

[Help](#)

```
#include "
href../../../../mod/lmm1d/lmm1d_std/ldi_h_src.pdfldi_h_src.pdf
#include "pnl/pnl_basis.h"
#include "
href../../../../common/math/mc_lmm_glassermanzhao_h_src.pdfmath/mc_lmm_glassermanzhao_h_src.pdf
#include "
href../../../../common/enums_h_src.pdfenums_h_src.pdf

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2010+2) //The "#els
static int CHK_OPT(MC_Schoenmakers_BermudanSwaption)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(MC_Schoenmakers_BermudanSwaption)(void *Opt, void *Mod, PricingMethod *
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else

/**
 * Lower bound for bermudan swaption using Longstaff-Schwartz algorithm
 * We store the regression coefficients in a matrix LS_RegressionCoeffMat
 * We also compute the coefficients regression to estimate the conditional expect
 * These coefficients are stored in a matrix Sch_RegressionCoeffMat
 * @param LS_LowerPrice lower price by Longstaff-Schwartz algorithm on exit
 * @param NbrMCsimulation the number of samples
 * @param ptLib Libor structure contains initial value of libor rates
 * @param ptBermSwpt Swaption structure contains bermudan swaption information
 * @param ptVol Volatility structure contains libor volatility deterministic fun
 * @param generator the index of the random generator to be used
 * @param basis_name regression basis
 * @param DimApprox dimension of regression basis
 * @param NbrStepPerTenor number of steps of discretization between T(i) and T(i
 * @param flag_numeraire measure under wich simulation is done.
 * flag_numeraire=0 -> Terminal measure, flag_numeraire=1 -> Spot measure
 * @param LS_RegressionCoeffMat contains Longstaff-Schwartz algorithm regression
 * @param Sch_RegressionCoeffMat contains regression coefficients needed in Scho
 * Rmk: Libor rates are simulated using the method proposed by Glasserman-Zhao.
```

```

*/
static void MC_BermSwaption_LongstaffSchwartz(double *LS_LowerPrice, int NbrMCsim
{
    int alpha, beta, i, j, m, k, N, NbrExerciseDates, time_index, save_brownian, s
    double tenor, regressed_value, payoff, dW;
    double *VariablesExplicatives;

    Libor *ptLib_current;
    Swaption *ptSwpt_current;
    PnlVect *OptimalPayoff, *LS_RegCoeffVect, *Sch_RegCoeffVect, *ToRegressSch_Vec
    PnlMat *LiborPathsMatrix, *BrownianPathsMatrix, *ExplicativeVariables;
    PnlBasis *basis;

    Nfac = ptVol->numberOfFactors;
    N = ptLib->numberOfMaturities;
    tenor = ptBermSwpt->tenor;
    alpha = (int)(ptBermSwpt->swaptionMaturity / tenor); // T(alpha) is the swapti
    beta = (int)(ptBermSwpt->swapMaturity / tenor); // T(beta) is the swap maturi
    NbrExerciseDates = beta - alpha;
    start_index = 0;
    end_index = beta - 1;
    Nsteps = end_index - start_index;

    save_brownian = 1;
    save_all_paths = 1;
    nbr_var_explicatives = Nfac;

    VariablesExplicatives = malloc(nbr_var_explicatives * sizeof(double));
    ExplicativeVariables = pnl_mat_create(NbrMCsimulation, nbr_var_explicatives);
    OptimalPayoff = pnl_vect_create(NbrMCsimulation);
    ToRegressSch_Vect = pnl_vect_create(NbrMCsimulation);
    LS_RegCoeffVect = pnl_vect_create(0);
    Sch_RegCoeffVect = pnl_vect_create(0);
    LiborPathsMatrix = pnl_mat_create(0, 0); // LiborPathsMatrix contains all the
    BrownianPathsMatrix = pnl_mat_create(0, 0); // We store also the brownian valu

    pnl_mat_resize(LS_RegressionCoeffMat, NbrExerciseDates - 1, DimApprox);
    pnl_mat_resize(Sch_RegressionCoeffMat, (NbrExerciseDates - 1)*Nfac, DimApprox)

    basis = pnl_basis_create(basis_name, DimApprox, nbr_var_explicatives);

```

```

mallocLibor(&ptLib_current, N, tenor, 0.1);

// ptSwpt_current := contains the information about the swap to be exercised
// The maturity of the swap stays the same.
mallocSwaption(&ptSwpt_current, ptBermSwpt->swaptionMaturity, ptBermSwpt->swap

Numeraire(0, ptLib, flag_numeraire);

// Simulation the "NbrMCsimulation" paths of Libor rates. We also store browni
Sim_Libor_Glasserman(start_index, end_index, ptLib, ptVol, generator, NbrMCsim

ptSwpt_current->swaptionMaturity = ptBermSwpt->swapMaturity - tenor; // Last e
time_index = end_index;

// At the last exercise date, price of the option = payoff.
for (m = 0; m < NbrMCsimulation; m++)
{
    pnl_mat_get_row(ptLib_current->libor, LiborPathsMatrix, time_index + m * N
    LET(OptimalPayoff, m) = Swaption_Payoff_Discounted(ptLib_current, ptSwpt_c
}

for (k = NbrExerciseDates - 1; k >= 1; k--)
{
    ptSwpt_current->swaptionMaturity -= tenor; // k'th exercise date
    time_index -= 1;

    /** Least square fitting. */
    for (m = 0; m < NbrMCsimulation; m++)
    {
        for (j = 0; j < Nfac; j++)
        {
            MLET(ExplicativeVariables, m, j) = MGET(BrownianPathsMatrix, time_
        }
    }

    pnl_basis_fit_ls(basis, LS_RegCoeffVect, ExplicativeVariables, OptimalPayo
    pnl_mat_set_row(LS_RegressionCoeffMat, LS_RegCoeffVect, k - 1); // Store r

    /** Regression coefficients needed in Schoenmakers et al. algorithm. */
    for (j = 0; j < Nfac; j++)
    {

```

```

for (m = 0; m < NbrMCsimulation; m++)
{
    for (i = 0; i < Nfac; i++)
    {
        VariablesExplicatives[i] = MGET(BrownianPathsMatrix, time_index + m * Nsteps, i);
    }
    regressed_value = pnl_basis_eval(basis, LS_RegCoeffVect, VariablesExplicatives);

    dW = MGET(BrownianPathsMatrix, time_index + m * Nsteps, j) - MGET(BrownianPathsMatrix, time_index + (m - 1) * Nsteps, j);
    LET(ToRegressSch_Vect, m) = (dW / tenor) * (GET(OptimalPayoff, m));
}

pnl_basis_fit_ls(basis, Sch_RegCoeffVect, ExplicativeVariables, ToRegressSch_Vect);
pnl_mat_set_row(Sch_RegressionCoeffMat, Sch_RegCoeffVect, (k - 1)*Nfac);
}

/** Dynamical programming. */
for (m = 0; m < NbrMCsimulation; m++)
{
    pnl_mat_get_row(ptLib_current->libor, LiborPathsMatrix, time_index + m * Nsteps);
    payoff = Swaption_Payoff_Discounted(ptLib_current, ptSwpt_current, p, time_index + m * Nsteps);

    // If the payoff is null, the OptimalPayoff doesnt change.
    if (payoff > 0)
    {
        for (j = 0; j < Nfac; j++)
        {
            VariablesExplicatives[j] = MGET(BrownianPathsMatrix, time_index + m * Nsteps, j);
        }

        regressed_value = pnl_basis_eval(basis, LS_RegCoeffVect, VariablesExplicatives);

        if (payoff > regressed_value)
        {
            LET(OptimalPayoff, m) = payoff;
        }
    }
}
}

```

// The price at date 0 is the conditional expectation of OptimalPayoff, ie it's

```

*LS_LowerPrice = pnl_vect_sum(OptimalPayoff) / NbrMCsimulation;

free(VariablesExplicatives);
pnl_basis_free(&basis);
pnl_mat_free(&LiborPathsMatrix);
pnl_mat_free(&ExplicativeVariables);
pnl_mat_free(&BrownianPathsMatrix);

pnl_vect_free(&OptimalPayoff);
pnl_vect_free(&LS_RegCoeffVect);
pnl_vect_free(&Sch_RegCoeffVect);
pnl_vect_free(&ToRegressSch_Vect);

freeSwaption(&ptSwpt_current);
freeLibor(&ptLib_current);
}

/** Upper bound for bermudan swaption using Schoenmakers et al. algorithm.
 * @param SwaptionPriceUpper upper bound for the price on exit.
 * @param NbrMCsimulationDual number of simulation in Schoenmakers et al. algorithm.
 * @param NbrMCsimulationPrimal number of simulation in Longstaff-Schwartz algorithm.
 */
static void Schoenmakers(double *SwaptionPriceUpper, double Nominal, long NbrMCs
{
    int i, j, m, N, k, Nfac, alpha, beta, Nsteps, save_all_paths, save_brownian;
    int NbrExerciseDates, start_index, end_index, nbr_var_explicatives;
    double tenor, payoff, numeraire_0, ContinuationValue, LowerPrice_0, LowerPrice;
    double DoobMeyerMartingale, MaxVariable, Delta_0, dW, Z;
    double *VariablesExplicatives;

    PnlMat *LiborPathsMatrix, *BrownianPathsMatrix;
    PnlMat *LS_RegressionCoeffMat, *Sch_RegressionCoeffMat;
    PnlVect *Sch_RegCoeffVect, *LS_RegressionCoeffVect;
    PnlBasis *basis;

    Libor *ptLib_current;
    Swaption *ptSwpt_current;

    Nfac = ptVol->numberOfFactors;

```

```

N = ptLib->numberOfMaturities;
tenor = ptBermSwpt->tenor;
alpha = (int)(ptBermSwpt->swaptionMaturity / tenor); // T(alpha) is the swapti
beta = (int)(ptBermSwpt->swapMaturity / tenor); // T(beta) is the swap maturi
NbrExerciseDates = beta - alpha;

nbr_var_explicatives = Nfac;
VariablesExplicatives = malloc(nbr_var_explicatives * sizeof(double));
basis = pnl_basis_create(basis_name, DimApprox, nbr_var_explicatives);

LS_RegressionCoeffVect = pnl_vect_create(0);
LS_RegressionCoeffMat = pnl_mat_create(0, 0);
Sch_RegCoeffVect = pnl_vect_create(0);
Sch_RegressionCoeffMat = pnl_mat_create(0, 0);

LiborPathsMatrix = pnl_mat_create(0, 0);
BrownianPathsMatrix = pnl_mat_create(0, 0);

mallocLibor(&ptLib_current , N, tenor, 0.);

numeraire_0 = Numeraire(0, ptLib, flag_numeraire);

// ptSwpt_current := le swap qui sera exerce à chaque date de la bermudeene. s
mallocSwaption(&ptSwpt_current, ptBermSwpt->swaptionMaturity, ptBermSwpt->swap

// calcul de la borne inf du prix et des coefficients de regression.
MC_BermSwaption_LongstaffSchwartz(&LowerPrice_0, NbrMCsimulationPrimal, p, ptL

Delta_0 = 0;

save_brownian = 2; // save_brownian = 2. We also save intermediate steps.
save_all_paths = 1; // If save_all_paths=1, we store the simulated value of li

start_index = 0;
end_index = beta - 1;
Nsteps = end_index - start_index;

// Simulate "NbrMCsimulationDual" paths
Sim_Libor_Glasserman(start_index, end_index, ptLib, ptVol, generator, NbrMCsim

for (m = 0; m < NbrMCsimulationDual; m++)

```

```

{
    start_index = alpha;

    pnl_mat_get_row(ptLib_current->libor, LiborPathsMatrix, start_index + m *
ptSwpt_current->swaptionMaturity = ptBermSwpt->swaptionMaturity; // First
payoff = Swaption_Payoff_Discounted(ptLib_current, ptSwpt_current, p, flag

    pnl_mat_get_row(LS_RegressionCoeffVect, LS_RegressionCoeffMat, 0);

    for (j = 0; j < Nfac; j++)
    {
        VariablesExplicatives[j] = MGET(BrownianPathsMatrix, start_index * Nbr
    }
    ContinuationValue = pnl_basis_eval(basis, LS_RegressionCoeffVect, Variable
    LowerPrice_alpha = MAX(ContinuationValue, payoff); // Price of the option

    DoobMeyerMartingale = LowerPrice_alpha; // Martingale value at t=T(alpha).

    MaxVariable = payoff - DoobMeyerMartingale; // Value of Duale Variable at

    for (k = 0; k < NbrExerciseDates - 1; k++)
    {
        start_index = alpha + k;
        end_index = start_index + 1;

        for (i = 1 ; i <= NbrStepPerTenor; i++)
        {
            for (j = 0; j < Nfac; j++)
            {
                VariablesExplicatives[j] = MGET(BrownianPathsMatrix, i - 1 + s
            }

            // Here we compute the stochastic integral of Z process with respec
            for (j = 0; j < Nfac; j++)
            {
                pnl_mat_get_row(Sch_RegCoeffVect, Sch_RegressionCoeffMat, k *
                Z = pnl_basis_eval(basis, Sch_RegCoeffVect, VariablesExplicati

                dW = MGET(BrownianPathsMatrix, i + start_index * NbrStepPerTen
                dW -= VariablesExplicatives[j];
                DoobMeyerMartingale += Z * dW;

```

```

        }
    }

    ptSwpt_current->swaptionMaturity += tenor;
    pnl_mat_get_row(ptLib_current->libor, LiborPathsMatrix, end_index + m
    payoff = Swaption_Payoff_Discounted(ptLib_current, ptSwpt_current, p,

    MaxVariable = MAX(MaxVariable, payoff - DoobMeyerMartingale); // Value
}

    Delta_0 += MaxVariable; // somme de MonteCarlo
}

Delta_0 /= NbrMCsimulationDual;

*SwaptionPriceUpper = (numeraire_0 * Nominal) * (LowerPrice_0 + 0.5 * Delta_0)

free(VariablesExplicatives);

pnl_basis_free(&basis);
pnl_mat_free(&LiborPathsMatrix);
pnl_mat_free(&BrownianPathsMatrix);
pnl_mat_free(&Sch_RegressionCoeffMat);
pnl_mat_free(&LS_RegressionCoeffMat);

pnl_vect_free(&Sch_RegCoeffVect);
pnl_vect_free(&LS_RegressionCoeffVect);

freeSwaption(&ptSwpt_current);
freeLibor(&ptLib_current);
}

static int MCSchoenmakers(NumFunc_1 *p, double l0, double sigma_const, int nb_fa
{
    Volatility *ptVol;
    Libor *ptLib;
    Swaption *ptBermSwpt;
    int init_mc;
    int Nbr_Maturities;

    Nbr_Maturities = (int)(swap_maturity / tenor + 0.1);

```



```

mallocLibor(&ptLib , Nbr_Maturities, tenor, 10);
mallocVolatility(&ptVol , nb_factors, sigma_const);
mallocSwaption(&ptBermSwpt, swaption_maturity, swap_maturity, 0.0, swaption_st

init_mc = pnl_rand_init(generator, nb_factors, NbrMCsimulationPrimal);
if (init_mc != OK) return init_mc;

Schoenmakers(swaption_price_upper, Nominal, NbrMCsimulationDual, NbrMCsimulati

freeLibor(&ptLib);
freeVolatility(&ptVol);
freeSwaption(&ptBermSwpt);

return init_mc;
}

int CALC(MC_Schoenmakers_BermudanSwaption)(void *Opt, void *Mod, PricingMethod *
{
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;
    TYPEMOD *ptMod = (TYPEMOD *)Mod;

    return MCSchoenmakers(ptOpt->PayOff.Val.V_NUMFUNC_1,
                           ptMod->10.Val.V_PDOUBLE,
                           ptMod->Sigma.Val.V_PDOUBLE,
                           ptMod->NbFactors.Val.V_ENUM.value,
                           ptOpt->BMaturity.Val.V_DATE - ptMod->T.Val.V_DATE,
                           ptOpt->OMaturity.Val.V_DATE - ptMod->T.Val.V_DATE,
                           ptOpt->Nominal.Val.V_PDOUBLE,
                           ptOpt->FixedRate.Val.V_PDOUBLE,
                           ptOpt->ResetPeriod.Val.V_DATE,
                           Met->Par[0].Val.V_LONG,
                           Met->Par[1].Val.V_LONG,
                           Met->Par[2].Val.V_ENUM.value,
                           Met->Par[3].Val.V_ENUM.value,
                           Met->Par[4].Val.V_INT,
                           Met->Par[5].Val.V_INT,
                           Met->Par[6].Val.V_ENUM.value,
                           &(Met->Res[0].Val.V_DOUBLE));
}

```

```

static int CHK_OPT(MC_Schoenmakers_BermudanSwaption)(void *Opt, void *Mod)
{
    if ((strcmp(((Option *)Opt)->Name, "PayerBermudanSwaption") == 0) || (strcmp((
        return OK;
    else
        return WRONG;
}

#endif //PremiaCurrentVersion

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
    if (Met->init == 0)
    {
        Met->init = 1;
        Met->Par[0].Val.V_LONG = 50000;
        Met->Par[1].Val.V_LONG = 10000;
        Met->Par[2].Val.V_ENUM.value = 0;
        Met->Par[2].Val.V_ENUM.members = &PremiaEnumMCRNGs;
        Met->Par[3].Val.V_ENUM.value = 0;
        Met->Par[3].Val.V_ENUM.members = &PremiaEnumBasis;
        Met->Par[4].Val.V_INT = 10;
        Met->Par[5].Val.V_INT = 10;
        Met->Par[6].Val.V_ENUM.value = 0;
        Met->Par[6].Val.V_ENUM.members = &PremiaEnumAfd;
    }

    return OK;
}

PricingMethod MET(MC_Schoenmakers_BermudanSwaption) =
{
    "MC_Schoenmakers_BermudanSwaption",
    {
        {"N iterations Primal", LONG, {100}, ALLOW},
        {"N iterations Dual", LONG, {100}, ALLOW},
        {"RandomGenerator", ENUM, {100}, ALLOW},
        {"Basis", ENUM, {100}, ALLOW},
        {"Dimension Approximation", INT, {100}, ALLOW},
        {"Nbr discretisation step per periode", INT, {100}, ALLOW},
    }
}

```

```

    {"Martingale Measure", ENUM, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
},
CALC(MC_Schoenmakers_BermudanSwaption),
{{"Price", DOUBLE, {100}, FORBID}, {" ", PREMIA_NULLTYPE, {0}, FORBID}},
CHK_OPT(MC_Schoenmakers_BermudanSwaption),
CHK_ok,
MET(Init)
};

```