Main Code

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

int main() {
    float x = M_PI/4, f1_3, f1_5, f1_e;
    float d_angle = 5;

    for (int i = 1; i <= 10; i++) {
        d_angle = d_angle / 10;

        f1_3 = (sin(x+d_angle) - sin(x-d_angle)) / (2*d_angle);
        f1_5 = (-sin(x+2*d_angle) + 8*sin(x+d_angle) - 8*sin(x-d_angle) + sin(x-2*d_angle))
/ (12*d_angle);
        f1_e = cos(x);

        cout << d_angle << "  " << f1_3 << "  " << f1_5 << "  " << f1_e << endl;
    }

    return 0;
}
```

This C++ code appears to be calculating the numerical derivatives of a function, specifically the first derivative of the sine function at a given angle `x`.

The code initializes the value of `x` to be `M_PI/4`, which is equivalent to 45 degrees in radians. It also initializes three variables: `f1_3`, `f1_5`, and `f1_e`.

The code then enters a loop that will run 10 times, each time with a smaller value of `d_angle`. Inside the loop, the code calculates `f1_3` and `f1_5`, which are approximations of the first derivative of the sine function at the angle `x` using the three-point and five-point formulas, respectively.

Finally, `f1_e` is set to the exact value of the first derivative of the sine function at `x`, which is cosine of `x`.

The code then outputs the value of `d_angle`, `f1_3`, `f1_5`, and `f1_e` for each iteration of the loop.

What is happening in this code?

This C++ code is implementing the Newton-Raphson method to find the roots of a given equation. The Newton-Raphson method is an iterative technique for finding the roots of a function. It requires an initial guess, and then refines the guess until the root is found with a desired level of accuracy.

The code defines a function `f(x)` which returns the value of the given equation at a given `x`. The code then initializes two variables `x0` and `eps`, which represent the initial guess and the desired level of accuracy, respectively.

The code enters a loop that will iterate until the difference between two consecutive approximations of the root is less than or equal to `eps`. Inside the loop, the code calculates the value of `f(x)` and its derivative `f'(x)` at the current guess `x0`.

The code then updates the guess for the root by subtracting the value of `f(x)/f'(x)` from the current guess `x0`. This is the Newton-Raphson iteration formula. The code repeats this process until the desired level of accuracy is achieved.

Finally, the code outputs the root value and the number of iterations it took to find the root.

It's worth noting that the Newton-Raphson method may not always converge to a root depending on the initial guess and the behavior of the function. In such cases, the loop may not terminate, or the method may converge to a different root or a local minimum or maximum of the function.

Why float d_angle = 5;?

The variable `d_angle` is initialized to `5` because it represents the step size used in the numerical differentiation formulas used in the code.

In the loop that follows, `d_angle` is repeatedly divided by `10` in each iteration, which effectively decreases the step size used in the differentiation formulas by a factor of `10` in each iteration. This allows the approximation of the derivative to become more accurate as the step size becomes smaller.

By initializing `d_angle` to `5`, the code is starting with a relatively large step size and then gradually refining the approximation of the derivative by decreasing the step size.

Why d_angle = d_angle / 10; inside the loop?

The statement `d_angle = d_angle / 10` inside the loop is updating the value of `d_angle` in each iteration. This is because `d_angle` is used as the step size in the numerical differentiation formulas used in the loop, and the code is refining the approximation of the derivative by decreasing the step size in each iteration.

By dividing `d_angle` by `10` in each iteration, the step size used in the differentiation formulas becomes smaller by a factor of `10` in each iteration. This allows the approximation of the derivative to become more accurate as the step size becomes smaller.

Therefore, by updating the value of `d_angle` in each iteration, the code is effectively refining the approximation of the derivative by using a smaller step size in each iteration.

Why inside the loop i <= 10? What if I use i <= 20??

The loop `for (int i = 1; i <= 10; i++)` is iterating 10 times because the code is designed to output the approximation of the derivative for a range of step sizes, starting with a relatively large step size and then refining the approximation by decreasing the step size in each iteration.

By iterating 10 times, the code calculates and outputs the approximation of the derivative for 10 different step sizes. The specific number of iterations and step sizes used in the loop depends on the desired level of accuracy and the properties of the function being differentiated.

If you change the condition `i <= 10` to `i <= 20`, the loop will iterate 20 times instead of 10 times. This means that the code will calculate and output the approximation of the derivative for 20 different step sizes, starting with a relatively large step size and then refining the approximation by decreasing the step size in each iteration.

However, it's important to note that increasing the number of iterations may not always lead to more accurate results. There may be a point where decreasing the step size further does not significantly improve the accuracy of the approximation, or where numerical errors begin to accumulate and affect the accuracy of the results. Therefore, the number of iterations used in the loop should be chosen based on the desired level of accuracy and the properties of the function being differentiated.

Why f1_3 = (sin(x+d_angle) - sin(x-d_angle))/(2*d_angle);?

The expression `f1_3 = (sin(x+d_angle) - sin(x-d_angle))/(2*d_angle)` is an approximation of the first derivative of the function `sin(x)` at the point `x`, using the central difference method with a step size of `2*d_angle`.

The central difference method approximates the derivative of a function at a point by taking the difference of function values at two neighboring points on either side of the point of interest, divided by the distance between those points. Specifically, the first derivative of a function `f(x)` at a point `x` can be approximated by:

f'(x) ≈ (f(x+h) - f(x-h)) / (2h)

where `h` is the step size.

In this case, the function being differentiated is `sin(x)`, and the code is using a step size of `2*d_angle`. Therefore, the expression can be rewritten as:

f1_3 = (sin(x+d_angle) - sin(x-d_angle))/(2*d_angle)

This expression approximates the first derivative of `sin(x)` at the point `x` by taking the difference between `sin(x+d_angle)` and `sin(x-d_angle)`, divided by `2*d_angle`.

By using the central difference method with a relatively small step size, the approximation of the derivative becomes more accurate, as the error associated with the approximation decreases as the step size decreases. However, using too small of a step size can lead to numerical instability and inaccuracies due to numerical roundoff errors. The specific step size used depends on the desired level of accuracy and the properties of the function being differentiated.

f1_5 = (-sin(x+2*d_angle) + 8*sin(x+d_angle) - 8*sin(x-d_angle) + sin(x-2*d_angle)) / (12*d_angle)

The expression `f1_5 = (-sin(x+2*d_angle) + 8*sin(x+d_angle) - 8*sin(x-d_angle) + sin(x-2*d_angle)) / (12*d_angle)` is also an approximation of the first derivative of the function `sin(x)` at the point `x`, but it uses a higher order method called the five-point stencil formula.

The five-point stencil formula approximates the first derivative of a function at a point by taking a weighted combination of function values at five neighboring points on either side of the point of interest, divided by the distance between those points. Specifically, the first derivative of a function `f(x)` at a point `x` can be approximated by:

f'(x) ≈ (-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)) / (12h)

where `h` is the step size.

In this case, the function being differentiated is `sin(x)`, and the code is using a step size of `2*d_angle`. Therefore, the expression can be rewritten as:

f1_5 = (-sin(x+2*d_angle) + 8*sin(x+d_angle) - 8*sin(x-d_angle) + sin(x-2*d_angle)) / (12*d_angle)

This expression approximates the first derivative of `sin(x)` at the point `x` by taking a weighted combination of function values at `x-2*d_angle`, `x-d_angle`, `x+d_angle`, `x+2*d_angle`, and `x+3*d_angle`, divided by `12*d_angle`.

By using the five-point stencil formula, this approximation of the derivative becomes more accurate than the three-point stencil formula used in `f1_3`, but it requires more

function evaluations and is more computationally expensive. Again, the specific step size used depends on the desired level of accuracy and the properties of the function being differentiated.

Here f1_3 and f1_5 both are first derivative of sinx , right? And f1_5 provide more accurate result than f1_3?

Yes, both `f1_3` and `f1_5` are approximations of the first derivative of `sin(x)` at `x = pi/4`. The difference between them is that `f1_3` is computed using a three-point stencil, while `f1_5` is computed using a five-point stencil.

The five-point stencil typically provides a more accurate approximation than the three-point stencil, because it takes into account more information about the function. In this case, `f1_5` provides a more accurate result than `f1_3` for the same step size.

However, it's important to note that the accuracy of the approximation depends not only on the stencil used, but also on the step size `h`. In general, smaller step sizes lead to more accurate approximations. So, it's a good idea to experiment with different stencils and step sizes to find the best combination for your particular problem.

Why f1_e = cos(x);?

The line `f1_e = cos(x);` is calculating the exact value of the first derivative of `sin(x)` at `x = pi/4`, which is `cos(pi/4) = sqrt(2)/2`. This value is then used in the output to compare with the approximations obtained using the three-point stencil and the five-point stencil.

In general, it's a good idea to compare the approximations obtained using numerical methods with the exact solution (if it's known) to assess the accuracy of the numerical method.

What is the order of f1_3 & f1_5 respectively?

In numerical differentiation, the order of accuracy refers to the number of significant digits in the approximation.

The formula used to calculate `f1_3` is a second-order approximation, meaning that it is accurate up to the second order of the step size `d_angle`. The formula used to calculate `f1_5` is a fourth-order approximation, meaning that it is accurate up to the fourth order of the step size `d_angle`.

In other words, `f1_5` is a higher-order approximation than `f1_3`, which means that it is expected to be more accurate than `f1_3`. However, it's worth noting that higher-order approximations can also be more sensitive to errors in the input data or rounding errors in the calculations.