Introduction to Computational Physics Lab

UPES Dehradun

Introduction to Computational Physics Lab - 2024

Nitesh Kumar

November 7, 2024

© 2024 Nitesh Kumar. All rights reserved.

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

5 Finding the Roots of a Quadratic Equation 23
5.1 Theory . . . 23
5.2 Concepts . . . 23
5.3 Formulas . . . 23
5.4 C++ Code . . . 23

6 Bisection and Newton-Raphson Method 25
6.1 Bisection Method . . . 25
6.1.1 Steps of the Bisection Method . . . 25
6.1.2 Disadvantages of the Bisection Method . . . 26
6.1.3 Secant Method . . . 28
6.1.4 Newton-Raphson Method . . . 30
6.2 C++ Code . . . 31
6.3 GNUPlot Script . . . 32

7 Motion of a Projectile 33
7.1 Theory . . . 33
7.2 Concepts . . . 33
7.3 C++ Code . . . 34
7.4 GNUPlot Script . . . 34
7.5 Conclusion . . . 35

8 Motion of a Simple Harmonic Oscillator **37**
8.1 Theory . . . 37
8.2 C++ Code . . . 37
8.3 GNUPlot Script . . . 38
8.4 Conclusion . . . 38

9 Motion of a Particle in a Central Force Field **39**
9.1 Theory . . . 39
9.2 C++ Code . . . 39
9.3 GNUPlot Script . . . 40
9.4 Conclusion . . . 41

10 Approximation of Functions **43**
10.1 Theory . . . 43
10.1.1 Lagrange Interpolation Formula . . . 43
10.1.2 Newton's Divided Difference Formula . . . 43
10.2 C++ Code . . . 44
10.3 GNUPlot Script . . . 45
10.4 Conclusion . . . 46

11 Numerical Integration **47**
11.1 Theory . . . 47
11.1.1 Trapezoidal Rule . . . 47
11.1.2 Simpson's Rule . . . 47
11.2 C++ Code . . . 48
11.3 GNUPlot Script . . . 49
11.4 Conclusion . . . 49

12 Ordinary Differential Equations **51**
12.1 Theory . . . 51
12.1.1 Euler's Method . . . 51
12.1.2 Runge-Kutta Method (RK4) . . . 51
12.2 C++ Code . . . 52
12.3 GNUPlot Script . . . 53
12.4 Conclusion . . . 53

© 2024 Nitesh Kumar. All rights reserved.

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.

© 2024 Nitesh Kumar. All rights reserved.

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`.

© 2024 Nitesh Kumar. All rights reserved.

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.

© 2024 Nitesh Kumar. All rights reserved.

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 ith 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:

```
1 ! Program for evaluating a finite series
2 PROGRAM finite_series
3     IMPLICIT NONE
4     REAL :: x, t, s
5     INTEGER :: n, i
6
7     PRINT *, 'Supply x and the number of terms n:'
```

© 2024 Nitesh Kumar. All rights reserved.

```fortran
    ! 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++:

```cpp
/* 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?

© 2024 Nitesh Kumar. All rights reserved.

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,

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

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

(2.20)

© 2024 Nitesh Kumar. All rights reserved.

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.

```
/* 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

© 2024 Nitesh Kumar. All rights reserved.

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;
}
```

© 2024 Nitesh Kumar. All rights reserved.

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 $x (0 \leq x \leq 2)$, 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 $0 \leq x \leq \pi$.

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 $0 \leq x \leq 10$ 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 $0 \leq z \leq 1$, 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 $0 \leq z \leq 5$. 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:

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

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

© 2024 Nitesh Kumar. All rights reserved.

© 2024 Nitesh Kumar. All rights reserved.

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:

```
#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
```

```
39     cout << "Dot product of the two matrices is:" << endl;
40     for (int i = 0; i < 3; i++) {
41         for (int j = 0; j < 3; j++) {
42             cout << result[i][j] << " ";
43         }
44         cout << endl;
45     }
46 
47     return 0;
48 }
```

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
}
```

© 2024 Nitesh Kumar. All rights reserved.

3.3 Fortran Code for Dot Product of a 3x3 Matrix

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

© 2024 Nitesh Kumar. All rights reserved.

© 2024 Nitesh Kumar. All rights reserved.

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:

```
#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:

```
#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.

© 2024 Nitesh Kumar. All rights reserved.

Chapter 5

Experiment No. 09: Finding the Roots of a Quadratic Equation

5.1 Theory

A quadratic equation is a second-order polynomial equation in a single variable with the form:

$$ax^2 + bx + c = 0$$

where a, b, and c are constants. The roots of a quadratic equation can be found using the quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

These roots may be real or complex depending on the discriminant $\Delta = b^2 - 4ac$.

5.2 Concepts

The roots of the quadratic equation are derived from the discriminant:

- If $\Delta > 0$, the equation has two distinct real roots.
- If $\Delta = 0$, the equation has exactly one real root (a repeated root).
- If $\Delta < 0$, the equation has two complex roots.

5.3 Formulas

$$x_1 = \frac{-b + \sqrt{\Delta}}{2a}, \quad x_2 = \frac{-b - \sqrt{\Delta}}{2a}$$

where $\Delta = b^2 - 4ac$ is the discriminant.

5.4 C++ Code

The following C++ program solves the quadratic equation using the quadratic formula:

```cpp
// C++ program to find roots of a quadratic equation
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    double a, b, c, discriminant, root1, root2;

    // Input coefficients
    cout << "Enter coefficients a, b, and c: ";
    cin >> a >> b >> c;

    discriminant = b*b - 4*a*c;

    if (discriminant > 0) {
        // Two real and distinct roots
        root1 = (-b + sqrt(discriminant)) / (2*a);
        root2 = (-b - sqrt(discriminant)) / (2*a);
        cout << "Roots are real and different." << endl;
        cout << "Root 1 = " << root1 << endl;
        cout << "Root 2 = " << root2 << endl;
    } else if (discriminant == 0) {
        // One real root
        root1 = -b / (2*a);
        cout << "Root is real and repeated." << endl;
        cout << "Root = " << root1 << endl;
    } else {
        // Complex roots
        double realPart = -b / (2*a);
        double imaginaryPart = sqrt(-discriminant) / (2*a);
        cout << "Roots are complex and different." << endl;
        cout << "Root 1 = " << realPart << " + " << imaginaryPart << "i" <<
    endl;
        cout << "Root 2 = " << realPart << " - " << imaginaryPart << "i" <<
    endl;
    }

    return 0;
}
```

This experiment demonstrates how to calculate the roots of a quadratic equation by analyzing the discriminant and solving for both real and complex roots using the quadratic formula.

© 2024 Nitesh Kumar. All rights reserved.

Chapter 6

Finding the Roots of a Polynomial Equation Using Bisection and Other Methods

A polynomial equation of the form:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = 0$$

has real or complex roots depending on the coefficients and degree of the polynomial. Numerical methods such as the Bisection method and the Newton-Raphson method are commonly used to find real roots when an analytical solution is difficult to obtain.

6.1 Bisection Method

The Bisection method is a simple and robust numerical technique to find roots of a continuous function $f(x)$ on an interval $[a, b]$ where $f(a)$ and $f(b)$ have opposite signs (i.e., $f(a)f(b) < 0$). - The method repeatedly bisects the interval and selects the subinterval in which the sign of the function changes. - The root is approximated as the midpoint of the interval when the interval becomes sufficiently small.

The iterative formula for Bisection is:

$$x_{mid} = \frac{a+b}{2}$$

If $f(x_{mid}) = 0$, then x_{mid} is the root; otherwise, continue with the subinterval where the function changes its sign.

6.1.1 Steps of the Bisection Method

The Bisection method follows these steps:

1. **Choose the initial interval** $[a, b]$ such that $f(a)$ and $f(b)$ have opposite signs, meaning $f(a)f(b) < 0$.

2. **Compute the midpoint** of the interval:

$$x_{mid} = \frac{a+b}{2}$$

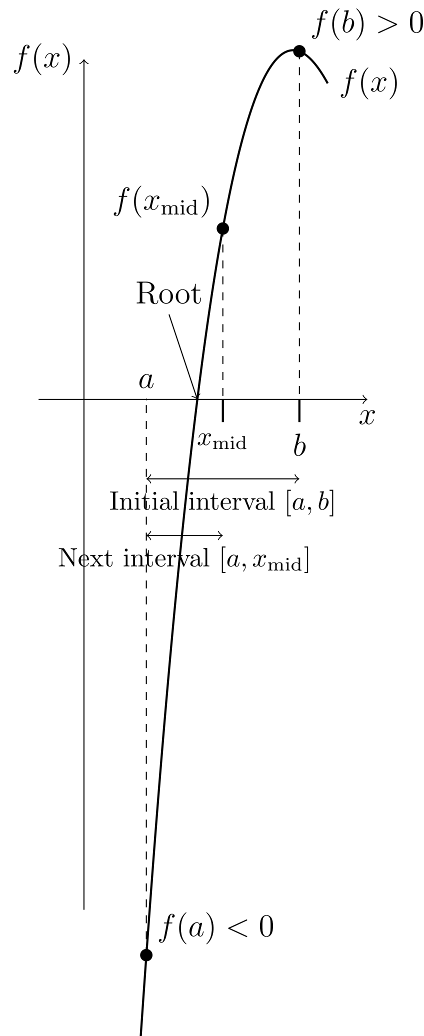

Figure 6.1: Illustration of the Bisection method using the function $f(x) = x^3 - 9x^2 + 23x - 15$. The method starts with the interval $[a, b]$, then iteratively bisects the interval to find the root by checking sign changes in $f(x)$.

3. **Evaluate the function** at the midpoint $f(x_{mid})$.

4. **Check if the root has been found:**
   - If $f(x_{mid}) = 0$, then x_{mid} is the root.
   - If $f(x_{mid}) \neq 0$, check the sign of $f(x_{mid})$ and proceed as follows:
     - If $f(a)f(x_{mid}) < 0$, then the root lies in the subinterval $[a, x_{mid}]$. Set $b = x_{mid}$.
     - If $f(b)f(x_{mid}) < 0$, then the root lies in the subinterval $[x_{mid}, b]$. Set $a = x_{mid}$.

5. **Repeat the process** until the interval becomes sufficiently small, i.e., $|b - a|$ is less than a pre-specified tolerance level.

6.1.2 Disadvantages of the Bisection Method

While the Bisection method is reliable and simple, it has several limitations:

© 2024 Nitesh Kumar. All rights reserved.

- **Slow Convergence:** The Bisection method converges linearly, which makes it slower compared to other methods such as Newton-Raphson that converge quadratically. This can be a disadvantage when higher accuracy is needed in fewer iterations.
- **Requires an Interval with Opposite Signs:** The method requires the initial interval $[a, b]$ to satisfy $f(a)f(b) < 0$, meaning that a sign change between $f(a)$ and $f(b)$ is essential. If no such interval is known, the method cannot be applied.
- **Not Suitable for Multiple or Complex Roots:** The Bisection method can only find one real root within an interval, and it does not work for complex roots or multiple roots within the same interval unless the function is redefined or additional methods are employed.
- **Cannot Handle Discontinuous Functions:** The method assumes the function is continuous over the interval $[a, b]$. If the function has discontinuities, the Bisection method might fail or produce incorrect results.

```
1  // C++ program to find the root of a polynomial using Bisection and Newton-
       Raphson methods
2  #include <iostream>
3  #include <cmath>
4  #include <fstream>
5  using namespace std;
6
7  // Define the polynomial function f(x) = x^3 - x - 2
8  double f(double x) {
9      return x*x*x - x - 2;
10 }
11
12 // Bisection method to find root
13 double BisectionMethod(double a, double b, double tol, ofstream &outfile) {
14     double mid;
15     int iterations = 0;
16     outfile << "# Iteration\tBisection_Root" << endl;
17     while ((b - a) >= tol) {
18         mid = (a + b) / 2.0;
19         outfile << iterations << "\t" << mid << endl;
20         if (f(mid) == 0.0)  // Exact root found
21             break;
22         else if (f(mid) * f(a) < 0)
23             b = mid;
24         else
25             a = mid;
26         iterations++;
27     }
28     return mid;
29 }
30
31 int main() {
32     double a, b, x0, tol;
33
34     // Open file to store the output
35     ofstream outfile("roots_output.txt");
36
37     // Input interval for Bisection
38     cout << "Enter the interval [a, b] for Bisection method: ";
```

© 2024 Nitesh Kumar. All rights reserved.

```
    cin >> a >> b;

    // Input initial guess for Newton-Raphson
    cout << "Enter the initial guess for Newton-Raphson method: ";
    cin >> x0;

    // Input tolerance level
    cout << "Enter the tolerance level: ";
    cin >> tol;

    // Finding root using Bisection Method
    double bisection_root = BisectionMethod(a, b, tol, outfile);

    return 0;
}
```

6.1.3 Secant Method

The Secant method is a numerical technique used to find the root of a function $f(x)$ by using a secant line to approximate the function near the root. Unlike the Bisection method, the two initial points for the Secant method do not need to lie on opposite sides of the root, but they must be sufficiently close to it. However, choosing points on opposite sides of the root often improves the stability of the method.

The Secant method uses two initial points, x_1 and x_2, and approximates the function by a straight line passing through these two points. The root is then estimated as the x-intercept of this secant line. The equation of the secant line passing through the points $(x_1, f(x_1))$ and $(x_2, f(x_2))$ is given by:

$$y - f(x_2) = \frac{f(x_2) - f(x_1)}{x_2 - x_1}(x - x_2)$$

Setting $y = 0$ to find the x-intercept (the approximation of the root), we get:

$$0 - f(x_2) = \frac{f(x_2) - f(x_1)}{x_2 - x_1}(x_3 - x_2)$$

Solving for x_3, the next approximation of the root is:

$$x_3 = x_2 - f(x_2)\frac{x_2 - x_1}{f(x_2) - f(x_1)}$$

This formula is iterated with the newly found point x_3 replacing x_1, and x_2 replacing x_3 in subsequent steps. The process is repeated until the values of x_n converge to a root with the desired level of accuracy.

Steps of the Secant Method

1. Choose two initial points x_1 and x_2 close to the expected root.
2. Evaluate $f(x_1)$ and $f(x_2)$.
3. Compute the next approximation of the root using the formula:

$$x_3 = x_2 - f(x_2)\frac{x_2 - x_1}{f(x_2) - f(x_1)}$$

© 2024 Nitesh Kumar. All rights reserved.

4. Replace x_1 with x_2 and x_2 with x_3, then repeat the process until $|x_{n+1} - x_n|$ is less than a specified tolerance.

5. Stop when the root is approximated to the desired level of accuracy.

Advantages and Disadvantages of the Secant Method

- **Advantages:**
  - The Secant method often converges faster than the Bisection method.
  - It does not require the calculation of the derivative of the function, unlike the Newton-Raphson method.
- **Disadvantages:**
  - The Secant method may fail to converge if the initial points are not close to the root or if the function behaves poorly in the region.
  - It has a lower order of convergence compared to Newton-Raphson (superlinear vs. quadratic).
  - It is less reliable than the Bisection method because it does not guarantee convergence if the initial points are not well chosen.

C++ Code

```
#include <iostream>
#include <cmath>
#include <fstream>
using namespace std;

// Define the function whose root we want to find, e.g., f(x) = x^3 - x - 2
double f(double x) {
    return x * x * x - x - 2;
}

// Secant method implementation
double SecantMethod(double x0, double x1, double tol, ofstream &outfile) {
    double x2, f_x0, f_x1, diff;
    int iterations = 0;

    outfile << "# Iteration\tSecant_Root_Estimate" << endl;

    do {
        f_x0 = f(x0);  // f(x0)
        f_x1 = f(x1);  // f(x1)

        if (fabs(f_x1 - f_x0) < tol) {  // Check for division by zero or near zero difference
            cout << "Error: Division by zero or very small difference between function values." << endl;
            return NAN;
        }

        // Compute next approximation using the secant formula
        x2 = x1 - f_x1 * (x1 - x0) / (f_x1 - f_x0);
```

© 2024 Nitesh Kumar. All rights reserved.

```
29          diff = fabs(x2 - x1);  // Difference between current and next
        approximation
30
31          outfile << iterations << "\t" << x2 << endl;
32
33          // Update x0 and x1 for the next iteration
34          x0 = x1;
35          x1 = x2;
36          iterations++;
37
38      } while (diff >= tol);  // Continue until the difference is less than
        the tolerance
39
40      return x2;  // The root estimate
41  }
42
43  int main() {
44      double x0, x1, tol;
45
46      // Open file to store the output
47      ofstream outfile("roots_output_secant.txt");
48
49      // Input initial guesses for Secant Method
50      cout << "Enter the first initial guess: ";
51      cin >> x0;
52      cout << "Enter the second initial guess: ";
53      cin >> x1;
54
55      // Input tolerance level
56      cout << "Enter the tolerance level: ";
57      cin >> tol;
58
59      // Finding root using Secant Method
60      double root = SecantMethod(x0, x1, tol, outfile);
61
62      outfile.close();  // Close the output file
63
64      if (!isnan(root)) {
65          cout << "Root found: " << root << endl;
66          cout << "Results written to roots_output_secant.txt for
        visualization in GNUPlot." << endl;
67      } else {
68          cout << "Failed to find a root due to numerical issues." << endl;
69      }
70
71      return 0;
72  }
```

6.1.4 Newton-Raphson Method

This method takes advantage of the Taylor's series expansion of a function. The Newton-Raphson method is an efficient root-finding algorithm that requires the function and its derivative. Starting from an initial guess x_0, the next approximation is given by:

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

© 2024 Nitesh Kumar. All rights reserved.

The method converges quadratically if the initial guess is sufficiently close to the root.

6.2 C++ Code

The following C++ program implements both the Bisection method and the Newton-Raphson method to find the roots of a given polynomial function. The results are written to a file for visualization in GNUPlot.

```
// C++ program to find the root of a polynomial using Bisection and Newton-Raphson methods
#include <iostream>
#include <cmath>
#include <fstream>
using namespace std;

// Define the polynomial function f(x) = x^3 - x - 2
double f(double x) {
    return x*x*x - x - 2;
}

// Define the derivative of the polynomial f'(x) = 3x^2 - 1
double df(double x) {
    return 3*x*x - 1;
}

// Bisection method to find root
double BisectionMethod(double a, double b, double tol, ofstream &outfile) {
    double mid;
    int iterations = 0;
    outfile << "# Iteration\tBisection_Root" << endl;
    while ((b - a) >= tol) {
        mid = (a + b) / 2.0;
        outfile << iterations << "\t" << mid << endl;
        if (f(mid) == 0.0)  // Exact root found
            break;
        else if (f(mid) * f(a) < 0)
            b = mid;
        else
            a = mid;
        iterations++;
    }
    return mid;
}

// Newton-Raphson method to find root
double NewtonRaphsonMethod(double x0, double tol, ofstream &outfile) {
    double x = x0, h;
    int iterations = 0;
    outfile << "# Iteration\tNewton_Raphson_Root" << endl;
    while (fabs(f(x)) >= tol) {
        h = f(x) / df(x);
        outfile << iterations << "\t" << x << endl;
        x = x - h;
        iterations++;
    }
    return x;
}
```

© 2024 Nitesh Kumar. All rights reserved.

```cpp

int main() {
    double a, b, x0, tol;

    // Open file to store the output
    ofstream outfile("roots_output.txt");

    // Input interval for Bisection
    cout << "Enter the interval [a, b] for Bisection method: ";
    cin >> a >> b;

    // Input initial guess for Newton-Raphson
    cout << "Enter the initial guess for Newton-Raphson method: ";
    cin >> x0;

    // Input tolerance level
    cout << "Enter the tolerance level: ";
    cin >> tol;

    // Finding root using Bisection Method
    double bisection_root = BisectionMethod(a, b, tol, outfile);

    // Finding root using Newton-Raphson Method
    double newton_raphson_root = NewtonRaphsonMethod(x0, tol, outfile);

    outfile.close();
    cout << "Roots written to roots_output.txt for visualization in GNUPlot." << endl;

    return 0;
}
```

6.3 GNUPlot Script

The following GNUPlot script reads the data from the file generated by the C++ program and plots the convergence of the root-finding methods.

```
# roots_plot.gp - GNUPlot script to plot roots found using Bisection and Newton-Raphson methods

set title "Root Finding using Bisection and Newton-Raphson Methods"
set xlabel "Iterations"
set ylabel "Root Estimate"
set grid
set key outside

# Plot the roots convergence
plot "roots_output.txt" using 1:2 with lines title "Bisection Method", \
     "roots_output.txt" using 1:3 with lines title "Newton-Raphson Method"
```

In this experiment, we implemented two numerical methods, Bisection and Newton-Raphson, to find the roots of a polynomial equation. The Bisection method is simple but slower, while the Newton-Raphson method converges faster when a good initial guess is provided. The solutions were visualized using GNUPlot to compare the convergence of both methods.

© 2024 Nitesh Kumar. All rights reserved.

Chapter 7

Experiment No. 10: Motion of a Projectile

7.1 Theory

Projectile motion is a form of motion in which an object is thrown near the Earth's surface, and it moves along a curved path under the action of gravity. The horizontal and vertical components of the motion are independent of each other. The horizontal motion is uniform (constant velocity), and the vertical motion is uniformly accelerated (constant acceleration due to gravity).

7.2 Concepts

The equations of motion for the projectile are:

- Horizontal displacement:

$$x = v_0 \cos(\theta) t$$

- Vertical displacement:

$$y = v_0 \sin(\theta) t - \frac{1}{2} g t^2$$

- Time of flight:

$$T = \frac{2 v_0 \sin(\theta)}{g}$$

- Maximum height:

$$H = \frac{v_0^2 \sin^2(\theta)}{2g}$$

- Range:

$$R = \frac{v_0^2 \sin(2\theta)}{g}$$

where v_0 is the initial velocity, θ is the angle of projection, g is the acceleration due to gravity, and t is the time.

7.3 C++ Code

The following C++ program simulates the motion of a projectile and stores the output into a file that can be visualized using GNUPlot:

```
// C++ program to simulate the motion of a projectile and write output to a file
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;

int main() {
    double v0, theta, g = 9.81, t, x, y;

    // Open file to store the output
    ofstream outfile("projectile_data.txt");

    // Input initial velocity and angle
    cout << "Enter initial velocity (m/s): ";
    cin >> v0;
    cout << "Enter angle of projection (degrees): ";
    cin >> theta;

    // Convert angle to radians
    theta = theta * M_PI / 180.0;

    // Calculate time of flight, maximum height, and range
    double T = (2 * v0 * sin(theta)) / g;
    double H = (v0 * v0 * sin(theta) * sin(theta)) / (2 * g);
    double R = (v0 * v0 * sin(2 * theta)) / g;

    // Write header for the file
    outfile << "# Time (s)\tX-Position (m)\tY-Position (m)" << endl;

    // Simulate the motion and write to file
    for (t = 0; t <= T; t += 0.1) {
        x = v0 * cos(theta) * t;
        y = v0 * sin(theta) * t - 0.5 * g * t * t;
        if (y < 0) y = 0; // Ensure y doesn't go below ground level
        outfile << t << "\t" << x << "\t\t" << y << endl;
    }

    outfile.close();
    cout << "Data written to projectile_data.txt for visualization in GNUPlot." << endl;

    return 0;
}
```

7.4 GNUPlot Script

The following GNUPlot script reads the data from the file generated by the C++ program and plots the trajectory of the projectile:

```
# projectile_plot.gp - GNUPlot script to plot projectile motion

```

© 2024 Nitesh Kumar. All rights reserved.

```
set title "Projectile Motion"
set xlabel "X-Position (m)"
set ylabel "Y-Position (m)"
set grid
set key off

# Plot the data from the file
plot "projectile_data.txt" using 2:3 with linespoints title "Projectile
    Trajectory"
```

7.5 Conclusion

This experiment simulated the motion of a projectile under the influence of gravity. The trajectory was visualized using GNUPlot by plotting the horizontal and vertical positions.

© 2024 Nitesh Kumar. All rights reserved.

© 2024 Nitesh Kumar. All rights reserved.

Chapter 8

Experiment No. 11: Motion of a Simple Harmonic Oscillator

8.1 Theory

A simple harmonic oscillator (SHO) experiences a restoring force proportional to its displacement from its equilibrium position. The motion of the SHO can be described by the second-order differential equation:

$$\frac{d^2x}{dt^2} + \omega^2 x = 0$$

where $x(t)$ is the displacement, and $\omega = \sqrt{\frac{k}{m}}$ is the angular frequency.

The general solution to this equation is:

$$x(t) = A\cos(\omega t + \phi)$$

where A is the amplitude and ϕ is the phase angle.

8.2 C++ Code

The following C++ program simulates the motion of a simple harmonic oscillator and stores the output into a file for visualization in GNUPlot:

```
// C++ program to simulate the motion of a simple harmonic oscillator and
    write output to a file
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;

int main() {
    double A, omega, t, x, v, a, T;

    // Open file to store the output
    ofstream outfile("sho_data.txt");

    // Input amplitude and angular frequency
    cout << "Enter amplitude (meters): ";
    cin >> A;
```

```
16     cout << "Enter angular frequency (rad/s): ";
17     cin >> omega;
18 
19     // Calculate period of oscillation
20     T = 2 * M_PI / omega;
21     cout << "Period of oscillation: " << T << " seconds" << endl;
22 
23     // Write header for the file
24     outfile << "# Time (s)\tDisplacement (m)\tVelocity (m/s)\tAcceleration
       (m/s^2)" << endl;
25 
26     // Simulate the motion and write to file
27     for (t = 0; t <= 2 * T; t += 0.1) {
28         x = A * cos(omega * t);
29         v = -A * omega * sin(omega * t);
30         a = -A * omega * omega * cos(omega * t);
31         outfile << t << "\t" << x << "\t\t" << v << "\t\t" << a << endl;
32     }
33 
34     outfile.close();
35     cout << "Data written to sho_data.txt for visualization in GNUPlot." <<
       endl;
36 
37     return 0;
38 }
```

8.3 GNUPlot Script

The following GNUPlot script reads the data from the file generated by the C++ program and plots the displacement, velocity, and acceleration of the simple harmonic oscillator:

```
1 # sho_plot.gp - GNUPlot script to plot the motion of a simple harmonic
    oscillator
2 
3 set title "Simple Harmonic Oscillator Motion"
4 set xlabel "Time (s)"
5 set ylabel "Displacement / Velocity / Acceleration"
6 set grid
7 set key outside
8 
9 # Plot displacement, velocity, and acceleration from the file
10 plot "sho_data.txt" using 1:2 with lines title "Displacement (m)", \
11      "sho_data.txt" using 1:3 with lines title "Velocity (m/s)", \
12      "sho_data.txt" using 1:4 with lines title "Acceleration (m/s^2)"
```

8.4 Conclusion

The numerical simulation of a simple harmonic oscillator was performed, and the displacement, velocity, and acceleration were visualized using GNUPlot.

© 2024 Nitesh Kumar. All rights reserved.

Chapter 9

Experiment No. 12: Motion of a Particle in a Central Force Field

9.1 Theory

A central force field is one where the force acting on a particle is always directed towards a fixed point and depends only on the distance of the particle from that point. Such forces include gravitational and electrostatic forces. The motion of a particle under a central force can be described using polar coordinates.

The equation of motion for a particle under a central force $F(r)$ is given by:

$$\frac{d^2r}{dt^2} - r\left(\frac{d\theta}{dt}\right)^2 = \frac{F(r)}{m}$$

where $r(t)$ is the radial distance, $\theta(t)$ is the angular displacement, m is the mass of the particle, and $F(r)$ is the central force.

For this experiment, we will assume the particle is under an inverse square law force, such as gravity:

$$F(r) = -\frac{GMm}{r^2}$$

where G is the gravitational constant and M is the mass of the central body.

9.2 C++ Code

The following C++ program simulates the motion of a particle in a central force field and stores the output into a file for visualization in GNUPlot:

```
// C++ program to simulate the motion of a particle in a central force field and write output to a file
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;

int main() {
    // Constants
    const double G = 6.67430e-11; // Gravitational constant (m^3/kg/s^2)
    double M, m, r, theta, t, v_r, v_theta, a_r, a_theta, delta_t;

```

```
12      // Open file to store the output
13      ofstream outfile("central_force_data.txt");
14
15      // Input parameters
16      cout << "Enter mass of central body (kg): ";
17      cin >> M;
18      cout << "Enter mass of particle (kg): ";
19      cin >> m;
20      cout << "Enter initial radial distance (m): ";
21      cin >> r;
22      cout << "Enter initial angular displacement (radians): ";
23      cin >> theta;
24      cout << "Enter initial radial velocity (m/s): ";
25      cin >> v_r;
26      cout << "Enter initial angular velocity (rad/s): ";
27      cin >> v_theta;
28      cout << "Enter time step for simulation (s): ";
29      cin >> delta_t;
30
31      // Write header for the file
32      outfile << "# Time (s)\tRadial Distance (m)\tAngular Displacement (rad)
    " << endl;
33
34      // Time integration loop (Euler method)
35      for (t = 0; t <= 1000; t += delta_t) {
36          // Calculate accelerations
37          double F_r = -G * M * m / (r * r);  // Gravitational force
38          a_r = F_r / m + r * v_theta * v_theta;  // Radial acceleration
39          a_theta = -2 * v_r * v_theta / r;  // Angular acceleration
40
41          // Update velocities
42          v_r += a_r * delta_t;
43          v_theta += a_theta * delta_t;
44
45          // Update position
46          r += v_r * delta_t;
47          theta += v_theta * delta_t;
48
49          // Write to file
50          outfile << t << "\t" << r << "\t\t" << theta << endl;
51
52          // Stop simulation if particle crashes into the central body
53          if (r <= 0) break;
54      }
55
56      outfile.close();
57      cout << "Data written to central_force_data.txt for visualization in
    GNUPlot." << endl;
58
59      return 0;
60  }
```

9.3 GNUPlot Script

The following GNUPlot script reads the data from the file generated by the C++ program and plots the radial distance as a function of time and the angular displacement.

© 2024 Nitesh Kumar. All rights reserved.

```
1 # central_force_plot.gp - GNUPlot script to plot the motion of a particle
    in a central force field
2
3 set title "Motion of a Particle in a Central Force Field"
4 set xlabel "Time (s)"
5 set ylabel "Radial Distance (m)"
6 set grid
7 set key outside
8
9 # First, plot radial distance as a function of time
10 set multiplot layout 2,1 title "Particle Motion in Central Force Field"
11
12 plot "central_force_data.txt" using 1:2 with lines title "Radial Distance
    vs Time"
13
14 # Now, plot angular displacement as a function of time
15 set xlabel "Time (s)"
16 set ylabel "Angular Displacement (rad)"
17
18 plot "central_force_data.txt" using 1:3 with lines title "Angular
    Displacement vs Time"
19
20 unset multiplot
```

9.4 Conclusion

This experiment simulated the motion of a particle under the influence of a central force, such as gravity. The simulation outputs the radial and angular positions of the particle over time, which are visualized using GNUPlot.

© 2024 Nitesh Kumar. All rights reserved.

© 2024 Nitesh Kumar. All rights reserved.

Chapter 10

Experiment No. 13: Approximation of Functions Using Lagrange and Newton's Divided Difference Schemes

10.1 Theory

Polynomial interpolation is a method of estimating values between known data points. Two common methods of polynomial interpolation are the Lagrange interpolation and Newton's divided difference interpolation.

10.1.1 Lagrange Interpolation Formula

Given $n + 1$ data points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$, the Lagrange interpolation polynomial is given by:

$$L(x) = \sum_{i=0}^{n} y_i \ell_i(x)$$

where $\ell_i(x)$ is the Lagrange basis polynomial defined as:

$$\ell_i(x) = \prod_{\substack{0 \le j \le n \\ j \ne i}} \frac{x - x_j}{x_i - x_j}$$

10.1.2 Newton's Divided Difference Formula

The Newton's divided difference formula uses divided differences to compute the interpolation polynomial incrementally. The interpolation polynomial is given by:

$$P(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + \ldots$$

where $f[x_0], f[x_0, x_1], \ldots$ are divided differences, calculated as:

$$f[x_i, x_{i+1}, \ldots, x_{i+k}] = \frac{f[x_{i+1}, \ldots, x_{i+k}] - f[x_i, \ldots, x_{i+k-1}]}{x_{i+k} - x_i}$$

10.2 C++ Code

The following C++ code implements both Lagrange interpolation and Newton's divided difference interpolation and writes the approximations to a file for visualization using GNUPlot.

```
// C++ program to perform Lagrange and Newton's divided difference interpolation
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

// Function for Lagrange interpolation
double LagrangeInterpolation(double x, vector<double>& X, vector<double>& Y, int n) {
    double result = 0.0;
    for (int i = 0; i < n; i++) {
        double term = Y[i];
        for (int j = 0; j < n; j++) {
            if (j != i) {
                term = term * (x - X[j]) / (X[i] - X[j]);
            }
        }
        result += term;
    }
    return result;
}

// Function for Newton's divided difference interpolation
double NewtonInterpolation(double x, vector<double>& X, vector<vector<double>>& F, int n) {
    double result = F[0][0];
    double product = 1.0;
    for (int i = 1; i < n; i++) {
        product *= (x - X[i-1]);
        result += product * F[0][i];
    }
    return result;
}

// Function to compute divided differences table for Newton's interpolation
void DividedDifferences(vector<double>& X, vector<double>& Y, vector<vector<double>>& F, int n) {
    for (int i = 0; i < n; i++) {
        F[i][0] = Y[i];
    }
    for (int j = 1; j < n; j++) {
        for (int i = 0; i < n - j; i++) {
            F[i][j] = (F[i+1][j-1] - F[i][j-1]) / (X[i+j] - X[i]);
        }
    }
}

int main() {
    int n;
    double x;
    vector<double> X, Y;
```

© 2024 Nitesh Kumar. All rights reserved.

```cpp

    // Open file to store the output
    ofstream outfile("interpolation_data.txt");

    // Input number of data points
    cout << "Enter number of data points: ";
    cin >> n;

    X.resize(n);
    Y.resize(n);

    // Input data points
    cout << "Enter data points (x y):" << endl;
    for (int i = 0; i < n; i++) {
        cin >> X[i] >> Y[i];
    }

    // Create divided differences table for Newton's method
    vector<vector<double>> F(n, vector<double>(n, 0.0));
    DividedDifferences(X, Y, F, n);

    // Write header to file
    outfile << "# X\tLagrange_Y\tNewton_Y" << endl;

    // Interpolate and store data for visualization
    for (x = X[0]; x <= X[n-1]; x += 0.1) {
        double L_y = LagrangeInterpolation(x, X, Y, n);
        double N_y = NewtonInterpolation(x, X, F, n);
        outfile << x << "\t" << L_y << "\t" << N_y << endl;
    }

    outfile.close();
    cout << "Data written to interpolation_data.txt for visualization in
    GNUPlot." << endl;

    return 0;
}
```

10.3 GNUPlot Script

The following GNUPlot script reads the data from the file generated by the C++ program and plots both the Lagrange and Newton's interpolated functions.

```gnuplot
# interpolation_plot.gp - GNUPlot script to plot Lagrange and Newton
    interpolation

set title "Lagrange and Newton Interpolation"
set xlabel "X"
set ylabel "Y"
set grid
set key outside

# Plot the Lagrange and Newton interpolation
plot "interpolation_data.txt" using 1:2 with lines title "Lagrange
    Interpolation", \
     "interpolation_data.txt" using 1:3 with lines title "Newton
    Interpolation"
```

© 2024 Nitesh Kumar. All rights reserved.

10.4 Conclusion

This experiment demonstrated the use of Lagrange and Newton's divided difference interpolation techniques for approximating functions. The interpolations were visualized using GNUPlot to compare the two methods.

© 2024 Nitesh Kumar. All rights reserved.

Chapter 11

Experiment No. 14: Numerical Integration of Functions and Discrete Data

11.1 Theory

Numerical integration refers to algorithms for computing the numerical value of a definite integral. While the exact analytical integration of some functions can be difficult or impossible, numerical methods allow for approximate solutions.

Two common methods of numerical integration are:

1. **Trapezoidal Rule**

2. **Simpson's Rule**

11.1.1 Trapezoidal Rule

The Trapezoidal rule approximates the area under a curve as a series of trapezoids. The rule is given by:

$$I \approx \frac{h}{2}\left(f(a) + f(b) + 2\sum_{i=1}^{n-1} f(x_i)\right)$$

where a and b are the limits of integration, $h = \frac{b-a}{n}$ is the step size, and $f(x_i)$ are the function values at the grid points.

11.1.2 Simpson's Rule

Simpson's rule approximates the integral by dividing the interval into an even number of subintervals and fitting a quadratic polynomial to the subintervals. It is given by:

$$I \approx \frac{h}{3}\left(f(a) + f(b) + 4\sum_{\text{odd } i} f(x_i) + 2\sum_{\text{even } i} f(x_i)\right)$$

11.2 C++ Code

The following C++ code implements both the Trapezoidal rule and Simpson's rule for numerical integration. It writes the integration results for different step sizes to a file for visualization using GNUPlot.

```
// C++ program to perform numerical integration using Trapezoidal and
    Simpson's Rule
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;

// Define the function to integrate
double f(double x) {
    return sin(x);  // Example: f(x) = sin(x)
}

// Trapezoidal rule implementation
double TrapezoidalRule(double a, double b, int n) {
    double h = (b - a) / n;
    double sum = f(a) + f(b);
    for (int i = 1; i < n; i++) {
        sum += 2 * f(a + i * h);
    }
    return (h / 2) * sum;
}

// Simpson's rule implementation
double SimpsonsRule(double a, double b, int n) {
    if (n % 2 != 0) n++; // Simpson's rule requires even number of
    intervals
    double h = (b - a) / n;
    double sum = f(a) + f(b);
    for (int i = 1; i < n; i++) {
        if (i % 2 == 0)
            sum += 2 * f(a + i * h);
        else
            sum += 4 * f(a + i * h);
    }
    return (h / 3) * sum;
}

int main() {
    int n;
    double a, b;

    // Open file to store the output
    ofstream outfile("integration_data.txt");

    // Input limits of integration and number of intervals
    cout << "Enter lower limit of integration: ";
    cin >> a;
    cout << "Enter upper limit of integration: ";
    cin >> b;
    cout << "Enter number of intervals: ";
    cin >> n;

```

© 2024 Nitesh Kumar. All rights reserved.

```
    // Write header to file
    outfile << "# N\tTrapezoidal\tSimpson" << endl;

    // Calculate and write the results for various numbers of intervals
    for (int i = 2; i <= n; i += 2) {
        double trap_result = TrapezoidalRule(a, b, i);
        double simp_result = SimpsonsRule(a, b, i);
        outfile << i << "\t" << trap_result << "\t" << simp_result << endl;
    }

    outfile.close();
    cout << "Data written to integration_data.txt for visualization in GNUPlot." << endl;

    return 0;
}
```

11.3 GNUPlot Script

The following GNUPlot script reads the data from the file generated by the C++ program and plots the results of numerical integration using the Trapezoidal and Simpson's rules for different numbers of intervals.

```
# integration_plot.gp - GNUPlot script to plot Trapezoidal and Simpson's rule integration results

set title "Numerical Integration: Trapezoidal vs Simpson's Rule"
set xlabel "Number of Intervals (N)"
set ylabel "Integral Value"
set grid
set key outside

# Plot the Trapezoidal and Simpson's integration results
plot "integration_data.txt" using 1:2 with lines title "Trapezoidal Rule", \
     "integration_data.txt" using 1:3 with lines title "Simpson's Rule"
```

11.4 Conclusion

This experiment demonstrated the use of numerical integration methods, specifically the Trapezoidal and Simpson's rules, to approximate definite integrals. The results were visualized using GNUPlot to compare the performance of both methods as the number of intervals increases.

© 2024 Nitesh Kumar. All rights reserved.

© 2024 Nitesh Kumar. All rights reserved.

Chapter 12

Experiment No. 15: Solving ODEs Using Euler and Runge-Kutta (RK) Methods

12.1 Theory

Ordinary differential equations (ODEs) arise in various physical systems, describing how a quantity changes with respect to another. Numerical methods, such as Euler's method and the Runge-Kutta (RK) methods, provide approximate solutions for ODEs that may not have analytical solutions.

12.1.1 Euler's Method

Euler's method is a simple, first-order numerical procedure for solving an initial value problem of the form:

$$y'(x) = f(x, y), \quad y(x_0) = y_0$$

The formula for updating y is given by:

$$y_{n+1} = y_n + hf(x_n, y_n)$$

where h is the step size, and $f(x_n, y_n)$ is the derivative evaluated at x_n and y_n.

12.1.2 Runge-Kutta Method (RK4)

The fourth-order Runge-Kutta method (RK4) is a higher-order method for solving ODEs. The formula for updating y is:

$$k_1 = hf(x_n, y_n)$$

$$k_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right)$$

$$k_3 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right)$$

$$k_4 = hf(x_n + h, y_n + k_3)$$

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

12.2 C++ Code

The following C++ code implements Euler's method and the fourth-order Runge-Kutta method to solve a first-order ODE and writes the solution to a file for visualization using GNUPlot.

```
// C++ program to solve ODE using Euler and RK4 methods
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;

// Define the function f(x, y) = dy/dx
double f(double x, double y) {
    return x * exp(-x) - y;  // Example: dy/dx = x*e^(-x) - y
}

// Euler's method implementation
void EulerMethod(double x0, double y0, double h, double x_end, ofstream &
    outfile) {
    double x = x0, y = y0;
    outfile << "# X\tEuler_Y\tRK4_Y" << endl;
    while (x <= x_end) {
        outfile << x << "\t" << y << "\t";
        y += h * f(x, y);
        x += h;
    }
}

// Runge-Kutta 4th Order method implementation
void RK4Method(double x0, double y0, double h, double x_end, ofstream &
    outfile) {
    double x = x0, y = y0;
    outfile.seekp(0);  // Reset the file position for the next method
    outfile.clear();   // Clear the file error flags

    while (x <= x_end) {
        outfile << y << endl;
        double k1 = h * f(x, y);
        double k2 = h * f(x + h/2, y + k1/2);
        double k3 = h * f(x + h/2, y + k2/2);
        double k4 = h * f(x + h, y + k3);
        y += (1.0/6.0) * (k1 + 2*k2 + 2*k3 + k4);
        x += h;
    }
}

int main() {
    double x0, y0, h, x_end;

    // Open file to store the output
    ofstream outfile("ode_solution.txt");

    // Input initial values
    cout << "Enter initial value of x (x0): ";
    cin >> x0;
    cout << "Enter initial value of y (y0): ";
    cin >> y0;
```

© 2024 Nitesh Kumar. All rights reserved.

```
    cout << "Enter step size (h): ";
    cin >> h;
    cout << "Enter final value of x: ";
    cin >> x_end;

    // Solve using Euler's method and Runge-Kutta method
    EulerMethod(x0, y0, h, x_end, outfile);
    RK4Method(x0, y0, h, x_end, outfile);

    outfile.close();
    cout << "Solution written to ode_solution.txt for visualization in GNUPlot." << endl;

    return 0;
}
```

12.3 GNUPlot Script

The following GNUPlot script reads the data from the file generated by the C++ program and plots the solutions obtained using Euler's method and the RK4 method.

```
# ode_plot.gp - GNUPlot script to plot Euler and RK4 solutions

set title "Solution of ODE using Euler and RK4 Methods"
set xlabel "X"
set ylabel "Y"
set grid
set key outside

# Plot the Euler and RK4 solutions
plot "ode_solution.txt" using 1:2 with lines title "Euler Method", \
     "ode_solution.txt" using 1:3 with lines title "Runge-Kutta Method"
```

12.4 Conclusion

This experiment demonstrated the use of numerical methods, specifically Euler's method and the fourth-order Runge-Kutta method (RK4), to solve first-order ordinary differential equations. The solutions were visualized using GNUPlot to compare the accuracy and stability of the two methods.

© 2024 Nitesh Kumar. All rights reserved.