

[Help](#)

```
#include <iostream>
#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_random.h"
#include "
href../../../../common/math/bsnd_math/svd_bs_tools_h_src.pdfsvd_bs_tools.h"

namespace svd_bs {

LogBS_Model::LogBS_Model()
{
    mu=NULL;
    vol=NULL;
    tmp_x=NULL;
    tmp_y=NULL;
    Rho = NULL;
}

LogBS_Model::LogBS_Model(double r_,double rho_,const PnlVect * vol_, double T, d
:r(r_),Inf_Bndry(Inf_Bndry_),Sup_Bndry(Sup_Bndry_),rho(rho_)
{
    vol= pnl_vect_copy(vol_);
    mu = pnl_vect_create(vol_>size);
    tmp_x = pnl_vect_create(vol_>size);
    tmp_y = pnl_vect_create(vol_>size);
    for(int i = 0; i < vol_>size; i++)
    {
        LET(mu,i) = (r-SQR(GET(vol,i)) * 0.5) * T;
    }
    Rho = pnl_mat_create_from_double(vol->size,vol->size,rho_);
    pnl_mat_set_diag (Rho, 1., 0);
    pnl_mat_chol(Rho);
    det_sigma=1.;
    for(int i = 0; i < vol->size; i++)
    {
        det_sigma *=SQR(GET(vol,i))*MGET(Rho,i,i)*MGET(Rho,i,i)*T;
    }
};
```

```

LogBS_Model::~LogBS_Model()
{
    pnl_vect_free(&mu);
    pnl_vect_free(&tmp_x);
    pnl_vect_free(&tmp_y);
    pnl_vect_free(&vol);
    pnl_mat_free(&Rho);
}

void LogBS_Model::grid_x(PnlVect * pos,const PnlVect * x, double T) const
{
    pnl_vect_resize(pos,x->size);
    for(int i =0 ; i<x->size;i++)
        LET(pos,i)= sqrt(T) * GET(vol,i) * (Inf_Bndry + (Sup_Bndry - Inf_Bndry)
            * GET(x,i))+ GET(mu,i);
}

double LogBS_Model::jacobian(double position,double T) const
{
    double det = 1.;
    for(int i = 0; i < vol->size; i++)
        det *= GET(vol,i)*sqrt(T);
    return det * pow(Sup_Bndry-Inf_Bndry,vol->size);
}

double LogBS_Model::density(const PnlVect * x,const double T) const
{
    int d = x->size;
    double ans;
    pnl_vect_resize(tmp_x,x->size);
    pnl_vect_resize(tmp_y,x->size);
    for(int i =0 ; i<x->size;i++)
        LET(tmp_x,i) = Inf_Bndry + (Sup_Bndry - Inf_Bndry) * GET(x,i);
    pnl_mat_chol_syslin(tmp_y,Rho,tmp_x);
    ans = exp(-0.5 * pnl_vect_scalar_prod(tmp_x,tmp_y)) * exp(- r * T);
    ans *= pow(M_1_SQRT2PI,d) * 1.0/sqrt(fabs(det_sigma));
    //ans *= pow(M_1_SQRT2PI,d) * 1.0/sqrt(fabs(det_sigma)) * sqrt(fabs(det_rho));
    return ans;
}

```

```

void LogBS_Model::print() const
{
    std::cout<<"V = ";pnl_vect_print_nsp(vol);
    std::cout<<"rho = "<<rho<<std::endl;
    std::cout<<"A_Rate = "<<(exp(r)-1)*100<<std::endl;
};

Option::Option()
{
    Spot=pnl_vect_create(0);
}

Option::Option(double Strike_,double T_,PnlVect* spot_,double Inf_Bndry_,double
{
    Spot = pnl_vect_copy(spot_);
}

Option::Option(const Option & m):Strike(m.Strike),T(m.T)
{
    Spot = pnl_vect_copy(m.Spot);
}

Option::~~Option()
{pnl_vect_free(&Spot);}

void Option::print() const
{
    std::cout<<"K = "<<Strike<<std::endl;
    std::cout<<"S = " ;
    pnl_vect_print_nsp(Spot);
}

double Option::payoff(const PnlVect * x,PnlVect* mu, double r) const
{
    double pit=0;
    double vol_i;
    for(int i = 0; i < x->size; i++)
    {
        vol_i = sqrt(2 * (r-GET(mu,i)/T));

```

```

        pit += GET(Spot,i) * exp(GET(mu,i)+sqrt(T) * vol_i * (Inf_Bndry + (Sup_Bnd
    }
    pit=Strike-pit/(x->size);
    return (pit>0) ? pit : 0.;
}

```

```

#define ITER_MAX_SVD 10
#define ITER_MAX_FIXE_POINT 200
#define EPS_FIXE_POINT 1e-3
#define EPS_SVD 1e-3

/*****
/*          Constructor and destructor
*****/
non_linear_approximation::non_linear_approximation()
{
    std::vector<PnlMat *>(0);
    Grid_mat = pnl_mat_create(0,0);
    tmp_pos = pnl_vect_int_create(0);
    tmp_r_step = pnl_vect_create(0);
    tmp_step = pnl_vect_create(0);
    Index = pnl_mat_int_create(0,0);
    Value = pnl_vect_create(0);
    Ms = pnl_mat_create(0,0);
    p_Ms = pnl_vect_create(0);
};
//!
//! Constructor of the non_linear_approximation class
//!
//! @param dim_ is the dimension of the problem
//! @param N_ is the number of points in the discretization at each dimension
//! @param f is a pointer to the function which we want to obtain the SVD
//! @param model_ contains the parameters of the function
//!
non_linear_approximation::non_linear_approximation(int dim_,int N_,double (*f)(c
{
    model = & model_;
    contract = & contract_;
    apply=f;
    //We stock the values of the Grid in a matrix to improve the running

```

```

//time. This Grid is used to compute the points where the function f is
//different from zero. It is used in general_increment function. If we
//consider a situation in a very big dimension (>9) we will have problems to
//stock this matrix
Grid_mat = construct_grid_mat(dim,N,*model,*contract,h,discretization);
//We decided to put the vectors pos,step and real step in the
//non_linear_approximation class because they are called many times in
//the SVD procedure, then we had to allow and to free memory at each
//iteration of SVD procedure and fixed point method and that can be slow
//in terms of running time.
tmp_pos = pnl_vect_int_create(dim - 1);
tmp_step= pnl_vect_create(dim - 1);
tmp_r_step= pnl_vect_create(dim);
Index = pnl_mat_int_create(0,0);
Value = pnl_vect_create(0);
compute_Index_Value();
Ms = pnl_mat_create(dim,N);
p_Ms = pnl_vect_create(Value->size);

};
//!
//! Destructor of the non_linear_approximation class
//!
non_linear_approximation::~non_linear_approximation()
{
    for (non_linear_approximation::iterator it=begin() ; it < end(); it++ )
        pnl_mat_free(&(*it));
    pnl_mat_free(&Grid_mat);
    pnl_vect_int_free(&tmp_pos);
    pnl_vect_free(&tmp_step);
    pnl_vect_free(&tmp_r_step);
    pnl_mat_int_free(&Index);
    pnl_vect_free(&Value);
    pnl_mat_free(&Ms);
    pnl_vect_free(&p_Ms);
};

/*****
/*                               Get and set functions
*****/

```

```

double non_linear_approximation::step_h() const
{ return h;}

//!
//! This function gives the evaluation in the Grid of the point in the direction
//!
//! @param i is the direction where the point is
//! @param p is the position where the point is. More precisely, the position is
//! @return MGET(Grid_mat,i,p) the evaluation in the Grid
//!
double non_linear_approximation::Grid(int i, int p) const
{ return MGET(Grid_mat,i,p);}

//!
//! This function allows to add the solution of one SVD iteration to the final s
//! @param M is the solution of one SVD iteration.
//!
void non_linear_approximation::add_element(const PnlMat * M)
{
    push_back(pnl_mat_copy(M));
};

/*****
/*          Pre calculation for itertiv fix point
*****/

//!
//! Compute M R(n) where M is a tridiag matrix (mass matrix)
//! and R(n) the svd decomposition in dimension n
//! store the result in Ms(n)
//!
//! @param M a pointer to the mass matrix
//!
void non_linear_approximation::update_Ms(PnlMat* R, int n, const PnlTridiagMat
{
    PnlVect V_R = pnl_vect_wrap_mat_row(R,n);
    PnlVect V_Ms = pnl_vect_wrap_mat_row(Ms,n);
#ifdef METHODE_ID
    pnl_vect_axpby(step_h(),&V_R,0.,&V_Ms);
#else
    pnl_tridiag_mat_mult_vect_inplace(&V_Ms,M,&V_R);

```

```

#endif
}

//!
//! Do update_Ms in each dimension
//!
//! @param M a pointer to the mass matrix
//!
void non_linear_approximation :: Ms_init(PnlMat* R, const PnlTridiagMat *M)
{
    for(int i = 0; i < dim; i++)
        update_Ms(R,i,M);
}

//!
//! Precompute the product term in the rhs of the svd method
//!
//! @param n direction on which we compute the rhs
//!
void non_linear_approximation::compute_rhs_product_term_in_dim(int n)
{
    double pit;
    for(int count = 0; count< Value->size; count++)
    {
        pit = 1;
        for(int i = 0; i < dim; i++)
            if(i != n)
            {
                pit *= MGET(Ms,i,pnl_mat_int_get(Index,count,i));
            }
        LET(p_Ms,count) = pit;
    }
}

/*****
/*                               Pre calculation for support of function
*****/

//New modifications because we want to do the update function in the fixed proce

```

```

    ///!
    ///! This function creates the matrix that stock the value of the function in the
    ///!
    ///! @param Funct_Vect is the row of the matrix that we will create
    ///! @param n is the number of the row that we will create
    ///!
    void non_linear_approximation::compute_Index_Value()
    {
        int size_f;
        PnlVectInt* pos = pnl_vect_int_create(dim);
        PnlVect* step = pnl_vect_create(dim);
        PnlVect* r_step = pnl_vect_create(dim);
        int count = 0;
        general_initialization(pos,1,step,Grid_mat,pos->size + 1);
        do
        {
            count++;
        }while(general_increment(pos,step,1,Grid_mat,pos->size + 1,N));
        size_f = count;
        if(pnl_mat_int_resize(Index,size_f,dim) == FAIL || pnl_vect_resize(Value,size_f))
        {
            std::cout<<"Memory Allocation Problem "<<std::endl;
            abort();
        }

        count =0;
        general_initialization(pos,1,step,Grid_mat,pos->size + 1);
        do
        {
            for(int i = 0; i < dim; i++)
            {
                pnl_mat_int_set(Index,count,i, pnl_vect_int_get(pos,i));
                LET(r_step,i) = pnl_vect_int_get(pos,i)*step_h();
            }
            LET(Value,count) = apply(r_step,*model,*contract);
            count++;
        }while(general_increment(pos,step,1,Grid_mat,pos->size + 1,N));
        pnl_vect_int_free(&pos);
        pnl_vect_free(&step);
        pnl_vect_free(&r_step);
    }

```



```

/*****
/*
step of the fix point method
*****/
//!
//! New!!!!
//! This function does the fixed point procedure
//!
// The Ri vector is the row i of the matrix R and aux2 is the row i of a matrix
// Compute lhs that is the left side of the Euler - Lagrange equation, in other
void non_linear_approximation::update(PnlMat * R, int n,const PnlTridiagMat * M,
{
    PnlVect Ri,aux2;
    PnlTridiagMatLU *M_lu;
    double lhs = 1.;
    for(int i = 0; i < dim; i++)
    {
        if(i != n)
        {
            Ri=pnl_vect_wrap_mat_row(R,i);
#ifdef METHODE_ID
            //Mass lumping
            lhs *= pnl_vect_scalar_prod(&Ri,&Ri)*step_h();
#else
            //We obtain this integral using a tridiagonal mass matrix
            lhs *= pnl_tridiag_mat_scalar_prod(M,&Ri,&Ri);
#endif
        }
    }

#ifdef METHODE_ID

#else
    double inc;
#endif
    double pit = 1, sum = 0;
    pnl_vect_set_double(rhs,0);
    for(int count = 0; count < Value->size; count++)
    {
#ifdef METHODE_ID
        LET(rhs,pnl_mat_int_get(Index,count,n)) += GET(p_Ms,count)*GET(Value,count

```

```

#else
    for(int j=((pnl_mat_int_get(Index,count,n)-1>-1)?-1:0);j<=((pnl_mat_int_ge
        {
            LET(rhs,pnl_mat_int_get(Index,count,n)+j) += GET(p_Ms,count)*GET(Value
                pnl_tridiag_mat_get(M,pnl_mat_int_get(Index,count,n),j);
        }
#endif
    }
    // Update of the term f_n
    // At the n SVD iteration, we have to use the SVD solution that is a sum of n-
    for(int k = 0; k < N; k++)
    {
        //The variable called it scans the tensor products obtained as solutions
        //in the previuos iterations. The variable called it is a matrix because
        //each SVD solution is represented by a matrix.
        sum = 0;
        for (non_linear_approximation::const_iterator it=begin() ; it < end(); it+
            {
                pit = 1;
                for(int i = 0; i < dim; i++)
                {
                    if(i != n)
                    {
                        aux2 = pnl_vect_wrap_mat_row(*it,i);
                        Ri=pnl_vect_wrap_mat_row(R,i);
#ifdef METHODE_ID
                        //Mass lumping
                        pit *= pnl_vect_scalar_prod(&aux2,&Ri)*step_h();
#else
                        //We have decomposed the function f in the base of the P1 func
                        pit *= pnl_tridiag_mat_scalar_prod(M,&aux2,&Ri);
#endif
                    }
                }
            }
#ifdef METHODE_ID
            pit *= MGET(*it,n,k);
#else
            inc = MGET(*it,n,k)*pnl_tridiag_mat_get(M,k,0);
            inc+=(k > 0)?MGET(*it,n,k-1)*pnl_tridiag_mat_get(M,k,- 1):0.;
            inc+=(k < N-1)?MGET(*it,n,k+1)*pnl_tridiag_mat_get(M,k,1):0.;

```

```

        pit*=inc;
#endif
        sum += pit;
    }
    LET(rhs,k)= LET(rhs,k) - sum;
}

//time(&fin);
Ri=pnl_vect_wrap_mat_row(R,n);

#ifdef METHODE_ID
    pnl_vect_axpby(1./lhs,rhs,0,&Ri);
#else
    //We divide by the left side of the equation because it is a real number
    pnl_vect_mult_double(rhs,1.0/lhs);
    M_lu = pnl_tridiag_mat_lu_new ();
    pnl_tridiag_mat_lu_compute (M_lu, M);
    pnl_tridiag_mat_lu_syslin(&Ri,M_lu,rhs);
    pnl_tridiag_mat_lu_free (&M_lu);
#endif

    //pnl_vect_print(&Ri);
}

/*****
/*          Main part of the SVD method
*****/

//!
//! Manage the fixed point method and the svd iterations
//!
//! @param M a pointer to the mass matrix
//!
void non_linear_approximation ::svd_decomposition(const PnlTridiagMat *M )
{
    PnlMat * R =  pnl_mat_create(dim,N);
    int n;
    int fix_point_iter,svd_iter = 0;
    double e_l2 = 0;//error L2 formule integrale
    double e_frob = 0;// error norme Frobenius

```

```

double integral_f_2;
double somme_f_2;
PnlVect * rhs = pnl_vect_create(N);
//For the random intial condition, we take randomly a point in [a,b]^dim
PnlVect* a = pnl_vect_create_from_double(dim,-1);
PnlVect* b = pnl_vect_create_from_double(dim,1);
//We stock the value of the L2 norm and Frobenius norm of f to obtain later a
integral_f_2 = e_l2;
somme_f_2 = e_frob;
//We deal with the case where dimension is equal to 1 separately.
// We know that the solution in this case is the evaluation of the function in
if(dim == 1)
{
    PnlVect* step = pnl_vect_create(1);
    for(int i = 0; i < N; i++)
    {
        //Basket put option A CHECKER!
        //pnl_vect_set(step,0,-model->dynamic->Bndry+i*h*2*model->dynamic->Bnd
        //Put payoff
        pnl_vect_set(step,0,i*h);
        pnl_mat_set(R,0,i,apply(step,*model,*contract));
    }
    pnl_vect_free(&step);
    e_l2 += 0.;//error_l2(R,M);
    std::cout<<"e_l2"<<" "<<sqrt(fabs(e_l2/integral_f_2))<<std::endl;
    e_frob += 0.;//error_frob(R);
    std::cout<<"e_frob"<<" "<<sqrt(fabs(e_frob/somme_f_2))<<std::endl;
    //pnl_mat_print(R);
    add_element(R);
}
else {
    do /* non linear approximation loop */
    {
        //Point Fixed procedure
        //We have to give an initial condition to begin the iterations

        //pnl_mat_rand_uni(R,dim,N,a,b,PNL_RNG_MERSENNE);

        pnl_mat_set_double(R,1.);

        Ms_init(R,M);
    }
}

```

```

        //Update parameters because they are used at each iteration of the Greedy
        fix_point_iter = 0;
        n = 0;
        //std::cout<<" start fix point procedure"<<std::endl;
        do /* fix point loop */
        {
            //Update row n of the matrix R
            compute_rhs_product_term_in_dim(n);
            update(R,n,M,rhs);
            update_Ms(R,n,M);
            n = (n + 1) % dim;

            //The following criterion allows us to study if at each iteration
            //of the fixed point procedure, with a n fixed, the error made is
            //below the constant EPS_FIXE_POINT
            fix_point_iter++;
        }while(fix_point_iter/dim < ITER_MAX_FIXE_POINT);
        //Linking the solution matrix
        add_element(R);
        svd_iter++;
    }
    while((svd_iter < ITER_MAX_SVD));
}

//DELETE
pnl_mat_free(&R);
pnl_vect_free(&rhs);
pnl_vect_free(&a);
pnl_vect_free(&b);
}

static void inc_step(PnlVect* step,const PnlMat * Grid_mat, int n,int d,int i);

static int general_increment_indent(PnlVectInt* pos, int d_,const double & go_out)

//!
//! This function initialize the vector pos and step, knowing the direction n the
//! @param n is the direction fixed
//! @param go_out is the quantity that the somme of entries of the step vector c
//!

```

```

int general_initialization(PnlVectInt* pos,
                           const double & go_out,
                           PnlVect* step,
                           PnlMat* Grid_mat,
                           int n)
{
    for(int i = 0; i < pos->size; i++)
    {
        pnl_vect_int_set(pos,i,0);
        inc_step(step,Grid_mat,n,i,0);
    }
    if(pnl_vect_sum(step) < go_out)
        return 1;
    return 0;
}

//!
//! This function calculates the points where the function is different from zero
//! @param n is the direction fixed
//! @param go_out is the quantity that the somme of entries of the step vector c
//! @param d_ allows to look in the other coordiantes of the step vector in a re
//!
int general_increment(PnlVectInt* pos, PnlVect* step,const double & go_out,PnlMa
{ return general_increment_indent(pos,pos->size-1,go_out,step,Grid_mat,n,N);}

//!
//! This function calculates the points where the function is different from zero
//! @param n is the direction fixed
//! @param go_out is the quantity that the somme of entries of the step vector c
//! @param d_ allows to look in the other coordiantes of the step vector in a re
//!
int general_increment_indent(PnlVectInt* pos, int d_,const double & go_out, PnlV
{
    //We suppose that we begin in the point (0,0,...,0) in the function called gen
    if(d_ < 0)
        return 0;
    //A valid position in the pos vector is always between 0 and N-1
    if(pnl_vect_int_get(pos,d_)<N-1)

```

```

{
    //We can move to the following position
    pnl_vect_int_set(pos,d_,pnl_vect_int_get(pos,d_) + 1);
    inc_step(step,Grid_mat,n,d_,pnl_vect_int_get(pos,d_));
    for(int i = d_ + 1; i < pos->size; i++)
    {
        //When we look the positions that are greater than d_, they begin with
        if(pnl_vect_int_get(pos,i)!=0)
        {
            pnl_vect_int_set(pos,i,0);
            inc_step(step,Grid_mat,n,i,0);
        }
    }
    //We check that the position is really valid
    if(pnl_vect_sum(step) < go_out)
        return 1;
    //If it is not valid, we change the coordinate (We use the fact that the f
    else
        return general_increment_indent(pos,d_-1,go_out,step,Grid_mat,n,N);
}
else
    return general_increment_indent(pos,d_-1,go_out,step,Grid_mat,n,N);
}

//!
//! This function allows us give the correct value of the Grid to the vector ste
//! @param n is the direction fixed
//! @param d is the dimension of the problem
//! @param i is the position in the step vector
//!
void inc_step(PnlVect* step,const PnlMat * Grid_mat, int n,int d,int i)
{
    if(d < n)
        pnl_vect_set(step,d,MGET(Grid_mat,d,i));
    else
        pnl_vect_set(step,d,MGET(Grid_mat,d+1,i));
}

```

```

//!
//! This function calculate the integral in [0,1] of the SVD of the function
//!
//! @return sum that is the value of the integral in [0,1] of the SVD of the fun
//!
double non_linear_approximation::svd_integral() const
{
    PnlVect aux2;
    double pit = 1., sum = 0., sum2 = 0.;
    for (std::vector<PnlMat*>::const_iterator it=begin() ; it < end(); it++ )
    {
        pit = 1.;
        for(int i = 0; i < dim; i++)
        {
            //Each row is the solution in the direction i.
            aux2 = pnl_vect_wrap_mat_row(*it,i);
            sum2 = 0.;
            for(int k = 0; k < N; k++)
            {
                //We know that the integral of a P1 function is h except if this f
                if(k == 0 || k == N-1)
                    sum2 += pnl_vect_get(&aux2,k)*0.5*h;
                else
                    sum2 += pnl_vect_get(&aux2,k)*h;
            }
            pit *= sum2;
        }
        //We have to add the jacobien respective to the variable changes.
        sum += pit;
    }
    //std::cout<<"The value of the integral of the svd is "<<sum<<std::endl;
    return sum*model->jacobian(0,contract->T);
}

//!
//! This function evaluates the SVD of the function in the point x, where x is a
//!
//! @param x is the vector where we want to evaluate the SVD of the function
//! @return sum that is the evaluation of the SVD of the function in the point x
//!

```



```

double non_linear_approximation::svd_evaluation(PnlVect * x) const
{
    PnlVect aux;
    //J stock the index of the functions that are different from zero in the point
    PnlVectInt * J = pnl_vect_int_create(dim);
    //phi_L evaluates the functions that are different from zero in the point x
    PnlVect * phi_L = pnl_vect_create(dim);
    //phi_R corresponds to the functions that are different from zero in the point
    PnlVect * phi_R = pnl_vect_create(dim);
    double pit = 1, sum = 0;

    for(int i = 0; i < dim; i++)
    {
        pnl_vect_int_set(J,i,floor(GET(x,i)*(N-1)));
        pnl_vect_set(phi_L,i,hat_function(pnl_vect_int_get(J,i),GET(x,i),N));
        pnl_vect_set(phi_R,i,1 - GET(phi_L,i));
    }

    for (std::vector<PnlMat*>::const_iterator it=begin() ; it < end(); it++ )
    {
        pit = 1;
        for(int i = 0; i < dim; i++)
        {
            aux = pnl_vect_wrap_mat_row(*it,i);
            if(pnl_vect_int_get(J,i) != N-1)
                pit *= (GET(&aux,pnl_vect_int_get(J,i)) * GET(phi_L,i) + GET(&aux,pnl_vect_int_get(J,i))) * GET(phi_R,i);
            else
                pit *= GET(&aux,pnl_vect_int_get(J,i)) * GET(phi_L,i);
        }
        sum += pit;
    }
    pnl_vect_free(&phi_L);
    pnl_vect_free(&phi_R);
    pnl_vect_int_free(&J);
    return sum;
}

/*****
/*                                Print Functions
*****/

```

```

    ///!
    ///! Print the svd solution obtained at each iteration
    ///!
    ///! At each iteration the solution is a dim x N matrix
    ///!
    void non_linear_approximation::print_svd()
    {
        for (std::vector<PnlMat*>::const_iterator it=begin() ; it < end(); it++ )
            pnl_mat_print_nsp(*it);
    }

    ///!
    ///! Mass matrix definition
    ///!
    PnlTridiagMat * initialise_mass_matrix(int N)
    {
        PnlTridiagMat * M;
        double h = 1./((N-1)*6);
        M =pnl_tridiag_mat_create_from_two_double(N,4.*h, h);
        M->D[0]*=0.5;
        M->D[N-1]*=0.5;
        return M;
    }

    ///!
    ///! Integral of the P1 functions
    ///! @param i=0..N-1 is the number of the P1 function
    ///! @param N is the number of points of discretization
    ///!
    double integral(int i,int N)
    {
        if(i == 0 || i == N-1)
            return 1./(2 * (N-1)) ;
        else
            return 1./(N-1) ;
    }

    ///!
    ///! P1 function definition
    ///! @param i is the index of the P1 function, i=0...N-1
    ///! @param N is the number of discretization points

```

```

//!
double hat_function(int i, double x, int N)
{
    double h = 1./(N-1);
    double y = (x - i * h)/h;
    if(fabs(y) <= 1)
        return 1 - fabs(y);
    else
        return 0;
}

void Grid_S(PnlVect* pos, PnlVect* step, const LogBS_Model &model, const Option &contract)
{
    model.grid_x(pos, step, contract.T);
    pnl_vect_mult_double(pos, 1./(contract.Strike * step->size * 1.));
}

void Grid_logS(PnlVect* pos, PnlVect* step, const LogBS_Model &model, const Option &contract)
{
    model.grid_x(pos, step, contract.T);
    for(int i = 0; i < pos->size; i++)
    {
        LET(pos, i) = GET(contract.Spot, i)/(contract.Strike * pos->size * 1.) * exp(-contract.r * step->size * 1.);
    }
}

PnlMat * construct_grid_mat(int dim, int N, const LogBS_Model &model, const Option &contract)
{
    PnlVect * Step = pnl_vect_create(dim);
    PnlMat * Grid_mat = pnl_mat_create(dim, N);
    PnlVect * pos = pnl_vect_create(dim);
    if(discretization == 0)
    {
        for(int j = 0; j < N; j++)
        {
            pnl_vect_set_double(Step, j*h);
            Grid_S(pos, Step, model, contract);
            for(int i = 0; i < dim; i++)
                MLET(Grid_mat, i, j) = GET(pos, i);
        }
    }
}

```

```

    }
else
{
    for(int j = 0; j < N; j++)
    {
        pnl_vect_set_double(Step,j*h);
        Grid_logS(pos,Step,model,contract);
        for(int i = 0; i < dim; i++)
            MLET(Grid_mat,i,j) = GET(pos,i);
    }
}
pnl_vect_free(&pos);
pnl_vect_free(&Step);
return Grid_mat;
}

};

```