Premia Kernel
version 5

C. Martini, A.Zanette

July 3, 2002

Contents

Architecture

Even if it is written in C (even ANSI C), the kernel of Premia is strongly object-oriented. Premia relies on 3 basic objects: *models*, *options*, and *pricing methods*. All the types of Premia are defined in optype.h.

1 Models and Options

By model, we mean the modeling of the financial environment (underlying of the option, interest rate for options on securities...). For instance the type of the Black-Scholes model defines the spot value, the current date, the instantaneous interest rate, the volatility, the trend (or yet historical drift) or the underlying.
The options directly correspond to the contingent claim itself. For instance, the parameters of a Call option will be its strike, maturity, exercise feature.

1.1 Options

The options are grouped by families, every family corresponding to a subdirectory of the directory OPT . This directory bears the family name (ex: STD) and contains in a systematic manner a .c and a .h which bear the same name (ex: std.c et std.h). The .h file contains the definition of the type of the options of the family, the .c contains input (Get), output (Show) and check (Check) functions for this type. All the other files of the directory are .c files, each corresponding to a precise option. They bear the name of the option at hand (ex: CallEuro.c). They contain an instance of the type of the family, and also an initialization function (Init).
Notice that we distinguish between a European Call and an American Call (i.e. there are two files).
To allow some kind of automatic treatment for the options of different families, we also designed a kind of super-type (or yet class) Option : every instance of a specific type (like STD) is wrapped in an instance of this type. The type Option is the following:

```
typedef struct Option{

    Label       ID;

    Label       Name;

    void*       TypeOpt;

    int             (*Get)(int user, Planning*,struct Option*);

    int             (*Show)(int user,Planning*,struct Option*);

    int             (*Check)(int user,Planning*,struct Option*);
```

```
    int             (*Init)(struct Option*);

} Option;
```

ID will store the name of the family (e.g. STD), Name the option's name (e.g. CallEuro), TypeOpt is a universal pointer which will point to the instance of the specific type we discussed above. The other pointers will store the addresses of the functions described above. user is only a flag which is intended to manage the input/output stuff (file or screen, etc...). Planning* is related to the possibility of iterating every scalar variable in the fields of TypeOpt*, Option* points to itself.
For every instance of a specific type of option, the wrapping instance of the super-type is automatically created by the macro MAKEOPT(X) (in optype.h). This macro is intended also to avoid conflicting names between two options of different families which bear the same name (e.g. CallDownOutEuro in lim and also limdisc . Indeed the new super-object is created under the name directoryname_optionname (ex: std_CallEuro).
The master file of the directory (e.g. std.c) contains an array of all the options of the directory:

```
extern Option OPT(CallEuro);
 extern Option OPT(CallSpreadAmer);
extern Option OPT(CallSpreadEuro);
 extern Option OPT(DigitAmer);
extern Option OPT(DigitEuro);
 extern Option OPT(PutAmer);
 extern Option OPT(PutEuro);
 extern Option OPT(CallAmer);

  Option*OPT(family)[ ]=
        {
                &OPT(CallEuro),
                &OPT(PutEuro),
                &OPT(CallSpreadEuro),
                &OPT(DigitEuro),
                &OPT(CallAmer),
                &OPT(PutAmer),
                &OPT(CallSpreadAmer),
                &OPT(DigitAmer),
                NULL
```

```
};
```

The above macro `OPT(X)` expands in `TYPEOPT\_X` where TYPEOPT is a macro defined in directoryname.h (e.g. std.h) which itself expands in directoryname:

```
#define TYPEOPT STD
```

In fact the name of the type of a given option family is imposed, it is `TYPEOPT`. Notice also that the name of the array of options is also imposed, it is `OPT(family)`.

1.2 Models

It is almost the same story for the models (subdirectories of the directory Model . The type model is:

```
typedef struct  Model{
        Label           ID;
        Label           Name;
        void*           TypeModel;
        int             (*Get)(int user, Planning*,struct Model*);
        int             (*Show)(int user,Planning*,struct Model*);
        int             (*Check)(int user,Planning*,void*);
        int             (*Init)(void*);
} Model;
```

The only difference is that there is only a single instance (and therefore no array) of the type defined in directoryname.h (e.g. bs1d.h, which is in the file directoryname.c along with the corresponding initialization function. The macros corresponding to `OPT(X)` and `MAKEOPT(X)` are `MOD(X)` and `MAKEMOD(X)`.

1.3 Accessing to Models and Options objects

In the main file Premia 5.c the above objects are first declared as `extern` ones, next they are stored in two arrays `models` and `families`. We need here of course to expand explicitly the macros `OPT` and `MOD` since we are neither in a subdirectory of Opt nor Mod .

```
extern Model BS1D_model;
extern Model BS2D_model;
```

```
Model*models[ ]=
        {
                &BS1D_model,
                &BS2D_model,
                NULL
        };

extern Family STD_family;
extern Family LIM_family;
extern Family LIMDISC_family;
extern Family DOUBLIM_family;
extern Family PAD_family;
extern Family STD2D_family;
Family *families[ ]=
        {
                &STD_family,
                &LIM_family,
                &LIMDISC_family,
                &DOUBLIM_family,
                &PAD_family,
                &STD2D_family,
                NULL
        };
```

where the type Family is

```
typedef Option* Family[MAX_OPT];
```

2 Pricing methods

2.1 Subdirectories of pricing methods

The pricing methods are grouped by families which are defined as a combination of a given model and a given option family. Each family is stored in a subdirectory of the corresponding model directory, the name of which is modelname_optionfamilyname (e.g. bs1d_std in the bs1d directory stores all the algorithms pertaining to the pricing of standard options within the bs1d model).
Every such directory contains a masterfile with name modelname_optionfamilyname.c, a header file modelname_optionfamilyname.h (which mostly includes the

files modelname.h and optionfamilyname.h and therefore the corresponding macros TYEMOD and TYPEOPT).
All the other files (but one, see later) are .c files, each corresponding to a precise pricing method. Every such files bears the name of the method (with a prefix which indicates the nature of the algorithm: fd_, tr_, mc_, ap_, cf_) and defines an object of type PricingMethod which stores all the parameters and address of the pricing method itself.
The last file of the directory is a file modelname_optionfamilyname_test.c which stores a DynamicalTest object which is pertaining to the simulation of a dynamical delta-hedge of any option of the family within the model at hand, using the prices and hedge ratios given by one of the pricing methods of the directory.

2.2 Pricing method files

Every pricing method is coded in a single C file which bears the method name: methodename.c (ex: Tree_CoxRossRubinstein.c). The structure of such a file is two-fold: in a first part, where there is nothing mandatory regarding input-output parameters, names, etc..., the pricing function itself is coded. There maybe one or more functions within this part, the only constraint is to declare everything as static in order to avoid conflicting names between different routines. In this part the functions and/or objects of mathtools.c, random.c, numfunc.c maybe used, also some flags or macros of error_msg.c and optype.h. These files are in the directory Common or Common\Math but the header files are included in the file modelname_optionfamilyname.h, which is itself included at the first line of every pricing method.
This first part of the file may look like:

```
#include  "bs1d_std.h"

static int CoxRossRubinstein_79(int am,double s,NumFunc_1  *p,double t,double r,
    double divid,double sigma,int N, double *ptprice,double *ptdelta)
    {
       int i,j;
       double u,d,h,pu,pd,a1,stock,upperstock;
       double *P,*iv;


       /*Price, intrisic value arrays*/
       P=(double *)malloc((N+1)*sizeof(double));
```

```
if (P==NULL)
    return MEMORY_ALLOCATION_FAILURE;
iv=(double *)malloc((2*N+1)*sizeof(double));
if (iv==NULL)
    return MEMORY_ALLOCATION_FAILURE;

/*Up and Down factors*/
h=t/(double)N;
a1= exp(h*(r-divid));
u = exp(sigma*sqrt(h));
d= 1./u;

/*Risk-Neutral Probability*/
pu=(a1-d)/(u-d);
pd=1.-pu;

 if ((pd>=1.) || (pd<=0.))
    return NEGATIVE_PROBABILITY;

pu*=exp(-r*h);
pd*=exp(-r*h);
/*Intrisic value initialisation*/
upperstock=s;
for (i=0;i<N;i++)
    upperstock*=u;
stock=upperstock;
for (i=0;i<2*N+1;i++)
    {
    iv[i]=(p->Compute)(p->Par,stock);
    stock*=d;
    }

/*Terminal Values*/
for (j=0;j<=N;j++)
    P[j]=iv[2*j];
/*Backward Resolution*/
for (i=1;i<=N-1;i++)
    for (j=0;j<=N-i;j++)
        {
        P[j]=pu*P[j]+pd*P[j+1];
        if (am)
```

```
                P[j]=MAX(iv[i+2*j],P[j]);
            }

        /*Delta*/
        *ptdelta=(P[0]-P[1])/(s*u-s*d);

        /*First time step*/
        P[0]=pu*P[0]+pd*P[1];
        if (am)
            P[0]=MAX(iv[N],P[0]);

        /*Price*/
        *ptprice=P[0];

        free(P);
        free(iv);

        return OK;
    }
```

The second part of the file consists of two functions and one object (with its initialization function) which are designed to connect the previous stuff to the software. First there is a wrapping function, with a mandatory name and parameter list, which calls the suitable pricing functions of the previous part with the parameters of the objects *option*, *model* at hand, and also the *PricingMethod* object parameters (this last one comes later on). In our case this could be:

```
int CALC(TR_CoxRossRubinstein)(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
    CoxRossRubinstein_79(ptOpt->EuOrAm.Val.V_BOOL,ptMod->S0.Val.V_PDOUBLE,
     ptOpt->PayOff.Val.V_NUMFUNC_1,ptOpt->Maturity.Val.V_DATE-ptMod->T.Val.V_DAT
```

```
    r,divid,ptMod->Sigma.Val.V_PDOUBLE,Met->Par[0].Val.V_INT,
    &(Met->Res[0].Val.V_DOUBLE),&(Met->Res[1].Val.V_DOUBLE));
   }
```

The macro `CALC(X)` expands in modelname_optionfamilyname_X, its main purpose is to avoid conflicting names between different modules. The return value of this function should be zero if everything is OK, something else otherwise. It is possible to make use of the error messages stuff discussed before by returning the adequate flag. The somewhat heavy way of accessing the fields of the objects `*Opt`, `*Mod`, and `*Met` will be discussed later.

The second function is a little test function which returns zero if the option `*Opt` may be priced with the pricing routine, anything else otherwise. Note that this function, with a mandatory name and format, also takes the object `*Mod` as argument: this is required for instance in the case of pricing methods which does not handle forward-starting options in the Asian or Lookback families. Notice also that it will be applied only to the options of the family *optionfamilyname*.

```
int CHK_OPT(TR_CoxRossRubinstein)(void *Opt, void *Mod)
    {
    Option* ptOpt=(Option*)Opt;
    TYPEOPT* opt=(TYPEOPT*)(ptOpt->TypeOpt);

    return  OK;
    }
```

The `CHK_OPT(X)` macro expands in CHK_OPT_modelname_optionfamilyname_X. In our example every option may be priced with this routine, so the body of the function is empty. Here is another example for a closed formula:

```
int CHK_OPT(CF_Call)(void *Opt, void *Mod)
    {
    return strcmp( ((Option*)Opt)->Name,"CallEuro");
    }
```

or yet for a routine which applies only to American options:

```
int CHK_OPT(FiniteDifference_Psor)(void *Opt)
        {
        Option* ptOpt=(Option*)Opt;
        TYPEOPT* opt=(TYPEOPT*)(ptOpt->TypeOpt);

        if ((opt->EuOrAm). Val.V_BOOL==AMER)
```

```
                return OK;

        return  WRONG;
        }
```

The last component of the file is the object PricingMethod itself. The routine should be accessed normally through this object, which therefore contains all the required information. the type PricingMethod is the following (in optype.h):

```
/*Pricing Methods*/
typedef struct PricingMethod{

    Label                          Name;

    VAR Par[MAX_PAR];

    int                           (*Compute)(void*,void*,struct PricingMethod*);

    VAR Res[MAX_PAR];

    int                           (*CheckOpt)(void*,void*);

    int                           (*Check)(int user, Planning*,void*);

    int                           (*Init)(struct PricingMethod*);

} PricingMethod;
```

The field Check is intended to check possible constrains between the input parameters of the routine. The functions written in chk.c in Common may be used for that purpose. In our case the last part of our file may look like:

```
static int MET(Init)(PricingMethod *Met)
    {
    static int first=1;

    if (first)
        {
        Met->Par[0].Val.V_INT2=100;

        first=0;
```

```
        }

    return OK;
    }

PricingMethod MET(TR_CoxRossRubinstein)=
    {
    "TR_CoxRossRubinstein",
        {{"StepNumber",INT2,100,ALLOW},
         {" ",END,0,FORBID}},
        CALC(TR_CoxRossRubinstein),
        {{"Price",DOUBLE,100,FORBID},
         {"Delta",DOUBLE,100,FORBID},
         {" ",END,0,FORBID}},
        CHK_OPT(TR_CoxRossRubinstein),
        CHK_tree,
        MET(Init)
    };
```

Notice that the names of the initialization function and method object are mandatory. For the generation of the documentation system and hyperlinks, the name of the routine should be that of the C file (without the extension) . The macro `MET(X)` expands in model-name_optionfamilyname_X.
The input (Get), output (Show) and check (Check) functions for the type PricingMethod are in method.c in the directory Common .

2.3 The master file (e.g. bs1d_std.c)

This file begins with a function which checks the compatibility of the parameters of the object `*Mod` and `*Opt`. For instance it checks that the current date is before maturity. It returns zero if everything is OK:

```
int MOD_OPT(ChkMix)(Option *Opt,Model *Mod)
    {
    TYPEOPT* ptOpt=( TYPEOPT*)(Opt->TypeOpt);
    TYPEMOD* ptMod=( TYPEMOD*)(Mod->TypeModel);
    int status=OK;

    if ((ptOpt->Maturity.Val.V_DATE)<=(ptMod->T.Val.V_DATE))
        {
        Fprintf(TOSCREENANDFILE,"Current date greater than maturity!\n");
```

```
        status+=1;
        };

    return status;
    }
```

The macro `MOD_OPT(X)` expands in modelname_optionfamilyname_X. The name of the function is mandatory.

The next item is an array of the pricing methods of the directory:

```
extern PricingMethod MET(CF_Call);
extern PricingMethod MET(CF_Put);
[...]
extern PricingMethod MET(TR_BBSR);

PricingMethod* MOD_OPT(methods)[]={
        &MET(CF_Call),
        &MET(CF_Put),
        [...]
        &MET(TR_BBSR),
        NULL
    };
```

Next comes an object of type Pricing which essentially points to the previous objects. The access to the routines of the directory should be made through this last one. The type Pricing (in optype.h) is defined as:

```
typedef struct Pricing{

    Label               ID;

    PricingMethod**     Methods;

    DynamicTest*        Test;

    int                 (*CheckMixing)(Option*,Model*);

} Pricing;
```

In every master file of a pricing methods directory the instance of this type should be the following:

```
extern DynamicTest MOD_OPT(test);
```

```
Pricing MOD_OPT(pricing)={
                ID_MOD_OPT,
                MOD_OPT(methods),
                &MOD_OPT(test),
                MOD_OPT(ChkMix)
        };
```

Note that the object `MOD_OPT(test)` of type DynamicTest is defined in the file modelname_optionfamilyname_test.c (e.g. bs1d_std_test.c), as we shall see later.

2.4 Shared routines

The feature 'one routine-one file' of course has its limitations. If within the same family (directory) of pricing methods a same function is used in several files it should be defined (not declared, but defined) with the keyword `static` in the file modelname_optionfamilyname_.h. This happens for instance in the bs1d_std family for which the file bs1d_std.h is the following:

```
#ifndef  _BS1D_STD_H
#define _BS1D_STD_H

#include "bs1d.h"
#include "std.h"

#include "mathtools.h"
#include "random.h"
#include "numfunc.h"
#include "transopt.h"

static double Nd1(double s,double r,double divid,
    double sigma,double T,double K)
    {
    double d1=(log(s/K)+(r-divid+0.5*sigma*sigma)*T)/(sigma*sqrt(T));

    return  N(d1);
    }

#endif
```

In the more exceptional case where a routine or function may be used in pricing method files across different directories, it should be declared in a

global way in the file transopt.h and defined, also in a global way and not as a static function, in one file method.c. For instance: In CF_Call.c:

```
int Call_BlackScholes_73(double s,double k,double t,double r,
    double divid,double sigma,double *ptprice,double *ptdelta)
```

instead of `static int Call` ... The file transopt.h could then be:

```
#ifndef  _TRANSOPT_H
#define _TRANSOPT_H

int Call_BlackScholes_73(double s,double k,double t,double r,
    double divid,double sigma,double *ptprice,double *ptdelta);
[..]
#endif
```

3 Dynamical tests

Within every directory of pricing methods there should be a single file modelname_optionfamilyname_test.c which contains the simulation of a dynamically delta-hedged selling of an option of the family until maturity, in discrete time.
The situation is almost the same as for a pricing method, except that there is no check on the option (i.e. the dynamical test should work for every option of the family). The type DynamicTest (in optype.h) is the following:

```
/*Dynamic Tests*/

typedef struct  DynamicTest {

    Label            Name;

    VAR Par[MAX_PAR];

    int             (*Simul)(void*,void*,PricingMethod *Met,struct DynamicTest *

    VAR Res[MAX_PAR];

    int             (*Check)(int user, Planning*,void*);

    int             (*Init)(struct DynamicTest*, Option*);
```

```
} DynamicTest ;
```

The Get, Show and Check utilities for this type are in the file test.c in Common .

4 The VAR system

For the purpose of easily modifying and also iterating the various parameters which come into play either through a model, an option, a pricing method, we have designed a somewhat elaborated type which is the following:

```
typedef struct VAR{
    Label   Vname;
    int Vtype;
    union   {
        int V_INT;
        int V_INT2;
        double V_DOUBLE;
        long V_LONG;
        double V_PDOUBLE;
        double V_SPDOUBLE;
        double V_RGDOUBLE051;
        double V_DATE;
        double V_RGDOUBLE;
        double V_RGDOUBLE12;
        int V_BOOL;
        int V_PADE;
        int V_RGINT13;
        int V_GENER;
        double V_RGDOUBLE14;
        struct NumFunc_1* V_NUMFUNC_1;
        struct NumFunc_2* V_NUMFUNC_2;
        struct PtVar* V_PTVAR;
        struct DoubleArray* V_DOUBLEARRAY;
        } Val;
    int   Viter;
    } VAR;
```

`Vname` stores the name of the parameter, `Vtype` is a flag which describes its type (which may be a user created type), `Val` stores its value. The various

keys of access to this union are all of the form V_Vtype for clarity's sake. The last field `Viter` is a flag which describes the state of the parameter regarding iteration and also whether the parameter is meaningful for the current session: `ALLOW` means it can be iterated, `FORBID` that it can not, `ALREADYITERATED` that is has already been selected for iteration during the current session, `IRRELEVANT` means that the parameter is not meaningful for the current session so that it should not be considered by the program.
If the parameter is selected for iteration (input stage), its address is stored in the first field of an object of type Iterator which is designed to keep all the information relevant to the iteration of this parameter. It will also store the minimum, maximum value of the iteration and the sampling size. The type Iterator is the following:

```
typedef struct Iterator{
    VAR*    Location;
    VAR     Min;
    VAR     Max;
    VAR     Default;
    int         StepNumber;
} Iterator;
```

The field `Default` will keep the initial value of the parameter (which is defined within the `Init` function of the object it belongs to) in order to reset the parameter at the end of the session. The maximum value for the field `StepNumber` is (in optype.h):

```
#define MAX_ITER 1000
```

The function (in var.c)

```
int LowerVar(int user,VAR *x, VAR*y);
```

allows the checking of consistency between the minimum and maximum values set by the user. During the current session the list of all the parameters selected for iteration is stored in an object of type Planning :

```
typedef struct Planning{
  Iterator    Par[MAX_ITERATOR];
  int         VarNumber;
  char        Action;
  int         NumberOfMethods;
} Planning;
```

where `MAX_ITERATOR` is set in optype.h:

```
#define MAX_ITERATOR 3
```

and `VarNumber` keeps the number of parameters currently selected for iteration. The flag `NumberOfMethods` stores the index of the method at hand in case of a comparison between pricing methods, the flag `Action` is set to `TOSCREEN` during the input stage, its other possible values are `TOFILE` and `TOSCREENANDFILE`, `NAMEONLYTOFILE`, `VALUEONLYTOFILE` for the output stage. If a parameter is selected for iteration, its field `Viter` will be set to the index of the `Iterator` array of the object `Planning`. This way there is a two-ways channel between this object and the parameter at hand. This is also the reason why the values of `ALLOW`,..., are negative.
The function (in var.c):

```
void NextValue(int count,Iterator* pt_iterator)
```

deals with the iteration operation of the parameter corresponding to the `*pt_iterator` object.
Other utilities (in var.c) are devoted to the management of a `Planning` object:

```
void ResetPlanning(Planning *pt_plan);
void ShowPlanning(int user,Planning *pt_plan);
void ShrinkPlanning(int index,Planning*pt_plan);
int ChkStepNumber(int user,Iterator *pt_iterator,int step);
```

4.1 Input-Output of a VAR

The following functions manage the input-output of a VAR:

```
int Fprintf(int user,const char *s,...);
int FprintfVar(int user,const char s[],VAR *x);
int PrintVar(Planning *pt_plan,int user,VAR*);
int ScanVar(Planning *pt_plan,int user,VAR*);
int ChkVar(Planning *pt_plan,VAR *x);
int GetParVar(Planning *pt_plan,int user,VAR *x);
int ShowParVar(Planning *pt_plan,int user,VAR *x);
int ChkParVar(Planning *pt_plan,VAR *x);
```

`Fprintf` is only a redefinition of `fprintf` which takes into account the `user` flag we discussed above (`TOSCREEN`,...).`FprintfVar` does what you think it does. `PrintVar` is more elaborated since it displays in a suitable way the fields of the adequate `Iterator` object in case the VAR has been selected for iteration. `ScanVar` displays the value of a VAR and prompt the user either

to agree with the current value, modify it or (if possible) iterate. `ChkVar` checks that the current value of a VAR pertains to its Vtype: this allows the definition of constrained types (the range $[0, 1]$ for a correlation factor for instance). Lastly the `..ParVar` functions are list versions of the above ones.

4.2 Initialization

One of the main interest of the VAR system is to allow the definition of new types (see above). Each type should be indexed by a flag in optype.h:

```
/*Vtype*/
#define FIRSTLEVEL 20

/*FirstClass*/
#define END 0
#define INT 1
#define DOUBLE 2
#define LONG 3
#define PDOUBLE 4
#define DATE 5
#define RGDOUBLE 6
#define BOOL 7
#define PADE 8
#define RGDOUBLE12 9
#define INT2 10
#define RGINT13 11
#define SPDOUBLE 12
#define RGDOUBLE051 13
#define GENER 14
#define RGDOUBLE14 15

/*SecondClass*/
#define NUMFUNC_1 20
#define NUMFUNC_2 21
#define PTVAR 22
#define DOUBLEARRAY 23
/*This last should be less than MAX_TYPE:*/

#define MAX_TYPE 30
```

`PDOUBLE`, for instance, is intended for strictly positive double. The maximum of user-defined types in this way is `MAX_TYPE`. Some VAR does not store their

values (for instance in case of arrays of doubles) directly in the field Val, but in another field of a structure to which the field Val should point: they are called second-level VARs, of course they require a peculiar treatment in the above Input-Output routines.

The 3 arrays (in var.c)

```
static char **formatV;
int **true_typeV;
static char **error_msgV;
```

WHICH MUST BE INITIALIZED BEFORE ANY USE OF THE VAR SYSTEM by the function

```
int InitVar(void);
```

with the corresponding desallocation function:

```
int ExitVar(void);
```

allow to deal with the various types at the input and output stages. `formatV` contains the formatting string of the 'true type' behind Vtype, which is itself in the array `true_typeV`. The values for the type `PDOUBLE` for instance are:

```
formatV[PDOUBLE]="%lf";
true_typeV[PDOUBLE]=DOUBLE;
error_msgV[PDOUBLE]="Should be greater than 0!";
```

The last thing to do to implement the type `PDOUBLE` is to write the adequate check in `ChkVar`:

```
case PDOUBLE:
    status=(x->Val.V_PDOUBLE<0.); /* PDOUBLE>=0.*/
break;
```

5 Numerical functions

Numerical functions are implemented through the types (in optype.h):

```
/*NumericalFunctions*/
typedef struct NumFunc_1{
    double          (*Compute)(VAR*,double);
    VAR             Par[MAX_PAR];
    int             (*Check)(int user,Planning*,void*);
} NumFunc_1;
```

```
typedef struct NumFunc_2{
    double          (*Compute)(VAR*,double,double);
    VAR             Par[MAX_PAR];
    int             (*Check)(int user,Planning*,void*);
} NumFunc_2;
```

where the suffix _1 or _2 correspond to the number of arguments of the *Compute field after the parameter VAR* which should be the field Par of the structure. Typically the payoffs of one-dimensional options will be implemented as NumFunc_1, those of two-dimensional or path-dependent claims as NumFunc_2. The field Par stores some parameters upon which the function may depend, like the strike for a Call or a Put payoff. The field Check is intended to check possible constrains between the parameters of Par: for instance the strikes $K1$ and $K2$ of a CallSpread should verify $K1 \leq K2$.

The field *Compute may point to a function of the file numfunc.c in the directory Common . The check function in the same way may be one of the functions of chk.c.
The implementation of a CallSpread payoff may thus look like (file callspreadeuro.c in Opt/Std):

```
static NumFunc_1 callspread=
    {
    CallSpread,
        {{"Strike 1",PDOUBLE,100,ALLOW},
        {"Strike 2",PDOUBLE,110,ALLOW},
        {" ",END,0,FORBID}
        },
    CHK_callspread
    };
```

where the function CallSpread is defined in numfunc.c:

```
double CallSpread(VAR *param,double spot)
    {
    double strike1=(*param).Val.V_PDOUBLE,strike2=(*(param+1)).Val.V_PDOUBLE;

    return MAX(0.,spot-strike1)-MAX(0.,spot-strike2);
    }
```

and also the function CHK_callspread in chk.c:

```
int CHK_callspread(int user, Planning *pt_plan,void* dum)
```

```
    {
    NumFunc_1* payoff=(NumFunc_1*)dum;
    int status=OK;

    status+=ChkParVar(pt_plan,payoff->Par);
    if (payoff->Par[1].Val.V_PDOUBLE<payoff->Par[0].Val.V_PDOUBLE)
        {
        Fprintf(TOSCREENANDFILE,"%s: lower than %s\n",
            payoff->Par[1].Vname,payoff->Par[0].Vname);
        status+=1;
        }

    return status;
    }
```

6 Computation time information

In order to get a basic information regarding the computation time of a routine, we have designed the following structure:

```
/*Time Info*/
typedef struct TimeInfo{
    Label   Name;
    VAR     Par[MAX_PAR];
    VAR     Res[MAX_PAR];
    int     (*Check)(int user, Planning *, struct TimeInfo *);
    int     (*Init)(struct TimeInfo*);
} TimeInfo;
```

There is a single instance of this type which is in timeinfo.c in Common :

```
TimeInfo computation_time_info=
{
    "No Computation Time Information",
    {{"Choice",INT,100,FORBID},
     {"AveragingTimeWidth",INT,100,FORBID},
     {"NumberOfRuns",LONG,100,FORBID},
     {" ",END,0,FORBID}},
    { {"MeanTime(ms)",DOUBLE,100,FORBID},
      {" ",END,0,FORBID}},
    Chk_TimeInfo_OK,
```

```
    Init
};
```

The corresponding Get, Show, Check and initialization function are in the same file.

Since many processes may be at work at the same time on your computer it is not that easy to get a meaningful result by a plain call to some kind of `difftime` function. A more robust result is obtained by averaging over several trials. The number of trials is the parameter AveragingTimeWidth of the `Par` field. Lastly the computation time of a given routine may be smaller than the unit of time measurement of the computer (typically for a closed formula). In such a case you will get a nil computation time regardless of the averaging procedure. So it may be necessary to measure a big enough number of trials instead of a single one as a elementary input to the averaging stuff. This is the parameter NumberOfRuns. The corresponding code, in the function `Action(...)` of the file tools.c is the following:

```
if (pt_time_info->Par[0].Val.V_INT==OK)
{
averaging=pt_time_info->Par[1].Val.V_INT;
number_of_runs=pt_time_info->Par[2].Val.V_INT;
diff_time=0.;
for (i=0;i<averaging;i++)
{
start=clock();
for (j=0;j<number_of_runs;j++)
    error=(pt_method->Compute)(pt_option->TypeOpt,
        pt_model->TypeModel,pt_method);
finish=clock();
diff_time+=((double)finish-(double)start)/(double)CLOCKS_PER_SEC;
}
pt_time_info->Res[0].Val.V_DOUBLE=diff_time/(double)averaging;
}
```