```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
        }
```

```cpp

            //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() {
```

```
    //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();
}
```