```
# Makefile for LMM example
#
# Daniel R. Reynolds
# Math 6321 @ SMU
# Fall 2016

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

HEADERS = lmm.hpp matrix.hpp trapezoidal.hpp newton.hpp resid.hpp rhs.hpp

# executable targets
all : prob3.exe

prob3.exe : prob3.o trapezoidal.o lmm.o newton.o matrix.o
	$(CXX) $(CXXFLAGS) -o $@ $^

%.o : %.cpp $(HEADERS)
	$(CXX) -c $(CXXFLAGS) $< -o $@

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

```
/* Linear multistep time stepper class header file. Unaltered from Dan
 * Reynolds' original file.

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

#ifndef LMM_DEFINED__
#define LMM_DEFINED__

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


// Linear multistep residual function class -- implements a
// LMM-specific ResidualFunction to be supplied to the Newton solver.
class LMMResid: public ResidualFunction {
public:

  // data required to evaluate LMM nonlinear residual
  RHSFunction *frhs;       // pointer to ODE RHS function
  double t;                // current time
  double h;                // current step size
  Matrix yold;             // matrix of old solution vectors
  Matrix fold;             // matrix of old right-hand side vectors
  std::vector<double> a;   // vector of LMM "a" coefficients
  std::vector<double> b;   // vector of LMM "b" coefficients

  // constructor (sets RHS function and old solution vector pointers)
  LMMResid(RHSFunction& frhs_, std::vector<double> y,
           std::vector<double> &a_, std::vector<double>& b_) {
    frhs = &frhs_;                        // set RHSFunction pointer
    a = a_;  b = b_;                      // copy LMM coefficients
    yold = Matrix(y.size(), a.size());  // allocate LMM arrays
    fold = Matrix(y.size(), a.size());
  };

  // initializer (fills initial set of 'old' vectors)
  int Initialize(double t, double h, Matrix& y);

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

  // updater (shifts 'old' vectors, adding new one)
  int Update(double t, std::vector<double>& ynew);

};


// Linear multistep residual Jacobian function class -- implements
// a LMM-specific ResidualJacobian to be supplied to the Newton solver.
class LMMResidJac: public ResidualJacobian {
public:

  // data required to evaluate LMM residual Jacobian
  RHSJacobian *Jrhs;   // ODE RHS Jacobian function pointer
  double t;            // current time
  double h;            // current step size
  double beta;         // b_{-1} coefficient
  // constructor (sets RHS Jacobian function pointer)
  LMMResidJac(RHSJacobian& Jrhs_, double beta_) {
    Jrhs = &Jrhs_;   // set RHSJacobian pointer
    beta = beta_;    // copy b_{-1} coefficient
  };

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



// LMM time stepper class
class LMMStepper {

 private:

  // private reusable local data
  RHSFunction *frhs;      // pointer to ODE RHS function
  LMMResid *resid;        // pointer to LMM residual function
  LMMResidJac *residJac;  // pointer to LMM residual Jacobian function
  std::vector<double> a;  // LMM coefficients
  std::vector<double> b;

 public:

  NewtonSolver *newt;     // Newton nonlinear solver pointer

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

  // destructor (frees local data)
  ~LMMStepper() {
    delete newt;
    delete resid;
    delete residJac;
  };

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

};

#endif
```

```
/* Linear multistep 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 a linear multistep time stepping method.  Although this
   class is written to directly support implicit LMM, it will work
   equally well for explicit LMM.  Also, the initial condition fed
   in to Newton solver is found using a step of explicit forward Euler.
   Based off of Dan Reynolds lmm.cpp, which was just slightly altered.

   Nicole Deyerl
   Math 6321 @ SMU
   Fall 2016  */

#include "lmm.hpp"



//////////// LMM Residual ////////////

// residual initialization routine
int LMMResid::Initialize(double t, double h, Matrix &y) {

  // copy y into stored yold object
  int ierr = yold.Copy(y);
  if (ierr != 0) {
    std::cerr << "Error in LMMResid::Initialize Copy call = " << ierr << "\n";
    return ierr;
  }

  // fill initial set of 'old' RHS vectors
  for (int j=0; j<y.Columns(); j++) {
    ierr = frhs->Evaluate(t-j*h, yold[j], fold[j]);
    if (ierr != 0) {
      std::cerr << "Error in LMMResid::Initialzie ODE RHS Evaluate call = " << ierr <<
"\n";
      return ierr;
    }
  }

  // return success
  return 0;
}

// residual evaluation routine
int LMMResid::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;
  }

  // combine pieces to fill residual, y - sum[a_j y_{n-j}] - h*sum[b_j*f_{n-j}]
  resid *= (-h*b[0]);              // resid = -h*b_{-1}*f(t+h,y)
  resid += y;                      // resid = y - h*b_{-1}*f(t+h,y)
  for (int j=0; j<a.size(); j++)
    resid -= ( a[j] * yold[j] + (h*b[j+1]) * fold[j] );

  // return success
  return 0;
}
```

```
// Routine to handle updates of "old" solutions and right-hand sides
//
// Inputs:  tnew  the current time for the new solution
//          ynew  the new solution
// Returns: 0 (success) or 1 (failure)
int LMMResid::Update(double t, std::vector<double>& ynew) {

  // update columns of yold and fold, starting at oldest and moving to newest
  for (int icol=fold.Columns()-1; icol>0; icol--) {
    fold[icol] = fold[icol-1];
    yold[icol] = yold[icol-1];
  }

  // fill first column of yold with ynew
  yold[0] = ynew;

  // evaluate RHS function at new time (store in first column of fold matrix)
  int ierr = frhs->Evaluate(t, ynew, fold[0]);
  if (ierr != 0) {
    std::cerr << "Error in ODE RHS function = " << ierr << "\n";
    return ierr;
  }

  return 0;
};


//////////// LMM Residual Jacobian ////////////

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

  // evaluate RHS function Jacobian (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 *= (-beta*h);                     // J = -beta*h*Jrhs
  for (int i=0; i<J.Rows(); i++)      // J = I - beta*h*Jrhs
    J(i,i) += 1.0;

  // return success
  return 0;
}


//////////// LMM Time Stepper ////////////

// LMM construction routine (allocates local data)
//
// Inputs:  frhs_  holds the RHSFunction to use
//          Jrhs_  holds the RHSJacobian to use
//          y      holds a template solution vector (for cloning)
//          a,b    hold the LMM coefficients
LMMStepper::LMMStepper(RHSFunction& frhs_,
                       RHSJacobian& Jrhs_,
                       std::vector<double>& y,
                       std::vector<double>& a_,
                       std::vector<double>& b_) {

  // check that LMM coefficients are compatible
  if (b_.size() != a_.size()+1) {
    std::cerr << "LMMStepper: Incompatible LMM coefficients; will not function!\n";
    return;
  }

  // point at and/or copy inputs
  frhs = &frhs_;   // set pointer to RHSFunction
  a = a_;          // copy LMM coefficient arrays
  b = b_;

  // construct objects for nonlinear residual and its Jacobian
  resid = new LMMResid(frhs_, y, a, b);
  residJac = new LMMResidJac(Jrhs_, b[0]);

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


// The actual LMM time step evolution routine
//
// Inputs:  tspan holds the current time interval, [t0, tf]
//          h holds the desired time step size
//          y holds the set of initial conditions, [y(0), y(-1), ..., y(-p)]
// Outputs: y holds the computed solution and set of p previous solutions,
//             [y(tf), y(tf-h), ..., y(tf-p*h)]
//
// The return value is a row vector containing all internal
// times at which the solution was computed,
//                 [t0, t1, ..., tN]
std::vector<double> LMMStepper::Evolve(std::vector<double>& tspan,
                                       double h, Matrix& 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;
  }
  if (y.Columns() != a.size()) {
    std::cerr << "LMMStepper: insufficient set of initial conditions, y.Columns() = "
              << y.Columns() << " and a.size() = " << a.size() << "\n";
    return times;
  }

  // initialize residual structures
  int ierr = resid->Initialize(tspan[0], h, y);
  if (ierr != 0) {
    std::cerr << "LMMStepper::Evolve error in residual Initialize call = " << ierr << "
\n";
    return times;
  }

  std::vector<double> f = y[0]; //create a vector to hold the feval

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

  // set vector ycur to contain one step of explicit euler
  std::vector<double> ycur = y[0];

  // 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;

    ycur += h*f; // one step of forward euler per time step -> one IC per time step

    // call Newton method to solve for the updated solution
    // (initial guess is one step of forward euler)
    int nerr = newt->Solve(ycur);
    if (nerr != 0) {
      std::cerr << "LMMStepper: Error in Newton solver function = " << nerr << "\n";
      return times;
    }

    // update current time (store in output array), and "old" data
    times.push_back(times[i] + h);
    resid->Update(times[i+1], ycur);
  }

  // copy yold data back into y Matrix
  y.Copy(resid->yold);

  return times;
}
```

```
/* Main routine to test a set of LMM on the scalar-valued ODE problem
     y’ = lambda*y + 1/(1+t^2) - lambda*arctan(t), t in [0,3],
     y(0) = 0,
   using the O(h^3) AM and BDF time steppers.
   Based off of Dan Reynolds’ LMM driver.

   Nicole Deyerl
   Math 6321 @ SMU
   Fall 2016  */

#include <iostream>
#include <vector>
#include "matrix.hpp"
#include "rhs.hpp"
#include "trapezoidal.hpp"
#include "lmm.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 loca
l 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+t*t)) - lambda*atan(t);
    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;
    J(0,0) = lambda;
    return 0;
  }
};


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



// main routine
int main() {

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

  // lambda value
  vector<double> lambda = {-10.0, -100.0, -1000.0, -10000.0};
  // set problem information
  vector<double> y0 = {0.0};
  long int M=1;
  double t0 = 0.0;
  double Tf = 3.0;
  double dtout = 0.3;

  vector<double> errs(h.size());

  for (int il = 0; il<lambda.size(); il++){

    // create ODE RHS and Jacobian objects, store lambda value for this test
    MyRHS rhs;
    MyJac Jac;
    rhs.lambda = lambda[il];
    Jac.lambda = lambda[il];

    cout << "\nlambda = " << lambda[il] << ":" << endl;

    ////////// AM-2 //////////
    // Third order Adams Moulton Method
    cout << "\n AM-2:\n";

    // create Trapezoid and LMM methods, set solver parameters
    TrapezoidalStepper TrapAM2(rhs, Jac, y0);
    vector<double> AM2_a = {1.0, 0.0};
    vector<double> AM2_b = {5.0/12.0, 8.0/12.0, -1.0/12.0};
    LMMStepper AM2(rhs, Jac, y0, AM2_a, AM2_b);
    TrapAM2.newt->SetTolerances(1.e-9, 1.e-11);
    TrapAM2.newt->SetMaxit(20);
    AM2.newt->SetTolerances(1.e-9, 1.e-11);
    AM2.newt->SetMaxit(20);

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

      // set the initial time, first output time
      double tcur = t0;
      double tout = t0 + dtout;

      // reset maxerr
      double maxerr = 0.0;

      // AM-2 requires two initial conditions
      Matrix y_AM2(M,2);
      //   first is just y0 (insert into 2nd column of y_AM2)
      y_AM2[1] = y0;
      //   second comes from trapezoid step step (insert into 1st column of y_AM2)
      vector<double> tspan = {tcur, tcur+h[ih]};
      y_AM2[0] = y0;
      vector<double> tvals = TrapAM2.Evolve(tspan, h[ih], y_AM2[0]);
      //   update tcur to end of initial conditions
      tcur += h[ih];

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

        // set the time interval for this solve
        tspan = {tcur, tout};

        // call the solver, update current time
        tvals = AM2.Evolve(tspan, h[ih], y_AM2);
        tcur = tvals.back();   // last entry in tvals
        // compute the error at tcur, output to screen and accumulate maximum
        vector<double> yerr = y_AM2[0] - ytrue(tcur);   // computed solution is in 1st
column of y_AM2
        double err = InfNorm(yerr);
        maxerr = std::max(maxerr, err);

        // update output time for next solve
        tout = std::min(tcur + dtout, Tf);

      }

      cout << "\t Max error = " << 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[
ih-1]));
          cout << endl;

    }

    ////////// BDF-3 //////////
    // Third order BDF method
    cout << "\n BDF-3:\n";

    // create Trapezoid and LMM methods, set solver parameters
    TrapezoidalStepper TrapBDF(rhs, Jac, y0);
    vector<double> BDF3_a = {18.0/11.0, -9.0/11.0, 2.0/11.0};
    vector<double> BDF3_b = {6.0/11.0, 0.0, 0.0, 0.0};
    LMMStepper BDF3(rhs, Jac, y0, BDF3_a, BDF3_b);
    TrapBDF.newt->SetTolerances(1.e-9, 1.e-11);
    TrapBDF.newt->SetMaxit(20);
    BDF3.newt->SetTolerances(1.e-9, 1.e-11);
    BDF3.newt->SetMaxit(20);

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

      // set the initial time, first output time
      double tcur = t0;
      double tout = t0 + dtout;

      // reset maxerr
      double maxerr = 0.0;

      // BDF-3 requires three initial conditions
      Matrix y_BDF3(M,3);
      //   first is just y0 (insert into 3rd column of y_BDF3)
      y_BDF3[2] = y0;
      //   second comes from trapezoid step step (insert into 2nd column of y_BDF3)
      vector<double> tspan = {tcur, tcur+h[ih]};
      y_BDF3[1] = y0;
      vector<double> tvals = TrapBDF.Evolve(tspan, h[ih], y_BDF3[1]);
      //   update tcur to end of initial conditions
      tcur += h[ih];
      //   third comes from trapezoid step step (insert into 1st column of y_BDF3)
      tspan = {tcur, tcur+h[ih]};
      y_BDF3[0] = y_BDF3[1];
      tvals = TrapBDF.Evolve(tspan, h[ih], y_BDF3[0]);
      //   update tcur to end of initial conditions
      tcur += h[ih];
      // loop over output step sizes: call solver and output error
      while (tcur < 0.99999*Tf) {

        // set the time interval for this solve
        tspan = {tcur, tout};

        // call the solver, update current time
        tvals = BDF3.Evolve(tspan, h[ih], y_BDF3);
        tcur = tvals.back();   // last entry in tvals

        // compute the error at tcur, output to screen and accumulate maximum
        vector<double> yerr = y_BDF3[0] - ytrue(tcur);   // computed solution is in 1st
 column of y_BDF3
        double err = InfNorm(yerr);
        maxerr = std::max(maxerr, err);

        // update output time for next solve
        tout = std::min(tcur + dtout, Tf);

      }

      cout << "\t Max error = " << 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
[ih-1]));
          cout << endl;

    }
  }
  return 0;
}
```