

[Help](#)

```
#include <stdio.h>
#include <stdlib.h>

#include "pnl/pnl_random.h"
#include "pnl/pnl_matrix.h"
#include "pnl/pnl_vector.h"
#include "pnl/pnl_mathtools.h"
#include "
href../../common/math/equity_pricer/carr_h_src.pdfcarr.h"
#include "
href../../common/math/equity_pricer/levy_process_h_src.pdflevy_process.h"
#include "
href../../common/math/equity_pricer/finance_tool_box_h_src.pdffinance_tool_bo
#include "
href../../common/math/equity_pricer/levy_calibration_h_src.pdflevy_calibratio
#include "
href../../common/math/equity_pricer/levy_diffusion_calibration_h_src.pdflevy_

//double s, double strike_min, double strike_max, int nbr_strike, double Tmin, d
int GeneratePrices_ForLevyDiffusion(Calibration_Data *data,
                                   Levy_diffusion *Levy)
{
    int j, k, last; //,error;
    //PnlVect strike_vector,price_vector;
    List_Option_Eqd *op = data->list_model;
    Option_Eqd op_ref = list_option_eqd_get_value(op, 0, 0);
    for (j = 0; j < op->nb_maturity; j++)
    {
        k = pnl_vect_int_get(op->index_maturity, j);
        last = (j < op->nb_maturity - 1) ? pnl_vect_int_get(op->index_maturity, j
        /*
        //>> Should be the method used
        //>> but need to work more for out-money options
        //>> need FFT in other function.
        strike_vector=pnl_vect_wrap_subvect(op->K,k,last-k);
        price_vector=pnl_vect_wrap_subvect(op->price,k,last-k);
```

```

error=CarrMethod_onStrikeList(&strike_vector,
                               &price_vector,
                               op->S0,
                               GET(op->T,j),
                               abs(pnl_vect_int_get(op->product,k)-2),
                               data->rate,
                               data->divid,
                               0.01,
                               Levy);

*/
//>> For test, consider slowly algorithm :
op_ref.T = GET(op->T, j);
while (k < last)
{
    op_ref.K = GET(op->K, k);
    CarrMethod_Vanilla_option_LD(&op_ref, 0.01, Levy);
    LET(op->price, k) = op_ref.price;
    k++;
}

}
return 1;
}

// ----- Bates_diffusion -----

void HestonConstraints(PnlVect *res, const PnlVect *x, void *params)
{

}

double QuadraticError_ForHeston(const PnlVect *GenerationParams,
                                void *data)
{
    double jump_drift;
    int error;
    Heston_diffusion *Process = Heston_diffusion_create(GET(GenerationParams, 2),
                                                         GET(GenerationParams, 1),
                                                         GET(GenerationParams, 4),

```

```

        GET(GenerationParams, 3),
        GET(GenerationParams, 0), &jump_drift);
    Levy_diffusion *Levy = Levy_diffusion_create(Process, &Heston_diffusion_characteristics,
    error = GeneratePrices_ForLevyDiffusion((Calibration_Data *)data, Levy);
    free(Levy);
    free(Process);
    return QuadraticError(data);
}

// ----- Bates_diffusion -----

void BatesConstraints(PnlVect *res, const PnlVect *x, void *params)
{

}

double QuadraticError_ForBates(const PnlVect *GenerationParams,
                               void *data)
{
    double jump_drift;
    int error;
    Bates_diffusion *Process = Bates_diffusion_create(GET(GenerationParams, 2),
    GET(GenerationParams, 1),
    GET(GenerationParams, 4),
    GET(GenerationParams, 3),
    GET(GenerationParams, 0),
    GET(GenerationParams, 5),
    GET(GenerationParams, 6),
    GET(GenerationParams, 7),
    &jump_drift);
    Levy_diffusion *Levy = Levy_diffusion_create(Process, &Bates_diffusion_characteristics,
    error = GeneratePrices_ForLevyDiffusion((Calibration_Data *)data, Levy);
    free(Levy);
    free(Process);
    return QuadraticError(data);
}

```

```
// ----- BNS_diffusion -----
```

```
void BNSConstraints(PnlVect *res, const PnlVect *x, void *params)
{
    double sigma0_min, ka_min, eta_min, theta_min, rhow_min;
    double sigma0_max, ka_max, eta_max, theta_max, rhow_max;

    sigma0_min = 0;
    ka_min = 0;
    eta_min = 0;
    theta_min = 0;
    rhow_min = -0.9;
    sigma0_max = 1;
    ka_max = 5;
    eta_max = 5;
    theta_max = 1;
    rhow_max = 0.9;

    pnl_vect_resize(res, 11);

    LET(res, 0) = sigma0_max - GET(x, 0);
    LET(res, 1) = -sigma0_min + GET(x, 0);
    LET(res, 2) = ka_max - GET(x, 1);
    LET(res, 3) = -ka_min + GET(x, 1);
    LET(res, 4) = eta_max - GET(x, 2);
    LET(res, 5) = -eta_min + GET(x, 2);
    LET(res, 6) = theta_max - GET(x, 3);
    LET(res, 7) = -theta_min + GET(x, 3);
    LET(res, 8) = rhow_max - GET(x, 4);
    LET(res, 9) = -rhow_min + GET(x, 4);
    LET(res, 10) = 2 * GET(x, 1) * GET(x, 2) - GET(x, 3) * GET(x, 3);
    // Condition de Feller.
}
```

```
double QuadraticError_ForBNS(const PnlVect *GenerationParams, void *data)
```

```

{
    double jump_drift;
    int error;
    BNS_diffusion *Process = BNS_diffusion_create(GET(GenerationParams, 2),
                                                    GET(GenerationParams, 1),
                                                    GET(GenerationParams, 4),
                                                    GET(GenerationParams, 3),
                                                    GET(GenerationParams, 0), &jump_drift);
    Levy_diffusion *Levy = Levy_diffusion_create(Process, &BNS_diffusion_character
    error = GeneratePrices_ForLevyDiffusion((Calibration_Data *)data, Levy);
    free(Levy);
    free(Process);
    return QuadraticError(data);
}

```

// ----- NIGGammaOU_diffusion -----

```

void NIGGammaOUConstraints(PnlVect *res, const PnlVect *x, void *params)
{

}

```

```

double QuadraticError_ForNIGGammaOU(const PnlVect *GenerationParams, void *data)
{
    double jump_drift;
    int error;
    double GammaOU_Alpha = GET(GenerationParams, 0);
    double GammaOU_Beta = GET(GenerationParams, 1);
    double GammaOU_Delta = GET(GenerationParams, 2);
    double OU_Lambda = GET(GenerationParams, 3);
    double OU_Alpha = GET(GenerationParams, 4);
    double OU_Beta = GET(GenerationParams, 5);
    double y0 = GET(GenerationParams, 6);
    NIG_process *Process = NIG_process_create(GammaOU_Alpha, GammaOU_Beta, GammaOU
    GammaOU_diffusion *Time_Clock_Levy = GammaOU_diffusion_create(OU_Lambda, OU_Al

```

```

OU_Beta, y0,
Process, &NIG_process_characteristic_exp
Levy_diffusion *Levy = Levy_diffusion_create(Time_Clock_Levy, &GammaOU_diffusi
NIG_process_kill_drift(Process);

error = GeneratePrices_ForLevyDiffusion((Calibration_Data *)data, Levy);
free(Levy);
free(Process);
return QuadraticError(data);
}

```