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_Right, 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, doub
{

  //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
}



// 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
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 retu
//an error estimate dy. If P(x) is the polynomial of degree N ? 1 such that P(xa
//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 update 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 routine polint", "tool
          //This error can occur only if two input xa's are (to within roundoff)
          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 correctio
      //we want to add to our accumulating value of y, i.e., which path to take
      //tableau-forking up or down. We do this in such a way as to take the most
      //line" route through the tableau to its apex, updating ns accordingly to
      //where we are. This route keeps the partial approximations centered (inso
      //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
//This routine computes the nth stage of refinement of an extended trapezoidal r
//func is input as a pointer to the function to be integrated between limits a a
//When called with n=1, the routine returns the crudest estimate of int_a^b f(x)
//Subsequent calls with n=2,3,...
//(in that sequential order) will improve the accuracy by adding 2n-2 additional
{
  double x, tnm, sum, del;
  int it, j;
  if (n == 1)
    {
      return (0.5 * (b - a) * (func->F(a, params) + func->F(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->F(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 performe
//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 determined by the extrapolat
  //JMAX limits the total number of steps; K is the number of points used in the

  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, &dss);
          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
  const double Weight_legendre_5[]={1.184634425280945e-01,2.393143352496832e-01,

*/
```