Help

```
#include <stdlib.h>
#include  "cir1d_stdi.h"
#include "error_msg.h"

/*Product*/
static double dt, dr, r_min, r_max;
static double *r_vect;
static double *V, *Vp, *Option_values, *Ps, * *Obst;
static double *beta, *alpha_r, *beta_r, *gamma_r_, *alpha_l, *beta_l, *gamma_l;

/*Memory Allocation*/
static int memory_allocation(int Nt, int Ns)
{
  int i;

  if ((Obst = malloc(sizeof(double *) * (Nt + 1))) == NULL)
    {
      printf("Allocation error");
      exit(1);
    }
  for (i = 0; i <= Nt; i++)
    {
      Obst[i] = malloc(sizeof(double) * (Ns + 1));
    }

  r_vect = malloc((Ns + 1) * sizeof(double));
  if (r_vect == NULL)
    return MEMORY_ALLOCATION_FAILURE;

  V = malloc((Ns + 1) * sizeof(double));
  if (V == NULL)
    return MEMORY_ALLOCATION_FAILURE;

  Vp = malloc((Ns + 1) * sizeof(double));
  if (Vp == NULL)
    return MEMORY_ALLOCATION_FAILURE;

  Option_values = malloc((Ns + 1) * sizeof(double));
  if (Option_values == NULL)
```

```
    return MEMORY_ALLOCATION_FAILURE;

  Ps = malloc((Ns + 1) * sizeof(double));
  if (Ps == NULL)
    return MEMORY_ALLOCATION_FAILURE;

  beta = malloc((Ns + 1) * sizeof(double));
  if (beta == NULL)
    return MEMORY_ALLOCATION_FAILURE;

  alpha_l = malloc((Ns + 1) * sizeof(double));
  if (alpha_l == NULL)
    return MEMORY_ALLOCATION_FAILURE;

  beta_l = malloc((Ns + 1) * sizeof(double));
  if (beta_l == NULL)
    return MEMORY_ALLOCATION_FAILURE;

  gamma_l = malloc((Ns + 1) * sizeof(double));
  if (gamma_l == NULL)
    return MEMORY_ALLOCATION_FAILURE;

  alpha_r = malloc((Ns + 1) * sizeof(double));
  if (alpha_r == NULL)
    return MEMORY_ALLOCATION_FAILURE;

  beta_r = malloc((Ns + 1) * sizeof(double));
  if (beta_r == NULL)
    return MEMORY_ALLOCATION_FAILURE;

  gamma_r_ = malloc((Ns + 1) * sizeof(double));
  if (gamma_r_ == NULL)
    return MEMORY_ALLOCATION_FAILURE;

  return OK;
}

/*Memory Desallocation*/
static void free_memory(int Nt)
{
  int i;
```

```
  for (i = 0; i < Nt + 1; i++)
    free(Obst[i]);
  free(Obst);

  free(beta);
  free(alpha_r);
  free(beta_r);
  free(gamma_r_);
  free(alpha_l);
  free(beta_l);
  free(gamma_l);

  free(r_vect);

  free(V);
  free(Vp);
  free(Ps);
  free(Option_values);

  return;
}

/*Compute Coupon Bearing*/
static int cb_cir(int Nt, int Nt0, int Ns, double K, double periodicity, double
{
  int i, z, TimeIndex;

  /*Maturity conditions for Coupon Bearing*/
  for (i = 0; i <= Ns; i++)
    Ps[i] = 1. + K * periodicity;

  /*Finite Difference Cycle*/
  for (TimeIndex = Nt - 1; TimeIndex >= Nt0; TimeIndex--)
    {
      /*Right factor*/
      V[0] = beta_r[0] * Ps[0] + gamma_r_[0] * Ps[1];
      for (i = 1; i < Ns; i++)
        V[i] = alpha_r[i] * Ps[i - 1] + beta_r[i] * Ps[i] + gamma_r_[i] * Ps[i +

      /*Backward Steps*/
```

```
      Vp[Ns - 1] = V[Ns - 1];
      beta[Ns - 1] = beta_l[Ns - 1];
      for (i = Ns - 2; i >= 0; i--)
        {
          beta[i] = beta_l[i] - gamma_l[i] * alpha_l[i + 1] / beta[i + 1];
          Vp[i] = V[i] - gamma_l[i] * Vp[i + 1] / beta[i + 1];
        }

      /*Forward Steps*/
      Ps[0] = Vp[0] / beta[0];
      for (i = 1; i < Ns; i++)
        Ps[i] = (Vp[i] - alpha_l[i] * Ps[i - 1]) / beta[i];

      /*Coupon adjustment*/
      for (i = 0; i < Ns; i++)
        for (z = 0; z < nb_coupon; z++)
          {
            if ((fabs((double)TimeIndex * dt - (first_payement + (double)z * per
              {
                Ps[i] += K * periodicity;
              }
          }
    }

  return 1.;
}

/*Finite Difference for the options prices*/
static int zbo_implicit(int Nt, int Ns, NumFunc_1 *p)
{
  int i, j, TimeIndex;

  /*Maturity conditions*/
  for (j = 0; j <= Ns; j++)
    Option_values[j] = (p->Compute)(p->Par, Ps[j]);

  /*Finite Difference Cycle*/
  for (TimeIndex = Nt - 1; TimeIndex >= 0; TimeIndex--)
    {
      /*Right factor*/
      V[0] = beta_r[0] * Option_values[0] + gamma_r_[0] * Option_values[1];
```

```
      for (i = 0; i < Ns; i++)
        V[i] = alpha_r[i] * Option_values[i - 1] + beta_r[i] * Option_values[i]

      /*Backward Steps*/
      Vp[Ns - 1] = V[Ns - 1];
      beta[Ns - 1] = beta_l[Ns - 1];
      for (i = Ns - 2; i >= 0; i--)
        {
          beta[i] = beta_l[i] - gamma_l[i] * alpha_l[i + 1] / beta[i + 1];
          Vp[i] = V[i] - gamma_l[i] * Vp[i + 1] / beta[i + 1];
        }

      /*Forward Steps*/
      Option_values[0] = Vp[0] / beta[0];
      for (i = 1; i < Ns; i++)
        Option_values[i] = (Vp[i] - alpha_l[i] * Option_values[i - 1]) / beta[i]

    }

  return 1.;
}


/*Swaption=Option on Coupon-Bearing Bond*/
static int swaption_cir1d(double r0, double k, double t0, double sigma, double t
{
  int i, j, nb_coupon, Nt0, Nt;
  double val, val1, tmp, first_payement, sigma2;

  /*Compute probabilities*/
  Nt = NtY * (long)((T - t0) / periodicity);
  memory_allocation(Nt, Ns);

  /*Space Localisation*/
  dt = (T - t0) / (double)Nt;
  r_min = 0.;
  r_max = 2.;
  dr = (r_max - r_min) / (double)Ns;
  r_vect[0] = r_min;
  for (i = 0; i <= Ns; i++)
    r_vect[i] = r_min + (double)i * dr;
```

```
sigma2 = SQR(sigma);

/*Boundary*/
/*Computation of Rhs coefficients*/
alpha_r[0] = 0.;
beta_r[0] = (1. - cn_theta) * (1 - k * theta * (dt / dr));
gamma_r_[0] = (1. - cn_theta) * (k * theta * (dt / dr));

/*Computation of Lhs coefficients*/
alpha_l[0] = 0.;
beta_l[0] = cn_theta * (1 + k * theta * (dt / dr));
gamma_l[0] = cn_theta * (-k * theta * (dt / dr));

/*Computation of the Matrix*/
for (i = 1; i < Ns; i++)
  {
    /*Computation of Rhs coefficients*/
    alpha_r[i] = (1. - cn_theta) * (0.5 * sigma2 * r_vect[i] * (dt / SQR(dr))
    beta_r[i] = 1. - (1. - cn_theta) * (sigma2 * r_vect[i] * (dt / SQR(dr)) +
    gamma_r_[i] = (1. - cn_theta) * (0.5 * sigma2 * r_vect[i] * (dt / SQR(dr))

    /*Computation of Lhs coefficients*/
    alpha_l[i] = cn_theta * (-0.5 * sigma2 * r_vect[i] * (dt / SQR(dr)) + 0.5
    beta_l[i] = 1. + cn_theta * (sigma2 * r_vect[i] * (dt / SQR(dr)) + r_vect[
    gamma_l[i] = cn_theta * (-0.5 * sigma2 * r_vect[i] * (dt / SQR(dr)) - 0.5
  }

/*Number of Step for the Option*/
Nt0 = NtY * (long)((t - t0) / periodicity);

/*Compute Coupon Bearing*/
first_payement = t + periodicity;
nb_coupon = (int)((T - first_payement) / periodicity);
cb_cir(Nt, Nt0, Ns, K, periodicity, first_payement, nb_coupon);

/*Compute Option Prices*/
tmp = p->Par[0].Val.V_DOUBLE;
p->Par[0].Val.V_DOUBLE = 1.;
zbo_implicit(Nt0, Ns, p);

/*Linear Interpolation*/
```

```
  j = 0;
  while (r_vect[j] < r0)
    j++;
  val = Option_values[j];
  val1 = Option_values[j - 1];

  /*Price*/
  *price = Nominal * (val + (val - val1) * (r0 - (r_vect[j])) / ((r_vect[j]) - (

  /*Memory Disallocation*/
  p->Par[0].Val.V_DOUBLE = tmp;
  free_memory(Nt);


  return OK;
}

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


  return swaption_cir1d(ptMod->r0.Val.V_PDOUBLE, ptMod->k.Val.V_DOUBLE, ptMod->T
                        ptMod->theta.Val.V_PDOUBLE, ptOpt->BMaturity.Val.V_DATE,
                        ptOpt->EuOrAm.Val.V_BOOL, ptOpt->Nominal.Val.V_PDOUBLE,
}


static int CHK_OPT(FD_GaussSWAPTION)(void *Opt, void *Mod)
{

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

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
  if (Met->init == 0)
```

```
    {
      Met->init = 1;

      Met->Par[0].Val.V_INT2 = 30;
      Met->Par[1].Val.V_INT2 = 300;
      Met->Par[2].Val.V_RGDOUBLE = 0.5;


    }
  return OK;
}

PricingMethod MET(FD_GaussSWAPTION) =
{
  "FD_Gauss_Cir1d_Swaption",
  { {"TimeStepNumber for Period", LONG, {100}, ALLOW}, {"SpaceStepNumber", INT2,
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(FD_GaussSWAPTION),
  {{"Price", DOUBLE, {100}, FORBID}, {" ", PREMIA_NULLTYPE, {0}, FORBID}},
  CHK_OPT(FD_GaussSWAPTION),
  CHK_ok,
  MET(Init)
} ;
```