# Assignment 3: Practice with C++ and Cython

PHYS 5v48.002 – HPC S24, Aaron Smith

## ABSTRACT

This assignment is designed to enhance your proficiency in computational methods using three different programming approaches: pure Python, Cython, and C++. You will tackle a series of numerical problems that are both fundamental to scientific computing and also challenging enough to give you valuable practice with these languages. The focus is on implementing efficient algorithms, understanding the additional syntax and structure of Cython and C++, and comparing the runtime performance across these programming paradigms. Each problem requires you to produce a numerical result, which you will verify against known solutions, and to generate plots illustrating the behavior or accuracy of your solution.

## 1. INSTRUCTIONS

The problems are designed to be self-contained, so you can tackle them in any order, but you should aim to complete all of them by the end of the assignment. You will submit your source code and a brief report for each problem, discussing your approach, runtime comparisons, and any interesting findings or challenges you encountered. Your code should be well-commented and adhere to good programming practices. Your work will be evaluated based on the correctness of your implementations, the efficiency of your solutions, the quality and clarity of your plots and discussion notes. Please reflect on the pros and cons of each language, taking advantage of their unique features. For example, use NumPy and Numba to optimize your Python code, leverage Cython's additional syntax for type declarations, and demonstrate the benefits of using template classes in C++ for generic programming. It is okay to use Fortran, Rust, or Julia, instead of C++ if this is more useful for your research.

**Implementation:** For each problem, provide three separate implementations: one in Python (using any library as this is mainly for prototyping and sanity checking), one in Cython (using only standard libraries and NumPy arrays), and one in C++ (using only standard libraries like `std::vector<double>` to get low-level practice). Each code can be implemented in a single file, but you should use classes and functions to organize your code and make it more readable and reusable. Use Github to manage your code, working with at least one commit per problem (you can also practice making and then merging feature branches). Parallelism is not required for this assignment, but you can use it if you want to practice.

**Runtime Comparison:** After implementing each problem in the three languages, measure and compare the runtime of each solution (e.g., either using time at the command line or `time` in Python and the `std::chrono` library in C++). Discuss the differences observed and how language features or optimizations might contribute to these differences.

**Plotting:** You should separate the plotting code from the main code and use a common plotting script on the outputs of all three languages by reading from different data files. Each problem includes a component that requires visualization, such as error analysis, histograms, or convergence behavior. The bulk of the calculations should be done in the language-specific codes.

### 1.1 Inverse Transform Sampling

**Goal:** Learn how to generate random numbers from a non-uniform distribution using inverse transform sampling, a fundamental technique in computational physics and Monte Carlo simulations.

1. Understand the principle of generating uniform random numbers and transforming them through the inverse cumulative distribution function (CDF) of a target distribution, in this case, the Lorentzian distribution.
2. Implement the transformation $x = \Gamma / \tan(\pi u)$, where $u$ is uniformly distributed between 0 and 1, and $\Gamma = 1$ is the half-width at half-maximum (HWHM). (Recall the programs to calculate $\pi$ already prototyped generating random numbers uniformly.)

3. Create a histogram of $n = 10^8$ sampled values and save the counts to a file. Normalize the histogram and overlay the theoretical probability density function (PDF) $p(x) = (\Gamma/\pi)/(\Gamma^2 + x^2)$ for visual comparison. The histogram should closely resemble the PDF, validating the inverse transform sampling method.

### 1.2 Numerical Integration

**Goal:** Approximate the Bessel function $J_0(x)$ through numerical integration of $f(\theta) = \cos(x \sin\theta)$, for a better understanding of special functions and numerical integration techniques.

1. Plot $f(\theta)$ for a small set of $x$ values to visualize the behavior of the integrand across the interval $[0, \pi]$.
2. Use the trapezoid and Simpson's rule to approximate $\frac{1}{\pi}\int_0^\pi \cos(x \sin\theta)\,\mathrm{d}\theta$, comparing the results against $J_0(x)$ computed using library functions for a range of $x$ values in the domain $[0, 10]$. (In C++, you can use `std::cyl_bessel_j(0, x)` from the `<cmath>` header in C++17 and later. In Python, you can use `scipy.special.j0(x)`).
3. Conduct error analysis by evaluating the accuracy of each method at $x = 3.83171$, where $J_0(x)$ has a known extremum. Specifically, plot the relative errors as functions of the (log) number of steps used in the approximation: $N = 10 \cdot 2^n$ with $n = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$. Can you determine the convergence order of each method (i.e., the slope of the error curve in the log-log plot)?

### 1.3 Series Convergence

**Goal:** Investigate the properties of a slowly converging series. You will analyze how the summation order affects the accuracy of the series approximation, using the alternating harmonic series: $\ln(2) = \sum_{n=1}^{\infty}(-1)^{n+1}/n$.

1. Implement the alternating harmonic series summation for both ascending (from 1 to $N$) and descending (from $N$ to 1) ordered summations.
2. For a range of $N$ terms (e.g., $N = 2^n$ with $n = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$), compute and store the partial sums of the ascending and descending series in arrays/vectors to facilitate subsequent error analysis.
3. Calculate the relative error for each partial sum by comparing it to the exact value of $\ln(2)$. (Note: For descending order sums with NumPy, you may need to use reverse cumulative sums to correctly calculate the errors from the smallest terms.)
4. Plot the (log) relative error as a function of the (log) number of terms for both ascending and descending order sums. This will visually demonstrate the effect of summation order on convergence accuracy.
5. Discuss the convergence behavior observed and explain why summing from the smallest terms typically results in a more accurate approximation for this series, for a given number of terms.
6. Optional: Apply this same analysis to a series that converges more quickly, such as the Maclaurin series for $e$: $e = \sum_{n=0}^{\infty} 1/n!$. Does the summation order have a similar effect on the accuracy?

### 1.4 Nonlinear Dynamics

**Goal:** Explore chaotic behavior and bifurcation in the logistic map, a simple yet powerful model for understanding nonlinear dynamics and chaos in discrete systems. The logistic map is defined as $x_{n+1} = r x_n(1 - x_n)$, where $0 \le x_n \le 1$ represents the normalized population size at the $n$-th generation, and $r$ is a parameter that influences the system's dynamics. This model exhibits a wide range of behaviors, from stable fixed points to chaotic dynamics, as $r$ varies.

1. Implement the logistic map using a function or class method to compute $x_{n+1}$ given $x_n$ and $r$.

2. For $r$ values in the range $[1, 4]$, iterate the logistic map for a large number of generations (e.g., 1000) to ensure the system reaches its steady state, starting from an initial condition $x_0 = 0.5$. Discard the initial transients by ignoring the first 500 iterations and save the subsequent $x_n$ values.

3. Plot a bifurcation diagram by gradually varying $r$ from 1 to 4 and plotting the steady-state $x_n$ values. This diagram reveals the transition from stability to chaos, showcasing fixed points, periodic orbits, and chaotic regions. Verify that your diagram correctly displays the transition from stable fixed points to periodic doubling and eventually to chaos as $r$ increases.

4. Calculate the Lyapunov exponent for a range of $r$ values to quantitatively characterize the system's sensitivity to initial conditions. The Lyapunov exponent $\lambda$ is defined as $\lambda = \lim_{N\to\infty} \frac{1}{N} \sum_{n=1}^{N} \ln |f'(x_n)|$, where $f'(x_n) = r(1 - 2x_n)$ is the derivative of the logistic map. A positive Lyapunov exponent indicates chaotic behavior, while a negative value suggests stability.

5. Discuss the implications of the bifurcation diagram and the variation of the Lyapunov exponent with $r$. Reflect on how this simple model demonstrates complex nonlinear dynamics, including the onset of chaos. Ensure that the calculated Lyapunov exponent transitions from negative to positive values as you move from stable to chaotic regions, confirming the expected dynamical behavior.

### 1.5 Root Finding

**Goal:** Apply Newton's method, Halley's method, and the method of successive approximation to find roots of the nonlinear equation $f(x) = \tan(x) - x$, which has an infinite number of roots and is representative of nonlinear problems encountered in the study of eigenvalues for boundary value problems. Due to its transcendental nature, analytical solutions are not feasible, making iterative numerical methods essential. Thus, this task aims to compare the convergence properties and efficiency of different iterative root-finding techniques. Ensure that the methods converge to a root within a specified tolerance (e.g., $10^{-12}$) for a range of initial guesses. The first root is trivially at $x = 0$, but the other roots are nontrivial and can be found in the range $[x_{\min}, x_{\max}] = [(n - \frac{1}{2})\pi, (n + \frac{1}{2})\pi]$ for $n \in \mathbb{N}$, and your functions should specify which root they are finding (`int n > 0`). The starting point for each method should be $x_0 = n\pi$ and if the updated guess is outside the range, it should be adjusted to lie within the range, e.g. if $x_{n+1} \geq x_{\max}$ then update with $x_{n+1} = (x_{n+1} + x_{\max})/2$, and similarly for $x_{n+1} \leq x_{\min}$.

1. To understand the behavior of the equation, plot $f(x)$ over the range $[0, 8]$ to visualize its roots and the challenges posed by its nonlinear and transcendental nature.

2. Implement Newton's method for this equation, which is an efficient and widely used root-finding technique when the derivative is known. It uses the first order Taylor expansion to iteratively refine the root estimate via $x_{n+1} = x_n - f_n/f'_n$ where $f_n = f(x_n)$ and $f'_n = f'(x_n)$. In this case the derivative is $f'(x) = \sec^2(x) - 1$.

3. Implement Halley's method, which is a higher-order extension of Newton's method that uses the second derivative to improve the rate of convergence. If we define $f''_n = f''(x_n)$ and the ratios $a_n = f_n/f'_n$ and $b_n = f''_n/f'_n$, then the iteration formula is $x_{n+1} = x_n - a_n/(1 - a_n b_n/2)$. In this case the second derivative is $f''(x) = 2\sec^2(x)\tan(x)$.

4. Implement the method of successive approximation (fixed-point iteration) by iteratively evaluating $x_{n+1} = \tan(x_n)$, which is a simple and intuitive method for solving transcendental equations. This method can be less efficient than Newton's and Halley's methods, but it is often more robust and easier to implement.

5. For each method, analyze the number of iterations required to converge to a root within a specified tolerance (e.g., $10^{-12}$). Plot the (log) relative error as a function of the iteration number to visually assess the convergence behavior. Which method converges most rapidly and reliably to the roots of the equation? Which method is the most efficient for this problem?

### 1.6 Implicit ODE Solver

**Goal:** Implement the backward Euler method, an implicit ordinary differential equation (ODE) solver, to approximate the solution of the exponential decay equation $\mathrm{d}y/\mathrm{d}t = -y$. Implicit methods are particularly useful for stiff ODEs, where the explicit methods may require very small time steps to maintain stability. The backward Euler method uses the function value at the next time step, $y_{n+1}$, to approximate the derivative at that step. For the exponential decay ODE, the backward Euler update formula is $y_{n+1} = y_n/(1 + \Delta t)$, where $\Delta t$ is the time step.

1. Implement the backward Euler method for the exponential decay ODE. Use an initial condition $y_0 = 1$ and solve the ODE over a time interval (e.g., $t = 0$ to $t = 16$) with a fixed time step $\Delta t$.
2. Compare the numerical solution at each time step with the analytical solution $y(t) = e^{-t}$. Plot the (log) relative error at $t = 16$ as a function of the size of the fixed time steps $\Delta t = 16/2^n$ with $n = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
3. Discuss how the time step influences the accuracy of the solution, especially for large $\Delta t$. Discuss the implicit method's stability characteristics and its suitability for stiff ODEs.

## 2. EXAMPLE CODE

**Cython Specifics:** Practice using Cython's additional syntax for type declarations and compile your Cython code effectively to optimize performance following the example here. Note that you can use the same `setup.py` file for all problems, but you should have a separate `.pyx` file for each problem. You can also use the `distutils` module to compile your Cython code and create a shared library that can be imported into Python. You can also use the `timeit` module to measure the runtime of your Cython code.

**C++ Specifics:** Your C++ programs should start with a basic main function that reads command line arguments (if necessary) and saves results to a text file or redirects screen output to a file. You can use common header files (e.g. `utils.h`) to store common functions and classes. As an example, here is a C++ program that generates a vector of numbers using a `linspace` function and saves the results to a file. First the `utils.h` file:

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <sstream>

// Analogous to the linspace function in Python
std::vector<double> linspace(double start, double end, int n) {
    std::vector<double> vec(n); // Create a vector of size n
    const double increment = (end - start) / (n - 1); // Calculate the increment
    for (int i = 0; i < n; ++i)
        vec[i] = start + double(i) * increment; // Fill the vector
    return vec;
}

// Print a vector to the standard output
template <typename T>
void print(const std::vector<T>& vec) {
    for (const auto& elem : vec)
        std::cout << elem << " "; // Print each element
    std::cout << std::endl; // Print a newline
}
```

```cpp
// Print a vector to a file
template <typename T>
void save(const std::vector<T>& vec, const std::string& filename,
            const std::string& header = "") {
    std::ofstream file(filename); // Open the file
    if (file.is_open()) { // Check for successful opening
        if (!header.empty())
            file << "# " << header << std::endl; // Write the header
        for (const auto& elem : vec)
            file << elem << " "; // Write each element
        file << std::endl; // Write a newline
        file.close(); // Close the file
    } else {
        std::cerr << "Unable to open file " << filename << std::endl;
    }
}
```

Then the `example.cpp` file:

```cpp
#include "utils.h"

// Main function
int main(int argc, char* argv[]) {
    if (argc < 3) {
        std::cerr << "Usage: " << argv[0] << " <int n> <string filename>" << std::endl;
        return 1;
    }

    // Parse command line arguments
    int n; std::istringstream(argv[1]) >> n; // Number of elements
    std::string filename = argv[2]; // Output filename

    // Create a vector of n elements
    auto vec = linspace(0., 1., n);

    // Print the vector to the standard output
    print(vec);

    // Print the vector to the specified file
    std::string header = "linspace(0, 1, " + std::to_string(n) + ")";
    save(vec, filename, header);

    return 0;
}
```

To compile and run this script (assuming it is saved as `example.cpp`), you can use the following commands saved in a bash script (e.g. `run.sh`) to compile and run the program with command line arguments: [Note: use clang++ instead of g++ on macOS]

```bash
#!/bin/bash
g++ -std=c++17 -O3 -o example example.cpp
time ./example 11 example.txt
```