```
/* This is a routine to test the forward Euler method for a scalar valued ODE.
     y' = f(t,y), t in [0,5],
     y(0) = y0.
   It is based off of Dan Reynold's driver_fwd_euler and uses his suite of
   forward Euler solvers.


   Nicole Deyerl
   Math 6321
   Fall 2016  */

#include <iostream>
#include <vector>
#include "matrix.hpp"
#include "rhs.hpp"
#include "fwd_euler.hpp"

using namespace std;


// Problem 1:  y' = -e^(-t)*y, y(0)=1
//    ODE RHS function class -- instantiates a RHSFunction
class RHS1: public RHSFunction {
public:
  int Evaluate(double t, vector<double>& y, vector<double>& f) {    // evaluates the RHS funct
ion, f(t,y)
    f = -exp(-t)*y;
    return 0;
  }
};
// Convenience function for analytical solution
//   y(t) = e^(e^(-t)-1)
vector<double> ytrue1(const double t) {
  vector<double> yt(1);
  yt[0] = exp(exp(-t)-1);
  return yt;
};



// main routine
int main() {

  // time steps to try
  vector<double> h = {0.2,0.1,0.05,0.025,0.0125}; //specified by
                                                   // problem 1a

  // set problem information
  vector<double> y0_1 = {1.0}; //initial condition y(0) = 1
  double t0 = 0.0;
  double Tf = 5.0; //problem specified t in [0,5]
  double tcur = t0;
  double dtout = 1.0;//problem specified output of soln and abs err every
                   // 1 unit of time
  double convg = 0.0; //initialize order of convergence
  double diff;
  // create ODE RHS function objects
  RHS1 f1;
  vector<double> maxerr1 (h.size(),0.0); //preallocate space to save all maxerrors
                                        // for each step size h, populate w/ 0's
```

```
  // create forward Euler stepper object
  ForwardEulerStepper FE1(f1, y0_1);


  // loop over time step sizes
  for (int ih=0; ih<h.size(); ih++) {

    // problem 1:
    vector<double> y = y0_1;
    tcur = t0;
    double maxerr = 0.0;

    cout << "\nRunning problem 1 with stepsize h = " << h[ih] << ":\n";

    //   loop over output step sizes: call solver and output error
    while (tcur < 0.99999*Tf) {

      // set the time interval for this solve
      vector<double> tspan = {tcur, std::min(tcur + dtout, Tf)};

      // call the solver, update current time
      vector<double> tvals = FE1.Evolve(tspan, h[ih], y);
      tcur = tvals.back();   // last entry in tvals

      // compute the error at tcur, output to screen and accumulate maximum
      vector<double> yerr = y - ytrue1(tcur);
      double err = InfNorm(yerr); //absolute error = diff between num + true soln
                                  // (InfNorm -> gives maximal entry)
      maxerr1[ih] = err; //keep the errors to calculate convergence
      maxerr = std::max(maxerr, err); //keep maximal error value
      // soln + error print out
      cout << "  y(" << tcur << ") = " << y[0]
           << "  \t||error|| = " << err
           << endl;
    }
    cout << "Max error = " << maxerr << endl;
    // compute convergence + print
    if(ih > 0){
      convg = (log(maxerr1[ih])-log(maxerr1[ih-1]))/(log(h[ih])-log(h[ih-1])); //soln of err =
 h^p for p
      cout << "The order of convergence is = " << convg << endl;
    }
  }


  return 0;
}
```

```
/* Forward Euler time stepper class header file.

   D.R. Reynolds
   Math 6321 @ SMU
   Fall 2016  */

#ifndef FORWARD_EULER_DEFINED__
#define FORWARD_EULER_DEFINED__

// Inclusions
#include <vector>
#include <math.h>
#include "matrix.hpp"
#include "rhs.hpp"


// Forward Euler time stepper class
class AdaptEuler {

 private:

  // private reusable local data
  std::vector<double> f;   // storage for ODE RHS vector
  RHSFunction *frhs;       // pointer to ODE RHS function
  double r; //storage for rtol
  double a; //storage for atol

 public:

  int fcalls = 0; //make calls to f part of the class so we can call it from
                  // the driver

  // constructor (sets RHS function pointer, copies y for local data)
  AdaptEuler(RHSFunction& frhs_, double rtol, double atol, std::vector<double>& y) {
    frhs = &frhs_;
    r = rtol; //from .cpp, can call rtol and atol from r and a
    a = atol;
    f = y;
  };

  // Evolve routine (evolves the solution via forward Euler)
  std::vector<double> Evolve(std::vector<double> tspan, std::vector<double>& y);

};

#endif
```

```
/* Forward Euler time stepper class implementation file.

   Class to perform time evolution of the IVP
        y' = f(t,y),  t in [t0, Tf],  y(t0) = y0
   using the forward Euler (explicit Euler) time stepping method.

   D.R. Reynolds
   Math 6321 @ SMU
   Fall 2016 */

#include <vector>
#include "matrix.hpp"
#include "adapt_euler.hpp"


// The forward Euler time step evolution routine
//
// Inputs:  tspan holds the current time interval, [t0, tf]
//          h holds the desired time step size
//          y holds the initial condition, y(t0)
// Outputs: y holds the computed solution, y(tf)
//
// The return value is a row vector containing all internal
// times at which the solution was computed,
//               [t0, t1, ..., tN]
std::vector<double> AdaptEuler::Evolve(std::vector<double> tspan, std::vector<double>& y) {

  // initialize output
  std::vector<double> times = {tspan[0]};

  // check for legal inputs
  if (tspan[1] <= tspan[0]) {
    std::cerr << "AdaptEuler: Illegal tspan\n";
    return times;
  }

  // figure out how many time steps
  long int N = pow(10,6);
  double h = (tspan[1]-tspan[0])/(tspan[1]*8); //init h should be func of tspan
  double err = 0.0;                            // in case tspan is v. small
  double relerr, abserr;
  double th, th2;
  int counter = 0; //counter for index of time steps
  std::vector<double> yh = y;
  std::vector<double> yh2 = y;

  // iterate over time steps
  while (times[counter]<0.99999*tspan[1]) {

    // last step only: update h to stop directly at final time
    if (times[counter]+h>0.99999*tspan[1]) {
      h = tspan[1]-times[counter];
    }

        //set up times for evals (at half and full step)
    th = times[counter] + h;
    th2 = times[counter] + h/2.0;

    //eval the full step
    if (frhs->Evaluate(th, y, f) != 0) {
      std::cerr << "AdaptEuler: Error in ODE RHS function\n";
```

```
      return times;
    }

    //1 full euler step, 1 half euler step
    yh = y + h*f; //1 full step
    yh2 = y + (h/2.0)*f; //2 half steps

    //eval the half step
    if (frhs->Evaluate(th2, yh2, f) != 0) {
      std::cerr << "AdaptEuler: Error in ODE RHS function\n";
      return times;
    }

    //second half euler step
    yh2 += (h/2.0)*f;

    //richardson error estimate
    err = Norm(2.0*yh2 - 2.0*yh);

    //update check
    if(err <= r*Norm(y) + a){
      y = 2.0*yh2 - yh; //richardson euler formula
          counter = counter + 1; //update counter
      times.push_back(th); //update current time, store in output array
      fcalls += 2; //update number of calls to f
        } else {
      h = h*((r*Norm(y) + a)/err); //reduce step size + try again
          fcalls += 2; //update number of calls to f
        }

    if(counter >= pow(10,6)){ //if the number of time steps exceeds 10^6
          break;                // exit the solver
        }

  }
  return times;
}
```

```
/* This is a routine to test the forward Euler method for a scalar valued ODE.
     y' = f(t,y), t in [0,5],
     y(0) = y0.
   It is based off of Dan Reynold's driver_fwd_euler and uses his suite of
   forward Euler solvers.


   Nicole Deyerl
   Math 6321
   Fall 2016  */

#include <iostream>
#include <vector>
#include "matrix.hpp"
#include "rhs.hpp"
#include "adapt_euler.hpp"

using namespace std;


// Problem 1:  y' = -e^(-t)*y, y(0)=1
// same as prob1.cpp
//    ODE RHS function class -- instantiates a RHSFunction
class RHS1: public RHSFunction {
public:
  int Evaluate(double t, vector<double>& y, vector<double>& f) {    // evaluates the RHS funct
ion, f(t,y)
    f = -exp(-t)*y;
    return 0;
  }
};
// Convenience function for analytical solution
//   y(t) = e^(e^(-t)-1)
vector<double> ytrue1(const double t) {
  vector<double> yt(1);
  yt[0] = exp(exp(-t)-1);
  return yt;
};



// main routine
int main() {

  // tolerances to try
  vector<double> rtol = {pow(10,-2),pow(10,-4),pow(10,-6),pow(10,-8)};
  double atol = pow(10,-11);

  // set problem information
  vector<double> y0_1 = {1.0}; //initial condition y(0) = 1
  double t0 = 0.0;
  double Tf = 5.0; //problem specified t in [0,5]
  double tcur = t0;
  double dtout = 1.0;//problem specified output of soln and errors every
                      // 1 unit of time
  double diff;
  // create ODE RHS function objects
  RHS1 f1;

  // loop over time step sizes
```

```cpp
  for (int ir=0; ir<rtol.size(); ir++) {

    // create an adaptive Euler stepper object for each rtol
    AdaptEuler FE1(f1, rtol[ir], atol, y0_1);

    // problem 1:
    vector<double> y = y0_1;
    tcur = t0; //initialize time
    double maxabserr = 0.0; //initialize abs and rel errors (max and current)
    double maxrelerr = 0.0;
    double relerr = 0.0;
    double abserr = 0.0;
    int timeind = 0; //counter for number of timesteps
    cout << "\nRunning problem 1 with rtol = " << rtol[ir] << " atol = " << atol << ":\n";

    //   loop over output step sizes: call solver and output error
    while (tcur < 0.9999*Tf) {

      // set the time interval for this solve
      vector<double> tspan = {tcur, std::min(tcur + dtout, Tf)};

      // call the solver, update current time
      vector<double> tvals = FE1.Evolve(tspan, y);
      tcur = tvals.back();   // last entry in tvals

      // compute the error at tcur, output to screen and accumulate maximum
      vector<double> yerr = y - ytrue1(tcur);
      double abserr = InfNorm(yerr); //abs error = norm of error between num + true soln
      double relerr = InfNorm(yerr)/InfNorm(y); //rel error = abs err / magnitude of soln
      maxabserr = std::max(maxabserr, abserr); //keep the maximal error values
      maxrelerr = std::max(maxrelerr, relerr);

      // soln + error print out
      cout << "  y(" << tcur << ") = " << y[0]
           << "  \t||abs. error|| = " << abserr << "  \t||rel. error|| = " << relerr
           << endl;
      timeind += tvals.size(); //remember number of time steps
    }
    cout << "Number of calls to f = " << FE1.fcalls << std::endl;
    cout << "Total Number of time steps = " << timeind << std::endl;
    cout << "Max absolute error = " << maxabserr
         << "  Max relative error = " << maxrelerr << endl;

  }


  return 0;
}
```