JOINT INSTITUTE FOR NUCLEAR RESEARCH

Meshcheryakov Laboratory of Information Technologies

FINAL REPORT ON START Program

Performance Comparison of Python and Numba for Calculation of physical Characteristics of Josephson Junction using Numerical Method

Supervisor:

Adiba Rahmonova

Student:
Sheimaa, Ibrahim, Egypt,

Participation period:

03 August - 13 September

Summer session 2025

Dubna 2025

Contents

Abstract

This report details a comparative study on the performance of Python and Numba for the numerical solution of physical systems, performed during a research internship at the Joint Institute for Nuclear Research (JINR).

The study uses the Josephson junction equations as a benchmark system due to the availability of well-established solutions. The project's methodology involves numerically integrating the Resistively and Capacitively Shunted Junction (RCSJ) model using each of the two computational approaches. The primary objective is to evaluate the performance differences—specifically in terms of computational speed and scalability—to determine the most efficient method for complex numerical problems.

The results demonstrate the significant performance gains of Numba over standard Python, highlighting the benefits of parallel computing for large-scale scientific simulations. This work provides valuable insight into the practical application and performance trade-offs of these tools for researchers in computational physics.

1 Introduction

1.1 Josephson Junction

The connection between two superconducting layers through a thin layer of a non-superconducting barrier forms a structure known as a Josephson junction (named after the English scientist Brian Josephson). When an electric current flows through the Josephson junction, stationary and non-stationary Josephson effects are observed, depending on the magnitude of the current.

The stationary Josephson effect. When the current is passed below the critical value (I < Ic) in a Josephson junction (JJ), there is no voltage, and a superconducting current flows through the junction. This current is proportional to the sine of the phase difference of the order parameters of the superconducting layers forming the junction.

$$I_s = I_c \sin\varphi \tag{1}$$

Non-stationary Josephson effect. When the current is increased above the critical value (I > Ic), an alternating voltage develops in the Josephson junction, which is proportional to the derivative of the phase difference.

$$V = \frac{\hbar}{2e}\frac{d\varphi}{dt} \tag{2}$$

The system of equations for describing the dynamics of the Josephson junction: The dynamics of the Josephson junction are described within the framework of the RCSJ model (Resistively Capacitively Shunted Junction). Within this model, the Josephson junction is modeled as a parallel connection of a capacitor, a resistor, and a superconductor (see Figure).

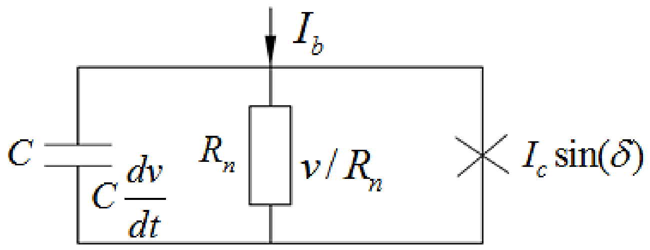


Figure 1. *RCSJ model (Resistively Capacitively Shunted Junction)*

A displacement current flows through the capacitor $I_{disp} = C\frac{dv}{dt}$, a quasiparticle current flows through the resistor $I_{qP} = \frac{V}{R}$, and the Josephson junction a (superconducting) current flows through the superconductor $I_s = I_c \sin\varphi$.

The total current through the system is equal to the sum of the above currents

$$I = C\frac{dV}{dt} + \frac{V}{R} + I_c \sin\varphi \tag{3}$$

By using (3) and (2) in normalized magnitudes, the closed system of differential equations is:

$$\begin{cases} \frac{dV}{dt} = I - \beta V - \sin\varphi, \\ \frac{d\varphi}{dt} = V \end{cases}$$

Under the external radiation on the current voltage characteristics of junction appears constant voltage step well known as "Shapiro step". To model this phenomenon, an additional alternating harmonic current is considered in the system of equations of the RCSJ model (I_R) created by external radiation with amplitude (A) and frequency (ω), i.e. $I_R = A\, sin(\omega t)$

Thus, the final form of the system of equations is:

$$\begin{cases} \frac{dV}{dt} = I + A\sin(u) - \beta V - \sin\varphi, \\ \frac{d\varphi}{dt} = V, \\ \frac{du}{dt} = \omega \end{cases}$$

The system parameters are:

- β – Dissipation parameter.
- A – External Radiation amplitude.
- V – Voltage.
- I – External DC current.
- ω – External Radiation frequency.
- u – External Radiation phase.
- φ – Phase difference across the junction

The main task in the framework of start practice was the calculation of current voltage characteristics of the Josephson junction and amplitude dependence of Shapiro step width

1.2 Numerical Methods for Differential Equations

The dynamics of the RCSJ model are described by a system of nonlinear ordinary differential equations. Since analytical solutions are often unavailable, numerical methods are essential for solving this equation. We have used the fourth-order Runge-Kutta (RK4) method to perform numerical integration. The RK4 method is a widely used iterative algorithm that provides a robust and accurate approximation of the solution by calculating the slope of the function at four different points within each time step. This approach allows for a precise simulation of the system's time evolution, providing a reliable foundation for studying the junction's voltage-current characteristics and other dynamic effects.

1.3 Computational Framework: Python and Numba

The performance of numerical simulations is highly dependent on the computational tools used. This study compares two distinct frameworks to solve the Josephson junction equations:

1. Python: As the foundational language, standard Python with libraries like NumPy provides a high-level, flexible environment for scientific computing. While excellent for prototyping and data analysis, it can be computationally slow for intensive numerical integration tasks.
2. Numba: A just-in-time (JIT) compiler that translates Python and NumPy code into fast, machine-native code. By compiling key functions, Numba significantly accelerates CPU-based numerical calculations, bridging the performance gap between Python and lower-level languages like C++.

2 Model, Methods and Algorithms

2.1 Jupyter Environment

The Jupyter Notebook serves as the primary environment for this project. It is an interactive, web-based computing environment that combines live code, narrative text, mathematical equations, and visualizations into a single document. This integrated structure provides a transparent and reproducible framework for scientific inquiry enabling a comprehensive presentation of the methodology, computational implementation, and the analysis of results, thereby supporting a rigorous and verifiable research process.

The Jupyter Notebook of the program contains functions which are:

1. *compute_cvc* Function: The Single-Sweep I-V Curve

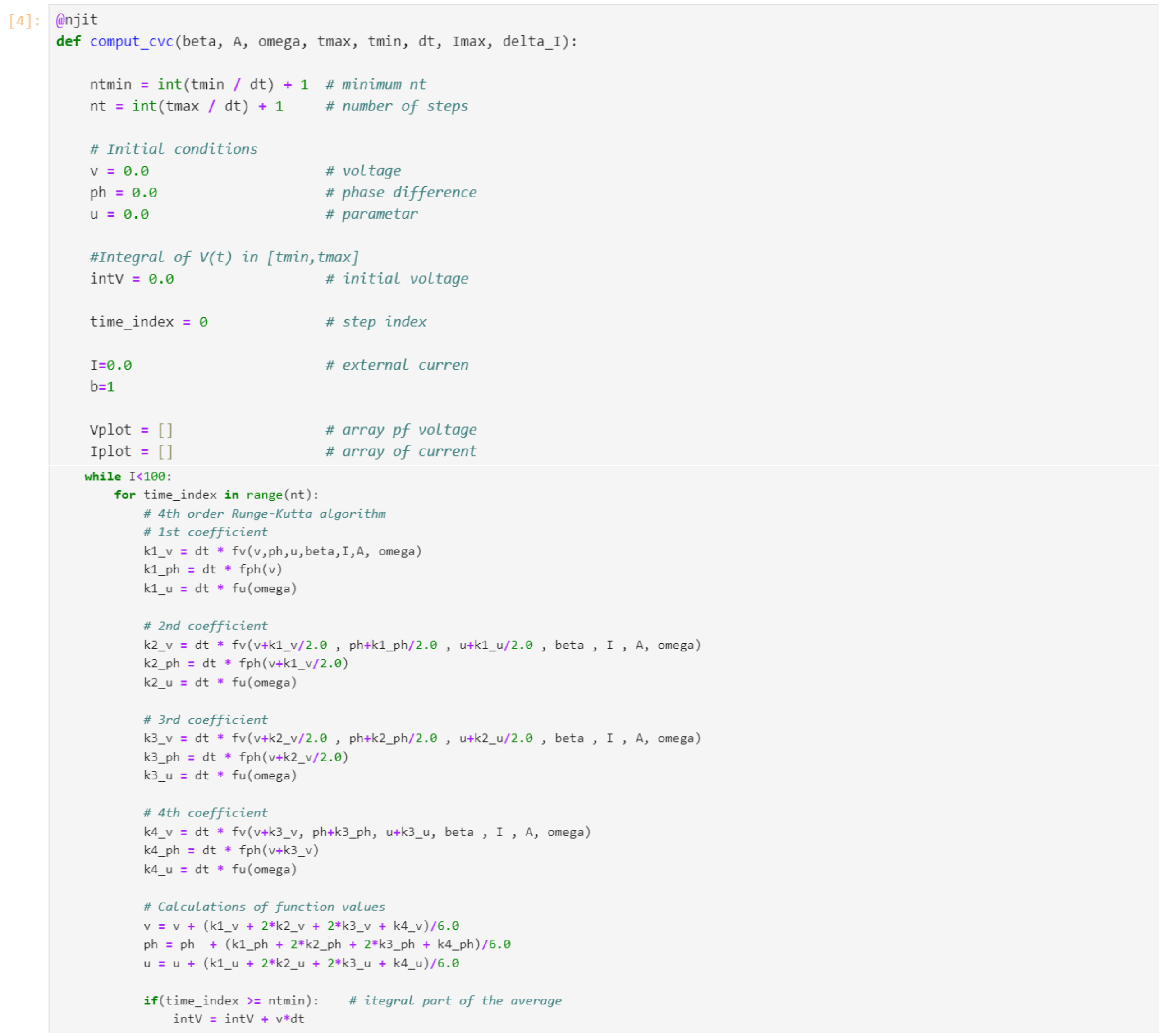
This function is the core of the simulation toolkit. Its sole purpose is to numerically solve the governing equations of the Josephson junction for a single, one-directional sweep of the DC current. The heart of this function is the 4th-Order Runge-Kutta (RK4) method, a powerful and widely trusted algorithm for solving ordinary differential equations.

Algorithm Steps and Detailed Calculations:

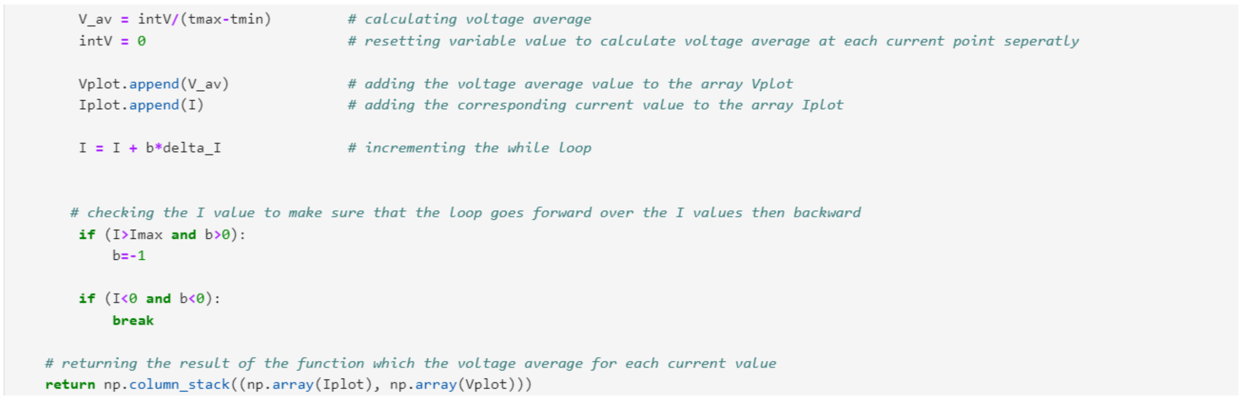
1. Initialization: The function takes the DC current array *I_array* and physical parameters like damping β**,** AC amplitude A, and AC frequency ω as input. It then initializes the key state variables, the voltage V, the phase ph (φ), and the radiation phase *u,* to zero. A blank array, *v_array*, is created to store the final averaged voltage values.
2. Define the System of Equations: The second-order RCSJ equation is broken down into a system of three first-order differential equations. The code defines these relationships in three helper functions:
   - *fv:* Calculates the rate of change of voltage, dv/dt.
   - *fph*: Calculates the rate of change of the phase, d(ph)/dt.
   - *fu:* Calculates the rate of change of the external radiation phase, du/dt.

   These functions calculate the instantaneous rate of change (or derivative) for each of the state variables based on the current state of the system.
3. The RK4 Time-Stepping Loop: The program enters a loop that iterates through each current value in the *I_array*. Inside this loop, a second, nested loop performs the time-stepping simulation using the RK4 method. For each small time-step *dt*, the algorithm performs the following four calculations to find the new state of the system:
   - *k_1*: Slope at the current time step for each variable (*V*, *ph*, *u*) using the current state
   - *k_2*: Slope at the midpoint. This is done using the initial state and adding half of the *k_1*.
   - *k_3*: Refined midpoint slope. It uses the state estimated with the *k_2* slopes providing a more accurate midpoint estimate.
   - *k_4*: Slope at the end. It calculates a final slope using the state at the end of the time step, based on the *k_3* slopes.

4. Updating the State Variables: The new values for (*V*, *ph*, *u*) are then calculated by taking a weighted average of the four slopes, with the midpoint slopes *k_2* and *k_3* being given more weight. The updated voltage *V* is stored at each time step.
5. Average Voltage Calculation: Once the time loop is complete for a given current, the algorithm discards the initial part of the voltage data (the transient phase) and calculates the average of the remaining values providing a stable, steady-state voltage measurement that corresponds to the simulated current. This process is repeated for every current in the *I_array*, and the final averaged currents and voltages are returned.



```
[4]: @njit
     def comput_cvc(beta, A, omega, tmax, tmin, dt, Imax, delta_I):

         ntmin = int(tmin / dt) + 1  # minimum nt
         nt = int(tmax / dt) + 1     # number of steps

         # Initial conditions
         v = 0.0                     # voltage
         ph = 0.0                    # phase difference
         u = 0.0                     # parametar

         #Integral of V(t) in [tmin,tmax]
         intV = 0.0                  # initial voltage

         time_index = 0              # step index

         I=0.0                       # external curren
         b=1

         Vplot = []                  # array pf voltage
         Iplot = []                  # array of current
         while I<100:
             for time_index in range(nt):
                 # 4th order Runge-Kutta algorithm
                 # 1st coefficient
                 k1_v = dt * fv(v,ph,u,beta,I,A, omega)
                 k1_ph = dt * fph(v)
                 k1_u = dt * fu(omega)

                 # 2nd coefficient
                 k2_v = dt * fv(v+k1_v/2.0 , ph+k1_ph/2.0 , u+k1_u/2.0 , beta , I , A, omega)
                 k2_ph = dt * fph(v+k1_v/2.0)
                 k2_u = dt * fu(omega)

                 # 3rd coefficient
                 k3_v = dt * fv(v+k2_v/2.0 , ph+k2_ph/2.0 , u+k2_u/2.0 , beta , I , A, omega)
                 k3_ph = dt * fph(v+k2_v/2.0)
                 k3_u = dt * fu(omega)

                 # 4th coefficient
                 k4_v = dt * fv(v+k3_v, ph+k3_ph, u+k3_u, beta , I , A, omega)
                 k4_ph = dt * fph(v+k3_v)
                 k4_u = dt * fu(omega)

                 # Calculations of function values
                 v = v + (k1_v + 2*k2_v + 2*k3_v + k4_v)/6.0
                 ph = ph  + (k1_ph + 2*k2_ph + 2*k3_ph + k4_ph)/6.0
                 u = u + (k1_u + 2*k2_u + 2*k3_u + k4_u)/6.0

                 if(time_index >= ntmin):    # itegral part of the average
                     intV = intV + v*dt
```



```
        V_av = intV/(tmax-tmin)         # calculating voltage average
        intV = 0                        # resetting variable value to calculate voltage average at each current point seperatly

        Vplot.append(V_av)              # adding the voltage average value to the array Vplot
        Iplot.append(I)                 # adding the corresponding current value to the array Iplot

        I = I + b*delta_I               # incrementing the while loop


       # checking the I value to make sure that the loop goes forward over the I values then backward
        if (I>Imax and b>0):
            b=-1

        if (I<0 and b<0):
            break

    # returning the result of the function which the voltage average for each current value
    return np.column_stack((np.array(Iplot), np.array(Vplot)))
```

2. compute_multiloop_cvc Function: Capturing Hysteresis

This function performs a bidirectional sweep of the DC current for accurately capturing the hysteretic loop that defines the junction's I-V characteristic and revealing the full picture of the junction's behaviour.

Algorithm Steps and Detailed Calculations:

1. Forward Sweep (High-to-Low Current): The algorithm first defines a descending array of current values using *np.linspace*, iterating through the DC current values from a high current down to a low current. For each current value in this descending sequence, it runs a full time-stepping RK4 simulation.
   Hysteresis Capture: As the current decreases, the junction maintains a high-voltage state until it reaches a specific current value where it suddenly switches back to the zero-voltage state. This sharp drop is a defining feature of the junction's dynamics and is only visible when sweeping the current downwards.
2. Reverse Sweep (Low-to-High Current): Once the current reaches its minimum value, the algorithm reverses direction. It then iterates through the current values from the low current back up to the high current, again running a full RK4 simulation for each value.
3. Concatenation and Return: The algorithm then combines the current and voltage data from both the forward and reverse sweeps to form a single, complete dataset. This combined data fully represents the hysteretic I-V curve, which is essential for any detailed analysis of the junction's dynamics.

```
[8]: @njit
def comput_multiloop_cvc(beta, A, omega, tmax, tmin, dt, delta_I):

    getstep=False                # Variable to check if you landed on the step

    ntmin = int(tmin / dt) + 1   # minimum nt
    nt = int(tmax / dt) + 1      # number of steps

    # Initial conditions
    v = 0.0                      # voltage
    ph = 0.0                     # phase difference
    u = 0.0                      # parametar

    #Integral of V(t) in [tmin,tmax]
    intV = 0.0                   # initial voltage

    time_index = 0               # step index

    I=0.0                        # external curren
    b=1

    Vplot = []                   # array pf voltage
    Iplot = []                   # array of current
    while I<100:
        for time_index in range(nt):
            # 4th order Runge-Kutta algorithm
            # 1st coefficient
            k1_v = dt * fv(v,ph,u,beta,I,A, omega)
            k1_ph = dt * fph(v)
            k1_u = dt * fu(omega)

            # 2nd coefficient
            k2_v = dt * fv(v+k1_v/2.0 , ph+k1_ph/2.0 , u+k1_u/2.0 , beta , I , A, omega)
            k2_ph = dt * fph(v+k1_v/2.0)
            k2_u = dt * fu(omega)

            # 3rd coefficient
            k3_v = dt * fv(v+k2_v/2.0 , ph+k2_ph/2.0 , u+k2_u/2.0 , beta , I , A, omega)
            k3_ph = dt * fph(v+k2_v/2.0)
            k3_u = dt * fu(omega)

            # 4th coefficient
            k4_v = dt * fv(v+k3_v, ph+k3_ph, u+k3_u, beta , I , A, omega)
            k4_ph = dt * fph(v+k3_v)
            k4_u = dt * fu(omega)

            # Calculations of function values
            v = v + (k1_v + 2*k2_v + 2*k3_v + k4_v)/6.0
            ph = ph  + (k1_ph + 2*k2_ph + 2*k3_ph + k4_ph)/6.0
            u = u + (k1_u + 2*k2_u + 2*k3_u + k4_u)/6.0

            if(time_index >= ntmin):
                intV = intV + v*dt
        V_av = intV/(tmax-tmin)          # calculating voltage average
        intV = 0                         # resetting variable value to calculate voltage average at each current point seperatly

        Vplot.append(V_av)               # adding the voltage average value to the array Vplot
        Iplot.append(I)                  # adding the corresponding current value to the array Iplot

        # The condition for changing the direction of current flow upon the first encounter with a step.
        # Necessary for full acquisition of the step.
        if ( (b==-1) and (getstep==False) and np.abs(V_av-(harmonic*omega)) < epsilon):
            b = 1
            getstep=True

        if ( (b==1) and (getstep==False) and np.abs(V_av-(harmonic*omega)) < epsilon):
            getstep=True

        I+= b*delta_I                    # increamenting the current to move

        # condition for the change in the direction of current when exiting the step
        if(V_av > harmonic*omega + 0.05):
            b = -1
        # Condition for stopping the cycle based on the current
        if (V_av < harmonic*omega - 0.05 and b==-1):
            break
    # End of cycle by current
    return np.column_stack((np.array(Iplot), np.array(Vplot)))
```

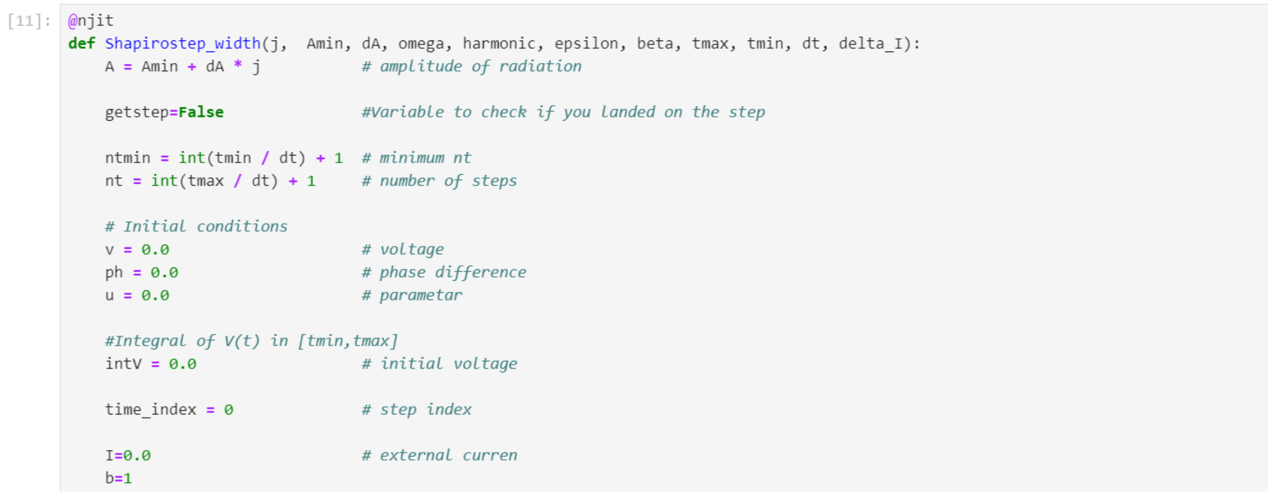
3. shapirostep_width Function: Measuring Step Widths

This function uses the full I-V curve to measure a crucial physical characteristic, the width of the Shapiro steps. These steps are flat voltage plateaus that appear when the Josephson oscillations of the junction synchronize with the frequency of the external AC radiation.

Algorithm Steps and Detailed Calculations:

1. Iterate Through Amplitudes: The function takes a range of AC radiation amplitudes *A_array* and starts a loop to process each one. The goal is to see how the step width changes with increasing amplitude.
2. Generate the Full I-V Curve: Inside the loop, the algorithm calls the *compute_multiloop_cvc* function for the current amplitude. This provides a complete I-V curve that includes the Shapiro steps for that specific radiation level.
3. Find the Step Plateau: The most critical calculation happens here. The program looks for the data points on the curve that lie on the first Shapiro step. The voltage of this step is directly related to the radiation frequency ω. The code uses a precise numerical check to find all points where the voltage is very close to this value. This ensures the entire plateau, not just a single point, is captured.
4. Calculate the Width: Once the current values corresponding to the step plateau are identified, the algorithm calculates the difference between the maximum and minimum current values within that range. This difference is the exact width of the Shapiro step for that amplitude.

This process is repeated for every amplitude, and the final list of step widths is returned for plotting, showing the relationship between step width and radiation amplitude.



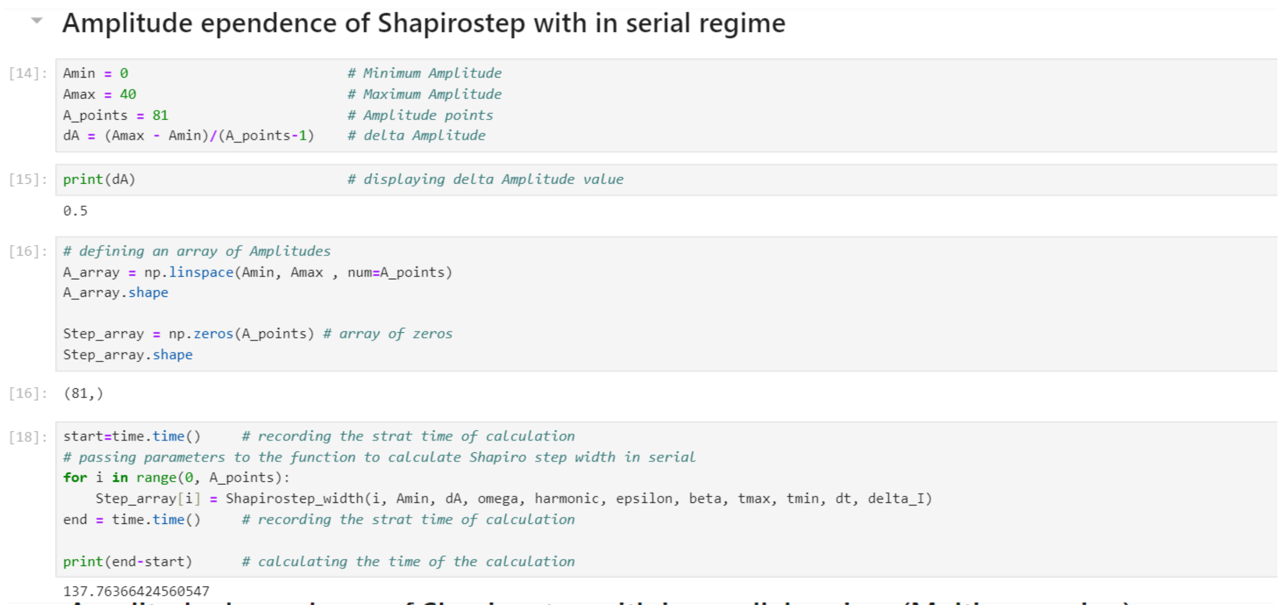
```
[11]: @njit
      def Shapirostep_width(j,  Amin, dA, omega, harmonic, epsilon, beta, tmax, tmin, dt, delta_I):
          A = Amin + dA * j           # amplitude of radiation

          getstep=False               #Variable to check if you landed on the step

          ntmin = int(tmin / dt) + 1  # minimum nt
          nt = int(tmax / dt) + 1     # number of steps

          # Initial conditions
          v = 0.0                     # voltage
          ph = 0.0                    # phase difference
          u = 0.0                     # parametar

          #Integral of V(t) in [tmin,tmax]
          intV = 0.0                  # initial voltage

          time_index = 0              # step index

          I=0.0                       # external curren
          b=1
```

```python
    while I<100:
        for time_index in range(nt):
            # 4th order Runge-Kutta algorithm
            # 1st coefficient
            k1_v = dt * fv(v,ph,u,beta,I,A, omega)
            k1_ph = dt * fph(v)
            k1_u = dt * fu(omega)

            # 2nd coefficient
            k2_v = dt * fv(v+k1_v/2.0 , ph+k1_ph/2.0 , u+k1_u/2.0 , beta , I , A, omega)
            k2_ph = dt * fph(v+k1_v/2.0)
            k2_u = dt * fu(omega)

            # 3rd coefficient
            k3_v = dt * fv(v+k2_v/2.0 , ph+k2_ph/2.0 , u+k2_u/2.0 , beta , I , A, omega)
            k3_ph = dt * fph(v+k2_v/2.0)
            k3_u = dt * fu(omega)

            # 4th coefficient
            k4_v = dt * fv(v+k3_v, ph+k3_ph, u+k3_u, beta , I , A, omega)
            k4_ph = dt * fph(v+k3_v)
            k4_u = dt * fu(omega)

            # Calculations of function values
            v = v + (k1_v + 2*k2_v + 2*k3_v + k4_v)/6.0
            ph = ph  + (k1_ph + 2*k2_ph + 2*k3_ph + k4_ph)/6.0
            u = u + (k1_u + 2*k2_u + 2*k3_u + k4_u)/6.0

            if(time_index >= ntmin):
                intV = intV + v*dt
        # Condition for changing the direction of current flow upon the first encounter with the step
        # Necessary for the complete acquisition of the step
        if ( (b==-1) and (getstep==False) and np.abs(V_av-(harmonic*omega)) < epsilon):
            b = 1
            getstep=True
            Istepmin = I
            Istepmax = I

        if ( (b==1) and (getstep==False) and np.abs(V_av-(harmonic*omega)) < epsilon):
            getstep=True
            Istepmin = I
            Istepmax = I
        #Algorithm for determining the maximum and minimum current values when landing on the step.
        if(np.abs(V_av-(harmonic*omega))<epsilon):
            if (I >= Istepmax):
                Istepmax = I
            if (I <= Istepmin):
                Istepmin = I
        I+= b*delta_I
        #Condition for the change in the direction of the current upon exiting the step.
        if(V_av > harmonic*omega + 0.05):
            b = -1
        #Condition for stopping the cycle based on current
        if (V_av < harmonic*omega - 0.05 and b==-1):
            break
        #End of cycle by current
        #Calculation of step width
    Istep = Istepmax - Istepmin

    return Istep
```

4. Parallel vs. Serial Regimes

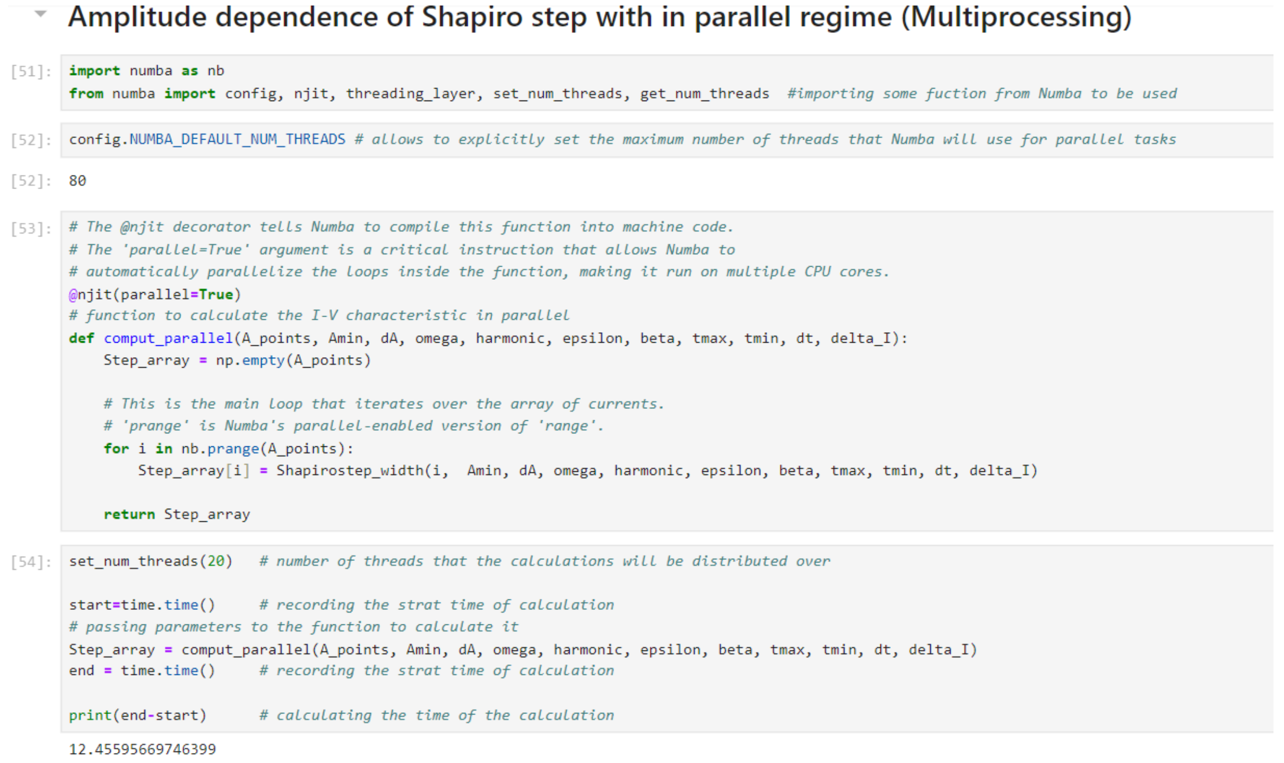
Serial Execution: In a traditional serial approach, the loops in all the above algorithms would run sequentially. This means that the program must complete the full RK4 simulation for the first current value before it can even begin the simulation for the second one. With thousands of data points, this process can be very time-consuming.

Parallel Execution: To dramatically speed up the process, the program utilizes Numba's parallel computing capability. By decorating the main simulation function with *@njit(parallel=True)* and using *prange* for the outer loop, instructing the compiler to automatically distribute the iterations across all available CPU cores. Each core can then independently run the full inner-loop simulation for its assigned current values. This allows for a simultaneous calculation of the I-V curve, which significantly reduces the total execution time.



Amplitude ependence of Shapirostep with in serial regime

```
[14]: Amin = 0                          # Minimum Amplitude
      Amax = 40                         # Maximum Amplitude
      A_points = 81                     # Amplitude points
      dA = (Amax - Amin)/(A_points-1)   # delta Amplitude

[15]: print(dA)                         # displaying delta Amplitude value

      0.5

[16]: # defining an array of Amplitudes
      A_array = np.linspace(Amin, Amax , num=A_points)
      A_array.shape

      Step_array = np.zeros(A_points) # array of zeros
      Step_array.shape

[16]: (81,)

[18]: start=time.time()     # recording the strat time of calculation
      # passing parameters to the function to calculate Shapiro step width in serial
      for i in range(0, A_points):
          Step_array[i] = Shapirostep_width(i, Amin, dA, omega, harmonic, epsilon, beta, tmax, tmin, dt, delta_I)
      end = time.time()     # recording the strat time of calculation

      print(end-start)      # calculating the time of the calculation

      137.76366424560547
```



Amplitude dependence of Shapiro step with in parallel regime (Multiprocessing)

```
[51]: import numba as nb
      from numba import config, njit, threading_layer, set_num_threads, get_num_threads  #importing some fuction from Numba to be used

[52]: config.NUMBA_DEFAULT_NUM_THREADS # allows to explicitly set the maximum number of threads that Numba will use for parallel tasks

[52]: 80

[53]: # The @njit decorator tells Numba to compile this function into machine code.
      # The 'parallel=True' argument is a critical instruction that allows Numba to
      # automatically parallelize the loops inside the function, making it run on multiple CPU cores.
      @njit(parallel=True)
      # function to calculate the I-V characteristic in parallel
      def comput_parallel(A_points, Amin, dA, omega, harmonic, epsilon, beta, tmax, tmin, dt, delta_I):
          Step_array = np.empty(A_points)

          # This is the main loop that iterates over the array of currents.
          # 'prange' is Numba's parallel-enabled version of 'range'.
          for i in nb.prange(A_points):
              Step_array[i] = Shapirostep_width(i,  Amin, dA, omega, harmonic, epsilon, beta, tmax, tmin, dt, delta_I)

          return Step_array

[54]: set_num_threads(20)   # number of threads that the calculations will be distributed over

      start=time.time()     # recording the strat time of calculation
      # passing parameters to the function to calculate it
      Step_array = comput_parallel(A_points, Amin, dA, omega, harmonic, epsilon, beta, tmax, tmin, dt, delta_I)
      end = time.time()     # recording the strat time of calculation

      print(end-start)      # calculating the time of the calculation

      12.45595669746399
```

2.2Numerical Experiment Setup

All simulations were performed on the servers of the Meshcheryakov Laboratory of Information Technologies (MLIT) at the Joint Institute for Nuclear Research (JINR). This research utilized the HybriLIT platform's comprehensive ecosystem for machine learning, deep learning, and data analysis. This platform provided the necessary computational resources for our study, including the ability to run our algorithms within a Jupyter notebook environment. The parameters for the Josephson junction were set to typical values: $\beta = 0.2$, $A = 1.0$, and $\omega = 2.0$. The time step *delta_t* was set to 0.01, and the total simulation time was 600.0. The performance of each approach was measured using Python's *time* module, recording the total execution time for each simulation run. The software used to implement the code is jhub2.jinr.ru

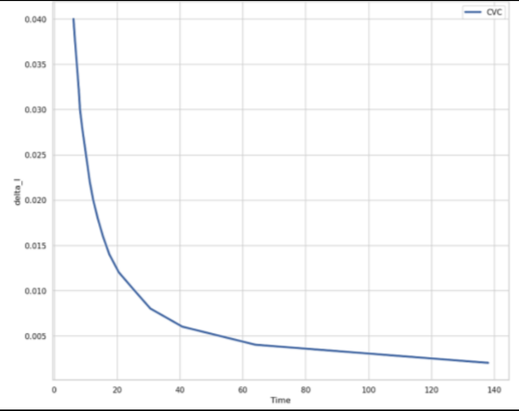
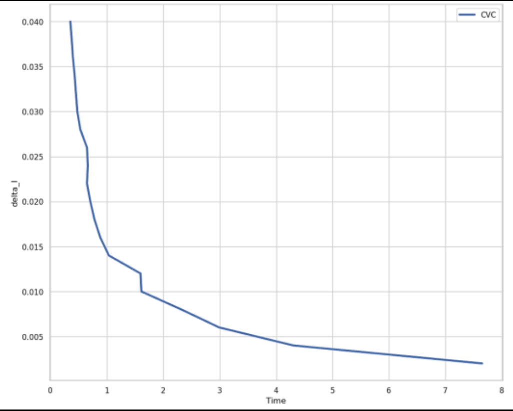
3 Results and Discussion

3.1 Time Comparison as a Function of Current Step (*delta_I*)

Our initial analysis focused on the effect of the current step size *delta_I* on the total simulation time. A smaller (*delta_I*) requires a greater number of steps to generate the full IV curve, thereby increasing the computational load. This experiment directly compared the efficiency of the standard Python implementation with the Numba-optimized version.

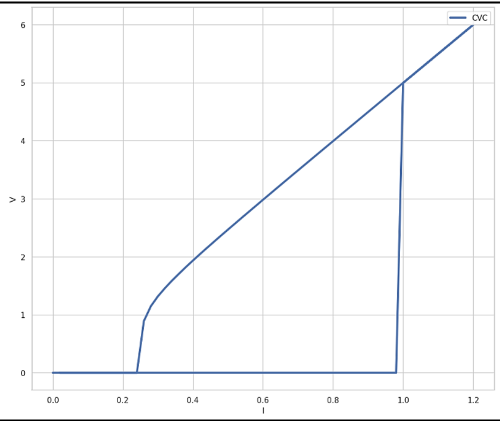
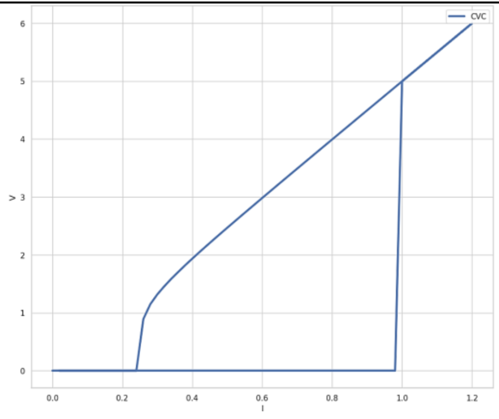
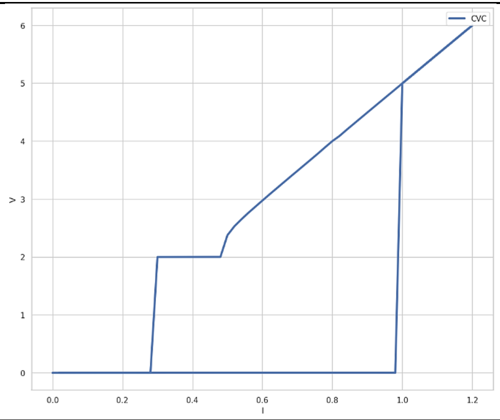
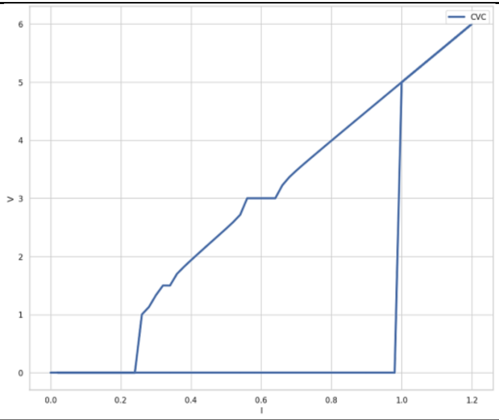
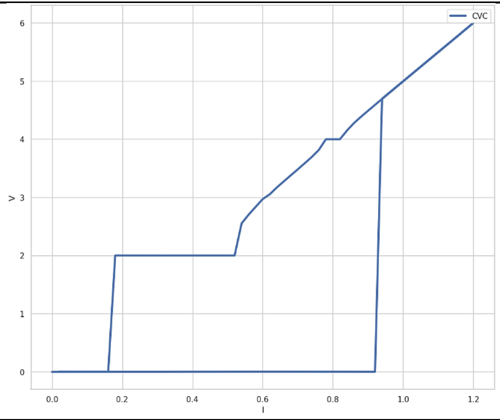
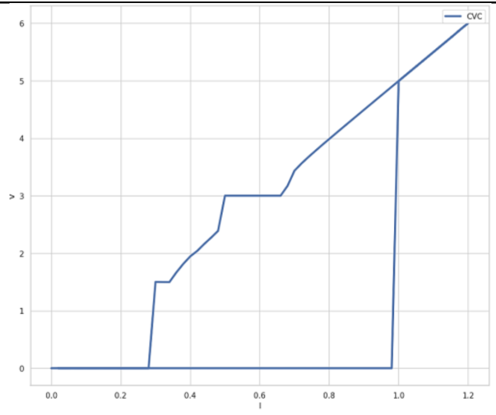
Delta_I	Time in Python	Time in Numba
0.002	138.10495972633362	7.651621103286743
0.004	64.00370717048645	4.305095434188843
0.006	40.72143363952637	2.9919044971466064
0.008	30.562224626541138	2.3209047317504883
0.010	25.48916482925415	1.6125006675720215
0.012	20.52458643913269	1.5969860553741455
0.014	17.481581926345825	1.0366101264953613
0.016	15.453481435775757	0.8847184181213379
0.018	13.78430461883545	0.7810559272766113
0.020	12.36565637588501	0.7076973915100098
0.022	11.27912974357605	0.6483149528503418
0.024	10.474793910980225	0.6627485752105713
0.026	9.659061431884766	0.6486399173736572
0.028	8.849708080291748	0.5307984352111816
0.030	8.161775827407837	0.47760939598083496
0.032	7.800190687179565	0.4531972408294678
0.034	7.378024578094482	0.4306211471557617
0.036	6.9257588386535645	0.39985203742980957
0.038	6.481976509094238	0.37814950942993164
0.040	6.0894434452056885	0.35345935821533203

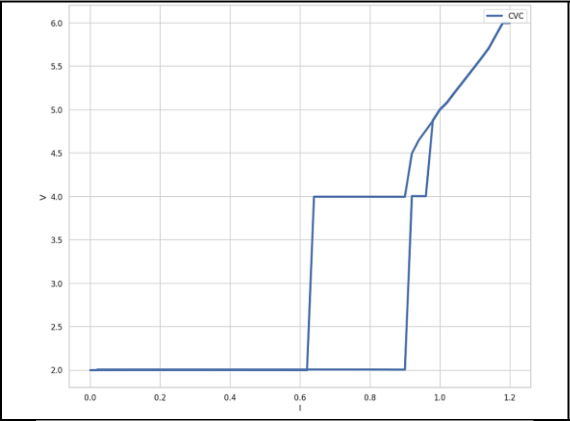
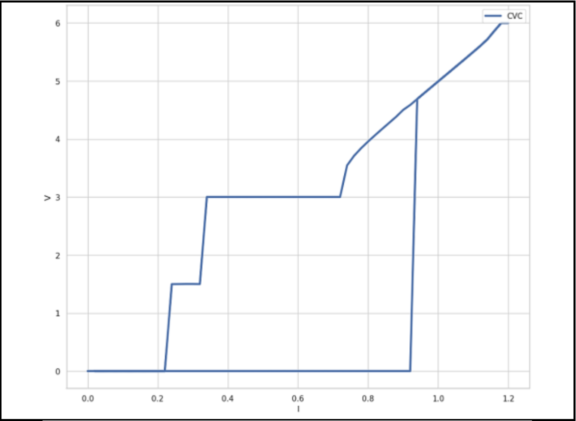
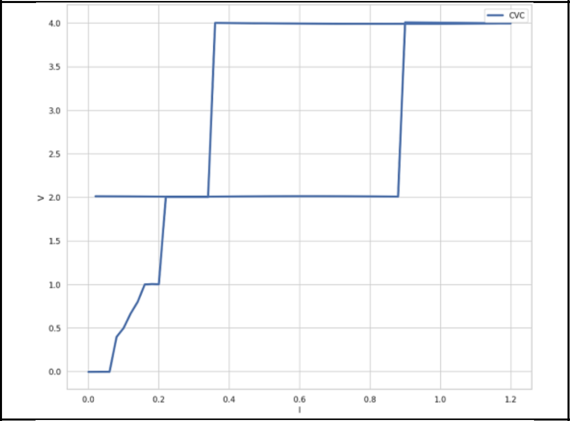
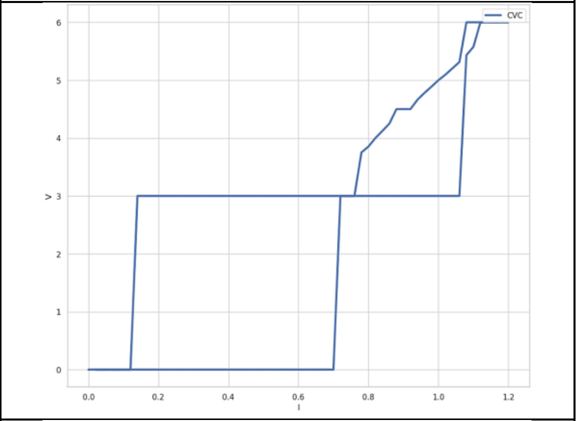
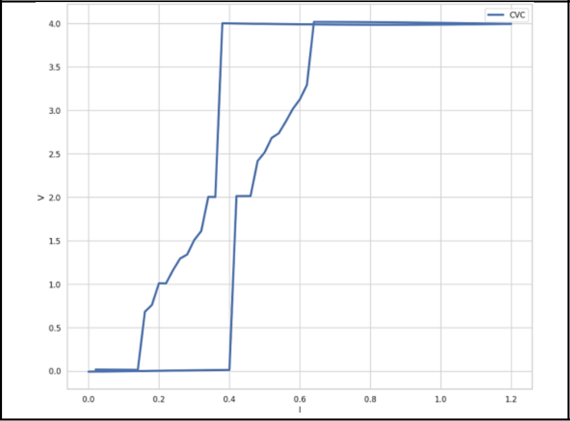
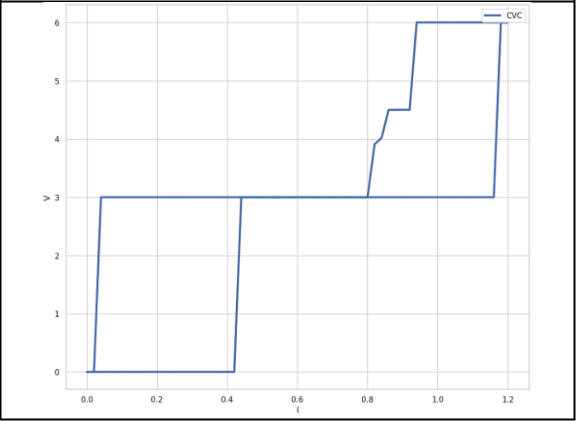
As the table indicates, Numba provides a consistent and substantial speedup, reducing the computation time by an average factor of nearly 18.

The Relationship between *delta_I* and Time in Python:	The Relationship between *delta_I* and Time in Numba:
	

3.2 The Effect of External Radiation

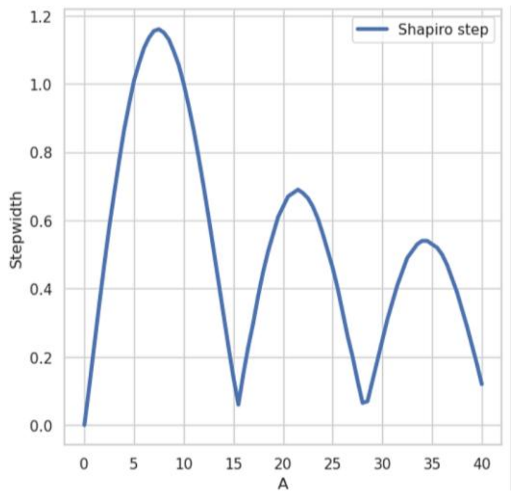
Beyond pure performance, our simulation successfully validated the underlying physics. We specifically examined the impact of external radiation on the junction's behavior. By changing the amplitude (A) and frequency (ω) of the radiation, we were able to observe the distinct "Shapiro steps" on the IV curve. The appearance of these voltage plateaus is a key indicator that our numerical model is physically accurate.

	ω = 2.0	ω = 3.0
A = 0		
A = 1		
A = 2		

A = 5		
A =10		
A =15		

Our results demonstrated that for a given frequency, increasing the radiation amplitude (A) leads to a wider Shapiro step. This confirms that the external AC signal is locking the Josephson oscillations, creating a constant voltage plateau. We also verified that the voltage height of these steps is directly proportional to the radiation frequency (ω) which was observed by running the simulation for $\omega = 2$ and $\omega = 4$.

Amplitude dependence of Shapiro step:



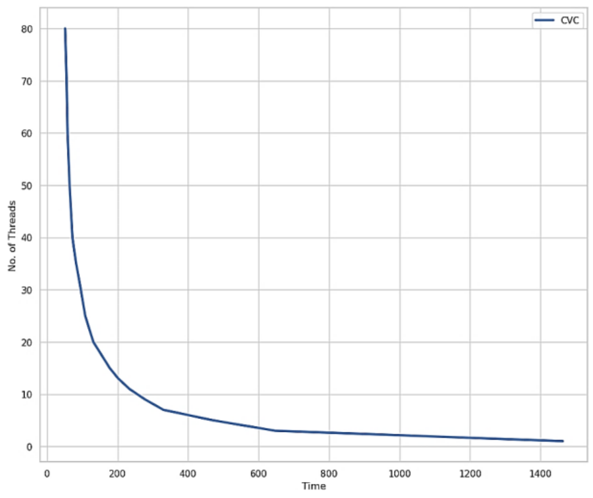
3.3 Number of Threads effect on Time in parallel processing using Numba

We measured the computation time for a large-scale simulation as we increased the number of threads. The results in the table below are a clear testament to the power of parallel processing. Computation time drops significantly as the workload is distributed across the CPU's thousands of cores.

For *Amin* = 0*, Amax* = 50*, A_points* = 1000, *delta_I* = 0.005

Number of threads	Time
1	1463.2822465896606
3	646.8552927970886
5	470.63821506500244
7	330.18310737609863
9	277.8020534515381
11	233.83460140228271
13	202.074951171875
15	177.89794945716858
20	131.38784074783325
25	108.38056492805481
30	96.0047459602356
35	82.63569712638855
40	72.0907576084137
50	64.08810877799988
60	58.373080015182495
70	55.60375905036926
80	51.36244773864746

The relationship between number of threads and time of calculations:



3.4 Performance Comparison

The combined results from our experiments establish a clear hierarchy of efficiency. The Python implementation, while flexible, is limited by its interpreted nature. Numba addresses this by providing a straightforward method to optimize CPU performance.

Framework	**Computational Approach**	**Suitability**	**Observed Performance**
Python	High-level, interpreted loops	Prototyping and testing	Slowest (Baseline)
Numba	just-in-time (JIT) compilation for CPU	Accelerating single-node tasks	Significant Speedup

3.5 Analysis and Scalability

Our findings suggest that the choice of framework should be determined by the scale and nature of the problem. While standard Python is excellent for rapid prototyping, its interpreted nature makes it unsuitable for performance-critical tasks, Numba provides a powerful and accessible tool for optimizing performance (CPU code) on a single machine. This project demonstrates that applying the correct high-performance computing tool can provide invaluable performance benefits in computational physics.

4 Conclusion

This project successfully demonstrated a robust and reliable numerical method for solving the Josephson junction equations. The simulation accurately reproduced the characteristic IV curve and the Shapiro steps, validating the correctness of the code. Furthermore, the comparative analysis of Python and Numba revealed significant performance differences. Numba provided a considerable speedup for CPU-based computation. This work provides a valuable framework for using high-performance computing tools to solve complex problems in physics.

5 References

- Python by example - NICHOLA LACEY
- https://docs.python.org/3/
- Numba documentation — Numba 0.52.0.dev0+274.g626b40e-py3.7-linux-x86_64.egg documentation
- Toolkit in Python for Simulation of Shapiro Step on the Current–Voltage Characteristic of a Josephson Junction
- short_jj_radiation_cvc_v5_parallel.ipynb

6 Acknowledgement

I'm incredibly grateful for the support and guidance I received throughout this research. My heartfelt thanks go to my supervisor Adiba Rahmonova, whose expertise and continuous encouragement were essential to this project. Thanks to the Laboratory of Information Technologies of JINR for the opportunity to use the computational resources of the HybriLIT Heterogeneous Platform I also want to thank the Joint Institute for Nuclear Research (JINR) for providing me with this invaluable opportunity and the resources I needed to complete my work.