```
/* 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 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;
}
```

```
/* Homework 3, problem 2: test adaptive Forward Euler for the
   matrix ODE problem
     y1' = -3*y1 + y2 - exp(-2*t), t in [0,3],
     y2' = y1 - 3y2 + exp(-t)

     matrix form:
     y' = A*y + g(t)  A=[-3,1:1,-3], g(t)=[-exp(-2*t);exp(-t)]

     y1(0) = 2
     y2(0) = 1

   Note: this driver uses Dr Reynolds adapt_euler code.
   Nicole Deyerl
   Math 6321 @ SMU
   Fall 2016  */

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

using namespace std;


// ODE RHS function class -- instantiates a RHSFunction
class MyRHS: public RHSFunction {
    Matrix A;
public:
  // sets up the matrix for this problem
  int Setup() {
    A = Matrix(2,2);
    A(0,0) = -3; A(0,1) = 1;
    A(1,0) = 1;  A(1,1) = -3;
  }
  // sets up the vector g(t) and the rhs f
  int Evaluate(double t, vector<double>& y, vector<double>& f) {
    vector<double> g(2);
    g[0] = -exp(-2.0*t);
    g[1] = exp(-t);
    f = A*y + g;
    return 0;
  }
};
// Analytic solution to the ode system
vector<double> ytrue(double t) {
    vector<double> eD(2);
    eD[0] = (1.0/3.0)*exp(-t) + ((3.0/4.0)-t/2.0)*exp(-2.0*t)
            + (11.0/12.0)*exp(-4.0*t);
    eD[1] =  (2.0/3.0)*exp(-t) + ((5.0/4.0)-t/2.0)*exp(-2.0*t)
            - (11.0/12.0)*exp(-4.0*t);
    return eD;
};


// main routine
int main() {

  // tolerances to try
  vector<double> rtols = {1.e-2, 1.e-4, 1.e-6, 1.e-8};
  double atol = 1.e-12;

  // initial condition and time span given by problem
```

```
  vector<double> y0 = {2.0,1.0}; // vector containing initial condition
  double t0 = 0.0;
  double Tf = 3.0;
  double tcur = t0;
  double dtout = 0.3;

  // create ODE RHS function objects
  MyRHS rhs;
  rhs.Setup();

  // create forward Euler stepper object (will reset rtol before each solve)
  AdaptEuler AE(rhs, 0.0, atol, y0);

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

    // set up the problem for this tolerance
    AE.rtol = rtols[ir];
    vector<double> y = y0;
    tcur = t0;
    double maxabserr = 0.0; //initialize holder for the error between dtouts
    double maxy = 0.0; // initialize holder over all y-values (for rtol*||y||...)
    long int totsteps = 0;
    long int totfails = 0;
    cout << "\nRunning problem 2 with rtol = " << AE.rtol
         << " and atol = " << atol << 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);
      maxy = std::max(maxy, InfNorm(y));
      maxabserr = std::max(maxabserr, abserr);
      cout << "  t = " << tcur << "  y1 = " << setprecision(17) << y[0]
           << "  y2 = " << setprecision(17) << y[1]
           << setprecision(6)
           << "\t abserr = " << abserr
           << endl;

    }

    // output final results for this tolerance
    cout << "\nOverall results for rtol = " << AE.rtol << ":\n"
         << "   max-norm err = " << maxabserr << endl
         << "   rtol*||y||+atol = " << rtols[ir]*maxy + atol << endl
             << "   steps = " << totsteps << endl;

  }

  return 0;
}
```

```
/* Generalized Trapezoidal time stepper class header file.

   Nicole Deyerl
   Math 6321 @ SMU
   Fall 2016  */

#ifndef TRAPEZOIDAL_DEFINED__
#define TRAPEZOIDAL_DEFINED__

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


// General Trapezoid residual function class -- implements a
// general-trapezoid-specific ResidualFunction to be supplied
// to the Newton solver.
class TrapResid: public ResidualFunction {
public:

  // data required to evaluate general trapezoid nonlinear residual
  RHSFunction *frhs;            // pointer to ODE RHS function
  double t;                     // current time
  double h;                     // current step size
  double theta;                 // current theta value
  std::vector<double> *yold;    // pointer to solution at old time step
  std::vector<double>  fold;    // extra vector for residual evaluation

  // constructor (sets RHSFunction and old solution pointers)
  TrapResid(RHSFunction& frhs_, std::vector<double>& yold_) {
    frhs = &frhs_;  yold = &yold_;  fold = yold_;
  };

  // residual evaluation routine
  int Evaluate(std::vector<double>& y, std::vector<double>& resid) {

    // evaluate RHS function at new time (store in resid)
    int ierr = frhs->Evaluate(t+h, y, resid);
    if (ierr != 0) {
      std::cerr << "Error in ODE RHS function = " << ierr << "\n";
      return ierr;
    }

    // evaluate RHS function at old time
    ierr = frhs->Evaluate(t, (*yold), fold);
    if (ierr != 0) {
      std::cerr << "Error in ODE RHS function = " << ierr << "\n";
      return ierr;
    }

    // combine pieces to fill residual, y-yold-h*[theta*f(t+h,y)+(1-theta)*f(t,yold)]
    resid = y - (*yold) - h*(theta*resid+(1.0-theta)*fold);

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

```
// General Trap. residual Jacobian function class -- implements
// a backward-Euler-specific ResidualJacobian to be supplied
// to the Newton solver.
class TrapResidJac: public ResidualJacobian {
public:

  // data required to evaluate General Trap. residual Jacobian
  RHSJacobian *Jrhs;    // ODE RHS Jacobian function pointer
  double t;             // current time
  double h;             // current step size
  double theta;         // current theta value
  // constructor (sets RHS Jacobian function pointer)
  TrapResidJac(RHSJacobian &Jrhs_) { Jrhs = &Jrhs_; };

  // Residual Jacobian evaluation routine
  int Evaluate(std::vector<double>& y, Matrix& J) {

    // evaluate RHS function Jacobian, Jrhs (store in J)
    int ierr = Jrhs->Evaluate(t+h, y, J);
    if (ierr != 0) {
      std::cerr << "Error in ODE RHS Jacobian function = " << ierr << "\n";
      return ierr;
    }
    // combine pieces to fill residual Jacobian,  J = I - h*theta*Jrhs
    J *= (-theta*h);
    for (int i=0; i<J.Rows(); i++)
      J(i,i) += 1.0;

    // return success
    return 0;
  }
};



// Trapezoidal time stepper class
class TrapezoidalStepper {

 private:

  // private reusable local data
  std::vector<double> yold;   // old solution vector
  TrapResid *resid;           // general trap. residual function pointer
  TrapResidJac *residJac;     // general trap. residual Jacobian function pointer

 public:

  NewtonSolver *newt;   // Newton nonlinear solver pointer

  // constructor (constructs residual, Jacobian, theta, and copies y for local data)
  TrapezoidalStepper(RHSFunction& frhs_, RHSJacobian& Jrhs_, double theta,
                     std::vector<double>& y_);

  // destructor (frees pointers to local objects)
  ~TrapezoidalStepper() {
    delete resid;
    delete residJac;
    delete newt;
  };

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

```
};

#endif
```

```
/* Generalized Trapezoidal 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 generalized trapezoidal time stepping method.

   Nicole Deyerl
   Math 6321 @ SMU
   Fall 2016  */

#include "trapezoidal.hpp"


// Trapezoidal stepper construction routine (allocates local data)
//
// Inputs:  frhs_  holds the RHSFunction to use
//          Jrhs_  holds the RHSJacobian to use
//          theta  holds the value for theta for desired trapezoid method
//          y      holds an example solution vector (only used for cloning)
TrapezoidalStepper::TrapezoidalStepper(RHSFunction& frhs_, RHSJacobian& Jrhs_, double t
heta,
                                       std::vector<double>& y) {

  // clone y to create local reusable data
  yold = y;

  // construct objects for nonlinear residual and its Jacobian
  resid = new TrapResid(frhs_, yold);
  residJac = new TrapResidJac(Jrhs_);

  // copy current theta value into resid and resid jac objects
  resid->theta = theta;
  residJac->theta = theta;

  // construct Newton solver object (only copies y)
  // (initialize with default solver parameters; user may override with, e.g.
  //  TrapezoidalStepper.newt->SetMaxit())
  newt = new NewtonSolver(*resid, *residJac, 1.0e-7, 1.0e-11, 100, y, false);
};


// The actual trapezoidal 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> TrapezoidalStepper::Evolve(std::vector<double>& tspan, double h,
                                               std::vector<double>& y) {

  // 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++;

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

    // last step only: update h to stop directly at final time
    if (i == N-1)
      h = tspan[1]-times[i];

    // update resid and residJac objects with information on current state
    resid->t    = times[i];    // copy current time into objects
    residJac->t = times[i];
    resid->h    = h;           // copy current stepsize into objects
    residJac->h = h;
    yold = y;                  // copy y into stored yold object

    // call Newton method to solve for the updated solution
    int ierr = newt->Solve(y);
    if (ierr != 0) {
      std::cerr << "TrapezoidalStepper: Error in Newton solver function = "
                << ierr << "\n";
      return times;
    }

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

  return times;
}
```

```
/* Homework 3, problem 5: test generalized trapezoid for the
   scalar ODE problem
     y' = -lambda*y + 1/(1+t^2) - lambda*atan(t), t in [0,1]
     y(0) = 0
   for various stiffness parameters lambda, step sizes h and
   trapezoid method parameters theta.

   Note: this driver was based off of Dr Reynolds Bwd_euler driver
   and trapezoid cpp and hpp files, and uses Dr Reynolds Newton
   and residual files.
   Nicole Deyerl
   Math 6321 @ SMU
   Fall 2016  */

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

using namespace std;


// Define classes to compute the ODE RHS function and its Jacobian

//    ODE RHS function class -- instantiates a RHSFunction
class MyRHS: public RHSFunction {
public:
  double lambda;                                    // stores some local data
  int Evaluate(double t, vector<double>& y, vector<double>& f) {    // evaluates the RH
S function, f(t,y)
    f[0] = lambda*y[0] + (1/(1+pow(t,2))) - lambda*atan(t); // given by problem
    return 0;
  }
};

//    ODE RHS Jacobian function class -- instantiates a RHSJacobian
class MyJac: public RHSJacobian {
public:
  double lambda;                                            // stores some local data
  int Evaluate(double t, vector<double>& y, Matrix& J) {    // evaluates the RHS Jacobi
an, J(t,y)
    J(0,0) = lambda; // from differentiating f given by problem
    return 0;
  }
};


// Convenience function for analytical solution
vector<double> ytrue(const double t) {
  vector<double> yt = {atan(t)}; // given by problem
  return yt;
};



// main routine
int main() {

  // time steps to try
  vector<double> h = {0.1, 0.01, 0.001, 0.0001};

  // lambda values to try
  vector<double> lambdas = {-200.0, -2000.0, -20000.0};
```

```
// theta values to try
vector<double> thetas = {1.0, 0.55, 0.5, 0.45};

// set problem information
vector<double> y0 = {0.0}; // initial condition
double t0 = 0.0;
double Tf = 1.0;
double dtout = 0.1;

// create ODE RHS and Jacobian objects
MyRHS rhs;
MyJac Jac;


//------ Trapezoidal tests ------

// loop over theta values
for (int it = 0; it<thetas.size(); it++) {

  // create time stepper objects
  // theta is a part of the object, so need to make a new one after each
  //loop over theta
  TrapezoidalStepper Tr(rhs, Jac, thetas[it], y0);

  // update Newton solver parameters
  Tr.newt->SetTolerances(1.e-11, 1.e-13);
  Tr.newt->SetMaxit(50);

  // loop over lambda values
  for (int il=0; il<lambdas.size(); il++) {

    // set current lambda value into rhs and Jac objects
    rhs.lambda = lambdas[il];
    Jac.lambda = lambdas[il];

    //error storage
    vector<double> abserrs(h.size());

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

      // set the initial condition, initial time
      vector<double> y(y0);
      double tcur = t0;

      // reset maxerr
      double maxerr = 0.0;

      cout << "\nRunning trapezoidal with stepsize h = " << h[ih]
      << ", lambda = " << lambdas[il] << ", theta = " << thetas[it] << ":\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 = Tr.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 - ytrue(tcur);
```

```
          double err = InfNorm(yerr);
          maxerr = std::max(maxerr, err);
          cout << "  y(" << tcur << ") = " << y[0]
          << "  \t||error|| = " << err
          << endl;
        }
        abserrs[ih] = maxerr;
        cout << "Max error = " << maxerr << endl;

      }

      // calculate orders of convergence between successive values of h (absolute error
)
      cout << "\nConvergence order estimates:\n";
      for (int ih=0; ih<h.size()-1; ih++) {
        double dlogh = log(h[ih+1]) - log(h[ih]);
        double dloge = log(abserrs[ih+1]) - log(abserrs[ih]);
        cout << "  h = " << h[ih] << "  order = " << dloge/dlogh << endl;
      }
    }
  }
  return 0;
}
```