```
# Makefile for homework 2
#
# Daniel R. Reynolds
# Math 6321 @ SMU
# Fall 2016

# compiler & flags
CXX = g++
CXXFLAGS = -O -std=c++11
#CXXFLAGS = -O0 -g -std=c++11

# executable targets
all: prob2.exe prob4.exe

prob2.exe : prob2.cpp erk4.cpp ab3.cpp matrix.cpp
	$(CXX) $(CXXFLAGS) -o $@ $^

prob4.exe : prob4.cpp adapt_rkf.cpp adapt_euler.cpp matrix.cpp
	$(CXX) $(CXXFLAGS) -o $@ $^

# utilities
clean :
	\rm -rf *.txt *.png *.exe *~
```

```
/* Explicit 4th order Adams-Bashforth time stepper class header file.
 * Based on Dan Reynolds ab1.hpp file.

   Nicole Deyerl
   Math 6321 @ SMU
   Fall 2016  */

#ifndef AB3_DEFINED__
#define AB3_DEFINED__

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


// Explicit AB3 time stepper class
class AB3Stepper {

 private:

  RHSFunction *frhs;             // pointer to ODE RHS function
  std::vector<double> f, f1, f2, f3;  // reused vectors

 public:

  // constructor (sets RHS function pointer, allocates local data)
  AB3Stepper(RHSFunction& frhs_, std::vector<double>& y) {
    frhs = &frhs_;      // store RHSFunction pointer
    f = y;              // allocate reusable data
    f1 = y;             // based on size of y, holds old f (f at yn-1)
    f2 = y;             // holds older f (f at yn-2)
    f3 = y;             // holds oldest f (f at yn-3)
  };

  // Evolve routine (evolves the solution)
  std::vector<double> Evolve(std::vector<double>& tspan,
                             double h,
                 std::vector<double>& y,
                             std::vector<double>& y1,
                             std::vector<double>& y2,
                             std::vector<double>& y3);

};

#endif
```

```
/* Explicit 4th-order Adams-Bashforth time stepper class implementation file.
 * Based on Dan Reynolds ab1.cpp file.

   Nicole Deyerl
   Math 6321 @ SMU
   Fall 2016  */

#include <vector>
#include "matrix.hpp"
#include "ab3.hpp"


// The explicit 4th-order Adams-Bashforth 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) or y(n)
//          y1 holds the previous initial condition, y(t0-h) or y(n-1)
//          y2 holds y(t0-2*h) or y(n-2)
//          y3 holds y(t0-3*h) or y(n-3)
// Outputs: y holds the computed solution, y(tf)
//          y1 holds the next-to-last computed solution, y(tf-h)
//          y2 holds the computed solution y(tf-2*h)
//          y3 holds the computed solution y(tf-3*h)
//
// The return value is a row vector containing all internal
// times at which the solution was computed,
//                  [t0, t1, ..., tN]
std::vector<double> AB3Stepper::Evolve(std::vector<double>& tspan,
                             double h,
                 std::vector<double>& y,
                             std::vector<double>& y1,
                             std::vector<double>& y2,
                             std::vector<double>& y3) {

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

  // check for legal inputs
  if (h <= 0.0) {
    std::cerr << "Evolve: Illegal h\n";
    return times;
  }
  if (tspan[1] <= tspan[0]) {
    std::cerr << "Evolve: Illegal tspan\n";
    return times;
  }

  // figure out how many time steps
  long int N = (tspan[1]-tspan[0])/h;
  if (tspan[1] > tspan[0]+N*h)  N++;

  //evaluate frhs at y1, y2, y3 and store in f, f1, f2
  // store these shifted over so that the update in the
  // iteration is preserved and works for first step and
  // all steps afterwards
  if (frhs->Evaluate(tspan[0]-h, y1, f) != 0) {
    std::cerr << "Evolve: Error in ODE RHS function\n";
    return times;
  }
  if (frhs->Evaluate(tspan[0]-2.0*h, y2, f1) != 0) {
    std::cerr << "Evolve: Error in ODE RHS function\n";
    return times;
  }
```

```
  if (frhs->Evaluate(tspan[0]-3.0*h, y3, f2) != 0) {
    std::cerr << "Evolve: Error in ODE RHS function\n";
    return times;
  }

  // iterate over time steps
  for (int i=0; i<N; i++) {

    // last step only: update h to stop directly at final time
    // NOTE: if  this actually differs from the input h, then
    //       the LMM will reduce to 1st order
    if (i == N-1)
      h = tspan[1]-times[i];

    // update old f's and y's
    y3 = y2;
    y2 = y1;
    y1 = y;
    f3 = f2;
    f2 = f1;
    f1 = f;

    // evaluate f at the current y
    if (frhs->Evaluate(times[i], y, f) != 0) {
      std::cerr << "Evolve: Error in ODE RHS function\n";
      return times;
    }

    // update the current solution using AB3 method from prob 1
    y += (h/24.0)*(55.0*f-59.0*f1+37.0*f2-9.0*f3);

    // update current time, store in output array
    times.push_back(times[i] + h);
  }

  return times;
}
```

```
/* Main routine to test the fourth-order Adams
   Bashforth method on the scalar-valued ODE problem
     y' = -exp(-t)*y, t in [0,5],
     y(0) = 1.

   Based off of Dan Reynolds ERK driver.

   Nicole Deyerl
   Math 6321 @ SMU
   Fall 2016  */

#include <iostream>
#include <vector>
#include "matrix.hpp"
#include "rhs.hpp"
#include "erk4.hpp"
#include "ab3.hpp"

using namespace std;


// ODE RHS function
class MyRHS: public RHSFunction {
public:
  int Evaluate(double t, vector<double>& y, vector<double>& f) {
    f[0] = -exp(-t)*y[0];
    return 0;
  }
};


// Convenience function for analytical solution
vector<double> ytrue(const double t) {
  vector<double> yt = {exp(exp(-t)-1.0)};
  return yt;
};


// main routine
int main() {

  // time steps to try
  vector<double> h = {0.1, 0.05, 0.01, 0.005, 0.001, 0.0005};

  // set problem information
  vector<double> y0 = {1.0};
  double t0 = 0.0;
  double Tf = 5.0;
  double dtout = 1.0;

  // create ODE RHS function objects
  MyRHS rhs;

  // create ERK4 and AB3 stepper objects
  ERK4Stepper ERK4(rhs, y0);
  AB3Stepper AB3(rhs, y0);

  // storage for error results
  vector<double> errs(h.size());

  ///////// Adams Bashforth 3 /////////
  cout << "\nAdams Bashforth 3 Method:\n";

  // loop over time step sizes
```

```
  for (int ih=0; ih<h.size(); ih++) {

    cout << "Running problem 2 with h = " << h[ih] << endl;

    // set the initial conditions (using RK4) and initial time
    vector<double> y(y0);
    double tcur = t0;
    vector<double> tspan = {tcur,tcur+h[ih]};
    vector<double> tvals = ERK4.Evolve(tspan, h[ih], y);
    tcur = tvals.back();
    vector<double> y3 = y;
    tspan = {tcur,tcur+h[ih]};
    tvals = ERK4.Evolve(tspan, h[ih], y);
    tcur = tvals.back();
    vector<double> y2 = y;
    tspan = {tcur,tcur+h[ih]};
    tvals = ERK4.Evolve(tspan, h[ih], y);
    tcur = tvals.back();
    vector<double> y1 = y;
    tspan = {tcur, tcur+h[ih]};
    tvals = ERK4.Evolve(tspan, h[ih], y);
    tcur = tvals.back();

    // reset maxerr
    double maxerr = 0.0;

    // variables to handle the timing being off from RK4 IC's
    double tf;
    int count = 0;
    // loop over output step sizes: call solver and output error
    while (tcur < 0.99999*Tf) {

      // set the time interval for this solve, fix it for the first
      // dtout
      if ( count == 0 ){
          tf = tcur-4.0*h[ih] + dtout;
      } else {
                tf = tcur + dtout;
          }
          count = count + 1;
      vector<double> tspan = {tcur, std::min(tf, Tf)};

      // call the solver, update current time
      vector<double> tvals = AB3.Evolve(tspan, h[ih], y, y1, y2, y3);
      tcur = tvals.back();     // last entry in tvals

      // compute the abs error at tcur, output to screen and accumulate maximum
      vector<double> yerr = y - ytrue(tcur);
      double err = InfNorm(yerr);
      maxerr = std::max(maxerr, err); // holds max abs error
      cout << "  y(" << tcur << ") = " << y[0]
      << "  abserr = " << err << endl;

    }
    cout << "   h = " << h[ih] << "\t  maxerror = " << maxerr;
    errs[ih] = maxerr;

    //convergence print out
    if (ih > 0)
      cout << "\t  conv rate = " << (log(errs[ih])-log(errs[ih-1]))/(log(h[ih])-log(h[i
h-1]));
    cout << endl;
  }
```

```
  return 0;
}
```

```
/* Adaptive forward Euler time stepper class header file.

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

#ifndef ADAPT_EULER_DEFINED__
#define ADAPT_EULER_DEFINED__

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


// Adaptive forward Euler time stepper class
class AdaptEuler {

 private:

  // private reusable local data
  RHSFunction *frhs;          // pointer to ODE RHS function
  std::vector<double> fn;     // local vector storage
  std::vector<double> y1;
  std::vector<double> y2;
  std::vector<double> yerr;

 public:

  double rtol;        // desired relative solution error
  double atol;        // desired absolute solution error
  double grow;        // maximum step size growth factor
  double safe;        // safety factor for step size estimate
  double fail;        // failed step reduction factor
  double ONEMSM;      // safety factors for
  double ONEPSM;      // floating-point comparisons
  double alpha;       // exponent relating step to error
  double error_norm;  // current estimate of the local error ratio
  double h;           // current time step size
  long int fails;     // number of failed steps
  long int steps;     // number of successful steps
  long int maxit;     // maximum number of steps

  // constructor (sets RHS function pointer & solver parameters, copies y for local dat
a)
  AdaptEuler(RHSFunction& frhs_, double rtol_, double atol_, std::vector<double>& y);

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

};

#endif
```

```
/* Adaptive explicit RKF-45 time stepper class header file.
 * Based off of Dan Reynolds' adapt_euler and RK4 files.

   Nicole Deyerl
   Math 6321 @ SMU
   Fall 2016  */

#ifndef ADAPT_RKF_DEFINED__
#define ADAPT_RKF_DEFINED__

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


// Adaptive explicit RKF-45 time stepper class
class AdaptRKF {

 private:

  // private reusable local data
  RHSFunction *frhs;          // pointer to ODE RHS function
  std::vector<double> fn;     // local vector storage
  std::vector<double> y4;
  std::vector<double> y5;
  std::vector<double> yerr;
  std::vector<double> z, f0, f1, f2, f3, f4, f5;    // reused RK vectors
  Matrix A;                                 // Butcher table
  std::vector<double> b5, b4, c; //RK5 and RK4 coeffs

 public:

  double rtol;        // desired relative solution error
  double atol;        // desired absolute solution error
  double grow;        // maximum step size growth factor
  double safe;        // safety factor for step size estimate
  double fail;        // failed step reduction factor
  double ONEMSM;      // safety factors for
  double ONEPSM;      // floating-point comparisons
  double alpha;       // exponent relating step to error
  double error_norm;  // current estimate of the local error ratio
  double h;           // current time step size
  long int fails;     // number of failed steps
  long int steps;     // number of successful steps
  long int maxit;     // maximum number of steps

  // constructor (sets RHS function pointer & solver parameters, copies y for local dat
a)
  AdaptRKF(RHSFunction& frhs_, double rtol_, double atol_, std::vector<double>& y) {
    frhs = &frhs_;                          // store RHSFunction pointer
    z  = y;   // allocate reusable data
    f0 = y;   //   based on size of y
    f1 = y;
    f2 = y;
    f3 = y;
    f4 = y;
    f5 = y;
    A = Matrix(6,6);                        // Butcher table data
    A(1,0) = 0.25;
    A(2,0) = 0.09375; A(2,1) = 0.28125;
    A(3,0) = 1932.0/2197.0;  A(3,1) = -7200.0/2197.0; A(3,2) = 7296.0/2197.0;
    A(4,0) = 439.0/216.0;  A(4,1) = -8.0;  A(4,2) = 3680.0/513.0;  A(4,3) = -845.0/4104
```

```
.0;
    A(5,0) = -8.0/27.0;  A(5,1) = 2.0;  A(5,2) = -3544.0/2565.0;  A(5,3) = 1859.0/4104.
0; A(5,4) = -11.0/40.0;
    b5 = {16.0/135.0, 0.0, 6656.0/12825.0, 28561.0/56430.0, -9.0/50.0, 2.0/55.0};
    b4 = {25.0/216.0, 0.0, 1408.0/2565.0, 2197.0/4104.0, -1.0/5.0, 0.0};
    c = {0.0, 0.25, 0.375, 12.0/13.0, 1.0, 0.5};

    rtol = rtol_;      // set tolerances
    atol = atol_;
    fn = y;            // clone y to create local vectors
    y4 = y;
    y5 = y;
    yerr = y;

    maxit = 1e6;       // set default solver parameters
    grow = 50.0;
    safe = 0.95;
    fail = 0.5;
    ONEMSM = 1.0 - 1.e-8;
    ONEPSM = 1.0 + 1.e-8;
    alpha = -0.5;
    fails = 0;
    steps = 0;
    error_norm = 0.0;
    h = 0.0;
  };

  // Evolve routine (evolves the solution via adaptive RKF45)
  std::vector<double> Evolve(std::vector<double>& tspan, std::vector<double>& y);

  // Single RK4, RK5 step calculation
  int Step(double t, double h, std::vector<double>& y, std::vector<double>& y4,
                             std::vector<double>& y5);

};

#endif
```

```
/* Adaptive forward Euler solver 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 "matrix.hpp"
#include "adapt_euler.hpp"



// Adaptive forward Euler class constructor routine
//
// Inputs:  frhs_ holds the ODE RHSFunction object, f(t,y)
//          rtol holds the desired relative solution accuracy
//          atol holds the desired absolute solution accuracy
//
// Sets default values for adaptivity parameters, all of which may
// be modified by the user after the solver object has been created
AdaptEuler::AdaptEuler(RHSFunction& frhs_, double rtol_,
                       double atol_, std::vector<double>& y) {
  frhs = &frhs_;    // set RHSFunction pointer
  rtol = rtol_;     // set tolerances
  atol = atol_;
  fn = y;           // clone y to create local vectors
  y1 = y;
  y2 = y;
  yerr = y;

  maxit = 1e6;      // set default solver parameters
  grow = 50.0;
  safe = 0.95;
  fail = 0.5;
  ONEMSM = 1.0 - 1.e-8;
  ONEPSM = 1.0 + 1.e-8;
  alpha = -0.5;
  fails = 0;
  steps = 0;
  error_norm = 0.0;
  h = 0.0;
};


// The adaptive forward Euler time step evolution routine
//
// Inputs:  tspan holds the current time interval, [t0, tf]
//          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> tvals = {tspan[0]};

  // check for positive tolerances
  if ((rtol <= 0.0) || (atol <= 0.0)) {
```

```
    std::cerr << "Evolve error: illegal tolerances, atol = "
              << atol << ",  rtol = " << rtol << std::endl;
    return tvals;
  }

  // reset solver statistics
  fails = 0;
  steps = 0;

  // get |y'(t0)|
  if (frhs->Evaluate(tspan[0], y, fn) != 0) {
    std::cerr << "Evolve error in RHS function\n";
    return tvals;
  }

  // estimate initial h value via linearization, safety factor
  error_norm = std::max(Norm(fn) / ( rtol * Norm(y) + atol ), 1.e-8);
  h = safe/error_norm;
  if (tspan[0]+h > tspan[1])   h = tspan[1]-tspan[0];

  // iterate over time steps (all but the last one)
  for (int tstep=1; tstep<=maxit; tstep++) {

    // reset both solution approximations to current solution
    y1 = y;
    y2 = y;

    // get RHS at this time, perform full/half step updates
    if (frhs->Evaluate(tvals[steps], y, fn) != 0) {
      std::cerr << "Evolve error in RHS function\n";
      return tvals;
    }
    y1 += h*fn;
    y2 += (0.5*h)*fn;

    // get RHS at half-step, perform half step update
    if (frhs->Evaluate(tvals[steps]+0.5*h, y2, fn) != 0) {
      std::cerr << "Evolve error in RHS function\n";
      return tvals;
    }
    y2 += (0.5*h)*fn;

    // compute error estimate
    yerr = y2 - y1;

    // compute error estimate success factor
    error_norm = std::max(InfNorm(yerr) / ( rtol * InfNorm(y2) + atol ), 1.e-8);

    // if solution has too much error: reduce step size, increment failure counter, and
 retry
    if (error_norm > ONEPSM) {
      h *= fail;
      fails++;
      continue;
    }

    // successful step
    tvals.push_back(tvals[steps++] + h);  // append updated time, increment step counte
r
    y = 2.0*y2 - y1;

    // exit loop if we've reached the final time
    if ( tvals[steps] >= tspan[1]*ONEMSM )  break;
```

```
    // pick next time step size based on this error estimate
    double eta = safe*std::pow(error_norm, alpha);    // step size estimate
    eta = std::min(eta, grow);                        // maximum growth
    h *= eta;                                         // update h
    h = std::min(h, tspan[1]-tvals[steps]);           // don't pass Tf

  }

  // set output array as the subset of tvals that we actually used
  return tvals;
}
```

```
/* Adaptive Runge Kutta Fehlberg (45) solver class implementation file.
   Based off of Dan Reynolds' adapt_euler and RK45 scripts.

   Class to perform time evolution of the IVP
        y' = f(t,y),  t in [t0, Tf],  y(t0) = y0
   using the RKF-45 time stepping method.

   Nicole Deyerl
   Math 6321 @ SMU
   Fall 2016  */

#include "matrix.hpp"
#include "adapt_rkf.hpp"


// The adaptive RKF45 time step evolution routine
//
// Inputs:  tspan holds the current time interval, [t0, tf]
//          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> AdaptRKF::Evolve(std::vector<double>& tspan,
                                     std::vector<double>& y) {

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

  // check for positive tolerances
  if ((rtol <= 0.0) || (atol <= 0.0)) {
    std::cerr << "Evolve error: illegal tolerances, atol = "
              << atol << ",  rtol = " << rtol << std::endl;
    return tvals;
  }

  // reset solver statistics
  fails = 0;
  steps = 0;

  // get |y'(t0)|
  if (frhs->Evaluate(tspan[0], y, fn) != 0) {
    std::cerr << "Evolve error in RHS function\n";
    return tvals;
  }

  // estimate initial h value via linearization, safety factor
  error_norm = std::max(Norm(fn) / ( rtol * Norm(y) + atol ), 1.e-8);
  h = safe/error_norm;
  if (tspan[0]+h > tspan[1])   h = tspan[1]-tspan[0];

  // iterate over time steps (all but the last one)
  for (int tstep=1; tstep<=maxit; tstep++) {

    // reset both solution approximations to current solution
    y4 = y;
    y5 = y;

    // perform a single step of RK45 to update y
    if (Step(tvals[steps], h, y, y4, y5) != 0) {
      std::cerr << "Evolve: Error in Step() function\n";
      return tvals;
    }
```

```
    // compute error estimate
    yerr = y5 - y4;

    // compute error estimate success factor
    error_norm = std::max(InfNorm(yerr) / ( rtol * InfNorm(y5) + atol ), 1.e-8);

    // if solution has too much error: reduce step size, increment failure counter, and
 retry
    if (error_norm > ONEPSM) {
      h *= fail;
      fails++;
      continue;
    }

    // successful step
    tvals.push_back(tvals[steps++] + h);  // append updated time, increment step counte
r
    y = y4;  //in general we keep the "worse" step, not the better one; the
             // better one is used solely as the error estimator

    // exit loop if we've reached the final time
    if ( tvals[steps] >= tspan[1]*ONEMSM )  break;

    // pick next time step size based on this error estimate
    double eta = safe*std::pow(error_norm, alpha);    // step size estimate
    eta = std::min(eta, grow);                         // maximum growth
    h *= eta;                                          // update h
    h = std::min(h, tspan[1]-tvals[steps]);            // don't pass Tf

  }

  // set output array as the subset of tvals that we actually used
  return tvals;
}

// Single step of explicit 4th and 5th-order Runge-Kutta
//
// Inputs:  t holds the current time
//          h holds the current time step size
//          z, f0-f5 hold temporary vectors needed for the problem
//          y holds the current solution (from RK4)
// Outputs: y4 holds the updated solution, y(t+h) for RK4
//          y5 holds the updated solution, y(t+h) for RK5
//
// The return value is an integer indicating success/failure,
// with 0 indicating success, and nonzero failure.
int AdaptRKF::Step(double t, double h, std::vector<double>& y,
                      std::vector<double>& y4, std::vector<double>& y5) {

  // stage 1: set stage and compute RHS
  z = y;
  if (frhs->Evaluate(t, z, f0) != 0) {
    std::cerr << "Step: Error in ODE RHS function\n";
    return 1;
  }

  // stage 2: set stage and compute RHS
  z = y + (h*A(1,0))*f0;
  if (frhs->Evaluate(t+c[1]*h, z, f1) != 0) {
    std::cerr << "Step: Error in ODE RHS function\n";
    return 1;
  }
```

```
  // stage 3: set stage and compute RHS
  z = y + h*(A(2,0)*f0 + A(2,1)*f1);
  if (frhs->Evaluate(t+c[2]*h, z, f2) != 0) {
    std::cerr << "Step: Error in ODE RHS function\n";
    return 1;
  }

  // stage 4: set stage and compute RHS
  z = y + h*(A(3,0)*f0 + A(3,1)*f1 + A(3,2)*f2);
  if (frhs->Evaluate(t+c[3]*h, z, f3) != 0) {
    std::cerr << "Step: Error in ODE RHS function\n";
    return 1;
  }

  // stage 5: set stage and compute RHS
  z = y + h*(A(4,0)*f0 + A(4,1)*f1 + A(4,2)*f2 + A(4,3)*f3);
  if (frhs->Evaluate(t+c[4]*h, z, f4) != 0) {
    std::cerr << "Step: Error in ODE RHS function\n";
    return 1;
  }

  // stage 6: set stage and compute RHS
  z = y + h*(A(5,0)*f0 + A(5,1)*f1 + A(5,2)*f2 + A(5,3)*f3 + A(5,4)*f4);
  if (frhs->Evaluate(t+c[5]*h, z, f5) != 0) {
    std::cerr << "Step: Error in ODE RHS function\n";
    return 1;
  }

  // compute next step solutions
  y4 += h*(b4[0]*f0 + b4[1]*f1 + b4[2]*f2 + b4[3]*f3 + b4[4]*f4);
  y5 += h*(b5[0]*f0 + b5[1]*f1 + b5[2]*f2 + b5[3]*f3 + b5[4]*f4 + b5[5]*f5);

  // return success
  return 0;
};
```

```
/* Homework 4, problem 4: test adaptive RKF-45 and compare with adaptive
 * Forward Euler for the scalar-valued ODE problem
     y' = -y + 2cos(t), t in [0,10],
     y(0) = 1.
   Based off of Dan Reynolds' driver for hw2.

   Nicole Deyerl
   Math 6321 @ SMU
   Fall 2016  */

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

using namespace std;


// ODE RHS function class -- instantiates a RHSFunction
class MyRHS: public RHSFunction {
public:
  int Evaluate(double t, vector<double>& y, vector<double>& f) {
    f[0] = -y[0] + 2.0*cos(t);
    return 0;
  }
};
//    Convenience function for analytical solution
vector<double> ytrue(const double t) {
  vector<double> yt(1);
  yt[0] = sin(t) + cos(t);
  return yt;
};


// main routine
int main() {

  // tolerances to try
  vector<double> rtols = {1.e-4, 1.e-6, 1.e-8};
  vector<double> atols = {1.e-6, 1.e-8, 1.e-10};

  // initial condition and time span
  vector<double> y0 = {1.0};
  double t0 = 0.0;
  double Tf = 10.0;
  double tcur = t0;
  double dtout = 1.0;

  // create ODE RHS function objects
  MyRHS rhs;

  // loop over relative tolerances
  for (int ir=0; ir<rtols.size(); ir++) {

    // create adaptive RKF-45 stepper object
    AdaptRKF ARKF(rhs, 0.0, atols[ir], y0);
    // set up the problem for this tolerance
    ARKF.rtol = rtols[ir];
    vector<double> y = y0;
    tcur = t0;
    double maxrelerr = 0.0;
```

```
    long int totsteps = 0;
    long int totfails = 0;
    cout << "\nRunning Adaptive RKF45 with rtol = " << ARKF.rtol
         << " and atol = " << atols[ir] << endl;

    //   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 for this time interval
      vector<double> tvals = ARKF.Evolve(tspan, y);
      tcur = tvals.back();  // last entry in tvals
      totsteps += ARKF.steps;
      totfails += ARKF.fails;

      // compute the errors at tcur, output to screen, and accumulate maxima
      vector<double> yerr = y - ytrue(tcur);
      double abserr = InfNorm(yerr);
      double relerr = abserr / InfNorm(ytrue(tcur));
      maxrelerr = std::max(maxrelerr, relerr);
      cout << "  y(" << tcur << ") = " << setprecision(15) << y[0]
           << endl;

    }

    // output final results for this tolerance
    cout << "\nOverall results for rtol = " << ARKF.rtol
     << " abstol = " << atols[ir] << ":\n"
         << "   maxrelerr = " << maxrelerr << endl
         << "   steps = " << totsteps << endl
         << "   fails = " << totfails << endl;

  }



  // loop over relative tolerances
  for (int ir=0; ir<rtols.size(); ir++) {

    // create adaptive forward Euler stepper object
    AdaptEuler AE(rhs, 0.0, atols[ir], y0);

    // set up the problem for this tolerance
    AE.rtol = rtols[ir];
    vector<double> y = y0;
    tcur = t0;
    double maxrelerr = 0.0;
    long int totsteps = 0;
    long int totfails = 0;
    cout << "\nRunning Adaptive Euler with rtol = " << AE.rtol
         << " and atol = " << atols[ir] << endl;

    //   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 for this time interval
      vector<double> tvals = AE.Evolve(tspan, y);
      tcur = tvals.back();  // last entry in tvals
      totsteps += AE.steps;
```

```
      totfails += AE.fails;

      // compute the errors at tcur, output to screen, and accumulate maxima
      vector<double> yerr = y - ytrue(tcur);
      double abserr = InfNorm(yerr);
      double relerr = abserr / InfNorm(ytrue(tcur));
      maxrelerr = std::max(maxrelerr, relerr);
      cout << "  y(" << tcur << ") = " << setprecision(15) << y[0]
           << endl;

    }

    // output final results for this tolerance
    cout << "\nOverall results for rtol = " << AE.rtol
     << " abstol = " << atols[ir] << ":\n"
         << "   maxrelerr = " << maxrelerr << endl
         << "   steps = " << totsteps << endl
         << "   fails = " << totfails << endl;

  }

  return 0;
}
```