Help

```
extern "C" {
#include "nonpar1d_vol.h"
}
#include <vector>
#include <cmath>
#include <pnl/pnl_finance.h>
#include <stdio.h>
#include "pnl/pnl_vector.h"

extern "C" {

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2016+2) //The "#els
  static int CHK_OPT(AP_NONPAR_VOLATILITYINDEX)(void *Opt, void *Mod)
  {
    return NONACTIVE;
  }
  int CALC(AP_NONPAR_VOLATILITYINDEX)(void *Opt, void *Mod, PricingMethod *Met)
  {
    return AVAILABLE_IN_FULL_PREMIA;
  }
#else

typedef struct option_index {
    double strike;
    double call_price;
    double put_price;
} option_index;

static double get_d2(double k,double t,double v)
{
    double a=v*sqrt(t);
    double d2=-k/a-a/2;
    return d2;
}


//compute distance between 2 points (x1,y1) and (x2,y2)
static double get_distance(double x1, double y1, double x2, double y2)
{
```

```
    return sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
}

//compute volatility index, fn - file name with option prices
//option strikes in file must be sorted
//s - asset price
//T - time to maturity

/*////////////////////////////////////////////////////*/
int ap_nonpar_volindex(char *fn, double s,  double T, double r,double *ptprice)
{

    const int max_options=50;
    option_index options[max_options];

    //Reading option data from file fn
    FILE* f=fopen(fn, "r");
    if (f == NULL)
    {
        perror("Could not open file");
        return 0;
    }
    int n=0;
    int iatm=0; // index of atm put
double min=s;
    while (!feof(f))
    {
        fscanf(f, "%lf %lf %lf\ n",&options[n].strike,&options[n].call_price,&op
        if ( std::abs(options[n].call_price-options[n].put_price)<min)
{min=std::abs(options[n].call_price-options[n].put_price); iatm=n;}
        n++;
    }

    fclose(f);

    std::vector<double> x1(n);
std::vector<double> y1(n);
double v,k,d2;
int m=0;
int imin=iatm;
    int e;
```

```
int st=1;
    for (int i=iatm;i>=0;i--)
    {
        if (options[i].put_price==0)
            continue;
        v=pnl_bs_implicit_vol(0, exp(r*T)*options[i].put_price, s, options[i].st
        k=log(options[i].strike/s);
if (v==0){v=0.0000000001;}
d2=get_d2(k,T,v);
        if (i<iatm)
  {
if (d2>x1[i+1]) {imin=imin-st;}

else {st=0;}
  }
        x1[i]=d2;
        y1[i]=v*v;
}
    m=iatm-imin+1;
st=1;
    for (int i=iatm+1;i<n;i++)
    {
        if (options[i].call_price==0)
            continue;
        v=pnl_bs_implicit_vol(1, exp(r*T)*options[i].call_price, s, options[i].s
        k=log(options[i].strike/s);
        if (v==0){v=0.0000000001;}
d2=get_d2(k,T,v);
if (d2<x1[i-1]){m=m+st;}
else {st=0;}
        x1[i]=d2;
        y1[i]=v*v;

    }

    std::vector<double> x(m);
std::vector<double> y(m);
for (int j=0;j<m;j++)
{
x[j]=x1[imin+m-j-1];
y[j]=y1[imin+m-j-1];
```

```
}
if (m<=1)
    {
        printf("Number of points must be at least 2\ n");
        return 0;
    }

    std::vector<double> dy(m); // derivative of y
    dy[0]=0;
    dy[m-1]=0;

    if (m>2)
    {
        double l1=get_distance(x[0], y[0], x[1], y[1]);
        for (int j=1;j<m-1;j++)
        {
            double l2=get_distance(x[j], y[j], x[j+1], y[j+1]);
            dy[j]=-((x[j+1]-x[j])/l2-(x[j]-x[j-1])/l1)/((y[j+1]-y[j])/l2-(y[j]-y
        l1=l2;
        }
    }

    //create a cubic polynomial approximation
    const std::vector<double>& b=dy;
    std::vector<double> c(m-1);
    std::vector<double> d(m-1);
    for (int j=0;j<m-1;j++)
    {
        double dxj=x[j+1]-x[j];
        double dyj=y[j+1]-y[j];
        c[j]=(3*dyj-dxj*dy[j+1]-2*dxj*dy[j])/(dxj*dxj);
        d[j]=(dyj-dy[j]*dxj-c[j]*dxj*dxj)/(dxj*dxj*dxj);
    }

    //integrate the approximate function
    double cdf_x1=pnl_cdfnor(x[0]);
    double sum=y[0]*cdf_x1;
    double pdf_x1=pnl_normal_density(x[0]);
    for (int j=0;j<m-1;j++)
    {
        double cdf_x2=pnl_cdfnor(x[j+1]);
```

```
        double pdf_x2=pnl_normal_density(x[j+1]);
        double ka=cdf_x2-cdf_x1;
        double kb=-(pdf_x2-pdf_x1)-x[j]*(cdf_x2-cdf_x1);
        double kc=-(x[j+1]*pdf_x2-x[j]*pdf_x1)+2*x[j]*(pdf_x2-pdf_x1)+(1+x[j]*x[
        double kd=(1-x[j+1]*x[j+1])*pdf_x2-(1-x[j]*x[j])*pdf_x1+3*x[j]*(x[j+1]*p
        sum+=y[j]*ka+b[j]*kb+c[j]*kc+d[j]*kd;
cdf_x1=cdf_x2;
        pdf_x1=pdf_x2;
    }
sum+=y[m-1]*(1-cdf_x1);

//Volatility Index
    *ptprice=sqrt(sum)*100.0;

  return OK;
}


  int CALC(AP_NONPAR_VOLATILITYINDEX)(void *Opt, void *Mod, PricingMethod *Met)
  {
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;
    TYPEMOD *ptMod = (TYPEMOD *)Mod;
    double r,  spot;
    NumFunc_1 *p;

    r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);
    p = ptOpt->PayOff.Val.V_NUMFUNC_1;
    spot = ptMod->S0_volindex.Val.V_DOUBLE;

    return ap_nonpar_volindex(ptMod->option_prices.Val.V_FILENAME, spot,
             ptOpt->Maturity.Val.V_DATE, r,
             &(Met->Res[0].Val.V_DOUBLE));


  }

  static int CHK_OPT(AP_NONPAR_VOLATILITYINDEX)(void *Opt, void *Mod)
  {
    if ((strcmp(((Option *)Opt)->Name, "VolatilityIndex") == 0))
      return OK;
```

```
    return WRONG;
  }

#endif //PremiaCurrentVersion

  static int MET(Init)(PricingMethod *Met, Option *Opt)
  {
    Met->HelpFilenameHint = "ap_nonparam_volatilityindex";
    return OK;
  }

  PricingMethod MET(AP_NONPAR_VOLATILITYINDEX) =
  {
    "AP_NONPAR_VOLATILITYINDEX_FUKASAWA",
    {{" ", PREMIA_NULLTYPE, {0}, FORBID}},
    CALC(AP_NONPAR_VOLATILITYINDEX),
    { {"VolatilityIndex", DOUBLE, {100}, FORBID},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CHK_OPT(AP_NONPAR_VOLATILITYINDEX),
    CHK_ok ,
    MET(Init)
  } ;

  /*////////////////////////////////////*/
}
```