Introduction to Computational Physics Lab

UPES Dehradun

Introduction to Computational Physics Lab - 2024

Nitesh Kumar

October 10, 2024

Contents

1 Frequency Distribution **5**

2 Finite and Infinite Series **9**
- 2.1 Introduction . . . 9
- 2.2 Finite Series . . . 9
- 2.3 Infinite Series . . . 12

3 Matrix multiplication **17**
- 3.1 C++ Code for Dot Product of Two 3x3 Matrices . . . 17
- 3.2 Explanation of `void` in C++ . . . 18
  - 3.2.1 As a Return Type for Functions . . . 18
  - 3.2.2 As an Empty Argument List . . . 18
- 3.3 Fortran Code for Dot Product of a 3x3 Matrix . . . 19

4 Prime numbers and Fibonacci Series **21**
- 4.1 C++ Code for Finding a Set of Prime Numbers . . . 21
- 4.2 C++ Code for Printing the Fibonacci Series . . . 22

Chapter 1

Frequency Distribution

Step 1: Prepare Your Data

Let's assume you have a data file named `data.txt` with raw data points:

```
10
12
10
15
17
10
12
15
20
15
```

Step 2: Create the Frequency Distribution and Save it to a File

Use a shell command to generate the frequency distribution and save it in a file named `freq_data.txt`:

```
$ sort data.txt | uniq -c | awk '{print $2, $1}' > freq_data.txt
```

Explanation:

- `sort data.txt`: Sorts the data.
- `uniq -c`: Counts the frequency of each unique value.
- `awk '{print $2, $1}'`: Reorders the output so that the value appears first, followed by the frequency.

The resulting `freq_data.txt` will look like this:

```
10 3
12 2
15 3
17 1
20 1
```

Step 3: Compile the Frequency Distribution and Evaluate Mean and Standard Deviation using Gnuplot

Now, you'll use Gnuplot to calculate the mean and standard deviation from this frequency distribution.

Create a Gnuplot script `calc_stats.gnuplot`:

```
# calc_stats.gnuplot

# Load the frequency distribution
stats "freq_data.txt" using 1:2 name "freq" nooutput

# Calculate the mean (weighted by frequency)
mean = freq_mean_y

# Calculate the standard deviation
sd = freq_stddev_y

# Print results
print "Mean =", mean
print "Standard Deviation =", sd
```

Step 4: Run the Gnuplot Script

Execute the Gnuplot script to calculate and display the mean and standard deviation:

```
$ gnuplot calc_stats.gnuplot
```

Explanation:

- `stats "freq_data.txt" using 1:2 name "freq" nooutput`: This command calculates the sum, sum of squares, and other statistical measures from the frequency distribution.
- `mean = freq_sum_y / freq_sum`: Computes the mean by dividing the sum of values (weighted by their frequency) by the total number of observations.
- `sd = sqrt((freq_sum_y2 / freq_sum) - (mean * mean))`: Computes the standard deviation using the variance formula.

Step 5: Output Interpretation

After running the Gnuplot script, it will print the mean and standard deviation to the console.

Summary

- You first sort your data and generate a frequency distribution using shell commands.
- Save the frequency distribution to a file.
- Use Gnuplot to load the frequency data, compute the mean, and calculate the standard deviation.

For Plotting:

Here's a complete Gnuplot script that plots the frequency distribution and marks the mean and standard deviation on the plot. We'll assume you have your frequency distribution saved in `freq_data.txt`.

Gnuplot Script: `plot_with_stats.gnuplot`

```
# plot_with_stats.gnuplot

# Load the frequency distribution data
stats "freq_data.txt" using 1:2 name "freq" nooutput

# Calculate mean and standard deviation
mean = freq_mean_y
sd = freq_stddev_y

# Configure the plot
set title "Frequency Distribution with Mean and Standard Deviation"
set xlabel "Value"
set ylabel "Frequency"
set style data histograms
set style fill solid 0.5 border -1
set boxwidth 0.9

# Plot the frequency distribution
plot "freq_data.txt" using 2:xtic(1) title "Frequency" with boxes lc rgb "blue", \
     "" using (mean):0 title "Mean" with lines lw 2 lc rgb "red", \
     "" using (mean-sd):0 title "Mean - 1 SD" with lines lw 1 lc rgb "green" dt 2, \
     "" using (mean+sd):0 title "Mean + 1 SD" with lines lw 1 lc rgb "green" dt 2

# Optional: Show the calculated mean and standard deviation on the plot
set label sprintf("Mean = %.2f", mean) at graph 0.02, graph 0.95 textcolor rgb "red"
set label sprintf("Standard Deviation = %.2f", sd) at graph 0.02, graph 0.90 textcolor rgb "green"

# Replot to ensure labels are included
replot
```

Step-by-Step Explanation

1. `stats "freq_data.txt" using 1:2 name "freq" nooutput`: This calculates the statistics, storing them with the prefix `freq_`.

2. The mean is calculated as `mean = freq_sum_y / freq_sum`.

3. The standard deviation is calculated using `sd = sqrt((freq_sum_y2 / freq_sum) - (mean * mean))`.

4. Plot settings are configured with `set style data histograms`, `set style fill solid 0.5 border -1`, and `set boxwidth 0.9`.

5. The frequency distribution is plotted as a histogram, with the mean and standard deviation bounds marked by vertical lines.

6. Labels showing the calculated mean and standard deviation are added to the plot.

Running the Script

To execute this script, save it as `plot_with_stats.gnuplot` and run it with Gnuplot:

```
$ gnuplot -persist plot_with_stats.gnuplot
```

This script will produce a histogram of your frequency distribution, with the mean and standard deviation clearly marked.

Chapter 2

Finite and Infinite Series

2.1 Introduction

In physics, we often require to derive the values of few functions such as `sin (x), cos (x),`. These functions can be expressed by infinite series, infinite products or continued ffractions. For example:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + ... = \sum_{n=0}^{\infty} \frac{x^n}{n!} \tag{2.1}$$

$$sin(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - .... = \sum_{n=0}^{\infty} (-1)^n \frac{x^{(2n+1)}}{(2n+1)!} \tag{2.2}$$

The numerical methods to derive the values of these expressions will be discussed here.

2.2 Finite Series

Consider the following finite sum of a series:

$$S_n(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + ... + \frac{x^n}{n!} \tag{2.3}$$

Here each term is of the form $\frac{x^i}{i!}$; with i = 0, 1, 2, ..., n. As long as n is a small number, there is no problem and we can actually evaluate each term and then sum them up. However, if we wish to find the sum of this series for large n, say n = 20, there is a serious problem - the computer cannot handle large numbers and 20! is a "very large" number ($\sim 2.4 \times 10^{18}$). So clearly we need to find another way to summing of series with very large or very small terms.

We overcome this problem by not evaluating individual terms of the series. Instead we find the ratio of two consecutive terms, t_i and t_{i-1} . Suppose this ratio is R. Then $t_i = Rt_{i-1}$. Since R is usually a small number, it is possible to find all the terms, given the first term t_0, by assigning to i the values 1, 2, 3, ... By adding these terms we get the required sum.

In the specific example of the series Eq 2.3 above, we can easily see that

$$t_i = \frac{x^i}{i!} \tag{2.4}$$

$$t_{i-1} = \frac{x^{i-1}}{(i-1)!} \tag{2.5}$$

$$R = \frac{t_i}{t_{i-1}} = \frac{x}{i} \tag{2.6}$$

Therefore, starting with $t_0 = 1$, we get

$$i = 1 \qquad t_1 = Rt_0 = x \tag{2.7}$$

$$i = 2 \qquad t_2 = Rt_1 = \frac{x}{2}x = \frac{x^2}{2} \tag{2.8}$$

$$i = 3 \qquad t_3 = Rt_2 = \frac{x}{3}\frac{x^2}{2} = \frac{x^3}{3\times 2} = \frac{x^3}{6} \tag{2.9}$$

and so on. We then define a quantity called the j-th partial sum S_j as

$$S_j = \sum_{i=0}^{j} t_i \tag{2.10}$$

Note an interesting property of this quantity. Any partial sum is by definition the sum of the previous partial sum and the term itself. Thus,

$$S_5 = \sum_{i=0}^{5} t_i = \left(\sum_{i=0}^{4} t_i\right) + t_5 = S_4 + t_5 \tag{2.11}$$

This is a property we can use to sum the series iteratively. Thus, the algorithm for summing a finite series to a given number of terms is simple.

1. Find t_0 or t_1 , the first term of the series.
2. Find R, the ratio of the i^{th} term.
3. Find S_0 or S_1 , the first partial Sum.
4. From t_0 or t_1 and R, find the next term.
5. Add the next term to the first partial Sum to get the second partial Sum.
6. Repeat this process till we get the required partial Sum which is the Sum of the finite series.

The following program can carry out this process:

```
! Program for evaluating a finite series
PROGRAM finite_series
    IMPLICIT NONE
    REAL :: x, t, s
    INTEGER :: n, i

    PRINT *, 'Supply x and the number of terms n:'
    ! If n = 20, the last term is x^20 / 19!
    READ *, x, n

    s = 1.0
    t = 1.0  ! Initial values of sum s and the first term t

    ! The following loop evaluates the terms and sums them
    DO i = 1, n-1  ! i starts at 1; t=0 term is the initial value
        t = t * x / i  ! x/i is simply the ratio R
        s = s + t
    END DO

    PRINT *
    PRINT *, 'x =', x, ' n =', n, ', sum =', s
END PROGRAM finite_series
```

Here is the same code in C++:

```
/* Program for evaluating a finite series */
#include <iostream>
#include <cmath>  // For math functions if needed

using namespace std;

int main() {
    float x, t, s;
    int n, i;

    cout << "Supply x and the number of terms n: \n";
    /* If n = 20, the last term is x^{20} / 19! */
    cin >> x >> n;

    s = 1.0;
    t = 1.0;  // Initial values of sum s and the first term t

    /* The following loop evaluates the terms and sums them */
    for (i = 1; i < n; i++) {  // i starts at 1; t=0 term is the initial
   value
        t *= x / i;  // x/i is simply the ratio R
        s += t;
    }

    cout << "\n";
    cout << "x = " << x << " n = " << n << ", sum = " << scientific << s <<
    "\n";

    return 0;
}
```

In this program, the statement $s+ = t$ generates the partial sums $S_2(x), S_3(x), \ldots$ while $t* = \frac{x}{i}$ generates the successive terms for $i = 1, 2, \ldots, n$.

Note

Note that the order of the statements $t* = \frac{x}{i}$ and $s+ = t$ is important. What happens if they are interchanged? Note also that the initialization $s = 1.0$ and $t = 1.0$ must be done outside the loop over i. What happens if these are done within the loop?

Of course, instead of taking the ratio of t_i and t_{i-1}, we could also take the ratio of t_{i+1} and t_i. In this case,

$$t_i = \frac{x^i}{i!} \tag{2.12}$$

$$t_{i+1} = \frac{x^{i+1}}{(i+1)!} \tag{2.13}$$

$$R = \frac{t_{i+1}}{t_i} = \frac{x}{i+1} \tag{2.14}$$

Note

Here that i will now start from 0 and the first term is $t_0 = 1$. These two methods are equivalent provided we take care of the initialization of i and t.

2.3 Infinite Series

Whereas a finite series can always be summed in principle, the sum of an in infinite series has a meaning only if the series is convergent. So it must be ensured that the series under consideration is indeed convergent before one embarks on its evaluation. For finite series, the number of terms to be summed is given in advance. However, in the case of an infinite series, obviously an infinite number of terms cannot be summed. So how do we sum an infinite series? The answer lies in the fact that if the series is convergent, then by definition it means that adding more and more terms to the partial sum, changes the partial sums by smaller and smaller amounts. Thus, if we decide that we want the sum of an infinite series to a given accuracy, then we can stop adding the sums. In effect, what we are doing is actually summing again a finite series though here we do not before hand how many terms we need to some to achieve the desired accuracy.

To illustrate, consider the simple case of sin(x). We know that this function can be written as an infinite series,

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} \tag{2.15}$$

and the term becomes:

$$t_i = (-1)^i \, \frac{x^{2i+1}}{(2i+1)!} \tag{2.16}$$

and,

$$t_{i-1} = (-1)^{i-1} \, \frac{x^{2(i-1)+1}}{(2i+1)!} \tag{2.17}$$

so,

$$R = \frac{t_i}{t_{i-1}} \tag{2.18}$$

$$= \frac{x^2}{(2i+1)(2i)} \tag{2.19}$$

$$\tag{2.20}$$

Clearly the first term, t is x. What about s? The initial partial sum is obviously the initial term. Thus the initial values are t = x = s; i = 1.

We can write a program to sum this series to any number of terms for a given value of x, say x = $\pi/4$. We know that the result of $\sin(\pi/4) = 0.7071$. The program below will evaluate the series upto increasing number of terms till 10. For each term, we will print the value of that term and the partial sum.

```cpp
/* Program for evaluating a finite series */
#include <iostream>
#include <cmath>  // For math functions if needed
#include <fstream>
#define pi 3.14159

using namespace std;

int main() {
    float x, t, sum;
    int n, i;
    ofstream fp;
    fp.open(”res.txt”)


    x = pi/4.0;
    sum = x
    t = x

    /* The following loop evaluates the terms and sums them */
    for (i = 1; i < 10; i++) {  // i starts at 1; t=0 term is the initial value
        t *= (x*x) / ((2*i+1)*(2*i));
        sum += t;
        fp << i << '\t' << t << '\t' << sum << '\t' << sin(x) << endl;
    }
    fp.close()
    return 0;
}
```

You should get the following output in the file:

i	t_i	sum	Sin(x)
1	-0.0807453	0.704652	0.707106
2	0.00249038	0.707143	0.707106
3	-3.6576e-05	0.707106	0.707106
4	3.13359e-07	0.707106	0.707106
5	-1.75723e-09	0.707106	0.707106
6	6.94838e-12	0.707106	0.707106
7	-2.041e-14	0.707106	0.707106
8	4.62864e-17	0.707106	0.707106
9	-8.34847e-20	0.707106	0.707106

Table 2.1: The output file ‘res.txt’.

As you see, this being a very rapidly converging series, after the first four terms, the partial sum really does not change and so adding more and more terms will not help. So instead of adding up a large number of terms, we can add a few terms and get the desired result. Of course, the successive terms after the n = 5 are not really zero but very small numbers which are being evaluated to zero because the variable defined is a single precision floating point variable.

So the question is how does one know when to stop adding more and more terms? Or what is the same thing, how do we check for the desired level of accuracy? Clearly, what we see from the example above is that if the relative value of the term to be added to a partial sum is very small compared to the partial sum itself, then it will not change the partial sum significantly. Thus the quantity that one would want to evaluate and see if it is small enough is:

$$\textbf{accuracy} = \left|\frac{t_n}{S_{n-1}}\right|$$

If this quantity is smaller than a predetermined value, then we can safely terminate the series.

```
/* Program for evaluating a infinite series */
#include <iostream>
#include <cmath>  // For math functions if needed
#include <fstream>
#define pi 3.14159

using namespace std;

int main() {
    float x, t, sum, acc=0.0001;
    int n, i;
    ofstream fp;
    fp.open("res.txt")


    x = pi/4.0;
    sum = x;
    t = x ;
    i = 1;
    /* The following loop evaluates the terms and sums them */
    do{  // i starts at 1; t=0 term is the initial value
        t *= (x*x) / ((2*i+1)*(2*i));
        sum += t;
        i +=1;
    }
    while(fabs(t/s) > acc);
    fp << i << '\t' << t << '\t' << sum << '\t' << sin(x) << endl;
    fp.close()
    return 0;
}
```

Problems

1. Write a program to evaluate the sum up to 20 terms of the series

$$1+\frac{1}{x^2}+\frac{1}{x^3}+\frac{1}{x^4}+\cdots$$

for a given $x(0 \le x \le 2)$, and compare your result with the analytic sum of the series.

2. Evaluate $\cos(x)$ using the series

$$\cos(x)=1-\frac{x^2}{2!}+\frac{x^4}{4!}-\cdots$$

accurate to four significant places. Plot $\cos(x)$ vs x in the range $0 \le x \le \pi$.

3. Write a program to evaluate $J_n(x)$ to an accuracy of four significant figures using the following series expansion:

$$J_n(x)=\left(\frac{x}{2}\right)^n \sum_{k=0}^{\infty} \frac{(-1)^k\left(\frac{x^2}{4}\right)^k}{k!(n+k)!}$$

Plot $J_n(x)$ against x for $0 \le x \le 10$ and $n = 0, 1, 2$. Compare with the known behaviour of these functions and explain the discrepancy at large x.

4. Evaluate $F(z)$ given by

$$F(z)=\cos\left(\frac{\pi z^2}{2}\right)\sum_{n=0}^{\infty}\frac{(-1)^n \pi^{2n}\, z^{4n+1}}{1\times 5\times 9\cdots(4n+1)}$$

correct to four significant figures, for $0 \le z \le 1$, at intervals of 0.1.

5. Write a program to plot the sum of the following series:

$$f(z,n)=\sum_{k=0,2,4}^{\infty}\frac{z^k}{2^{n-k}\; k!\; \Gamma\left(\frac{1}{2}+\frac{n-k}{2}\right)}$$

for $n = 2$ and z in the range $0 \le z \le 5$. You would require the following relations:

$$\Gamma\left(\frac{1}{2}\right)=\sqrt{\pi}$$

$$\Gamma(z+1)=z\Gamma(z)$$

6. Write a program to plot the following function:

$$f(z)=C\left(1+\frac{z^3}{3!}+\frac{1\times 4z^6}{6!}+\frac{1\times 4\times 7z^9}{9!}+\cdots\right)$$

where $C = 0.35503$, for z in the range $-10 \le z \le 0$, at intervals of 0.05.

Chapter 3

Matrix multiplication

3.1 C++ Code for Dot Product of Two 3x3 Matrices

The following C++ code computes the dot product of two 3x3 matrices:

```cpp
#include <iostream>
using namespace std;

// Function to calculate the dot product of two 3x3 matrices
void dotProduct(int matrix1[3][3], int matrix2[3][3], int result[3][3]) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            result[i][j] = 0; // Initialize result element to 0
            for (int k = 0; k < 3; k++) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}

int main() {
    int matrix1[3][3], matrix2[3][3], result[3][3];

    // Input first 3x3 matrix
    cout << "Enter the elements of the first 3x3 matrix:" << endl;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            cin >> matrix1[i][j];
        }
    }

    // Input second 3x3 matrix
    cout << "Enter the elements of the second 3x3 matrix:" << endl;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            cin >> matrix2[i][j];
        }
    }

    // Call the dotProduct function
    dotProduct(matrix1, matrix2, result);

    // Display the result
    cout << "Dot product of the two matrices is:" << endl;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            cout << result[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

The above code defines the function `dotProduct()` that computes the dot product of two 3x3 matrices. It uses nested loops to multiply the matrices and store the result.

3.2 Explanation of `void` in C++

In C++, the keyword `void` is used in two main contexts:

3.2.1 As a Return Type for Functions

When `void` is used as a function's return type, it indicates that the function does not return any value. The function executes its operations and exits without giving back any result to the caller.

For example:

```
void sayHello() {
    cout << "Hello, world!" << endl;
}
```

In this case, the function `sayHello()` performs an action (printing "Hello, world!") but does not return anything, so its return type is `void`.

In the context of the matrix dot product code:

```
void dotProduct(int matrix1[3][3], int matrix2[3][3], int result[3][3]) {
    // Code to calculate the dot product
}
```

Here, the `dotProduct()` function performs matrix multiplication and stores the result in the `result` array, but it does not return anything directly. Thus, its return type is `void`.

3.2.2 As an Empty Argument List

In C++, when `void` is used in the parameter list of a function, it indicates that the function takes no arguments. For example:

```
void functionName(void) {
    // Code
}
```

3.3 Fortran Code for Dot Product of a 3x3 Matrix

The following code calculates the dot product of two 3x3 matrices in Fortran:

```fortran
program matrix_dot_product
    implicit none
    integer, parameter :: n = 3
    real :: A(n, n), B(n, n), result(n, n)
    integer :: i, j, k

    ! Initialize matrices A and B
    A = reshape([1.0, 2.0, 3.0, &
                 4.0, 5.0, 6.0, &
                 7.0, 8.0, 9.0], [n, n])

    B = reshape([9.0, 8.0, 7.0, &
                 6.0, 5.0, 4.0, &
                 3.0, 2.0, 1.0], [n, n])

    ! Initialize the result matrix to zero
    result = 0.0

    ! Perform dot product
    do i = 1, n
        do j = 1, n
            do k = 1, n
                result(i, j) = result(i, j) + A(i, k) * B(k, j)
            end do
        end do
    end do

    ! Print the result matrix
    print *, 'Result matrix:'
    do i = 1, n
        print *, result(i, :)
    end do

end program matrix_dot_product
```

This code defines two 3x3 matrices `A` and `B`, performs the dot product, and stores the result in the matrix `result`. The final result is printed row by row.

Chapter 4

Prime numbers and Fibonacci Series

4.1 C++ Code for Finding a Set of Prime Numbers

The following C++ code finds and prints prime numbers up to a specified limit using the Sieve of Eratosthenes algorithm:

```cpp
#include <iostream>
#include <vector>
using namespace std;

void findPrimes(int limit) {
    vector<bool> isPrime(limit + 1, true);
    isPrime[0] = isPrime[1] = false;

    for (int p = 2; p * p <= limit; ++p) {
        if (isPrime[p]) {
            for (int i = p * p; i <= limit; i += p) {
                isPrime[i] = false;
            }
        }
    }

    // Print all prime numbers
    cout << "Prime numbers up to " << limit << " are: \n";
    for (int p = 2; p <= limit; ++p) {
        if (isPrime[p]) {
            cout << p << " ";
        }
    }
    cout << endl;
}

int main() {
    int limit;
    cout << "Enter the upper limit: ";
    cin >> limit;
    findPrimes(limit);
    return 0;
}
```

This code defines a function `findPrimes` that uses a boolean vector to mark non-prime numbers. It prints all prime numbers up to the user-specified limit. The Sieve of Eratosthenes algorithm efficiently identifies the prime numbers by iterating over multiples of known primes.

4.2 C++ Code for Printing the Fibonacci Series

The following C++ code generates and prints the Fibonacci series up to a specified number of terms:

```cpp
#include <iostream>
using namespace std;

void printFibonacci(int terms) {
    int first = 0, second = 1, next;

    cout << "Fibonacci Series: ";
    for (int i = 0; i < terms; i++) {
        if (i <= 1) {
            next = i;  // First two terms are 0 and 1
        } else {
            next = first + second;  // Next term is the sum of the previous two
            first = second;  // Update first
            second = next;   // Update second
        }
        cout << next << " ";  // Print the current term
    }
    cout << endl;
}

int main() {
    int terms;
    cout << "Enter the number of terms: ";
    cin >> terms;
    printFibonacci(terms);
    return 0;
}
```

This code defines a function `printFibonacci` that calculates and displays the Fibonacci series. It uses a loop to compute each term based on the previous two terms, starting with 0 and 1. The user specifies how many terms of the series to print.