho posible que llegue a ser la persona que soy hoy: mis amigos, ayudándome cuando tengo problemas, los profesores de la facultad por enseñarme la química y demostrarme que no se límita a la triste química que me enseñaron en bachillerato, y por último, a mi tutor por ayudarme con este TFG, muchas gracias por toda la ayuda otorgada a todos, no seria nadie sin vosotros.

REPORT

IDENTIFICATION AND REFLECTION ON THE SUSTAINABLE DEVELOPMENT GOALS (SDG)

The main objective of this TFG relates to the 5 Ps of the Sustainable Development Goals in two ways: Fist, it relates to people by providing computational tools, thus facilitating high-level scientific and technical training in electrochemistry, aiding students with the visualization of the theoretical equations, providing a visual representation that furthers their understainding on the math. And second, it focuses on the planet by providing alternatives to chemical experiments, making possible the prevention of generating chemical waste, which could be hazarduous.

This work aligns with three specific SDG goals and targets: SDG 4, SDG 9 and SDG 12.

Quality education (SDG 4) is reflected in the before mentioned P of the SDGs, People. The software supports target 4.4 by increasing the number of people who have vocation for chemistry, allowing them to further their skills for employment in the chemical and digital industries.

Industry, innovation and infrastructure (SDG 5) is shown in this work by providing a simple numerical solver, using RK4 and implicit methods, which as point 9.5 implies, it's a manner of enhancing industrical technological capabilities.

To finalize, respondible consumption and production (SDG 12) is apparent in this work by its adherence to the point 12.4, adhering to green chemistry principles which reduce the release of chemicals into the air, water and other mediums.

Should this project grow beyond its scope, the final projection would be the optimization and adoption into an open-access educational platform, which would benefit enormously students who have a hard time grasping the meaning behind the mathematical equations in chemistry.

Contents

1. Summary

The development of programs focused on chemistry has saved time, money, and, most importantly, reagents. Programs such as Gaussian, by solving Schrödinger's equation, have enabled a speed-up in chemical and technological development that was previously inconceivable. Other programs, such as LeaP, allow simulations of systems of hundreds of thousands of atoms, making it possible to simulate protein-drug interactions and obtain potential drugs for diseases that were previously incurable.

In order to solve the necessary equations, analytical solutions are not always possible, so the so-called numerical methods are used. Numerical methods are mathematical techniques that, instead of finding an exact solution as analytical methods do, find an approximate solution, which means that they involve an error (the difference between the solution obtained and the exact solution) that is not necessarily negligible in the answer. The magnitude of the error depends on the numerical method, where the most accurate ones are usually the most expensive in terms of time and computation, so in some cases a method with a reasonable error may be preferred.

All programs start from code in a file, and this TFG will explain the creation and development of a program in Python, where the program in question will attempt to solve equations posed in the subject "Physical Chemistry III" of the chemistry degree at the UB, focusing on the topic of electrochemistry. Relevant numerical methods for solving differential equations will also be mentioned, specifically the Crank-Nicolson method, an implicit method, and the Runge-Kutta methods of order 2 and 4, which are explicit methods.

Keywords: Mathematics, Chemical problems, Programming, Python.

2. Resumen

El desarrollo de programas enfocados para la química ha permitido el ahorro de tiempo, dinero y, más importante, reactivos. Programas como Gaussian, mediante la resolucion de la ecuación de Schrödinger, han permitido una agilicación en el desarrollo químico y tecnológico imposibles de concebir previamente. Otros programas como LeaP, permiten simulaciones de sistemas de átomos del orden de cientos de miles, pudiendo simular interacciones proteína-fármaco y obtener potenciales fármacos contra enfermedades antes sin cura.

Para poder resolver las ecuaciones necesarias, las soluciones analíticas no siempre son posibles, sino que se utilizan los llamados métodos numéricos. Los métodos númericos son técnicas matemáticas que en vez de encontrar una solución exacta, como lo hacen las técnicas analíticas, encuentran una solución aproximada, por lo que conllevan un error (la diferencia entre la solución obtenida y la exacta) que no necesáriamente es negligible en la respuesta. La magnitud del error depende del método numérico, donde los más exactos suelen ser los más caros en cuanto a tiempo y computacionalmente, por lo que en algunos casos el método con más error puede ser el preferido.

Todos los programas parten de código en un archivo, y en este TFG se explicará la creación y desarrollo de un programa en Python, donde el programa en cuestión tratará de resolver ecuaciones planteadas en la asignatura “Química Física III” del grado de química de la UB, centrado en el tema de electroquímica. También se mencionarán métodos numéricos relevantes en la resolución de ecuaciones diferenciales, específicamente el método de Crank-Nicolson, un método implícito, y los Runge-Kutta de orden 2 y 4, métodos explícitos.

Palabras clave: Matemáticas, Problemas Químicos, Programación, Python.

3. Introduction

3.1 Electrochemistry

Electrochemistry is the branch of chemistry that focuses on reactions where oxidation and reduction happen on the electrodes. These reactions are commonly called redox reactions, inheriting the “red” from reduction and the “ox” from oxidation. A general form for an oxidized species being reduced or vice-versa would be[1]:

$$O + e^- \rightleftharpoons R \tag{Eq.1}$$

The cathode is the electrode where the reduction process occurs, while the anode is the electrode where the oxidation process happens. Electrochemical reactions and processes might be referred as anodic or cathodic. To express the reaction rate of the cathodic and anodic processes of a electrochemical reaction, the following relation exists:

$$\begin{aligned} \nu_c &= k_c[O] \\ \nu_a &= k_a[R] \end{aligned} \tag{Eq.2}$$

The constants, k_a and k_c, can be represented through the transition state theory as[2]:

$$\begin{aligned} k_c &= \frac{k_b T}{h} \exp\left(\frac{-\Delta G_c^{\ddagger}}{RT}\right) \\ k_a &= \frac{k_b T}{h} \exp\left(\frac{-\Delta G_a^{\ddagger}}{RT}\right) \end{aligned} \tag{Eq.3}$$

When applying a potential to an electrode, the energy of the electrons in it canges, which shifts the activation energy barriers. The shift can be represented by the transfer coefficient (β), which represents the fraction of the electrical energy that affects the transition state. Since E^0 is the formal potential, the change in activation energy due to the overpotential relating to it ($\eta = E_{ap} - E^0$) is:

$$\begin{aligned} \Delta G_c^{\ddagger}(E) &= \Delta G_{0,c}^{\ddagger} + \beta F(E - E^0) \\ \Delta G_a^{\ddagger}(E) &= \Delta G_{0,a}^{\ddagger} + (1-\beta) F(E - E^0) \end{aligned} \tag{Eq.4}$$

At the formal potential ($E = E^0$), $\Delta G_{0,c}^{\ddagger} = \Delta G_{0,a}^{\ddagger}$, allowing the definition of the standard rate constant $k_0 = k_c = k_a$. However, when the overpotential is zero ($\eta = 0$), the condition $v_c = v_a$ must be met, which does not necessarily imply that the rate constants or the activation energies are identical if the concentrations differ. The cathodic barrier shifts into higher energies when the overpotential is positive by a factor of $\beta F\eta$, while the anodic shifts into lower energies by a factor of $(1-\beta)F\eta$. The opposite is true for negative overpotentials. Substituting brings this new form of Eq.3 alive:

$$\begin{aligned} k_c &= k_0 \exp\left(\frac{-\Delta G_c^{\ddagger}}{RT}\right) \\ k_a &= k_0 \exp\left(\frac{-\Delta G_a^{\ddagger}}{RT}\right) \end{aligned} \tag{Eq.5}$$

3.1.1 The Butler-Volmer’s equation

The Butler-Volmer equation connects the thermodynamics of the reactions, given by the overpotential (η), which acts as the driving force of the reaction, with kinetics given by the total current i, which is a sum of the partial anodic and cathodic currents i_a and i_c. The common writing of the Butler-Volmer uses current densities, which are defined as:

$$j = \frac{i}{A} \tag{Eq.6}$$

Since $j = j_a - j_c$ and $j = F\nu$, then follows $j = F(\nu_a - \nu_c)$ and using Eq.2, an expanded version can be written as:

[Eq.7] $$j = Fk_0([R]_s k_a - [O]_s k_c)$$

The resulting equation uses the concentrations on the surface of the electrode, as there's where the reaction happens, and it can be further expanded using Eq.3:

[Eq.8] $$j = Fk_0\left([R]_s \exp\left(\frac{(1-\beta)F\eta}{RT}\right) - [O]_s \exp\left(\frac{\beta F\eta}{RT}\right)\right)$$

The final expanded equation now relates the overpotential directly with the current density. This, however, is not the common writing of the Butler-Volmer equation either, as it can be compacted by defining the exchange current density j_0, using the concentrations at equilibrium as:

[Eq.9] $$j_0 = Fk_0[R]_{\text{eq}}^{\beta}[O]_{\text{eq}}^{(1-\beta)}$$

j_0 is defined as he dynamic equilibrium current that flows equally in both directions when the overpotential is zero, ie. $\eta = 0$, and reflects the intrinsic speed of the reaction. The exchange current density uses the bulk concentrations as no net reaction is happening. The Butler-Volmer equation is written commonly in a compact form as:

[Eq.10] $$j = j_0\left(\exp\left(\frac{(1-\beta)nF\eta}{RT}\right) - \exp\left(-\frac{\beta nF\eta}{RT}\right)\right)$$

where n is the number of electrons transfered. The previous considerations were made considering a single electron tranfer.

3.1.2 Levich and rotating disk electrode

A Rotating Disk Electrode (RDE) is a tool used to measure the hydrodynamics and study reaction kinetics and mass transport. Unlike a stationary electrode, diffusion does not depend on a stagnant layer that grows over time, since through rotation the RDE creates a constant, well-defined flow of electrolyte toward the surface of the electrode.
The Levich equation is able to link the rotation of the RDE with the resulting limit current i_L, which is defined as the maximum current achieved when the reaction is diffusion controlled, i.e., by how fast reactants can be diffused through the boundary layer, throughout this formula:

[Eq.11] $$i_L = 0.620nFAD^{\frac{2}{3}}\omega^{\frac{1}{2}}\nu^{-\frac{1}{6}}C$$

which can be written alternatively, integrating the diffusion layer as a parameter:

[Eq.12] $$i_L = nFAD\frac{C}{\delta}$$

with the diffusion layer being defined as:

[Eq.13] $$\delta = 1.61D^{\frac{1}{3}}\omega^{-\frac{1}{2}}\nu^{\frac{1}{6}}$$

3.1.3 Battery discharge equation

The voltage though an electrical circuit can also be calculated with Ohm's law:

[Eq.14] $$V = IR$$

A battery can be thought as a simple electrical circuit, with an internal resistance, so that:

[Eq.15] $$V = V - IR_{\text{int}}$$

The voltage when there's no current is known as the open circuit voltage, and is commonly denoted by the initials OCV. It depends on the state of charge of the battery (SOC), which is a percentage that represents the capacity in a rechargeable battery, where at 100% the battery is at full capacity and at 0% the battery is depleted. The discharge of a battery can be thought as the decreasing of the SOC. The SOC, then, can be thought of as a function depending on time, where the current $I(t)$ acts as a drain to the capacity (Q_{nom}) of the battery, expresed as[4]:

[Eq.16] $$\text{SOC}(t) = \text{SOC}_{t=0} - \frac{1}{Q_{\text{nom}}}\int_0^t I(t)dt$$

The voltage of the battery then can be expressed as the difference of the OCV and the sum of causes of energy loss:

[Eq.17] $$V = \text{OCV}(\text{SOC}) - IR_{\text{int}} - \eta_{\text{act}}$$

While IR_{int} accounts for the energy loss as a "physical" resistance, as electrons are moving through the material, η_{act} accounts for energy loss as a "chemical" resistance, as the energy required to make the redox reaction happen. Additionally, η_{act} obeys the following relation:

$$\eta_{\text{act}} = \frac{RT}{\beta F} \ln\left(\frac{I}{I_0}\right) \quad [\text{Eq.18}]$$

Thus, for low currents $IR_{\text{int}} \gg \eta_{\text{act}}$, and the voltage of a battery can be simplified as the following equation:

$$V = \text{OCV}(\text{SOC}) - IR_{\text{int}} \quad [\text{Eq.19}]$$

3.1.4 Cyclic Voltammetry

The Cyclic Voltammetry (CV) is a electrochemical technique where an electrical potential E is applied to the working electrode in sweeps and a current is obtained between the working and the counter electrodes[6]. The current i obtained is determined by the Butler-Volmer equation (Eq.10), however, another important factor to consider is the diffusion of the electroactive species, which is dictated by the Fick's laws:

$$J = -D\frac{\partial C}{\partial x} \quad [\text{Eq.20}]$$

The first Fick law describes how a diffusion flux is formed when a concentration gradient exists throughout a distance.The negative sign is crucial because it shows that the species move from high to low density regions to reach equilibrium.

$$\frac{\partial C}{\partial t} = D\frac{\partial^2 C}{\partial x^2} \quad [\text{Eq.21}]$$

The second fick law is obtained by differentiating the first law and describes how the rate of change of concentration respect to time relates to the curvature of the concentration gradient.
It is useful to think of the CV sweep as a transition between two regimes. First, the system is kinetically controlled, as the Butler-Volmer equation dictates the current based on the applied overpotential. However, when the overpotential exceeds a threshold, the concentration at the surface of the electrode drops to zero and the system becomes diffusion controlled. At that point, diffusion of the electrolyte to the surface of the electrode becomes the limiting factor and the current begins to decrease as the diffusion layer extends further into the bulk solution.

3.2 Numerical methods for differential equations

Numerical methods for differential equations are specific numerical methods for solving differential equations, which are equations that relate an unknown function to its derivatives. Some numerical methods approximate derivatives with finite differences. Finding a first order derivative is as simple as substituting the infinitesimally small h by a finitely small Δx in the definition of derivative:

$$f'(x) \approx \frac{f(x+\Delta x) - f(x)}{\Delta x} \quad [\text{Eq.22}]$$

This substitution cannot lead to the same value as the exact derivative, and brings a truncation error of the order of Δx, commonly denoted $O(\Delta x)$. For higher order derivatives, we can repeat the same process, however, the associated error can be minimized by using a clever trick with Taylor expansions:

$$f(x+\Delta x) = f(x) + f'(x)\Delta x + \frac{f''(x)}{2!}\Delta x^2 + \frac{f'''(x)}{3!}\Delta x^3 + O(\Delta x^4) \quad [\text{Eq.23}]$$

$$f(x-\Delta x) = f(x) - f'(x)\Delta x + \frac{f''(x)}{2!}\Delta x^2 - \frac{f'''(x)}{3!}\Delta x^3 + O(\Delta x^4) \quad [\text{Eq.24}]$$

Adding $f(x+\Delta x)$ to $f(x-\Delta x)$ we obtain the three point formula:

$$f''(x) \approx \frac{f(x+\Delta x) - 2f(x) + f(x-\Delta x)}{\Delta x^2} \quad [\text{Eq.25}]$$

This numerical second order derivative has a error of order $O(\Delta x^4)$, as both the first and third order terms vanish when adding, which makes it both efficient and precise for numerical computations.

3.2.1 Euler's method

The simplest numerical method to solve differential equations was published by Leonhard Euler in 1768, and it's known as forward Euler, as it's an explicit method, or just Euler's method. The method's core idea is to treat a differential equation as a formula for the slope of the tangent line. The curve $f(x)$ is initially unknown, but a defined starting point (x_0, y_0) is required, as computing the derivative $f'(x)$ at this point enables the recursive calculation of the next points. The tangent line obtained through the derivative acts as the direction towards the next step, with the distance moved depending on Δx:

$$f(x_0 + \Delta x) = f(x_0) + f'(x_0)\Delta x \tag{Eq.26}$$

or written alternatively, using subscripts instead:

$$\begin{aligned} y_1 &= y_0 + y'_0\Delta x \\ y_2 &= y_1 + y'_1\Delta x \\ &\dots \end{aligned} \tag{Eq.27}$$

Recursively, we arrive to a general formula:

$$y_{n+1} = y_n + y'_n\Delta x \tag{Eq.28}$$

3.2.2 Heun's method

The Heun's method, also named improved forward Euler's method fixes Euler's method by enforcing a slope correction, which is the average of the current slope and the Euler's predicteed next point. It adresses the primary weakness of Euler's approach, i.e., the assumption that the slope remains constant over the entire interval Δx. The process begins by calculating an initial estimate of the next point using a standard Euler step. Although more expensive to compute, averaging this new slope with the initial slope brings a much more accurate result, as it accounts for the curvature of the function $f(x)$. The resulting function is written as:

$$y_{n+1} = y_n + \frac{\Delta x}{2}(k_1 + k_2) \tag{Eq.29}$$

where the slopes k_1 and k_2 are:

$$\begin{cases} k_1 = f(x_n, y_n) \\ k_2 = f(x_n + \Delta x, y_n + \Delta x k_1) \end{cases} \tag{Eq.30}$$

where $f(x_n, y_n)$ corresponds to the derivative at point x_n given the value y_n.

3.2.3 Runge-Kutta 4 method

The Runge-Kutta methods are a family of explicit numerical methods, as it's a generalized form for explicit ODE methods. The Euler's method is taken as starting point, and slope corrections are introduced, with higher order Runge-Kutta methods being more precise, at the cost of being more computationally expensive. Euler's method by definition is RK1, then Heun's method is RK2, with the most famous method being the fourth order RK4, which expands on the Heun's slope correction:

$$y_{n+1} = y_n + \frac{\Delta x}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{Eq.31}$$

with slopes corresponding to:

$$\begin{cases} k_1 = f(x_n, y_n) \\ k_2 = f\left(x_n + \frac{\Delta x}{2}, y_n + \frac{\Delta x}{2}k_1\right) \\ k_3 = f\left(x_n + \frac{\Delta x}{2}, y_n + \frac{\Delta x}{2}k_2\right) \\ k_4 = f(x_n + \Delta x, y_n + \Delta x k_3) \end{cases} \tag{Eq.32}$$

3.2.4 Crank-Nicolson's method

The Runge-Kutta family of methods provide high accuracy for ordinary differential equations, however, for partial differential equations, which involve more than one variable, significant stability challenges are introduced. The Crank-Nicolson method is a second-order implicit method designed to overcome the numerical instability that arises with explicit methods when the time step Δt is large relative to the distance step Δx. It uses a finite difference method for the derivatives, using the three point formula, to solve differental equations similar to Fick's second law, where a first order derivative in the LHS with respect to one variable is related to a second order derivative in the RHS with respect to another variable. Then it averages the current time step with the next time step. Using the Crank-Nicolson for the diffusion equation (Eq.21) produces the following equation:

[Eq.33] $$\frac{u_i^{n+1}-u_i^n}{\Delta t}=\frac{D}{2}\left(\frac{u_{i+1}^n-2u_i^n+u_{i-1}^n}{\Delta x^2}+\frac{u_{i+1}^{n+1}-2u_i^{n+1}+u_{i-1}^{n+1}}{\Delta x^2}\right)$$

where u is the concentration and at time n and distance i. By defining $\lambda=\frac{D\Delta t}{2\Delta x^2}$, and separating by time step it can be rewritten as:

[Eq.34] $$-\lambda u_{n+1}^{n+1}+(1+2\lambda)u_i^{n+1}-\lambda u_i^{n+1}=\lambda u_{n+1}^n+(1-2\lambda)u_i^n+\lambda u_i^n$$

As the terms of the RHS are all known, the obtained row of the matrix has the following form:

[Eq.35] $$(-\lambda\ \ 1+2\lambda\ \ -\lambda)\begin{pmatrix}u_{i-1}^{n+1}\\u_i^{n+1}\\u_{i+1}^{n+1}\end{pmatrix}=\begin{pmatrix}\lambda u_{i-1}^n\\(1-2\lambda)u_i^n\\\lambda u_{i+1}^n\end{pmatrix}$$

This has the form $ax_{i-1}+bx_i+cx_{i+1}=d$. When extended to the whole matrix, the matrix obtained is a tridiagonal matrix, easily solved by the Thomas algorithm.

3.2.5 Thomas algorithm

The Thomas algorithm replaces standard gaussian elimination by providing a more efficient manner of solving tridiagonal matrices, consisting of a forward sweep and back substitution[5].

[Eq.36] $$\begin{pmatrix}b_1&c_1&&&0\\a_2&b_2&c_2&&\\&a_3&b_3&\ddots&\\&&\ddots&\ddots&c_{n-1}\\0&&&a_n&b_n\end{pmatrix}\begin{pmatrix}x_1\\x_2\\x_3\\\vdots\\x_n\end{pmatrix}=\begin{pmatrix}d_1\\d_2\\d_3\\\vdots\\d_n\end{pmatrix}$$

The forward sweep consists of eliminating every element of the a diagonal by performing row elimination, using the previous row's b as the pivot. By reducing each row into having one less unknown, the final row ends up with only one unknown, x_n , which can be easily solved algebraically. Now that x_n's value is known, the $n-1_{\text{th}}$ row can be solved.This holds recursively, substituting the newly found value into the previous row until the first row is reached, solving the full vector of unknowns.

4. Objectives

The primary objective of this TFG is both pedagogical and didactical: the creation, coding and implementation of the electrochemical systems studied in class and the knowledge acquired through them are the main goal of this work.

5. Experimental section

This TFG's program was made in python 3.13. The main structure of the program is shown by Figure 1:

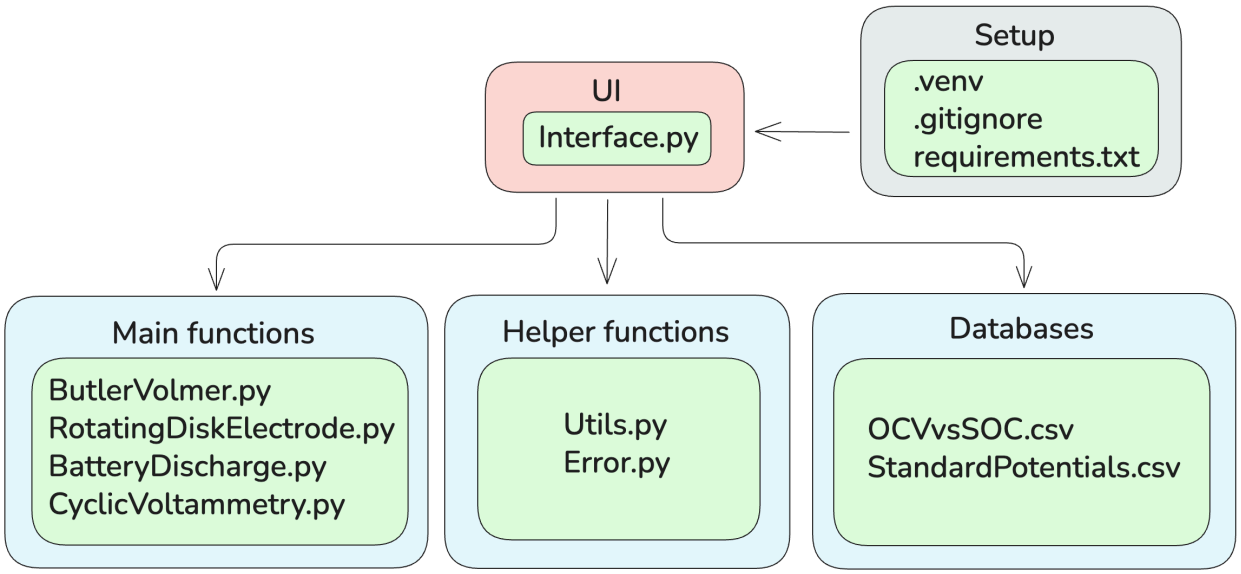


Figure 1: High-level structure overview

5.1 Setup

A virtual enviroment is used, so generation of a python3 venv is required. The generation of the virtual enviroment was done using the following shell command: `python3 -m venv .venv && source .venv/bin/activate && pip -r requirements.txt` This shell command is actually three commands merged into one by the "and" operator `&&`. First, an empty virtual enviroment is created, then the "source" command executes the .venv/bin/activate shell script and the context is shifted from the shell enviroment to the venv enviroment. Finally, pip, a python packet manager, installs the required packages listed in the requirements.txt file, which are: numpy for efficient arrays, scipy for constants and matplotlib for plotting graphs. Once the venv is created and has the required packages installed, the program can be executed by typing python3 interface.py or just ./interface.py.

5.2 Interface

The python file interface.py is the entrypoint of the program, which loads and parses the databases into memory and displays the UI by printing text in the terminal. When executed, the program first displays a selection of options ranging from one to five, as shown in Figure 2, where each mode (excluding 5) executes one of the four main functions.

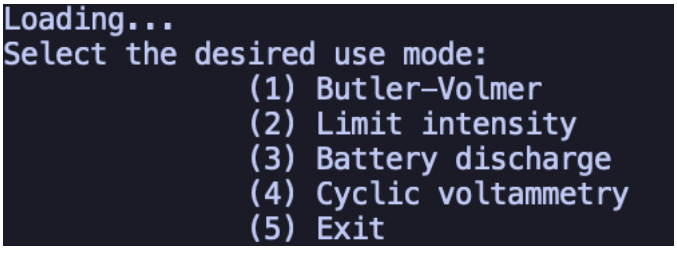


Figure 2: program modes

5.3 Helper functions

Helper functions can be thought of rogue functions, as they do not pertain to any specific section. Parsing functions, as "getElectrons" or "error", which are used to check if the input values are physically meaningful, are helper functions.

5.4 Databases

Databases store large amounts of useful data. Two databases are used for this project as .csv files (comma separated values). One is given in StandardPotentials.csv, a database that stores half reactions and their corresponding standard reduction potentials (in volts)[3], while the other, which is found in BatteryValues.csv relates the SOC of a Li-ion battery to its OCV[4]. These databases are parsed at runtime, when the program starts, and are stored as python dictionaries.

5.5 Main functions

When selecting a mode, without including exit, the program will shift the execution from interface.py to ButlerVolmer.py, RotatingDiskElectrode.py, BatteryDischarge.py or CyclicVoltammetry.py. All these files contain functions which ask for inputs and will plot a graph with the results obtained using the values given as inputs.

6. Implementation of the code

6.1 ButlerVolmer.py

In the Butler-Volmer equation, as described in Eq.10, some assumptions had to be made: First, the system is composed by a three electrodes, a working electrode, a reference electrode and a counter electrode. Second, only a redox system contributes to the output intensity and no other reactions occur. Third, to provide a meaningful graph, the function must be provided for a range of values, which will be given by the overpotential values explored.

The Butler-Volmer function has the following structure:

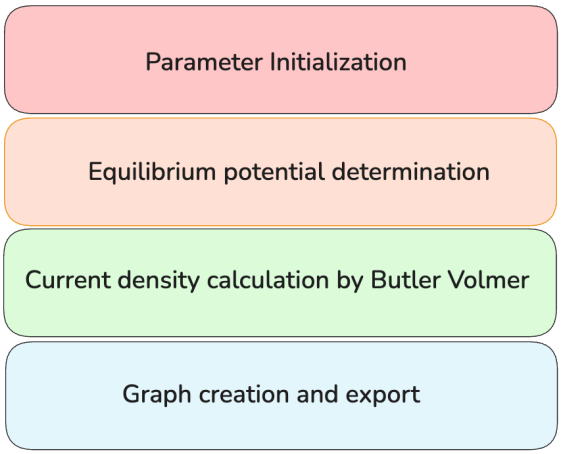


Figure 3: structure overview of ButlerVolmer.py

First, inputs are obtained through the error.py module, which consists of a huge helper function which takes as an input a string and returns either a float or a int. This function's structure consists of a match-case list (other languages might use switch-case instead), where if the input string finds a match, executes a try-catch which checks if the input is valid, and if it's not asks again until a valid input is provided. It should be mentioned that physical constants are not needed to be provided, as are assigned its proper value through the scipy.constants module. The final input, the half reaction, does undergo a special helper function, which checks if the provided half reaction exists in the StandardPotentials database and obtains the associated standard equilibrium potential. The following variables are obtained as mentioned: temperature, concentration of both the reduced and oxidized species, j_0, the applied potential and the reference electrode potential (using the Standard Hydrogen Electrode as 0 V). A function named "getEeq" is called next, using two helper functions, "getElectrons" and "getStoichCoeffs", to calculate the equilibrium potential through the Nernst equation. The overpotential η is calculated and a numpy ndarray is created through numpy's linspace function, resulting in a ndarray from $-|\eta|$ to $|\eta|$[7]. Finally, using numpy's exp function, which applies a exponential to each element of an array without any need to loop, j_a and j_c are assigned, and the currents array obtained from $j_0 * (j_a - j_c)$ is used to plot a E vs i graph with the "plotGraph" helper function and saved with whichever name the user wants.

6.2 RotatingDiskElectrode.py

In the same spirit as before, first some assumptions have to be explicited to determine the scope of the function. The function calculates the limiting current through the Levich equation, assuming the system uses a Rotating Disk Electrode or RDE to allow a definite diffusion layer δ to exist by providing laminar flow toward the electrode surface. The reaction must be always diffusion controlled, as the electron transfer should not be the limiting factor for the creation of the boundary layer. Finally, the solution uses a Newtonian fluid with constant viscosity. The structure of the function is the following:

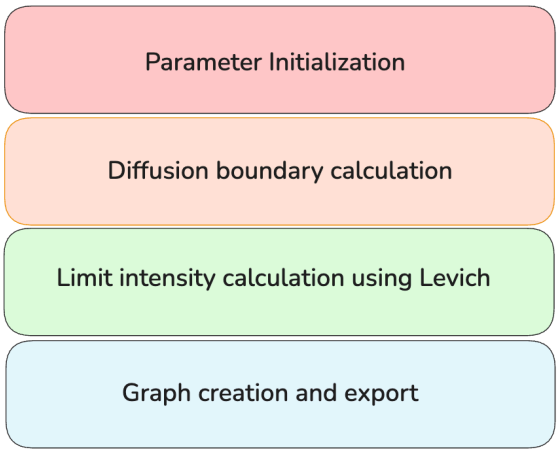


Figure 4: structure overview of RotatingDiskElectrode.py

Inputs are once again handled by the error helper function, ensuring that each physical parameter is valid before proceeding with the calculation, and the following variables are obtained from the user: the number of electrons involved in the redox process, the diffusion coefficient, the bulk concentration, the viscosity of the electrolyte solution, the electrode area and the maximum rotation speed in RPM. A numpy ndarray for the rotation speed is created using numpy's linspace function, ranging from 100RPM to the input provided. The RPM array is converted into angular velocity ω by the equivalence relation $\omega = \frac{2\pi}{60}\ \mathrm{RPM}$. Then the diffusion layer thickness δ is calculated using Eq.13 and the limiting current i_{lim} is computed for the entire range. Finally, the results are plotted as a graph of i_{lim} vs. RPM using the "plotGraph" helper function.

6.3 BatteryDischarge.py

The simulation of a battery discharge process, as formulated in the previous introduction section, relies on continuous tracking of the State of Charge (SOC) and its relation with the Open Circuit Voltage (OCV). The assumptions integrated into this function are the following: First, the discharge occurs at a constant current I, with a low enough value so that $R_{\mathrm{int}} \gg \eta$ and the simplyfing Eq.16 to the following equation is possible:

$$\mathrm{SOC(t)} = \mathrm{SOC}_{t=0} - \frac{I(t)}{Q_{\mathrm{nom}}} \qquad [\mathrm{Eq.37}]$$

Second, the internal resistance R_{int} is treated as a constant parameter throughout the discharge process, as only one type of battery is considered (a Li-ion battery). Finally, the relation between the SOC and the OCV is derived from empirical data stored in the database BatteryValues.csv, interpolating linearly between known data points. The structure of the battery discharge simulation is the following:

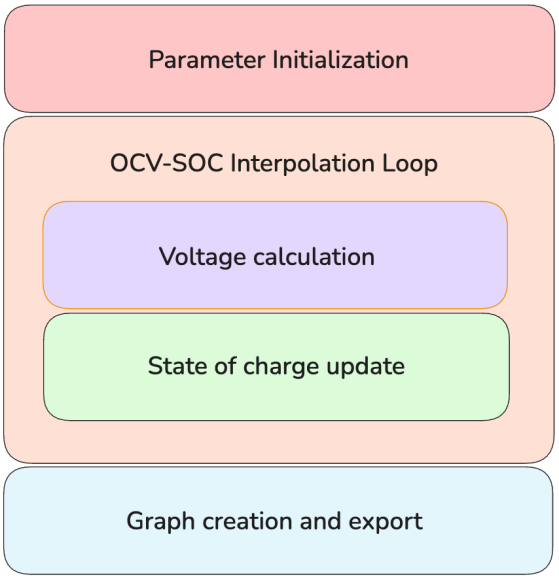


Figure 5: structure overview of BatteryDischarge.py

Inputs are once again handled by the error helper function. Variables as the current I, the internal resistance R_{int}, the nominal capacity Q_{nom}, initial SOC, $\text{SOC}_{t=0}$, and a time delimiter are obtained through the user's input. An empty python list is created for the output voltages, as unfortunately is impossible to take advantage of Numpy's ndarrays, as the end time is only known when the limiter is set: the dynamic duration of the battery's SOC is the source of the problem. Then, the SOC differential is calculated through the simplified Eq.37 ($d\text{SOC} = -\frac{I(t)}{Q_{\text{nom}}}$) and the if the time limiter is unset (input is 0) then it's set as 9e34 to ensure the battery discharges fully. The main loop of the function starts: for each step, the voltage through Eq.19 is calculated and appended into its corresponding position in the voltages list. The new SOC is set and the time differential is added to the current time, the loop is ready to perform a new iteration. To obtain the OCV atrributed to the current times' SOC, the "interpOCV" function is used. It's a function that takes the current SOC as an input and checks if the exact value exists in the database. If the value does not exist, it returns a value calculated through linear interpolation of the two closest values found in the database. As the condition of the while loop relies only on whether the time has reached or exceeded the delimited time or the SOC is a positive non-zero value, if the next time's SOC is negative no problem arises. Once the while loop loops through its last iteration, a graph is plotted through the "plotGraph" function using the voltages array turned into a Numpy ndarray with the array function and the time array, which is created using Numpy's "arange" with the length of the voltages multiplied by dt as an input. After generating a voltage vs. time plot, the user is prompted for a filename, and the resulting graph is saved, providing a visual representation of the battery's discharge curve.

6.4 CyclicVoltammetry.py

The simulation of a Cyclic Voltammetry (CV) is the most complex main funcion file in this TFG, as it uses three different numerical solvers for differential equations: Heun, RK4, and Crank-Nicolson. The "CyclicVoltammetry" is the first one that takes an input, the order of the RK method, if 0, Crank-Nicolson is used, if 2 or 4, RK2 (Heun) or RK4 is selected instead. It simulates the concentration profiles of the oxidized species (C_O) and the reduced species (C_R) as the electrode potential sweeps linearly over time. To ensure physical accuracy, the following assumptions are made: First, the solution is semi-infinite, meaning that the furthest concentrations do not sense the electrode. Second, the electron transfer follows Butler-Volmer kinetics at the electrode surface, and the spatial grid x is large enough to prevent boundary effects. Third, the solution is homogenous at $t = 0$, having the same concentration at every point of the solution. Fourth and last, the support electrolyte does not interact, and all migration effects are negligible. The CyclicVoltammetry function is structured into several components in this fashion:

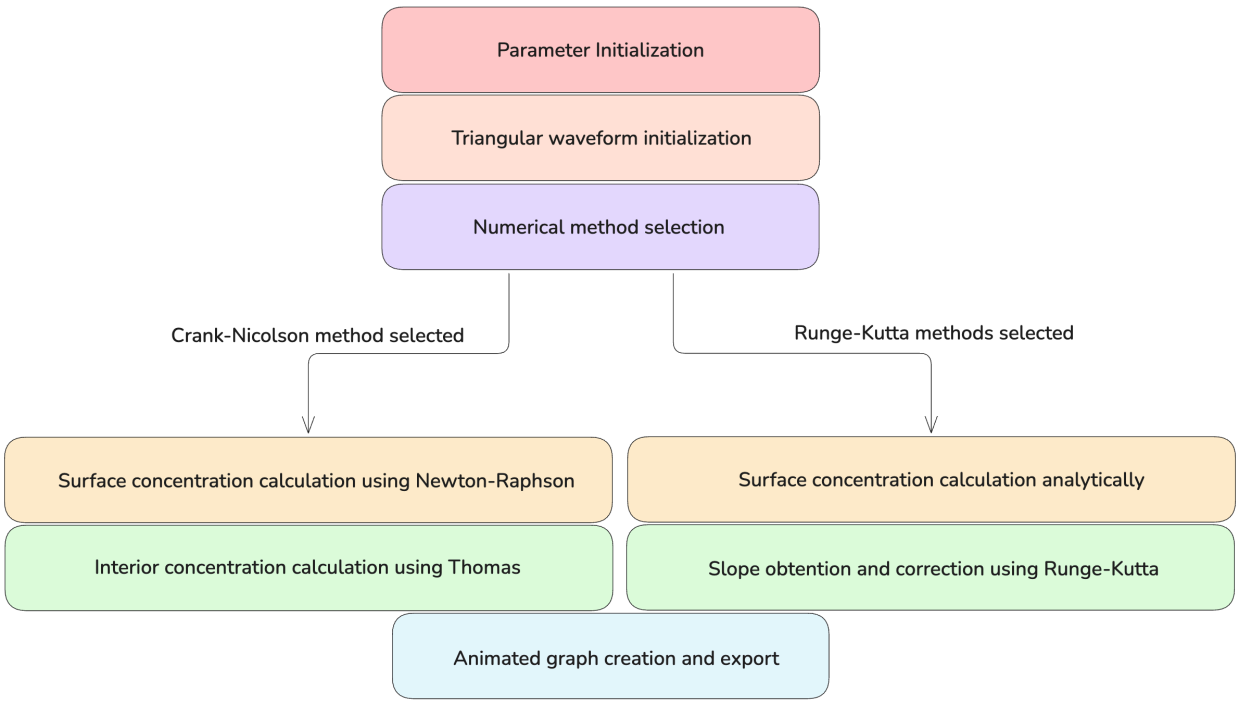


Figure 6: structure overview of CyclicVoltammetry.py

As before, the "error" function is used to check for valid/sensible physical inputs. The following variables are obtained from user input: starting potential, vertex potential, scan rate v, number of cycles, the starting bulk concentration for both the oxidized $C_O(\text{bulk})$and reduced species, the diffusion coefficient, the reaction's rate constant k_0, and the cathodic symmetry factor. Through the "getEo" helper function, the number of electrons are obtained, along with the standard reduction potential of the input half reaction. The following variables are calculated based on the input obtained values: t_{max}, x_{max}, dx and dt. The times

array and potential signals array is generated using the “init_time_potential” function. This function creates a triangular potential waveform based on the input starting potential, vertex potential and scan rate (v). The function uses a phase shift and the modulo operator to ensure the potential array correctly ranges between the minimum potential $E_{\text{start}} - E_{\text{vertex}}$ and maximum potential $E_{\text{start}} + E_{\text{vertex}}$ for the input number of cycles by the user. Then the space grid array x is generated through Numpy’s “arange” function, always of size 301 values, where x_0 is the surface of the electrode and $x_{301} = x_{\max}$. Then the concentration profiles are built through Numpy’s “full” function, filling two arrays of the same size than the times array with the input bulk concentrations for each species. Next, a python dictionary is created to provide all the needed variables to the functions in a less clutered manner, and finally, through if statements a numerical method is selected based on the order.

When order is 0, the Crank-Nicolson method is selected. The implementation of the the method employs a function named “newton_thomas” as a solver, which applies the implicit Crank-Nicolson method by averaging the current (u_t) and future (u_{t+1}) three-point formulas, as done in Eq.33. Then solves the resulting tridiagonal system with the implemented Thomas algorithm. Because Crank-Nicolson cannot solve non-linear equations and the surface concentrations are non-linear, a Newton-Raphson iteration to find a self-consistent solution (a solution with low enough error) for $C_{O,0}$ and $C_{R,0}$ at each time step is employed.

When order is 2 is Heun’s method (RK2) is selected. The implementation employs the a technique called Method of Lines, which discretizes the spacial second derivative of Eq.21, effectively transforming the diffusion PDE into a system of coupled (ODEs). To solve the system obtained first, the algorithm calculates an initial derivative using the “get_derivatives” function and then computes an average slope over the interval Δt as a slope corrector. This approach achieves a time precision of $O(\Delta t^2)$.

Finally , when order is 4,the Runge-Kutta 4 (RK4) algorithm is employed. This method has the highest time precision of all the available methods in the file, obtaining derivatives at four distinct points within each time step: at the beginning (k_1), two estimates at the midpoint (k_2, k_3) and the end (k_4). This achieves a temporal truncation error of $O(\Delta t^4)$. It is important to note that for all Runge-Kutta methods, the spatial precision remains $O(\Delta x^2)$ due to the second-order central difference approximation being used for the diffusion term, which is the same for all methods of the Runge-Kutta family. Throughout the Heun and RK4 simulations, surface kinetics are handled by the “solve_surface_analytic” function. This function treats the electrode surface as a flux balance between the rate of diffusion and the Butler-Volmer electron transfer kinetics.

Finally, for all the methods, the current density j is calculated as the product of the reaction rate and the Faraday constant. The concentration profiles for each step are saved for animation purposes and the voltammogram is plotted as a standard E vs. j graph using the “plotGraph” helper function. The concentration profiles are also plotted as C_O vs x and C_R vs x, using the profiles at time steps as frames for the animation.

7. Results and discussion

To prove the program works and the simulations are legitimate, the graphical output from the simulations must be compared with experimental sources. The validation process focuses on three pillars: thermodynamic consistency (Nernstian behavior), kinetic accuracy (Butler-Volmer response), and numerical stability (convergence of the Crank-Nicolson vs. RK4 solvers).

By comparing the simulated data against established literature values, the quantification of the error margins of our discrete model. This section discusses the transition from the continuous physical laws to the numerical approximations required for computational execution.

7.1 Butler Volmer

The Butler-Volmer output graph must imitate closely the graph studied in "Química Física 3", that means: at an overpotential of 0 V, the current density must be exactly 0 A/m^2. This represents the system at equilibrium where the anodic and cathodic currents cancel each other out. There also must be a linear region at small overpotentials ($\eta < 10^{-4}V$), where the curve must look like a straight line where current is directly proportional to overpotential. At larger overpotentials ($|\eta| > 0.1V$) a Tafel region where current should increase exponentially as moving further away from equilibrium must be noticeable. Finally, for a $\beta_c = 0.5$, the positive y-axis curve for oxidation ($\eta > 0$) and the negative y-axis curve for reduction ($\eta < 0$) should be mirror images of each other.
First, the function with no reactives was tried:

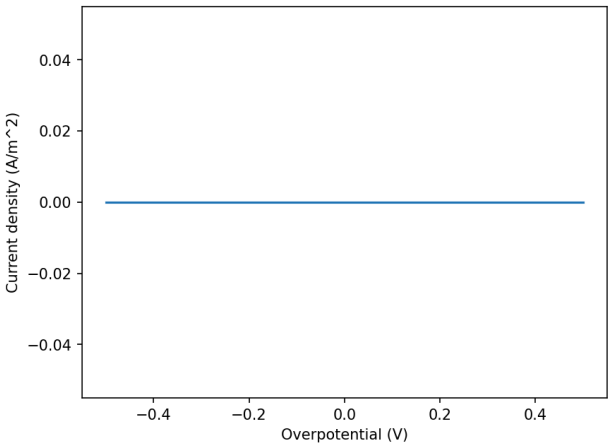


Figure 7: Butler Volmer with no reactives

Then, ferrocinium reduction to ferrocene with the same concentration ($E_{\text{eq}} = 0.4$) and with 10 times the concentration of the oxidized species ($E_{\text{eq}} = 0.46$) was tried:

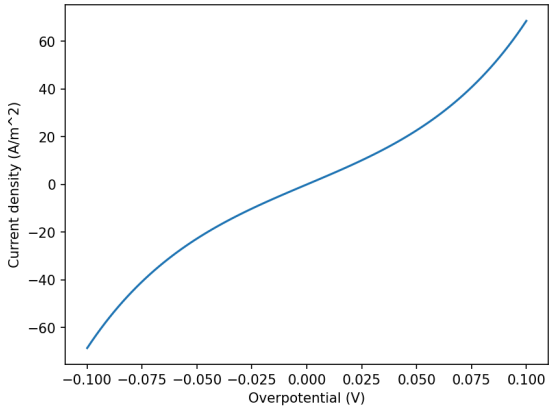


Figure 8: $\mathrm{Fc}^+ + e^- \rightarrow \mathrm{Fc}$ with $[\mathrm{Fc}^+] = 10^{-4}\ \mathrm{mol}/m^3, [\mathrm{Fc}] = 10^{-4}\ \mathrm{mol}/m^3$

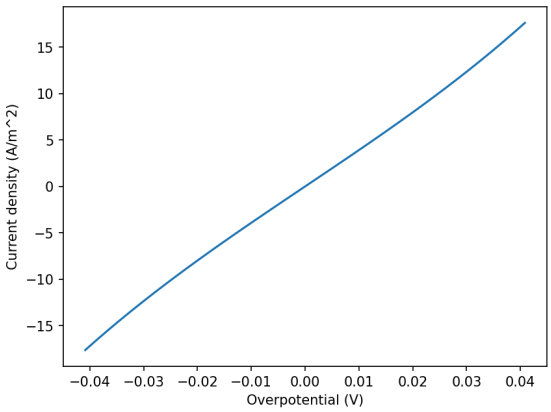


Figure 9: $\mathrm{Fc}^{+} + e^{-} \rightarrow \mathrm{Fc}$ with $[\mathrm{Fc}^{+}] = 10^{-3}\ \mathrm{mol}/m^3, [\mathrm{Fc}] = 10^{-4}\ \mathrm{mol}/m^3$

7.2 Rotating Disk Electrode

The Rotating Disk Electrode output graph must follow properly a graph resembling the function $\sqrt{x}$, as the graph measures current vs RPM and the Levich equation (Eq.11) states $i_L \propto \omega^{\frac{1}{2}}$. The Levich equation states the limit intensity must be linearly proportional to both the bulk concentration and electrode area. Testing the relation of the obtained limit intensity shows these graphs: First, the standard ferrocene measurement with common parameters was tried:

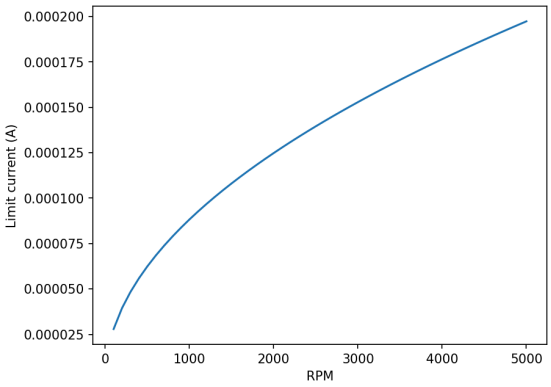


Figure 10: Standard limit intensity measurement with ferrocene

The afromentioned proportionally could be checked, obtaining $i_L = 2 \quad 10^{-4} A$. Next the linear relation with concentration and area was tested. With the same parameters, only varying concentration first, then area, the resulting plots were obtained:

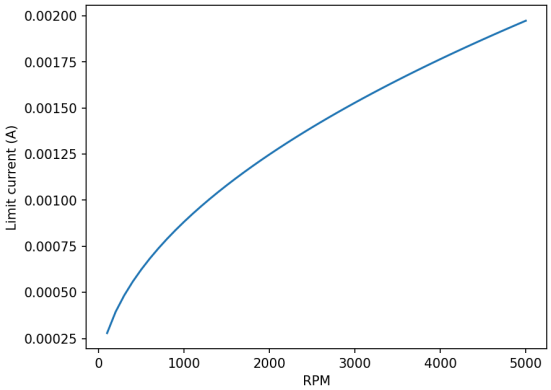


Figure 11: Limit current for 10 times the concentration of reactants

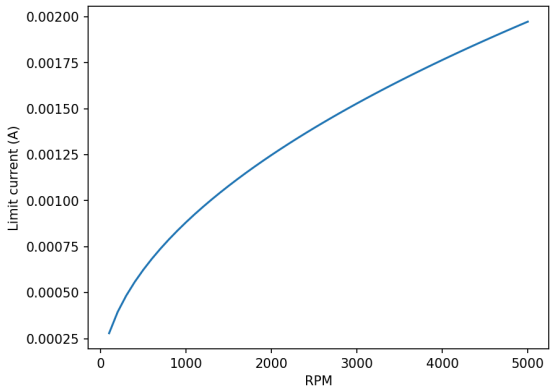


Figure 12: Limit current for 10 times the area of the electrode

7.3 Battery Discharge

To test whether the discharge curve from the output graph is close to a experimental discharge curve is redundant, as the output graph is a literal interpolation of the real graph. This does not mean no test is needed to be done, as parameters can be given as an input. The terminal voltage obtained from Eq.19 has two terms: OCV is directly interpolated from experimental output, and the other is a product of $I(t)$, which we assume constant, and R_{int}, which is a constant. The only modification IR_{int} can bring is a constant change on the y-axis, but not a slope change.

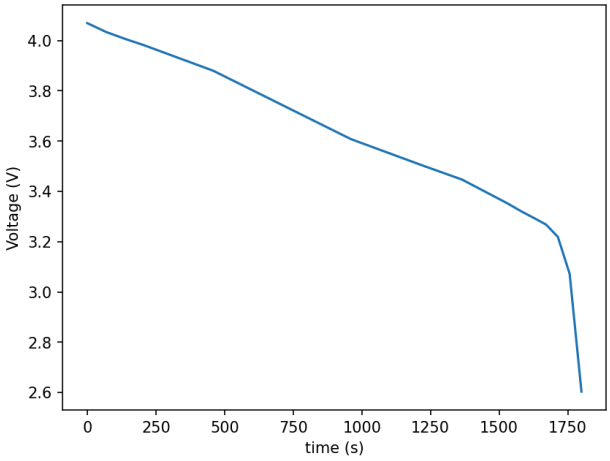


Figure 13: Battery Discharge

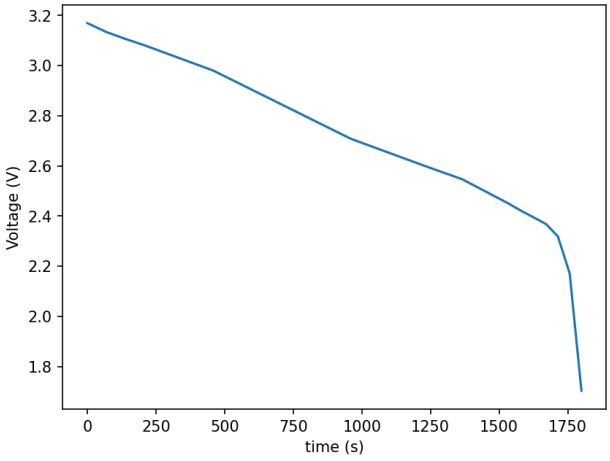


Figure 14: Battery Discharge for 10 times IR_{int}

The rate at which SOC(t) changes, through Eq.37, is the only meaningful change, as can be seen in the next graph:

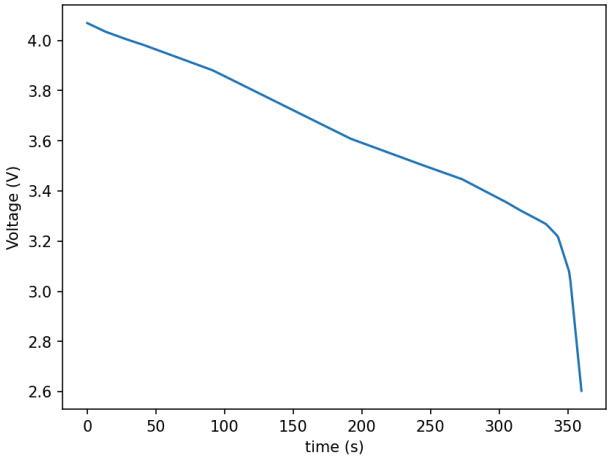


Figure 15: Battery Discharge for $1/5\ Q_{\text{nom}}$

As seen, even if the time until full discharge is shorter, the discharge curve still mantains the same form. That's because at constant discharge current the form of the discharge curve is intrinsic to the chemistry redox reactions within the battery.

7.4 Cyclic Voltammetry

The core of this project, the cyclic voltammetry, introduces three numerical methods, with different precision from each other, one implict and two explicit. To properly test the resulting voltammogram, the following must be considered: Peak separation (ΔE_p), the ratio of anodic current density $j_{p,a}$ to cathodic current density $j_{p,a}$, Randles-Sevcik linearity and numerical stability and precision between numerical methods. To ensure the highest precision with the tests, Runge-Kutta 4 will be chosen as the reference method.

A common voltammogram for a linear sweep has two peaks, one positive, denoting the anodic current from the oxidation, and one negative, denoting the cathodic current from the reduction. The standard ferrocene test will be the system considered.

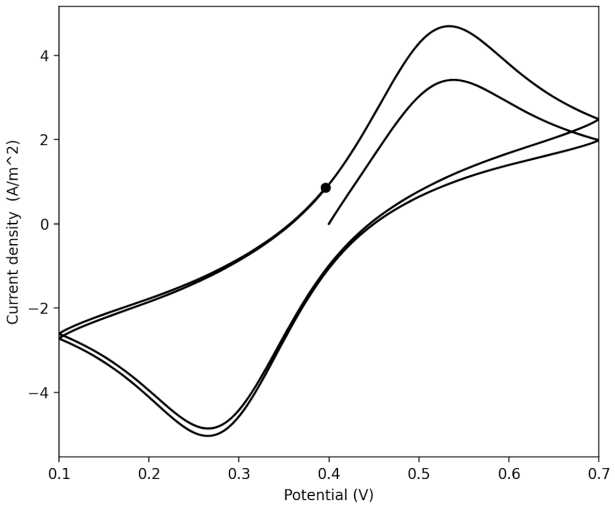


Figure 16: Voltammogram, ferrocene, $C_{O,\text{bulk}} = 1\ \text{mM}, C_{R,\text{bulk}} = 1\ \text{mM}$

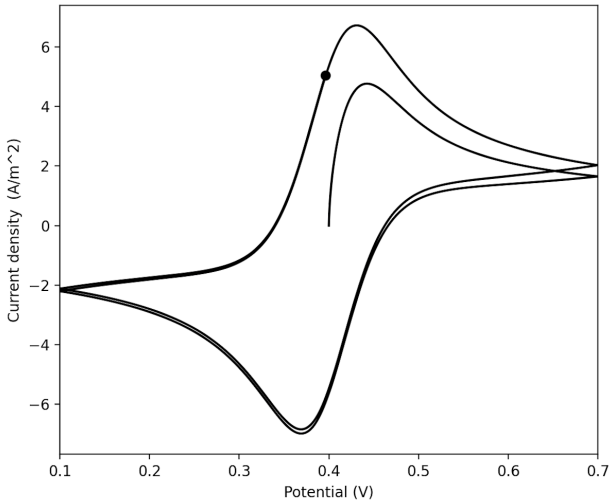


Figure 17: As figure 16, but $\frac{1}{1000}k_0$

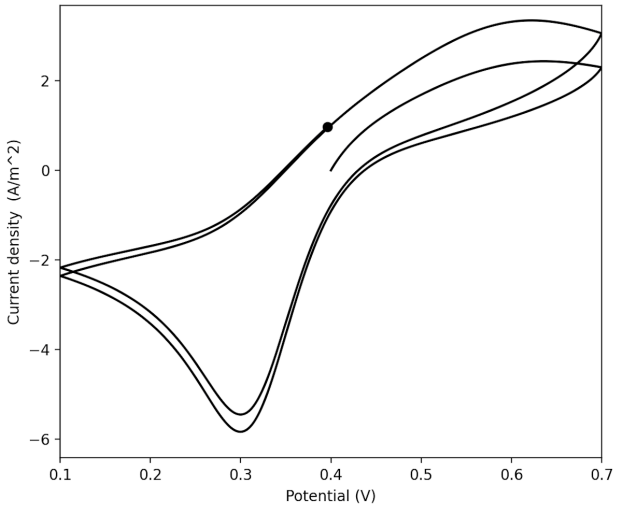


Figure 18: As figure 16, but $\beta_c = 0.75$

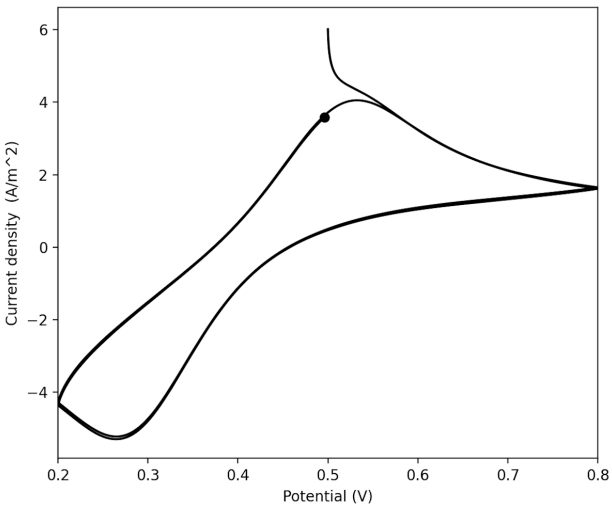
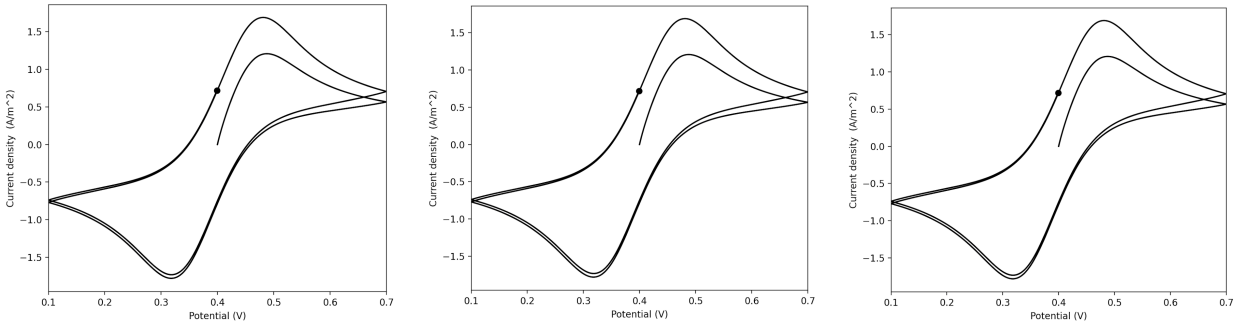


Figure 19: As figure 16, but E_{start} not at the equilibrium potential

A strange peak appears at the beggining of the simulation when starting at a different potential than the equilibrium potential. The peak appears since the system is found suddenly in an unstable potential with concentrations physically impossible to achieve at those potentials, so the excess is all oxidized suddenly. Relaxation was tried to solve the problem, however proper relaxation needed more loop iterations so it was discarded. Implementation of migration on top of the initial diffusion was considered, but the addition of migration brought non-linearty to the interior concentration too. As Crank-Nicolson was already implemented, the

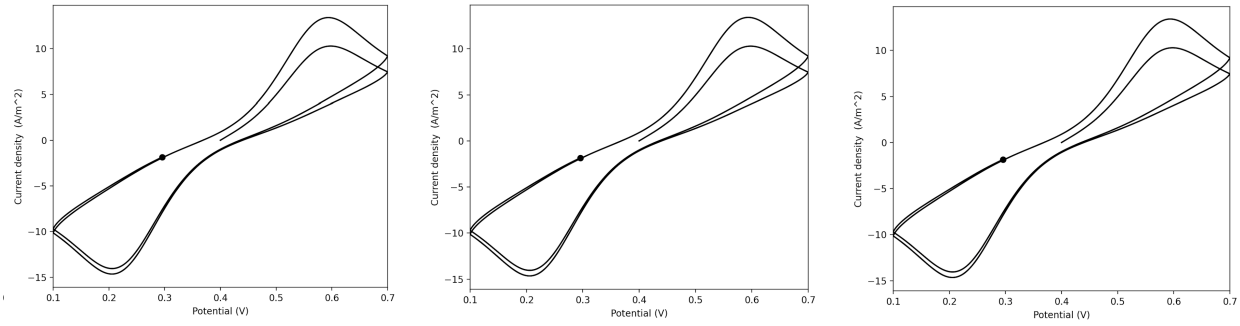
decision of not adding migration contribution was chosen.
Finally, to test the precision of the three methods, a $\frac{1}{10}$ of the baseline's scan rate is tried:



Heun (RK2) Runge-Kutta 4 Crank-Nicolson

Figure 20: Comparison at a low time step between all methods

And 10 times of the baseline's scan rate:



Heun (RK2) Runge-Kutta 4 Crank-Nicolson

Figure 21: Comparison at a high time step between all methods

As it can be seen, there's not much difference between the precision for the test scans, but it can be seen how Randall-Sevcik's relation is followed.

8. Improvements and optimizations

The current program can be improved and optimized further, however, as the program was made prioritizing a pedagogical viewpoint some limits were assumed. This section will cover further improvements for the overall structure and performance of the program.

First, the UI is very minimal, only a few lines of text printed to the terminal. The UI could be improved greatly by using tools like Streamlit or Custom Tkinter. Streamlit could be used to turn the program into a web app, providing it of a simple yet elegant interface, while Custom Tkinter would make of the program an application, with its own window and a personalized interface, tailored to the code written.

To make the code more readable, Object Oriented Programming (OOP), along classes and methods could make the code modular and more readable, in case other programmers join the codebase. Other external modules would also make the code more compact. Using already existing functions from well known modules will make the code clearer and possibly faster, as several optimizations can be done for the solvers.

Another aspect that could be improved is the way inputs are provided to the program. Thanks to the “error” function which handles bad inputs the program won’t crash, but for batches of simulations, manually inputing each parameter every time a simulation is needed might be a cause of unnecessary burden for the user. Taking as input a .json file or a .yaml would be a great improvement over the current input system. Although less intuitive for new users, the easy reproductibility and the possibility of providing multiple inputs would allow for easy parallelization and for supercomputers to not wait idle waiting while another input is being sent.

On the topic of parallelization, python 3.14 removes the Global Interpreter Lock (GIL) allowing for true multithreading in Python 3. This update allows for the python interpreter not to be locked into only being used by one thread. By rewritting the program for Python 3.14, and importing the new multithreading module, making correct use of threading, an exponential performance improvement on current code would be observed. Without any version change, the program can be made concurrent by Inter Process Comunication (IPC), creating new processes and comunicating them. Although more expensive than multithreading, can be achieved without any version change.

Python’s interpreter is a massive slowdown compared to compiled languages, rewritting the project in other programming languages which are compiled, like C, C++, or new programming languages like Zig or Rust would benefit from compiler optimizations, enabling a huge optimization. It should be mentioned that if performance is the only objective even if compiled, programming languages like Java or C# should be avoided, as not requiring memory management implies a underlying garbage collector, which slows down the runtime process. To truly optimize it fully, the cache should be taken advantage of as much as possible. Compressing the data needed in such a way all is contained in a the cache will turn the program into a very performant simulator. Avoiding cache flushes can lessen the runtime by orders of magnitude, especially if memory is stored in L1 cache or L2 instead. Albeit buying CPUs with megabytes of cache can improve cache hits, optimizing the code will be the principal manner to achieve true performance.

Finally, vectorization through Single Instruction, Multiple Data (SIMD) coupled with branchless programming will take the code to another level of performance, as no branch predictions will be needed for the CPUs and SIMD instructions perform multiple operations in one CPU cycle.

9. Conclusions

To conclude this TFG, the author would like to think that the pedagogical objective has been achieved. A program has been made, which simulates accurately electrochemical phenomena using numerical methods, as seen in the sections before.

10. Refences and notes

1. Bard, A. J., & Faulkner, L. R. (2001). Electrochemical Methods: Fundamentals and Applications. John Wiley & Sons.
2. Atkins, P., & de Paula, J. (2014). Physical chemistry (10th ed.). Oxford University Press.
3. Rumble, J. R. (Ed.). (2021). CRC handbook of chemistry and physics (102nd ed.). CRC Press.
4. Zhang, S., Zhang, Y., Liu, Y., & Zhang, J. (2022). SOC estimation of lithium-ion battery based on an improved equivalent circuit model and an improved adaptive unscented Kalman filter. Energies, 15(23), 9142. https://doi.org/10.3390/en15239142
5. Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). Numerical recipes: The art of scientific computing (3.ª ed.). Cambridge University Press.
6. Elgrishi, N., Rountree, K. J., McCarthy, B. D., Rountree, E. S., Eisenhart, T. T., y Dempsey, J. L. (2018). A Practical Beginner's Guide to Cyclic Voltammetry. Journal of Chemical Education, 95(2), 197–206 . https://doi.org/10.1021/acs.jchemed.7b00361.
7. Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., & del Río, J. F. (2020). Array programming with NumPy. Nature, 585(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

11. Acronyms

- $[O]$: oxidized species
- $[R]$: reduced species
- e^- : electron with its negative charge
- ν_a : anodic reaction rate
- ν_c : cathodic reaction rate
- k_a : anodic constant
- k_c : cathodic constant
- k_b : Boltzmann's constant
- $\Delta G^\ddagger$: Gibbs activation free energy
- R : ideal gas constant
- T : temperature
- j : current density
- j_0 : exchange current density
- i : current
- I : current
- I_0 : exchange current
- E_0 : formal potential
- n : number of electrons of the reaction
- A : area
- D : diffusion coefficient
- J : diffusion flux
- $O(\Delta x^n)$: Error of order n
- SOC: State of Charge
- OCV: Open Current Voltage
- RK: Runge-Kutta
- ODE: Ordinary Differential Equation
- PDE: Partial Differential Equation
- UI: User Interface
- LHS: Left Hand Side
- RHS: Right Hand Side
- parse: analyse text data and transform it to manageable data

APPENDICES

PROGRAM CODE

```python
#!/usr/bin/env python3
from ButlerVolmer import ButlerVolmer
from RotatingDiskElectrode  import RotatingDiskElectrode
from BatteryDischarge import BatteryDischarge
from CyclicVoltammetry import CyclicVoltammetry
import utils

# INFO: Prints the mode selections for the simulations and Runge-Kutta methods.
def selection(mode: int) -> int:
    if mode == 0:
        print("""Select the desired use mode:
              (1) Butler-Volmer
              (2) Rotating Disk Electrode
              (3) Battery discharge
              (4) Cyclic voltammetry
              (5) Exit""")
    elif mode == 1:
        print("""Select the desired algorithm:
              (1) Crank-Nicolson + Thomas + Newton
              (2) Runge-Kutta""")
    else:
        print("""Select the desired order:
              (1) Runge-Kutta 2
              (2) Runge-Kutta 4""")
    resp = int(input())
    return resp

# INFO: Loads the databses and selects which simulation the user desires, handling
#       parsing errors.
def main() -> None:
    try:
        print("Loading.", end='')
        utils.stdRedPotFile = utils.parse("standard_potentials.csv")
        print(".", end='')
        utils.BatteryValuesFile = utils.convert(utils.parse("BatteryValues.csv"))
        print(".")
        resp = selection(0)
        while resp not in (1, 2, 3, 4, 5):
            print("Value not allowed, select a number from 1-5")
            resp = selection(0)
        match resp:
            case 1:
                print("Butler-Volmer has been selected")
                try:
                    ButlerVolmer()
                except Exception as e:
                    print("Error:", e)
            case 2:
                print("Rotating Disk Electrode has been selected")
                try:
                    RotatingDiskElectrode()
                except Exception as e:
                    print("Error:", e)
            case 3:
                print("Battery discharge has been selected")
                try:
                   BatteryDischarge()
                except Exception as e:
                    print("Error:", e)
            case 4:
                print("Cyclic voltamperometry has been selected")
                resp = selection(1)
                while resp not in (1, 2):
                    print("Value not allowed, select a valid number")
                    resp = selection(1)
                order = 0
                if resp == 2:
                    order = selection(2)
                    while resp not in (1, 2):
                        print("Value not allowed, select a valid number")
                        order = selection(2)
                try:
                    CyclicVoltammetry(2 * order)
                except Exception as e:
                    print("Error:", e)
            case 5:
                exit()
    except Exception as e:
        print("Value not allowed, select a number from 1-5")
        main()

if __name__ == "__main__":
    main()
```

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

stdRedPotFile : dict[str, float] = {}
BatteryValuesFile : dict[float, float] = {}

# INFO: Parses the .csv files and stores them in dictionaries, that act as
#       databses. Throws an error if the file is not found.
def parse(name: str) -> dict[str, float]:
    d: dict[str, float] = {}
    try:
        with open(name, "r") as f:
            next(f)
            for line in f:
                key, value = [s.strip() for s in line.split(",")]
                d[key] = float(value)
            return d
    except OSError:
        print(f'Could not read file: ', name)
        exit()

# INFO: Converts a dictionary with string keys into another dictionary, the
#       new one using float keys.
def convert(strdict: dict[str, float]) -> dict[float, float]:
    d: dict[float, float] = {}
    for key, value in strdict.items():
        d[float(key)] = value
    return d

# INFO: Parses the number of electrons from the half reaction input, checking
#       the number before "e-". If no number is found, 1 is returned.
#       new one using float keys.
def getElectrons(elec: str) -> int:
    mult_el = elec[elec.find("e-") - 2]
    if str.isdigit(mult_el):
        n_el = int(mult_el)
    else:
        n_el = 1
    return n_el

# INFO: Parses the stochiometric coefficient from the half reaction input, checking
#       the first number in the LHS and RHS of the reaction. If no number is found,
#        1 is returned.
def getStoichCoeffs(elec: str) -> tuple[int, int]:

    parts = elec.split("->")
    ox, red = parts[0].strip(), parts[1].strip()

    if str.isdigit(ox[0]):
        nu_ox = int(ox[0])
    else:
        nu_ox = 1

    if str.isdigit(red[0]):
        nu_red = int(red[0])
    else:
        nu_red = 1

    return nu_red, nu_ox

# INFO: Plots a graph using matplotlibs.pyplot module. Takes as inputs the both
#       axis of the graph and their labels. The asked input is the name of the
#       .png that will be saved.
def plotGraph(x: np.ndarray, y: np.ndarray, x_name: str, y_name: str) -> None:
    plt.plot(x, y)
    plt.xlabel(x_name)
    plt.ylabel(y_name)
    name = input("Save the plot as:")
    out = "/tmp/" + name + ".png"
    plt.savefig(out, dpi=150, bbox_inches='tight')
    print(f'Saved plot as: {out}')
    plt.close()

```

```python
# INFO: Handles errors and ensures physically sense inputs. The input decides
#       which case should it go into.
def error(type: str) -> int | float:
    temp = 0
    match type:
        case "n_el":
            try:
                temp = int(input("Set the number of electrons involved in the reaction:\n"))
                while temp < 1 or temp > 4:
                    print('Error: the number of electrons cannot be less than 1 or larger than 4')
                    temp = error("n_el")
            except Exception as e:
                print("Error:", e)
                temp = error("n_el")
        case "n_step":
            try:
                temp = int(input("Set the number of steps calculated:\n"))
                while temp < 10:
                    print('Error: the number of steps cannot be less than 10')
                    temp = error("n_step")
            except Exception as e:
                print("Error:", e)
                temp = error("n_step")
        case "rpm_max":
            try:
                temp = int(input("Set the max RPM:\n"))
                while temp < 500:
                    print('Error: the max RPM must be at least 500')
                    temp = error("rpm_max")
            except Exception as e:
                print("Error:", e)
                temp = error("rpm_max")
        case "temperature":
            try:
                temp = float(input("Set the desired temperature (K):\n"))
                if temp == 0:
                    temp = 1e-12
                while temp < 0:
                    print('Error: the temperature cannot be negative')
                    temp = error("temperature")
            except Exception as e:
                print("Error:", e)
                temp = error("temperature")
        case "endtime":
            try:
                temp = int(input("Set time limit in seconds (0 for no time limit):\n"))
                while temp < 0:
                    print('Error: time cannot be negative')
                    temp = error("endtime")
            except Exception as e:
                print("Error:", e)
                temp = error("endtime")
        case "n_cycles":
            try:
                temp = int(input("Set the desired number of cycles:\n"))
                while temp <= 0:
                    print('Error: the number of cycles should be at least 1')
                    temp = error("n_cycles")
            except Exception as e:
                print("Error:", e)
                temp = error("n_cycles")
        case "srate":
            try:
                temp = float(input("Set scan rate (V/s):\n"))
                while temp < 0:
                    print('Error: scan rate cannot be negative')
                    temp = error("srate")
            except Exception as e:
                print("Error:", e)
                temp = error("srate")
        case "diff":
            try:
                temp = float(input("Set the diffusivity (m^2/s):\n"))
                while temp < 0:
                    print('Error: the diffusivity cannot be negative')
                    temp = error("diff")
            except Exception as e:
                print("Error:", e)
                temp = error("diff")
        case "viscosity":
            try:
                temp = float(input("Set the viscosity (m^2/s):\n"))
                while temp < 0:
                    print('Error: the viscosity cannot be negative')
                    temp = error("viscosity")
            except Exception as e:
                print("Error:", e)
                temp = error("viscosity")
        case "area":
            try:
                temp = float(input("Set the electrode area (m^2):\n"))
                if temp == 0:
                    print('Error: area cannot be 0, set to 1')
                    temp = 1.0
                while temp < 0:
                    print('Error: concentration cannot be negative')
                    temp = error("area")
            except Exception as e:
                print("Error:", e)
                temp = error("area")
```

```
        case "C0":
            try:
                temp = float(input("Set the initial concentration (mol/m^3):\n"))
                if temp == 0:
                    temp = 1.0
                while temp < 0:
                    print('Error: concentration cannot be negative')
                    temp = error("C0")
            except Exception as e:
                print("Error:", e)
                temp = error("C0")
        case "Cob":
            try:
                temp = float(input("Set the concentration of the oxidized species (mol/m^3):\n"))
                if temp == 0:
                    temp = 1.0
                while temp < 0:
                    print('Error: concentration cannot be negative')
                    temp = error("Cob")
            except Exception as e:
                print("Error:", e)
                temp = error("Cob")
        case "Crb":
            try:
                temp = float(input("Set the concentration of the reduced species (mol/m^3):\n"))
                if temp == 0:
                    temp = 1.0
                while temp < 0:
                    print('Error: concentration cannot be negative')
                    temp = error("Crb")
            except Exception as e:
                print("Error:", e)
                temp = error("Crb")
        case "c_red":
            try:
                temp = float(input("Set the concentration of the reduced species in mol/m^3 (1 for solids):\n"))
                if temp == 0:
                    temp = 1.0
                while temp < 0:
                    print('Error: concentration cannot be negative')
                    temp = error("c_red")
            except Exception as e:
                print("Error:", e)
                temp = error("c_red")
        case "c_ox":
            try:
                temp = float(input("Set the concentration of the oxidized species in mol/m^3 (1 for solids):\n"))
                if temp == 0:
                    temp = 1.0
                while temp < 0:
                    print('Error: concentration cannot be negative')
                    temp = error("c_ox")
            except Exception as e:
                print("Error:", e)
                temp = error("c_ox")
        case "intensity":
            try:
                temp = float(input("Set the intensity of the battery (A):\n"))
                if temp == 0:
                    temp = 1e-12
                if temp < 0:
                    temp *= -1
            except Exception as e:
                print("Error:", e)
                temp = error("intensity")
        case "resistance":
            try:
                temp = float(input("Set the internal resistance of the battery (ohms):\n"))
                if temp == 0:
                    temp = 1e-12
                while temp < 0:
                    print('Error: resistance cannot be negative')
                    temp = error("resistance")
            except Exception as e:
                print("Error:", e)
                temp = error("resistance")
        case "Q_nom":
            try:
                temp = float(input("Set the nominal capacity of the battery (A/s):\n"))
                if temp == 0:
                    temp = 1e-12
                while temp < 0:
                    print('Error: the nominal capacity cannot be negative')
                    temp = error("Q_nom")
            except Exception as e:
                print("Error:", e)
                temp = error("Q_nom")
        case "jo":
            try:
                temp = float(input("Set initial jo (A/m^2):\n"))
            except Exception as e:
                print("Error:", e)
                temp = error("jo")
        case "beta_c":
            try:
                temp = float(input("Set the cathodic symmetry factor:\n"))
                while temp < 0 or temp > 1:
                    print('Error: the symmetry factor must be between 0-1')
                    temp = error("beta_c")
            except Exception as e:
```

```
                print("Error:", e)
                temp = error("beta_c")
        case "SOC":
            try:
                temp = float(input("Set state of charge of the battery (%):\n"))
                while temp < 0:
                    print('Error: The state of charge of the battery cannot be negative')
                    temp = error("SOC")
                while temp > 100:
                    print('Error: The state of charge must be 100% or less')
                    temp = error("SOC")
            except Exception as e:
                print("Error:", e)
                temp = error("SOC")
        case "k0":
            try:
                temp = float(input("Set k0 (m/s):\n"))
            except Exception as e:
                print("Error:", e)
                temp = error("k0")
        case "j0":
            try:
                temp = float(input("Set j0 (A/m^2):\n"))
            except Exception as e:
                print("Error:", e)
                temp = error("j0")
        case "eta":
            try:
                temp = float(input("Set the overpotential (V):\n"))
            except Exception as e:
                print("Error:", e)
                temp = error("eta")
        case "E_start":
            try:
                temp = float(input("Set the starting potential for the sweep (V):\n"))
            except Exception as e:
                print("Error:", e)
                temp = error("E_start")
        case "E_vertex":
            try:
                temp = float(input("Set the peak potential for the sweep (V):\n"))
            except Exception as e:
                print("Error:", e)
                temp = error("E_vertex")
        case "E_ap":
            try:
                temp = float(input("Set the applied potential (V):\n"))
            except Exception as e:
                print("Error:", e)
                temp = error("E_ap")
        case "E_ref":
            try:
                temp = float(input("Set the potential of the reference electrode, 0V for SHE (V):\n"))
            except Exception as e:
                print("Error:", e)
                temp = error("E_ref")
        case _:
            print(f"ERROR, {type} unbound")

    return temp
```

```python
import math as m
import numpy as np
import scipy.constants as cnt
import utils
import error as e
import matplotlib.pyplot as plt

# INFO: Parses the user input and obtains the standard equilibrium potential via
#       the standard reduction potential database. Returns both the user input
#       string and the standard reduction potential of the input half reaction.
def getEo() -> tuple[str, float]:

    try:
        resp = input(f'Input the half reaction happening on the working electrode:\n')
        value = utils.stdRedPotFile.get(resp)
        if value == None:
            resp, value = getEo()
    except Exception as e:
        print("Error:", e)
        resp, value = getEo()
    return resp, value

# INFO: Provides the corresponding equilibrium potential using the nerst equation.
#       Returns the equilibrium potential and the number of electrons of the reaction.
def getEeq(c_red: float, c_ox: float, T: float) -> tuple[float, int]:
    R = cnt.gas_constant
    F = cnt.value("Faraday constant")

    str_half, Eo = getEo()
    print(f'Eo half-reaction = {Eo}')

    n_el = utils.getElectrons(str_half)
    nu_red, nu_ox = utils.getStoichCoeffs(str_half)

    E_eq =  Eo - (R * T / (n_el * F)) * m.log(c_red ** nu_red / c_ox ** nu_ox)

    return E_eq, n_el


# INFO: Calculates the current density in the range [-abs(overpotential), abs(overpotential)].
#       Plots the current density vs the overpotential range obtained.
def ButlerVolmer() -> None:
    F = cnt.value("Faraday constant")
    R = cnt.gas_constant
    T = e.error("temperature")

    c_red = e.error("c_red")
    c_ox = e.error("c_ox")
    j0 = e.error("j0")
    beta_c = e.error("beta_c")
    beta_a = 1.0 - beta_c
    E_ap = e.error("E_ap")
    E_ref = e.error("E_ref")
    E_eq, n_el = getEeq(c_red, c_ox, T)

    eta = E_ap - E_eq - E_ref
    nsteps = 1000 * abs(eta)
    etas = np.linspace(-abs(eta), abs(eta), int(nsteps))
    ja = np.exp(F * n_el * beta_a * etas / (T * R))
    jc = np.exp(F * n_el * -beta_c * etas / (T * R))
    currents = j0 * (ja - jc)

    utils.plotGraph(etas, currents,'Overpotential (V)', 'Current density (A/m^2)')
```

```python
import numpy as np
import error as e
import utils
import scipy.constants as cnt
import matplotlib.pyplot as plt


# INFO: Calculates the limit current using the Levich equation. Plots the current vs
#       the RPM, then prints the limit current.
def RotatingDiskElectrode() -> None:
    n_el = e.error("n_el")
    F = cnt.value("Faraday constant")
    D_coef = e.error("diff")
    C0 = e.error("C0")
    nu = e.error("viscosity")
    A = e.error("area")
    rpm_max = e.error("rpm_max")

    rpm = np.linspace(100, rpm_max, 50)
    omega = 2 * np.pi * rpm / 60.0

    delta = 1.61 * (D_coef**(1/3)) * (omega**(-0.5)) * (nu**(1/6))
    i_lim = n_el * F * D_coef * C0 * A / delta

    utils.plotGraph(rpm, i_lim, "RPM", "Limit current (A)")
    print(f"With {rpm[-1]:.0f} RPM, the limit current is {i_lim[-1]:.4f} A")
```

```python
import numpy as np
import error as e
import matplotlib.pyplot as plt
import utils


# INFO: Linearly interpolates the input SOC's corresponding potential value,
#       using as interpolation points experimentally  obtained values for a
#       Li-ion battery from the BatteryValues database.  Returns the corresponding
#       linearly interpolated potential.
def interpOCV(val: float) -> float :
    x1 = 0.0000
    x2 = 0.0000

    keys = list(utils.BatteryValuesFile.keys())
    if val == keys[-1]:
        return utils.BatteryValuesFile[keys[-1]]

    for i in range(1, len(keys)):
        x1 = keys[i - 1]
        x2 = keys[i]
        if x1 <= val <= x2:
            y1 = utils.BatteryValuesFile[x1]
            y2 = utils.BatteryValuesFile[x2]
            t = (y2 - y1) / (x2 - x1)
            return y1 + t * (val - x1)
    return 0.0

# INFO: Calculates the terminal voltage of the Li-ion battery, looping until
#       the time from endtime is reached or the battery's fully discharged.
#       Plots the discharge curve (Voltage vs time):
def BatteryDischarge() -> None:
    I = e.error("intensity")
    R_int = e.error("resistance")
    Q_nom = e.error("Q_nom")
    State = e.error("SOC") / 100
    endtime = float(e.error("endtime"))
    t = 0.0
    dt = 1.0
    voltages = []
    dSOC = - I * dt / (Q_nom)
    if endtime == 0.0:
        endtime = 9e34

    while State > 0 and t <= endtime:
        V = interpOCV(State) - I * R_int
        voltages.append(V)
        State += dSOC
        t += dt
    voltages = np.array(voltages)
    times = np.arange(len(voltages) * dt)

    utils.plotGraph(times, voltages, 'time (s)', 'Voltage (V)')
```

```python
import numpy as np
import math as m
import scipy.constants as cnt
import error as e
import utils
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# INFO: Creates and saves a GIF of concentration profiles vs time and a voltammogram
#       using Pillow. The name of the GIF is handled as user input.
def animate_cv(t: np.ndarray, x: np.ndarray, Co_hist: np.ndarray, Cr_hist: np.ndarray, potentials: np.ndarray, currents: np.ndarray):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

    ax1.set_xlim(0, np.max(x))
    ax1.set_ylim(0, np.max(Co_hist) * 1.1)
    ax1.set_xlabel('Distance from electrode (UA)')
    ax1.set_ylabel('Concentration (mol/m^2)')
    line_o, = ax1.plot([], [], color='blue', label='[Ox]')
    line_r, = ax1.plot([], [], color='red', label='[Red]')
    ax1.legend(loc='upper right')

    ax2.set_xlim(np.min(potentials), np.max(potentials))
    ax2.set_ylim(np.min(currents) * 1.1, np.max(currents) * 1.1)
    ax2.set_xlabel('Potential (V)')
    ax2.set_ylabel('Current density  (A/m^2)')
    line_j, = ax2.plot([], [], color='black', lw=1.5)
    point_j, = ax2.plot([], [], 'ko')

    plt.tight_layout()

    num_frames = int(10 * t[-1])
    step = max(1, len(potentials) // num_frames)
    frames = range(0, len(potentials), step)

    def update(i):
        line_o.set_data(x, Co_hist[i])
        line_r.set_data(x, Cr_hist[i])

        line_j.set_data(potentials[:i], currents[:i])
        point_j.set_data([potentials[i]], [currents[i]])

        return line_o, line_r, line_j, point_j

    ani = FuncAnimation(fig, update, frames=frames, blit=True, interval=50)

    name = input("Save the plot as:")
    out = "/tmp/" + name + ".gif"
    ani.save(out, writer='pillow', fps=20)
    print(f"Saved CV analysis as: {out}")
    plt.close()

# INFO: Creates two numpy ndarrays, both the time array and the potential triangular
#       waveform array. Returns the arrays before mentioned.
def init_time_potential(E_start: float, E_vertex: float, v: float, dt: float, n_cycles: int) -> tuple[np.ndarray, np.ndarray]:
    E_min = E_start - E_vertex
    E_max = E_start + E_vertex

    t_cycle = 4.0 * E_vertex / v
    t_max = n_cycles * t_cycle
    t = np.linspace(0.0, t_max, int(round(t_max / dt)) + 1)

    phase_shift = (E_start - E_min) / v
    u = np.mod(t + phase_shift, t_cycle) / t_cycle
    tri = 1.0 - 2.0 * np.abs(u - 0.5)
    E = E_min + tri * (E_max - E_min)

    return t, E

# INFO: Parses the user input and obtains the standard equilibrium potential via
#       the standard reduction potential database. Returns the number of electrons
#       of the reaction and the standard reduction potential of the input half reaction.
def getVal() -> tuple[int, float] :

    value = None
    n_el = 0
    while value == None:
        resp = input("Input the half reaction corresponding to the oxidation:\n")
        value = utils.stdRedPotFile.get(resp)
        n_el = utils.getElectrons(resp)
    return n_el, value

# INFO: Implemention of the Thomas algorithm, a, b and c are the three diagonals of
#       the triadiagonal matrix, d is the solution vector and x is the vector of
#       unknowns. Returns x solved.
def thomas(a_temp, b_temp, c_temp, d_temp):
    n = len(d_temp)

    a, b, c, d = map(np.array, (a_temp, b_temp, c_temp, d_temp))
    for i in range(1, n):
        m = a[i] / b[i - 1]
        b[i] = b[i] - m * c[i - 1]
        d[i] = d[i] - m * d[i - 1]

    x = np.zeros(n)
    x[-1] = d[-1] / b[-1]

    for i in range(n - 2, -1, -1):
        x[i] = (d[i] - c[i] * x[i+1]) / b[i]
    return x
```

```python
# INFO: Calculates the vector of solutions for the Thomas algorithm. Returns the
#       calculated vector.
def rhs(conc: np.ndarray, c0: float, bulk: float, lam: float) -> np.ndarray:
    res = 0.5 * lam * conc[0:-2] + (1.0 - lam) * conc[1:-1] + 0.5 * lam * conc[2:]
    res[0] += 0.5 * lam * c0
    res[-1] += 0.5 * lam * bulk
    return res

# INFO: Calculates the reaction rate and the partial derivatives for the surface
#       concentrations and returns them.
def reaction_rate_and_partials(Co0: float, Cr0: float, E_ap: float, params: dict[str, int | float]):

    R = params["R"]
    T = params["T"]
    F = params["F"]
    k0 = params["k0"]
    n_el = params["n_el"]
    beta_c = params["beta_c"]
    beta_a = 1 - beta_c
    E_std = params["E_std"]

    f = F / (R * T)
    xi = f * (E_ap - E_std)

    k_red = k0 * m.exp(-beta_c * n_el * xi)
    k_ox  = k0 * m.exp((beta_a) * n_el * xi)

    r = k_ox * Cr0 - k_red * Co0

    dr_dCo0 = -k_red
    dr_dCr0 = k_ox

    return r, dr_dCo0, dr_dCr0

# INFO: Solves the differential equations using the Thomas algorithm for the
#       interior concentrations and Newton-Raphson for the surface concentrations.
#       Returns the solved concentrations and the reaction rate.
def newton_thomas(Co: np.ndarray, Cr: np.ndarray, E_ap: float, lam: float, params: dict[str, int | float]) -> tuple[float, float, np.ndarray, np.ndarray, float]  :
    tol=1e-9
    max_iter=12

    dx = params["dx"]
    D_coef = params["D"]

    n_unknowns = Co.size - 2

    low_diag = np.full(n_unknowns, -0.5 * lam)
    mid_diag = np.full(n_unknowns, (1.0 + lam))
    upp_diag = np.full(n_unknowns, -0.5 * lam)

    u = np.array([Co[0], Cr[0]])
    for it in range(max_iter):
        Co0, Cr0 = u[0], u[1]

        do = rhs(Co, Co0, Co[-1], lam)
        dr = rhs(Cr, Cr0, Cr[-1], lam)

        Co_interior = thomas(low_diag.copy(), mid_diag.copy(), upp_diag.copy(), do)
        Cr_interior = thomas(low_diag.copy(), mid_diag.copy(), upp_diag.copy(), dr)

        r_surf, dr_dCo0, dr_dCr0 = reaction_rate_and_partials(Co0, Cr0, E_ap, params)

        Co1 = Co_interior[0]
        Cr1 = Cr_interior[0]

        resid_ox = Co0 - Co1 - dx / D_coef * r_surf
        resid_red = Cr0 - Cr1 + dx / D_coef * r_surf


        if max(abs(resid_ox), abs(resid_red)) < tol:
            return Co0, Cr0, Co_interior, Cr_interior, r_surf

        b_sens = np.zeros(n_unknowns)
        b_sens[0] = 0.5 * lam
        s_vec = thomas(low_diag, mid_diag, upp_diag, b_sens)
        s_factor = s_vec[0]

        J11 = 1.0 - s_factor - dx / D_coef * dr_dCo0
        J12 = - dx / D_coef * dr_dCr0

        J21 = dx / D_coef * dr_dCo0
        J22 = 1.0 - s_factor + dx / D_coef * dr_dCr0

        J = np.array([[J11, J12], [J21, J22]])
        R_vec = np.array([resid_ox, resid_red])

        try:
            delta = np.linalg.solve(J, -R_vec)
        except np.linalg.LinAlgError:
            delta = -R_vec * 0.1

        u += delta
        u[u < 0] = 1e-15

    return u[0], u[1], Co_interior, Cr_interior, r_surf
```

```python
# INFO: Analitically solves the concentrations in the surface of the electrode for
#       the Runge-Kutta methods. Returns the solved surface concentrations along
#       with the reduction and oxidation rate constants.
def solve_surface_analytic(Co1: float, Cr1: float, E_ap: float, params: dict[str, int | float]) -> tuple[float, float, float, float]:
    R = params["R"]
    T = params["T"]
    F = params["F"]
    k0 = params["k0"]
    n_el =  params["n_el"]
    beta_c = params["beta_c"]
    beta_a = 1 - beta_c
    E_std = params["E_std"]
    D_coef = params["D"]
    dx = params["dx"]

    f = F / (R * T)
    xi = f * (E_ap - E_std)
    k_red = k0 * m.exp(-beta_c * n_el * xi)
    k_ox  = k0 * m.exp(beta_a * n_el * xi)

    alpha = D_coef / dx

    A = np.array([
        [alpha + k_red, -k_ox],
        [-k_red, alpha + k_ox]
    ])
    b_vec = np.array([alpha * Co1, alpha * Cr1])

    res = np.linalg.solve(A, b_vec)
    return res[0], res[1], k_ox, k_red

# INFO: Calculates the derivative of the interior concentrations with respect to
#       time by discretizing the spatial dimension, into a coupled system of
#       differential equations. Returns the derivatives of the system and the
#       reaction rate at the surface of the electrode.
def get_derivatives(t: float, Co_int: np.ndarray, Cr_int: np.ndarray, E_now: float, params: dict[str, int | float]) -> tuple[np.ndarray, np.ndarray, float]:
    D = params["D"]
    dx = params["dx"]
    bulk_Co = params["bulk_Co"]
    bulk_Cr = params["bulk_Cr"]

    Co0, Cr0, k_ox, k_red = solve_surface_analytic(Co_int[0], Cr_int[0], E_now, params)

    Co_full = np.concatenate(([Co0], Co_int, [bulk_Co]))
    Cr_full = np.concatenate(([Cr0], Cr_int, [bulk_Cr]))

    dCo_dt = D * (Co_full[2:] - 2 * Co_full[1:-1] + Co_full[:-2]) / (dx ** 2)
    dCr_dt = D * (Cr_full[2:] - 2 * Cr_full[1:-1] + Cr_full[:-2]) / (dx ** 2)

    r_surf = k_ox * Cr0 - k_red * Co0

    return dCo_dt, dCr_dt, r_surf
# INFO: Selects from three numerical methods based on the order and stores the
#       solutions obtained for the current density. Also stores the concentration
#       profiles at each time step for the resulting animated graph, containing
#       both a voltammogram and the animated concentration vs distance from the
#       electrode at each time step.
def CyclicVoltammetry(order: int):
    F = cnt.value("Faraday constant")
    R = cnt.gas_constant
    T = e.error("temperature")
    nx = 300

    E_start = e.error("E_start")
    E_vertex = abs(e.error("E_vertex") - E_start)
    srate = e.error("srate")
    n_cycles = e.error("n_cycles")
    bulk_Co = e.error("Cob")
    bulk_Cr = e.error("Crb")
    D_coef = e.error("diff")
    k0 = e.error("k0")
    beta_c = e.error("beta_c")
    beta_a = 1.0 - beta_c
    n_el, E_std = getVal()

    t_max = 4.0 * n_cycles * E_vertex / srate
    x_max = 3.0 * m.sqrt(2.0 * D_coef * t_max)
    dx = x_max / nx
    dt = dx ** 2 / (5.0 * D_coef)
    t, E = init_time_potential(E_start, E_vertex, srate, dt, int(n_cycles))
    x = np.arange(0, nx + 1, 1)

    Co = np.full(nx + 1, bulk_Co)
    Cr = np.full(nx + 1, bulk_Cr)
    j = np.empty_like(t)

    Co_anim = np.full((len(t), nx + 1), bulk_Co)
    Cr_anim = np.full((len(t), nx + 1), bulk_Cr)

    lam =  D_coef * dt / (dx ** 2)

    params = {
        "k0": k0,
        "beta_c": beta_c,
        "n_el": n_el,
        "F": F,
        "R": R,
        "T": T,
        "dx": dx,
        "D": D_coef,
        "E_std": E_std,
        "bulk_Co": bulk_Co,
        "bulk_Cr": bulk_Cr,
        "x_max": x_max
    }

    dt2 = dt / 2
    dt6 = dt / 6

```

```
    if (order == 0):
        for i in range(len(t) - 1):

            Co0_new, Cr0_new, Co_int, Cr_int, r_surf = newton_thomas(Co, Cr, E[i], lam, params)


            newCo = np.concatenate(([Co0_new], Co_int, [bulk_Co]))
            newCr = np.concatenate(([Cr0_new], Cr_int, [bulk_Cr]))

            Co = newCo
            Cr = newCr

            j[i] = n_el * F * r_surf
            Co_anim[i + 1] = newCo
            Cr_anim[i + 1] = newCr

    elif order == 2:

        Co_int = Co[1:-1]
        Cr_int = Cr[1:-1]

        for i in range(len(t) - 1):

            k1_Co, k1_Cr, r_surf = get_derivatives(t[i], Co_int, Cr_int, E[i], params)

            Co_pred = Co_int + dt * k1_Co
            Cr_pred = Cr_int + dt * k1_Cr

            k2_Co, k2_Cr, _ = get_derivatives(t[i + 1], Co_pred, Cr_pred, E[i + 1], params)

            Co_int = Co_int + dt2 * (k1_Co + k2_Co)
            Cr_int = Cr_int + dt2 * (k1_Cr + k2_Cr)

            Co0_new, Cr0_new, _, _ = solve_surface_analytic(Co_int[0], Cr_int[0], E[i + 1], params)

            Co_anim[i + 1] = np.concatenate(([Co0_new], Co_int, [bulk_Co]))
            Cr_anim[i + 1] = np.concatenate(([Cr0_new], Cr_int, [bulk_Cr]))

            j[i] = n_el * F * r_surf

    elif order == 4:

        Co_int = Co[1:-1]
        Cr_int = Cr[1:-1]

        for i in range(len(t) - 1):

            k1_Co, k1_Cr, r_surf = get_derivatives(t[i], Co_int, Cr_int, E[i], params)
            k2_Co, k2_Cr, _ = get_derivatives(t[i] + dt2, Co_int + dt2 * k1_Co, Cr_int + dt2 * k1_Cr, 0.5 * (E[i] + E[i + 1]), params)
            k3_Co, k3_Cr, _ = get_derivatives(t[i] + dt2, Co_int + dt2 * k2_Co, Cr_int + dt2 * k2_Cr, 0.5 * (E[i] + E[i + 1]), params)
            k4_Co, k4_Cr, _ = get_derivatives(t[i + 1], Co_int + dt * k3_Co, Cr_int + dt * k3_Cr, E[i + 1], params)

            Co_int = Co_int + dt6 * (k1_Co + 2 * k2_Co + 2 * k3_Co + k4_Co)
            Cr_int = Cr_int + dt6 * (k1_Cr + 2 * k2_Cr + 2 * k3_Cr + k4_Cr)

            Co0_new, Cr0_new, _, _ = solve_surface_analytic(Co_int[0], Cr_int[0], E[i + 1], params)

            Co_anim[i + 1] = np.concatenate(([Co0_new], Co_int, [bulk_Co]))
            Cr_anim[i + 1] = np.concatenate(([Cr0_new], Cr_int, [bulk_Cr]))

            j[i] = n_el * F * r_surf

        j[-1] = j[-2]

    animate_cv(t, x, Co_anim, Cr_anim, E, j)
```