ISYE 6679 Computational Methods

Interim Project Report

On

Parallel Computing in finance for estimating risk-neutral densities in option prices

Ana M. Monteiro, Antonio A.F. Santos

Team Members

Karuna Kumar

Rishi Dasgupta

INDEX

1. Introduction

This project aims to replicate results from the paper titled, “Parallel Computing in finance for estimating risk-neutral densities in option prices” by Ana M. Monteiro and Antonio A.F. Santos from the University of Coimbra, Portugal. This paper discusses the computational advantages offered by parallel computing implementations for solving complex calculations over large datasets, which are commonly involved in assessing financial risk.

Financial options markets assume a central role in all economic systems, which makes estimating the options pricing function imperative for generating sound risk-management strategies. For this purpose, non-parametric estimation methods can be used to estimate the options pricing function (as opposed to parametric estimation which may impose no-negligible constraints). However, this requires large datasets and can be computationally intensive, especially in estimating the optimal kernel bandwidths associated with the same.

The paper presents a case study to identify optimal kernel bandwidths for the non-parametric estimation of risk-neutral densities in options prices. For this purpose, two different datasets are used – the intraday S&P500 index and the CBOE’s VIX (volatility index) put/call/strike values extracted from Yahoo Finance between April 16th to April 20th, 2018, for contracts with a maturity date of May 18th, 2018. Two alternate implementations are provided – a sequential C++ program that utilizes CPU on local hardware, and a parallel computing implementation using CUDA/C++ code that utilizes GPU computing on NVIDIA hardware provided by Pace-ICE.

The algorithm used includes a tailor-made cross-validation criterion function and minimum grid-search methods to identify optimal kernel bandwidths over both datasets. The elapsed time is recorded for grid sizes - 16x16, 32x32, 64x64, 128x128, and 256x256 for both datasets, and the results are compared with its parallel computing implementation.

For the interim project report, we have implemented the sequential algorithm and tabulated results for the optimal kernel bandwidths for *Calls* and *Puts*, as well as the total computational times for the same. In the next phase of the project, we shall implement the parallel computing algorithm and compare the results to assess the latter’s computational efficiency.

2. Model Outline

2.1 Option pricing

A European-type call/put option is a contract giving the right, but not the obligation for the holder to purchase or sell a given stock S, at a predetermined price X, at the maturity date T. A central object in option pricing is the risk-neutral density (RND) which informs derivative security pricing at an expected value at the time of contract maturity. This would allow for lower (neutral) risk in negotiating option contracts as the risk profile of equities can be predicted and assessed prior to maturity.

The expected values of equities at the time of maturity can be calculated through a risk-neutral density, discounted by a risk-free interest rate. Let us consider the price of a call option contract as the function C (S_t, X, r, τ, σ), which depends on the stock price S_t, risk-free interest rate r, time to maturity τ = T – t, and σ, a parameter characterizing the volatility of the stock price. Considering q(.) as the risk-neutral density at maturity, the prices of call and put options are given as follows:

$$C(S_t, X, r, \tau, \sigma) = e^{-r\tau} \int_X^\infty (S_T - X) q(S_T)\, dS_T, \qquad (1)$$

$$P(S_t, X, r, \tau, \sigma) = e^{-r\tau} \int_0^X (X - S_T) q(S_T)\, dS_T. \qquad (2)$$

A fundamental result due to [1] and [2] relates the risk-neutral density with the second-order derivative of the option pricing function with respect to the strike. The first and second-order derivatives of (1) and (2) are as follows:

$$\frac{\partial C(X, S_t, r, \tau, \sigma)}{\partial X}_{|X=S_T} = G(S_T | S_t, r, \tau, \sigma) - 1 \qquad (3)$$

$$\frac{\partial P(X, S_t, r, \tau, \sigma)}{\partial X}_{|X=S_T} = G(S_T | S_t, r, \tau, \sigma) \qquad (4)$$

$$\frac{\partial^2 C(X, S_t, r, \tau, \sigma)}{\partial X^2}_{|X=S_T} = \frac{\partial^2 P(X, S_t, r, \tau, \sigma)}{\partial X^2}_{|X=S_T} = e^{-r\tau} q(S_T), \qquad (5)$$

Where G is the cumulative distribution function associated with q. These results allow the risk-neutral density to be estimated from the second derivative of the option pricing function. The no-arbitrage assumption implies an additional set of constraints as follows:

$$C(X, S_t, r, \tau, \sigma) \geq 0; \qquad P(X, S_t, r, \tau, \sigma) \geq 0, \qquad (6)$$

$$\begin{aligned} -e^{-r\tau} \leq \frac{\partial C}{\partial X}(X, S_t, r, \tau, \sigma) \leq 0; \\ 0 \leq \frac{\partial P}{\partial X}(X, S_t, r, \tau, \sigma) \leq e^{-r\tau}, \end{aligned} \qquad (7)$$

$$\frac{\partial^2 C}{\partial X^2}(X, S_t, r, \tau, \sigma) \geq 0; \qquad \frac{\partial^2 P}{\partial X^2}(X, S_t, r, \tau, \sigma) \geq 0. \qquad (8)$$

These constraints provide preliminary information to be considered in the elimination process jointly with sample price data, which allows for a more robust estimation with readily interpretable estimates.

2.2 Non-parametric risk-neutral density

In the non-parametric setting, observed options prices are subjected to random noise. The theoretical call price C (S_t, X, r, τ, σ) relates to the observed C* (S_t, X, r, τ, σ) through the regression model as follows:

$$C^*(S_t, X_i, r, \tau, \sigma) = C(S_t, X_i, r, \tau, \sigma) + \varepsilon_i, \ i = 1, \ldots, n_c, \qquad (9)$$

Here, n_c is the sample size of all call prices, and the same treatment is given to put prices. The Nadaraya-Watson (NW) estimator is used to approximate C (S_t, X_i) considering a partition for strike values formed by $\{x_j\}_{j=1}^{m}$. For a given point x_j, the estimates are as follows:

$$\hat{C}(S_t, X_i) = \frac{\sum_{i=1}^{n_c} K\left(\frac{X_i - x_j}{h_c}\right) C^*(S_t, X_i)}{\sum_{i=1}^{n_c} K\left(\frac{X_i - x_j}{h_c}\right)}, \ j = 1, \ldots, m. \qquad (10)$$

where K(·) is a kernel function used to weight observations neighboring x j, which depends on the bandwidth parameter h_c. The fitting to data depends essentially on the bandwidth value, more than the kernel K(·) functional form.

Regarding option contracts observations, call and put prices are available, and both give relevant information for risk-neutral density estimation. By approximating the call (put) option pricing function as $\hat{f}_c$ $(\hat{f}_p)$ through a Taylor polynomial expansion of order three, at point x_j, j = 1, ...,n.

$$\hat{f}_c(x_j) = \beta_{0,c} + \beta_{1,c}(X_{i,c} - x_j) + \frac{\beta_{2,c}}{2!}(X_{i,c} - x_j)^2 + \frac{\beta_{3,c}}{3!}(X_{i,c} - x_j)^3, \tag{11}$$

$$\hat{f}_p(x_j) = \beta_{0,p} + \beta_{1,p}(X_{i,p} - x_j) + \frac{\beta_{2,p}}{2!}(X_{i,p} - x_j)^2 + \frac{\beta_{3,p}}{3!}(X_{i,p} - x_j)^3, \tag{12}$$

where $X_{i,c}$ and $X_{i,p}$ are the call and put strikes. The variables $\beta_{0,c}$ and $\beta_{0,p}$ represent the function level estimates for calls and puts, respectively, $\beta_{1,c}$ and $\beta_{1,p}$ are the first derivatives, $\beta_{2,c}$ and $\beta_{2,p}$ are the second derivatives, and finally, $\beta_{3,c}$ and $\beta_{3,p}$ the third derivatives estimates.

Considering the information on the call and put prices and the set of constraints implied from (3) to (8), the risk-neutral density is estimated by solving the problem:

$$\underset{\beta}{\text{minimize}} \sum_{i=1}^{n_c} \left(C_i^* - \sum_{k=0}^{3} \frac{\beta_{k,c}}{k!}(X_{i,c} - x_j)^k \right)^2 K\left(\frac{X_{i,c} - x_j}{h_c}\right) w_{c,i} + \sum_{i=1}^{n_p} \left(P_i^* - \sum_{k=0}^{3} \frac{\beta_{k,p}}{k!}(X_{i,p} - x_j)^k \right)^2 K\left(\frac{X_{i,p} - x_j}{h_p}\right) w_{p,i} \tag{13}$$

$$\text{subject to} \quad -\beta_{1,c} + \beta_{1,p} = 1 \tag{14}$$

$$\beta_{2,c} - \beta_{2,p} = 0 \tag{15}$$

$$-e^{-r\tau} < \beta_{1,c} < 0 \tag{16}$$

$$0 < \beta_{1,p} < e^{-r\tau} \tag{17}$$

$$\beta_{0,c}, \beta_{0,p}, \beta_{2,c}, \beta_{2,p} \geq 0, \tag{18}$$

where $\beta = \{\beta_{0,c}, \ldots, \beta_{3,c}, \beta_{0,p}, \ldots, \beta_{3,p}\}$, and $w_{c,i}$ and $w_{p,i}$ are weights associated with each observation, usually accounting for data heteroscedasticity. In option prices, the weights can be associated with volume or open interest, which usually assume significant values for options near-the-money and tend to zero for deep-out- and deep-in-the-money options.

Solving the problem and finding the estimates for each x_j is not computationally challenging. If the grid associated with x_j contains a few hundred elements, available computational capabilities can solve the problem in a few seconds. The main computational challenge is finding the optimal kernel bandwidths h_c and h_p. For the practical application of these methods, when they are applied sequentially in risk-neutral density updates, excessive computational times needed to calculate bandwidths can turn such time-varying analysis infeasible.

2.3 Cross-Validation

Cross-validation (CV) is a standard method to define the necessary bandwidth values for nonparametric estimators. It is the most computationally intensive element in a nonparametric estimation setting. The common feature is the inclusion in the cross-validation criterion function of a squared-error component. There are many cases where it is sound to introduce new factors in the CV function apart from the squared-error component, which intends to accommodate the particularities of the problem at hand. By its nature, second derivative estimators are subjected to estimation errors when compared with estimators for the function itself. In a non-adapted framework, the increased estimation errors can lead to estimates incompatible with an interpretation that links them to density functions. Furthermore, the density must not present great shape diversity for different sub-ranges; for example, a highly multi-modal density is difficult to interpret. On the other hand, extreme homogeneity within sub-ranges is also not desirable, for example, a uniform density. Measurements of area, variation, and entropy redefine the cross-validation function to adequately define bandwidth values that influence the shape of estimated density functions.

Regarding standard forms of cross-validation criterion functions, the leave-k$_i$-out estimator is used instead of the leave-one-out for a fixed design setting. The problem is defined as:

$$\underset{h}{\text{minimize}} \ \frac{1}{n}\sum_{i=1}^{n}\frac{1}{k_i}\sum_{j=1}^{k_i}\left(y_{i,j}-\hat{f}_{(-k_i)}(X_i)\right)^2. \tag{19}$$

The area associated with the second derivative function is used to penalize the criterion function when the area associated with the second derivative differs substantially from one,

$$A(h)=1+\left|\sum_{j=1}^{m}\left(x_j^*-x_{j-1}^*\right) f^{(2)}\left(\frac{x_j^*+x_{j-1}^*}{2}\right)-1\right|. \tag{20}$$

The estimates for the RND should not present spikes and non-differentiable points since it is difficult for meaningful economic interpretations. To avoid such scenarios, the function,

$$V(h)=\sum_{j=1}^{m}\left|f^{(2)}(x_j^*)-f^{(2)}(x_{j-1}^*)\right|, \tag{21}$$

is used to penalize the criterion function when extreme variations occur. Finally, entropy measures compensate for the tendency to define a flat density. Considering the same approach adopted for the area, a function,

$$H(h) = -\sum_{j=1}^{m}\left(x_j^*-x_{j-1}^*\right) f^{(2)}\left(\frac{x_j^*+x_{j-1}^*}{2}\right)\log\left(f^{(2)}\left(\frac{x_j^*+x_{j-1}^*}{2}\right)\right). \tag{22}$$

is defined as the approximation of an entropy measure.

The CV criterion function is tailored-made for this context. The bandwidth values hc and hp are defined by solving the problem,

$$\underset{h_c,h_p}{\text{minimize}} \ \frac{1}{n_c}\sum_{i=1}^{n_c}\frac{1}{k_{i,c}}\sum_{j=1}^{k_{i,c}}\left[\left(C_{i,j}^*-\hat{f}_{c(-k_{i,c})}(X_i)\right)^2 V_{c,p}+\frac{A_{c,p}}{H_{c,p}}\right] + \frac{1}{n_p}\sum_{i=1}^{n_p}\frac{1}{k_{i,p}}\sum_{j=1}^{k_{i,p}}\left[\left(P_{i,j}^*-\hat{f}_{p(-k_{i,p})}(X_i)\right)^2 V_{c,p}+\frac{A_{c,p}}{H_{c,p}}\right]. \tag{23}$$

The standard optimization problem has been modified to apply the CV method to choose optimal bandwidths, including what can be interpreted as the addition of a regularization factor. Finding the optimal values of hc and hp that solve the problem (23) represents a tremendous computational challenge, mainly when a grid-search approach is used. In a sequential framework, thinner grids are used for more accurate approximations; estimates are obtained by solving constrained quadratic programming problems.

3. Algorithm Design

We have utilized grid-based optimization methods to obtain the optimal values for equation (23), it is not possible to use gradient-based methods for the same since the assumptions of convexity and continuity cannot be postulated.

Algorithm 1 programs the minimization of an objective function through matrix-grid search. Given a matrix with the objective function values, lines are associated with the first parameter grid values, and columns are associated with the second. A correspondence is established between the function minimum value and grid indexes with each parameter.

Algorithm 1: Minimum grid-search.

```
Data: $h_c = \{h_{c_1}, \ldots, h_{c_n}\}$; $h_p = \{h_{p_1}, \ldots, h_{p_m}\}$; $n$; $m$; $size = n \times m$;
      $cvvec = \{cv_1, \ldots, cv_{size}\}$
Result: $h_c^*$; $h_p^*$

$min \leftarrow cvvec[1]$;
$row \leftarrow 1$; $col \leftarrow 1$;
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $m$ do
        if $cvvec[(i-1) \times n + j] < min$ then
            $min \leftarrow cvvec[(i-1) \times n + j]$;
            $row \leftarrow i$; $col \leftarrow j$;
        end
    end
end
$h_c^* \leftarrow hc[row]$; $h_p^* \leftarrow hp[col]$;
```

Algorithm 2 presents a sequential approach to populate the matrix associated with the grid search. The algorithm is based on a 3-layer loop, with an inner function *estimCVElements* which can take several seconds to compute.

Algorithm 2: CV matrix build sequential.

```
Data: $h_c = \{h_{c_1}, \ldots, h_{c_n}\}$; $h_p = \{h_{p_1}, \ldots, h_{p_m}\}$; $strike = \{s_1, \ldots, s_{n_u}\}$;
      $size = n \times m$
Result: $cvvec = \{cv_1, \ldots, cv_{size}\}$

for $k \leftarrow 1$ to $n_u$ do
    $pos \leftarrow 0$;
    for $i \leftarrow 1$ to $n$ do
        for $j \leftarrow 1$ to $m$ do
            $pos \leftarrow pos + 1$;
            $[cv, var, area, entropy] \leftarrow$
              estimCVElements($hc[i], hp[j], strike[k]$);
            $cvvec[pos] \leftarrow cvvec[pos] + cv \times var + \frac{1+|area-1|}{entropy}$;
        end
    end
end
```

Algorithm 3 presents the code for the *estimCVElements* function. To define the output elements cv, area, entropy, and var may require several tens of thousands of constrained quadratic optimization problems to be solved.

```
Algorithm 3: estimCVElements.
Data: $x = \{x_1, \ldots, x_m\}$; $\{callprice, putprice\}$; $x_{ex}$
Result: $cv$; $area$; $entropy$; $var$

$ddf \leftarrow$ allocate vector density estimates dimension $m$;
for $i \leftarrow 1$ to $m$ do
    $ddf[i] \leftarrow$ estimxex($x[i], x_{ex}$);
end

$area \leftarrow$ areaEstim($x, ddf$);
$entropy \leftarrow$ entropyEstim($x, ddf$);
$var \leftarrow$ varEstim($ddf$);

$[fcall, fput] \leftarrow$ estimxex($x_{ex}, x_{ex}$);

$cvcall \leftarrow 0.0$;
for $i \leftarrow 1$ to $n_c$ do
    if $x[i] = x_{ex}$ then
        $cvcall \leftarrow cvall + (callprice[i] - fcall)^2$;
    end
end

$cvput \leftarrow 0.0$;
for $i \leftarrow 1$ to $n_p$ do
    if $x[i] = x_{ex}$ then
        $cvput \leftarrow cvput + (putprice[i] - fput)^2$;
    end
end
$cv \leftarrow cvcall + cvput$;
```

4. Results

VIX Sequential

The following table tabulates the optimal kernel bandwidths h_c and h_p for Calls and Puts in VIX data for the sequential C++ algorithm, performed using Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz. The grid [0.75 2.0] was assumed for h_c and h_p. The underlying [10.0 47.5] range within a grid of 128 values was considered. These values can be changed in the main.cpp file. Sample Size – Calls: 1465, Puts: 1196

Grid Dimensions	hc	hp	Elapsed Time (sec)
16x16	1.4167	1.0833	208.925
32x32	1.3952	1.0726	823.958
64x64	1.3849	1.0873	5178.250
128x128	1.3799	1.0846	13117.904

Table 1: Results for VIX index RND bandwidths

S&P 500 Sequential

The following table tabulates the optimal kernel bandwidths h_c and h_p for Calls and Puts in SP500 data for the sequential C++ algorithm. The grid [0.25 1.25] was assumed for h_c and h_p. The underlying [24.0 28.0] range within a grid of 128 values was considered (though the algorithm terminated only for grid dimensions up to 32x32 within a day, only for which results are shown below). These values can similarly be changed in the main.cpp file. Sample Size – Calls: 9785, Puts: 13574

Grid Dimensions	hc	hp	Elapsed Time (sec)	CPU
16x16	0.5833	0.6500	12828.699	Intel(R) Celeron(R) N4020 CPU @ 1.10GHz
32x32	0.6052	0.6691	>28800	11th Gen Intel(R) Core (TM) i5-1135G7 @ 2.40GHz

Table 2: Results for SP500 index RND bandwidths

Running with grid dimensions of 64x64 on an Intel(R) Celeron(R) N4020 CPU @ 1.10GHz exceeds runtime of 48 hours.

5. Inferences and Future Implementation

In the first phase of this project, we have identified the optimal kernel bandwidth values for calls/puts through risk-neutral densities in option pricing. We have used a grid-based optimization method with a tailor-made cross-validation criterion function using sequential C++ codes run on local CPUs, with grid sizes ranging in 16x16, 32x32, 64x64, and 128x128 dimensions.

For the VIX RND bandwidth values, we ran a sample size of 1465 Calls and 1196 Puts over 29 Strikes recorded between 16 to 20 April 2018 for contracts maturing on May 18, 2018. Since timing is imperative for making decisions regarding equity price expectations and assessing financial risks associated with options contracts, lower computational times are preferred, especially considering the Big Data required to generate the non-parametric RN densities for the options pricing function.

As we can see from Table 1, the computational times associated with higher matrix grid dimensions are significant and exponentially increasing. This opens up the possibility of the options pricing function losing relevance in real-time decision-making as the optimal values take longer and longer to estimate. Thus, it is imperative for a parallel computing alternative to be implemented so that computational times are shortened to within a couple of hours at higher grid dimensions.

For the SP500 RND bandwidth values, we ran a sample size of 9785 Calls and 13574 Puts over 90 strikes, also recorded between 16 to 20 April 2018 for contracts maturing on May 18, 2018. As we can see in table 2, computational times in sequential operations are significantly higher owing to larger sample size, with higher grid dimensions (64x64 and above) taking several days to compute. In practical applications, these computational times may far exceed the affordable threshold for real-time decision making in financial risk which, accounting for the time required for auxiliary decisions, may even exceed the time to maturity. This would render the computation irrelevant, which is why we require faster computational processes using parallel computing methods.

Future improvements regarding sequential implementations would involve narrowing down the bandwidth range and checking the impact on computational times, and implementing a parallel computing algorithm in CUDA for the same.

Potential Challenges to be encountered in the parallel computing implementation could be with regards to data transfer overhead between CPU and GPU considering the large data size, possible thread divergence, possible synchronization issues (missed sync() functions in code), and possible limitations in memory resources, especially for the S&P 500 data.

6. References

[1] D.T. Breeden, R.H. Litzenberger, Prices of state-contingent claims implicit in option prices, J. Bus. (1978) 621-651

[2] R.W. Banz, M.H. Miller, Prices for state-contingent claims: some estimates and Applications. J. Bus (1978) 653-672

[3] Ana M. Monteiro, Antonio A.F. Santos, Parallel computing in finance for estimating risk-neutral densities through options prices, Journal of Parallel and Distributed Computing, 2021

7. Appendix

```
#include "runfunc.hpp"
```

```
//
//
template <typename T>
T* readArray(const char* file,int *n)
{
        FILE *fp;
        fp = fopen(file, "r");
        char buff[64];
        int nrow = -1;
        while (!feof(fp)) {
            int tt = fscanf(fp, "%s",buff);
            nrow++;
        }
        *n = nrow;
        rewind(fp);
        T *result = (T*)malloc(nrow*sizeof(T));
        for(int i=0;i<nrow;i++){
            int tt = fscanf(fp, "%s",buff);
            result[i] = (T)atof(buff);
        }
        fclose(fp);
        return result;
}
//
//
//
//
void minMatrix(double *res,double *xxx,int nx,double *yyy,int ny,double *mat)
{
    double min = mat[0];
    int row = 0;
    int col = 0;
    for(int i=0;i<nx;i++){
        for(int j=0;j<ny;j++){
            if(mat[i*nx + j] < min){
                min = mat[i*nx + j];
                row = i;
                col = j;
            }
        }
    }
    res[0] = xxx[row];
    res[1] = yyy[col];
}
//
//
//
//
```

```c
double* linspace(double x0,double x1,int n)
{
    double *xxx = (double*)malloc(n*sizeof(double));
    double step = (x1 - x0)/(double)(n - 1);
    xxx[0] = x0;
    for(int i=1;i<n;i++){
        xxx[i] = xxx[i-1] + step;
    }
    return xxx;
}
//
//
//
double areaEstim(double *x0,double *yy,int n)
{
     double sum = 0.0;
     for(int i=0;i<n;i++){
        if(i==0){
             sum  += 0.5*(x0[i+1] - x0[i])*yy[i];
             }else if(i==(n-1)){
                sum += 0.5*(x0[i] - x0[i-1])*yy[i];
             }else{
                sum += 0.5*(x0[i+1] - x0[i-1])*yy[i];
            }
     }
    return sum;
}
//
//
//
double entropyEstim(double *x0,double *yy,int n)
{
        double area = areaEstim(x0, yy, n);
        double *y0 = (double*)calloc(n,sizeof(double));
        for(int i=0;i<n;i++) y0[i] = yy[i]/area;
        double sum = 0.0;
        for(int i=0;i<n;i++){
            if(i==0){
                    if(y0[i] > 0.0001) sum  += 0.5*(x0[i+1] - x0[i])*y0[i]*log(y0[i]);
                }else if(i==(n-1)){
                    if(y0[i] > 0.0001) sum += 0.5*(x0[i] - x0[i-1])*y0[i]*log(y0[i]);
                }else{
                    if(y0[i] > 0.0001) sum += 0.5*(x0[i+1] - x0[i-1])*y0[i]*log(y0[i]);
                }
        }
        free(y0);
        return -1.0*sum;
}
```

```
//
//
//
double varEstim(double *yy,int n)
{
   double sum = 0.0;
   for(int i=1;i<n;i++){
      sum += fabs(yy[i] - yy[i-1]);
   }
   return sum;
}
//
//
//
void run_in_cpu(double hcmin,double hcmax,int nhc,double hpmin,double hpmax,int nhp,double
xmin,double xmax,int nx,double r,double tau)
{
        // sample data
        int nc;
        double *callprice = readArray<double>("callprice.txt", &nc);
        double *callstrike = readArray<double>("callstrike.txt", &nc);
        double *callopenint = readArray<double>("callopenint.txt", &nc);
        //
        int np;
        double *putprice = readArray<double>("putprice.txt", &np);
        double *putstrike = readArray<double>("putstrike.txt", &np);
        double *putopenint = readArray<double>("putopenint.txt", &np);
        //
        int nu;
        double *strike = readArray<double>("strike.txt", &nu);
        //
        // end of sample data
        //
        printf("Number of calls:   %d\n",nc);
        printf("Number of puts:    %d\n",np);
        printf("Number of strikes: %d\n",nu);
        //
    //
    //
    int mRange = nx;
    double *xRange = linspace(xmin,xmax,mRange);
    //
        NPRND *nprnd = new NPRND(callprice, callstrike, callopenint, nc,  //call data
                                 putprice, putstrike, putopenint, np,     //put  data
                                 xRange,mRange,
                                 strike, nu,
                                 r, tau);
        //
        double time_spent = 0.0;
        clock_t begin = clock();
        //
        //
        double *hcv = linspace(hcmin, hcmax, nhc);
        double *hpv = linspace(hpmin, hpmax, nhp);
        //
        double sol[2];
        nprnd->optim_bandwidth(sol,hcv, nhc, hpv, nhp);
        //
        free(hcv);
        free(hpv);
        //
        printf("hc: %.4f\n",sol[0]);
        printf("hp: %.4f\n",sol[1]);
        printf("number of solved QP problems: %lu\n",nprnd->get_nqpprob());
        printf("number of iterations: %lu\n",nprnd->get_niterations());
        //
        //
        //
        clock_t end = clock();
        time_spent += (double)(end - begin) / CLOCKS_PER_SEC;
        printf("The elapsed time is %f seconds\n", time_spent);
        //
        //
        delete nprnd;

        //
        //
        // free memory
        free(callprice);
        free(callstrike);
        free(callopenint);
        //
        free(putprice);
        free(putstrike);
        free(putopenint);
        //
        free(strike);
        free(xRange);
        //
}
```