Help

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "tool_box.h"
#include "pnl/pnl_mathtools.h"



X_Grid * x_grid_create(  double dBnd_Left,double dBnd_Righ
    t,int dN)
{
  X_Grid * grid = malloc(sizeof(X_Grid));
  grid->Bnd_Left=dBnd_Left;
  grid->Bnd_Right=dBnd_Right;
  grid->N=dN;
  grid->step=(dBnd_Right-dBnd_Left)/(dN-1);
  return grid;
}

double x_grid_point(X_Grid *grid ,int i)
{
  return grid->Bnd_Left+i*grid->step;
}

void quadratic_interpolation(double Fm1, double F0,double
    Fp1,double Xm1,double X0,double Xp1,double X,double * FX,
    double * dFX)
{

  //quadratic interpolation
  double A = Fm1;
  double B = (F0-Fm1)/(X0-Xm1);
  double C = (Fp1-A-B*(Xp1-Xm1))/((Xp1-Xm1)*(Xp1-X0));
  (*FX) = A+B*(X-Xm1)+C*(X-Xm1)*(X-X0);
  (*dFX)= B + C*(2*X-Xm1-X0);
  //printf(">>> %7.4f  -  %7.4f  -  %7.4f   ->>  %7.4f   -
     %7.4f  %7.4f {n",X0,Xp1,X,F0,Fp1,*FX);
}
```

```
// All these functions ara not yet tested
void polint(double* xa,double* ya, int n, double x, double
    *y, double *dy);
double trapzd( PnlFunc *func,void * params, double a,
    double b, int n,double old_s);
double qromb(PnlFunc *func,void * params, double a, double
    b);


void polint(double* xa,double* ya, int n, double x, double
    *y, double *dy)
//Given arrays xa[0..n-1] and ya[0..n-1], and given a value
     x, this routine returns a value y, and
//an error estimate dy. If P(x) is the polynomial of degre
    e N ? 1 such that P(xai) = yai, i =
//1, . . . , n, then the returned value y = P(x).
{
  int i,m,ns=0;
  double den,dif,dift,ho,hp,w;
  double* c = malloc(n*sizeof(double));
  double* d = malloc(n*sizeof(double));
  dif=fabs(x-xa[0]);
  for (i=0;i<n;i++)
    { //Here we ?nd the index ns of the closest table entry
    ,
      if ( (dift=fabs(x-xa[i])) < dif) {
        ns=i;
        dif=dift;
      }
      c[i]=ya[i];
      //and initialize the tableau of c's and d's.
      d[i]=ya[i];
    }
  *y=ya[ns];
  //This is the initial approximation to y.
  for (m=1;m<n;m++)
    { //For each column of the tableau,
      for (i=0;i<n-m;i++)
        {
```

```
          //we loop over the current c's and d's and upda
    te them.
          ho=xa[i]-x;
          hp=xa[i+m]-x;
          w=c[i+1]-d[i];
          if ( (den=ho-hp) == 0.0) PNL_ERROR("Error in rout
    ine polint","tool_box/polint");
          //This error can occur only if two input xa's ar
    e (to within roundoff) identical.
          den=w/den;
          d[i]=hp*den; //Here the c's and d's are updated.
          c[i]=ho*den;
        }
      *y += (*dy=(2*ns < (n-m) ? c[ns] : d[ns]));
      //After each column in the tableau is completed, we
    decide which correction, c or d,
      //we want to add to our accumulating value of y, i.e.
    , which path to take through the
      //tableau-forking up or down. We do this in such a wa
    y as to take the most "straight
      //line" route through the tableau to its apex, upda
    ting ns accordingly to keep track of
      //where we are. This route keeps the partial approx
    imations centered (insofar as possible)
      //on the target x. The last dy added is thus the
    error indication.
    }
  c++;
  d++;
  free(c);
  free(d);
}

double trapzd( PnlFunc *func,void * params, double a,
    double b, int n,double old_s)
//This routine computes the nth stage of refinement of an
    extended trapezoidal rule.
//func is input as a pointer to the function to be integrat
    ed between limits a and b, also input.
//When called with n=1, the routine returns the crudest es
    timate of int_a^b f(x)dx.
```

```
//Subsequent calls with n=2,3,...
//(in that sequential order) will improve the accuracy by
    adding 2n-2 additional interior points.
{
  double x,tnm,sum,del;
  int it,j;
  if (n == 1)
    {
      return (0.5*(b-a)*(func->function(a,params)+func->
    function(b,params)));
    }
  else
    {
      for (it=1,j=1;j<n-1;j++)
        it <<= 1;
      //it=2^n-2
      tnm=it;
      del=(b-a)/tnm;
      //This is the spacing of the points to be added.
      x=a+0.5*del;
      for (sum=0.0,j=1;j<=it;j++,x+=del)
        sum += func->function(x,params);
      //This replaces s by its refined value.
      return 0.5*(old_s+(b-a)*sum/tnm);
    }
}

double qromb(PnlFunc *func,void * params, double a, double
    b)
//Returns the integral of the function func from a to b.
    Integration is performed by Romberg's
//method of order 2K, where, e.g., K=2 is Simpson's rule.
{
  int j;
  double ss,dss;
  PnlVect *s,*h;
  const double EPS = 1.0e-8;
  const int JMAX1 = 50;
  const int JMAXP1 = JMAX1+1;
  const int K = 5;
```

```
  //Here EPS is the fractional accuracy desired, as determi
    ned by the extrapolation error estimate;
  //JMAX limits the total number of steps; K is the number
    of points used in the extrapolation.

  s=pnl_vect_create(JMAXP1);
  h=s=pnl_vect_create(JMAXP1+1);
  //These store the successive trapezoidal approximations
  //and their relative stepsizes.
  LET(h,0)=1.0;
  for (j=0;j<JMAX1;j++)
    {
      LET(s,j)=trapzd(func,params,a,b,j,(j>0)?GET(s,j-1):0.
    0);
      if (j >= K)
        {
          polint(&(LET(h,j-K)),&(LET(s,j-K)),K,0.0,&ss,&ds
    s);
          if (fabs(dss) <= EPS*fabs(ss))
            {
              pnl_vect_free(&h);
              pnl_vect_free(&s);
              return ss;
            }
        }
      LET(h,j+1)=0.25*GET(h,j);
    }
  pnl_vect_free(&h);
  pnl_vect_free(&s);
  PNL_ERROR("Too many steps in routine qromb"," ");
  return 0.0;   //Never get here.
}



/*
  const double Point_legendre_5[]={4.691007703066802e-02,2.
    307653449471585e-01,0.5,7.692346550528415e-01,9.5308992296
    93319e-01};
  const double Weight_legendre_5[]={1.184634425280945e-01,2
    .393143352496832e-01,2.844444444444444e-01,2.3931433524968
```

```
    32e-01,1.184634425280945e-01};

*/



  /* point15, abscissae 15-point rule */
static const double point15[15] = {
  -0.991455371120813, -0.949107912342759, -0.86486442335976
    9, -0.741531185599394,-0.586087235467691, -0.4058451513773
    97, -0.207784955007898, 0,
  0.207784955007898, 0.405845151377397, 0.586087235467691,
    0.741531185599394,0.864864423359769, 0.949107912342759, 0.
    991455371120813};
/* weight15, weights of the 15-points formula for abscissae
     point15 */
static const double weight15[15] = {
  0.022935322010529, 0.063092092629979, 0.104790010322250,
    0.140653259715525,0.169004726639267, 0.190350578064785, 0.
    204432940075298, 0.209482141084728,
  0.204432940075298, 0.190350578064785, 0.169004726639267,
    0.140653259715525, 0.104790010322250, 0.063092092629979, 0
    .022935322010529
};

/* weightgauss7, weights of the 7-points of gauss formula *
    /
static const double weightgauss[7] = {
  0.129484966168870, 0.279705391489277, 0.381830050505119,
    0.417959183673469,
  0.381830050505119, 0.279705391489277, 0.129484966168870
};
/*
function Q = quadvgk(fv, Subs, NF, AbsTol)
  //QUADVGK: Vectorised adaptive G7,K15 Gaussian quadratu
    re on vector of integrals
  //
  //       This function calculates the integration of a
    vector of functions via adaptive G7,K15
  //       Gaussian quadrature. The procedure follows that
    of Shampine [1]. This function is
```

```
//       similar to quadv, but uses the quadgk algorithm.
//
//Usage:
//       Q = quadvgk(fv, Subs, NF)
//
//       Q               = Returned vector of numerical
  approximations to the integrals
//       fv              = Function handle which returns
  a matrix of values (see below)
//       Subs            = Matrix of intervals (see below
  )
//       NF              = Number of functions to be calc
  ulated (see below)
//
//Example:
//       Y = @(x,n) 1./((1:n)+x);
//       Qv = quadvgk(@(x) Y(x,10), [0;1], 10);
//
//       Y is a vector of functions, where each row rep
  resents a different function and each
//       column corresponds to a different point where th
  e function is evaluated. The function
//       needs to return a matrix of size (NF, NX) where
  NF is the number of functional
//       integrals to evaluate, and NX is the number of
  data points to evaluate. In the above
//       example, the quadvgk calculates:
//       [int(1./(1+x), 0..1); int(1./(2+x), 0..1); int(1
  ./(3+x), 0..1); ...; int(1./(10+x), 0..1)]
//
//       Subs is the initial set of subintervals to evalu
  ate the integrand over. Should be in
//       the form [a1 a2 a3 ... aN; b1 b2 b3 ... bN], wh
  ere "an" and "bn" are the limits of
//       each subinterval. In the above example, Subs = [
  a; b]. If the function contains sharp
//       features at known positions, the boundaries of
  these features should be added to Subs.
//       Note that in order to integrate over a continuo
  us span of subintervals, Subs = [A a1
//       a2 a3 ... aN; a1 a2 a3 ... aN B] where "A" and "
```

```
    B" are the limits of the whole
  //       integral, and a1..aN are points within this rang
    e.
  //
  //   Based on "quadva" by Lawrence F. Shampine.
  //   Ref: L.F. Shampine, "Vectorized Adaptive Quadrature
    in Matlab",
  //   Journal of Computational and Applied Mathematics,
    to appear.
{
  if nargin < 4
  AbsTol = 1e-6;

  NK = length(weight7);
  G = (2:2:NK);
  // 7-point Gaussian poisitions (subset of the Kronrod po
    ints)

  Q = zeros(NF, 1);
  A = Subs(1);
  B = Subs(2);
  path_len = B - A;
  while (~isempty(Subs))
    {
      GetSubs;
      M = (Subs(2,:)-Subs(1,:))/2;
      C = (Subs(2,:)+Subs(1,:))/2;
      NM = length(M);
      //x = reshape(point15*M + ones(NK,1)*C, 1, []);
      x = reshape(point15*M + repmat(C,NK,1), 1, []);
      FV = fv(x);
      Q1 = zeros(NF, NM);
      Q2 = zeros(NF, NM);
      for (n=1; n<=NF;n++)
        {
          F = reshape(FV(n,:), NK, []);
          Q1(n,:) = M.*sum((repmat(weight7,1,NM)).*F);
          Q2(n,:) = M.*sum((repmat(WG,1,NM)).*F(G,:));
          // Q1(n,:) = M.*sum((repmat(weight7*ones(1,NM)).*
    F);
          //  Q2(n,:) = M.*sum((WG*ones(1,NM)).*F(G,:));
```

```
        }
      ind = find((max(abs((Q1-Q2)), [], 1)<= 2*AbsTol*M./
    path_len )|((Subs(2,:)-Subs(1,:))<=eps));
      //ind = find((max(abs((Q1-Q2)./Q1), [], 1)<=1e-6)|((
    Subs(2,:)-Subs(1,:))<=eps));
      Q = Q + sum(Q1(:, ind), 2);
      Subs(:, ind) = [];
    }
  clear Q1 Q2 M C x FV;
  void  GetSubs()
  {
    M = (Subs(2,:)-Subs(1,:))/2;
    C = (Subs(2,:)+Subs(1,:))/2;
    I = point15*M + ones(NK,1)*C;
    A = [Subs(1,:); I];
    B = [I; Subs(2,:)];
    Subs = [reshape(A, 1, []); reshape(B, 1, [])];
    clear I A B;
  }
}
  */
```

References