Help

```
extern "C" {
#include  "
href../../../mod/mer1d/mer1d_std/mer1d_std_h_src.pdfmer1d_std.h"
#include "
href../../../common/enums_h_src.pdfenums.h"
#include <pnl/pnl_finance.h>
#include <pnl/pnl_random.h>
}

extern "C" {
#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (20017+2) //The "#el
  static int CHK_OPT(FD_iFGT)(void *Opt, void *Mod)
  {
    return NONACTIVE;
  }
  int CALC(FD_iFGT)(void *Opt, void *Mod, PricingMethod *Met)
  {
    return AVAILABLE_IN_FULL_PREMIA;
  }
#else

////////////////////////////////////////////////////////////////////////////
// Helper functions are taken with a slight modification from the file figtree.c
// belonging to the FIGTree library (a library for fast computation of Gauss tra
// using the Improved Fast Gauss Transform and Approximate Nearest Neighbor sear
// The full version of the FIGTree library  can be found at http://www.umiacs.um
////////////////////////////////////////////////////////////////////////////

// k-center clustering
// INPUT
// ----------------
//
// Dim               --> dimension of the points.
// NSources          --> number of sources.
// pSources          --> pointer to sources, (d*N).
// NumClusters       --> number of clusters.
//
// OUTPUT
// ----------------
```

```
//
// MaxClusterRadius  --> maximum radius of the clusters, (rx).
// pClusterIndex     --> vector of length N where the i th element is the
//                       cluster number to which the i th point belongs.
//                       pClusterIndex[i] varies between 0 to K-1.
// pClusterCenters   --> pointer to the cluster centers, (d*K).
// pNumPoints        --> pointer to the number of points in each cluster, (K).
// pClusterRadii     --> pointer to the radius of each cluster, (K).
//---------------------------------------------------------------------------

#ifndef K_CENTER_CLUSTERING_H
#define K_CENTER_CLUSTERING_H

class KCenterClustering{
  public:

    //Output parameters

    double MaxClusterRadius;  //maximum cluster radius

    //Functions

    //constructor
    KCenterClustering(int Dim,
      int NSources,
      double *pSources,
      int *pClusterIndex,
      int NumClusters
    );

    //destructor
    ~KCenterClustering();

    //K-center clustering
    //Returns the number of actual clusters (it might have stopped early if all
    //    radius of 0 -- which means that the number of clusters has reached the
    //    of unique pts)
    int Cluster();

    //Incremental k-center clustering
    //  nClusters - if non-NULL, value is set to the # of clusters at end of cal
```

```
    //  maxRadius - if non-NULL, value is set to the max radius of all clusters
    void ClusterIncrement( int * nClusters, double * maxRadius );

    //Compute cluster centers and the number of points in each cluster
    //and the radius of each cluster.
    void ComputeClusterCenters( int NumClusters,
                                double *pClusterCenters,
                                int *pNumPoints,
                                double *pClusterRadii
                              );

  private:
    //Input Parameters

    int d;           // dimension of the points.
    int N;           // number of sources.
    double *px;      // pointer to sources, (d*N).
    int K;           // max number of clusters
    int *pci;        // pointer to a vector of length N where the i th element i
                     // cluster number to which the i th point belongs.
    double *dist_C;  // distances to the center.
    double *r;

    int *pCenters;   // indices of the centers.
    int *cprev;      // index to the previous node
    int *cnext;      // index to the next node
    int *far2c;      // farthest node to the center

    int numClusters; // added by Vlad to keep track of # of clusters

    //Functions
    double ddist(const int d, const double *x, const double *y);
    int idmax(int n, double *x);

};


#endif
```

```
// k-center clustering
//---------------------------------------------------------------------
// Constructor
//
// PURPOSE
// -------
// Initialize the class.
// Read the parameters.
//
// INPUT
// ----------
// Dim         --> dimension of the points.
// NSources       --> number of sources.
// pSources       --> pointer to sources, (d*N).
// pClusterIndex   --> pointer to a vector of length N where the
//                      i th element is the cluster number to
//             which the i th point belongs.
//
//---------------------------------------------------------------------

KCenterClustering::KCenterClustering(int Dim,
      int NSources,
      double *pSources,
      int *pClusterIndex,
      int NumClusters
      )
{

  //Read the parameters

  d=Dim;
  N=NSources;
  px=pSources;
  pci=pClusterIndex;
  K=NumClusters;
  dist_C = new double[N]; //distances to the center.
  r=new double[K];

  // moved from Cluster() by Vlad 1/24/07
  pCenters = new int[K]; //indices of the centers.
```

```
  cprev    = new int[N]; // index to the previous node
  cnext    = new int[N]; // index to the next node
  far2c    = new int[K]; // farthest node to the center

  numClusters = 0;

  // KCenterClustering has errors if these are not all zeros to begin with.
  memset( pci, 0, sizeof(int)*N );
}

//------------------------------------------------------------------
// Destructor
//------------------------------------------------------------------

KCenterClustering::~KCenterClustering()
{
  delete [] dist_C;
  delete [] r;

  // moved from Cluster() by Vlad 1/24/07
  delete []cprev;
  delete []cnext;
  delete []far2c;
  delete []pCenters;
}



//------------------------------------------------------------------
// ddist is the square of the distance of two vectors(double)
//------------------------------------------------------------------


double
KCenterClustering::ddist(const int d, const double *x, const double *y)
{
  double t, s = 0.0;
  for (int i = d; i != 0; i--)
  {
    t = *x++ - *y++;
    s += t * t;
```

```
  }
  return s;
}



//---------------------------------------------------------------------
// Find the largest element from a vector
//---------------------------------------------------------------------

int
KCenterClustering::idmax(int n, double *x)
{
  int k = 0;
  double t = -1.0;
  for (int i = 0; i < n; i++, x++)
    if( t < *x )
    {
      t = *x;
      k = i;
    }
  return k;

}



//---------------------------------------------------------------------
// k-center Clustering.
//---------------------------------------------------------------------
//
// Gonzalez's farthest-point clustering algorithm.
//
// OUTPUT
// ----------------
//
// MaxClusterRadius  --> maximum radius of the clusters, (rx).
// pci               --> vector of length N where the i th element is the
//                       cluster number to which the i th point belongs.
//                       pci[i] varies between 0 to K-1.
//---------------------------------------------------------------------
```

```
int
KCenterClustering::Cluster()
{
  // randomly pick one node as the first center.
  int nc = (int) pnl_rand_uni(0)*N;  // new center

  // add the ind-th node to the first center.
  pCenters[0] = nc;

  // compute the distances from each node to the first center.
  // initialize the circular linked list, the center is the
  // sentinel node.
  const double *x_nc, *x_j;
  x_nc = px + nc*d;
  x_j = px;
  for (int j = 0; j < N; x_j += d, j++)
  {
    dist_C[j] = (j==nc)? 0.0:ddist(d, x_j, x_nc);
    cnext[j] = j+1;
    cprev[j] = j-1;
  }
  cnext[N-1] = 0; // link the tail to the head.
  cprev[0] = N-1; // link the head to the tail.

  // compute the radius of the first cluster and the farthest
  // node to the center.
  nc = idmax(N,dist_C);
  far2c[0] = nc;
  r[0] = dist_C[nc];
  MaxClusterRadius=sqrt(r[0]);
  numClusters = 1;

  for(int i = 1; i < K && MaxClusterRadius > 0; i++)
  {
    //find the maximum of vector dist_C, i.e., find the node
    //that is farthest away from C. It is a new center.
    nc = idmax(i,r);
    nc = far2c[nc];
    pCenters[i] = nc; //add the ind-th node to the current center.
    r[i] = dist_C[nc] = 0.0;pci[nc]=i;
```

```
far2c[i] = nc;
cnext[cprev[nc]] = cnext[nc]; // delete nc
cprev[cnext[nc]] = cprev[nc];
cnext[nc] = cprev[nc] = nc; //self-loop

//update the distances from each point to the current center.
x_nc = px + nc*d;
for (int j = 0; j < i; j++)
{
  int ct_j = pCenters[j];
  x_j = px + ct_j*d;
  double dc2cq = ddist(d, x_j, x_nc) / 4;
  if (dc2cq < r[j]) // neighbor cluster
  {
    r[j] = 0.0;
    far2c[j] = ct_j;
    int k = cnext[ct_j];
    while (k != ct_j) // visit the circular linked list
    {
      int nextk = cnext[k];
      //compare the distances from new center
      //and from current center.
      double dist2c_k = dist_C[k];
      if ( dc2cq < dist2c_k )
      {

        x_j = px + k*d;
        double dd = ddist(d, x_j, x_nc);
        if ( dd < dist2c_k )
        {
          dist_C[k] = dd; // update distances to center
          pci[k]=i;
          if (r[i] < dd)  // find max r
          {
            r[i] = dd;
            far2c[i] = k;

          }
          cnext[cprev[k]] = nextk; // delete nextk from ct_j
          cprev[nextk] = cprev[k];
          cnext[k] = cnext[nc]; // insert nextk to nc
```

```
            cprev[cnext[nc]] = k;
            cnext[nc] = k;
            cprev[k] = nc;


          }
          else if ( r[j] < dist2c_k )
          {
            r[j] = dist2c_k;
            far2c[j] = k;

          }
        }
        else if ( r[j] < dist2c_k )
        {
          r[j] = dist2c_k;
          far2c[j] = k;
        } // if d < 2 r_k
        k = nextk;
      } // while k
    } // if d < 2 r
  } // for j

  // added by vlad 2/6/07 to make sure that we don't keep clustering once each
  //   otherwise some clusters will have no pts assigned to them
  nc = idmax(i+1,r);
  MaxClusterRadius=sqrt(r[nc]);
  numClusters = i+1;
} // for i

// commented by vlad 2/6/07 to move it above inside of the loop
//nc = idmax(K,r);
//MaxClusterRadius=sqrt(r[nc]);
//numClusters = K; // added by Vlad 1/24/07

return numClusters;
}

void
KCenterClustering::ClusterIncrement( int * nClusters, double * maxRadius )
{
```

```
if( numClusters == 0 )
{
  // randomly pick one node as the first center.
   int nc = (int) pnl_rand_uni(0)*N;
  // add the ind-th node to the first center.
  pCenters[0] = nc;

  // compute the distances from each node to the first center.
  // initialize the circular linked list, the center is the
  // sentinel node.
  const double *x_nc, *x_j;
  x_nc = px + nc*d;
  x_j = px;
  for (int j = 0; j < N; x_j += d, j++)
  {
    dist_C[j] = (j==nc)? 0.0:ddist(d, x_j, x_nc);
    cnext[j] = j+1;
    cprev[j] = j-1;
  }
  cnext[N-1] = 0; // link the tail to the head.
  cprev[0] = N-1; // link the head to the tail.

  // compute the radius of the first cluster and the farthest
  // node to the center.
  nc = idmax(N,dist_C);
  far2c[0] = nc;
  r[0] = dist_C[nc];

  MaxClusterRadius=sqrt(r[0]);
  numClusters++;
}
else
{
  if( numClusters < K && MaxClusterRadius > 0 )
  {
    int i = numClusters;
    int nc;
    const double *x_nc, *x_j;

    //find the maximum of vector dist_C, i.e., find the node
    //that is farthest away from C. It is a new center.
```

```
nc = idmax(i,r);
nc = far2c[nc];
pCenters[i] = nc; //add the ind-th node to the current center.
r[i] = dist_C[nc] = 0.0;pci[nc]=i;
far2c[i] = nc;
cnext[cprev[nc]] = cnext[nc]; // delete nc
cprev[cnext[nc]] = cprev[nc];
cnext[nc] = cprev[nc] = nc; //self-loop

//update the distances from each point to the current center.
x_nc = px + nc*d;
for (int j = 0; j < i; j++)
{
  int ct_j = pCenters[j];
  x_j = px + ct_j*d;
  double dc2cq = ddist(d, x_j, x_nc) / 4;
  if (dc2cq < r[j]) // neighbor cluster
  {
    r[j] = 0.0;
    far2c[j] = ct_j;
    int k = cnext[ct_j];
    while (k != ct_j) // visit the circular linked list
    {
      int nextk = cnext[k];
      //compare the distances from new center
      //and from current center.
      double dist2c_k = dist_C[k];
      if ( dc2cq < dist2c_k )
      {
        x_j = px + k*d;
        double dd = ddist(d, x_j, x_nc);
        if ( dd < dist2c_k )
        {
          dist_C[k] = dd; // update distances to center
          pci[k]=i;
          if (r[i] < dd)  // find max r
          {
            r[i] = dd;
            far2c[i] = k;
          }
          cnext[cprev[k]] = nextk; // delete nextk from ct_j
```

```
              cprev[nextk] = cprev[k];
              cnext[k] = cnext[nc]; // insert nextk to nc
              cprev[cnext[nc]] = k;
              cnext[nc] = k;
              cprev[k] = nc;
            }
            else if ( r[j] < dist2c_k )
            {
              r[j] = dist2c_k;
              far2c[j] = k;
            }
          }
          else if ( r[j] < dist2c_k )
          {
            r[j] = dist2c_k;
            far2c[j] = k;
          } // if d < 2 r_k
          k = nextk;
        } // while k
      } // if d < 2 r
    } // for j

    numClusters++;
    nc = idmax(numClusters,r);
    MaxClusterRadius=sqrt(r[nc]);
  } // if( numClusters < K && MaxClusterRadius > 0 )
} // else ( numClusters > 0 )

  if( nClusters != NULL )
    *nClusters = numClusters;
  if( maxRadius != NULL )
    *maxRadius = MaxClusterRadius;
}


//-----------------------------------------------------------------------
// Computes
// [1] the cluster centers by taking the mean of all the points
// belonging to a cluster.
// [2] the number of points in each cluster.
// [3] the radius of each cluster.
```

```
//-------------------------------------------------------------------------
// NumClusters     --> number of clusters
// pClusterCenters --> pointer to the cluster centers, (d*K),
// pNumPoints      --> pointer to the num of points in each cluster, (K).
// pClusterRadii   --> pointer to the radius of each cluster, (K).
//-------------------------------------------------------------------------

void
KCenterClustering::ComputeClusterCenters(
  int NumClusters,
  double *pClusterCenters,
  int *pNumPoints,
  double *pClusterRadii
  )
{
  int K=NumClusters;

  for(int k=0; k<K; k++)
  {
    pNumPoints[k]=0;
    pClusterRadii[k]=sqrt(r[k]);
    for(int dim=0; dim<d; dim++)
    {
      pClusterCenters[(k*d)+dim]=0.0;
    }
  }

  for(int i=0; i<N; i++)
  {

    pNumPoints[pci[i]] += 1;

    for(int dim=0; dim<d; dim++)
    {
      pClusterCenters[(pci[i]*d)+dim] += px[(i*d)+dim];
}
  }

  for(int k=0; k<K; k++)
  {
    for(int dim=0; dim<d; dim++)
```

```
    {
      pClusterCenters[(k*d)+dim]=pClusterCenters[(k*d)+dim]/pNumPoints[k];
    }
  }

}
//------------------------------------------------------------------------------
// This function computes the constants  2^alpha/alpha!.
// Originally compute_constant_series from ImprovedFastGaussTransform.cpp (IFGT
// source code).
//
//------------------------------------------------------------------------------
static void computeConstantSeries( int d, int pMaxTotal, int pMax, double * cons
{
  int *heads = new int[d+1];
  int *cinds = new int[pMaxTotal];

  for (int i = 0; i < d; i++)
    heads[i] = 0;
  heads[d] = INT_MAX;

  cinds[0] = 0;
  constantSeries[0] = 1.0;
  for (int k = 1, t = 1, tail = 1; k < pMax; k++, tail = t)
  {
    for (int i = 0; i < d; i++)
    {
      int head = heads[i];
      heads[i] = t;
      for ( int j = head; j < tail; j++, t++)
      {
        cinds[t] = (j < heads[i+1])? cinds[j] + 1 : 1;
        constantSeries[t] = 2.0 * constantSeries[j];
        constantSeries[t] /= (double) cinds[t];
      }
    }
  }

  delete [] cinds;
  delete [] heads;
}
```

```
//------------------------------------------------------------------------
// This function computes the monomials [(x_i-c_k)/h]^{alpha} and
// norm([(x_i-c_k)/h])^2.
// Originally compute_source_center_monomials from
// ImprovedFastGaussTransform.cpp (IFGT source code).
//
//------------------------------------------------------------------------
static void computeSourceCenterMonomials( int d, double h, double * dx,
                                  int p, double * sourceCenterMonomials )
{
  int * heads = new int[d];

  for (int i = 0; i < d; i++)
  {
    dx[i]=dx[i]/h;
    heads[i] = 0;
  }

  sourceCenterMonomials[0] = 1.0;
  for (int k = 1, t = 1, tail = 1; k < p; k++, tail = t)
  {
    for (int i = 0; i < d; i++)
    {
      int head = heads[i];
      heads[i] = t;
      for ( int j = head; j < tail; j++, t++)
        sourceCenterMonomials[t] = dx[i] * sourceCenterMonomials[j];
    }
  }

  delete [] heads;
}

//------------------------------------------------------------------------
// This function computes the monomials [(y_j-c_k)/h]^{alpha}
// Originally compute_target_center_monomials from
// ImprovedFastGaussTransform.cpp (IFGT source code).
//------------------------------------------------------------------------
static void computeTargetCenterMonomials( int d, double h, double * dy,
                                  int pMax, double * targetCenterMonomials )
```

```
{
  int *heads = new int[d];

  for (int i = 0; i < d; i++)
  {
    dy[i] = dy[i]/h;
    heads[i] = 0;
  }

  targetCenterMonomials[0] = 1.0;
  for (int k = 1, t = 1, tail = 1; k < pMax; k++, tail = t)
  {
    for (int i = 0; i < d; i++)
    {
      int head = heads[i];
      heads[i] = t;
      for ( int j = head; j < tail; j++, t++)
        targetCenterMonomials[t] = dy[i] * targetCenterMonomials[j];
    }
  }

  delete [] heads;
}
//------------------------------------------------------------------------------
// This function computes the coefficients C_k for all clusters.
// Originally compute_C from ImprovedFastGaussTransform.cpp (IFGT source code)
//------------------------------------------------------------------------------
static void computeC( int d, int N, int W, int K, int pMaxTotal, int pMax,
               double h, int * clusterIndex, double * x, double * q,
               double * clusterCenter, double * C )
{
  double * sourceCenterMonomials = new double[pMaxTotal];
  double * constantSeries = new double[pMaxTotal];
  double hSquare = h*h;
  double * dx = new double[d];

  for (int i = 0; i < W*K*pMaxTotal; i++)
  {
    C[i] = 0.0;
  }
```

```
for(int i = 0; i < N; i++)
{
  int k = clusterIndex[i];
  int sourceBase = i*d;
  int centerBase = k*d;
  double sourceCenterDistanceSquare = 0.0;

  for (int j = 0; j < d; j++)
  {
    dx[j] = (x[sourceBase+j] - clusterCenter[centerBase+j]);
    sourceCenterDistanceSquare += (dx[j]*dx[j]);
  }

  computeSourceCenterMonomials( d, h, dx, pMax, sourceCenterMonomials );

  for(int w = 0; w < W; w++ )
  {
    double f = q[N*w + i]*exp(-sourceCenterDistanceSquare/hSquare);
    for(int alpha = 0; alpha < pMaxTotal; alpha++)
    {
        C[(K*w + k)*pMaxTotal + alpha] += (f*sourceCenterMonomials[alpha]);
    }
  }
}

computeConstantSeries( d, pMaxTotal, pMax, constantSeries );

for(int w = 0; w < W; w++)
{
  for(int k = 0; k < K; k++)
  {
    for(int alpha = 0; alpha < pMaxTotal; alpha++)
    {
      C[(K*w + k)*pMaxTotal + alpha] *= constantSeries[alpha];
    }
  }
}

delete [] sourceCenterMonomials;
delete [] constantSeries;
delete [] dx;
```

```
}
//-----------------------------------------------------------------------------
// This function approximates Gauss Transform.
// Originally constructor, Evaluate(), and destructor from
// ImprovedFastGaussTransform.cpp (IFGT source code).
//-----------------------------------------------------------------------------
static int Ifgt( int d, int N, int M, int W, double * x,
                          double h, double * q, double * y,
                          int pMax, int K, int * clusterIndex,
                          double * clusterCenter, double * clusterRadii,
                          double r, double epsilon, double * g )
{
  //Memory allocation
 // int pMaxTotal = nchoosek(pMax - 1 + d, d);
  int pMaxTotal = pMax;
  double hSquare=h*h;
  double * targetCenterMonomials = new double[pMaxTotal];
  double * dy = new double[d];
  double * C = new double[W*K*pMaxTotal];
  double * ry = new double[K];
  double * rySquare = new double[K];

  for(int i = 0; i < K; i++)
  {
    ry[i] = r + clusterRadii[i];
    rySquare[i] = ry[i]*ry[i];
  }

  ////////////////////////////////////////////////////////////////////////////
  // Evaluate
  ////////////////////////////////////////////////////////////////////////////
  computeC( d, N, W, K, pMaxTotal, pMax, h, clusterIndex, x, q, clusterCenter, C

  for(int j = 0; j < M; j++)
  {
    for( int w = 0; w < W; w++ )
    {
      g[M*w + j] = 0.0;
    }

    int targetBase = j*d;
```

```
    for(int k = 0; k < K; k++)
    {
      int centerBase = k*d;
      double targetCenterDistanceSquare = 0.0;
      for(int i = 0; i < d; i++)
      {
        dy[i] = y[targetBase + i] - clusterCenter[centerBase + i];
        targetCenterDistanceSquare += dy[i]*dy[i];
        if(targetCenterDistanceSquare > rySquare[k]) break;
      }

      if(targetCenterDistanceSquare <= rySquare[k])
      {
        computeTargetCenterMonomials( d, h, dy, pMax, targetCenterMonomials );
        double f=exp(-targetCenterDistanceSquare/hSquare);
        for(int w = 0; w < W; w++ )
        {
          for(int alpha = 0; alpha < pMaxTotal; alpha++)
          {
            g[M*w + j] += (C[(K*w + k)*pMaxTotal + alpha]*f*targetCenterMonomial
          }
        }
      }
    }
  }

  ////////////////////////////////////////////////////////////////////////////
  // Release memory
  ////////////////////////////////////////////////////////////////////////////
  delete [] rySquare;
  delete [] ry;
  delete [] C;
  delete [] dy;
  delete [] targetCenterMonomials;

  return 0;
}
//////////////////end - Helper functions

  static int iFGT(double S0, NumFunc_1  *p, double T, double r, double divid, do
{
```

```
  int ifCall;

  if ((p->Compute) == &Call)
ifCall=1;
  else ifCall=0;

    double K = p->Par[0].Val.V_DOUBLE;
double *x,*y, *d, *q, *v0, *v1, *Int, *b, *c;
int *clusterIndex, *numPoints;
    double  *clusterCenters, *clusterRadii;
int kMax=6;
int dim=1;
double d0=pow(2*var,0.5);
int pMax=35;
double epsilon=0.00001;
double pp;
int i;
double Sl, Sm, Sr;
double pricel, pricem, pricer;
double A, B, C;

  x = new double[M]; // source points
y = new double[M]; // target points
b = new double[M]; // coefficients to solve tridiagonal system
c = new double[M]; // coefficients to solve tridiagonal system
d = new double[M]; //RHS
Int = new double[M]; //integral term
q = new double[M]; // weights for iGFT
v1 = new double[M]; // unknown price at j+1 time step
v0 = new double[M]; // known price at j time step

// INPUT
// ----------------
//
// dim               --> dimension of the points.
// M                 --> number of sources.
// kMax              --> maximal number of clusters.
//
// OUTPUT of k-center clustering
// ----------------
//
```

```
// MaxClusterRadius  --> maximum radius of the clusters, (rx).
// clusterIndex     --> vector of length M where the i th element is the
//                      cluster number to which the i th point belongs.
//                      clusterIndex[i] varies between 0 to kMax-1.
// clusterCenters   --> pointer to the cluster centers, (kMax).
// numPoints        --> pointer to the number of points in each cluster, (kMax).
// clusterRadii     --> pointer to the radius of each cluster, (kMax).


//////////
clusterIndex = new int[M];
  numPoints    = new int[kMax];
  clusterCenters = new double[kMax];
  clusterRadii = new double[kMax];
  //////////

double mu=r-divid-sig*sig/2-lam*(exp(mean+var/2.0)-1.0);
double dt=T/N;
double coef=8.0;
double xmin=coef*log(0.5);
double xmax=coef*log(2.0);
double h=(xmax-xmin)/M;
double t=0.0;
  int Knum;
  double rx;


double l,a,a1,u; // l - lower diagonal element, u - upper diagonal element, a -

l=dt*(-pow(0.5*sig/h,2)+0.25*mu/h);
u=dt*(-pow(0.5*sig/h,2)-0.25*mu/h);
a1=1+0.5*dt*(pow(sig/h,2)+r+lam);
a=1-0.5*dt*(pow(sig/h,2)+r+lam);

double C0=dt*lam/pow(2*M_PI*var,0.5);

    b[0]=0.0;
c[0]=a1;
if((c[0]=a1)==0)
  {
PNL_ERROR("division by zero","increase the number of time steps");
```

```
  }
for (int i=1;i<M;i++)
  {
b[i]=l/c[i-1];
c[i]=a1-b[i]*u;
if((c[i]=a1-b[i]*u)==0)
  {
PNL_ERROR("division by zero","increase the number of time steps");
  }
  }

    for (int i=0;i<M;i++)
  {
x[i]=xmin+i*h;
y[i]=x[i]+mean;
Int[i]=0.;
if (ifCall==1) // Call
  { if (x[i]>0)
  {v0[i]=K*exp(x[i])-K;}
else {v0[i]=0;}
  }
else   // Put
  { if (x[i]<0)
  {v0[i]=K-K*exp(x[i]);}
else {v0[i]=0;}
  }
  }

    // do k-center clustering

    KCenterClustering* pKCC = new KCenterClustering(dim,M,x,clusterIndex,kMax);
    Knum = pKCC->Cluster();
    rx = pKCC->MaxClusterRadius;
    pKCC->ComputeClusterCenters(Knum, clusterCenters, numPoints, clusterRadii);

    delete pKCC;
// end k-center clustering


//////////////////
```

```
    if (ABS(a1)>ABS(u)+ABS(l))
  {
for (int j=1;j<=N;j++)
  { t=j*dt;

for (int i=0;i<M;i++)
  {   q[i]=v0[i]*h;
Int[i]=0;
  }
q[0]=0.5*q[0];
q[M-1]=0.5*q[M-1];
Ifgt( 1, M, M, 1, x, d0, q, y, pMax, Knum, clusterIndex, clusterCenters, cluster
////////// // Do Down-Solve (I+0.5A) vj+1 = (I-0.5A) vj+C Intj   //
d[0]=a*v0[0]-u*v0[1]+C0*Int[0];
for (int i=1;i<M-1;i++)
  {
d[i]=(-l*v0[i-1]+a*v0[i]-u*v0[i+1]+C0*Int[i])-d[i-1]*b[i];
  }
d[M-1]=(-l*v0[M-2]+a*v0[M-1]+C0*Int[M-1])-d[M-2]*b[M-1];
////////// // Do Up-Solve (I+0.5A) vj+1 = (I-0.5A) vj+C Intj   //
v1[M-1]=d[M-1]/c[M-1];
for (int i=M-2;i>=0;i--)
  {
v1[i]=(d[i]-u*v1[i+1])/c[i];

  }
for (int i=0;i<M;i++){v0[i]=v1[i];}
  }
  }
else
  {
for (int i=0;i<M;i++)
  {
v1[i]=0.;
v0[i]=0.;
  }
  }
pp=log(S0/K);
i=0;
while ((x[i]<=pp)&&(i<M-1)) {i++;}
i=i-1;
```

```
//Price, quadratic interpolation
Sl = K*exp(x[i-1]);
Sm = K*exp(x[i]);
Sr = K*exp(x[i+1]);
// S0 is between Sm and Sr
pricel = v1[i-1];
pricem = v1[i];
pricer = v1[i+1];
//quadratic interpolation
A = pricel;
B = (pricem-pricel)/(Sm-Sl);
C = (pricer-A-B*(Sr-Sl))/(Sr-Sl)/(Sr-Sm);
//Price
*price = A+B*(S0-Sl)+C*(S0-Sl)*(S0-Sm);
if(*price<0) {*price=0.0;}
//Delta
*delta = B + C*(2*S0-Sl-Sm);
//printf("price =  %f %f %f \ n", x[i], *price, v1[i]);

delete [] x;
delete [] y;
delete [] v0;
delete [] v1;
delete [] q;
delete [] d;
delete [] Int;
delete [] b;
delete [] c;

delete [] clusterIndex;
delete [] numPoints;
delete [] clusterCenters;
delete [] clusterRadii;

  return OK;
}

  int CALC(FD_iFGT)(void *Opt, void *Mod, PricingMethod *Met)
  {
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;
    TYPEMOD *ptMod = (TYPEMOD *)Mod;
```

```
    double r, divid;

    r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);
    divid = log(1. + ptMod->Divid.Val.V_DOUBLE / 100.);

    return iFGT(ptMod->S0.Val.V_PDOUBLE,
ptOpt->PayOff.Val.V_NUMFUNC_1, ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
  }

  static int CHK_OPT(FD_iFGT)(void *Opt, void *Mod)
  {
    if ((strcmp(((Option *)Opt)->Name, "CallEuro") == 0) || (strcmp(((Option *)O
      return OK;

    return WRONG;
  }
#endif //PremiaCurrentVersion


  static int MET(Init)(PricingMethod *Met, Option *Opt)
  {
    static int first = 1;

    if (first)
      {
        Met->Par[0].Val.V_INT2 = 800;
        Met->Par[1].Val.V_INT2 = 800;
        first = 0;
      }

    return OK;
  }

  PricingMethod MET(FD_iFGT) =
  {
    "FD_iFGT",
    { {"TimeStepNumber", INT2, {500}, ALLOW}, {"SpaceStepNumber", INT2, {100}, A
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CALC(FD_iFGT),
    {{"Price", DOUBLE, {100}, FORBID}, {"Delta", DOUBLE, {100}, FORBID}, {" ", P
```

```
    CHK_OPT(FD_iFGT),
    CHK_split,
    MET(Init)
  };

}
```