Técnicas matemáticas y programación en Python para la resolución de problemas químicos

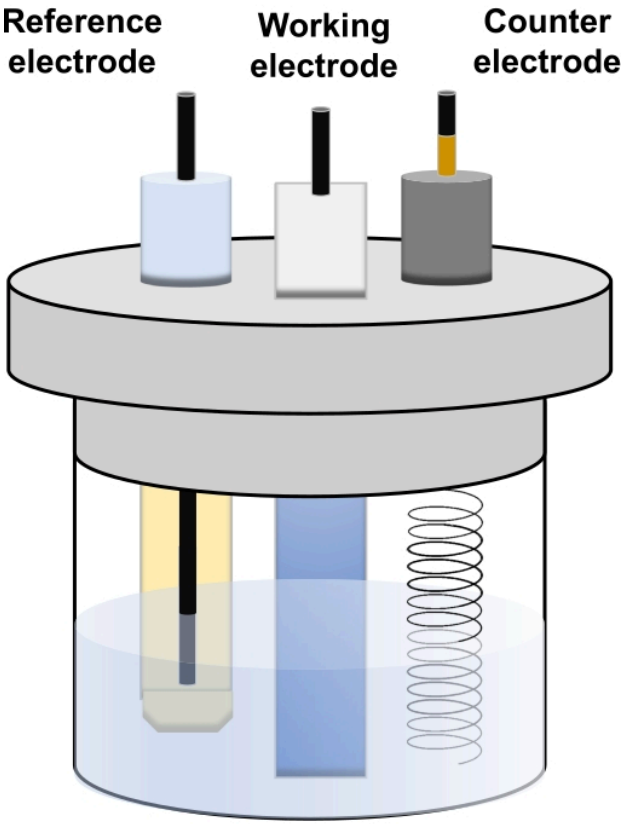
Cristian Angelo Manica Georgiev

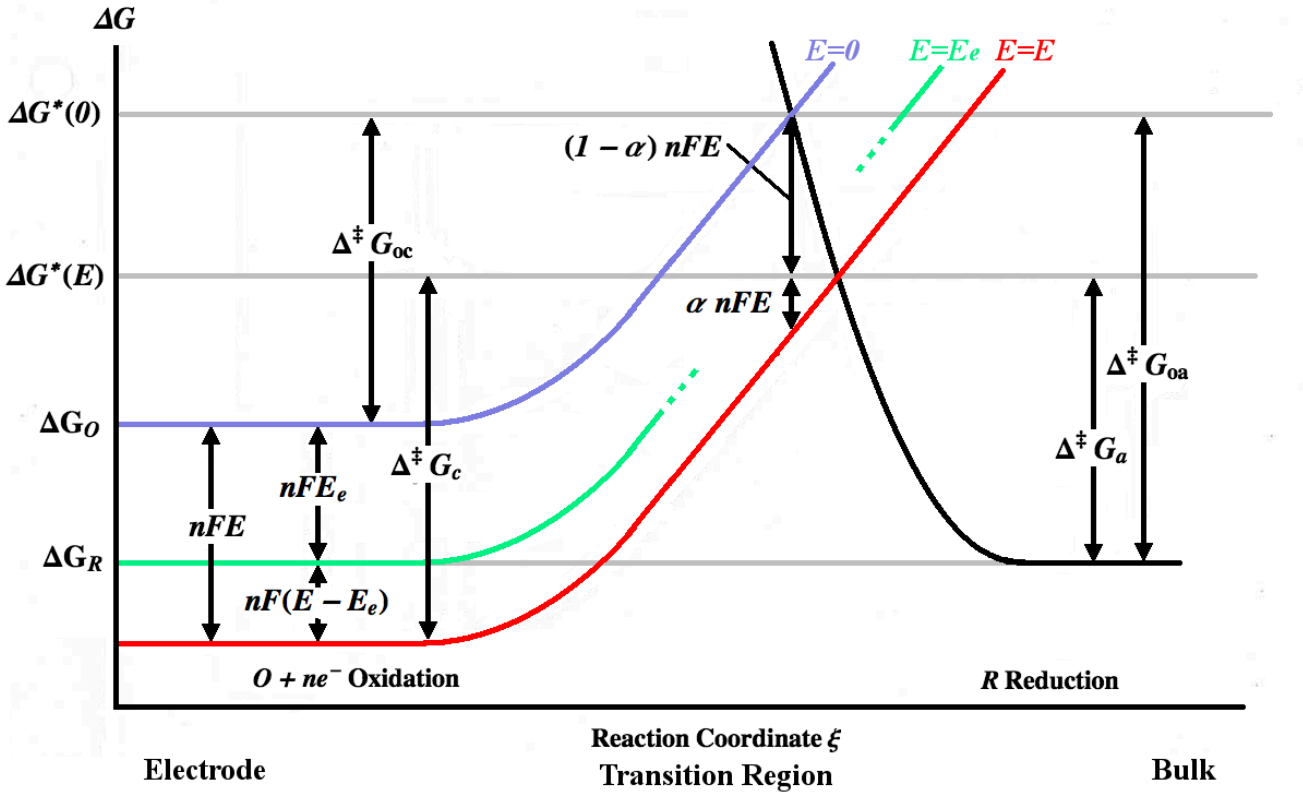
Universitat de Barcelona

26-01-2026

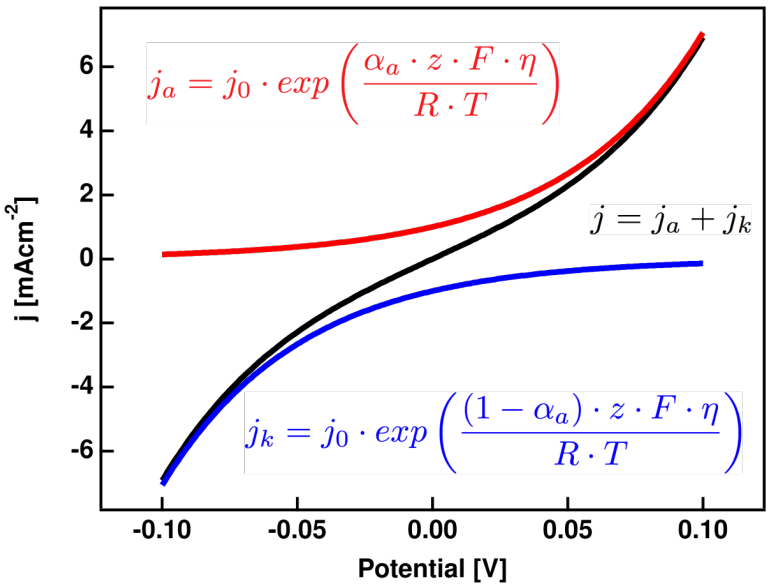
Índice

Electroquímica

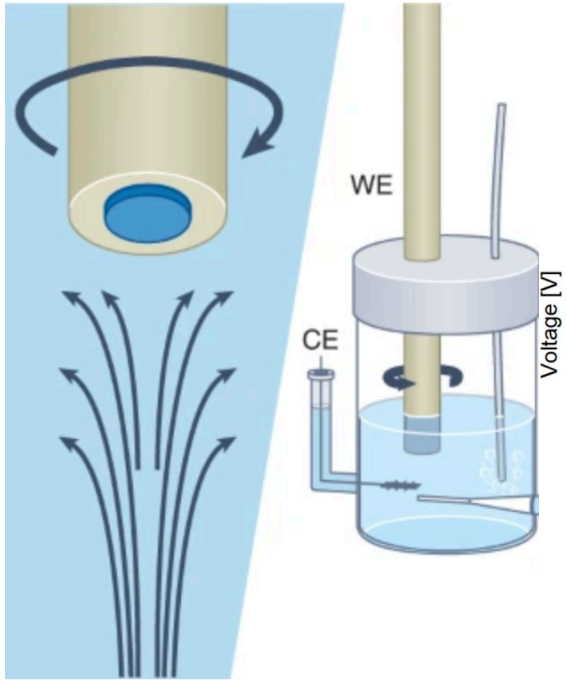




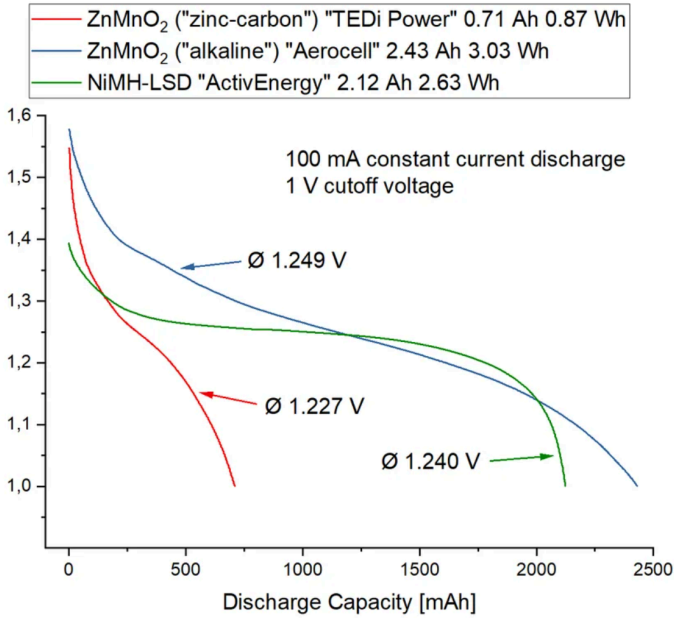
Electroquímica



$$j = j_0\left(e^{\frac{(1-\beta)nF\eta}{RT}} - e^{-\frac{\beta nF\eta}{RT}}\right)$$

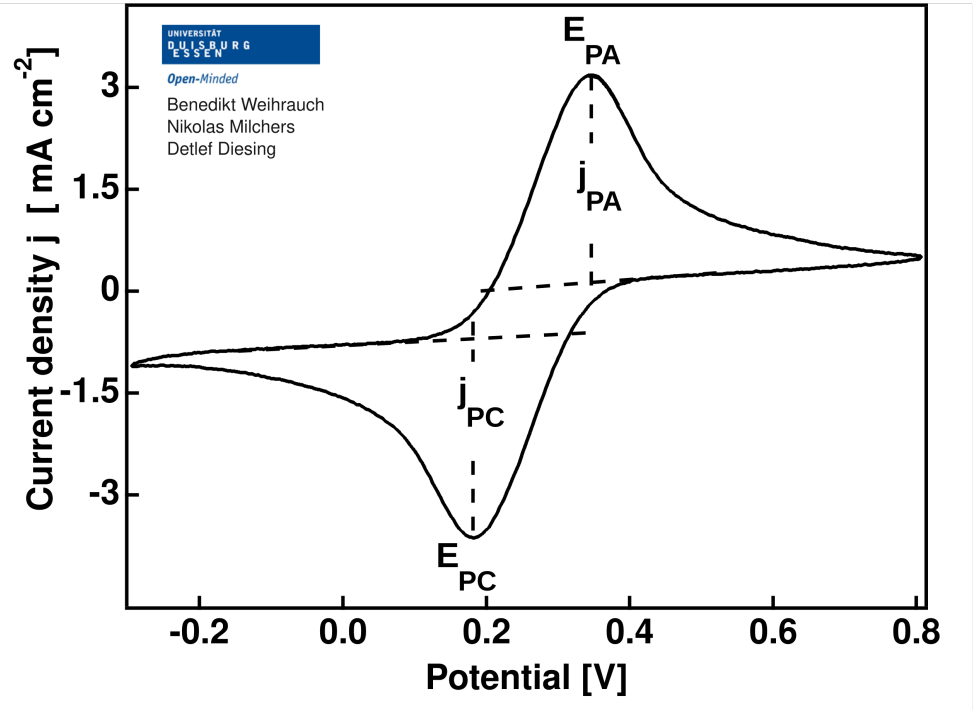


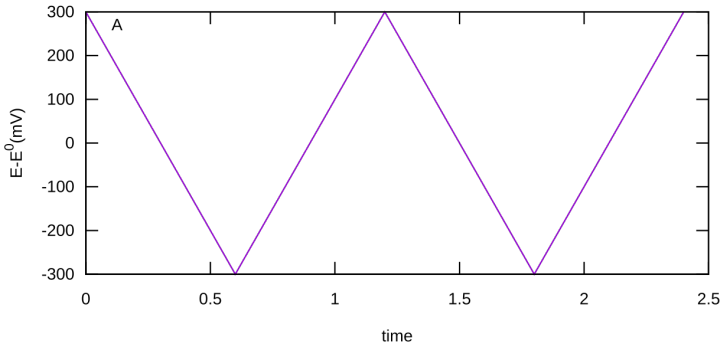
$$i_L = 0.62nFAD^{\frac{2}{3}}\omega^{\frac{1}{2}}\nu^{-\frac{1}{6}}C$$

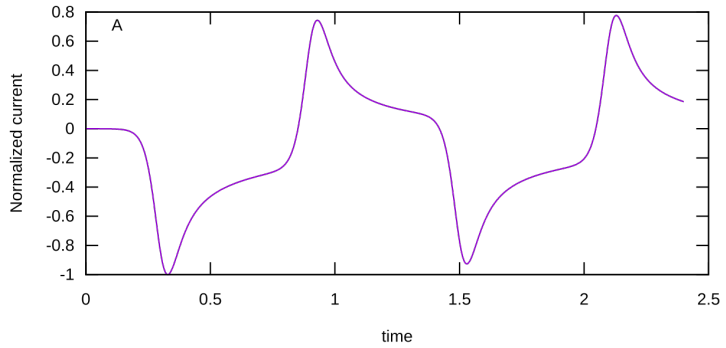


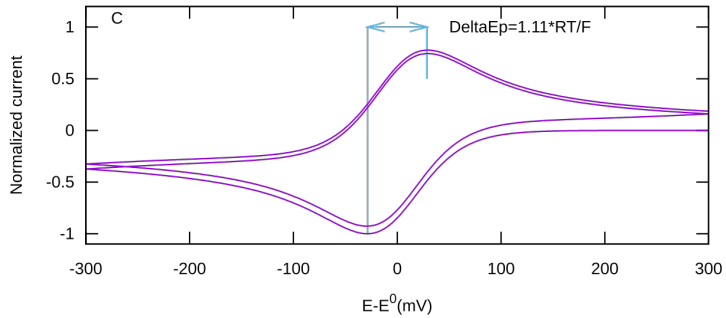
$$V = \mathrm{OCV(SOC)} - IR_{\mathrm{int}}$$

Electroquímica

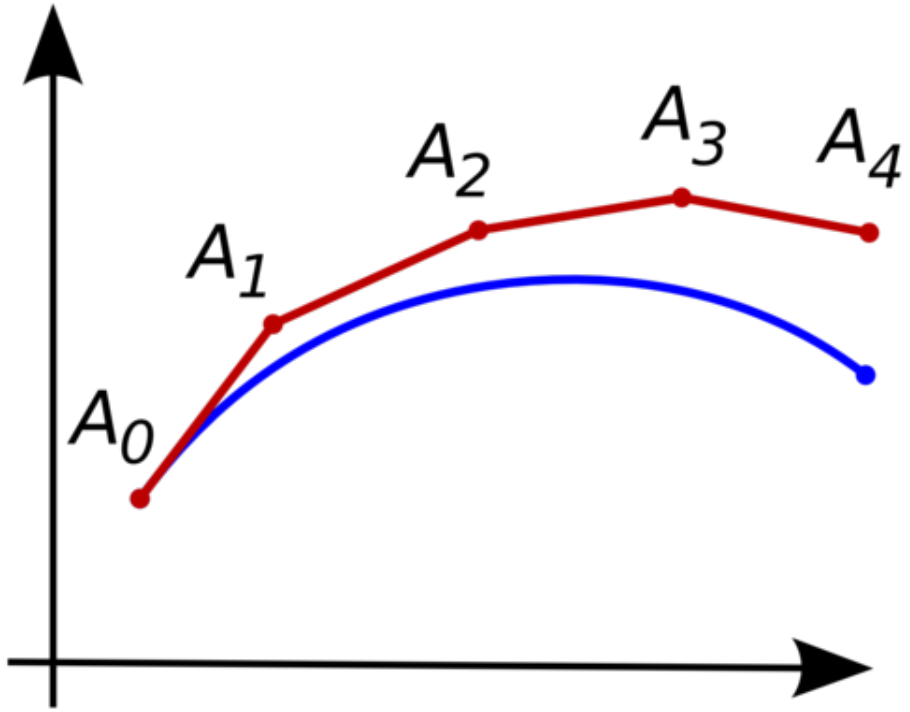








Métodos numéricos



Tipos:

- Integración numérica
- Derivación numérica
- Álgebra lineal numérica
- Resolucionadores de EDOs
- Resolucionadores de EDPs
- Interpolación y ajuste de curvas

Métodos numéricos

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Newton-Raphson

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

Algoritmo de Thomas

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

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

Runge-Kutta 4

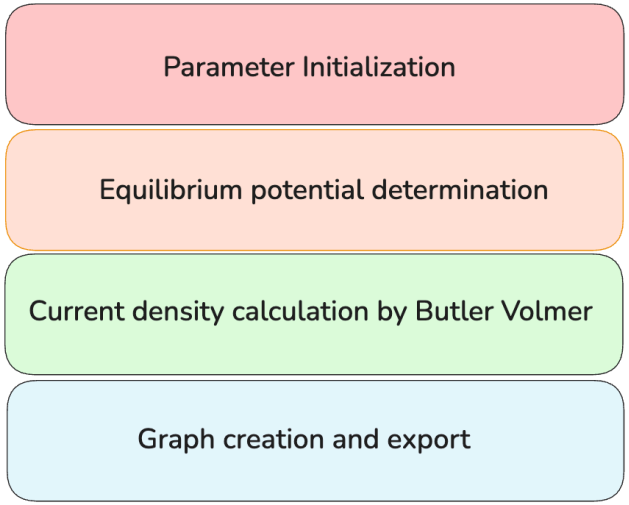
$$f''(x) \approx \frac{f(x + \Delta x) - 2f(x) + f(x - \Delta x)}{\Delta x^2}$$

Fórmula de tres puntos

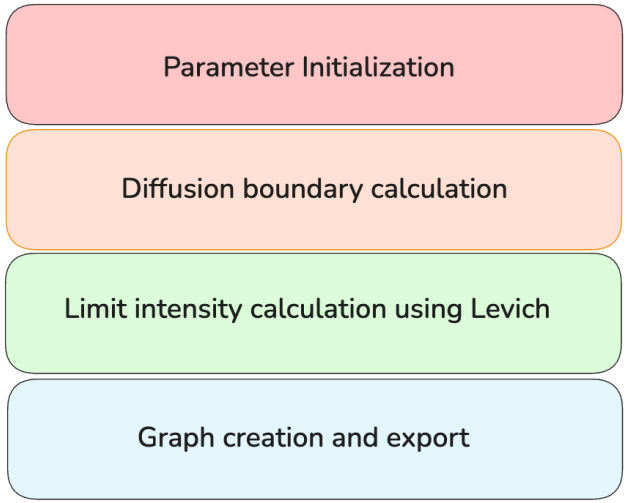
Objetivos

- **Enfoque Pedagógico y Didáctico**
- **Desarrollo técnico de sistemas electroquímicos**
- **Obtención de output visual sobre los sistemas**

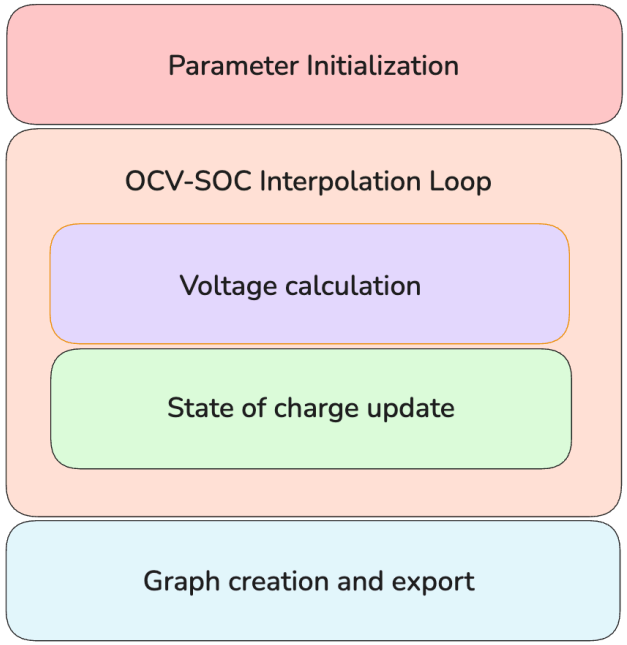
Implementación



ButlerVolmer.py

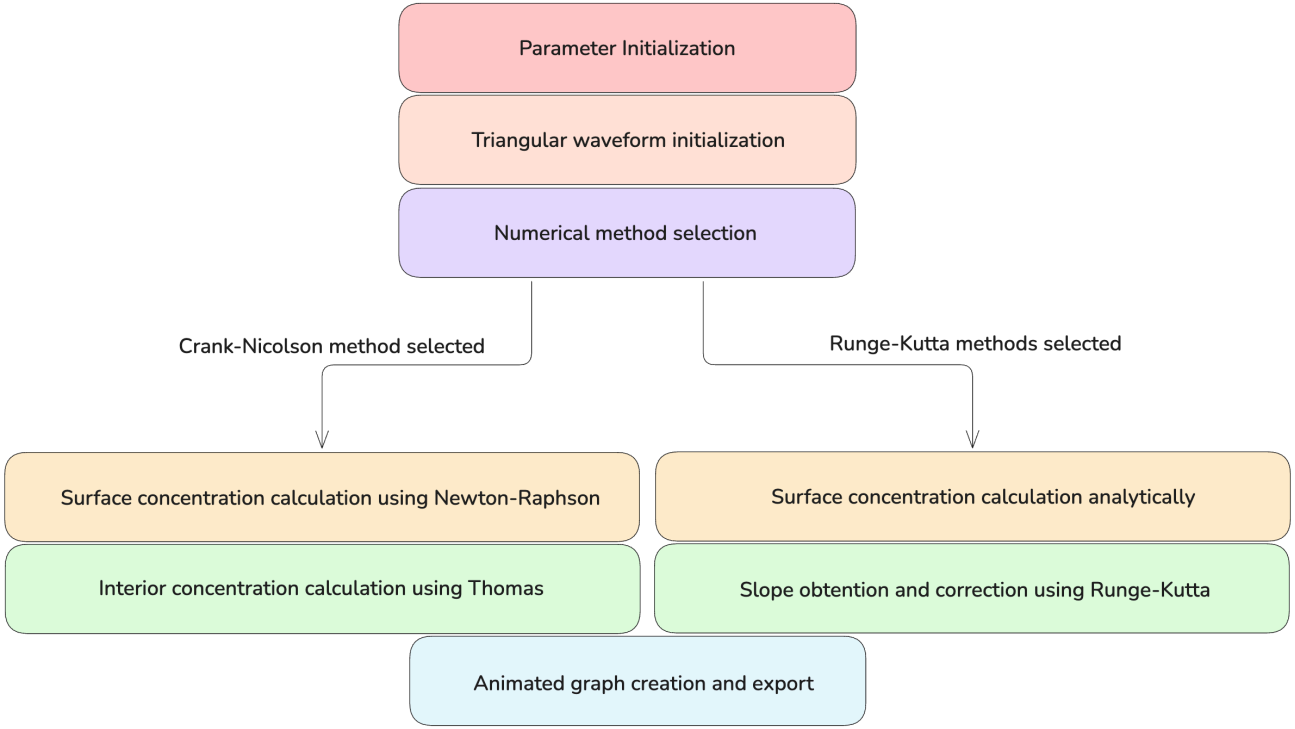


RotatingDiskElectrode.py



BatteryDischarge.py

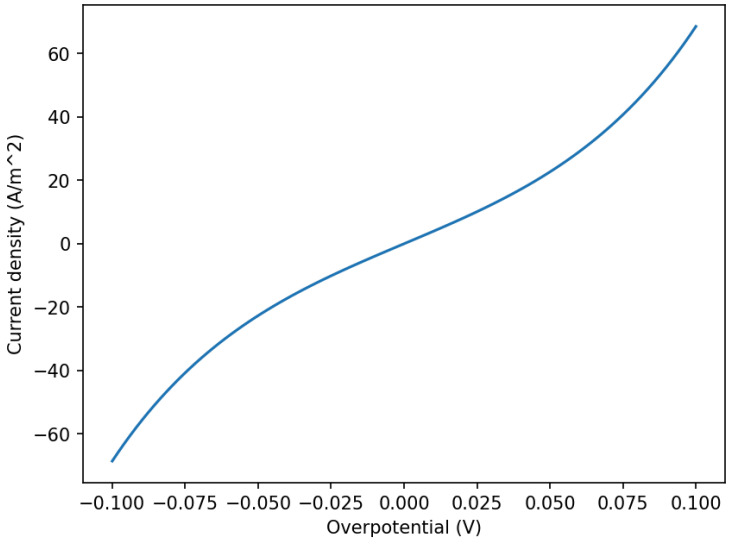
Implementación



CyclicVoltammetry.py

Resultados

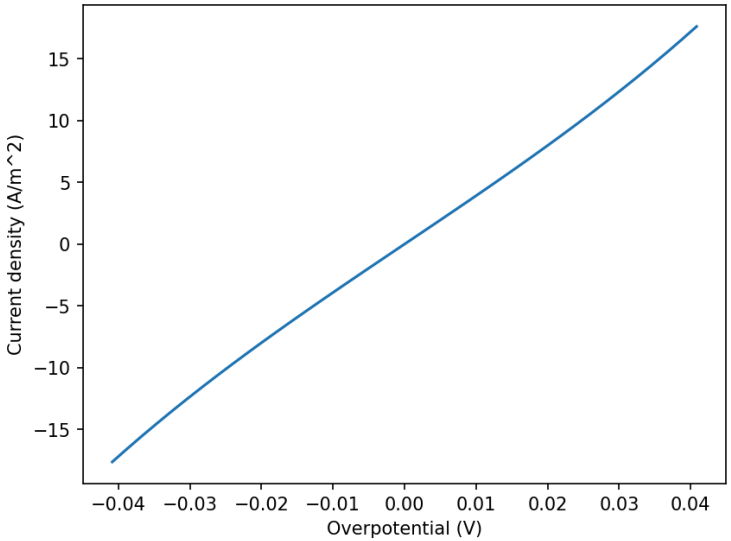
ButlerVolmer.py



$\mathrm{Fc}^+ + e^- \rightarrow \mathrm{Fc}$ with

$[\mathrm{Fc}^+] = 10^{-4}\ \mathrm{mol}/m^3$,

$[\mathrm{Fc}] = 10^{-4}\ \mathrm{mol}/m^3$



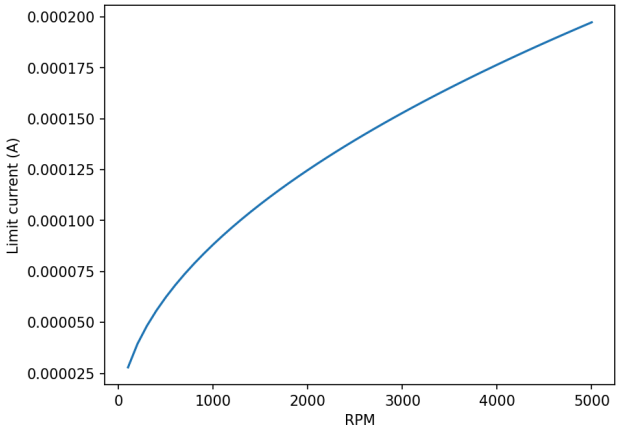
$\mathrm{Fc}^+ + e^- \rightarrow \mathrm{Fc}$ with

$[\mathrm{Fc}^+] = 10^{-3}\ \mathrm{mol}/m^3$,

$[\mathrm{Fc}] = 10^{-4}\ \mathrm{mol}/m^3$

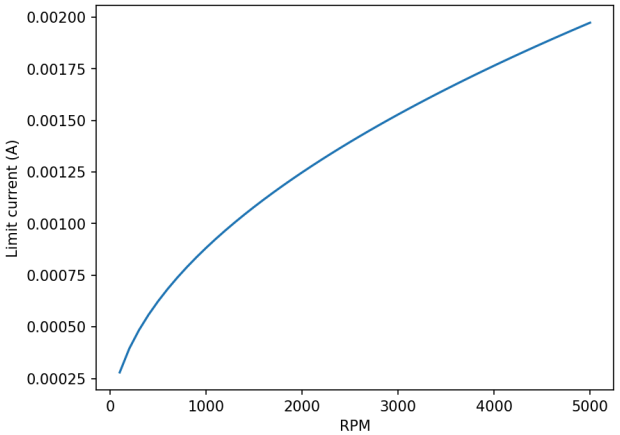
$$j = j_0 \left(\exp\left(\frac{(1-\beta) n F \eta}{RT} \right) - \exp\left(-\frac{\beta n F \eta}{RT} \right) \right)$$

RotatingDiskElectrode.py



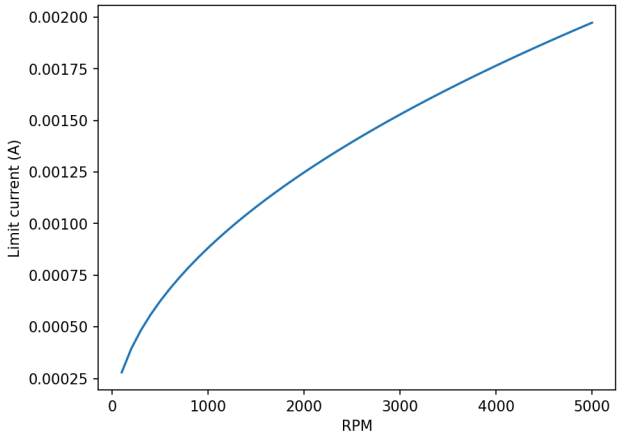
Standard limit intensity measurement with ferrocene

$i_L = 2 \cdot 10^{-4} A$



Limit current for 10 times the concentration of reactants

$i_L = 2 \cdot 10^{-3} A$

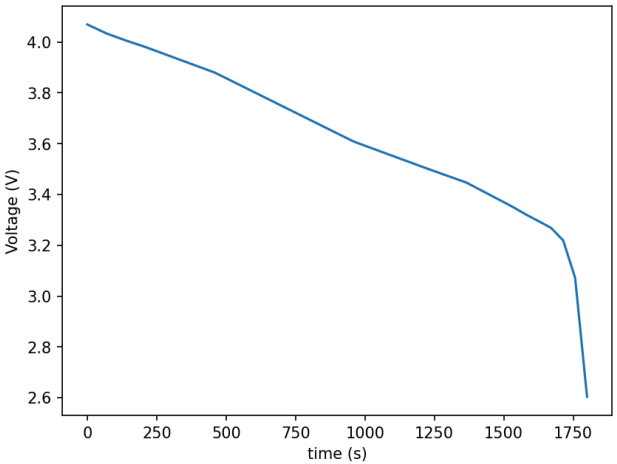


Limit current for 10 times the area of the electrode

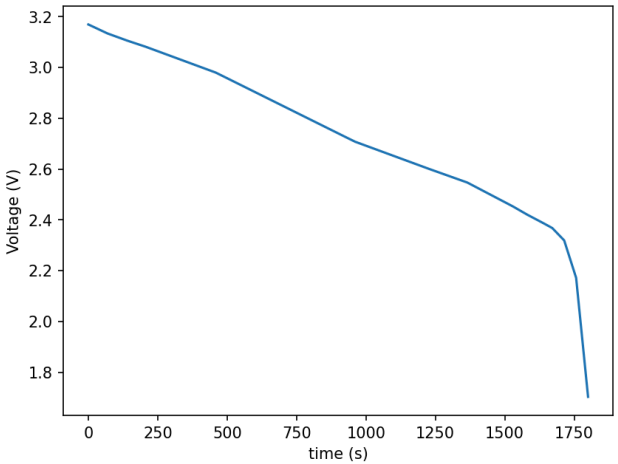
$i_L = 2 \cdot 10^{-3} A$

$$i_L = 0.620 n F A D^{\frac{2}{3}} \omega^{\frac{1}{2}} \nu^{-\frac{1}{6}} C$$

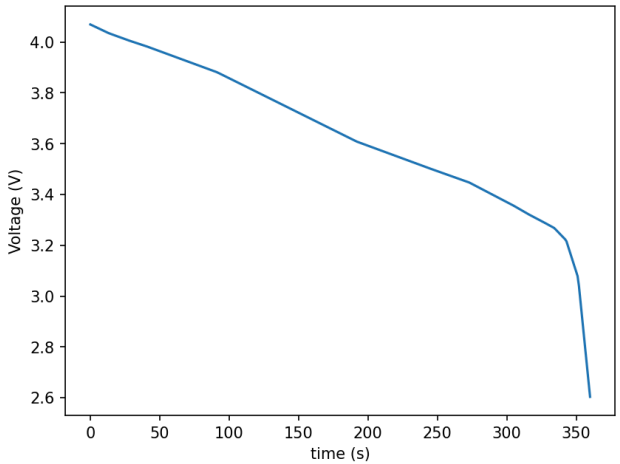
BatteryDischarge.py



Battery Discharge



Battery Discharge for 10 IR_{int}

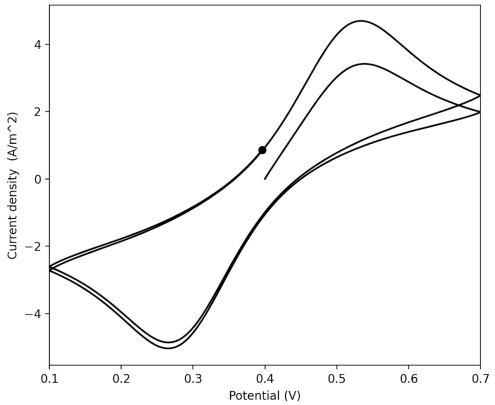


Battery Discharge for $1/5$ Q_{nom}

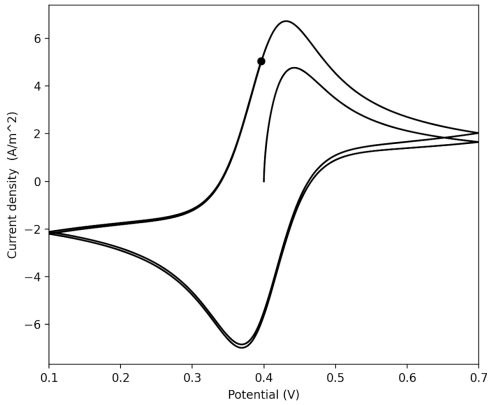
$$V = \text{OCV}(\text{SOC}) - IR_{\text{int}}$$

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

CyclicVoltammetry.py



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

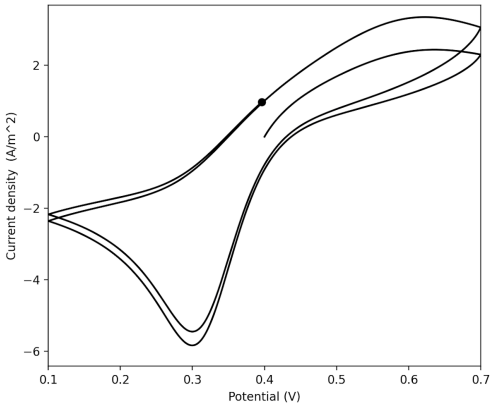


Voltammogram, $1000k_0$

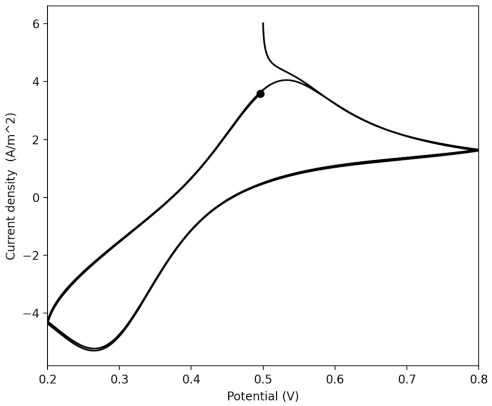
$$J = -D\frac{\partial C}{\partial x}$$

$$\frac{\partial C}{\partial t} = D\frac{\partial^2 C}{\partial x^2}$$

CyclicVoltammetry.py



Voltammogram, $\beta_c = 0.75$

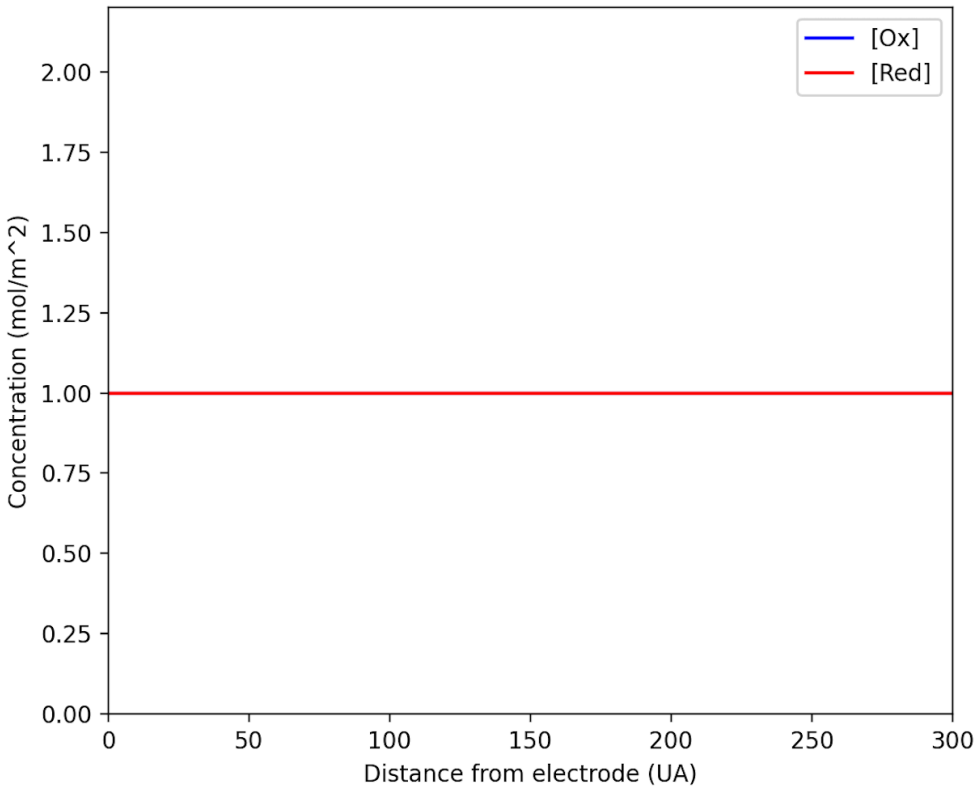


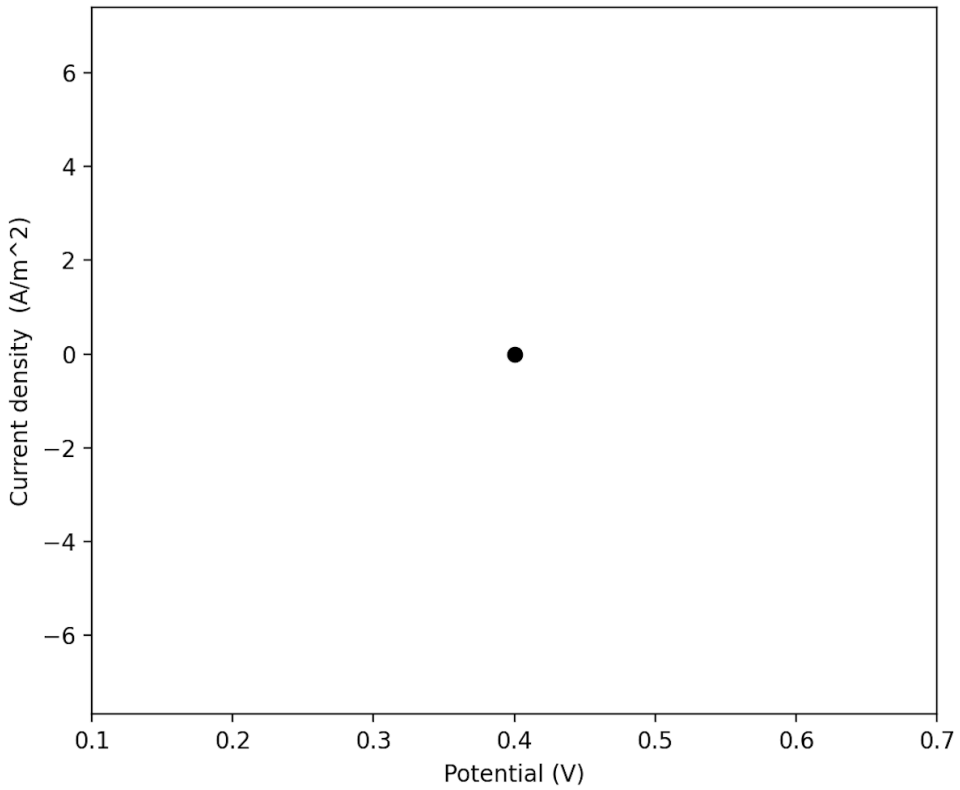
Voltammogram, E_{start} not at the equilibrium potential

$$J = -D\frac{\partial C}{\partial x}$$

$$\frac{\partial C}{\partial t} = D\frac{\partial^2 C}{\partial x^2}$$

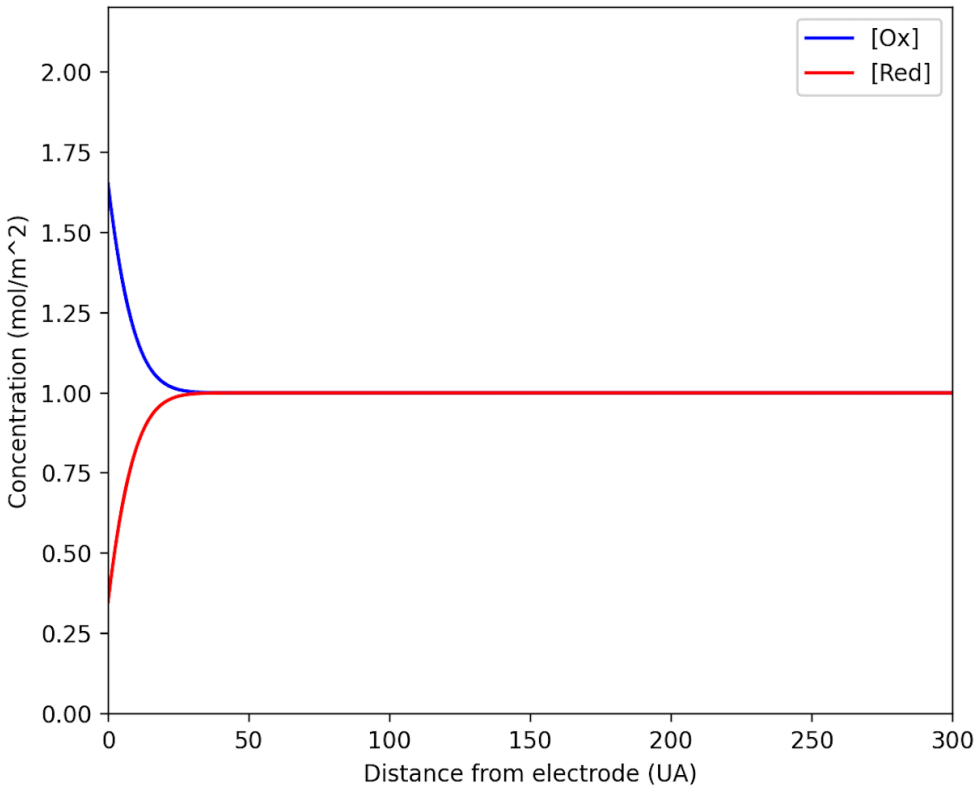
CyclicVoltammetry.py

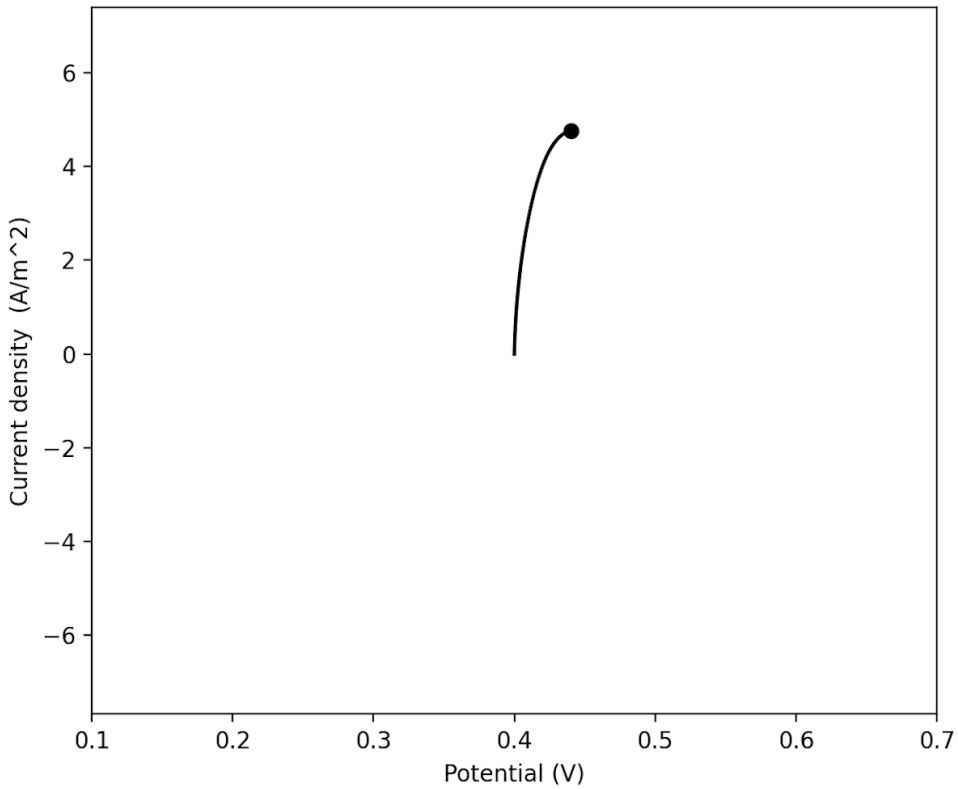




$$t = t_0$$

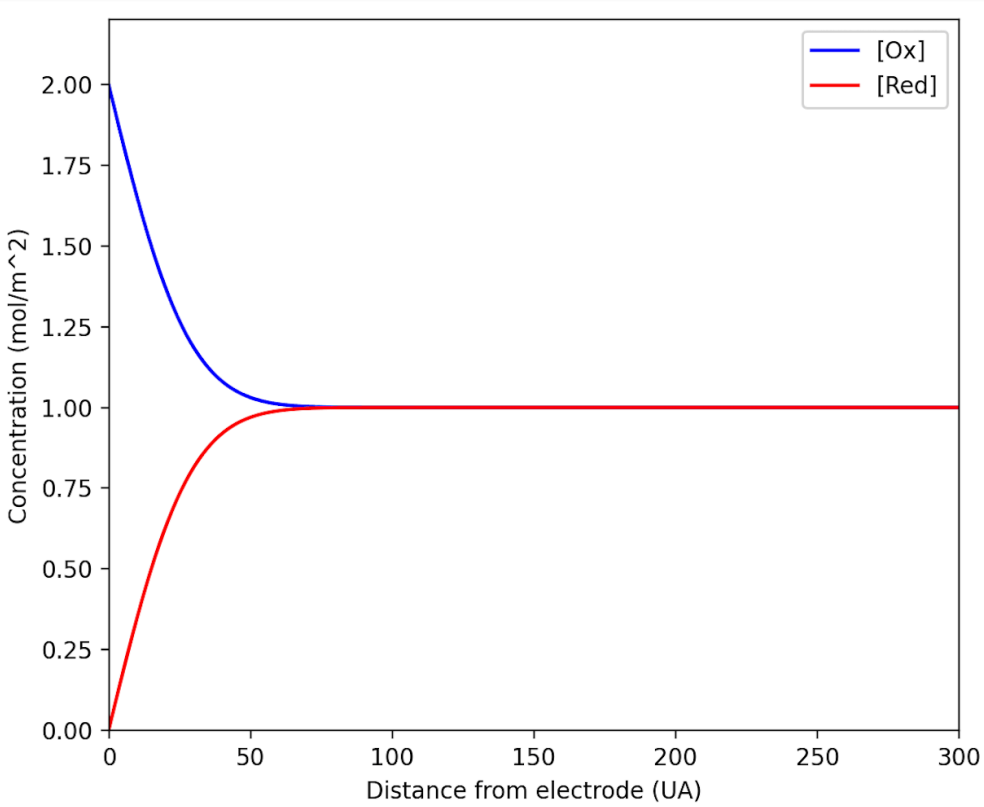
CyclicVoltammetry.py

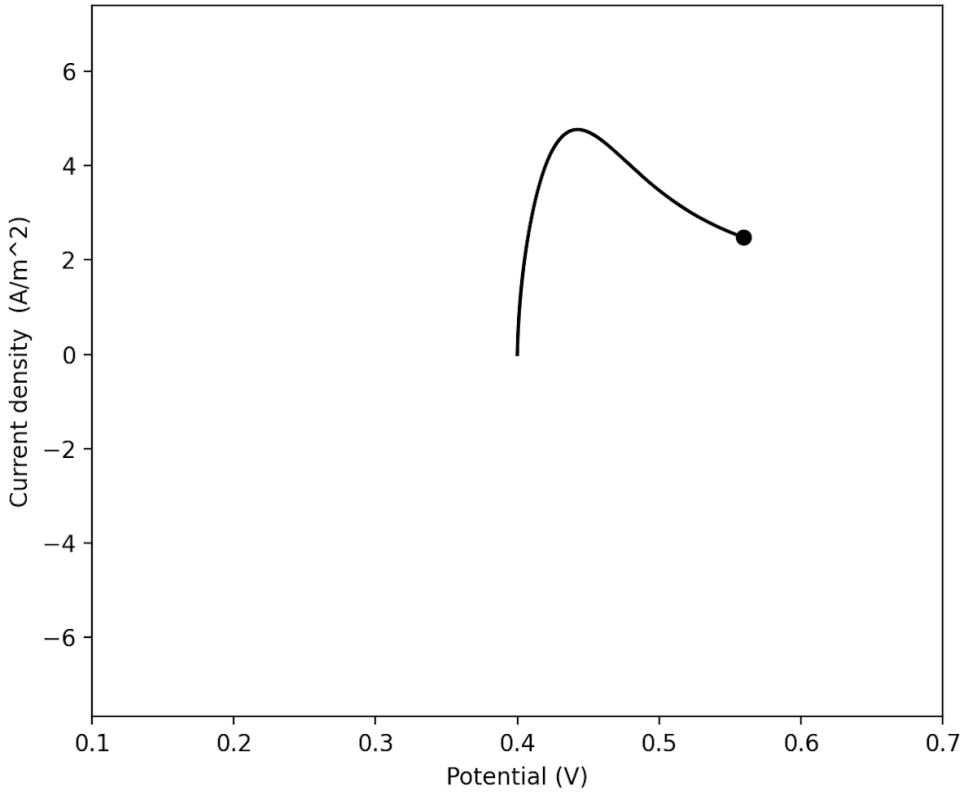




Alcanza el pico de oxidación

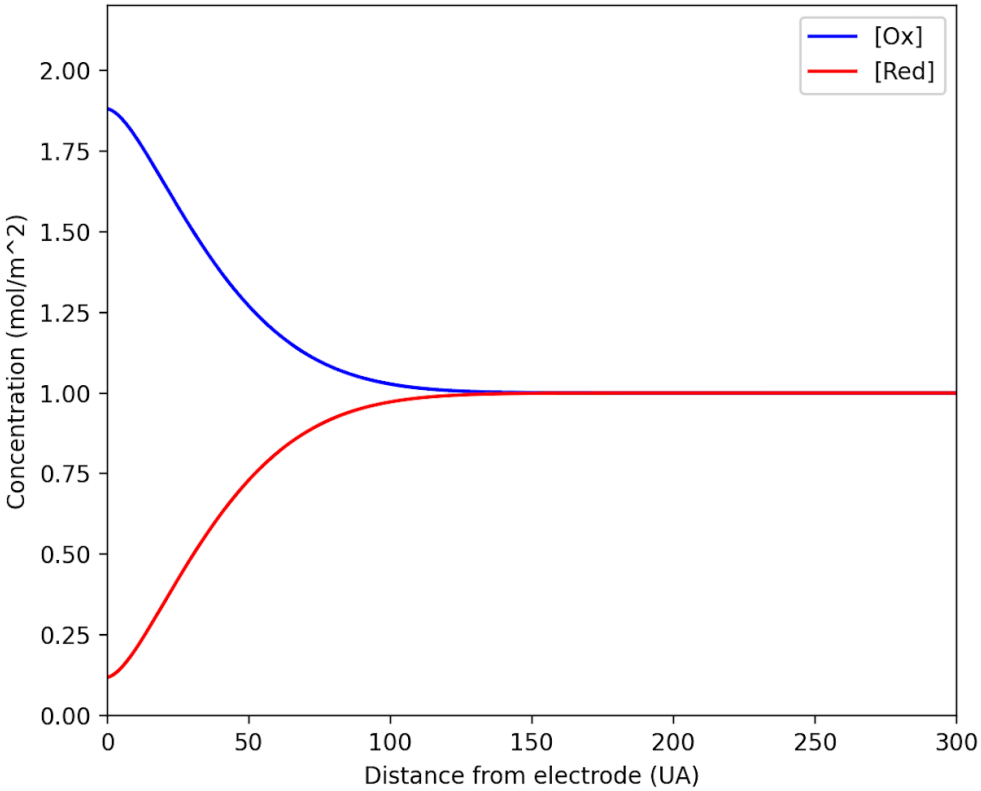
CyclicVoltammetry.py

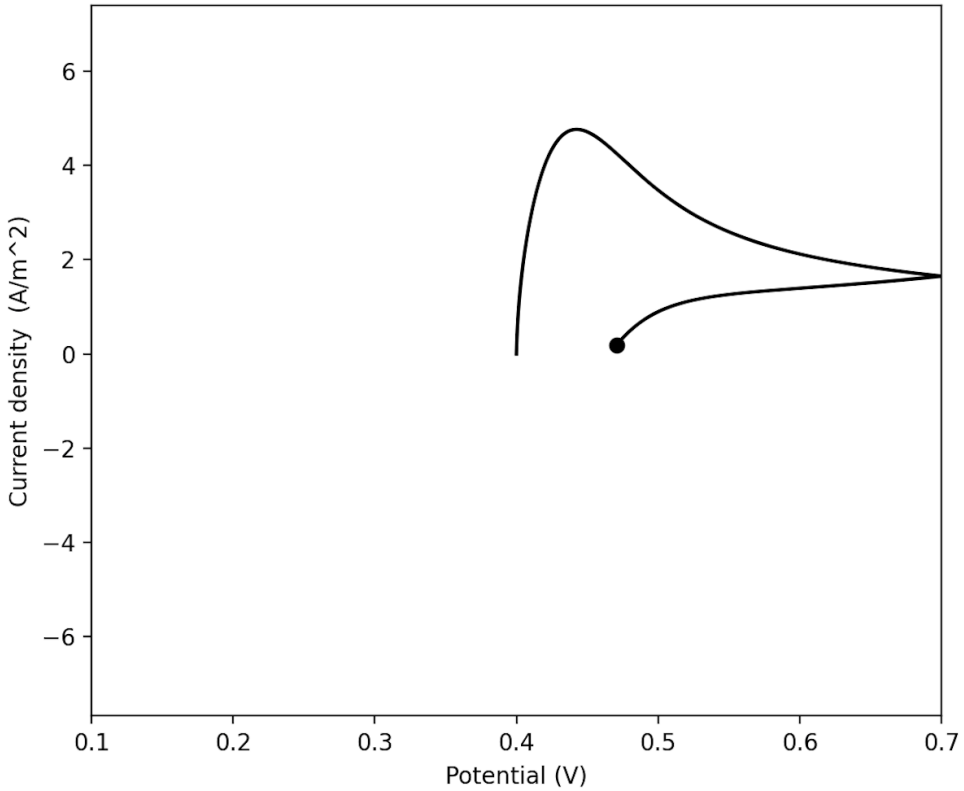




No [Red] restante, control por difusión

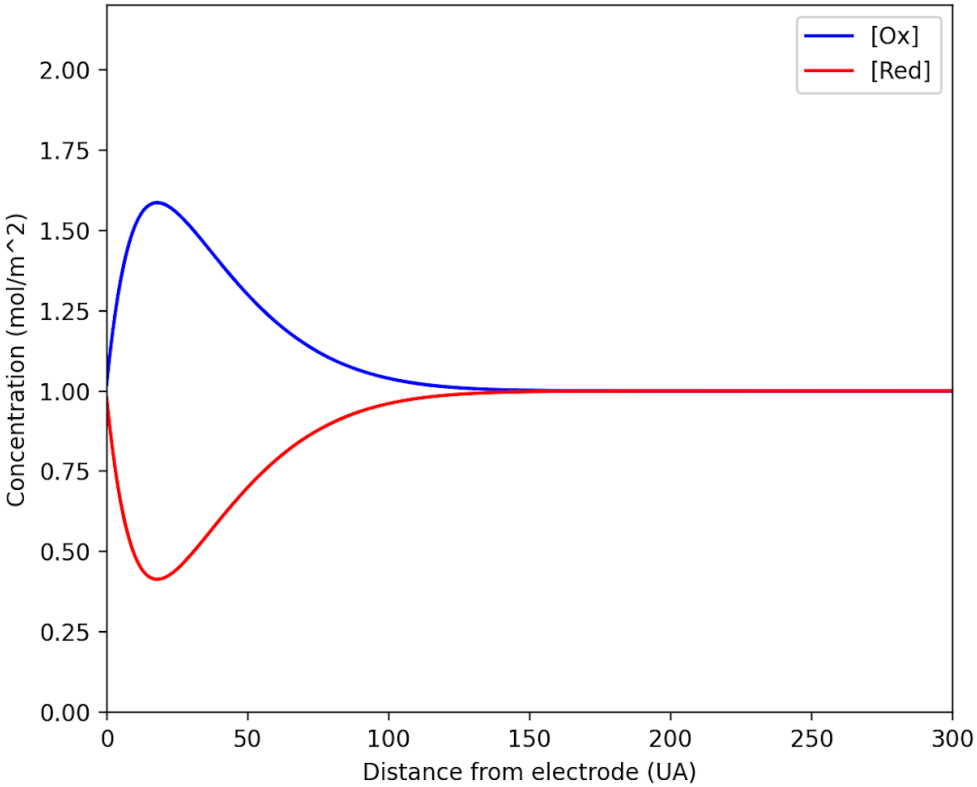
CyclicVoltammetry.py

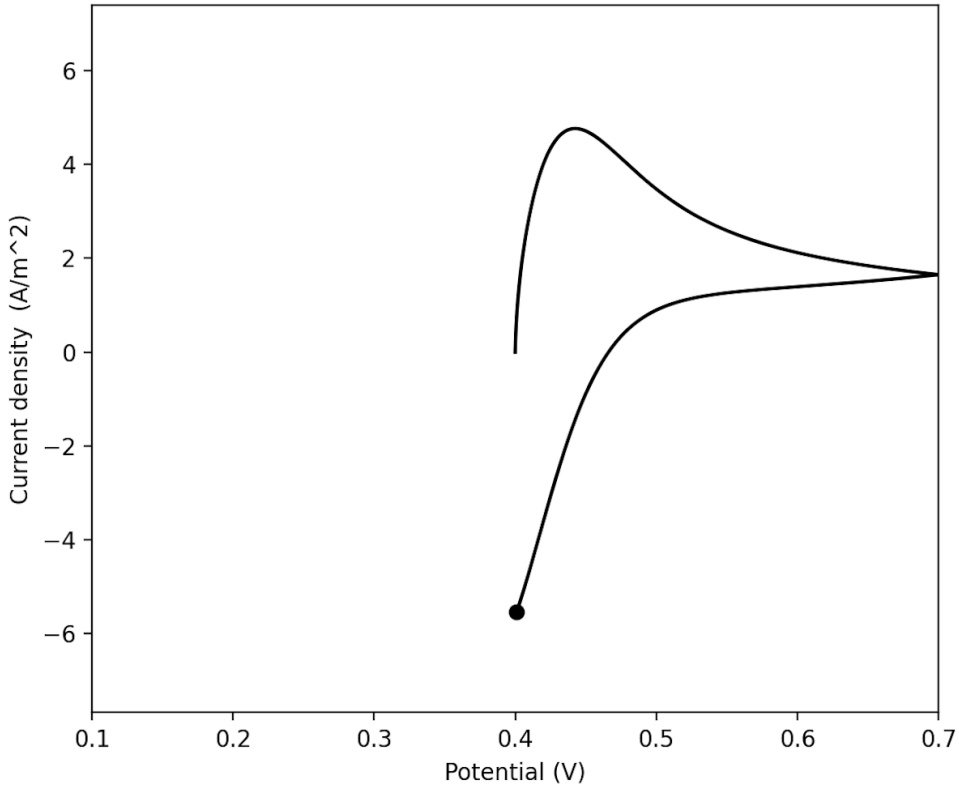




Comienza el barrido hacia el lado opuesto

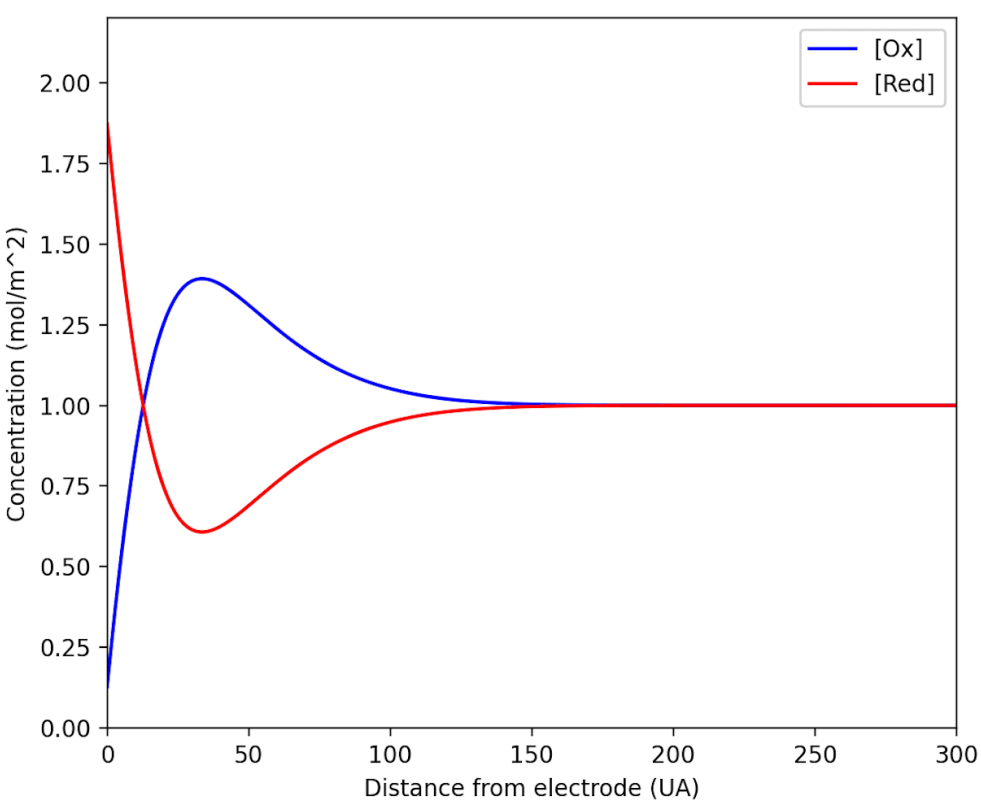
CyclicVoltammetry.py

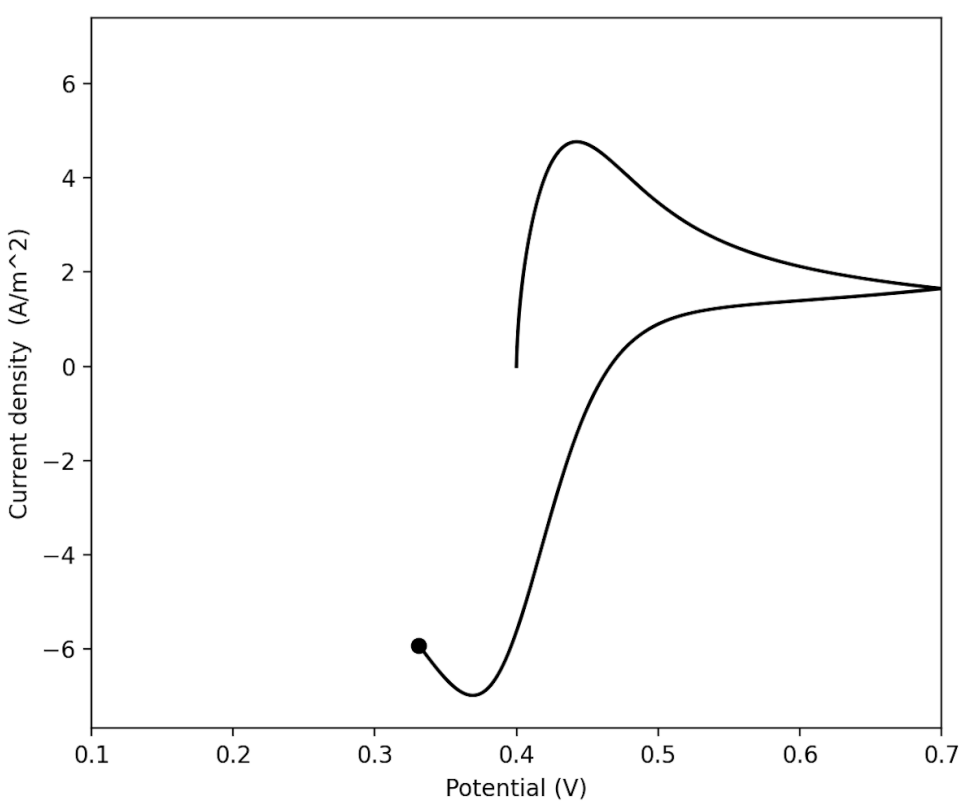




$$E = E_{\text{eq}}, [\text{Red}] = [\text{Ox}]$$

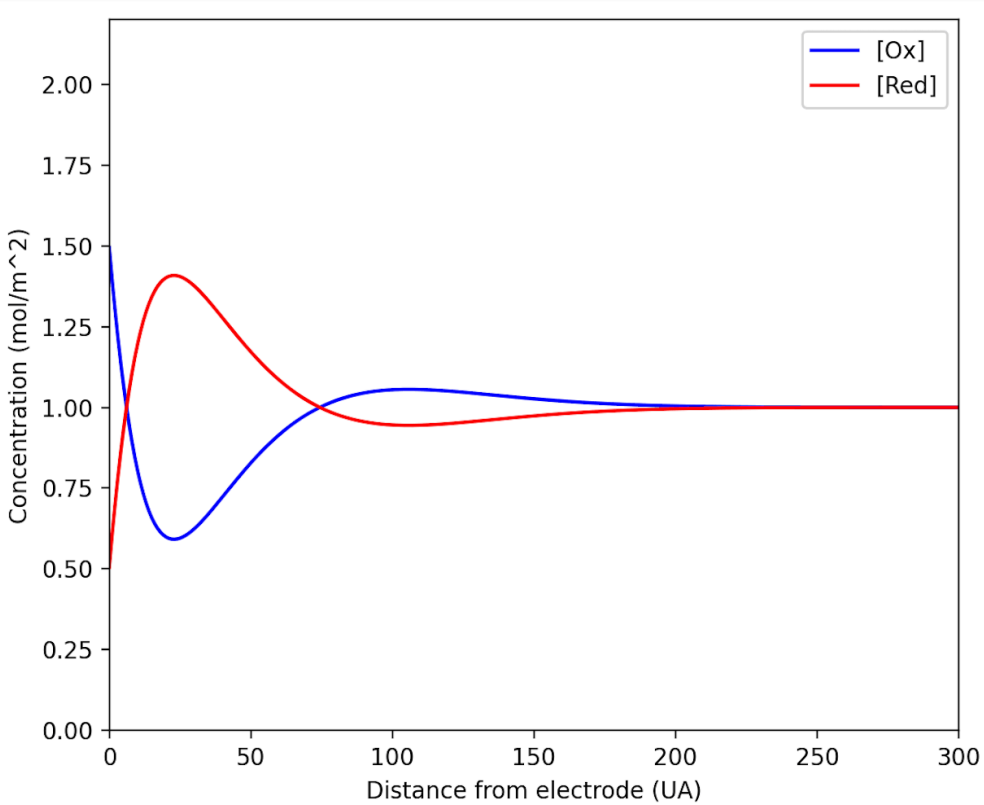
CyclicVoltammetry.py

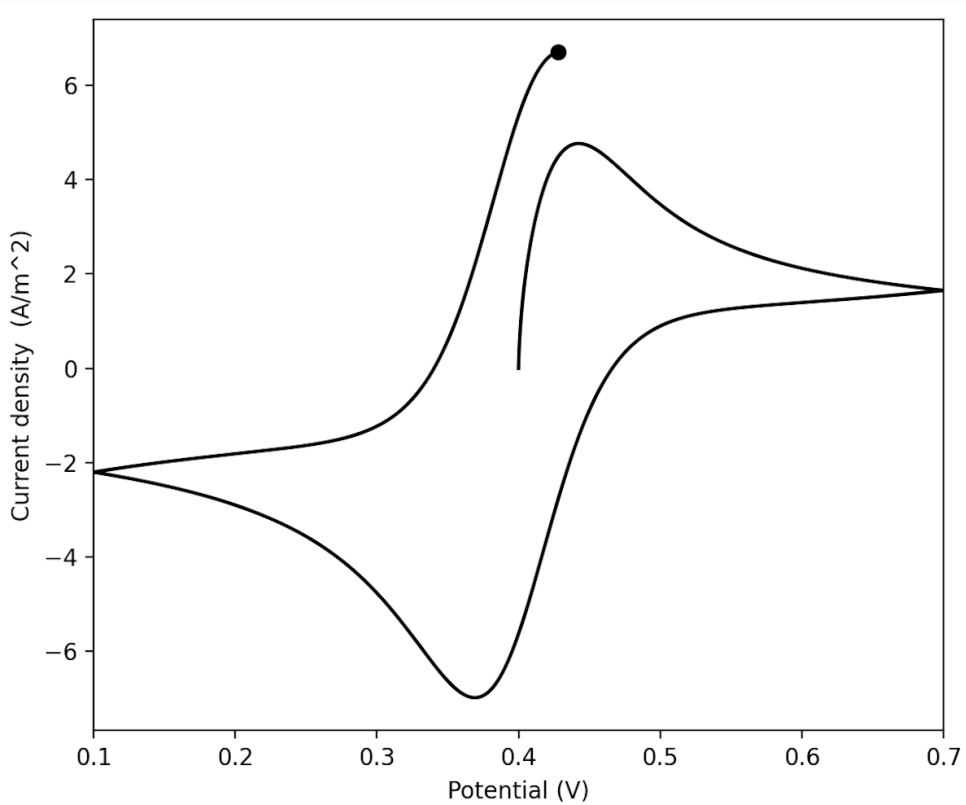




Se alcanza el pico de reducción, las concentraciones lejos del electrodo tardan en ecualizarse

CyclicVoltammetry.py





Se repite el ciclo

Conclusions

- **Development of functional software**
- **Precision through numerical methods**
- **Achievement of the pedagogical objective**

GRACIAS POR SU ATENCIÓN

```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

```

```python
# 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()
```

```python
# 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
```

```python
# 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
```

```python
# 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
```

```python
# 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

```

```python
    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)
```