C-PROGRAM

PROGRAM IN C FOR ADAMS-MOULTON METHOD

```
#include <stdio.h>

// Define the function f(x, y)
double f(double x, double y) {
    return 3 * x * x * (1 + y);
}

// Runge-Kutta method to compute y-values
void rungeKutta(double x0, double y0, double h, int steps, double y_values[]) {
    double x = x0, y = y0;

    // Store the initial y-value
    y_values[0] = y0;

    for (int i = 1; i <= steps; i++) {
        double k1 = h * f(x, y);
        double k2 = h * f(x + h / 2.0, y + k1 / 2.0);
        double k3 = h * f(x + h / 2.0, y + k2 / 2.0);
        double k4 = h * f(x + h, y + k3);

        y += (k1 + 2 * k2 + 2 * k3 + k4) / 6.0;
        x += h;

        y_values[i] = y; // Store the computed y-value
    }
}

// Adams-Moulton method to compute y-values
void adams_moulton(double h, double t[], double y[], int n_points) {
```

```
    for (int i = 3; i < n_points - 1; i++) {
        t[i + 1] = t[i] + h; // Next time step

        // Predictor: Adams-Bashforth 4-step formula
        double y_pred = y[i] + (h / 24.0) * (
            55 * f(t[i], y[i]) -
            59 * f(t[i - 1], y[i - 1]) +
            37 * f(t[i - 2], y[i - 2]) -
             9 * f(t[i - 3], y[i - 3])
        );

        // Corrector: Iterative Adams-Moulton formula
        double y_corr = y_pred;
        for (int iter = 0; iter < 5; iter++) { // Perform 5 iterations for refinement
            y_corr = y[i] + (h / 24.0) * (
                9 * f(t[i + 1], y_corr) +
                19 * f(t[i], y[i]) -
                5 * f(t[i - 1], y[i - 1]) +
                f(t[i - 2], y[i - 2])
            );
        }

        y[i + 1] = y_corr; // Update the solution
    }
}

int main() {
    // Step size
    double h = 0.05;

    // Initial conditions
```

```
    double x0 = 0.0, y0 = 0.0;

    // Number of steps to reach t = 0.2
    int steps = 4;

    // Time array and y-values array
    double t[steps + 1];
    double y[steps + 1];

    // Initialize time steps
    for (int i = 0; i <= steps; i++) {
        t[i] = x0 + i * h;
    }

    // Compute initial y-values using Runge-Kutta method
    rungeKutta(x0, y0, h, steps, y);

    // Print the computed y-values using Runge-Kutta
    printf("Initial values computed using Runge-Kutta:\n");
    for (int i = 0; i <= steps; i++) {
        printf("t = %.2f, y = %.6f\n", t[i], y[i]);
    }

    // Use Adams-Moulton for further refinement if needed
    adams_moulton(h, t, y, steps + 1);

    // Print the final result
    printf("\nApproximate value of y(0.2): %.6f\n", y[steps]);

    return 0;
}
```

****
****

PROGRAM FOR RK METHOD

```
#include <stdio.h>


// Define the function f(x, y)
double f(double x, double y) {
   return 3 * x * x * (1 + y);
}


// Runge-Kutta method to compute y-values
void rungeKutta(double x0, double y0, double h, int steps, double y_values[]) {
   double x = x0, y = y0;


   // Store the initial y-value
   y_values[0] = y0;


   for (int i = 1; i <= steps; i++) {
      double k1 = h * f(x, y);
      double k2 = h * f(x + h / 2.0, y + k1 / 2.0);
      double k3 = h * f(x + h / 2.0, y + k2 / 2.0);
      double k4 = h * f(x + h, y + k3);


      y += (k1 + 2 * k2 + 2 * k3 + k4) / 6.0;
      x += h;


      y_values[i] = y; // Store the computed y-value
   }
}


// Adams-Moulton method to compute y-values
```

```
void adams_moulton(double h, double t[], double y[], int n_points) {
    for (int i = 3; i < n_points - 1; i++) {
        t[i + 1] = t[i] + h; // Next time step

        // Predictor: Adams-Bashforth 4-step formula
        double y_pred = y[i] + (h / 24.0) * (
            55 * f(t[i], y[i]) -
            59 * f(t[i - 1], y[i - 1]) +
            37 * f(t[i - 2], y[i - 2]) -
             9 * f(t[i - 3], y[i - 3])
        );

        // Corrector: Iterative Adams-Moulton formula
        double y_corr = y_pred;
        for (int iter = 0; iter < 5; iter++) { // Perform 5 iterations for refinement
            y_corr = y[i] + (h / 24.0) * (
                9 * f(t[i + 1], y_corr) +
                19 * f(t[i], y[i]) -
                5 * f(t[i - 1], y[i - 1]) +
                f(t[i - 2], y[i - 2])
            );
        }

        y[i + 1] = y_corr; // Update the solution
    }
}

int main() {
    // Step size
    double h = 0.05;
```

```
    // Initial conditions
    double x0 = 0.0, y0 = 0.0;

    // Number of steps to reach t = 0.2
    int steps = 4;

  // Time array and y-values array
    double t[steps + 1];
    double y[steps + 1];

    // Initialize time steps
    for (int i = 0; i <= steps; i++) {
        t[i] = x0 + i * h;
    }

    // Compute initial y-values using Runge-Kutta method
    rungeKutta(x0, y0, h, steps, y);

    // Print the computed y-values using Runge-Kutta
    printf("Initial values computed using Runge-Kutta:\n");
    for (int i = 0; i <= steps; i++) {
        printf("t = %.2f, y = %.6f\n", t[i], y[i]);
    }

    // Use Adams-Moulton for further refinement if needed
    adams_moulton(h, t, y, steps + 1);

    // Print the final result
    printf("\nApproximate value of y(0.2): %.6f\n", y[steps]);

    return 0;
```

```
}
```

****
****

PROGRAM FOR EULERS METHOD

```c
#include <stdio.h>

// Define the function f(x, y)
double f(double x, double y) {
    // Example differential equation: dy/dx = x + y
    return x + y;
}

// Euler's Method Function
void eulerMethod(double x0, double y0, double h, int steps) {
    double x = x0, y = y0;

    // Print table header
    printf("\nStep\t x\t\t y\n");
    printf("--------------------------------\n");

    // Iterative calculation using Euler's method
    for (int i = 0; i <= steps; i++) {
        printf("%d\t %.6f\t %.6f\n", i, x, y);

        // Update values using Euler's formula
        y = y + h * f(x, y);
        x = x + h;
    }
}

// Main function
```

```
int main() {
    // Declare variables
    double x0, y0, h;
    int steps;

    // Input initial conditions
    printf("Enter initial value of x (x0): ");
    scanf("%lf", &x0);

    printf("Enter initial value of y (y0): ");
    scanf("%lf", &y0);

    // Input step size
    printf("Enter step size (h): ");
    scanf("%lf", &h);

    // Input number of steps
    printf("Enter number of steps: ");
    scanf("%d", &steps);

    // Confirm inputs
    printf("\nInitial conditions: x0 = %.6f, y0 = %.6f\n", x0, y0);
    printf("Step size: h = %.6f\n", h);
    printf("Number of steps: %d\n\n", steps);

    // Solve using Euler's method
    printf("Solving the differential equation using Euler's Method...\n");
    eulerMethod(x0, y0, h, steps);

    printf("\nSolution completed.\n");
```

```
    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>

// Function defining the ODE: y' = f(t, y)
double f(double t, double y) {
    return t - y; // Example: Replace this with your ODE
}

// Function to initialize the first few steps using Euler's method
void euler_method(double t[], double y[], int initial_steps, double h) {
    for (int i = 1; i < initial_steps; i++) {
        t[i] = t[i - 1] + h;
        y[i] = y[i - 1] + h * f(t[i - 1], y[i - 1]);
    }
}

// Adams-Bashforth coefficients for up to 4 steps
void get_ab_coefficients(int steps, double coeff[]) {
    switch (steps) {
        case 1:
            coeff[0] = 1.0;
            break;
        case 2:
            coeff[0] = 3.0 / 2.0;
            coeff[1] = -1.0 / 2.0;
            break;
        case 3:
            coeff[0] = 23.0 / 12.0;
            coeff[1] = -16.0 / 12.0;
```

```
            coeff[2] = 5.0 / 12.0;
            break;
        case 4:
            coeff[0] = 55.0 / 24.0;
            coeff[1] = -59.0 / 24.0;
            coeff[2] = 37.0 / 24.0;
            coeff[3] = -9.0 / 24.0;
            break;
        default:
            printf("Unsupported number of steps: %d\n", steps);
            exit(1);
    }
}

// Adams-Bashforth multistep method
void adams_bashforth_multistep(double t0, double y0, double t_end, double h, int steps) {
    int n = (int)((t_end - t0) / h); // Total number of steps
    double t[n + 1], y[n + 1];
    double coeff[steps]; // Coefficients for the Adams-Bashforth method

    // Initialize time and solution arrays
    t[0] = t0;
    y[0] = y0;

    // Get the coefficients for the specified method
    get_ab_coefficients(steps, coeff);

    // Initialize the first few steps using Euler's method
    euler_method(t, y, steps, h);

    // Print the initial values
```

```c
  for (int i = 0; i < steps; i++) {
    printf
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void gaussian_elimination(int n, double a[n][n + 1], double x[n]) {
  for (int i = 0; i < n; i++) {
    // Partial Pivoting
    for (int k = i + 1; k < n; k++) {
      if (fabs(a[i][i]) < fabs(a[k][i])) {
        for (int j = 0; j <= n; j++) {
          double temp = a[i][j];
          a[i][j] = a[k][j];
          a[k][j] = temp;
        }
      }
    }

    // Forward Elimination
    for (int k = i + 1; k < n; k++) {
      double factor = a[k][i] / a[i][i];
      for (int j = 0; j <= n; j++) {
        a[k][j] -= factor * a[i][j];
      }
    }
  }

  // Back Substitution
  for (int i = n - 1; i >= 0; i--) {
    x[i] = a[i][n];
```

```
        for (int j = i + 1; j < n; j++) {
            x[i] -= a[i][j] * x[j];
        }
        x[i] /= a[i][i];
    }
}

int main() {
    int n = 3; // Example: Size of the system
    double a[3][4] = {
        {2, -1, 1, 3},
        {1, 3, 2, 12},
        {1, -1, 2, 2}
    }; // Example augmented matrix
    double x[3];

    gaussian_elimination(n, a, x);

    printf("Solution:\n");
    for (int i = 0; i < n; i++) {
        printf("x%d = %.4f\n", i + 1, x[i]);
    }

    return 0;
}
```

```
#include <stdio.h>
#include <math.h>

double f(double x) {
    return x * x - 4; // Example: f(x) = x^2 - 4
}
```

```
double f_prime(double x) {
    return 2 * x; // Derivative: f'(x) = 2x
}

void newton_raphson(double initial_guess, double tolerance, int max_iterations) {
    double x = initial_guess;
    for (int i = 0; i < max_iterations; i++) {
        double fx = f(x);
        double fpx = f_prime(x);
        if (fabs(fpx) < 1e-10) {
            printf("Derivative is too small. Method fails.\n");
            return;
        }
        double x_next = x - fx / fpx;

        printf("Iteration %d: x = %.6f, f(x) = %.6f\n", i + 1, x_next, f(x_next));

        if (fabs(x_next - x) < tolerance) {
            printf("Root found: x = %.6f\n", x_next);
            return;
        }
        x = x_next;
    }
    printf("Maximum iterations reached. No solution found.\n");
}

int main() {
    double initial_guess = 2.0; // Starting point
    double tolerance = 1e-6;
    int max_iterations = 20;
```

```
    newton_raphson(initial_guess, tolerance, max_iterations);

    return 0;
}
```

```
#include <stdio.h>
#include <math.h>

double f(double x) {
    return x * x; // Example: f(x) = x^2
}

double trapezoidal_rule(double a, double b, int n) {
    double h = (b - a) / n;
    double sum = f(a) + f(b);

    for (int i = 1; i < n; i++) {
        double x = a + i * h;
        sum += 2 * f(x);
    }

    return (h / 2) * sum;
}

int main() {
    double a = 0.0; // Start of the interval
    double b = 1.0; // End of the interval
    int n = 100;    // Number of subintervals

    double result = trapezoidal_rule(a, b, n);
```

```
    printf("Integral result: %.6f\n", result);

    return 0;
}
```

```
#include <stdio.h>
#include <math.h>

void matrix_vector_mult(int n, double A[n][n], double v[n], double result[n]) {
    for (int i = 0; i < n; i++) {
        result[i] = 0.0;
        for (int j = 0; j < n; j++) {
            result[i] += A[i][j] * v[j];
        }
    }
}

void normalize(int n, double v[n]) {
    double norm = 0.0;
    for (int i = 0; i < n; i++) {
        norm += v[i] * v[i];
    }
    norm = sqrt(norm);
    for (int i = 0; i < n; i++) {
        v[i] /= norm;
    }
}

double power_method(int n, double A[n][n], double v[n], int max_iterations, double tolerance) {
    double eigenvalue = 0.0;
    double prev_eigenvalue = 0.0;
```

```
    for (int iter = 0; iter < max_iterations; iter++) {
        double w[n];
        matrix_vector_mult(n, A, v, w);
        normalize(n, w);

        prev_eigenvalue = eigenvalue;
        eigenvalue = 0.0;
        for (int i = 0; i < n; i++) {
            eigenvalue += w[i] * w[i];
        }

        for (int i = 0; i < n; i++) {
            v[i] = w[i];
        }

        if (fabs(eigenvalue - prev_eigenvalue) < tolerance) {
            return eigenvalue;
        }
    }
    return eigenvalue;
}

int main() {
    int n = 2; // Matrix size
    double A[2][2] = {
        {4, 1},
        {2, 3}
    };
    double v[2] = {1, 1}; // Initial guess
    int max_iterations = 1000;
    double tolerance = 1e-6;
```

```
    double  eigenvalue = power_method(n,  A, v, max_iterations,  tolerance);


    printf("Dominant  eigenvalue: %.6f\n",  eigenvalue);
    printf("Eigenvector:  [%.6f, %.6f]\n", v[0], v[1]);


    return 0;
}
```