```
   Help
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <time.h>

# include "rk45.h"

/******************************************************************
    ********************/

double r4_abs ( double x )

/******************************************************************
    ********************/
/*
  Purpose:

    R4_ABS returns the absolute value of an R4.

  Licensing:

    This code is distributed under the GNU LGPL license.

  Modified:

    12 January 2007

  Author:

    John Burkardt

  Parameters:

    Input, double X, the quantity whose absolute value is
    desired.

    Output, double R4_ABS, the absolute value of X.
*/
{
  double value;
```

```
  if ( 0.0 <= x )
  {
    value = x;
  }
  else
  {
    value = - x;
  }
  return value;
}
/************************************************************
    ********************/

double r4_epsilon ( void )

/************************************************************
    ********************/
/*
  Purpose:

    R4_EPSILON returns the R4 roundoff unit.

  Discussion:

    The roundoff unit is a number R which is a power of 2
    with the
    property that, to the precision of the computer's arith
    metic,
      1 < 1 + R
    but
      1 = ( 1 + R / 2 )

  Licensing:

    This code is distributed under the GNU LGPL license.

  Modified:

    01 July 2004
```

```
  Author:

    John Burkardt

  Parameters:

    Output, double R4_EPSILON, the R4 round-off unit.
*/
{
  double value;

  value = 1.0;

  while ( 1.0 < ( double ) ( 1.0 + value )  )
  {
    value = value / 2.0;
  }

  value = 2.0 * value;

  return value;
}
/**********************************************************
    ********************/

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

/**********************************************************
    ********************/
/*
  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 or
  dinary 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 sign
  ificance.
  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 GNU LGPL license.

Modified:

  27 March 2004

Author:

  Original FORTRAN77 version by Herman Watts, Lawrence Sh
  ampine.
  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 - Th
    e State of the Art,
    SIAM Review,
    Volume 18, pages 376-411, 1976.

  Parameters:

    Input, external F, a user-supplied subroutine to evalu
    ate the
    derivatives Y'(T), of the form:

      void f ( double t, double y[], double yp[], void* pt
    )

    Input, int NEQN, the number of equations to be integrat
    ed.

    Input, double Y[NEQN], the current value of the depend
    ent 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 deriv
    ative 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, pt);

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, pt );

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, pt );

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, pt );

  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, pt );
/*
  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;
}
/**********************************************************
    ********************/

double r4_max ( double x, double y )

/**********************************************************
    ********************/
```

```
/*
  Purpose:

    R4_MAX returns the maximum of two R4's.

  Licensing:

    This code is distributed under the GNU LGPL license.

  Modified:

    07 May 2006

  Author:

    John Burkardt

  Parameters:

    Input, double X, Y, the quantities to compare.

    Output, double R4_MAX, the maximum of X and Y.
*/
{
  double value;

  if ( y < x )
  {
    value = x;
  }
  else
  {
    value = y;
  }
  return value;
}
/************************************************************
    ********************/

double r4_min ( double x, double y )
```

```
/**********************************************************
    ********************/
/*
  Purpose:

    R4_MIN returns the minimum of two R4's.

  Licensing:

    This code is distributed under the GNU LGPL license.

  Modified:

    07 May 2006

  Author:

    John Burkardt

  Parameters:

    Input, double X, Y, the quantities to compare.

    Output, double R4_MIN, the minimum of X and Y.
*/
{
  double value;

  if ( y < x )
  {
    value = y;
  }
  else
  {
    value = x;
  }
  return value;
}
/**********************************************************
    ********************/
```

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

/*****************************************************
    ********************/
/*
  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 deman
    ding
    high accuracy.

    This routine integrates a system of NEQN first-order or
    dinary 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 righ
t 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 po
int 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 fir
st 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 rou
tine 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 TOU
T.  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 th
e 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 (MAX
NFE/6) steps.
  The user may continue by simply calling again.  The
function counter
  will be reset to 0, and another MAXNFE function evalu
ations are allowed.

* 5, integration was not completed because the solutio
n vanished,
  making a pure relative error test impossible.  The u
ser 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 reques
  ted accuracy
    could not be achieved, even using the smallest allow
  able 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 oc
  currence 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 na
  tural stepsize
    choice.  The user should use the one-step integratio
  n mode with
    the stepsize determined by the code.  If the user ins
  ists upon
    continuing the integration, reset FLAG to 2 before
  calling
    again.  Otherwise, execution will be terminated.

  * 8, invalid input parameters, indicates one of the fo
  llowing:
    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 GNU LGPL license.
```

```
Modified:

  27 March 2004

Author:

  Original FORTRAN77 version by Herman Watts, Lawrence Sh
  ampine.
  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 - Th
  e State of the Art,
  SIAM Review,
  Volume 18, pages 376-411, 1976.

Parameters:

  Input, external F, a user-supplied subroutine to evalu
  ate the
  derivatives Y'(T), of the form:

    void f ( double t, double y[], double yp[], void* pt
  )

  Input, int NEQN, the number of equations to be integrat
  ed.

  Input/output, double Y[NEQN], the current solution vec
  tor 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 indep
    endent 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 absolu
    te error tolerances
    for the local error test.  At each step the code requir
    es:
      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 integratio
    n.  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;
  static int flag_save = -1000;
  static double h = -1.0;
  int 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;
  int 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;
/*
  Check the input parameters.
*/
  eps = r4_epsilon ( );

  if ( neqn < 1 )
  {
    return 8;
```

```
  }

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

  if ( abserr < 0.0 )
  {
    return 8;
  }

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

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

```
      else if ( kflag == 4 )
      {
        nfe = 0;
      }
      else if ( kflag == 5 && abserr == 0.0 )
      {
        exit ( 1 );
      }
      else if ( kflag == 6 && (*relerr) <= relerr_save &&
    abserr <= abserr_save )
      {
        exit ( 1 );
      }
    }
/*
  FLAG = 3, 4, 5, 6, 7 or 8.
*/
    else
    {
      if ( flag == 3 )
      {
        flag = flag_save;
        if ( kflag == 3 )
        {
          mflag = abs ( flag );
        }
      }
      else if ( flag == 4 )
      {
        nfe = 0;
        flag = flag_save;
        if ( kflag == 3 )
        {
          mflag = abs ( flag );
        }
      }
      else if ( flag == 5 && 0.0 < abserr )
      {
        flag = flag_save;
        if ( kflag == 3 )
        {
```

```
          mflag = abs ( flag );
        }
      }
/*
  Integration cannot be continued because the user did not
    respond to
  the instructions pertaining to FLAG = 5, 6, 7 or 8.
*/
      else
      {
        exit ( 1 );
      }
    }
  }
/*
  Save the input value of FLAG.
  Set the continuation flag KFLAG for subsequent input che
    cking.
*/
  flag_save = flag;
  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 * r4_epsilon ( ) + remin;
/*
  Is the relative error tolerance too small?
*/
  if ( (*relerr) < relerr_min )
```

```
  {
    (*relerr) = relerr_min;
    kflag = 3;
    return 3;
  }

  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 = ( double * ) malloc ( neqn * sizeof ( double ) );
  f2 = ( double * ) malloc ( neqn * sizeof ( double ) );
  f3 = ( double * ) malloc ( neqn * sizeof ( double ) );
  f4 = ( double * ) malloc ( neqn * sizeof ( double ) );
  f5 = ( double * ) malloc ( neqn * sizeof ( double ) );

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

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

  }

  if ( init == 0 )
  {
    init = 1;
    h = r4_abs ( dt );
    toln = 0.0;
```

```
    for ( k = 0; k < neqn; k++ )
    {
      tol = (*relerr) * r4_abs ( y[k] ) + abserr;
      if ( 0.0 < tol )
      {
        toln = tol;
        ypk = r4_abs ( yp[k] );
        if ( tol < ypk * pow ( h, 5 ) )
        {
          h = ( double ) pow ( ( double ) ( tol / ypk ), 0.
    2 );
        }
      }
    }

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

    h = r4_max ( h, 26.0 * eps * r4_max ( r4_abs ( *t ), r4
    _abs ( dt ) ) );

    if ( flag < 0 )
    {
      flag_save = -2;
    }
    else
    {
      flag_save = 2;
    }
  }
/*
  Set stepsize for integration in the direction from T to
    TOUT.
*/
  h = r4_sign ( dt ) * r4_abs ( h );
/*
  Test to see if too may output points are being requested.
*/
```

```
  if ( 2.0 * r4_abs ( dt ) <= r4_abs ( h ) )
  {
    kop = kop + 1;
  }
/*
  Unnecessary frequency of output.
*/
  if ( kop == 100 )
  {
    kop = 0;
    free ( f1 );
    free ( f2 );
    free ( f3 );
    free ( f4 );
    free ( f5 );
    return 7;
  }
/*
  If we are too close to the output point, then simply ext
    rapolate and return.
*/
  if ( r4_abs ( dt ) <= 26.0 * eps * r4_abs ( *t ) )
  {
    *t = tout;
    for ( i = 0; i < neqn; i++ )
    {
      y[i] = y[i] + dt * yp[i];
    }
    f ( *t, y, yp, pt );
    nfe = nfe + 1;

    free ( f1 );
    free ( f2 );
    free ( f3 );
    free ( f4 );
    free ( f5 );
    return 2;
  }
/*
  Initialize the output point indicator.
*/
```

```
  output = 0;
/*
  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 = 0;
/*
  Set the smallest allowable stepsize.
*/
    hmin = 26.0 * eps * r4_abs ( *t );
/*
  Adjust the stepsize if necessary to hit the output point.

  Look ahead two steps to avoid drastic changes in the step
    size and
  thus lessen the impact of output points on the code.
*/
    dt = tout - *t;

    if ( 2.0 * r4_abs ( h ) <= r4_abs ( dt ) )
    {
    }
    else
/*
  Will the next successful step complete the integration
    to the output point?
*/
    {
      if ( r4_abs ( dt ) <= r4_abs ( h ) )
      {
        output = 1;
        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 underf
    low 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 enfor
    ced to
  smooth the stepsize selection process and to avoid excess
    ive
  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 inc
    rease for
  the next attempted step.  This makes the code more effic
    ient on
  problems having discontinuities and more effective in general
  since local extrapolation is being used and extra cautio
    n 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;
        free ( f1 );
        free ( f2 );
        free ( f3 );
        free ( f4 );
        free ( f5 );
        return 4;
      }
/*
  Advance an approximate solution over one step of length
    H.
*/
      r4_fehl ( f, pt, 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 = r4_abs ( y[k] ) + r4_abs ( f1[k] ) + ae;

        if ( et <= 0.0 )
        {
          free ( f1 );
```

```
          free ( f2 );
          free ( f3 );
          free ( f4 );
          free ( f5 );
          return 5;
        }

        ee = r4_abs
        ( ( -2090.0 * yp[k]
          + ( 21970.0 * f3[k] - 15048.0 * f4[k] )
          )
        + ( 22528.0 * f2[k] - 27360.0 * f5[k] )
        );

        eeoet = r4_max ( eeoet, ee / et );

      }

      esttol = r4_abs ( 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 = 1;
      output = 0;

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

```
      h = s * h;

      if ( r4_abs ( h ) < hmin )
      {
        kflag = 6;
        free ( f1 );
        free ( f2 );
        free ( f3 );
        free ( f4 );
        free ( f5 );
        return 6;
      }

    }
/*
  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, pt );
    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 / ( double ) pow ( ( double ) esttol, 0.2 );
    }
    else
    {
      s = 5.0;
    }
```

```
    if ( hfaild )
    {
      s = r4_min ( s, 1.0 );
    }

    h = r4_sign ( h ) * r4_max ( s * r4_abs ( h ), hmin );
/*
  End of core integrator

  Should we take another step?
*/
    if ( output )
    {
      *t = tout;
      free ( f1 );
      free ( f2 );
      free ( f3 );
      free ( f4 );
      free ( f5 );
      return 2;
    }

    if ( flag <= 0 )
    {
      free ( f1 );
      free ( f2 );
      free ( f3 );
      free ( f4 );
      free ( f5 );
      return (-2);
    }

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

double r4_sign ( double x )

/******************************************************************
```

```
    ********************/
/*
  Purpose:

    R4_SIGN returns the sign of an R4.

  Licensing:

    This code is distributed under the GNU LGPL license.

  Modified:

    08 May 2006

  Author:

    John Burkardt

  Parameters:

    Input, double X, the number whose sign is desired.

    Output, double R4_SIGN, the sign of X.
*/
{
  double value;

  if ( x < 0.0 )
  {
    value = -1.0;
  }
  else
  {
    value = 1.0;
  }
  return value;
}
/*************************************************************
    ********************/

double r8_abs ( double x )
```

```
/**************************************************************
    ********************/
/*
  Purpose:

    R8_ABS returns the absolute value of an R8.

  Licensing:

    This code is distributed under the GNU LGPL license.

  Modified:

    07 May 2006

  Author:

    John Burkardt

  Parameters:

    Input, double X, the quantity whose absolute value is
    desired.

    Output, double R8_ABS, the absolute value of X.
*/
{
  double value;

  if ( 0.0 <= x )
  {
    value = x;
  }
  else
  {
    value = - x;
  }
  return value;
}
/**************************************************************
```

```
    ********************/

double r8_epsilon ( void )

/**************************************************************
    ********************/
/*
  Purpose:

    R8_EPSILON returns the R8 round off unit.

  Discussion:

    R8_EPSILON is a number R which is a power of 2 with th
    e property that,
    to the precision of the computer's arithmetic,
      1 < 1 + R
    but
      1 = ( 1 + R / 2 )

  Licensing:

    This code is distributed under the GNU LGPL license.

  Modified:

    08 May 2006

  Author:

    John Burkardt

  Parameters:

    Output, double R8_EPSILON, the R8 round-off unit.
*/
{
  double r;

  r = 1.0;
```

```
  while ( 1.0 < ( double ) ( 1.0 + r )  )
  {
    r = r / 2.0;
  }
  r = 2.0 * r;

  return r;
}
/*****************************************************************
    ********************/

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

/*****************************************************************
    ********************/
/*
  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 or
    dinary 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 sign
  ificance.
  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 GNU LGPL license.

Modified:

  27 March 2004

Author:

  Original FORTRAN77 version by Herman Watts, Lawrence Sh
  ampine.
  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 - Th
  e State of the Art,
  SIAM Review,
  Volume 18, pages 376-411, 1976.

Parameters:

  Input, external F, a user-supplied subroutine to evalu
  ate the
  derivatives Y'(T), of the form:
```

```
      void f ( double t, double y[], double yp[], void* pt
    )

    Input, int NEQN, the number of equations to be integrat
    ed.

    Input, double Y[NEQN], the current value of the depend
    ent 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 deriv
    ative 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, pt );

  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, pt );

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, pt );

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, pt );

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, pt );
/*
  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;
}
/**************************************************************
    ********************/

double r8_max ( double x, double y )

/**************************************************************
    ********************/
/*
  Purpose:

    R8_MAX returns the maximum of two R8's.

  Licensing:

    This code is distributed under the GNU LGPL license.

  Modified:

    07 May 2006
```

```
  Author:

    John Burkardt

  Parameters:

    Input, double X, Y, the quantities to compare.

    Output, double R8_MAX, the maximum of X and Y.
*/
{
  double value;

  if ( y < x )
  {
    value = x;
  }
  else
  {
    value = y;
  }
  return value;
}
/**************************************************************
    ********************/

double r8_min ( double x, double y )

/**************************************************************
    ********************/
/*
  Purpose:

    R8_MIN returns the minimum of two R8's.

  Licensing:

    This code is distributed under the GNU LGPL license.

  Modified:
```

```
    07 May 2006

  Author:

    John Burkardt

  Parameters:

    Input, double X, Y, the quantities to compare.

    Output, double R8_MIN, the minimum of X and Y.
*/
{
  double value;

  if ( y < x )
  {
    value = y;
  }
  else
  {
    value = x;
  }
  return value;
}
/**************************************************************
    ********************/

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

/**************************************************************
    ********************/
/*
  Purpose:

    R8_RKF45 carries out the Runge-Kutta-Fehlberg method.
```

```
Discussion:

  This version of the routine uses DOUBLE real arithmeti
  c.

  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 deman
  ding
  high accuracy.

  This routine integrates a system of NEQN first-order or
  dinary 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 righ
  t 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 po
```

```
int 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 fir
st 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 rou
tine 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 TOU
T.  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 th
e 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 (MAX
NFE/6) steps.
  The user may continue by simply calling again.  The
function counter
  will be reset to 0, and another MAXNFE function evalu
ations are allowed.

* 5, integration was not completed because the solutio
n vanished,
  making a pure relative error test impossible.  The u
ser 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 reques
ted accuracy
  could not be achieved, even using the smallest allow
able 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 oc
currence 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 na
  tural stepsize
    choice.  The user should use the one-step integratio
  n mode with
    the stepsize determined by the code.  If the user ins
  ists upon
    continuing the integration, reset FLAG to 2 before
  calling
    again.  Otherwise, execution will be terminated.

  * 8, invalid input parameters, indicates one of the fo
  llowing:
    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 GNU LGPL license.

Modified:

  27 March 2004

Author:

  Original FORTRAN77 version by Herman Watts, Lawrence Sh
  ampine.
  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 - Th
  e State of the Art,
  SIAM Review,
  Volume 18, pages 376-411, 1976.

Parameters:

  Input, external F, a user-supplied subroutine to evalu
  ate the
  derivatives Y'(T), of the form:

    void f ( double t, double y[], double yp[], void* pt
  )

  Input, int NEQN, the number of equations to be integrat
  ed.

  Input/output, double Y[NEQN], the current solution vec
  tor 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 indep
  endent 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 absolu
```

```
    te error tolerances
    for the local error test.  At each step the code requir
    es:
      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 integratio
    n.  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;
  static int flag_save = -1000;
  static double h = -1.0;
  int 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;
  int 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;
/*
  Check the input parameters.
*/
  eps = r8_epsilon ( );

  if ( neqn < 1 )
  {
    return 8;
  }

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

  if ( abserr < 0.0 )
  {
    return 8;
  }
```

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

  mflag = abs ( flag );
/*
  Is this a continuation call?
*/
  if ( mflag != 1 )
  {
    if ( *t == tout && kflag != 3 )
    {
      return 8;
    }
/*
  FLAG = -2 or +2:
*/
    if ( mflag == 2 )
    {
      if ( kflag == 3 )
      {
        flag = flag_save;
        mflag = abs ( flag );
      }
      else if ( init == 0 )
      {
        flag = flag_save;
      }
      else if ( kflag == 4 )
      {
        nfe = 0;
      }
      else if ( kflag == 5 && abserr == 0.0 )
      {
        exit ( 1 );
      }
      else if ( kflag == 6 && (*relerr) <= relerr_save &&
    abserr <= abserr_save )
      {
        exit ( 1 );
```

```
      }
    }
/*
  FLAG = 3, 4, 5, 6, 7 or 8.
*/
    else
    {
      if ( flag == 3 )
      {
        flag = flag_save;
        if ( kflag == 3 )
        {
          mflag = abs ( flag );
        }
      }
      else if ( flag == 4 )
      {
        nfe = 0;
        flag = flag_save;
        if ( kflag == 3 )
        {
          mflag = abs ( flag );
        }
      }
      else if ( flag == 5 && 0.0 < abserr )
      {
        flag = flag_save;
        if ( kflag == 3 )
        {
          mflag = abs ( flag );
        }
      }
/*
  Integration cannot be continued because the user did not
    respond to
  the instructions pertaining to FLAG = 5, 6, 7 or 8.
*/
      else
      {
        exit ( 1 );
      }
```

```
    }
  }
/*
  Save the input value of FLAG.
  Set the continuation flag KFLAG for subsequent input che
    cking.
*/
  flag_save = flag;
  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 * r8_epsilon ( ) + remin;
/*
  Is the relative error tolerance too small?
*/
  if ( (*relerr) < relerr_min )
  {
    (*relerr) = relerr_min;
    kflag = 3;
    return 3;
  }

  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 = ( double * ) malloc ( neqn * sizeof ( double ) );
  f2 = ( double * ) malloc ( neqn * sizeof ( double ) );
  f3 = ( double * ) malloc ( neqn * sizeof ( double ) );
  f4 = ( double * ) malloc ( neqn * sizeof ( double ) );
  f5 = ( double * ) malloc ( neqn * sizeof ( double ) );

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

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

  }

  if ( init == 0 )
  {
    init = 1;
    h = r8_abs ( dt );
    toln = 0.0;

    for ( k = 0; k < neqn; k++ )
    {
      tol = (*relerr) * r8_abs ( y[k] ) + abserr;
      if ( 0.0 < tol )
      {
        toln = tol;
        ypk = r8_abs ( yp[k] );
        if ( tol < ypk * pow ( h, 5 ) )
        {
          h = pow ( ( tol / ypk ), 0.2 );
        }
```

```
      }
    }

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

    h = r8_max ( h, 26.0 * eps * r8_max ( r8_abs ( *t ), r8
    _abs ( dt ) ) );

    if ( flag < 0 )
    {
      flag_save = -2;
    }
    else
    {
      flag_save = 2;
    }
  }
/*
  Set stepsize for integration in the direction from T to
    TOUT.
*/
  h = r8_sign ( dt ) * r8_abs ( h );
/*
  Test to see if too may output points are being requested.
*/
  if ( 2.0 * r8_abs ( dt ) <= r8_abs ( h ) )
  {
    kop = kop + 1;
  }
/*
  Unnecessary frequency of output.
*/
  if ( kop == 100 )
  {
    kop = 0;
    free ( f1 );
    free ( f2 );
    free ( f3 );
```

```
    free ( f4 );
    free ( f5 );
    return 7;
  }
/*
  If we are too close to the output point, then simply ext
    rapolate and return.
*/
  if ( r8_abs ( dt ) <= 26.0 * eps * r8_abs ( *t ) )
  {
    *t = tout;
    for ( i = 0; i < neqn; i++ )
    {
      y[i] = y[i] + dt * yp[i];
    }
    f ( *t, y, yp, pt );
    nfe = nfe + 1;

    free ( f1 );
    free ( f2 );
    free ( f3 );
    free ( f4 );
    free ( f5 );
    return 2;
  }
/*
  Initialize the output point indicator.
*/
  output = 0;
/*
  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 = 0;
/*
  Set the smallest allowable stepsize.
*/
    hmin = 26.0 * eps * r8_abs ( *t );
/*
  Adjust the stepsize if necessary to hit the output point.

  Look ahead two steps to avoid drastic changes in the step
    size and
  thus lessen the impact of output points on the code.
*/
    dt = tout - *t;

    if ( 2.0 * r8_abs ( h ) <= r8_abs ( dt ) )
    {
    }
    else
/*
  Will the next successful step complete the integration
    to the output point?
*/
    {
      if ( r8_abs ( dt ) <= r8_abs ( h ) )
      {
        output = 1;
        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 underf
    low 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 enfor
    ced to
  smooth the stepsize selection process and to avoid excess
    ive
  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 inc
    rease for
  the next attempted step.  This makes the code more effic
    ient on
  problems having discontinuities and more effective in general
  since local extrapolation is being used and extra cautio
    n 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;
        free ( f1 );
        free ( f2 );
        free ( f3 );
```

```
        free ( f4 );
        free ( f5 );
        return 4;
      }
/*
  Advance an approximate solution over one step of length
    H.
*/
      r8_fehl ( f, pt, 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 = r8_abs ( y[k] ) + r8_abs ( f1[k] ) + ae;

        if ( et <= 0.0 )
        {
          free ( f1 );
          free ( f2 );
          free ( f3 );
          free ( f4 );
          free ( f5 );
          return 5;
        }

        ee = r8_abs
        ( ( -2090.0 * yp[k]
          + ( 21970.0 * f3[k] - 15048.0 * f4[k] )
          )
        + ( 22528.0 * f2[k] - 27360.0 * f5[k] )
        );
```

```
        eeoet = r8_max ( eeoet, ee / et );

      }

      esttol = r8_abs ( 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 = 1;
      output = 0;

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

      h = s * h;

      if ( r8_abs ( h ) < hmin )
      {
        kflag = 6;
        free ( f1 );
        free ( f2 );
        free ( f3 );
        free ( f4 );
        free ( f5 );
        return 6;
      }

    }
```

```
/*
  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, pt );
    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 = r8_min ( s, 1.0 );
    }

    h = r8_sign ( h ) * r8_max ( s * r8_abs ( h ), hmin );
/*
  End of core integrator

  Should we take another step?
*/
    if ( output )
    {
      *t = tout;
```

```
      free ( f1 );
      free ( f2 );
      free ( f3 );
      free ( f4 );
      free ( f5 );
      return 2;
    }

    if ( flag <= 0 )
    {
      free ( f1 );
      free ( f2 );
      free ( f3 );
      free ( f4 );
      free ( f5 );
      return (-2);
    }

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

double r8_sign ( double x )

/**********************************************************
    ********************/
/*
  Purpose:

    R8_SIGN returns the sign of an R8.

  Licensing:

    This code is distributed under the GNU LGPL license.

  Modified:

    08 May 2006
```

```
  Author:

    John Burkardt

  Parameters:

    Input, double X, the number whose sign is desired.

    Output, double R8_SIGN, the sign of X.
*/
{
  double value;

  if ( x < 0.0 )
  {
    value = -1.0;
  }
  else
  {
    value = 1.0;
  }
  return value;
}
/**************************************************************
    ********************/

void timestamp ( void )

/**************************************************************
    ********************/
/*
  Purpose:

    TIMESTAMP prints the current YMDHMS date as a time stam
    p.

  Example:

    31 May 2001 09:45:54 AM

  Licensing:
```

```
    This code is distributed under the GNU LGPL license.

  Modified:

    24 September 2003

  Author:

    John Burkardt

  Parameters:

    None
*/
{
# define TIME_SIZE 40

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

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

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

  printf ( "%s{n", time_buffer );

  return;
# undef TIME_SIZE
}
```

References