

[Help](#)

```
#include "variancegamma1d_pad.h"
#include "enums.h"
#include "pnl/pnl_cdf.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_specfun.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2010+2) //The "#els
static int CHK_OPT(MC_VarianceGamma_Floating)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(MC_VarianceGamma_Floating)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else
//Compute the positive or negative jump size between the smallest and the bigges
static double jump_generator_VG(double *cdf_jump_vect, double *cdf_jump_points,
{
    double z, v, y;
    int test, temp, l, j, q;
    test = 0;
    v = pnl_rand_uni(generator);
    y = cdf_jump_vect[cdf_jump_vect_size] * v;
    l = cdf_jump_vect_size / 2;
    j = cdf_jump_vect_size;
    z = 0;
    if (cdf_jump_vect[l] > y)
    {
        l = 0;
        j = cdf_jump_vect_size / 2;
    }
    if (v == 1)
    {
        z = cdf_jump_points[cdf_jump_vect_size];
    }
    if (v == 0)
    {
        z = cdf_jump_points[0];
    }
}
```

```

    }
    if (v != 1 && v != 0)
    {
        while (test == 0)
        {
            if (cdf_jump_vect[l + 1] > y)
            {
                q = l;
                test = 1;
            }
            else
            {
                temp = (j - l - 1) / 2 + 1;
                if (cdf_jump_vect[temp] > y)
                {
                    j = temp;
                    l = l + 1;
                }
                else
                {
                    l = temp * (temp > 1) + (l + 1) * (temp <= 1);
                }
            }
        }
        z = cdf_jump_points[q] * exp((y - cdf_jump_vect[q]) * exp(M_G * cdf_jump_p
    }
    return z;
}

static int VG_Mc_Floating(double s_maxmin, NumFunc_2 *P, double S0, double T, do
{
    double payoff, s, s1, sup, inf, eps, err;
    double *Xg, *Xd, *jump_time_vect_p, *jump_time_vect_m, proba, lambda_p, lambda
    double cdf_jump_bound, drift, control, s2, s3, s4, s5, s6, u, u0, w1, w2, z, C
    double control_expec, cov_payoff_control, var_payoff, var_control, cor_payoff_
    double control_coef, tau, *v1, *v2, pas, *cdf_jump_points, *cdf_jump_vect_p, *
    double min_M_G, var_proba, infS, supS;
    int i, j, jump_number, jump_number_p, jump_number_m, cdf_jump_vect_size, m1, m
    G = sqrt(2 / kappa + theta * theta / (sigma * sigma)) / sigma + theta / (sigma
    M = sqrt(2 / kappa + theta * theta / (sigma * sigma)) / sigma - theta / (sigma
    C = 1 / kappa;
    control_expec = exp((r - divid) * T) * S0;

```

```

err = 1e-16;
eps = 1e-3;
cdf_jump_vect_size = 100000;
s = 0;
s1 = 0;
s2 = 0;
s3 = 0;
s4 = 0;
s5 = 0;
s6 = 0;
lambda_p = 0;
lambda_m = 0;
proba = 0;
////////////////////////////////////
lambda_p = C * pnl_sf_gamma_inc(0., eps * M); //positive jump intensity
while (lambda_p * T < 20)
{
    eps = eps * 0.9;
    lambda_p = C * pnl_sf_gamma_inc(0., eps * M);
}
lambda_m = C * pnl_sf_gamma_inc(0., eps * G); //negative jump intensity
while (lambda_m * T < 20)
{
    eps = eps * 0.9;
    lambda_m = C * pnl_sf_gamma_inc(0., eps * G);
}
lambda_p = C * pnl_sf_gamma_inc(0., eps * M);
drift = (r - divid) + log(1 - (theta + sigma * sigma / 2) * kappa) / kappa + t
////////////////////////////////////
m1 = (int)(10 * lambda_p * T);
m2 = (int)(10 * lambda_m * T);
v1 = malloc((m1) * sizeof(double));
v1[0] = 0;
v2 = malloc((m2) * sizeof(double));
v2[0] = 0;
cdf_jump_bound = 5;
min_M_G = MIN(M, G);
//Computation of the biggest jump that we tolerate
while (C * exp(-min_M_G * cdf_jump_bound) / (min_M_G * cdf_jump_bound) > err)
    cdf_jump_bound++;
pas = (cdf_jump_bound - eps) / cdf_jump_vect_size;

```

```

cdf_jump_points = malloc((cdf_jump_vect_size + 1) * sizeof(double));
cdf_jump_vect_p = malloc((cdf_jump_vect_size + 1) * sizeof(double));
cdf_jump_vect_m = malloc((cdf_jump_vect_size + 1) * sizeof(double));
cdf_jump_points[0] = eps;
cdf_jump_vect_p[0] = 0;
cdf_jump_vect_m[0] = 0;
//computation of the cdf of the positive and negative jumps at some points
for (i = 1; i <= cdf_jump_vect_size; i++)
{
    cdf_jump_points[i] = i * pas + eps;
    cdf_jump_vect_p[i] = cdf_jump_vect_p[i - 1] + exp(-M * cdf_jump_points[i - 1]);
    cdf_jump_vect_m[i] = cdf_jump_vect_m[i - 1] + exp(-G * cdf_jump_points[i - 1]);
}
////////////////////////////////////
pnl_rand_init(generator, 1, n_paths);
//Call options case
if ((P->Compute) == &Call_StrikeSpot2)
{
    for (i = 0; i < n_paths; i++)
    {
        //simulation of the positive jump times and number
        tau = -1 / (lambda_p) * log(pnl_rand_uni(generator));
        jump_number_p = 0;
        while (tau < T)
        {
            jump_number_p++;
            v1[jump_number_p] = tau;
            tau += -1 / (lambda_p) * log(pnl_rand_uni(generator));
        }
        jump_time_vect_p = malloc((jump_number_p + 2) * sizeof(double));
        jump_time_vect_p[0] = 0;
        for (j = 1; j <= jump_number_p; j++)
            jump_time_vect_p[j] = v1[j];
        jump_time_vect_p[jump_number_p + 1] = T;
        //simulation of the negative jump times and number
        tau = -1 / (lambda_m) * log(pnl_rand_uni(generator));
        jump_number_m = 0;
        while (tau < T)
        {
            jump_number_m++;
            v2[jump_number_m] = tau;

```

```

        tau += -1 / (lambda_m) * log(pnl_rand_uni(generator));
    }
    jump_time_vect_m = malloc((jump_number_m + 2) * sizeof(double));
    jump_time_vect_m[0] = 0;
    for (j = 1; j <= jump_number_m; j++)
        jump_time_vect_m[j] = v2[j];
    jump_time_vect_m[jump_number_m + 1] = T;
    jump_number = jump_number_p + jump_number_m; //total jump number
    //////////////////////////////////////
    Xg = malloc((jump_number + 2) * sizeof(double)); //left value of X at
    Xg[0] = 0;
    Xd = malloc((jump_number + 2) * sizeof(double)); //right value of X at
    Xd[0] = 0;
    k1 = 1;
    k2 = 1;
    u0 = 0;
    //computation of Xg and Xd
    for (k = 1; k <= jump_number; k++)
    {
        w1 = jump_time_vect_p[k1];
        w2 = jump_time_vect_m[k2];
        if (w1 < w2)
        {
            u = w1;
            k1++;
            z = jump_generator_VG(cdf_jump_vect_p, cdf_jump_points, cdf_ju
        }
        else
        {
            u = w2;
            k2++;
            z = -jump_generator_VG(cdf_jump_vect_m, cdf_jump_points, cdf_j
        }
        Xg[k] = drift * (u - u0) + Xd[k - 1];
        Xd[k] = Xg[k] + z;
        u0 = u;
    }
    Xg[jump_number + 1] = drift * (T - u0) + Xd[jump_number];
    Xd[jump_number + 1] = Xg[jump_number + 1];
    //////////////////////////////////////
    //computation of the supremum and the infimum of the Levy path

```

```

inf = 0;
sup = 0;
for (j = 1; j <= jump_number; j++)
{
    if (drift > 0)
    {
        if (inf > Xd[j])
            inf = Xd[j];
        if (sup < Xg[j])
            sup = Xg[j];
    }
    else
    {
        if (inf > Xg[j])
            inf = Xg[j];
        if (sup < Xd[j])
            sup = Xd[j];
    }
}
infS = S0 * exp(inf);
if (infS > s_maxmin)
{
    infS = s_maxmin;
    proba = 1;
}
payoff = infS;
infS = S0 * exp(Xd[jump_number + 1] - sup); //antithetic variate assoc
if (infS > s_maxmin)
{
    infS = s_maxmin;
    proba += 1;
}
payoff = (payoff + infS) / 2;
proba /= 2;
s1 += payoff;
s += payoff * payoff;
control = S0 * exp(Xd[jump_number + 1]);
s2 += control;
s3 += control * control;
s4 += control * payoff;
s5 += proba;

```

```

        s6 += proba * proba;
        free(Xd);
        free(Xg);
        free(jump_time_vect_p);
        free(jump_time_vect_m);
    }
    cov_payoff_control = s4 / n_paths - s1 * s2 / ((double)n_paths * n_paths);
    var_payoff = (s - s1 * s1 / ((double)n_paths)) / (n_paths - 1);
    var_control = (s3 - s2 * s2 / ((double)n_paths)) / (n_paths - 1);
    cor_payoff_control = cov_payoff_control / (sqrt(var_payoff) * sqrt(var_control));
    control_coef = cov_payoff_control / var_control;
    var_proba = (s6 - s5 * s5 / ((double)n_paths)) / (n_paths - 1);
    *ptprice = exp(-divid * T) * S0 - (exp(-r * T) * s1 / n_paths - control_coef * var_payoff);
    *errorprice = 1.96 * sqrt(var_payoff * (1 - cor_payoff_control * cor_payoff));
    *ptdelta = (*ptprice + exp(-r * T) * s_maxmin * s5 / (n_paths)) / S0;
    *errordelta = (*errorprice + 1.96 * exp(-r * T) * s_maxmin * sqrt(var_payoff));
}
else//Put
if ((P->Compute) == &Put_StrikeSpot2)
{
    for (i = 0; i < n_paths; i++)
    {
        //simulation of the positive jump times and number
        tau = -1 / (lambda_p) * log(pnl_rand_uni(generator));
        jump_number_p = 0;
        while (tau < T)
        {
            jump_number_p++;
            v1[jump_number_p] = tau;
            tau += -1 / (lambda_p) * log(pnl_rand_uni(generator));
        }
        jump_time_vect_p = malloc((jump_number_p + 2) * sizeof(double));
        jump_time_vect_p[0] = 0;
        for (j = 1; j <= jump_number_p; j++)
            jump_time_vect_p[j] = v1[j];
        jump_time_vect_p[jump_number_p + 1] = T;
        //simulation of the negative jump times and number
        tau = -1 / (lambda_m) * log(pnl_rand_uni(generator));
        jump_number_m = 0;
        while (tau < T)
        {

```

```

        jump_number_m++;
        v2[jump_number_m] = tau;
        tau += -1 / (lambda_m) * log(pnl_rand_uni(generator));
    }
    jump_time_vect_m = malloc((jump_number_m + 2) * sizeof(double));
    //simulation of the negative jump times and number
    jump_time_vect_m[0] = 0;
    for (j = 1; j <= jump_number_m; j++)
        jump_time_vect_m[j] = v2[j];
    jump_time_vect_m[jump_number_m + 1] = T;
    jump_number = jump_number_p + jump_number_m;
    //////////////////////////////////////
    //computation of Xg and Xd
    Xg = malloc((jump_number + 2) * sizeof(double)); //left value of X a
    Xg[0] = 0;
    Xd = malloc((jump_number + 2) * sizeof(double)); //right value of X
    Xd[0] = 0;
    k1 = 1;
    k2 = 1;
    u0 = 0;
    for (k = 1; k <= jump_number; k++)
    {
        w1 = jump_time_vect_p[k1];
        w2 = jump_time_vect_m[k2];
        if (w1 < w2)
        {
            u = w1;
            k1++;
            z = jump_generator_VG(cdf_jump_vect_p, cdf_jump_points, cdf_
        }
        else
        {
            u = w2;
            k2++;
            z = -jump_generator_VG(cdf_jump_vect_m, cdf_jump_points, cdf_
        }
        Xg[k] = drift * (u - u0) + Xd[k - 1];
        Xd[k] = Xg[k] + z;
        u0 = u;
    }
    Xg[jump_number + 1] = drift * (T - u0) + Xd[jump_number];

```



```

Xd[jump_number + 1] = Xg[jump_number + 1];
////////////////////////////////////
//computation of the supremum and the infimum of the Levy path
inf = 0;
sup = 0;
for (j = 1; j <= jump_number; j++)
{
    if (drift > 0)
    {
        if (inf > Xd[j])
            inf = Xd[j];
        if (sup < Xg[j])
            sup = Xg[j];
    }
    else
    {
        if (inf > Xg[j])
            inf = Xg[j];
        if (sup < Xd[j])
            sup = Xd[j];
    }
}
supS = S0 * exp(sup);
if (supS < s_maxmin)
{
    supS = s_maxmin;
    proba = 1;
}
payoff = supS;
supS = S0 * exp(Xd[jump_number + 1] - inf); //antithetic variate ass
if (supS < s_maxmin)
{
    supS = s_maxmin;
    proba += 1;
}
payoff = (payoff + supS) / 2;
proba /= 2;
s1 += payoff;
s += payoff * payoff;
control = S0 * exp(Xd[jump_number + 1]);
s2 += control;

```

```

        s3 += control * control;
        s4 += control * payoff;
        s5 += proba;
        s6 += proba * proba;
        free(Xd);
        free(Xg);
        free(jump_time_vect_p);
        free(jump_time_vect_m);
    }
    cov_payoff_control = s4 / n_paths - s1 * s2 / ((double)n_paths * n_paths);
    var_payoff = (s - s1 * s1 / ((double)n_paths)) / (n_paths - 1);
    var_control = (s3 - s2 * s2 / ((double)n_paths)) / (n_paths - 1);
    cor_payoff_control = cov_payoff_control / (sqrt(var_payoff) * sqrt(var_control));
    control_coef = cov_payoff_control / var_control;
    var_proba = (s6 - s5 * s5 / ((double)n_paths)) / (n_paths - 1);
    *ptprice = exp(-r * T) * (s1 / n_paths - control_coef * (s2 / n_paths -
    *errorprice = 1.96 * sqrt(var_payoff * (1 - cor_payoff_control * cor_payoff_control));
    *ptdelta = (*ptprice - exp(-r * T) * s_maxmin * s5 / (n_paths)) / S0;
    *errordelta = (*errorprice + 1.96 * exp(-r * T) * s_maxmin * sqrt(var_payoff));
    }
    free(v1);
    free(v2);
    free(cdf_jump_vect_p);
    free(cdf_jump_vect_m);
    free(cdf_jump_points);
    return OK;
}

int CALC(MC_VarianceGamma_Floating)(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 VG_Mc_Floating((ptOpt->PathDep.Val.V_NUMFUNC_2)->Par[4].Val.V_PDOUBLE,
    }

static int CHK_OPT(MC_VarianceGamma_Floating)(void *Opt, void *Mod)

```

```

{
    if ((strcmp(((Option *)Opt)->Name, "LookBackCallFloatingEuro") == 0) || (strcmp
        return OK;
    return WRONG;
}

#endif //PremiaCurrentVersion
static int MET(Init)(PricingMethod *Met, Option *Mod)
{
    if (Met->init == 0)
    {
        Met->init = 1;
        Met->Par[0].Val.V_ENUM.value = 0;
        Met->Par[0].Val.V_ENUM.members = &PremiaEnumMCRNGs;
        Met->Par[1].Val.V_LONG = 10000;
    }
    return OK;
}

PricingMethod MET(MC_VarianceGamma_Floating) =
{
    "MC_VG_LookbackFloating",
    { {"RandomGenerator", ENUM, {100}, ALLOW},
      {"N iterations", LONG, {100}, ALLOW}, {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CALC(MC_VarianceGamma_Floating),
    {"Price", DOUBLE, {100}, FORBID}, {"Delta", DOUBLE, {100}, FORBID}, {"Price E
    CHK_OPT(MC_VarianceGamma_Floating),
    CHK_ok,
    MET(Init)
} ;

```