C++ Project - Monte Carlo Pricing Engine

C-A.Gourio, G.Ounnoughene, A.Bourdon, T.Cadiou

May 31, 2024

1 Project purpose

The aim of the project is to implement an option pricer in C++. More specifically, *c_pricer* is a pricing and GSE library comprising:

- Several models (Black & Scholes, Dupire, Heston)
- Two path simulators (Euler & Brodie Kaya)
- Several pricing possibilities (American & European call/put, etc…)
- A calibration protocol

2 Installation

As the development of the project straddles several platforms, the installation method uses the *CMake* technology. *Cmake* is available on Micorsoft Windows, MacOsX and Linux. Moreover, a *CMake* project is compatible with *Microsoft Visual Studio*, *Visual Studio Code* as well as a simple *Terminal* (Best solution !).

2.1 Install a compiler and cmake (Linux & MacOs)

On a *terminal* window, run

```
$ sudo apt-get upgrade -y
$ sudo apt-get install build-essential cmake wget
```

With *CMake* and a compiler, you are now ready to code ! A quick check to ensure that everything works :

```
$ cmake --version
cmake version 3.28.3

CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

(continues on next page)

(continued from previous page)

```
$ gcc --version
gcc (Ubuntu 13.2.0-23ubuntu4) 13.2.0
Copyright (C) 2023 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

2.2 Install *Eigen*

Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.

Download Eigen

```
$ wget https://gitlab.com/libeigen/eigen/-/archive/3.4.0/eigen-3.4.0.tar.gz

--2024-05-12 22:50:07--  https://gitlab.com/libeigen/eigen/-/archive/3.4.0/eigen-3.4.0.tar.gz
Resolving gitlab.com (gitlab.com)... 2606:4700:90:0:f22e:fbec:5bed:a9b9, 172.65.251.78
Connecting to gitlab.com (gitlab.com)|2606:4700:90:0:f22e:fbec:5bed:a9b9|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [application/octet-stream]
Saving to: ‘eigen-3.4.0.tar.gz’

2024-05-12 22:50:08 (14.8 MB/s) - ‘eigen-3.4.0.tar.gz’ saved [2705005]
```

Extract the archive

```
$ tar -xf eigen-3.4.0.tar.gz
$ rm eigen-3.4.0.tar.gz
$ cd eigen-3.4.0
```

Move the library in the right place

```
$ sudo cp -r Eigen/ /usr/local/include/.
```

Note: In most linux systems, C++ libraries are located in */usr/local*. *include* folder stands for header files (Like the one we have just copied). As for *bin* or *lib* folders, they stored binaries/executables of every libraries (For instance… *c_pricer.a*)

Note: One can find more information on *Eigen* library here

2.3 Install *Google Test* (Gtest)

Googletest helps us to write C++ tests. Independent and Repeatable: Googletest isolates the tests by running each of them on a different object.

Download Gtest

Is is easy, just download it from the internet.

```
$ sudo apt-get install libgtest-dev
```

Build the library

The library has been saved in */usr/src*, we should configure it with cmake and then build it. As everything is done *next to* the root of the system, many action requires admin rights.

```
$ cd /usr/src/gtest
$ sudo cmake CMakelist.txt

-- Configuring done (0.1s)
-- Generating done (0.0s)
-- Build files have been written to: /usr/src/googletest/googletest

$ sudo make

[ 50%] Built target gtest
[100%] Built target gtest_main
```

Put the library in the right place

Like *Eigen*, one may move binaries and headers into the right place */usr/local/...*. We start in the *gtest* folder (*/usr/src/gtest*)

```
# Copy the binaries
$ cd lib
$ sudo cp *.a /usr/local/lib.

$ cd ../include
$ sudo cp -r gtest/ /usr/local/include/.
```

Note: One can find more information on *Gtest* library here You can also run *make setup* command to install the library.

2.4 Test and dry run

It is now time to test the compilation of the pricer !

Clone the project

Following your git installation, you should be able to clone the project with this command line:

```
$ git clone https://github.com/Charles-Auguste/c_pricer.git

Cloning into 'c_pricer'...
remote: Enumerating objects: 386, done.
remote: Counting objects: 100% (310/310), done.
remote: Compressing objects: 100% (212/212), done.
remote: Total 386 (delta 160), reused 232 (delta 98), pack-reused 76
Receiving objects: 100% (386/386), 1.43 MiB | 2.94 MiB/s, done.
Resolving deltas: 100% (177/177), done.
```

If it failed, just download a *.zip* extract of the project and place it wherever you like.

You can now open the project in visual studio code.

Dev tool kit

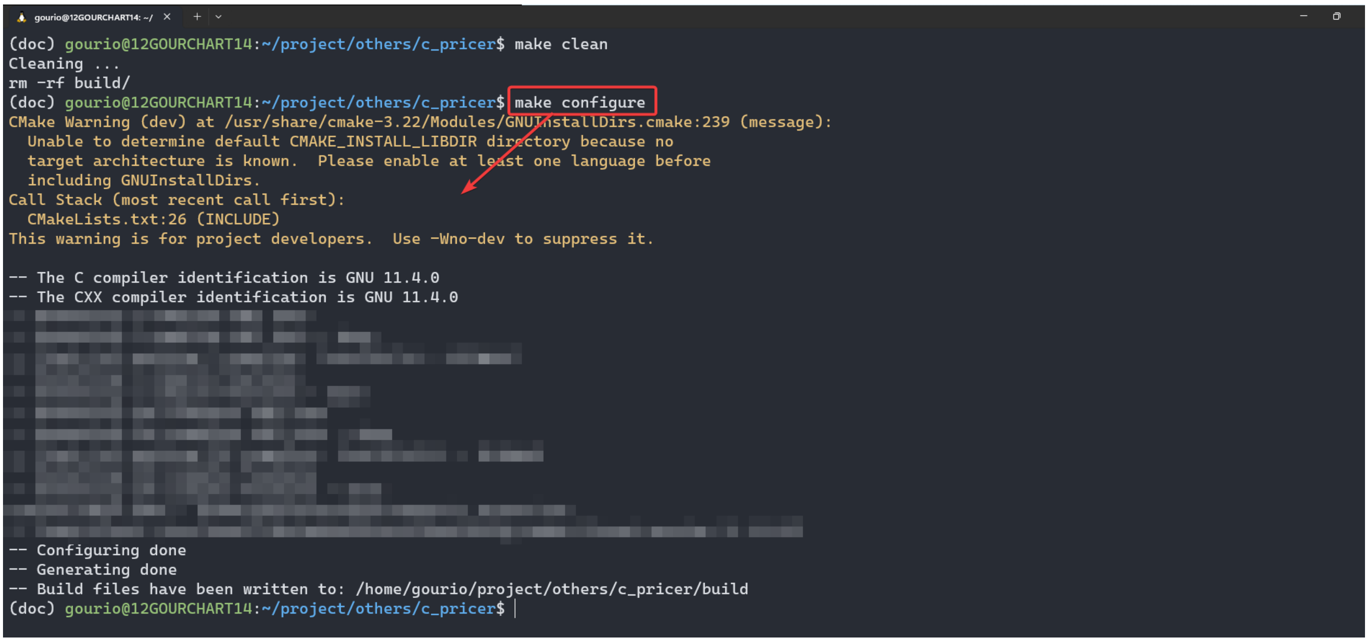
A *Makefile* is available at the root of the projet. It allows us to easily configure, build, clean and format the library.

Note: Makefile is a way of automating software building procedure and other complex tasks with dependencies. • Makefile contains: dependency rules, macros and suffix(or implicit) rules.

To configure the project, simply run:

```
$ make configure

CMake Warning (dev) at /usr/share/cmake-3.28/Modules/GNUInstallDirs.cmake:243 (message):
Unable to determine default CMAKE_INSTALL_LIBDIR directory because no
target architecture is known.  Please enable at least one language before
including GNUInstallDirs.
Call Stack (most recent call first):
  CMakeLists.txt:26 (INCLUDE)
This warning is for project developers.  Use -Wno-dev to suppress it.

Library Base path : /home/charles-auguste/c_pricer/src
-- Configuring done (0.1s)
-- Generating done (0.0s)
-- Build files have been written to: /home/charles-auguste/c_pricer/build
```

As you can see, the building configuration has been saved to */build* folder. Now one can esasily compile the library.



Note: *$ make clean* instruction kindly remove the build directory (and the *CMakeCache.txt* file). It is usefull if some changes has been done to the *CMakeList.txt* file.

To compile the library run:

```
$ make build_project

Building ...
gmake[3]: Entering directory '/home/charles-auguste/c_pricer/build'
[  5%] Building CXX object CMakeFiles/pricer.dir/src/ThomasSolver/ThomasSolver.cpp.o
...
[ 47%] Linking CXX static library libpricer.a
gmake[3]: Leaving directory '/home/charles-auguste/c_pricer/build'
[ 47%] Built target pricer
```

(continues on next page)

(continued from previous page)

```
gmake[3]: Entering directory '/home/charles-auguste/c_pricer/build'
gmake[3]: Leaving directory '/home/charles-auguste/c_pricer/build'
gmake[3]: Entering directory '/home/charles-auguste/c_pricer/build'
[ 52%] Building CXX object CMakeFiles/test.dir/main.cpp.o
...
[100%] Linking CXX executable test
gmake[3]: Leaving directory '/home/charles-auguste/c_pricer/build'
[100%] Built target test
gmake[2]: Leaving directory '/home/charles-auguste/c_pricer/build'
gmake[1]: Leaving directory '/home/charles-auguste/c_pricer/build'
```

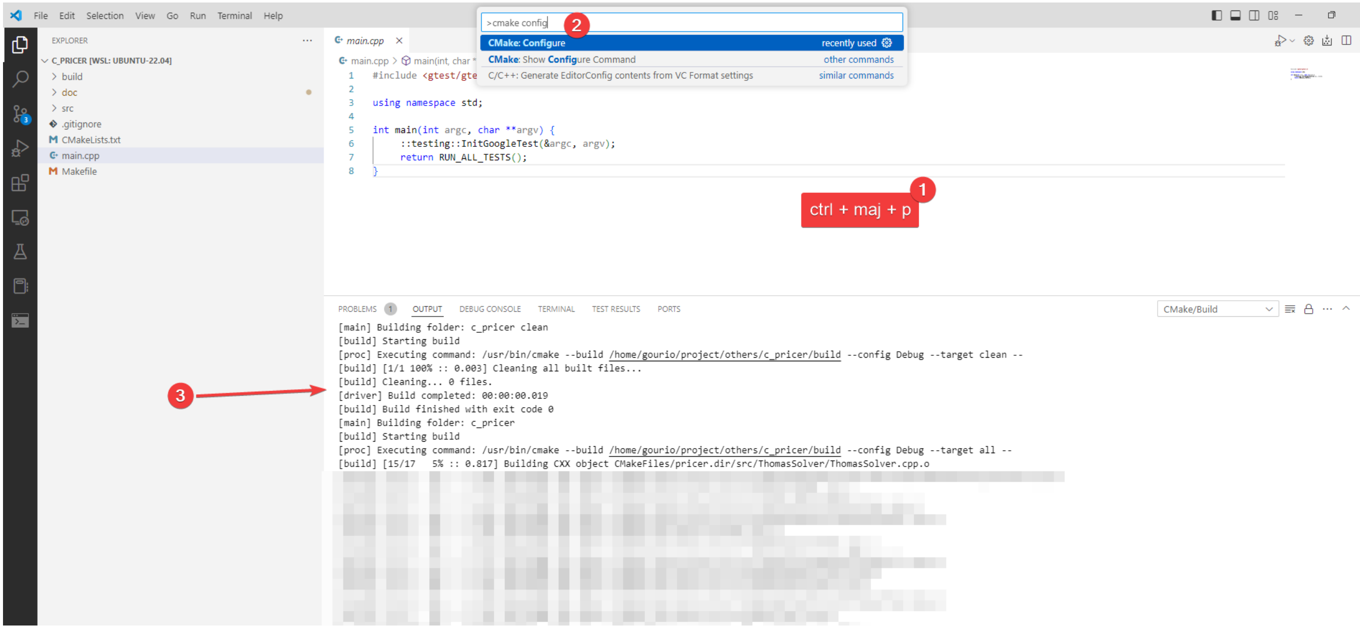
To lauch the test procedure run:

```
$ make test
```

Note: This last instruction run the whole test procedure. With visual Studion (Code), the Gtest framework allows us to launch each test individually.

On visual Studio Code

All these steps can also be done one *Visual studio code*. With the shortcut *Ctrl* + *Maj* + *P*, you can configure, build and clean rebuild the project !!



3 Architecture and project key points

3.1 Global structure description

The project architectue is as follow. At the root, a *doc* folder contains all the documentation elements (*Sphinx* & *Python*). The *src* folder contains all the submodules of our library. We will go deeper into them later. Finally , *Makefile* and *CMakeLists.txt* are some kind of configuration file for the compilation.

```
$ tree

.
├── CMakeLists.txt
├── doc
│   └── ...
```

(continues on next page)

(continued from previous page)

```
├── main.cpp
├── Makefile
└── src
    ├── Calibration
    │   ├── Calibration.cpp
    │   ├── Calibration.h
    │   └── test.cpp
    ├── ImpliedVolatilitySurface
    │   └── ...
    ├── Model
    │   └── ...
    ├── MonteCarloEngine
    │   └── ...
    ├── PathSimulator
    │   └── ...
    ├── Pricing
    │   └── ...
    └── ThomasSolver
        └── ...
```

CMakeList.txt

Let's take a look at the *CMakeList.txt* file and analyse it.

Settings and parameters

```
# ----> We tell Cmake which version to use (here 3.22)
cmake_minimum_required(VERSION 3.22)
# ----> We also set C++ standard
set(CMAKE_CXX_STANDARD 20)

if(POLICY CMP0079 )
  cmake_policy(SET CMP0079 NEW)
endif()

# ----> We define some global variables that addresses
# ----> project name and description
SET(PROJECT_NAME "C++ pricer")
SET(BINARY_NAME "pricer")
SET(PROJECT_DESCRIPTION "C++ project - ENPC - Master MFD")

INCLUDE(GNUInstallDirs)

# ----> Important, here we officially defined the project !
project("${PROJECT_NAME}" DESCRIPTION "${PROJECT_DESCRIPTION}")

# ----> We define a global variable of the code folder (src)
SET(LIBRARY_BASE_PATH "${PROJECT_SOURCE_DIR}/src")
message("Library Base path : " ${LIBRARY_BASE_PATH})

# ----> And tell CMake to include this folder
# ----> This helps a lot when including header files, it
# ----> avoids this kind of horror: "../../test.h"
INCLUDE_DIRECTORIES (
  "${LIBRARY_BASE_PATH}"
)
```

Files to compile …

```
# ----> We define global variables that contains the
# ----> list of file to compile in each category

# ----> Firstly source files (.cpp)
```

(continues on next page)

(continued from previous page)

```
SET(PUBLIC_SOURCES_FILES
  "${LIBRARY_BASE_PATH}/MonteCarloEngine/MonteCarlo.cpp"
  ...
  "${LIBRARY_BASE_PATH}/Pricing/Pricing.cpp"
)

# ----> Secondly header files (.h, .hpp)
SET(PUBLIC_HEADERS_FILES
  "${LIBRARY_BASE_PATH}/MonteCarloEngine/MonteCarlo.h"
  ...
  "${LIBRARY_BASE_PATH}/Pricing/Pricing.h"
)

# ----> Finally test files (test.cpp)
set(TEST_SOURCE_FILES
  "${LIBRARY_BASE_PATH}/MonteCarloEngine/test.cpp"
  ...
  "${LIBRARY_BASE_PATH}/Calibration/test.cpp"
)
```

Building the library

```
# ----> Eventually, we define our library
# ----> with its source and header files
ADD_LIBRARY (
  ${BINARY_NAME} STATIC "${PUBLIC_HEADERS_FILES}" "${PUBLIC_SOURCES_FILES}"
)

# ----> And some properties of our future library
SET_TARGET_PROPERTIES (
  ${BINARY_NAME} PROPERTIES
  VERSION             ${LIBRARY_VERSION_STRING}
  SOVERSION    ${LIBRARY_VERSION_MAJOR}
  PUBLIC_HEADER   "${PUBLIC_HEADERS_FILES}"
    LINKER_LANGUAGE CXX
)
```

Building the executable

```
# ----> We look for Gtest in the system
find_package(GTest REQUIRED)
include_directories(${GTEST_INCLUDE_DIRS})

# ----> Eventually, we define our executable
# ----> with its source and header files
add_executable(test main.cpp
    "${TEST_SOURCE_FILES}"
    "${PUBLIC_HEADERS_FILES}"
)

# ----> And we link it with our library and gtest
target_link_libraries(test PUBLIC ${BINARY_NAME} ${GTEST_LIBRARIES} pthread)

set_target_properties(test PROPERTIES LINKER_LANGUAGE CXX)
```

Dependencies (Eigen)

```
# ----> We add Eigen (header only library) folder to our project
if (EXISTS "${EIGEN_PATH_INCLUDE}/Eigen")
  include_directories("${EIGEN_PATH_INCLUDE}")
else()
  message("Cannot find Eigen. Make sure you install it correctly")
endif()
```

That's it. Now and to summarize, the purpose of this file is to give building instruction to our compiler. Our project is building two things:

- A pricing library and its header files (*libpricer.a*). This library is static and can be used in other C++ projects.
- A testing executable (*test*) that is used to test our library.

Tests

A submodule (like *Calibration*) is actually composed of at least three elements:

- A header file
- A source file (following the name of header file)
- A test file which is alway named *test.cpp*

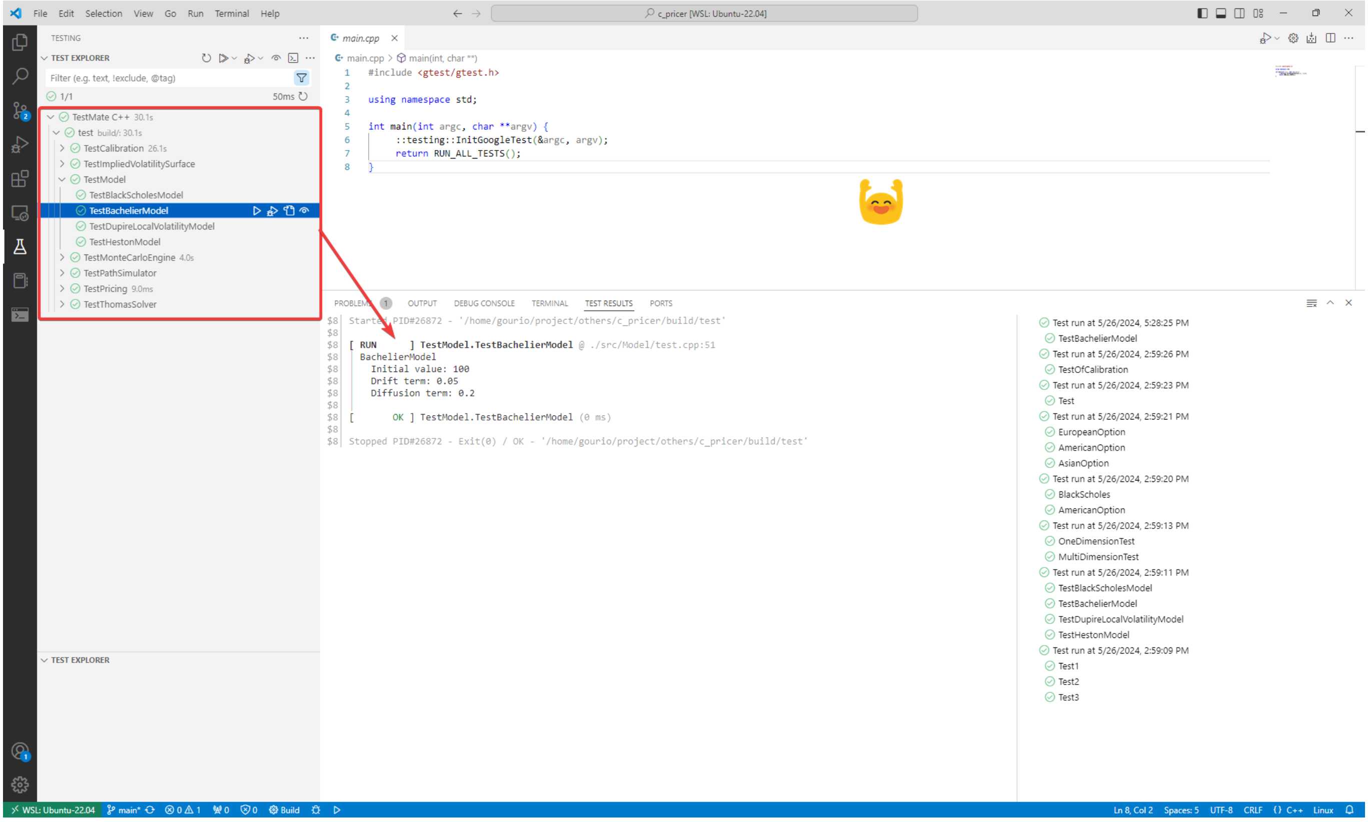
The test file is here to check whether or not submodule components works. This file is designed using *Gtest* framework.

```cpp
#include "ThomasSolver.h"
#include <gtest/gtest.h>

// We define a namespace for our tests
// Then we define a test within the namespace with the name "test"
namespace TestThomasSolver {
TEST(TestThomasSolver, Test) {
  cout << "Test for Thomas solver" << endl;
  cout << "----------------------" << endl << endl;
  Vector central_diagonal{1, 1, 1, 1};
  Vector lower_diagonal{0, 0, 0, 0};
  Vector upper_diagonal{0, 0, 0, 0};
  Vector right_side{1, 1, 1, 1};

  ThomasSolver solver_instance = ThomasSolver(lower_diagonal, central_diagonal,
                                              upper_diagonal, right_side);
  Vector result = solver_instance.solve();
  for (size_t k = 0; k < result.size(); k++) {
    cout << "Result index " << k << ": \t" << result[k] << endl;
  };
  // A test always end with at least one assertion
  // the assertion represents the test itself
  ASSERT_TRUE( result[0] == 1.);
}
}; // namespace TestThomasSolver
```

To run all the test, one can execute the following command define in the *Makefile*:

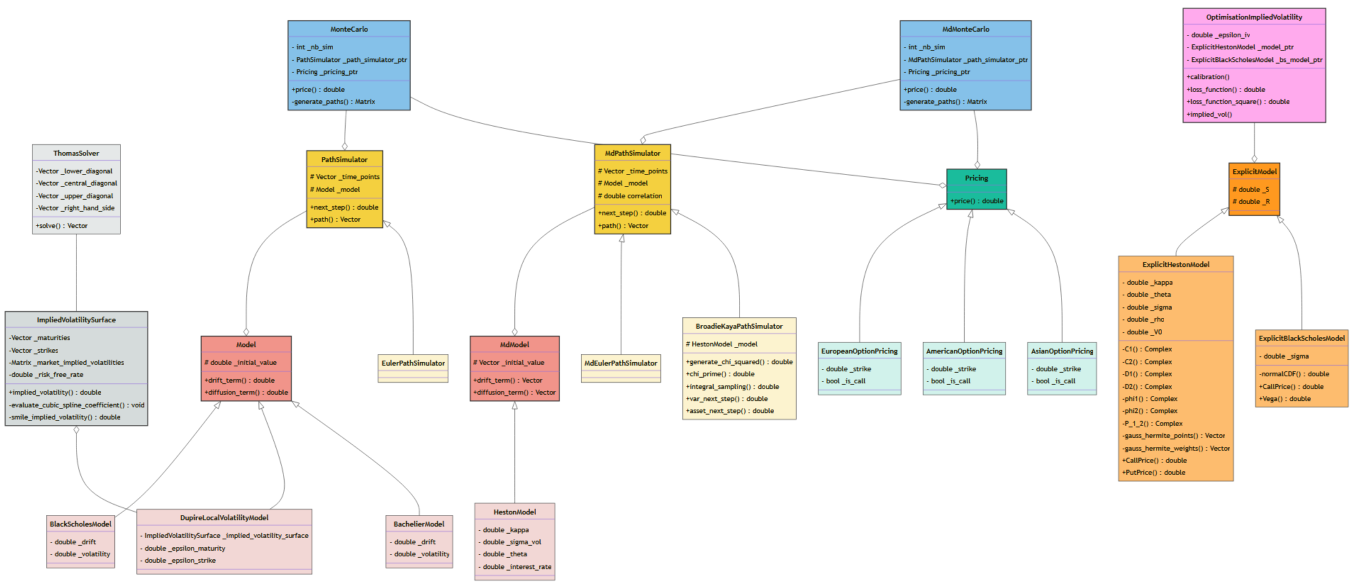
```
$ make test
```

Moreover, visual studio code allows us to execute test individually (with the *Gtest* framework).



Classes and objects

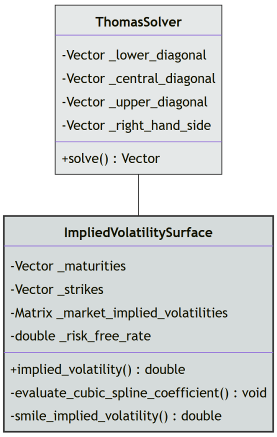
A good way to understand our library is by taking a closer look at the class diagramm



3.2 Modules and Submodules

Thomas Solver & Implied Volatility Surface

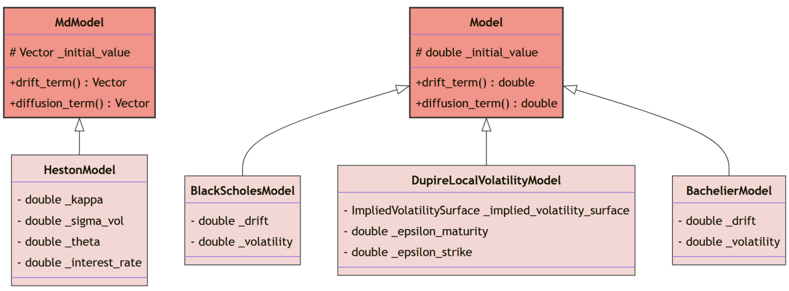
Thomas solver module is used in *ImpliedVolatilitySurface* class. Thanks to it, one can interpolate a finite volatility surface into a function that return a volatility for all maturity and strike.



Using *ImpliedVolatilitySurface* is easy:

```
// first we set the risk free rate
double risk_free_rate = 0.03;
// We define a vector of Strikes and maturities
Vector strikes{90, 100, 110, 120};
Vector maturities{0.5, 1, 1.5, 2};
// And its volatility surface Matrix (not define here)
Matrix market_volatilities; // Matrix stands for vector<vector<double>>

// We create an instance of the class
ImpliedVolatilitySurface surface_instance = ImpliedVolatilitySurface(
  maturities, strikes,
  market_volatilities, risk_free_rate
);

// And that's all, we can now get a volatility
double volatility = surface_instance.implied_volatility(
  maturity = 0.75, strike = 105
);
```

Model



Model is one of the main point of our library. This class represents a **one dimension** model. (On the contrary *MdModel* defines multi dimensionnal models). Our library defines several model child classes:

- BlackScholesModel
- BachelierModel

- DupireLocalVolatilityModel
- HestonModel

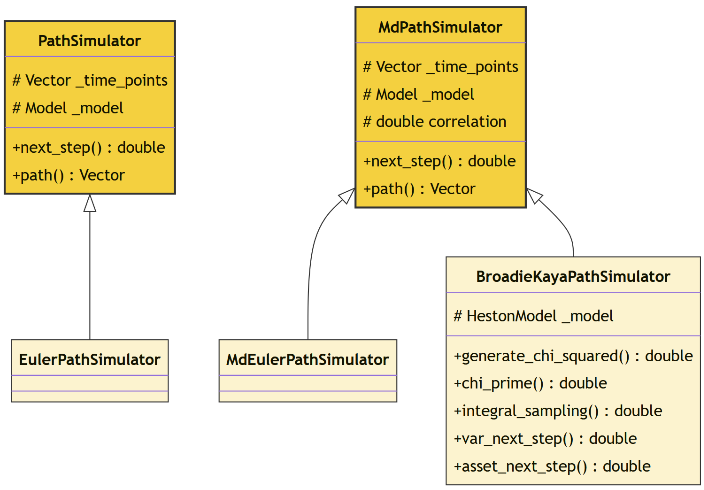
Using a model is simple:

```
// For Black & Scholes Model
// We define an initial value as well as a drift parameter and volatility
double init_value = 100;
double drift = 0.05;
double volatility = 0.2;

// We create an instance of the object
BlackScholesModel bs_model = BlackScholesModel(init_value, drift, volatility);

// For a given time and value pair, the model class return the value of th edrift term
// and the diffusion term
double _drift_term = ba_model.drift_term(1, 115);
double _diffusion_term = bs_model.diffusion_term(1, 115);
```

Note: Distinction between one dimensional models and multi dimensionnal models is a major weakness in our library as it introduces too much complexity. In a future update, the class *Model* will be deprecated and replace with *MdModel*

Path Simulator

The next step of our library are *PathSimulator* and *MdPathSimulator* classes. With a model and some parameters, those classes return a path for our diffusion.



Here is how it works with *MdPathSimulator*:

```
// First we define some parameters
double init_value = 100; // Initial spot price
// For heston
double r = 0.0319;      // Risk-free rate
double v_0 = 0.010201; // Initial volatility
double rho = -0.7;      // Correlation of asset and volatility
double kappa = 6.21;    // Mean-reversion rate
double theta = 0.019;   // Long run average volatility
double xi = 0.61;       // "Vol of vol"
// For simulation
double T = 1.00; // One year until expiry
double nb_plot = 1000;

// We create a model, and a MdPathSimulator instance
Vector init_values_heston{init_value, v_0};
```

(continues on next page)

(continued from previous page)

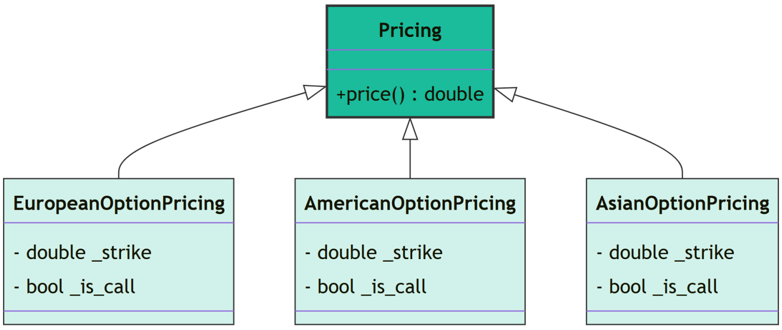
```
HestonModel he_model(init_values_heston, kappa, xi, theta, r);

MdEulerPathSimulator he_path =
    MdEulerPathSimulator(&he_model, T, nb_plot, rho);
Vector he_path = he_path.path();
```

Pricing

Pricing class is a kind of Payoff class. It takes a matrix as input and return the expectation of the payoff. In our library, three types of payoff are available:

- European
- American
- Asian

Both payoff is available with *CALL* and *PUT*



How to use the pricing class:

```
// Define the type of price (call or put)
CALL_PUT call = CALL_PUT::CALL;

// Define a strike, a free-risk rate and a maturity
double r = 0.03;
int T = 1;
double K = 110;

// Create an instance of the class
EuropeanOptionPricing eu_call = EuropeanOptionPricing(K, call, r, T);

// For a given matrix P (vector<vector<double>>), you get the price of the call
double call_price = eu_call.price(P)
```

Calibration

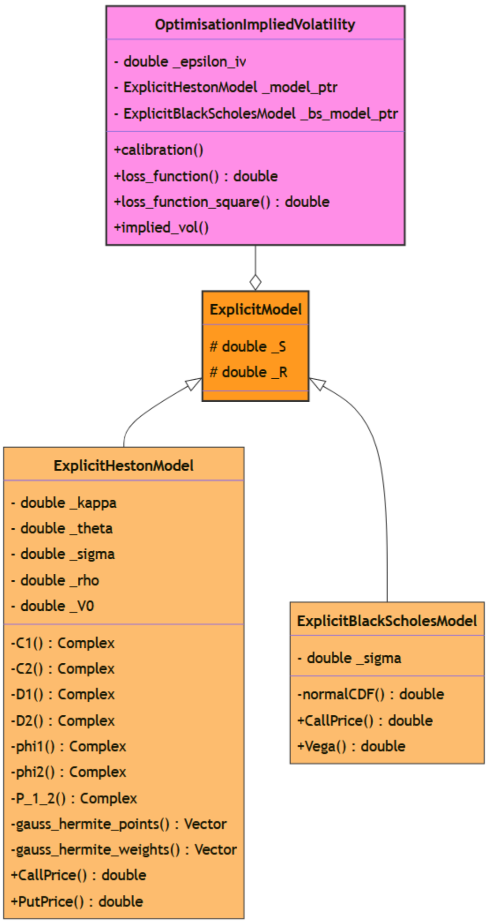
This class was designed to calibrate an Heston model. It is indeed a very specific class. Yet, the library also defined *ExplicitModel* classes. Those classes use closed formula to compute the price of a call or a put. This is how to use them:

```
// Define some parameters
double S0 = 100; // Initial Spot
double r = 0.05; // Risk free rate

double kappa = 0.5; // mean return
double theta = 1.;
double sigma = 0.1; // Volatility of volatility
double rho = 0.5;   // correlation
double v0 = 0;      // initial variance
```

(continues on next page)

(continued from previous page)

```
double K;
double T;

// Create excplicit models
ExplicitHestonModel model_he(kappa, theta, sigma, rho, v0, S0, r);
ExplicitBlackScholesModel model_bs(S0, r, 0.5);

// Use them !
double call_price_bs = model_bs.CallPrice(K, T);
double call_price_he = model_he.CallPrice(K, T);
```



ExplicitModel are used to calibrate a Heston model. The class *OptimisationImpliedVolatility* performs such optimisation. With a matrix of implied volatility as input, one can get optimal parameters for a model. We defined two types of loss function. The first one is based on L1 loss over call prices and the second one over computed volatility. This is how to calibrate a model:

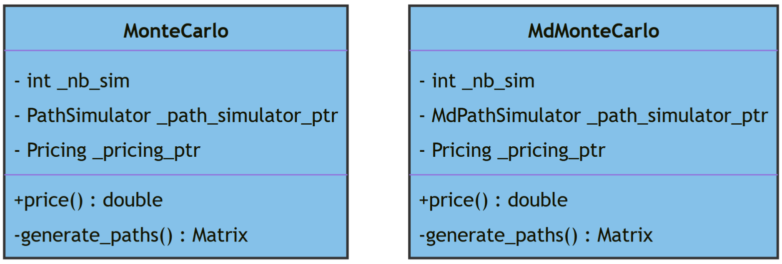
```
// We define a volatility surface
Matrix IV_surface = {
    {0., 20., 40., 60., 80., 100., 120., 140., 160., 180.},
    {0.25, 0.39, 0.31, 0.24, 0.22, 0.16, 0.19, 0.23, 0.29, 0.38},
    {0.5, 0.44, 0.36, 0.27, 0.21, 0.17, 0.21, 0.27, 0.35, 0.4},
    {0.75, 0.45, 0.3, 0.25, 0.21, 0.18, 0.22, 0.29, 0.37, 0.45},
    {1., 0.48, 0.42, 0.34, 0.28, 0.2, 0.26, 0.31, 0.42, 0.5},
    {2., 0.52, 0.43, 0.34, 0.26, 0.21, 0.27, 0.38, 0.45, 0.55},
    {3., 0.54, 0.46, 0.34, 0.27, 0.23, 0.28, 0.36, 0.49, 0.58},
    {4., 0.57, 0.5, 0.46, 0.35, 0.25, 0.32, 0.45, 0.54, 0.6},
    {5., 0.6, 0.52, 0.41, 0.31, 0.26, 0.34, 0.4, 0.55, 0.62}};
}

// and some parameters
double epsilon = 0.000001;
```

(continues on next page)

(continued from previous page)

```
// We define explicit models
ExplicitHestonModel model_test(kappa, theta, sigma, rho, v0, S0, r);
ExplicitBlackScholesModel model_bs(S0, r, T);

// And create an instance of calibration class
OptimisationImpliedVolatility optim(epsilon, model_test, model_bs);

// Then, we optimise
optim.calibration(IV_surface, LOSS_FUNCTION::VOL);
```

Monte Carlo Engine



The last submodule of our library is actually the monte carlo engine. This is how to use it:

```
// First ... define some parameters
double S0 = 100.;
double r = 0.03;
double volatility = 0.2;

Vector init_values{100., 0.04};
double theta = 0.04;
double kappa = 1;
double sigma = 0.05;
double rho = -0.7;

double T = 1;
size_t nb_points = 100;

double K = 105;
CALL_PUT type_option = CALL_PUT::CALL;

size_t nb_sims = 1000;

// For one dimension
BlackScholesModel bs_model(S0, r, volatility);
EulerPathSimulator bs_path(&bs_model, T, nb_points);
AsianOptionPricing pricing_as = AsianOptionPricing(K, type_option, r, T);

MonteCarlo bs_simulation_as(nb_sims, bs_path, pricing_as);

double price_as = bs_simulation_as.price();

// For two dimensions
HestonModel he_model(init_values, kappa, sigma, theta, r);
MdEulerPathSimulator he_path(&he_model, T, nb_points, rho);
EuropeanOptionPricing pricing_eu = EuropeanOptionPricing(K, type_option, r, T);

MdMonteCarlo he_simulation_eu(nb_sims, he_path, pricing_eu);

double price_eu = he_simulation_eu.price();
```