PH 707: Assignment #4

August 25, 2022

Rajat Kumar Mandal

Roll No - 226121014

1 Fourth Order Runge Kutta Error Term.

The Runge Kutta approximation for $\int_{t_n}^{t_{n+1}=t_n+h} \overrightarrow{f}\left(\overrightarrow{x}(t), t\right) dt$ to solve the equation $\overrightarrow{x}(t) = \overrightarrow{f}\left(\overrightarrow{x}(t), t\right)$ is,

$$\overrightarrow{x}_{n+1} = \overrightarrow{x}_n + \frac{h}{6}\left(\overrightarrow{k}_1 + 2\overrightarrow{k}_2 + 2\overrightarrow{k}_3 + \overrightarrow{k}_4\right)$$

where,

$$\begin{aligned}
\overrightarrow{k}_1 &= \overrightarrow{f}\left(\overrightarrow{x}_n, t_n\right) \\
\overrightarrow{k}_2 &= \overrightarrow{f}\left(\overrightarrow{x}_n + \frac{h}{2}\overrightarrow{k}_1, t_n + \frac{h}{2}\right) \\
\overrightarrow{k}_3 &= \overrightarrow{f}\left(\overrightarrow{x}_n + \frac{h}{2}\overrightarrow{k}_2, t_n + \frac{h}{2}\right) \\
\overrightarrow{k}_4 &= \overrightarrow{f}\left(\overrightarrow{x}_n + \overrightarrow{k}_3, t_n + h\right).
\end{aligned}$$

Putting these together,

$$\begin{aligned}
\overrightarrow{x}_{n+1} = \overrightarrow{x}_n &+ \frac{h}{6}\overrightarrow{f}\left(\overrightarrow{x}_n, t_n\right) + \frac{h}{3}\overrightarrow{f}\left(\overrightarrow{x}_n + \frac{h}{2}\overrightarrow{f}\left(\overrightarrow{x}_n, t_n\right), t_n + \frac{h}{2}\right) \\
&+ \frac{h}{3}\overrightarrow{f}\left(\overrightarrow{x}_n + \frac{h}{2}\overrightarrow{f}\left(\overrightarrow{x}_n + \frac{h}{2}\overrightarrow{f}\left(\overrightarrow{x}_n, t_n\right), t_n + \frac{h}{2}\right), t_n + \frac{h}{2}\right) \\
&+ \frac{h}{6}\overrightarrow{f}\left(\overrightarrow{x}_n + \overrightarrow{f}\left(\overrightarrow{x}_n + \frac{h}{2}\overrightarrow{f}\left(\overrightarrow{x}_n + \frac{h}{2}\overrightarrow{f}\left(\overrightarrow{x}_n, t_n\right), t_n + \frac{h}{2}\right), t_n + \frac{h}{2}\right), t_n + h\right).
\end{aligned}$$

Now we can expand $E(h) = \int_{t_n}^{t_n+h} \overrightarrow{f}\left(\overrightarrow{x}(t), t\right) dt - \overrightarrow{x}_{n+1}(h)$ in a Taylor expansion of h up to 5th order in Mathematica. **The code and the results are given in the next page**. Note that the lowest order term is $o\left(h^5\right)$ so that the locally it is correct upto 4th order. Globally(in the whole problem range, not just in (t_n, t_{n+1})), one upper bound of the error is $O\left(h^4\right)$(in general difficult to prove for arbitrary differential equations and arbitrary ranges).

```
(*Definition of the exact integral and its RK4 approximation*)
K1[h_] := h f[tn, x[tn]]
K2[h_] := h f[tn + 1 / 2 h, x[tn] + 1 / 2 K1[h]]
K3[h_] := h f[tn + 1 / 2 h, x[tn] + 1 / 2 K2[h]]
K4[h_] := h f[tn + h, x[tn] + K3[h]]
RK4Approx[h_] := 1 / 6 (K1[h] + 2 K2[h] + 2 K3[h] + K4[h])
Exact[h_] := Integrate[ f[t, x[t]], {t, tn, tn + h} ]

(*We define y[t] so that higher order total derivatives of x[k]
 in terms of derivatives of f[t,x[t]] are automatically calculated*)
y[t_] := f[t, x[t]]
x'[t_] := y[t]
x''[t_] := y'[t]
x'''[t_] := y''[t]
x''''[t_] := y'''[t]
(*Simplify the difference of exact integral and RK4 approximation up to 5th order*)
FullSimplify[Series[Exact[h] - RK4Approx[h], {h, 0, 5}]]
```

Out[•]=

$$
\begin{aligned}
&\frac{1}{2880}\Big(-f[tn,x[tn]]^4 f^{(0,4)}[tn,x[tn]] + 24 f^{(0,1)}[tn,x[tn]]^3 f^{(1,0)}[tn,x[tn]] + f[tn,x[tn]]^3 \\
&\quad \big(6 f^{(0,2)}[tn,x[tn]]^2 - 2 f^{(0,1)}[tn,x[tn]] f^{(0,3)}[tn,x[tn]] - 4 f^{(1,3)}[tn,x[tn]]\big) - \\
&\quad 6 f^{(0,1)}[tn,x[tn]]^2 f^{(2,0)}[tn,x[tn]] + 6 f^{(1,1)}[tn,x[tn]] f^{(2,0)}[tn,x[tn]] - \\
&\quad 6 f^{(1,0)}[tn,x[tn]] \big(3 f^{(0,2)}[tn,x[tn]] f^{(1,0)}[tn,x[tn]] + f^{(2,1)}[tn,x[tn]]\big) - \\
&\quad 6 f[tn,x[tn]]^2 \big(f^{(0,3)}[tn,x[tn]] f^{(1,0)}[tn,x[tn]] + \\
&\qquad 3 f^{(0,2)}[tn,x[tn]] \big(2 f^{(0,1)}[tn,x[tn]]^2 - f^{(1,1)}[tn,x[tn]]\big) + f^{(2,2)}[tn,x[tn]]\big) + \\
&\quad 4 f^{(0,1)}[tn,x[tn]] \big(-3 f^{(1,0)}[tn,x[tn]] f^{(1,1)}[tn,x[tn]] + f^{(3,0)}[tn,x[tn]]\big) + \\
&\quad 2 f[tn,x[tn]] \\
&\qquad \big(3 \big(4 f^{(0,1)}[tn,x[tn]]^4 - 4 f^{(0,1)}[tn,x[tn]]^2 f^{(1,1)}[tn,x[tn]] + 2 f^{(1,1)}[tn,x[tn]]^2 - \\
&\qquad\quad 2 f^{(1,0)}[tn,x[tn]] f^{(1,2)}[tn,x[tn]] + f^{(0,2)}[tn,x[tn]] f^{(2,0)}[tn,x[tn]] + \\
&\qquad\quad f^{(0,1)}[tn,x[tn]] \big(-8 f^{(0,2)}[tn,x[tn]] f^{(1,0)}[tn,x[tn]] + f^{(2,1)}[tn,x[tn]]\big)\big) - \\
&\qquad 2 f^{(3,1)}[tn,x[tn]]\big) - f^{(4,0)}[tn,x[tn]]\Big) h^5 + O[h]^6
\end{aligned}
$$

Figure 1: Cleaner Mathematica code for the lowest order error term in RK4 approximation

2 Physical Pendulum Using Runge Kutta 4th Order.

In the following pages, the codes and the results are presented in the following order:

1. Physical pendulum solution for stepsize $\frac{2\pi-0}{1000} = 0.00628319$ in the range $(0, 2\pi)$ and comparison with Taylor approximation.

2. The solution for various step-sizes and the optimal step size, by inspection.

3. A better, easier, less computationally expensive and very well-known method of controlling for the step-sizes in RK, an Adaptive Runge-Kutta method (Euler-Heun) applied to the physical pendulum.

It helps to summarize the method using the Butcher tableaux as follows:

$$
\begin{array}{c|cccc}
0 & & & & \\
1/2 & 1/2 & & & \\
1/2 & 0 & 1/2 & & \\
1 & 0 & 0 & 1 & \\
\hline
 & 1/6 & 1/3 & 1/3 & 1/6
\end{array}
\qquad
\begin{array}{c|cc}
0 & & \\
1 & 1 & \\
\hline
 & 1/2 & 1/2 \\
 & 1 & 0
\end{array}
$$

Figure 2: Butcher Tables for 4th order Runge Kutta and Adaptive Runge Kutta of order 1(2) (Euler-Heun)

C++ code for RK4 method


```cpp
#include <iostream>
#include <fstream>
#include <cmath>
#include <array>
#include <map>

//Constant expressions appearing in the problem
constexpr size_t dimension = 2;   //dimension of the reduced 1st-order problem
constexpr double PI = 3.14159265359;    //value of PI
constexpr double rollnum = 0.226121014;  //my roll number

//Definition of data types in the problem
typedef std::array<double, dimension> state_type;  //data type definition for dependant variables - array of x_0, x_1, ... x_n
typedef std::map<double, state_type> solution;   //data type definition for storing the list of calculated values ((hash)map of time -> state)

//Overload the + operator to be able to add two vectors
state_type operator + (state_type const &x, state_type const &y) {
    state_type z;
    for (size_t i = 0; i < dimension; i++) {
        z[i] = x[i] + y[i]; //add the individual components and store in z
    }
    return z;   //return the resulting vector z
}

//Overload the * operator to be able to multiply numbers and vectors
state_type operator * (double const &a, state_type const &x) {
    state_type z;
    for (size_t i = 0; i < dimension; i++) {
        z[i] = a * x[i];    //multiply the individual components and store in z
    }
    return z;   //return the resulting vector z
}

//This is the differential Equation, reduced to first-order
void Pendulum(const state_type& x, const double& t, state_type& dxdt){
    dxdt[0] = x[1];
    dxdt[1] = -4.0 * PI * PI * sin(x[0]);
}

//The stepper function, iteratively calculates x_{n+1} given the differential equation, x_{n} and step size
void rk4_step(void (*Diff_Equation)(const state_type& x, const double& t, state_type& dxdt), state_type& x, const double& t, const double& dt){
    //temporary variables for intermediate steps
    state_type k1, k2, k3, k4;
```

```cpp

    //calculate the intermediate values
    Diff_Equation(x, t, k1);    //calculate k1
    Diff_Equation(x + (dt / 2.0) * k1, t + dt / 2.0, k2);   //calculate k2
    Diff_Equation(x + (dt / 2.0) * k2, t + dt / 2.0, k3);   //calculate k3
    Diff_Equation(x + dt * k3, t + dt, k4); //calculate k4

    //calculate x_{n+1} using the RK4 formula and return the results
    x = x + (dt / 6.0) * (k1 + 2 * k2 + 2 * k3 + k4);
}

int main(){
    solution x_t;   //variable to store the calculations

    size_t STEPS = 1000;  //number of steps
    double t_0 = 0.0;   //initial time
    double t_1 = 1.0;   //final time
    double dt = (t_1 - t_0) / (STEPS - 1); //step size
    state_type x = {0.0, rollnum};   //initial values for dependant variables

    //Step through the domain of the problem and store the solutions
    x_t[t_0] = x;   //store initial values
    for (size_t i = 0; i < STEPS; i++) {
        rk4_step(Pendulum, x, NULL, dt);    //step forward
        x_t[t_0 + i * dt] = x;  //store the calculation
    }

    std::ofstream outfile;  //file handle to save the results in a file
    outfile.open("rk4.txt", std::ios::out | std::ios::trunc );
    for (auto const& temp : x_t){
        outfile << temp.first << "\t" << temp.second[0] << "\t" << temp.second[1] << std::endl;
    }
    outfile.close();
}
```

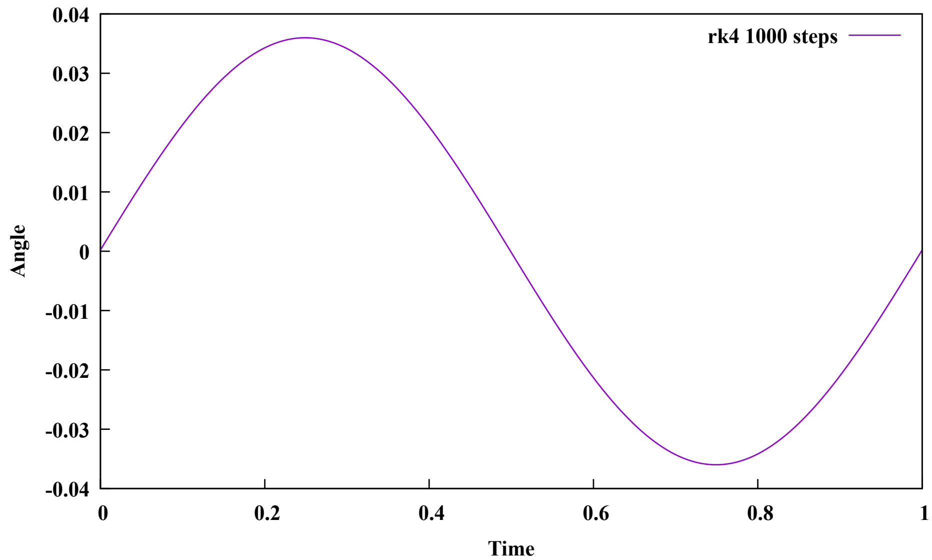

Figure 3: RK4 with 1000 steps for equation $x''(t) = -4\pi^2 \sin x(t)$, $x(0) = 0$, $x'(0) = 0.226121014$(my roll number)

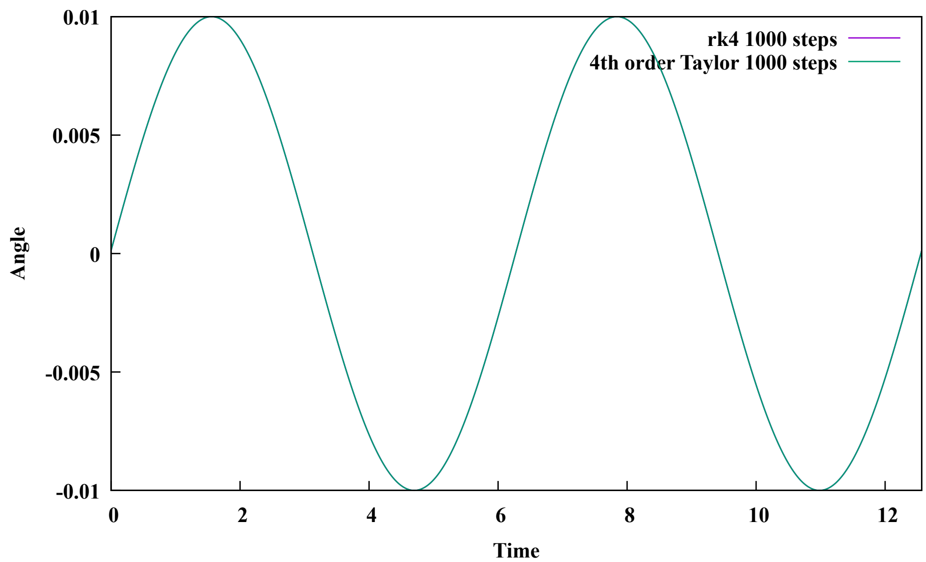

Figure 4: Comparison with 4th order Taylor approximation with 1000 steps for equation $x''(t) = -\sin x(t)$, $x(0) = 0$, $x'(0) = 0.01$

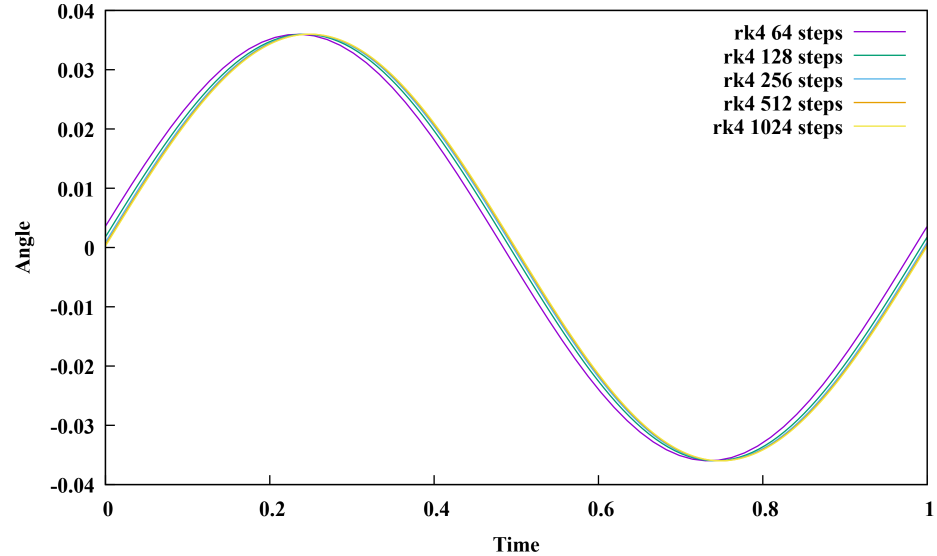

Figure 5: Comparison for different stepsizes for equation $x''(t) = -4\pi^2 \sin x(t)$, $x(0) = 0$, $x'(0) = 0.226121014$(my roll number), 256 steps (stepsize = 1/255) onwards we get almost no change.

C++ code for Adaptive RK1(2) using Butcher Tableau
Since everything is templated, it is easy to extend this to higher order by just supplying the proper Butcher tableau.


```cpp
#include <iostream>
#include <fstream>
#include <cmath>
#include <array>
#include <map>

//Constant expressions appearing in the problem
constexpr size_t dimension = 2;   //dimension of the reduced 1st-order problem
constexpr double PI = 3.14159265359;    //value of PI
constexpr double rollnum = 0.226121014;  //my roll number
constexpr double initial_stepsize = 0.01;  //initial step-size
constexpr size_t max_iter = 10; //maximum number of iterations in adapting stepsize

//Definition of data types in the problem
typedef std::array<double, dimension> state_type;   //data type definition for dependant variables - array of x_0, x_1, ... x_n
typedef std::map<double, state_type> solution;  //data type definition for storing the list of calculated values ((hash)map of time -> state)

//Overload the + operator to be able to add two vectors
state_type operator + (state_type const& x, state_type const& y) {
    state_type z;
    for (size_t i = 0; i < dimension; i++) {
        z[i] = x[i] + y[i]; //add the individual components and store in z
    }
    return z;   //return the resulting vector z
}

//Overload the * operator to be able to multiply numbers and vectors
state_type operator * (double const& a, state_type const& x) {
    state_type z;
    for (size_t i = 0; i < dimension; i++) {
        z[i] = a * x[i];    //multiply the individual components and store in z
    }
    return z;   //return the resulting vector z
}

//Overload the + operator to be able to add two vectors
double absdiff(state_type const& x) {
    double result = 0;
    for (size_t i = 0; i < dimension; i++) {
        result += x[i] * x[i]; //add the individual components and store in z
    }
    return sqrt(result);   //return the resulting vector z
}
```

```cpp

//Class template for the Runge Kutta solver using Butcher tableau
template <class State_Type, size_t order> class explicit_rk {
    //data type definnitions for storing the Butcher tableau
    typedef std::array<double, order> butcher_coefficients;
    typedef std::array<std::array<double, order>, order> butcher_matrix;
private:
    //information about the Butcher tableau
    butcher_matrix a;
    butcher_coefficients bh, bt, c;
    //temporary variables for intermediate steps
    std::array<State_Type, order> k;
    //Properties of the adaptive method
    double tolerance;
    size_t max_iters;

    //The stepper function, calculates x_{n+1} given the differential equation, x_{n}, t and step size
    void stepper(void (*Diff_Equation)(const State_Type& x, const double& t, State_Type& dxdt), const State_Type& x, const double& t, const double& dt, State_Type& result, double& error) {
        State_Type res = x;  //temporary variable for storing the result
        State_Type err = {};  //temporary variable for storing the result

        //loops for evaluating k1, k2 ... k_n
        for (size_t i = 0; i < order; i++) {
            State_Type sum{}, dxdt; //temporary variables for k's and the derivatives
            for (size_t j = 0; j < i; j++) {
                sum = sum + dt * a[i][j] * k[j];    //compute a_{ij} * k_j
            }
            sum = x + sum;  //compute x_{n} + a_{ij} * k_j
            Diff_Equation(sum, t + c[i] * dt, dxdt);    //evaluate dx/dt at (x_{n} + a_{ij} * k_j, t_{n} + c_{i} * dt) according to Runge Kutta
            k[i] = dxdt;    //store the dx/dt as k_i
        }

        //loop for calculating x_{n+1} using the k's
        for (size_t i = 0; i < order; i++) {
            res = res + dt * bh[i] * k[i]; //weighted average of k's with b's as weights
        }

        //loop for calculating x_{n+1} using the k's
        for (size_t i = 0; i < order; i++) {
            err = err + dt * bt[i] * k[i]; //weighted average of k's with b's as weights
        }
```

```

        //return the result
        result = res;
        error = absdiff(err);
    }
public:
    //Constructor - just copy the Butcher tableau
    explicit_rk(butcher_matrix A, butcher_coefficients BH, butcher_coefficients BT, butcher_coefficients C, double Tolerance, size_t Max_iters) : a(A), bh(BH), bt(BT), c(C), tolerance(Tolerance), max_iters(Max_iters) {
        k = {};    //zero-initialize k
    }

    //Destructor - nothong to do
    ~explicit_rk() {

    }

    //The stepper function, calculates x_{n+1} given the differential equation, x_{n}, t and step size
    void do_step(void (*Diff_Equation)(const State_Type& x, const double& t, State_Type& dxdt), State_Type& x, double& t, double& dt) {
        State_Type result = {};  //temporary variable for storing the result
        double error = 1.0e6;
        size_t numiter = 0;
        double h = dt;

        while (error > tolerance && numiter < max_iters) {
            dt = h;
            stepper(Diff_Equation, x, t, dt, result, error);
            h = dt * pow(tolerance / error, 1.0 / 2.0);
            numiter++;
        }

        t = t + dt;
        dt = h;
        x = result;
    }
};

//This is the differential Equation, reduced to first-order
void Pendulum(const state_type& x, const double& t, state_type& dxdt) {
    dxdt[0] = x[1];
    dxdt[1] = -4.0 * PI * PI * sin(x[0]);
}

int main() {
```

```cpp
    //Using the class template, creates a class object for the Runge Kutta solver with the butcher tableau of Runge Kutta 1(2) also known as Euler-Heun
    explicit_rk <state_type, 2> rk12_stepper(
        {
            0.0,    0.0,
            1.0,    0.0
        },

        { 0.5, 0.5 },    //Butcher bh coefficiants

        {0.5, -0.5 },    //Butcher bt coefficiants

        { 0.0, 1.0 },   //Butcher c coefficients

        0.001, max_iter);   //tolerance and maximum number of step recalculation at each step

    solution x_t;   //variable to store the calculations

    double t_0 = 0.0;   //initial time
    double t_1 = 1.0;   //final time
    double t = t_0; //time variable
    double dt = initial_stepsize; //step size(adaptive)
    state_type x = { 0.0, rollnum };   //initial values for dependant variables

    //Step through the domain of the problem and store the solutions
    x_t[t_0] = x;   //store initial values
    while (t < t_1) {
        rk12_stepper.do_step(Pendulum, x, t, dt);
        x_t[t] = x;
    }


    std::ofstream outfile;  //file handle to save the results in a file
    outfile.open("rk1(2) Euler-Heun.txt", std::ios::out | std::ios::trunc);
    for (auto const& temp : x_t) {
        outfile << temp.first << "\t" << temp.second[0] << "\t" << temp.second[1] << std::endl;
    }
    outfile.close();
}
```

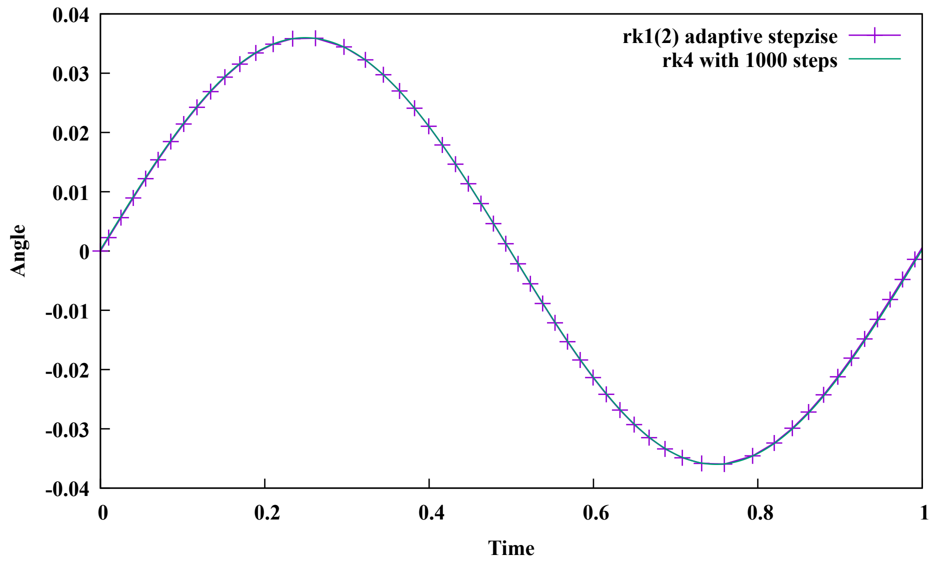

Figure 6: Comparison between RK4 with 1000 steps and adaptive Runge Kutta method of order 1(2) (Euler-Heun) for equation $x''(t) = -4\pi^2 \sin x(t)$, $x(0) = 0$, $x'(0) = 0.226121014$(my roll number)