Федеральное государственное автономное образовательное учреждение
высшего образования «Санкт-Петербургский политехнический университет
Петра Великого»
Институт компьютерных наук и технологий
Программная инженерия

ПОЛИТЕХ
Санкт-Петербургский
политехнический университет
Петра Великого

Отчёт по лабораторным работам

по дисциплине «Вычислительная математика»

Студент гр. 5130904/30008 Ребдев П.А
Руководитель: Скуднева Е.В

Санкт-Петербург
2025

Оглавление

Лабораторная работа №3

1. Задание

Вариант №16

Решить систему дифференциальных уравнений:

$$\frac{dx_1}{dt} = -14x_1 + 13x_2 + \cos(1+t); \frac{dx_2}{dt} = 20x_1 - 30x_2 + arctg(1+t^2);$$

$x_1(0) = 2$, $x_2(0) = 0.5$, $t \in [0, 1.5]$

следующими способами с одним и тем же шагом печати $h_{print} = 0.075$:

I) по программе RKF45 с EPS = 0.0001;

II) методом Рунге-Кутты 2-й степени точности

$$z_{n+\frac{2}{3}} = z_n + \frac{2h}{3} f(t_n, z_n)$$

$$z_{n+1} = z_n + \frac{h}{4}(f(t_n, z_n) + 3f(t_{n+\frac{2}{3}}, z_{n+\frac{2}{3}}))$$

с двумя постоянными шагами интегрирования:

а) $h_{int} = 0.075$

б) любой другой, позволяющий получить качественно верное решение.

Сравните результаты.

2. Решение

2.1 Ход решения

1. Вычисления значений функций в точках с помощью RKF45, с погрешность 0.0001 и шагом 0.075

2. Вычисления значений функций в точках с помощью метода Рунге-Кутты второй степени с шагом 0.075

3. Нахождение оптимального шага для метода Рунге-Кутты второй степени

4. Вычисления значений функций в точках с помощью метода Рунге-Кутты второй степени с оптимальным шагом

2.2 Основная программа

main.cpp

```
#include <iostream>
#include <cmath>
#include <iomanip>

#include "rkf45.hpp"
#include "myFunc.hpp"

void f (double t, double x[], double xp[])
{
  xp[0] = -14.0 * x[0] + 13.0 * x[1] + std::cos(1.0 + t);
  xp[1] = 20.0 * x[0] - 30.0 * x[1] + std::atan(1.0 + t * t);
}

int main()
{
  std::cout <<  std::fixed << std::setprecision(6);

  double eps = 0.0001;
  double xStart[2] = {2, 0.5};
  double xpStart[2] = {0, 0};
  double tStart = 0;
  double standartStep = 0.075;
  double t = 0;

  f(0, xStart, xpStart);
  std::cout << "\033[1;32mStarting values:\033[1;0m\nx1(0) = " << xStart[0] << "; x2(0) = " << xStart[1] <<
"\ndx1/xdt(0) = " << xpStart[0] << "; dx2/xdt(0) = " << xpStart[1] << "\n\n\033[1;32mrkf45:\033[1;0m\n";
  for (t = 0; t < 1.5; t += standartStep)
  {
    r8_rkf45(f, 2, xStart, xpStart, &tStart, t, &eps, eps, 1);
    std::cout << "t = " << t << ";\tx1 = " << xStart[0] << ";\tx2 = " << xStart[1] << '\n';
  }
```

```
 std::cout << "\n\033[1;32mrk2 with 0.075 step:\033[1;0m\n";
 xStart[0] = 2;
 xStart[1] = 0.5;
 for (t = standartStep; t < 1.5; t += standartStep)
 {
  rk2(f, xStart, t, standartStep);
  std::cout << "t = " << t << ";\tx1 = " << xStart[0] << ";\tx2 = " << xStart[1] << '\n';
 }

 double bestStep = 0.05;
 std::cout << "\n\033[1;32mrk2 with optimal step (0.05):\033[1;0m\n";
 xStart[0] = 2;
 xStart[1] = 0.5;
 for (t = bestStep; t < 1.5; t += bestStep)
 {
  rk2(f, xStart, t, bestStep);
  std::cout << "t = " << t << ";\tx1 = " << xStart[0] << ";\tx2 = " << xStart[1] << '\n';
 }

 std::cout << "\n\033[1;32mrkf45 with 0.05 step:\033[1;0m\n";
 xStart[0] = 2;
 xStart[1] = 0.5;
 tStart = 0;
 f(0, xStart, xpStart);
 for (t = 0; t < 1.5; t += bestStep)
 {
  r8_rkf45(f, 2, xStart, xpStart, &tStart, t, &eps, eps, 1);
  std::cout << "t = " << t << ";\tx1 = " << xStart[0] << ";\tx2 = " << xStart[1] << '\n';
 }

 return 0;
}
```

2.3 Программа RKF45

rkf45.cpp

```
//****************************************************************************80

int r8_rkf45 ( void f ( double t, double y[], double yp[] ), int neqn,
  double y[], double yp[], double *t, double tout, double *relerr,
  double abserr, int flag )

//****************************************************************************80
//
//  Purpose:
//
//    R8_RKF45 carries out the Runge-Kutta-Fehlberg method.
//
//  Discussion:
//
//    This version of the routine uses DOUBLE real arithmetic.
//
//    This routine is primarily designed to solve non-stiff and mildly stiff
//    differential equations when derivative evaluations are inexpensive.
//    It should generally not be used when the user is demanding
//    high accuracy.
```

```
//
//   This routine integrates a system of NEQN first-order ordinary differential
//   equations of the form:
//
//     dY(i)/dT = F(T,Y(1),Y(2),...,Y(NEQN))
//
//   where the Y(1:NEQN) are given at T.
//
//   Typically the subroutine is used to integrate from T to TOUT but it
//   can be used as a one-step integrator to advance the solution a
//   single step in the direction of TOUT.  On return, the parameters in
//   the call list are set for continuing the integration.  The user has
//   only to call again (and perhaps define a new value for TOUT).
//
//   Before the first call, the user must
//
//   * supply the subroutine F(T,Y,YP) to evaluate the right hand side;
//     and declare F in an EXTERNAL statement;
//
//   * initialize the parameters:
//     NEQN, Y(1:NEQN), T, TOUT, RELERR, ABSERR, FLAG.
//     In particular, T should initially be the starting point for integration,
//     Y should be the value of the initial conditions, and FLAG should
//     normally be +1.
//
//   Normally, the user only sets the value of FLAG before the first call, and
//   thereafter, the program manages the value.  On the first call, FLAG should
//   normally be +1 (or -1 for single step mode.)  On normal return, FLAG will
//   have been reset by the program to the value of 2 (or -2 in single
//   step mode), and the user can continue to call the routine with that
//   value of FLAG.
//
//   (When the input magnitude of FLAG is 1, this indicates to the program
//   that it is necessary to do some initialization work.  An input magnitude
//   of 2 lets the program know that that initialization can be skipped,
//   and that useful information was computed earlier.)
//
//   The routine returns with all the information needed to continue
//   the integration.  If the integration reached TOUT, the user need only
//   define a new TOUT and call again.  In the one-step integrator
//   mode, returning with FLAG = -2, the user must keep in mind that
//   each step taken is in the direction of the current TOUT.  Upon
//   reaching TOUT, indicated by the output value of FLAG switching to 2,
//   the user must define a new TOUT and reset FLAG to -2 to continue
//   in the one-step integrator mode.
//
//   In some cases, an error or difficulty occurs during a call.  In that case,
//   the output value of FLAG is used to indicate that there is a problem
//   that the user must address.  These values include:
//
//   * 3, integration was not completed because the input value of RELERR, the
//     relative error tolerance, was too small.  RELERR has been increased
//     appropriately for continuing.  If the user accepts the output value of
//     RELERR, then simply reset FLAG to 2 and continue.
//
//   * 4, integration was not completed because more than MAXNFE derivative
```

```
//     evaluations were needed.  This is approximately (MAXNFE/6) steps.
//     The user may continue by simply calling again.  The function counter
//     will be reset to 0, and another MAXNFE function evaluations are allowed.
//
//   * 5, integration was not completed because the solution vanished,
//     making a pure relative error test impossible.  The user must use
//     a non-zero ABSERR to continue.  Using the one-step integration mode
//     for one step is a good way to proceed.
//
//   * 6, integration was not completed because the requested accuracy
//     could not be achieved, even using the smallest allowable stepsize.
//     The user must increase the error tolerances ABSERR or RELERR before
//     continuing.  It is also necessary to reset FLAG to 2 (or -2 when
//     the one-step integration mode is being used).  The occurrence of
//     FLAG = 6 indicates a trouble spot.  The solution is changing
//     rapidly, or a singularity may be present.  It often is inadvisable
//     to continue.
//
//   * 7, it is likely that this routine is inefficient for solving
//     this problem.  Too much output is restricting the natural stepsize
//     choice.  The user should use the one-step integration mode with
//     the stepsize determined by the code.  If the user insists upon
//     continuing the integration, reset FLAG to 2 before calling
//     again.  Otherwise, execution will be terminated.
//
//   * 8, invalid input parameters, indicates one of the following:
//     NEQN <= 0;
//     T = TOUT and |FLAG| /= 1;
//     RELERR < 0 or ABSERR < 0;
//     FLAG == 0  or FLAG < -2 or 8 < FLAG.
//
//  Licensing:
//
//    This code is distributed under the MIT license.
//
//  Modified:
//
//    13 October 2012
//
//  Author:
//
//    Original FORTRAN77 version by Herman Watts, Lawrence Shampine.
//    C++ version by John Burkardt.
//
//  Reference:
//
//    Erwin Fehlberg,
//    Low-order Classical Runge-Kutta Formulas with Stepsize Control,
//    NASA Technical Report R-315, 1969.
//
//    Lawrence Shampine, Herman Watts, S Davenport,
//    Solving Non-stiff Ordinary Differential Equations - The State of the Art,
//    SIAM Review,
//    Volume 18, pages 376-411, 1976.
//
//  Parameters:
```

```
//
//    Input, external F, a user-supplied subroutine to evaluate the
//    derivatives Y'(T), of the form:
//
//      void f ( double t, double y[], double yp[] )
//
//    Input, int NEQN, the number of equations to be integrated.
//
//    Input/output, double Y[NEQN], the current solution vector at T.
//
//    Input/output, double YP[NEQN], the derivative of the current solution
//    vector at T.  The user should not set or alter this information!
//
//    Input/output, double *T, the current value of the independent variable.
//
//    Input, double TOUT, the output point at which solution is desired.
//    TOUT = T is allowed on the first call only, in which case the routine
//    returns with FLAG = 2 if continuation is possible.
//
//    Input, double *RELERR, ABSERR, the relative and absolute error tolerances
//    for the local error test.  At each step the code requires:
//      abs ( local error ) <= RELERR * abs ( Y ) + ABSERR
//    for each component of the local error and the solution vector Y.
//    RELERR cannot be "too small".  If the routine believes RELERR has been
//    set too small, it will reset RELERR to an acceptable value and return
//    immediately for user action.
//
//    Input, int FLAG, indicator for status of integration. On the first call,
//    set FLAG to +1 for normal use, or to -1 for single step mode.  On
//    subsequent continuation steps, FLAG should be +2, or -2 for single
//    step mode.
//
//    Output, int RKF45_D, indicator for status of integration.  A value of 2
//    or -2 indicates normal progress, while any other value indicates a
//    problem that should be addressed.
//
{
# define MAXNFE 3000

  static double abserr_save = -1.0;
  double ae;
  double dt;
  double ee;
  double eeoet;
  double eps;
  double esttol;
  double et;
  double *f1;
  double *f2;
  double *f3;
  double *f4;
  double *f5;
  int flag_return;
  static int flag_save = -1000;
  static double h = -1.0;
  bool hfaild;
```

```
  double hmin;
  int i;
  static int init = -1000;
  int k;
  static int kflag = -1000;
  static int kop = -1;
  int mflag;
  static int nfe = -1;
  bool output;
  double relerr_min;
  static double relerr_save = -1.0;
  static double remin = 1.0E-12;
  double s;
  double scale;
  double tol;
  double toln;
  double ypk;

  flag_return = flag;
//
//  Check the input parameters.
//
  eps = DBL_EPSILON;

  if ( neqn < 1 )
  {
    flag_return = 8;
    cerr << "\n";
    cerr << "R8_RKF45 - Fatal error!\n";
    cerr << "  Invalid input value of NEQN.\n";
    return flag_return;
  }

  if ( (*relerr) < 0.0 )
  {
    flag_return = 8;
    cerr << "\n";
    cerr << "R8_RKF45 - Fatal error!\n";
    cerr << "  Invalid input value of RELERR.\n";
    return flag_return;
  }

  if ( abserr < 0.0 )
  {
    flag_return = 8;
    cerr << "\n";
    cerr << "R8_RKF45 - Fatal error!\n";
    cerr << "  Invalid input value of ABSERR.\n";
    return flag_return;
  }

  if ( flag_return == 0 || 8 < flag_return  || flag_return < -2 )
  {
    flag_return = 8;
    cerr << "\n";
    cerr << "R8_RKF45 - Fatal error!\n";
```

```
    cerr << "  Invalid input.\n";
    return flag_return;
  }

  mflag = abs ( flag_return );
//
//  Is this a continuation call?
//
  if ( mflag != 1 )
  {
    if ( *t == tout && kflag != 3 )
    {
      flag_return = 8;
      return flag_return;
    }
//
//  FLAG = -2 or +2:
//
    if ( mflag == 2 )
    {
      if ( kflag == 3 )
      {
        flag_return = flag_save;
        mflag = abs ( flag_return );
      }
      else if ( init == 0 )
      {
        flag_return = flag_save;
      }
      else if ( kflag == 4 )
      {
        nfe = 0;
      }
      else if ( kflag == 5 && abserr == 0.0 )
      {
        cerr << "\n";
        cerr << "R8_RKF45 - Fatal error!\n";
        cerr << "  KFLAG = 5 and ABSERR = 0.0\n";
        exit ( 1 );
      }
      else if ( kflag == 6 && (*relerr) <= relerr_save && abserr <= abserr_save )
      {
        cerr << "\n";
        cerr << "R8_RKF45 - Fatal error!\n";
        cerr << "  KFLAG = 6 and\n";
        cerr << "  RELERR <= RELERR_SAVE and\n";
        cerr << "  ABSERR <= ABSERR_SAVE\n";
        exit ( 1 );
      }
    }
//
//  FLAG = 3, 4, 5, 6, 7 or 8.
//
    else
    {
      if ( flag_return == 3 )
```

```
    {
      flag_return = flag_save;
      if ( kflag == 3 )
      {
        mflag = abs ( flag_return );
      }
    }
    else if ( flag_return == 4 )
    {
      nfe = 0;
      flag_return = flag_save;
      if ( kflag == 3 )
      {
        mflag = abs ( flag_return );
      }
    }
    else if ( flag_return == 5 && 0.0 < abserr )
    {
      flag_return = flag_save;
      if ( kflag == 3 )
      {
        mflag = abs ( flag_return );
      }
    }
//
//  Integration cannot be continued because the user did not respond to
//  the instructions pertaining to FLAG = 5, 6, 7 or 8.
//
    else
    {
      cerr << "\n";
      cerr << "R8_RKF45 - Fatal error!\n";
      cerr << "  Integration cannot be continued.\n";
      cerr << "  The user did not respond to the output\n";
      cerr << "  value FLAG = 5, 6, 7, or 8.\n";
      exit ( 1 );
    }
  }
 }
//
//  Save the input value of FLAG.
//  Set the continuation flag KFLAG for subsequent input checking.
//
 flag_save = flag_return;
 kflag = 0;
//
//  Save RELERR and ABSERR for checking input on subsequent calls.
//
 relerr_save = (*relerr);
 abserr_save = abserr;
//
//  Restrict the relative error tolerance to be at least
//
//    2*EPS+REMIN
//
//  to avoid limiting precision difficulties arising from impossible
```

```
//  accuracy requests.
//
  relerr_min = 2.0 * DBL_EPSILON + remin;
//
//  Is the relative error tolerance too small?
//
  if ( (*relerr) < relerr_min )
  {
    (*relerr) = relerr_min;
    kflag = 3;
    flag_return = 3;
    return flag_return;
  }

  dt = tout - *t;
//
//  Initialization:
//
//  Set the initialization completion indicator, INIT;
//  set the indicator for too many output points, KOP;
//  evaluate the initial derivatives
//  set the counter for function evaluations, NFE;
//  estimate the starting stepsize.
//
  f1 = new double[neqn];
  f2 = new double[neqn];
  f3 = new double[neqn];
  f4 = new double[neqn];
  f5 = new double[neqn];

  if ( mflag == 1 )
  {
    init = 0;
    kop = 0;
    f ( *t, y, yp );
    nfe = 1;

    if ( *t == tout )
    {
      flag_return = 2;
      return flag_return;
    }

  }

  if ( init == 0 )
  {
    init = 1;
    h = fabs ( dt );
    toln = 0.0;

    for ( k = 0; k < neqn; k++ )
    {
      tol = (*relerr) * fabs ( y[k] ) + abserr;
      if ( 0.0 < tol )
      {
```

```
      toln = tol;
      ypk = fabs ( yp[k] );
      if ( tol < ypk * pow ( h, 5 ) )
      {
        h = pow ( ( tol / ypk ), 0.2 );
      }
    }
  }

  if ( toln <= 0.0 )
  {
    h = 0.0;
  }

  h = fmax ( h, 26.0 * eps * fmax ( fabs ( *t ), fabs ( dt ) ) );

  if ( flag_return < 0 )
  {
    flag_save = -2;
  }
  else
  {
    flag_save = 2;
  }
 }
//
//  Set stepsize for integration in the direction from T to TOUT.
//
 h = r8_sign ( dt ) * fabs ( h );
//
//  Test to see if too may output points are being requested.
//
 if ( 2.0 * fabs ( dt ) <= fabs ( h ) )
 {
  kop = kop + 1;
 }
//
//  Unnecessary frequency of output.
//
 if ( kop == 100 )
 {
  kop = 0;
  delete [] f1;
  delete [] f2;
  delete [] f3;
  delete [] f4;
  delete [] f5;
  flag_return = 7;
  return flag_return;
 }
//
//  If we are too close to the output point, then simply extrapolate and return.
//
 if ( fabs ( dt ) <= 26.0 * eps * fabs ( *t ) )
 {
  *t = tout;
```

```
    for ( i = 0; i < neqn; i++ )
    {
      y[i] = y[i] + dt * yp[i];
    }
    f ( *t, y, yp );
    nfe = nfe + 1;

    delete [] f1;
    delete [] f2;
    delete [] f3;
    delete [] f4;
    delete [] f5;
    flag_return = 2;
    return flag_return;
  }
//
//  Initialize the output point indicator.
//
  output = false;
//
//  To avoid premature underflow in the error tolerance function,
//  scale the error tolerances.
//
  scale = 2.0 / (*relerr);
  ae = scale * abserr;
//
//  Step by step integration.
//
  for ( ; ; )
  {
    hfaild = false;
//
//  Set the smallest allowable stepsize.
//
    hmin = 26.0 * eps * fabs ( *t );
//
//  Adjust the stepsize if necessary to hit the output point.
//
//  Look ahead two steps to avoid drastic changes in the stepsize and
//  thus lessen the impact of output points on the code.
//
    dt = tout - *t;

    if ( 2.0 * fabs ( h ) <= fabs ( dt ) )
    {
    }
    else
//
//  Will the next successful step complete the integration to the output point?
//
    {
      if ( fabs ( dt ) <= fabs ( h ) )
      {
        output = true;
        h = dt;
      }
```

```
    else
    {
      h = 0.5 * dt;
    }

  }
//
//  Here begins the core integrator for taking a single step.
//
//  The tolerances have been scaled to avoid premature underflow in
//  computing the error tolerance function ET.
//  To avoid problems with zero crossings, relative error is measured
//  using the average of the magnitudes of the solution at the
//  beginning and end of a step.
//  The error estimate formula has been grouped to control loss of
//  significance.
//
//  To distinguish the various arguments, H is not permitted
//  to become smaller than 26 units of roundoff in T.
//  Practical limits on the change in the stepsize are enforced to
//  smooth the stepsize selection process and to avoid excessive
//  chattering on problems having discontinuities.
//  To prevent unnecessary failures, the code uses 9/10 the stepsize
//  it estimates will succeed.
//
//  After a step failure, the stepsize is not allowed to increase for
//  the next attempted step.  This makes the code more efficient on
//  problems having discontinuities and more effective in general
//  since local extrapolation is being used and extra caution seems
//  warranted.
//
//  Test the number of derivative function evaluations.
//  If okay, try to advance the integration from T to T+H.
//
  for ( ; ; )
  {
//
//  Have we done too much work?
//
    if ( MAXNFE < nfe )
    {
      kflag = 4;
      delete [] f1;
      delete [] f2;
      delete [] f3;
      delete [] f4;
      delete [] f5;
      flag_return = 4;
      return flag_return;
    }
//
//  Advance an approximate solution over one step of length H.
//
    r8_fehl ( f, neqn, y, *t, h, yp, f1, f2, f3, f4, f5, f1 );
    nfe = nfe + 5;
//
```

```
//  Compute and test allowable tolerances versus local error estimates
//  and remove scaling of tolerances.  The relative error is
//  measured with respect to the average of the magnitudes of the
//  solution at the beginning and end of the step.
//
    eeoet = 0.0;

    for ( k = 0; k < neqn; k++ )
    {
      et = fabs ( y[k] ) + fabs ( f1[k] ) + ae;

      if ( et <= 0.0 )
      {
        delete [] f1;
        delete [] f2;
        delete [] f3;
        delete [] f4;
        delete [] f5;
        flag_return = 5;
        return flag_return;
      }

      ee = fabs
      ( ( -2090.0 * yp[k]
        + ( 21970.0 * f3[k] - 15048.0 * f4[k] )
        )
      + ( 22528.0 * f2[k] - 27360.0 * f5[k] )
      );

      eeoet = fmax ( eeoet, ee / et );

    }

    esttol = fabs ( h ) * eeoet * scale / 752400.0;

    if ( esttol <= 1.0 )
    {
      break;
    }
//
//  Unsuccessful step.  Reduce the stepsize, try again.
//  The decrease is limited to a factor of 1/10.
//
    hfaild = true;
    output = false;

    if ( esttol < 59049.0 )
    {
      s = 0.9 / pow ( esttol, 0.2 );
    }
    else
    {
      s = 0.1;
    }

    h = s * h;
```

```
    if ( fabs ( h ) < hmin )
    {
      kflag = 6;
      delete [] f1;
      delete [] f2;
      delete [] f3;
      delete [] f4;
      delete [] f5;
      flag_return = 6;
      return flag_return;
    }

  }
//
//  We exited the loop because we took a successful step.
//  Store the solution for T+H, and evaluate the derivative there.
//
  *t = *t + h;
  for ( i = 0; i < neqn; i++ )
  {
    y[i] = f1[i];
  }
  f ( *t, y, yp );
  nfe = nfe + 1;
//
//  Choose the next stepsize.  The increase is limited to a factor of 5.
//  If the step failed, the next stepsize is not allowed to increase.
//
  if ( 0.0001889568 < esttol )
  {
    s = 0.9 / pow ( esttol, 0.2 );
  }
  else
  {
    s = 5.0;
  }

  if ( hfaild )
  {
    s = fmin ( s, 1.0 );
  }

  h = r8_sign ( h ) * fmax ( s * fabs ( h ), hmin );
//
//  End of core integrator
//
//  Should we take another step?
//
  if ( output )
  {
    *t = tout;
    delete [] f1;
    delete [] f2;
    delete [] f3;
    delete [] f4;
```

```
    delete [] f5;
    flag_return = 2;
    return flag_return;
   }

   if ( flag_return <= 0 )
   {
    delete [] f1;
    delete [] f2;
    delete [] f3;
    delete [] f4;
    delete [] f5;
    flag_return = -2;
    return flag_return;
   }

  }
# undef MAXNFE
}
```

2.4 Метод Рунге-Кунты второй степени

myFunc.cpp

```
#include "myFunc.hpp"

void rk2(void f (double t, double y[], double yp[]), double y[], double t, double h)
{
  double k1[2];
  f(t, y, k1);
  double z23[2];
  for (int i = 0; i < 2; ++i)
  {
    z23[i] = y[i] + ((2.0 / 3.0) * h) * k1[i];
  }

  double k2[2];
  f(t + (2.0 / 3.0) * h, z23, k2);
  for (int i = 0; i < 2; ++i)
  {
    y[i] = y[i] + (h / 4.0) * (k1[i] + 3.0 * k2[i]);
  }
}
```

3. Результаты

3.1 Нахождение значения функций в точках с помощью RKF45

t	x1	x2
0.000000	2.000000	0.500000
0.075000	1.261111	0.928071
0.150000	0.958931	0.746607
0.225000	0.744288	0.585584
0.300000	0.582751	0.463244
0.375000	0.460015	0.370715
0.450000	0.365983	0.300437
0.525000	0.293265	0.246630
0.600000	0.236366	0.205060
0.675000	0.191228	0.172572
0.750000	0.154846	0.146830
0.825000	0.125002	0.126104
0.900000	0.100058	0.109113
0.975000	0.078813	0.094914
1.050000	0.060387	0.082814
1.125000	0.044141	0.072308
1.200000	0.029617	0.063033
1.275000	0.016488	0.054729
1.350000	0.004530	0.047217
1.425000	-0.006408	0.040373
1.500000	-0.016425	0.034120

3.2 Нахождение значения функций в точках с помощью метода Рунге-Кутты второй степени, с шагом 0.075

t	x1	x2
0.075000	2.191385	-0.924101
0.150000	3.327612	-3.980367
0.225000	6.662864	-11.242780
0.300000	15.374183	-29.114284
0.375000	37.435133	-73.578402
0.450000	92.802675	-184.576165
0.525000	231.386384	-461.946912
0.600000	577.972976	-1155.279112
0.675000	1444.538760	-2888.536576
0.750000	3611.031590	-7221.623342
0.825000	9027.326373	-18054.295411

0.900000	22568.114293	-45135.939735
0.975000	56420.126218	-112840.021436
1.050000	141050.191461	-282100.201631
1.125000	352625.384892	-705250.631877
1.200000	881563.394845	-1763126.690155
1.275000	2203908.442998	-4407816.820756
1.350000	5509771.084147	-11019542.133930
1.425000	13774427.705712	-27548855.404963
1.500000	34436069.276538	-68872138.571839

3.3 Нахождение оптимального шага

Оптимальный шаг вычисляется по следующей формуле:

$$h_{opt} = \frac{2}{|\lambda|_{max}}$$

Где $|\lambda|_{max}$ — максимальный модуль собственных значений матрицы Ябоки.

Матрица Якоби для заданной системы дифференциальных уравнений выглядит следующим образом:

$$J = \begin{pmatrix} -14 & 13 \\ 20 & -30 \end{pmatrix} \rightarrow \lambda 1 = -4 ; \lambda 2 = -40 \rightarrow h_{opt} = 0.05$$

3.4 Нахождение значения функций в точках с помощью метода Рунге-Кутты второй степени, с оптимальным шагом

t	x1	x2
0.050000	1.735360	0.297355
0.100000	1.516897	0.130315
0.150000	1.336300	-0.007519
0.200000	1.186755	-0.121393
0.250000	1.062678	-0.215610
0.300000	0.959490	-0.293696
0.350000	0.873440	-0.358546
0.400000	0.801452	-0.412534
0.450000	0.741003	-0.457607
0.500000	0.690029	-0.495362
0.550000	0.646835	-0.527110
0.600000	0.610034	-0.553928
0.650000	0.578488	-0.576697
0.700000	0.551266	-0.596142
0.750000	0.527605	-0.612855
0.800000	0.506883	-0.627322

0.850000	0.488590	-0.639941
0.900000	0.472310	-0.651037
0.950000	0.457704	-0.660876
1.000000	0.444498	-0.669672
1.050000	0.432469	-0.677601
1.100000	0.421437	-0.684804
1.150000	0.411257	-0.691395
1.200000	0.401815	-0.697465
1.250000	0.393019	-0.703087
1.300000	0.384797	-0.708317
1.350000	0.377093	-0.713199
1.400000	0.369866	-0.717768
1.450000	0.363084	-0.722049
1.500000	0.356722	-0.726061

3.5 Построение графиков зависимостей

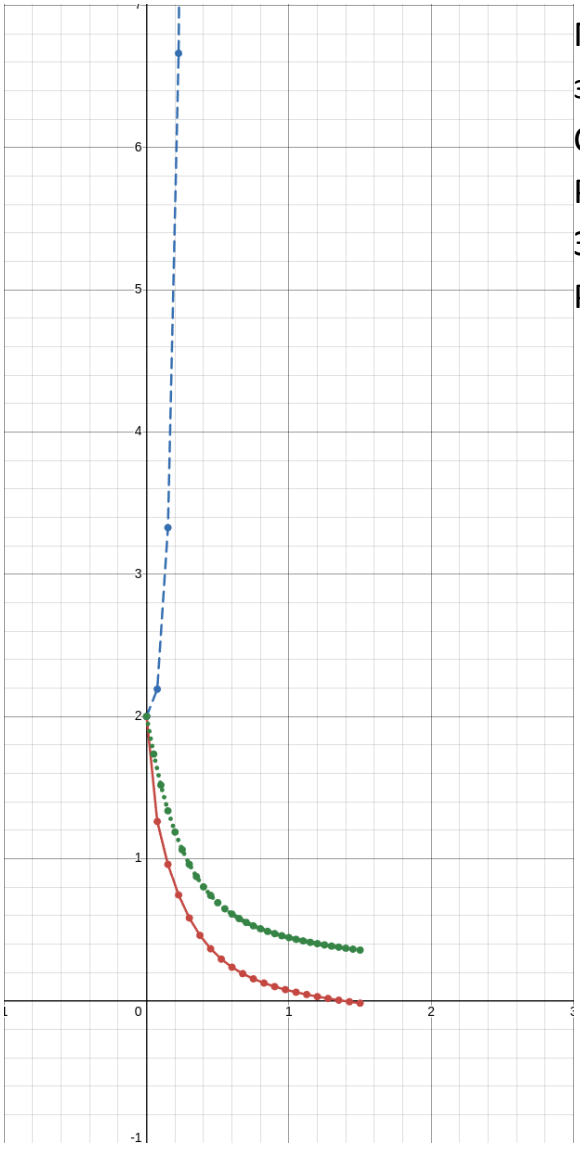

График x1(t). Красным цветом обозначены значения, вычисленные с помощью RKF45. Синим значения, вычисленные методом Рунге-Кунты второй степени с шагом 0.075. Зелёным значения, вычисленные методом Рунге-Кунты второй степени с шагом 0.05.

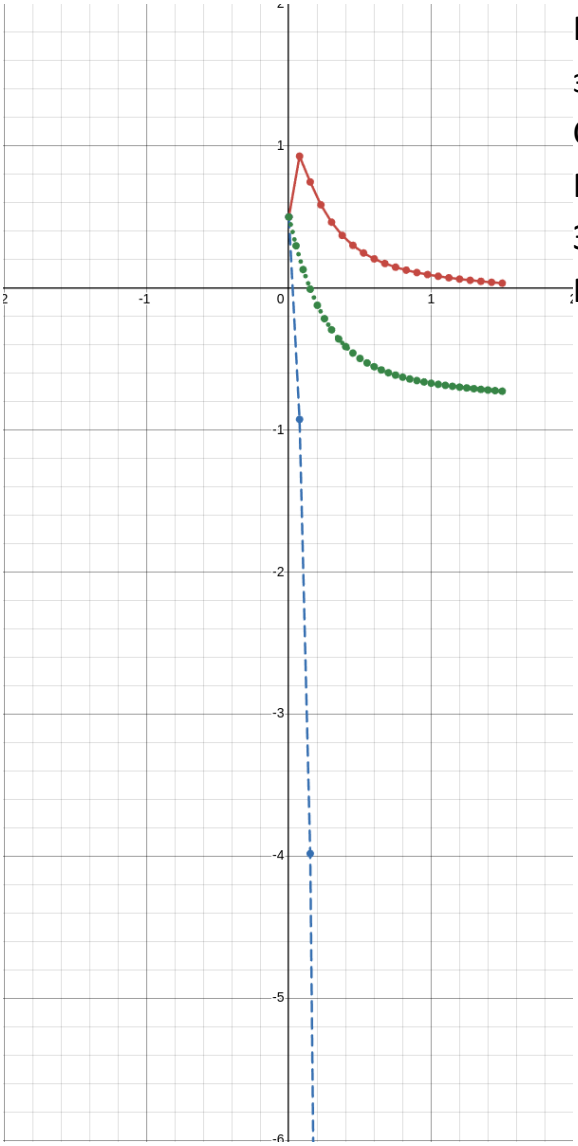

График x2(t). Красным цветом обозначены значения, вычисленные с помощью RKF45. Синим значения, вычисленные методом Рунге-Кунты второй степени с шагом 0.075. Зелёным значения, вычисленные методом Рунге-Кунты второй степени с шагом 0.05.

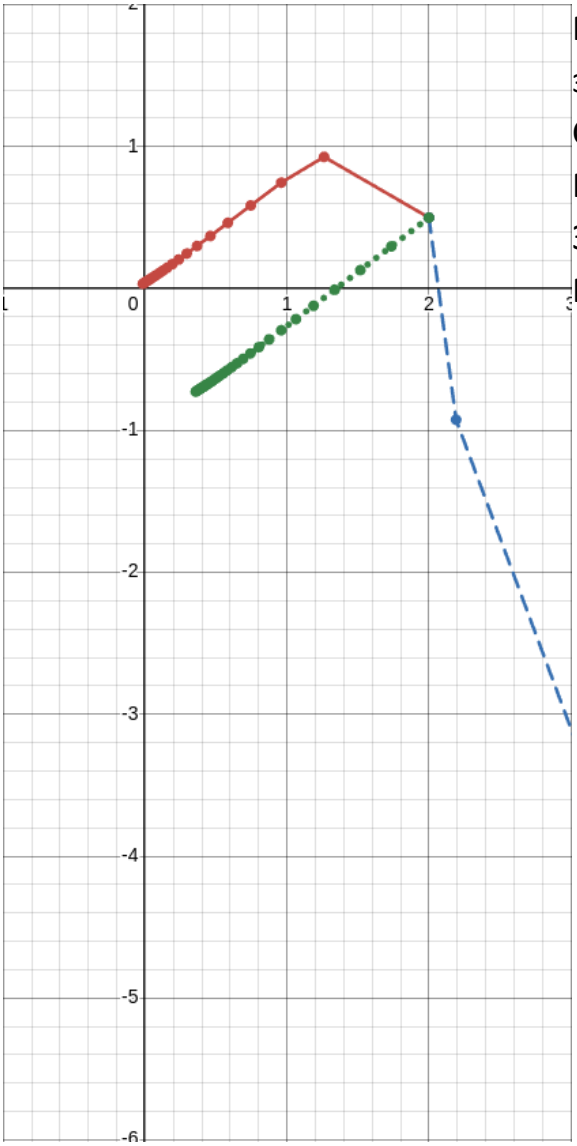

График x2(x1). Красным цветом обозначены значения, вычисленные с помощью RKF45. Синим значения, вычисленные методом Рунге-Кунты второй степени с шагом 0.075. Зелёным значения, вычисленные методом Рунге-Кунты второй степени с шагом 0.05.

4. Выводы

В ходе выполнения лабораторной работы были вычислены значения функций в промежутке $t \in [0, 1.5]$ с помощью RKF45, с погрешностью 0.0001 и шагом 0.075, с помощью метода Рунге-Кутты второй степени с шагом 0.075 и с помощью метода Рунге-Кутты второй степени с шагом 0.05.

При вычисление с помощью метода Рунге-Кутты второй степени с шагом 0.075 значения кардинально отличают, о вычисленных с помощью RKF45 с тем же шагом. Минимальная разница равна 0.191385, максимальная равна 68872138.606.

При вычисление с помощью метода Рунге-Кутты второй степени с оптимальным шагом (0.05) значения отличаются гораздо меньше, однако погрешность всё ещё довольно высока. Минимальная разница равна 0.040209, максимальная равна 0.742041.

При дальнейшем уменьшение шага, по очевидным причинам, разница уменьшается. Так при шаге 0.01 минимальная разница равна 0.000050, максимальная равна 0.010032, а при шаге 0.001 минимальная разница равна равна нулю, при округлению до шестого порядка, максимальная равна 0.000147.

5. Дополнения

5.1 Полный код всех программ

Описание файлов:

main.cpp отвечает за начальные данные и их вывод в консоль

myFunc.* содержит функцию, являющуюся реализацией метода Рунге-Кутты второй степени

rkf45.* содержит ряд алгоритмов Рунге-Кутты, в том числе RKF45

main.cpp

```
#include <iostream>
#include <cmath>
#include <iomanip>

#include "rkf45.hpp"
#include "myFunc.hpp"

void f (double t, double x[], double xp[])
{
  xp[0] = -14.0 * x[0] + 13.0 * x[1] + std::cos(1.0 + t);
  xp[1] = 20.0 * x[0] - 30.0 * x[1] + std::atan(1.0 + t * t);
}

int main()
{
  std::cout <<  std::fixed << std::setprecision(6);

  double eps = 0.0001;
  double xStart[2] = {2, 0.5};
  double xpStart[2] = {0, 0};
  double tStart = 0;
  double standartStep = 0.075;
  double t = 0;

  f(0, xStart, xpStart);
  std::cout << "\033[1;32mStarting values:\033[1;0m\nx1(0) = " << xStart[0] << "; x2(0) = " << xStart[1] <<
"\ndx1/xdt(0) = " << xpStart[0] << "; dx2/xdt(0) = " << xpStart[1] << "\n\n\033[1;32mrkf45:\033[1;0m\n";
  for (t = 0; t < 1.5; t += standartStep)
  {
    r8_rkf45(f, 2, xStart, xpStart, &tStart, t, &eps, eps, 1);
    std::cout << "t = " << t << ";\tx1 = " << xStart[0] << ";\tx2 = " << xStart[1] << '\n';
  }

  std::cout << "\n\033[1;32mrk2 with 0.075 step:\033[1;0m\n";
  xStart[0] = 2;
  xStart[1] = 0.5;
  for (t = standartStep; t < 1.51; t += standartStep)
  {
    rk2(f, xStart, t, standartStep);
```

```
    std::cout << "t = " << t << ";\tx1 = " << xStart[0] << ";\tx2 = " << xStart[1] << '\n';
  }

  double bestStep = 0.05;
  std::cout << "\n\033[1;32mrk2 with optimal step (0.05):\033[1;0m\n";
  xStart[0] = 2;
  xStart[1] = 0.5;
  for (t = bestStep; t < 1.51; t += bestStep)
  {
    rk2(f, xStart, t, bestStep);
    std::cout << "t = " << t << ";\tx1 = " << xStart[0] << ";\tx2 = " << xStart[1] << '\n';
  }

  std::cout << "\n\033[1;32mrkf45 with 0.05 step:\033[1;0m\n";
  xStart[0] = 2;
  xStart[1] = 0.5;
  tStart = 0;
  f(0, xStart, xpStart);
  for (t = 0; t < 1.5; t += bestStep)
  {
    r8_rkf45(f, 2, xStart, xpStart, &tStart, t, &eps, eps, 1);
    std::cout << "t = " << t << ";\tx1 = " << xStart[0] << ";\tx2 = " << xStart[1] << '\n';
  }

  return 0;
}
```

myFunc.hpp

```
#ifndef MYFUNC_HPP
#define MYFUNC_HPP

void rk2(void f (double t, double y[], double yp[]), double y[], double t, double h);

#endif
```

myFunc.cpp

```
#include "myFunc.hpp"

void rk2(void f (double t, double y[], double yp[]), double y[], double t, double h)
{
  double k1[2];
  f(t, y, k1);
  double z23[2];
  for (int i = 0; i < 2; ++i)
  {
    z23[i] = y[i] + ((2.0 / 3.0) * h) * k1[i];
  }

  double k2[2];
  f(t + (2.0 / 3.0) * h, z23, k2);
  for (int i = 0; i < 2; ++i)
  {
    y[i] = y[i] + (h / 4.0) * (k1[i] + 3.0 * k2[i]);
  }
}
```

rkf45.hpp

```
void r4_fehl ( void f ( float t, float y[], float yp[] ), int neqn,
  float y[], float t, float h, float yp[], float f1[], float f2[], float f3[],
  float f4[], float f5[], float s[] );
int r4_rkf45 ( void f ( float t, float y[], float yp[] ), int neqn,
  float y[], float yp[], float *t, float tout, float *relerr, float abserr,
  int flag );
float r4_sign ( float x );

void r8_fehl ( void f ( double t, double y[], double yp[] ), int neqn,
  double y[], double t, double h, double yp[], double f1[], double f2[], double f3[],
  double f4[], double f5[], double s[] );
int r8_rkf45 ( void f ( double t, double y[], double yp[] ), int neqn,
  double y[], double yp[], double *t, double tout, double *relerr, double abserr,
  int flag );
double r8_sign ( double x );

void timestamp ( );
```

rkf45.cpp

```
# include <cfloat>
# include <cmath>
# include <cstdlib>
# include <ctime>
# include <iomanip>
# include <iostream>

using namespace std;

# include "rkf45.hpp"

//****************************************************************************80

void r4_fehl ( void f ( float t, float y[], float yp[] ), int neqn,
  float y[], float t, float h, float yp[], float f1[], float f2[], float f3[],
  float f4[], float f5[], float s[] )

//****************************************************************************80
//
//  Purpose:
//
//    R4_FEHL takes one Fehlberg fourth-fifth order step.
//
//  Discussion:
//
//    This version of the routine uses FLOAT real arithmetic.
//
//    This routine integrates a system of NEQN first order ordinary differential
//    equations of the form
//      dY(i)/dT = F(T,Y(1:NEQN))
//    where the initial values Y and the initial derivatives
//    YP are specified at the starting point T.
//
//    The routine advances the solution over the fixed step H and returns
//    the fifth order (sixth order accurate locally) solution
```

```
//    approximation at T+H in array S.
//
//    The formulas have been grouped to control loss of significance.
//    The routine should be called with an H not smaller than 13 units of
//    roundoff in T so that the various independent arguments can be
//    distinguished.
//
//  Licensing:
//
//    This code is distributed under the MIT license.
//
//  Modified:
//
//    27 March 2004
//
//  Author:
//
//    Original FORTRAN77 version by Herman Watts, Lawrence Shampine.
//    C++ version by John Burkardt
//
//  Reference:
//
//    Erwin Fehlberg,
//    Low-order Classical Runge-Kutta Formulas with Stepsize Control,
//    NASA Technical Report R-315, 1969.
//
//    Lawrence Shampine, Herman Watts, S Davenport,
//    Solving Non-stiff Ordinary Differential Equations - The State of the Art,
//    SIAM Review,
//    Volume 18, pages 376-411, 1976.
//
//  Parameters:
//
//    Input, external F, a user-supplied subroutine to evaluate the
//    derivatives Y'(T), of the form:
//
//      void f ( double t, double y[], double yp[] )
//
//    Input, int NEQN, the number of equations to be integrated.
//
//    Input, float Y[NEQN], the current value of the dependent variable.
//
//    Input, float T, the current value of the independent variable.
//
//    Input, float H, the step size to take.
//
//    Input, float YP[NEQN], the current value of the derivative of the
//    dependent variable.
//
//    Output, float F1[NEQN], F2[NEQN], F3[NEQN], F4[NEQN], F5[NEQN], derivative
//    values needed for the computation.
//
//    Output, float S[NEQN], the estimate of the solution at T+H.
//
{
  float ch;
```

```
int i;

ch = h / 4.0;

for ( i = 0; i < neqn; i++ )
{
  f5[i] = y[i] + ch * yp[i];
}

f ( t + ch, f5, f1 );

ch = 3.0 * h / 32.0;

for ( i = 0; i < neqn; i++ )
{
  f5[i] = y[i] + ch * ( yp[i] + 3.0 * f1[i] );
}

f ( t + 3.0 * h / 8.0, f5, f2 );

ch = h / 2197.0;

for ( i = 0; i < neqn; i++ )
{
  f5[i] = y[i] + ch *
  ( 1932.0 * yp[i]
  + ( 7296.0 * f2[i] - 7200.0 * f1[i] )
  );
}

f ( t + 12.0 * h / 13.0, f5, f3 );

ch = h / 4104.0;

for ( i = 0; i < neqn; i++ )
{
  f5[i] = y[i] + ch *
  (
    ( 8341.0 * yp[i] - 845.0 * f3[i] )
  + ( 29440.0 * f2[i] - 32832.0 * f1[i] )
  );
}

f ( t + h, f5, f4 );

ch = h / 20520.0;

for ( i = 0; i < neqn; i++ )
{
  f1[i] = y[i] + ch *
  (
    ( -6080.0 * yp[i]
    + ( 9295.0 * f3[i] - 5643.0 * f4[i] )
    )
  + ( 41040.0 * f1[i] - 28352.0 * f2[i] )
  );
```

```
  }

  f ( t + h / 2.0, f1, f5 );
//
//  Ready to compute the approximate solution at T+H.
//
  ch = h / 7618050.0;

  for ( i = 0; i < neqn; i++ )
  {
    s[i] = y[i] + ch *
    (
      ( 902880.0 * yp[i]
      + ( 3855735.0 * f3[i] - 1371249.0 * f4[i] ) )
    + ( 3953664.0 * f2[i] + 277020.0 * f5[i] )
    );
  }

  return;
}
//****************************************************************************80

int r4_rkf45 ( void f ( float t, float y[], float yp[] ), int neqn,
  float y[], float yp[], float *t, float tout, float *relerr, float abserr,
  int flag )

//****************************************************************************80
//
//  Purpose:
//
//    R4_RKF45 carries out the Runge-Kutta-Fehlberg method.
//
//  Discussion:
//
//    This version of the routine uses FLOAT real arithmetic.
//
//    This routine is primarily designed to solve non-stiff and mildly stiff
//    differential equations when derivative evaluations are inexpensive.
//    It should generally not be used when the user is demanding
//    high accuracy.
//
//    This routine integrates a system of NEQN first-order ordinary differential
//    equations of the form:
//
//      dY(i)/dT = F(T,Y(1),Y(2),...,Y(NEQN))
//
//    where the Y(1:NEQN) are given at T.
//
//    Typically the subroutine is used to integrate from T to TOUT but it
//    can be used as a one-step integrator to advance the solution a
//    single step in the direction of TOUT.  On return, the parameters in
//    the call list are set for continuing the integration.  The user has
//    only to call again (and perhaps define a new value for TOUT).
//
//    Before the first call, the user must
//
```

```
//   * supply the subroutine F(T,Y,YP) to evaluate the right hand side;
//     and declare F in an EXTERNAL statement;
//
//   * initialize the parameters:
//     NEQN, Y(1:NEQN), T, TOUT, RELERR, ABSERR, FLAG.
//     In particular, T should initially be the starting point for integration,
//     Y should be the value of the initial conditions, and FLAG should
//     normally be +1.
//
//   Normally, the user only sets the value of FLAG before the first call, and
//   thereafter, the program manages the value.  On the first call, FLAG should
//   normally be +1 (or -1 for single step mode.)  On normal return, FLAG will
//   have been reset by the program to the value of 2 (or -2 in single
//   step mode), and the user can continue to call the routine with that
//   value of FLAG.
//
//   (When the input magnitude of FLAG is 1, this indicates to the program
//   that it is necessary to do some initialization work.  An input magnitude
//   of 2 lets the program know that that initialization can be skipped,
//   and that useful information was computed earlier.)
//
//   The routine returns with all the information needed to continue
//   the integration.  If the integration reached TOUT, the user need only
//   define a new TOUT and call again.  In the one-step integrator
//   mode, returning with FLAG = -2, the user must keep in mind that
//   each step taken is in the direction of the current TOUT.  Upon
//   reaching TOUT, indicated by the output value of FLAG switching to 2,
//   the user must define a new TOUT and reset FLAG to -2 to continue
//   in the one-step integrator mode.
//
//   In some cases, an error or difficulty occurs during a call.  In that case,
//   the output value of FLAG is used to indicate that there is a problem
//   that the user must address.  These values include:
//
//   * 3, integration was not completed because the input value of RELERR, the
//     relative error tolerance, was too small.  RELERR has been increased
//     appropriately for continuing.  If the user accepts the output value of
//     RELERR, then simply reset FLAG to 2 and continue.
//
//   * 4, integration was not completed because more than MAXNFE derivative
//     evaluations were needed.  This is approximately (MAXNFE/6) steps.
//     The user may continue by simply calling again.  The function counter
//     will be reset to 0, and another MAXNFE function evaluations are allowed.
//
//   * 5, integration was not completed because the solution vanished,
//     making a pure relative error test impossible.  The user must use
//     a non-zero ABSERR to continue.  Using the one-step integration mode
//     for one step is a good way to proceed.
//
//   * 6, integration was not completed because the requested accuracy
//     could not be achieved, even using the smallest allowable stepsize.
//     The user must increase the error tolerances ABSERR or RELERR before
//     continuing.  It is also necessary to reset FLAG to 2 (or -2 when
//     the one-step integration mode is being used).  The occurrence of
//     FLAG = 6 indicates a trouble spot.  The solution is changing
//     rapidly, or a singularity may be present.  It often is inadvisable
```

```
//     to continue.
//
//   * 7, it is likely that this routine is inefficient for solving
//     this problem.  Too much output is restricting the natural stepsize
//     choice.  The user should use the one-step integration mode with
//     the stepsize determined by the code.  If the user insists upon
//     continuing the integration, reset FLAG to 2 before calling
//     again.  Otherwise, execution will be terminated.
//
//   * 8, invalid input parameters, indicates one of the following:
//     NEQN <= 0;
//     T = TOUT and |FLAG| /= 1;
//     RELERR < 0 or ABSERR < 0;
//     FLAG == 0  or FLAG < -2 or 8 < FLAG.
//
//  Licensing:
//
//    This code is distributed under the MIT license.
//
//  Modified:
//
//    05 April 2011
//
//  Author:
//
//    Original FORTRAN77 version by Herman Watts, Lawrence Shampine.
//    C++ version by John Burkardt.
//
//  Reference:
//
//    Erwin Fehlberg,
//    Low-order Classical Runge-Kutta Formulas with Stepsize Control,
//    NASA Technical Report R-315, 1969.
//
//    Lawrence Shampine, Herman Watts, S Davenport,
//    Solving Non-stiff Ordinary Differential Equations - The State of the Art,
//    SIAM Review,
//    Volume 18, pages 376-411, 1976.
//
//  Parameters:
//
//    Input, external F, a user-supplied subroutine to evaluate the
//    derivatives Y'(T), of the form:
//
//      void f ( float t, float y[], float yp[] )
//
//    Input, int NEQN, the number of equations to be integrated.
//
//    Input/output, float Y[NEQN], the current solution vector at T.
//
//    Input/output, float YP[NEQN], the derivative of the current solution
//    vector at T.  The user should not set or alter this information!
//
//    Input/output, float *T, the current value of the independent variable.
//
//    Input, float TOUT, the output point at which solution is desired.
```

```
//   TOUT = T is allowed on the first call only, in which case the routine
//   returns with FLAG = 2 if continuation is possible.
//
//   Input, float *RELERR, ABSERR, the relative and absolute error tolerances
//   for the local error test.  At each step the code requires:
//     abs ( local error ) <= RELERR * abs ( Y ) + ABSERR
//   for each component of the local error and the solution vector Y.
//   RELERR cannot be "too small".  If the routine believes RELERR has been
//   set too small, it will reset RELERR to an acceptable value and return
//   immediately for user action.
//
//   Input, int FLAG, indicator for status of integration. On the first call,
//   set FLAG to +1 for normal use, or to -1 for single step mode.  On
//   subsequent continuation steps, FLAG should be +2, or -2 for single
//   step mode.
//
//   Output, int RKF45_S, indicator for status of integration.  A value of 2
//   or -2 indicates normal progress, while any other value indicates a
//   problem that should be addressed.
//
{
# define MAXNFE 3000

  static float abserr_save = -1.0;
  float ae;
  float dt;
  float ee;
  float eeoet;
  float eps;
  float esttol;
  float et;
  float *f1;
  float *f2;
  float *f3;
  float *f4;
  float *f5;
  int flag_return;
  static int flag_save = -1000;
  static float h = -1.0;
  bool hfaild;
  float hmin;
  int i;
  static int init = -1000;
  int k;
  static int kflag = -1000;
  static int kop = -1;
  int mflag;
  static int nfe = -1;
  bool output;
  float relerr_min;
  static float relerr_save = -1.0;
  static float remin = 1.0E-12;
  float s;
  float scale;
  float tol;
  float toln;
```

```
  float ypk;

  flag_return = flag;
//
//  Check the input parameters.
//
  eps = FLT_EPSILON;

  if ( neqn < 1 )
  {
    flag_return = 8;
    return flag_return;
  }

  if ( ( *relerr ) < 0.0 )
  {
    flag_return = 8;
    return flag_return;
  }

  if ( abserr < 0.0 )
  {
    flag_return = 8;
    return flag_return;
  }

  if ( flag_return == 0 || 8 < flag_return  || flag_return < -2 )
  {
    flag_return = 8;
    return flag_return;
  }

  mflag = abs ( flag_return );
//
//  Is this a continuation call?
//
  if ( mflag != 1 )
  {
    if ( *t == tout && kflag != 3 )
    {
      flag_return = 8;
      return flag_return;
    }
//
//  FLAG = -2 or +2:
//
    if ( mflag == 2 )
    {
      if ( kflag == 3 )
      {
        flag_return = flag_save;
        mflag = abs ( flag_return );
      }
      else if ( init == 0 )
      {
        flag_return = flag_save;
```

```
    }
    else if ( kflag == 4 )
    {
      nfe = 0;
    }
    else if ( kflag == 5 && abserr == 0.0 )
    {
      cerr << "\n";
      cerr << "R4_RKF45 - Fatal error!\n";
      cerr << "  KFLAG = 5 and ABSERR = 0.0\n";
      exit ( 1 );
    }
    else if ( kflag == 6 && (*relerr) <= relerr_save && abserr <= abserr_save )
    {
      cerr << "\n";
      cerr << "R4_RKF45 - Fatal error!\n";
      cerr << "  KFLAG = 6 and\n";
      cerr << "  RELERR <= RELERR_SAVE and\n";
      cerr << "  ABSERR <= ABSERR_SAVE\n";
      exit ( 1 );
    }
  }
//
//  FLAG = 3, 4, 5, 6, 7 or 8.
//
  else
  {
    if ( flag_return == 3 )
    {
      flag_return = flag_save;
      if ( kflag == 3 )
      {
        mflag = abs ( flag_return );
      }
    }
    else if ( flag_return == 4 )
    {
      nfe = 0;
      flag_return = flag_save;
      if ( kflag == 3 )
      {
        mflag = abs ( flag_return );
      }
    }
    else if ( flag_return == 5 && 0.0 < abserr )
    {
      flag_return = flag_save;
      if ( kflag == 3 )
      {
        mflag = abs ( flag_return );
      }
    }
//
//  Integration cannot be continued because the user did not respond to
//  the instructions pertaining to FLAG = 5, 6, 7 or 8.
//
```

```
    else
    {
      cerr << "\n";
      cerr << "R4_RKF45 - Fatal error!\n";
      cerr << "  Integration cannot be continued.\n";
      cerr << "  The user did not respond to the output\n";
      cerr << "  value FLAG = 5, 6, 7, or 8.\n";
      exit ( 1 );
    }
  }
 }
//
//  Save the input value of FLAG.
//  Set the continuation flag KFLAG for subsequent input checking.
//
 flag_save = flag_return;
 kflag = 0;
//
//  Save RELERR and ABSERR for checking input on subsequent calls.
//
 relerr_save = (*relerr);
 abserr_save = abserr;
//
//  Restrict the relative error tolerance to be at least
//
//    2*EPS+REMIN
//
//  to avoid limiting precision difficulties arising from impossible
//  accuracy requests.
//
 relerr_min = 2.0 * FLT_EPSILON + remin;
//
//  Is the relative error tolerance too small?
//
 if ( (*relerr) < relerr_min )
 {
   (*relerr) = relerr_min;
   kflag = 3;
   flag_return = 3;
   return flag_return;
 }

 dt = tout - *t;
//
//  Initialization:
//
//  Set the initialization completion indicator, INIT;
//  set the indicator for too many output points, KOP;
//  evaluate the initial derivatives
//  set the counter for function evaluations, NFE;
//  estimate the starting stepsize.
//
 f1 = new float[neqn];
 f2 = new float[neqn];
 f3 = new float[neqn];
 f4 = new float[neqn];
```

```
  f5 = new float[neqn];

  if ( mflag == 1 )
  {
    init = 0;
    kop = 0;
    f ( *t, y, yp );
    nfe = 1;

    if ( *t == tout )
    {
      flag_return = 2;
      return flag_return;
    }

  }

  if ( init == 0 )
  {
    init = 1;
    h = fabs ( dt );
    toln = 0.0;

    for ( k = 0; k < neqn; k++ )
    {
      tol = (*relerr) * fabs ( y[k] ) + abserr;
      if ( 0.0 < tol )
      {
        toln = tol;
        ypk = fabs ( yp[k] );
        if ( tol < ypk * pow ( h, 5 ) )
        {
          h = ( float ) pow ( ( double ) ( tol / ypk ), 0.2 );
        }
      }
    }

    if ( toln <= 0.0 )
    {
      h = 0.0;
    }

    h = fmax ( h, 26.0 * eps * fmax ( fabs ( *t ), fabs ( dt ) ) );

    if ( flag_return < 0 )
    {
      flag_save = -2;
    }
    else
    {
      flag_save = 2;
    }
  }
//
//  Set stepsize for integration in the direction from T to TOUT.
//
```

```
  h = r4_sign ( dt ) * fabs ( h );
//
//  Test to see if too may output points are being requested.
//
  if ( 2.0 * fabs ( dt ) <= fabs ( h ) )
  {
    kop = kop + 1;
  }
//
//  Unnecessary frequency of output.
//
  if ( kop == 100 )
  {
    kop = 0;
    delete [] f1;
    delete [] f2;
    delete [] f3;
    delete [] f4;
    delete [] f5;
    flag_return = 7;
    return flag_return;
  }
//
//  If we are too close to the output point, then simply extrapolate and return.
//
  if ( fabs ( dt ) <= 26.0 * eps * fabs ( *t ) )
  {
    *t = tout;
    for ( i = 0; i < neqn; i++ )
    {
      y[i] = y[i] + dt * yp[i];
    }
    f ( *t, y, yp );
    nfe = nfe + 1;

    delete [] f1;
    delete [] f2;
    delete [] f3;
    delete [] f4;
    delete [] f5;
    flag_return = 2;
    return flag_return;
  }
//
//  Initialize the output point indicator.
//
  output = false;
//
//  To avoid premature underflow in the error tolerance function,
//  scale the error tolerances.
//
  scale = 2.0 / (*relerr);
  ae = scale * abserr;
//
//  Step by step integration.
//
```

```
 for ( ; ; )
 {
   hfaild = false;
//
//  Set the smallest allowable stepsize.
//
   hmin = 26.0 * eps * fabs ( *t );
//
//  Adjust the stepsize if necessary to hit the output point.
//
//  Look ahead two steps to avoid drastic changes in the stepsize and
//  thus lessen the impact of output points on the code.
//
   dt = tout - *t;

   if ( 2.0 * fabs ( h ) <= fabs ( dt ) )
   {
   }
   else
//
//  Will the next successful step complete the integration to the output point?
//
   {
     if ( fabs ( dt ) <= fabs ( h ) )
     {
       output = true;
       h = dt;
     }
     else
     {
       h = 0.5 * dt;
     }

   }
//
//  Here begins the core integrator for taking a single step.
//
//  The tolerances have been scaled to avoid premature underflow in
//  computing the error tolerance function ET.
//  To avoid problems with zero crossings, relative error is measured
//  using the average of the magnitudes of the solution at the
//  beginning and end of a step.
//  The error estimate formula has been grouped to control loss of
//  significance.
//
//  To distinguish the various arguments, H is not permitted
//  to become smaller than 26 units of roundoff in T.
//  Practical limits on the change in the stepsize are enforced to
//  smooth the stepsize selection process and to avoid excessive
//  chattering on problems having discontinuities.
//  To prevent unnecessary failures, the code uses 9/10 the stepsize
//  it estimates will succeed.
//
//  After a step failure, the stepsize is not allowed to increase for
//  the next attempted step.  This makes the code more efficient on
//  problems having discontinuities and more effective in general
```

```
//  since local extrapolation is being used and extra caution seems
//  warranted.
//
//  Test the number of derivative function evaluations.
//  If okay, try to advance the integration from T to T+H.
//
   for ( ; ; )
   {
//
//  Have we done too much work?
//
     if ( MAXNFE < nfe )
     {
       kflag = 4;
       delete [] f1;
       delete [] f2;
       delete [] f3;
       delete [] f4;
       delete [] f5;
       flag_return = 4;
       return flag_return;
     }
//
//  Advance an approximate solution over one step of length H.
//
     r4_fehl ( f, neqn, y, *t, h, yp, f1, f2, f3, f4, f5, f1 );
     nfe = nfe + 5;
//
//  Compute and test allowable tolerances versus local error estimates
//  and remove scaling of tolerances.  The relative error is
//  measured with respect to the average of the magnitudes of the
//  solution at the beginning and end of the step.
//
     eeoet = 0.0;

     for ( k = 0; k < neqn; k++ )
     {
       et = fabs ( y[k] ) + fabs ( f1[k] ) + ae;

       if ( et <= 0.0 )
       {
         delete [] f1;
         delete [] f2;
         delete [] f3;
         delete [] f4;
         delete [] f5;
         flag_return = 5;
         return flag_return;
       }

       ee = fabs
       ( ( -2090.0 * yp[k]
         + ( 21970.0 * f3[k] - 15048.0 * f4[k] )
         )
       + ( 22528.0 * f2[k] - 27360.0 * f5[k] )
       );
```

```
      eeoet = fmax ( eeoet, ee / et );

    }

    esttol = fabs ( h ) * eeoet * scale / 752400.0;

    if ( esttol <= 1.0 )
    {
      break;
    }
//
//  Unsuccessful step.  Reduce the stepsize, try again.
//  The decrease is limited to a factor of 1/10.
//
    hfaild = true;
    output = false;

    if ( esttol < 59049.0 )
    {
      s = 0.9 / ( float ) pow ( ( double ) esttol, 0.2 );
    }
    else
    {
      s = 0.1;
    }

    h = s * h;

    if ( fabs ( h ) < hmin )
    {
      kflag = 6;
      delete [] f1;
      delete [] f2;
      delete [] f3;
      delete [] f4;
      delete [] f5;
      flag_return = 6;
      return flag_return;
    }

  }
//
//  We exited the loop because we took a successful step.
//  Store the solution for T+H, and evaluate the derivative there.
//
  *t = *t + h;
  for ( i = 0; i < neqn; i++ )
  {
    y[i] = f1[i];
  }
  f ( *t, y, yp );
  nfe = nfe + 1;
//
//  Choose the next stepsize.  The increase is limited to a factor of 5.
//  If the step failed, the next stepsize is not allowed to increase.
```

```
//
   if ( 0.0001889568 < esttol )
   {
     s = 0.9 / ( float ) pow ( ( double ) esttol, 0.2 );
   }
   else
   {
     s = 5.0;
   }

   if ( hfaild )
   {
     s = fmin ( s, 1.0 );
   }

   h = r4_sign ( h ) * fmax ( s * fabs ( h ), hmin );
//
//  End of core integrator
//
//  Should we take another step?
//
   if ( output )
   {
     *t = tout;
     delete [] f1;
     delete [] f2;
     delete [] f3;
     delete [] f4;
     delete [] f5;
     flag_return = 2;
     return flag_return;
   }

   if ( flag_return <= 0 )
   {
     delete [] f1;
     delete [] f2;
     delete [] f3;
     delete [] f4;
     delete [] f5;
     flag_return = -2;
     return flag_return;
   }

  }
# undef MAXNFE
}
//****************************************************************************80

float r4_sign ( float x )

//****************************************************************************80
//
//  Purpose:
//
//    R4_SIGN returns the sign of an R4.
```

```
//
//  Licensing:
//
//    This code is distributed under the MIT license.
//
//  Modified:
//
//    27 March 2004
//
//  Author:
//
//    John Burkardt
//
//  Parameters:
//
//    Input, float X, the number whose sign is desired.
//
//    Output, float R4_SIGN, the sign of X.
//
{
  if ( x < 0.0 )
  {
    return ( -1.0 );
  }
  else
  {
    return ( +1.0 );
  }
}
//****************************************************************************80

void r8_fehl ( void f ( double t, double y[], double yp[] ), int neqn,
  double y[], double t, double h, double yp[], double f1[], double f2[],
  double f3[], double f4[], double f5[], double s[] )

//****************************************************************************80
//
//  Purpose:
//
//    R8_FEHL takes one Fehlberg fourth-fifth order step.
//
//  Discussion:
//
//    This version of the routine uses DOUBLE real arithemtic.
//
//    This routine integrates a system of NEQN first order ordinary differential
//    equations of the form
//      dY(i)/dT = F(T,Y(1:NEQN))
//    where the initial values Y and the initial derivatives
//    YP are specified at the starting point T.
//
//    The routine advances the solution over the fixed step H and returns
//    the fifth order (sixth order accurate locally) solution
//    approximation at T+H in array S.
//
//    The formulas have been grouped to control loss of significance.
```

```
//    The routine should be called with an H not smaller than 13 units of
//    roundoff in T so that the various independent arguments can be
//    distinguished.
//
//  Licensing:
//
//    This code is distributed under the MIT license.
//
//  Modified:
//
//    27 March 2004
//
//  Author:
//
//    Original FORTRAN77 version by Herman Watts, Lawrence Shampine.
//    C++ version by John Burkardt.
//
//  Reference:
//
//    Erwin Fehlberg,
//    Low-order Classical Runge-Kutta Formulas with Stepsize Control,
//    NASA Technical Report R-315, 1969.
//
//    Lawrence Shampine, Herman Watts, S Davenport,
//    Solving Non-stiff Ordinary Differential Equations - The State of the Art,
//    SIAM Review,
//    Volume 18, pages 376-411, 1976.
//
//  Parameters:
//
//    Input, external F, a user-supplied subroutine to evaluate the
//    derivatives Y'(T), of the form:
//
//      void f ( double t, double y[], double yp[] )
//
//    Input, int NEQN, the number of equations to be integrated.
//
//    Input, double Y[NEQN], the current value of the dependent variable.
//
//    Input, double T, the current value of the independent variable.
//
//    Input, double H, the step size to take.
//
//    Input, double YP[NEQN], the current value of the derivative of the
//    dependent variable.
//
//    Output, double F1[NEQN], F2[NEQN], F3[NEQN], F4[NEQN], F5[NEQN], derivative
//    values needed for the computation.
//
//    Output, double S[NEQN], the estimate of the solution at T+H.
//
{
  double ch;
  int i;

  ch = h / 4.0;
```

```
for ( i = 0; i < neqn; i++ )
{
  f5[i] = y[i] + ch * yp[i];
}

f ( t + ch, f5, f1 );

ch = 3.0 * h / 32.0;

for ( i = 0; i < neqn; i++ )
{
  f5[i] = y[i] + ch * ( yp[i] + 3.0 * f1[i] );
}

f ( t + 3.0 * h / 8.0, f5, f2 );

ch = h / 2197.0;

for ( i = 0; i < neqn; i++ )
{
  f5[i] = y[i] + ch *
  ( 1932.0 * yp[i]
  + ( 7296.0 * f2[i] - 7200.0 * f1[i] )
  );
}

f ( t + 12.0 * h / 13.0, f5, f3 );

ch = h / 4104.0;

for ( i = 0; i < neqn; i++ )
{
  f5[i] = y[i] + ch *
  (
    ( 8341.0 * yp[i] - 845.0 * f3[i] )
  + ( 29440.0 * f2[i] - 32832.0 * f1[i] )
  );
}

f ( t + h, f5, f4 );

ch = h / 20520.0;

for ( i = 0; i < neqn; i++ )
{
  f1[i] = y[i] + ch *
  (
    ( -6080.0 * yp[i]
    + ( 9295.0 * f3[i] - 5643.0 * f4[i] )
    )
  + ( 41040.0 * f1[i] - 28352.0 * f2[i] )
  );
}

f ( t + h / 2.0, f1, f5 );
```

```
//
//  Ready to compute the approximate solution at T+H.
//
  ch = h / 7618050.0;

  for ( i = 0; i < neqn; i++ )
  {
    s[i] = y[i] + ch *
    (
      ( 902880.0 * yp[i]
      + ( 3855735.0 * f3[i] - 1371249.0 * f4[i] ) )
    + ( 3953664.0 * f2[i] + 277020.0 * f5[i] )
    );
  }

  return;
}
//****************************************************************************80

int r8_rkf45 ( void f ( double t, double y[], double yp[] ), int neqn,
  double y[], double yp[], double *t, double tout, double *relerr,
  double abserr, int flag )

//****************************************************************************80
//
//  Purpose:
//
//    R8_RKF45 carries out the Runge-Kutta-Fehlberg method.
//
//  Discussion:
//
//    This version of the routine uses DOUBLE real arithmetic.
//
//    This routine is primarily designed to solve non-stiff and mildly stiff
//    differential equations when derivative evaluations are inexpensive.
//    It should generally not be used when the user is demanding
//    high accuracy.
//
//    This routine integrates a system of NEQN first-order ordinary differential
//    equations of the form:
//
//      dY(i)/dT = F(T,Y(1),Y(2),...,Y(NEQN))
//
//    where the Y(1:NEQN) are given at T.
//
//    Typically the subroutine is used to integrate from T to TOUT but it
//    can be used as a one-step integrator to advance the solution a
//    single step in the direction of TOUT.  On return, the parameters in
//    the call list are set for continuing the integration.  The user has
//    only to call again (and perhaps define a new value for TOUT).
//
//    Before the first call, the user must
//
//    * supply the subroutine F(T,Y,YP) to evaluate the right hand side;
//      and declare F in an EXTERNAL statement;
//
```

```
//   * initialize the parameters:
//     NEQN, Y(1:NEQN), T, TOUT, RELERR, ABSERR, FLAG.
//     In particular, T should initially be the starting point for integration,
//     Y should be the value of the initial conditions, and FLAG should
//     normally be +1.
//
//   Normally, the user only sets the value of FLAG before the first call, and
//   thereafter, the program manages the value.  On the first call, FLAG should
//   normally be +1 (or -1 for single step mode.)  On normal return, FLAG will
//   have been reset by the program to the value of 2 (or -2 in single
//   step mode), and the user can continue to call the routine with that
//   value of FLAG.
//
//   (When the input magnitude of FLAG is 1, this indicates to the program
//   that it is necessary to do some initialization work.  An input magnitude
//   of 2 lets the program know that that initialization can be skipped,
//   and that useful information was computed earlier.)
//
//   The routine returns with all the information needed to continue
//   the integration.  If the integration reached TOUT, the user need only
//   define a new TOUT and call again.  In the one-step integrator
//   mode, returning with FLAG = -2, the user must keep in mind that
//   each step taken is in the direction of the current TOUT.  Upon
//   reaching TOUT, indicated by the output value of FLAG switching to 2,
//   the user must define a new TOUT and reset FLAG to -2 to continue
//   in the one-step integrator mode.
//
//   In some cases, an error or difficulty occurs during a call.  In that case,
//   the output value of FLAG is used to indicate that there is a problem
//   that the user must address.  These values include:
//
//   * 3, integration was not completed because the input value of RELERR, the
//     relative error tolerance, was too small.  RELERR has been increased
//     appropriately for continuing.  If the user accepts the output value of
//     RELERR, then simply reset FLAG to 2 and continue.
//
//   * 4, integration was not completed because more than MAXNFE derivative
//     evaluations were needed.  This is approximately (MAXNFE/6) steps.
//     The user may continue by simply calling again.  The function counter
//     will be reset to 0, and another MAXNFE function evaluations are allowed.
//
//   * 5, integration was not completed because the solution vanished,
//     making a pure relative error test impossible.  The user must use
//     a non-zero ABSERR to continue.  Using the one-step integration mode
//     for one step is a good way to proceed.
//
//   * 6, integration was not completed because the requested accuracy
//     could not be achieved, even using the smallest allowable stepsize.
//     The user must increase the error tolerances ABSERR or RELERR before
//     continuing.  It is also necessary to reset FLAG to 2 (or -2 when
//     the one-step integration mode is being used).  The occurrence of
//     FLAG = 6 indicates a trouble spot.  The solution is changing
//     rapidly, or a singularity may be present.  It often is inadvisable
//     to continue.
//
//   * 7, it is likely that this routine is inefficient for solving
```

```
//    this problem.  Too much output is restricting the natural stepsize
//    choice.  The user should use the one-step integration mode with
//    the stepsize determined by the code.  If the user insists upon
//    continuing the integration, reset FLAG to 2 before calling
//    again.  Otherwise, execution will be terminated.
//
//   * 8, invalid input parameters, indicates one of the following:
//    NEQN <= 0;
//    T = TOUT and |FLAG| /= 1;
//    RELERR < 0 or ABSERR < 0;
//    FLAG == 0  or FLAG < -2 or 8 < FLAG.
//
//  Licensing:
//
//   This code is distributed under the MIT license.
//
//  Modified:
//
//   13 October 2012
//
//  Author:
//
//   Original FORTRAN77 version by Herman Watts, Lawrence Shampine.
//   C++ version by John Burkardt.
//
//  Reference:
//
//   Erwin Fehlberg,
//   Low-order Classical Runge-Kutta Formulas with Stepsize Control,
//   NASA Technical Report R-315, 1969.
//
//   Lawrence Shampine, Herman Watts, S Davenport,
//   Solving Non-stiff Ordinary Differential Equations - The State of the Art,
//   SIAM Review,
//   Volume 18, pages 376-411, 1976.
//
//  Parameters:
//
//   Input, external F, a user-supplied subroutine to evaluate the
//   derivatives Y'(T), of the form:
//
//     void f ( double t, double y[], double yp[] )
//
//   Input, int NEQN, the number of equations to be integrated.
//
//   Input/output, double Y[NEQN], the current solution vector at T.
//
//   Input/output, double YP[NEQN], the derivative of the current solution
//   vector at T.  The user should not set or alter this information!
//
//   Input/output, double *T, the current value of the independent variable.
//
//   Input, double TOUT, the output point at which solution is desired.
//   TOUT = T is allowed on the first call only, in which case the routine
//   returns with FLAG = 2 if continuation is possible.
//
```

```
//    Input, double *RELERR, ABSERR, the relative and absolute error tolerances
//    for the local error test.  At each step the code requires:
//      abs ( local error ) <= RELERR * abs ( Y ) + ABSERR
//    for each component of the local error and the solution vector Y.
//    RELERR cannot be "too small".  If the routine believes RELERR has been
//    set too small, it will reset RELERR to an acceptable value and return
//    immediately for user action.
//
//    Input, int FLAG, indicator for status of integration. On the first call,
//    set FLAG to +1 for normal use, or to -1 for single step mode.  On
//    subsequent continuation steps, FLAG should be +2, or -2 for single
//    step mode.
//
//    Output, int RKF45_D, indicator for status of integration.  A value of 2
//    or -2 indicates normal progress, while any other value indicates a
//    problem that should be addressed.
//
{
# define MAXNFE 3000

  static double abserr_save = -1.0;
  double ae;
  double dt;
  double ee;
  double eeoet;
  double eps;
  double esttol;
  double et;
  double *f1;
  double *f2;
  double *f3;
  double *f4;
  double *f5;
  int flag_return;
  static int flag_save = -1000;
  static double h = -1.0;
  bool hfaild;
  double hmin;
  int i;
  static int init = -1000;
  int k;
  static int kflag = -1000;
  static int kop = -1;
  int mflag;
  static int nfe = -1;
  bool output;
  double relerr_min;
  static double relerr_save = -1.0;
  static double remin = 1.0E-12;
  double s;
  double scale;
  double tol;
  double toln;
  double ypk;

  flag_return = flag;
```

```
//
//  Check the input parameters.
//
  eps = DBL_EPSILON;

  if ( neqn < 1 )
  {
    flag_return = 8;
    cerr << "\n";
    cerr << "R8_RKF45 - Fatal error!\n";
    cerr << "  Invalid input value of NEQN.\n";
    return flag_return;
  }

  if ( (*relerr) < 0.0 )
  {
    flag_return = 8;
    cerr << "\n";
    cerr << "R8_RKF45 - Fatal error!\n";
    cerr << "  Invalid input value of RELERR.\n";
    return flag_return;
  }

  if ( abserr < 0.0 )
  {
    flag_return = 8;
    cerr << "\n";
    cerr << "R8_RKF45 - Fatal error!\n";
    cerr << "  Invalid input value of ABSERR.\n";
    return flag_return;
  }

  if ( flag_return == 0 || 8 < flag_return  || flag_return < -2 )
  {
    flag_return = 8;
    cerr << "\n";
    cerr << "R8_RKF45 - Fatal error!\n";
    cerr << "  Invalid input.\n";
    return flag_return;
  }

  mflag = abs ( flag_return );
//
//  Is this a continuation call?
//
  if ( mflag != 1 )
  {
    if ( *t == tout && kflag != 3 )
    {
      flag_return = 8;
      return flag_return;
    }
//
//  FLAG = -2 or +2:
//
    if ( mflag == 2 )
```

```
  {
    if ( kflag == 3 )
    {
      flag_return = flag_save;
      mflag = abs ( flag_return );
    }
    else if ( init == 0 )
    {
      flag_return = flag_save;
    }
    else if ( kflag == 4 )
    {
      nfe = 0;
    }
    else if ( kflag == 5 && abserr == 0.0 )
    {
      cerr << "\n";
      cerr << "R8_RKF45 - Fatal error!\n";
      cerr << "  KFLAG = 5 and ABSERR = 0.0\n";
      exit ( 1 );
    }
    else if ( kflag == 6 && (*relerr) <= relerr_save && abserr <= abserr_save )
    {
      cerr << "\n";
      cerr << "R8_RKF45 - Fatal error!\n";
      cerr << "  KFLAG = 6 and\n";
      cerr << "  RELERR <= RELERR_SAVE and\n";
      cerr << "  ABSERR <= ABSERR_SAVE\n";
      exit ( 1 );
    }
  }
//
//  FLAG = 3, 4, 5, 6, 7 or 8.
//
  else
  {
    if ( flag_return == 3 )
    {
      flag_return = flag_save;
      if ( kflag == 3 )
      {
        mflag = abs ( flag_return );
      }
    }
    else if ( flag_return == 4 )
    {
      nfe = 0;
      flag_return = flag_save;
      if ( kflag == 3 )
      {
        mflag = abs ( flag_return );
      }
    }
    else if ( flag_return == 5 && 0.0 < abserr )
    {
      flag_return = flag_save;
```

```
      if ( kflag == 3 )
      {
        mflag = abs ( flag_return );
      }
    }
//
//  Integration cannot be continued because the user did not respond to
//  the instructions pertaining to FLAG = 5, 6, 7 or 8.
//
    else
    {
      cerr << "\n";
      cerr << "R8_RKF45 - Fatal error!\n";
      cerr << "  Integration cannot be continued.\n";
      cerr << "  The user did not respond to the output\n";
      cerr << "  value FLAG = 5, 6, 7, or 8.\n";
      exit ( 1 );
    }
  }
 }
//
//  Save the input value of FLAG.
//  Set the continuation flag KFLAG for subsequent input checking.
//
 flag_save = flag_return;
 kflag = 0;
//
//  Save RELERR and ABSERR for checking input on subsequent calls.
//
 relerr_save = (*relerr);
 abserr_save = abserr;
//
//  Restrict the relative error tolerance to be at least
//
//    2*EPS+REMIN
//
//  to avoid limiting precision difficulties arising from impossible
//  accuracy requests.
//
 relerr_min = 2.0 * DBL_EPSILON + remin;
//
//  Is the relative error tolerance too small?
//
 if ( (*relerr) < relerr_min )
 {
   (*relerr) = relerr_min;
   kflag = 3;
   flag_return = 3;
   return flag_return;
 }

 dt = tout - *t;
//
//  Initialization:
//
//  Set the initialization completion indicator, INIT;
```

```
//  set the indicator for too many output points, KOP;
//  evaluate the initial derivatives
//  set the counter for function evaluations, NFE;
//  estimate the starting stepsize.
//
  f1 = new double[neqn];
  f2 = new double[neqn];
  f3 = new double[neqn];
  f4 = new double[neqn];
  f5 = new double[neqn];

  if ( mflag == 1 )
  {
    init = 0;
    kop = 0;
    f ( *t, y, yp );
    nfe = 1;

    if ( *t == tout )
    {
      flag_return = 2;
      return flag_return;
    }

  }

  if ( init == 0 )
  {
    init = 1;
    h = fabs ( dt );
    toln = 0.0;

    for ( k = 0; k < neqn; k++ )
    {
      tol = (*relerr) * fabs ( y[k] ) + abserr;
      if ( 0.0 < tol )
      {
        toln = tol;
        ypk = fabs ( yp[k] );
        if ( tol < ypk * pow ( h, 5 ) )
        {
          h = pow ( ( tol / ypk ), 0.2 );
        }
      }
    }

    if ( toln <= 0.0 )
    {
      h = 0.0;
    }

    h = fmax ( h, 26.0 * eps * fmax ( fabs ( *t ), fabs ( dt ) ) );

    if ( flag_return < 0 )
    {
      flag_save = -2;
```

```
    }
    else
    {
      flag_save = 2;
    }
  }
//
//  Set stepsize for integration in the direction from T to TOUT.
//
  h = r8_sign ( dt ) * fabs ( h );
//
//  Test to see if too may output points are being requested.
//
  if ( 2.0 * fabs ( dt ) <= fabs ( h ) )
  {
    kop = kop + 1;
  }
//
//  Unnecessary frequency of output.
//
  if ( kop == 100 )
  {
    kop = 0;
    delete [] f1;
    delete [] f2;
    delete [] f3;
    delete [] f4;
    delete [] f5;
    flag_return = 7;
    return flag_return;
  }
//
//  If we are too close to the output point, then simply extrapolate and return.
//
  if ( fabs ( dt ) <= 26.0 * eps * fabs ( *t ) )
  {
    *t = tout;
    for ( i = 0; i < neqn; i++ )
    {
      y[i] = y[i] + dt * yp[i];
    }
    f ( *t, y, yp );
    nfe = nfe + 1;

    delete [] f1;
    delete [] f2;
    delete [] f3;
    delete [] f4;
    delete [] f5;
    flag_return = 2;
    return flag_return;
  }
//
//  Initialize the output point indicator.
//
  output = false;
```

```
//
//  To avoid premature underflow in the error tolerance function,
//  scale the error tolerances.
//
  scale = 2.0 / (*relerr);
  ae = scale * abserr;
//
//  Step by step integration.
//
  for ( ; ; )
  {
    hfaild = false;
//
//  Set the smallest allowable stepsize.
//
    hmin = 26.0 * eps * fabs ( *t );
//
//  Adjust the stepsize if necessary to hit the output point.
//
//  Look ahead two steps to avoid drastic changes in the stepsize and
//  thus lessen the impact of output points on the code.
//
    dt = tout - *t;

    if ( 2.0 * fabs ( h ) <= fabs ( dt ) )
    {
    }
    else
//
//  Will the next successful step complete the integration to the output point?
//
    {
      if ( fabs ( dt ) <= fabs ( h ) )
      {
        output = true;
        h = dt;
      }
      else
      {
        h = 0.5 * dt;
      }

    }
//
//  Here begins the core integrator for taking a single step.
//
//  The tolerances have been scaled to avoid premature underflow in
//  computing the error tolerance function ET.
//  To avoid problems with zero crossings, relative error is measured
//  using the average of the magnitudes of the solution at the
//  beginning and end of a step.
//  The error estimate formula has been grouped to control loss of
//  significance.
//
//  To distinguish the various arguments, H is not permitted
//  to become smaller than 26 units of roundoff in T.
```

```
//  Practical limits on the change in the stepsize are enforced to
//  smooth the stepsize selection process and to avoid excessive
//  chattering on problems having discontinuities.
//  To prevent unnecessary failures, the code uses 9/10 the stepsize
//  it estimates will succeed.
//
//  After a step failure, the stepsize is not allowed to increase for
//  the next attempted step.  This makes the code more efficient on
//  problems having discontinuities and more effective in general
//  since local extrapolation is being used and extra caution seems
//  warranted.
//
//  Test the number of derivative function evaluations.
//  If okay, try to advance the integration from T to T+H.
//
   for ( ; ; )
   {
//
//  Have we done too much work?
//
     if ( MAXNFE < nfe )
     {
       kflag = 4;
       delete [] f1;
       delete [] f2;
       delete [] f3;
       delete [] f4;
       delete [] f5;
       flag_return = 4;
       return flag_return;
     }
//
//  Advance an approximate solution over one step of length H.
//
     r8_fehl ( f, neqn, y, *t, h, yp, f1, f2, f3, f4, f5, f1 );
     nfe = nfe + 5;
//
//  Compute and test allowable tolerances versus local error estimates
//  and remove scaling of tolerances.  The relative error is
//  measured with respect to the average of the magnitudes of the
//  solution at the beginning and end of the step.
//
     eeoet = 0.0;

     for ( k = 0; k < neqn; k++ )
     {
       et = fabs ( y[k] ) + fabs ( f1[k] ) + ae;

       if ( et <= 0.0 )
       {
         delete [] f1;
         delete [] f2;
         delete [] f3;
         delete [] f4;
         delete [] f5;
         flag_return = 5;
```

```
        return flag_return;
      }

      ee = fabs
      ( ( -2090.0 * yp[k]
        + ( 21970.0 * f3[k] - 15048.0 * f4[k] )
        )
      + ( 22528.0 * f2[k] - 27360.0 * f5[k] )
      );

      eeoet = fmax ( eeoet, ee / et );

    }

    esttol = fabs ( h ) * eeoet * scale / 752400.0;

    if ( esttol <= 1.0 )
    {
      break;
    }
//
//  Unsuccessful step.  Reduce the stepsize, try again.
//  The decrease is limited to a factor of 1/10.
//
    hfaild = true;
    output = false;

    if ( esttol < 59049.0 )
    {
      s = 0.9 / pow ( esttol, 0.2 );
    }
    else
    {
      s = 0.1;
    }

    h = s * h;

    if ( fabs ( h ) < hmin )
    {
      kflag = 6;
      delete [] f1;
      delete [] f2;
      delete [] f3;
      delete [] f4;
      delete [] f5;
      flag_return = 6;
      return flag_return;
    }

  }
//
//  We exited the loop because we took a successful step.
//  Store the solution for T+H, and evaluate the derivative there.
//
  *t = *t + h;
```

```
    for ( i = 0; i < neqn; i++ )
    {
      y[i] = f1[i];
    }
    f ( *t, y, yp );
    nfe = nfe + 1;
//
//  Choose the next stepsize.  The increase is limited to a factor of 5.
//  If the step failed, the next stepsize is not allowed to increase.
//
    if ( 0.0001889568 < esttol )
    {
      s = 0.9 / pow ( esttol, 0.2 );
    }
    else
    {
      s = 5.0;
    }

    if ( hfaild )
    {
      s = fmin ( s, 1.0 );
    }

    h = r8_sign ( h ) * fmax ( s * fabs ( h ), hmin );
//
//  End of core integrator
//
//  Should we take another step?
//
    if ( output )
    {
      *t = tout;
      delete [] f1;
      delete [] f2;
      delete [] f3;
      delete [] f4;
      delete [] f5;
      flag_return = 2;
      return flag_return;
    }

    if ( flag_return <= 0 )
    {
      delete [] f1;
      delete [] f2;
      delete [] f3;
      delete [] f4;
      delete [] f5;
      flag_return = -2;
      return flag_return;
    }

  }
# undef MAXNFE
}
```

```
//****************************************************************************80

double r8_sign ( double x )

//****************************************************************************80
//
//  Purpose:
//
//    R8_SIGN returns the sign of an R8.
//
//  Licensing:
//
//    This code is distributed under the MIT license.
//
//  Modified:
//
//    27 March 2004
//
//  Author:
//
//    John Burkardt
//
//  Parameters:
//
//    Input, double X, the number whose sign is desired.
//
//    Output, double R8_SIGN, the sign of X.
//
{
  if ( x < 0.0 )
  {
    return ( -1.0 );
  }
  else
  {
    return ( +1.0 );
  }
}
//****************************************************************************80

void timestamp ( )

//****************************************************************************80
//
//  Purpose:
//
//    TIMESTAMP prints the current YMDHMS date as a time stamp.
//
//  Example:
//
//    May 31 2001 09:45:54 AM
//
//  Licensing:
//
//    This code is distributed under the MIT license.
//
```

```
//  Modified:
//
//    24 September 2003
//
//  Author:
//
//    John Burkardt
//
//  Parameters:
//
//    None
//
{
# define TIME_SIZE 40

  static char time_buffer[TIME_SIZE];
  const struct tm *tm;
  time_t now;

  now = time ( NULL );
  tm = localtime ( &now );

  strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm );

  cout << time_buffer << "\n";

  return;
# undef TIME_SIZE
}
```