El método numérico para resolver una ecuación diferencial ordinaria de la forma $\frac{dy}{dx} = f(x, y)$ es, en términos matemáticos:

$$y_{i+1} = y_i + \phi h \tag{1}$$

De acuerdo con esta ecuación, la pendiente estimada ϕ se usa para extrapolar desde un valor anterior y_i a un nuevo valor y_{i+1} en una distancia h. Esta fórmula se aplica paso a paso para calcular un valor posterior y, por lo tanto, para trazar la trayectoria de la solución.

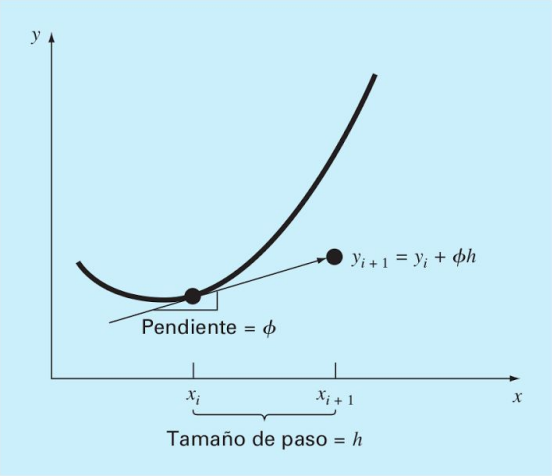

Figura 1. Ilustración gráfica del método de un paso

Todos los métodos de un paso que se expresen de esta forma general, tan sólo van a diferir en la manera en la que se estima la pendiente. Como en el problema del paracaidista en caída, el procedimiento más simple consiste en usar la ecuación diferencial, para estimar la pendiente, en la forma de la primera derivada en x_i.

Método de Euler

La primera derivada ofrece una estimación directa de la pendiente en x_i:

$$\phi = f(x_i, y_i)$$

donde $f(x_i, y_i)$ es la ecuación diferencial evaluada en x_i y y_i. La estimación se sustituye en la ecuación (1):

$$y_{i+1} = y_i + f(x_i, y_i)h \tag{2}$$

Esta fórmula se conoce como método de Euler (o de Euler-Cauchy o de punto pendiente). Se predice un nuevo valor de y usando la pendiente (igual a la primera derivada en el valor original de x) para extrapolar linealmente sobre el tamaño de paso h.

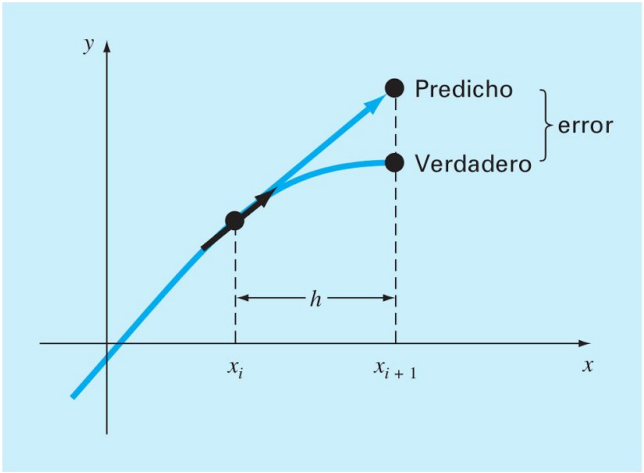

Figura 2. Método de Euler

Métodos de Runge-Kutta

Los métodos de Runge-Kutta (RK) logran la exactitud del procedimiento de la serie de Taylor sin necesitar el cálculo de derivadas de orden superior. Existen muchas variantes, pero todas tienen la forma generalizada de la ecuación (1):

$$y_{i+1} = y_i + \phi(x_i, y_i, h)h \tag{3}$$

donde $\phi(x_i, y_i, h)$ se conoce como función incremento, la cual puede interpretarse como una pendiente representativa en el intervalo. La función incremento se escribe en forma general como

$$\phi = a_i k_i + a_2 k_2 + \cdots + a_n k_n \tag{4}$$

donde las a son constantes y las k son

$$k_1 = f(x_i, y_i) \tag{4a}$$

$$k_2 = f(x_i + p_1 h,\ y_i + q_{11} k_1 h) \tag{4b}$$

$$k_3 = f(x_i + p_2 h,\ y_i + q_{21} k_1 h + q_{22} k_2 h) \tag{4c}$$

.

.

.

$$k_n = f(x_i + p_{n-1} h,\ y_i + q_{n-1,1} k_1 h + q_{n-1,2} k_2 h + \cdots + q_{n-1,n} k_{n-1} h) \tag{4d}$$

donde las p y las q son constantes. Observe que las k son relaciones de recurrencia. Es decir, k_1 aparece en la ecuación k_2, la cual aparece en la ecuación k_3, etcétera. Como cada k es una evaluación funcional, esta recurrencia vuelve eficientes a los métodos RK para cálculos en computadora.

Es posible tener varios tipos de métodos de Runge-Kutta empleando diferentes números de términos en la función incremento especificada por n. Una vez que se elige n, se evalúan las a, p y q igualando la ecuación (3) a los términos en la expansión de la serie de Taylor.

Métodos de Runge-Kutta de cuarto orden

El más popular de los métodos RK es el de cuarto orden. La siguiente, es la forma comúnmente usada y, por lo tanto, le llamamos método clásico RK de cuarto orden:

$$y_{i+1} = y_i + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)h \qquad (5)$$

Donde

$$k_1 = f(x_i, y_i) \qquad (5a)$$

$$k_2 = f(x_i + \tfrac{1}{2}h,\ y_i + \tfrac{1}{2}k_1 h) \qquad (5b)$$

$$k_3 = f(x_i + \tfrac{1}{2}h,\ y_i + \tfrac{1}{2}k_2 h) \qquad (5c)$$

$$k_4 = f(x_i + h,\ y_i + k_3 h) \qquad (5d)$$

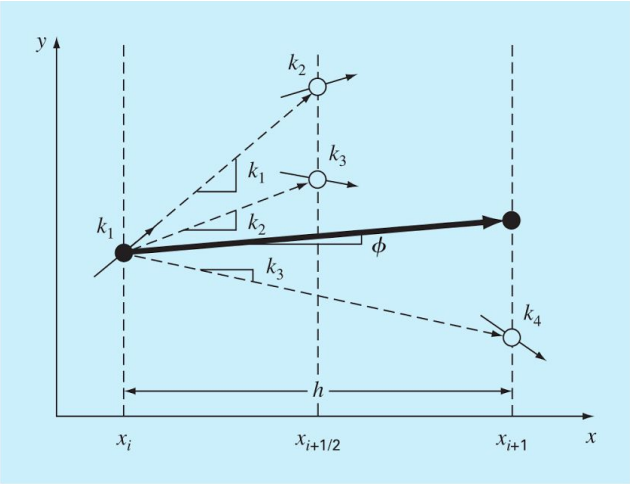

Figura 3. Representación gráfica de las pendientes estimadas empleadas en el método Rk de cuarto orden

Programa 3: Métodos Runge-Kutta

Características del programa

- **Lenguaje de programación:** C++11
- **Paradigma de programación:** Orientado a Objetos
- **Herramienta GNU/Linux para generar gráficas 2D/3D:** Gnuplot
- **Compilador:** GNU C++ (g++)

Se desarrollaron tres programas diferentes, uno para el método de Euler, otro para el método de Runge-Kutta de cuarto orden, y otro con ambos métodos. Para cada uno se generó un archivo .hpp y otro .cpp

1. **El problema del paracaidista con tamaño de paso de 1 y de 0.5**

Modelo matemático:

$$\frac{dv}{dt} = g - \frac{c}{m}v \tag{6}$$

Para el caso donde $g = 9.8$, $c = 12.4$, $m = 68.1$, y $v = 0$ en $t = 0$.

La solución exacta está dada por la ecuación:

$$v = \frac{mg}{c}(1 - e^{-\frac{c}{m}t}) \tag{7}$$

Método de Euler

Euler.hpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <iostream>
#include <iomanip>

#define xi 0
#define xf 12
#define yi 0
#define g 9.8
#define c 12.5
#define m 68.1

class Euler {
 private:
  float h;
  FILE *fxpoints;
  FILE *fxpoints2;
  FILE *gnuplot;

 public:
  Euler();
  ~Euler();

  float f(float x, float y);
  float real(float x);
  float Eulerformula(float x, float y);
  void getValues(float step, int n);
  void Eulermethod(float step1, float step2);

  void printElement(float x, float yr, float y, float e);
  void printHeader();

  void savePoints(float x, float y, int n);
  void plot();
```

```
  void replot();
};
```

Euler.cpp

```
// Copyright
#include "Euler.hpp"

Euler::Euler() {
  fxpoints = fopen("fxpoints.txt", "w");
  fxpoints2 = fopen("fxpoints2.txt", "w");
}

Euler::~Euler() {}

float Euler::f(float x, float y) {
  float fx;
  fx = g - (c/m)*y;
  // fx = -2*pow(x, 3) + 12*pow(x, 2) - 20*x + 8.5;
  return fx;
}

float Euler::real(float x) {
  float y;
  y = (g*m)/c * (1 - exp((-c/m)*x));
  // y = -0.5*pow(x, 4) + 4*pow(x, 3) - 10*pow(x, 2) + 8.5*x +1;
  return y;
}

float Euler::Eulerformula(float x, float y) {
  float yk;
  yk = y + f(x, y)*h;
  return yk;
}

void Euler::getValues(float step, int n) {
  float x, y, yk, yreal, error;
  h = step;
  x = xi;
  y = yi;
  error = 0;
  std::cout << "Euler's method with step size: " << h << std::endl;
  printHeader();
  printElement(x, y, y, error);
  savePoints(x, y, n);
  while (x < xf) {
          yk = Eulerformula(x, y);
          x += h;
```

```
            y = yk;
            yreal = real(x);
            error = 100*fabs(yreal-y)/yreal;
            printElement(x, yreal, y, error);
            savePoints(x, y, n);
  }
  std::cout << std::endl;
  if (n == 1)
            plot();
  else
            replot();
}

void Euler::Eulermethod(float step1, float step2) {
  getValues(step1, 1);
  getValues(step2, 2);
}

void Euler::printElement(float x, float yr, float y, float e) {
  const int precision = 5;
  const int& width = 10;
  std::cout << '|' << std::setw(width) << std::setprecision(precision) << x << std::fixed << std::setw(width);
  std::cout << '|' << std::setw(width) << std::setprecision(precision) << yr << std::fixed << std::setw(width);
  std::cout << '|' << std::setw(width) << std::setprecision(precision) << y << std::fixed <<  std::setw(width);
  std::cout << '|' << std::setw(width) << std::setprecision(precision) << e << std::fixed <<  std::setw(width);
  std::cout << '|' << std::endl;
}

void Euler::printHeader() {
  const int& width = 10;
  std::cout << '|' << std::setw(width) << "x" << std::setw(width);
  std::cout << '|' << std::setw(width) << "yReal" << std::setw(width);
  std::cout << '|' << std::setw(width) << "yEuler" << std::setw(width);
  std::cout << '|' << std::setw(width) << "error" << std::setw(width);
  std::cout << '|' << std::endl;
}

void Euler::savePoints(float x, float y, int n) {
  if (n == 1) {
            fprintf(fxpoints, "%lf %lf \n", x, y);
  }
  else {
            fprintf(fxpoints2, "%lf %lf \n", x, y);
  }
}

void Euler::plot() {
  gnuplot = popen("gnuplot -persist", "w");
```

```
    // fprintf(gnuplot, "set title \"Euler's Method for y' = -2x^3 + 12x^2 -20x + 8.5\" font \"Times-Roman,14\"\n");
  // fprintf(gnuplot, "f(x) = %s\n", "-0.5*x**4 + 4*x**3 - 10 *x**2 + 8.5*x +1");
  fprintf(gnuplot, "set title \"Euler's Method for y' = g - c/m *y\" font \"Times-Roman,14\"\n");
  fprintf(gnuplot, "f(x) = (%f*%f)/%f * (1 - exp((-%f/%f)*x))\n", g, m, c, c, m);
  fprintf(gnuplot, "set xrange[%d:%d]\n", xi, xf);
   fprintf(gnuplot, "plot f(x) title \"real solution\", \"fxpoints.txt\" u 1:2 w l lc rgb '#0072bd' title \"h = %.1f\", " u 1:2 w p ls 7 ps 2 lc rgb '#0072bd' notitle\n", h);
  fclose(fxpoints);
}

void Euler::replot() {
   fprintf(gnuplot, "replot \"fxpoints2.txt\" u 1:2 w l lc rgb '#DC143C' title \"h = %.1f\", '' u 1:2 w p ls 7 ps 2 lc rgb '#DC143C' notitle\n", h);
  fclose(fxpoints2);
}

int main() {
  Euler euler;
  euler.Eulermethod(0.5,1);
  return 0;
}
```

Compilación del programa usando g++

Se compila la cabecera del archivo:

```
g++  Euler.hpp
```

Se compila el archivo fuente y se genera un ejecutable:

```
g++  Euler.cpp -o Euler1.x
```

Ejecución

Se ejecuta el archivo ejecutable ./Euler1.x:

```
cynthia@taiga:~/Documents/Mate/Programas$ ./Euler1.x
```

Resultados

Al ejecutar el programa se imprimen en cada iteración, los valores de x, y real, y obtenida por el método de Euler, y el error. Sin embargo, en este documento, se incluirán únicamente las tablas donde se encuentran ambos métodos para no tener redundancia de información. Por otro lado, sí se mostrarán las gráficas generadas para cada caso.

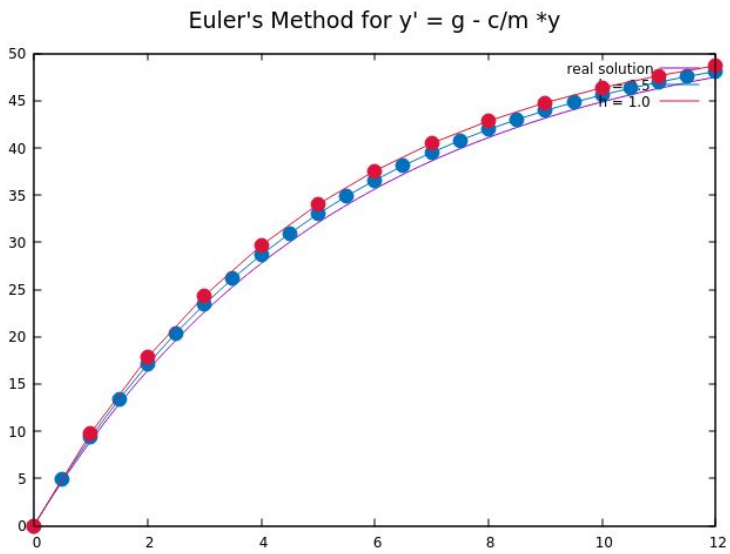

Figura 4. Comparación de la solución verdadera con una solución numérica usando el método de Euler, para la integral del problema del paracaidista desde $x = 0$ hasta $x = 12$ con tamaños de paso de 0.5 y 1 respectivamente. La condición inicial en $x = 0$ es $y = 0$

Método de Runge-Kutta de cuarto orden

RungeKutta.hpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <iostream>
#include <iomanip>

#define xi 0
#define xf 12
#define yi 0
#define g 9.8
#define c 12.5
#define m 68.1

class RungeKutta {
 private:
  float h;
  FILE *fxpoints;
  FILE *fxpoints2;
  FILE *gnuplot;

 public:
  RungeKutta();
  ~RungeKutta();
```

```
  float f(float x, float y);
  float real(float x);
  float RKfourth(float x, float y);
  void getValues(float step, int n);
  void RKMethod(float step1, float step2);

  void printElement(float x, float yr, float y, float e);
  void printHeader();

  void savePoints(float x, float y, int n);
  void plot();
  void replot();
};
```

RungeKutta.cpp

```
// Copyright
#include "RungeKutta.hpp"

RungeKutta::RungeKutta() {
  fxpoints = fopen("fxpoints.txt", "w");
  fxpoints2 = fopen("fxpoints2.txt", "w");
}

RungeKutta::~RungeKutta() {}

float RungeKutta::f(float x, float y) {
  float fx;
  fx = g - (c/m)*y;
  // fx = -2*pow(x, 3) + 12*pow(x, 2) - 20*x + 8.5;
  return fx;
}

float RungeKutta::real(float x) {
  float y;
  y = (g*m)/c * (1 - exp((-c/m)*x));
  // y = -0.5*pow(x, 4) + 4*pow(x, 3) - 10*pow(x,2) + 8.5*x +1;
  return y;
}

float RungeKutta::RKfourth(float x, float y) {
  float yk, k1, k2, k3, k4;
  k1 = f(x, y);
  k2 = f(x + 0.5*h, y + 0.5*k1*h);
  k3 = f(x + 0.5*h, y + 0.5*k2*h);
  k4 = f(x + h, y + k3*h);
  yk = y + ((k1 + 2*k2 + 2*k3 + k4)/6)*h;
  return yk;
}
```

```
void RungeKutta::getValues(float step, int n) {
  float x, y, yk, yreal, error;
  h = step;
  x = xi;
  y = yi;
  error = 0;
  std::cout << "Fourth-order Runge-Kutta's method with step size: " << h << std::endl;
  printHeader();
  printElement(x, y, y, error);
  savePoints(x, y, n);
  while (x < xf) {
          yk = RKfourth(x, y);
          x += h;
          y = yk;
          yreal = real(x);
          error = 100*fabs(yreal-y)/yreal;
          printElement(x, yreal, y, error);
          savePoints(x, y, n);
  }
  std::cout << std::endl;
  if (n == 1)
          plot();
  else
          replot();
}

void RungeKutta::RKMethod(float step1, float step2) {
  getValues(step1, 1);
  getValues(step2, 2);
}

void RungeKutta::printElement(float x, float yr, float y, float e) {
  const int precision = 5;
  const int& width = 10;
  std::cout << '|' << std::setw(width) << std::setprecision(precision) << x << std::fixed << std::setw(width);
  std::cout << '|' << std::setw(width) << std::setprecision(precision) << yr << std::fixed << std::setw(width);
  std::cout << '|' << std::setw(width) << std::setprecision(precision) << y << std::fixed <<  std::setw(width);
  std::cout << '|' << std::setw(width) << std::setprecision(precision) << e << std::fixed <<  std::setw(width);
  std::cout << '|' << std::endl;
}

void RungeKutta::printHeader() {
  const int& width = 10;
  std::cout << '|' << std::setw(width) << "x" << std::setw(width);
  std::cout << '|' << std::setw(width) << "yReal" << std::setw(width);
  std::cout << '|' << std::setw(width) << "yRK" << std::setw(width);
  std::cout << '|' << std::setw(width) << "error" << std::setw(width);
```

```
  std::cout << '|' << std::endl;
}

void RungeKutta::savePoints(float x, float y, int n) {
  if (n == 1) {
          fprintf(fxpoints, "%lf %lf \n", x, y);
  }
  else {
          fprintf(fxpoints2, "%lf %lf \n", x, y);
  }
}

void RungeKutta::plot() {
  gnuplot = popen("gnuplot -persist", "w");
  // fprintf(gnuplot, "set title \"Fourth-order Runge-Kutta for y' = -2x^3 + 12x^2 -20x + 8.5\" font \"Times-Roman,14\"\n");
  // fprintf(gnuplot, "f(x) = %s\n", "-0.5*x**4 + 4*x**3 - 10 *x**2 + 8.5*x +1");
  fprintf(gnuplot, "set title \"Fourth-order Runge-Kutta for y' = g - c/m *y\" font \"Times-Roman,14\"\n");
  fprintf(gnuplot, "f(x) = (%f*%f)/%f * (1 - exp((-%f/%f)*x))\n", g, m, c, c, m);
  fprintf(gnuplot, "set xrange[%d:%d]\n", xi, xf);
  fprintf(gnuplot, "plot f(x) title \"real solution\", \"fxpoints.txt\" u 1:2 w l lc rgb '#0072bd' title \"h = %.1f\", " u 1:2 w p ls 7 ps 2 lc rgb '#0072bd' notitle\n", h);
  fclose(fxpoints);
}

void RungeKutta::replot() {
  fprintf(gnuplot, "replot \"fxpoints2.txt\" u 1:2 w l lc rgb '#DC143C' title \"h = %.1f\", " u 1:2 w p ls 7 ps 1.5 lc rgb '#DC143C' notitle\n", h);
  fclose(fxpoints2);
}

int main() {
  RungeKutta rungeKutta;
  rungeKutta.RKMethod(0.5, 1);
  return 0;
}
```

Compilación del programa usando g++

Se compila la cabecera del archivo:

```
g++  RungeKutta.hpp
```

Se compila el archivo fuente y se genera un ejecutable:

```
g++  RungeKutta.cpp -o RK1.x
```

Ejecución

Se ejecuta el archivo ejecutable ./RK1.x:

Resultados

Al ejecutar el programa se imprimen en cada iteración, los valores de x, y real, y obtenida por el método de Runge-Kutta de cuarto orden, y el error. Como se menciona al inicio del documento, se puede observar como lo único que cambia en estos métodos es la forma en que se calcula la pendiente ϕ, siendo el método de Euler, el método de Runge-Kutta de primer orden.

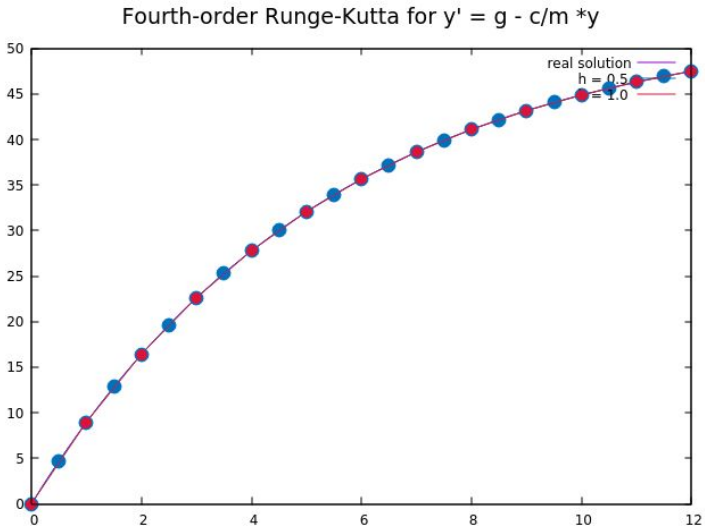

Figura 5. Comparación de la solución verdadera con una solución numérica usando el método de Runge-Kutta de cuarto orden, para la integral del problema del paracaidista desde $x = 0$ hasta $x = 12$ con tamaños de paso de 0.5 y 1 respectivamente. La condición inicial en $x = 0$ es $y = 0$

Versión con ambos métodos

EulerRK.hpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <iostream>
#include <iomanip>

#define xi 0
#define xf 12
#define yi 0
#define g 9.8
#define c 12.5
#define m 68.1

class EulerRK {
 private:
  float h;
  FILE *fxpoints;
```

```
  FILE *fxpoints2;
  FILE *gnuplot;

 public:
  EulerRK();
  ~EulerRK();

  float f(float x, float y);
  float real(float x);
  float Eulerformula(float x, float y);
  float RKfourth(float x, float y);
  void getValues(float step, int n);
  void EulerRKmethods(float step1, float step2);

  void printElement(float x, float yr, float y, float yRK, float e, float eRK);
  void printHeader();

  void savePoints(float x, float y, float yRK, int n);
  void plot();
  void replot();
};
```

EulerRK.cpp

```
// Copyright
#include "EulerRK.hpp"

EulerRK::EulerRK() {
  fxpoints = fopen("fxpoints.txt", "w");
  fxpoints2 = fopen("fxpoints2.txt", "w");
}

EulerRK::~EulerRK() {}

float EulerRK::f(float x, float y) {
  float fx;
  fx = g - (c/m)*y;
  //fx = -2*pow(x, 3) + 12*pow(x, 2) - 20*x + 8.5;
  return fx;
}

float EulerRK::real(float x) {
  float y;
  y = (g*m)/c * (1 - exp((-c/m)*x));
  // y = -0.5*pow(x, 4) + 4*pow(x, 3) - 10*pow(x,2) + 8.5*x +1;
  return y;
}

float EulerRK::Eulerformula(float x, float y) {
```

```
  float yk;
  yk = y + f(x, y)*h;
  return yk;
}

float EulerRK::RKfourth(float x, float y) {
  float yk, k1, k2, k3, k4;
  k1 = f(x, y);
  k2 = f(x + 0.5*h, y + 0.5*k1*h);
  k3 = f(x + 0.5*h, y + 0.5*k2*h);
  k4 = f(x + h, y + k3*h);
  yk = y + ((k1 + 2*k2 + 2*k3 + k4)/6)*h;
  return yk;
}

void EulerRK::getValues(float step, int n) {
  float x, y, yk, yRK, yreal, error, errorRK;
  h = step;
  x = xi;
  y = yi;
  yRK = yi;
  error = 0;
  std::cout << "First and Fourth order Runge-Kutta methods with step size: " << h << std::endl;
  printHeader();
  printElement(x, y, y, y, error, error);
  savePoints(x, y, yRK, n);
  while (x < xf) {
          yk = Eulerformula(x, y);
          y = yk;
          yk = RKfourth(x, yRK);
          yRK = yk;
          x += h;
          yreal = real(x);
          error = 100*fabs(yreal-y)/yreal;
          errorRK = 100*fabs(yreal-yRK)/yreal;
          printElement(x, yreal, y, yRK, error, errorRK);
          savePoints(x, y, yRK, n);
  }
  std::cout << std::endl;
  if (n == 1)
          plot();
  else
          replot();

}

void EulerRK::EulerRKmethods(float step1, float step2) {
  getValues(step1, 1);
```

```
  getValues(step2, 2);
}

void EulerRK::printElement(float x, float yr, float y, float yRK, float e, float eRK) {
  const int precision = 5;
  const int& width = 10;
  std::cout << '|' << std::setw(width) << std::setprecision(precision) << x << std::fixed << std::setw(width);
  std::cout << '|' << std::setw(width) << std::setprecision(precision) << yr << std::fixed << std::setw(width);
  std::cout << '|' << std::setw(width) << std::setprecision(precision) << y << std::fixed <<  std::setw(width);
  std::cout << '|' << std::setw(width) << std::setprecision(precision) << yRK << std::fixed <<
std::setw(width);
  std::cout << '|' << std::setw(width) << std::setprecision(precision) << e << std::fixed <<  std::setw(width);
  std::cout << '|' << std::setw(width) << std::setprecision(precision) << eRK << std::fixed <<
std::setw(width);
  std::cout << '|' << std::endl;
}

void EulerRK::printHeader() {
  const int& width = 10;
  std::cout << '|' << std::setw(width) << "x" << std::setw(width);
  std::cout << '|' << std::setw(width) << "yReal" << std::setw(width);
  std::cout << '|' << std::setw(width) << "yEuler" << std::setw(width);
  std::cout << '|' << std::setw(width) << "yRK" << std::setw(width);
  std::cout << '|' << std::setw(width) << "e Euler" << std::setw(width);
  std::cout << '|' << std::setw(width) << "e RK" << std::setw(width);
  std::cout << '|' << std::endl;
}

void EulerRK::savePoints(float x, float y, float yRK, int n) {
  if (n == 1) {
          fprintf(fxpoints, "%lf %lf %lf \n", x, y, yRK);
  }
  else {
          fprintf(fxpoints2, "%lf %lf %lf \n", x, y, yRK);
  }
}

void EulerRK::plot() {
  gnuplot = popen("gnuplot -persist", "w");
  // fprintf(gnuplot, "set title \"First and Fourth order RK Methods for y' = -2x^3 + 12x^2 -20x + 8.5\" font
\"Times-Roman,14\"\n");
  // fprintf(gnuplot, "f(x) = %s\n", "-0.5*x**4 + 4*x**3 - 10 *x**2 + 8.5*x +1");
  fprintf(gnuplot, "set title \"First and Fourth order RK Methods for y' = g - c/m *y\" font
\"Times-Roman,14\"\n");
  fprintf(gnuplot, "f(x) = (%f*%f)/%f * (1 - exp((-%f/%f)*x))\n", g, m, c, c, m);
  fprintf(gnuplot, "set xrange[%d:%d]\n", xi, xf);
```

```
  fprintf(gnuplot, "plot f(x) title \"real solution\", \"fxpoints.txt\" u 1:2 w l lc rgb '#DC143C' title \"Euler h =
%.1f\", '' u 1:2 w p ls 7 ps 2 lc rgb '#DC143C' notitle, '' u 1:3 w l lc rgb '#0072bd' title \"RK h = %.1f\", '' u
1:3 w p ls 7 ps 2 lc rgb '#0072bd' notitle\n", h, h);
  fclose(fxpoints);
}

void EulerRK::replot() {
  fprintf(gnuplot, "replot \"fxpoints2.txt\" u 1:2 w l lc rgb '#edb120' title \"Euler h = %.1f\", '' u 1:2 w p ls 7 ps
2 lc rgb '#edb120' notitle, '' u 1:3 w l lc rgb '#77ac30' title \"RK h = %.1f\", '' u 1:3 w p ls 7 ps 2 lc rgb
'#77ac30' notitle\n", h, h);
  fclose(fxpoints2);
}

int main() {
  EulerRK eulerRK;
  eulerRK.EulerRKmethods(0.5,1);
  return 0;
}
```

Compilación del programa usando g++

Se compila la cabecera del archivo:

```
g++  EulerRK.hpp
```

Se compila el archivo fuente y se genera un ejecutable:

```
g++  EulerRK.cpp -o EulerRK3.x
```

Ejecución

Se ejecuta el archivo ejecutable ./EulerRK3.x:

```
cynthia@taiga:~/Documents/Mate/Programas$ ./EulerRK3.x
```

Resultados

Al ejecutar el programa se imprime en cada iteración los valores de x, y real, y obtenida por el método de Euler, y obtenida por el método de Runge-Kutta de cuarto orden, el error obtenido por el método de Euler y el error obtenido por el método de RK de cuarto orden.

Es posible correr el programa con el tamaño de paso por separado. Al hacerlo se obtienen las siguientes figuras.

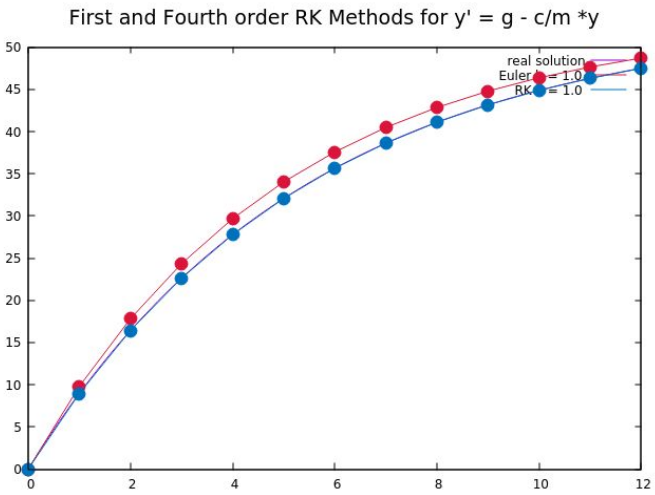

Figura 6. Gráfica resultante de la comparación realizada en la tabla 1

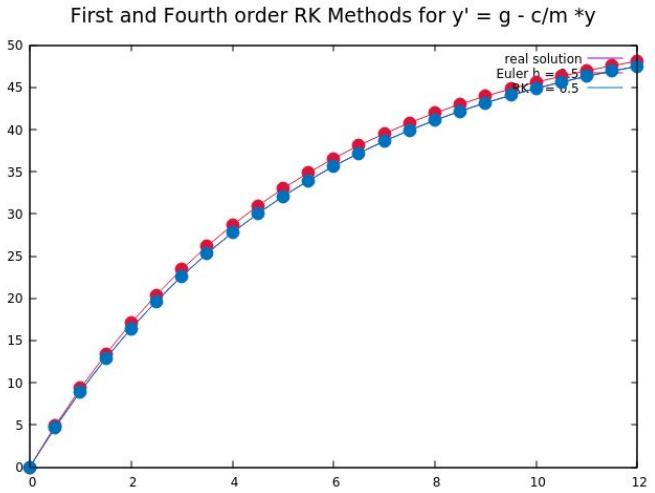

Figura 7. Gráfica resultante de la comparación realizada en la tabla 2

First and Fourth order Runge-Kutta methods with step size: 1.00000					
x	**yReal**	**yEuler**	**yRK**	**e Euler**	**e RK**
0.0	0.00000	0.00000	0.00000	0.00000	0.00000
1.0	8.95318	9.80000	8.95309	9.45829	0.00101
2.0	16.40498	17.80117	16.40483	8.51078	0.00092
3.0	22.60717	24.33370	22.60698	7.63713	0.00083
4.0	27.76929	29.66717	27.76908	6.83444	0.00075
5.0	32.06577	34.02165	32.06555	6.09961	0.00068
6.0	35.64175	37.57685	35.64153	5.42932	0.00061
7.0	38.61807	40.47949	38.61786	4.82006	0.00055
8.0	41.09528	42.84933	41.09508	4.26825	0.00049

9.0	43.15709	44.78418	43.15689	3.77018	0.00044
10.0	44.87314	46.36388	44.87296	3.32213	0.00039
11.0	46.30142	47.65363	46.30126	2.92044	0.00035
12.0	47.49019	48.70663	47.49005	2.56145	0.00031

Tabla 1. Comparación de los valores verdadero y aproximados de la integral del problema del paracaidista, con la condición de $v = 0$ en $t = 0$. Los valores aproximados se calcularon empleando el método de Euler y el método de Runge-Kutta de cuarto orden con un tamaño de paso de 1. El error es la discrepancia total debida a los pasos anterior y presentes.

First and Fourth order Runge-Kutta methods with step size: 0.5					
x	**yReal**	**yEuler**	**yRK**	**error(%) Euler**	**error(%) RK**
0.0	0.00000	0.00000	0.00000	0.00000	0.00000
0.5	4.68187	4.90000	4.68187	4.65903	0.00006
1.0	8.95318	9.35029	8.95318	4.43543	0.00006
1.5	12.84994	13.39215	12.84993	4.21961	0.00006
2.0	16.40498	17.06306	16.40497	4.01148	0.00006
2.5	19.64828	20.39707	19.64827	3.81098	0.00006
3.0	22.60717	23.42509	22.60715	3.61800	0.00005
3.5	25.30659	26.17521	25.30657	3.43242	0.00005
4.0	27.76929	28.67294	27.76928	3.25412	0.00005
4.5	30.01604	30.94143	30.01602	3.08298	0.00005
5.0	32.06577	33.00172	32.06575	2.91886	0.00005
5.5	33.93575	34.87293	33.93573	2.76163	0.00004
6.0	35.64175	36.57240	35.64174	2.61113	0.00004
6.5	37.19815	38.11591	37.19814	2.46720	0.00005
7.0	38.61807	39.51775	38.61805	2.32968	0.00005
7.5	39.91348	40.79094	39.91346	2.19841	0.00004
8.0	41.09528	41.94727	41.09527	2.07321	0.00004
8.5	42.17346	42.99749	42.17344	1.95390	0.00004
9.0	43.15709	43.95131	43.15707	1.84032	0.00004
9.5	44.05446	44.81760	44.05444	1.73228	0.00003
10.0	44.87314	45.60439	44.87312	1.62960	0.00003
10.5	45.62003	46.31897	45.62001	1.53209	0.00003
11.0	46.30142	46.96796	46.30141	1.43956	0.00003
11.5	46.92306	47.55739	46.92305	1.35185	0.00002
12.0	47.49019	48.09273	47.49018	1.26876	0.00002

Tabla 2. Comparación de los valores verdadero y aproximados de la integral del problema del paracaidista, con la condición de $v = 0$ en $t = 0$. Los valores aproximados se calcularon empleando el método de Euler y el método de Runge-Kutta de cuarto orden con un tamaño de paso de 0.5. El error es la discrepancia total debida a los pasos anterior y presentes.

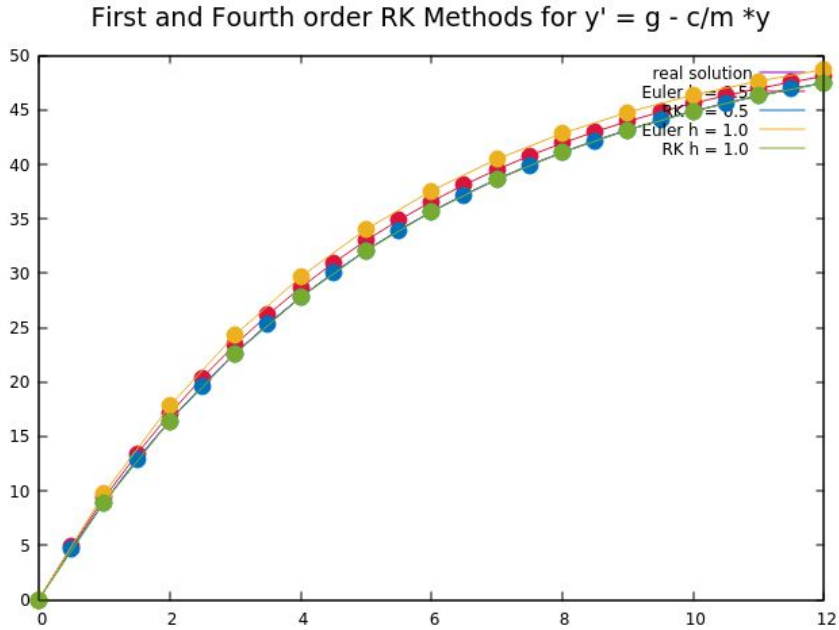

Figura 8. Comparación de la solución verdadera con una solución numérica usando el método de Euler y el método de Runge-Kutta de cuarto orden, para la integral del problema del paracaidista desde $x = 0$ hasta $x = 12$ con tamaños de paso de 0.5 y 1 respectivamente. La condición inicial en $x = 0$ es $y = 0$

2. Integrar numéricamente la ecuación $\frac{dy}{dx} = -2x^3 + 12x^2 - 20x + 8.5$ desde $x = 0$ hasta $x = 4$ con un tamaño de paso de 1 y 0.2. La condición inicial en $x = 0$ es $y = 1$.

La solución exacta está dada por la ecuación:

$$y = -0.5x^4 + 4x^3 - 10x^2 + 8.5x + 1$$

Método de Euler

Se utiliza el mismo programa, solo se cambia el valor de xf y yi en el .hpp y se descomentan y comentan un par de líneas de código en el .cpp

Euler.hpp

Se cambia el valor inicial de y y el valor final de x. Se modifican las líneas 8 y 9.

```
#define xf 4
#define yi 1
```

Euler.cpp

Se modifica la ecuación diferencial a integrar. Se comenta la línea 13 y se descomenta la línea 14.

```
float Euler::f(float x, float y) {
```

```
  float fx;
  // fx = g - (c/m)*y;
  fx = -2*pow(x, 3) + 12*pow(x, 2) - 20*x + 8.5;
  return fx;
}
```

Se modifica la solución real exacta. Se comenta la línea 20 y se descomenta la línea 21.

```
float Euler::real(float x) {
  float y;
  // y = (g*m)/c * (1 - exp((-c/m)*x));
  y = -0.5*pow(x, 4) + 4*pow(x, 3) - 10*pow(x, 2) + 8.5*x +1;
  return y;
}
```

Se modifica el título de la gráfica y la función a graficar. Se comentan las líneas 94 y 95 y se descomentan las líneas 92 y 93.

```
void Euler::plot() {
  gnuplot = popen("gnuplot -persist", "w");
  fprintf(gnuplot, "set title \"Euler's Method for y' = -2x^3 + 12x^2 -20x + 8.5\" font \"Times-Roman,14\"\n");
  fprintf(gnuplot, "f(x) = %s\n", "-0.5*x**4 + 4*x**3 - 10 *x**2 + 8.5*x +1");
  // fprintf(gnuplot, "set title \"Euler's Method for y' = g - c/m *y\" font \"Times-Roman,14\"\n");
  // fprintf(gnuplot, "f(x) = (%f*%f)/%f * (1 - exp((-%f/%f)*x))\n", g, m, c, c, m);
  fprintf(gnuplot, "set xrange[%d:%d]\n", xi, xf);
  fprintf(gnuplot, "plot f(x) title \"real solution\", \"fxpoints.txt\" u 1:2 w l lc rgb '#0072bd' title \"h = %.1f\", " u
1:2 w p ls 7 ps 2 lc rgb '#0072bd' notitle\n", h);
  fclose(fxpoints);
}
```

Se cambia el tamaño de paso. En la línea 108, se sustituye el 0.5 por 0.2.

```
int main() {
  Euler euler;
  euler.Eulermethod(0.2,1);
  return 0;
}
```

Método de Runge-Kutta de cuarto orden

Se utiliza el mismo programa, solo se realizan los mismo cambios que en el método de Euler.

RungeKutta.hpp

Se cambia el valor inicial de y y el valor final de x. Se modifican las líneas 8 y 9.

```
#define xf 4
#define yi 1
```

RungeKutta.cpp

Se modifica la ecuación diferencial a integrar. Se comenta la línea 13 y se descomenta la línea 14.

```
float RungeKutta::f(float x, float y) {
  float fx;
  // fx = g - (c/m)*y;
  fx = -2*pow(x, 3) + 12*pow(x, 2) - 20*x + 8.5;
  return fx;
}
```

Se modifica la solución real exacta. Se comenta la línea 20 y se descomenta la línea 21.

```
float RungeKutta::real(float x) {
  float y;
  // y = (g*m)/c * (1 - exp((-c/m)*x));
  y = -0.5*pow(x, 4) + 4*pow(x, 3) - 10*pow(x, 2) + 8.5*x +1;
  return y;
}
```

Se modifica el título de la gráfica y la función a graficar. Se comentan las líneas 98 y 99 y se descomentan las líneas 96 y 97.

```
void RungeKutta::plot() {
  gnuplot = popen("gnuplot -persist", "w");
    fprintf(gnuplot, "set title \"Fourth-order Runge-Kutta for y' = -2x^3 + 12x^2 -20x + 8.5\" font \"Times-Roman,14\"\n");
  fprintf(gnuplot, "f(x) = %s\n", "-0.5*x**4 + 4*x**3 - 10 *x**2 + 8.5*x +1");
  // fprintf(gnuplot, "set title \"Fourth-order Runge-Kutta for y' = g - c/m *y\" font \"Times-Roman,14\"\n");
  // fprintf(gnuplot, "f(x) = (%f*%f)/%f * (1 - exp((-%f/%f)*x))\n", g, m, c, c, m);
  fprintf(gnuplot, "set xrange[%d:%d]\n", xi, xf);
   fprintf(gnuplot, "plot f(x) title \"real solution\", \"fxpoints.txt\" u 1:2 w l lc rgb '#0072bd' title \"h = %.1f\", " u 1:2 w p ls 7 ps 2 lc rgb '#0072bd' notitle\n", h);
  fclose(fxpoints);
}
```

Se cambia el tamaño de paso. En la línea 112, se sustituye el 0.5 por 0.2.

```
int main() {
  RungeKutta rungeKutta;
  rungeKutta.RKMethod(0.2, 1);
  return 0;
}
```

Versión con ambos métodos

Se utiliza el mismo programa, solo se realizan los mismo cambios que en los casos anteriores.

EulerRK.hpp

Se cambia el valor inicial de y y el valor final de x. Se modifican las líneas 8 y 9.

```
#define xf 12
#define yi 0
```

EulerRK.cpp

Se modifica la ecuación diferencial a integrar. Se comenta la línea 13 y se descomenta la línea 14.

```
float EulerRK::f(float x, float y) {
  float fx;
  // fx = g - (c/m)*y;
  fx = -2*pow(x, 3) + 12*pow(x, 2) - 20*x + 8.5;
  return fx;
}
```

Se modifica la solución real exacta. Se comenta la línea 20 y se descomenta la línea 21.

```
float EulerRK::real(float x) {
  float y;
  // y = (g*m)/c * (1 - exp((-c/m)*x));
  y = -0.5*pow(x, 4) + 4*pow(x, 3) - 10*pow(x, 2) + 8.5*x +1;
  return y;
}
```

Se modifica el título de la gráfica y la función a graficar. Se comentan las líneas 113 y 114 y se descomentan las líneas 111 y 112.

```
void EulerRK::plot() {
  gnuplot = popen("gnuplot -persist", "w");
   fprintf(gnuplot, "set title \"First and Fourth order RK Methods for y' = -2x^3 + 12x^2 -20x + 8.5\" font \"Times-Roman,14\"\n");
  fprintf(gnuplot, "f(x) = %s\n", "-0.5*x**4 + 4*x**3 - 10 *x**2 + 8.5*x +1");
     // fprintf(gnuplot, "set title \"First and Fourth order RK Methods for y' = g - c/m *y\" font \"Times-Roman,14\"\n");
  // fprintf(gnuplot, "f(x) = (%f*%f)/%f * (1 - exp((-%f/%f)*x))\n", g, m, c, c, m);
  fprintf(gnuplot, "set xrange[%d:%d]\n", xi, xf);
   fprintf(gnuplot, "plot f(x) title \"real solution\", \"fxpoints.txt\" u 1:2 w l lc rgb '#DC143C' title \"Euler h = %.1f\", '' u 1:2 w p ls 7 ps 2 lc rgb '#DC143C' notitle, '' u 1:3 w l lc rgb '#0072bd' title \"RK h = %.1f\", '' u 1:3 w p ls 7 ps 2 lc rgb '#0072bd' notitle\n", h, h);
  fclose(fxpoints);
}
```

Se cambia el tamaño de paso. En la línea 127, se sustituye el 0.5 por 0.2.

```
int main() {
  EulerRK eulerRK;
  eulerRK.EulerRKmethods(0.2,1);
  return 0;
}
```

Resultados

Gráfica del método de Euler

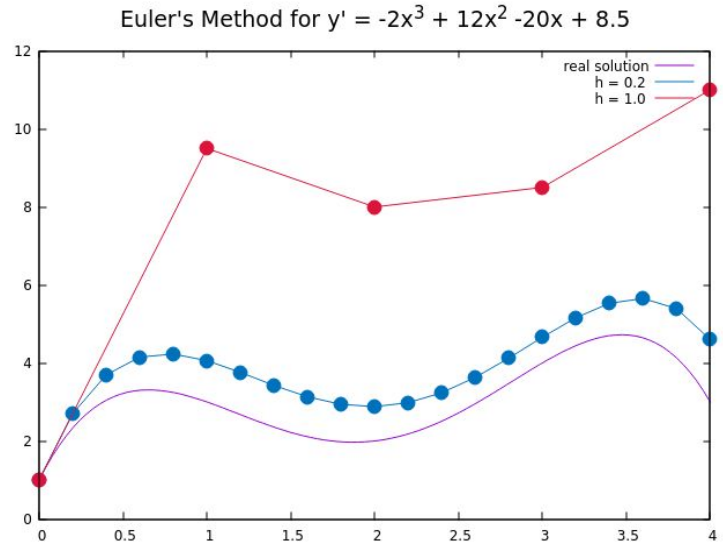

Figura 9. Comparación de la solución verdadera con una solución numérica usando el método de Euler, para la integral de $y' = -2x^3 + 12x^2 - 20x + 8.5$ desde $x = 0$ hasta $x = 4$ con tamaños de paso de 0.2 y 1 respectivamente. La condición inicial en $x = 0$ es $y = 1$

Gráfica del método de Runge-Kutta de cuarto orden

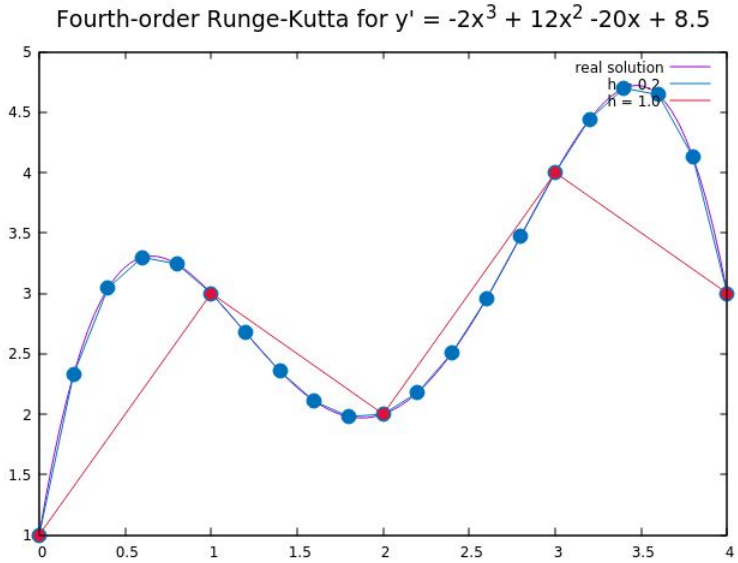

Figura 10. Comparación de la solución verdadera con una solución numérica usando el método de Runge-Kutta de cuarto orden, para la integral de $y' = -2x^3 + 12x^2 - 20x + 8.5$ desde $x = 0$ hasta $x = 4$ con tamaños de paso de 0.2 y 1 respectivamente. La condición inicial en $x = 0$ es $y = 1$

Versión con ambos métodos

Euler's method with step size: 1.00000					
x	**yReal**	**yEuler**	**yRK**	**error(%) Euler**	**error(%) RK**
0.0	1.00000	1.00000	1.00000	0.00000	0.00000
1.0	3.00000	9.50000	3.00000	216.66667	0.00000
2.0	2.00000	8.00000	2.00000	300.00000	0.00000
3.0	4.00000	8.50000	4.00000	112.50000	0.00000
4.0	3.00000	11.00000	3.00000	266.66666	0.00000

Tabla 3. Comparación de los valores verdadero y aproximados de la integral de $y' = -2x^3 + 12x^2 - 20x + 8.5$, con la condición de $y = 1$ en $x = 0$. Los valores aproximados se calcularon empleando el método de Euler y el método de Runge-Kutta de cuarto orden con un tamaño de paso de 1. El error es la discrepancia total debida a los pasos anterior y presentes.

Euler's method with step size: 0.2					
x	**yReal**	**yEuler**	**yRK**	**error(%) Euler**	**error(%) RK**
0.0	1.00000	1.00000	1.00000	0.00000	0.00000
0.2	2.33120	2.70000	2.33120	15.82017	0.00000
0.4	3.04320	3.69280	3.04320	21.34595	0.00001
0.6	3.29920	4.15120	3.29920	25.82445	0.00001
0.8	3.24320	4.22880	3.24320	30.38975	0.00001
1.0	3.00000	4.06000	3.00000	35.33335	0.00002
1.2	2.67520	3.76000	2.67520	40.55026	0.00002
1.4	2.35520	3.42480	2.35520	45.41444	0.00003
1.6	2.10720	3.13120	2.10720	48.59534	0.00003
1.8	1.97920	2.93680	1.97920	48.38323	0.00004
2.0	2.00000	2.88000	2.00000	44.00002	0.00004
2.2	2.17920	2.98000	2.17920	36.74746	0.00002
2.4	2.50720	3.23680	2.50720	29.10020	0.00001
2.6	2.95520	3.63120	2.95520	22.87494	0.00001
2.8	3.47520	4.12480	3.47520	18.69246	0.00000
3.0	4.00000	4.66000	4.00000	16.50002	0.00000
3.2	4.44320	5.16000	4.44320	16.13253	0.00000
3.4	4.69920	5.52880	4.69920	17.65411	0.00001
3.6	4.64320	5.65120	4.64320	21.70920	0.00001

3.8	4.13120	5.39280	4.13120	30.53839	0.00001
4.0	3.00000	4.60000	3.00000	53.33339	0.00003

Tabla 4. Comparación de los valores verdadero y aproximados de la integral de $y' = -2x^3 + 12x^2 - 20x + 8.5$, con la condición de $y = 1$ en $x = 0$. Los valores aproximados se calcularon empleando el método de Euler y el método de Runge-Kutta de cuarto orden con un tamaño de paso de 0.2. El error es la discrepancia total debida a los pasos anterior y presentes.

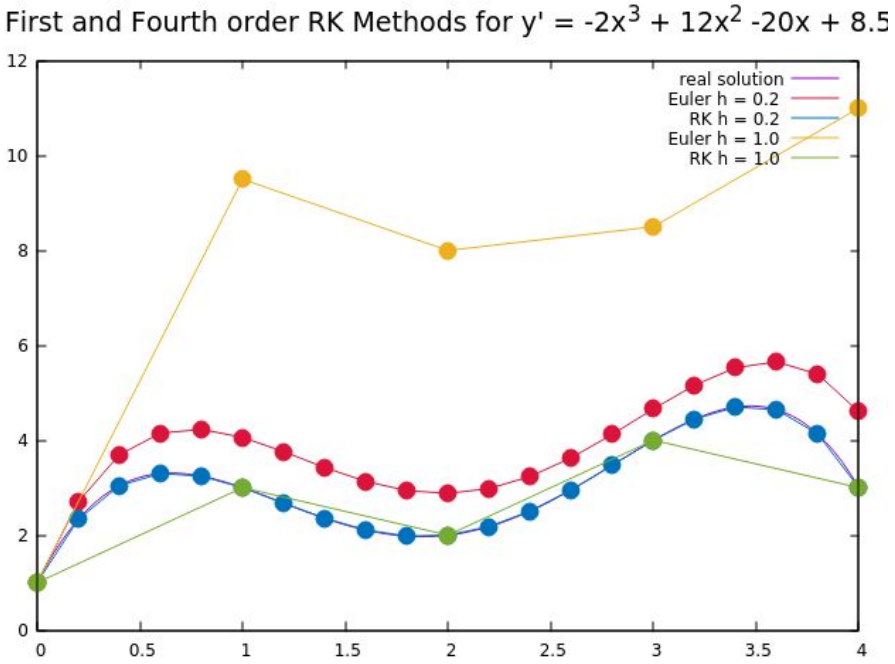

Figura 11. Comparación de la solución verdadera con una solución numérica usando el método de Euler y de Runge-Kutta de cuarto orden, para la integral de $y' = -2x^3 + 12x^2 - 20x + 8.5$ desde $x = 0$ hasta $x = 4$ con tamaños de paso de 0.2 y 1 respectivamente. La condición inicial en $x = 0$ es $y = 1$