Foundations of a Risk Management System

FE5226 Class Project

Fabio Cannizzo

2017-9-1
version 1.1

Abstract

This source code constitutes the foundation of a risk management system. It can load trades from a database, compute the price of trades and their sensitivities with respect to risk factors and construct market objects on demand.

A real risk management system entails of many more features, e.g. sophisticate market objects and pricing models; efficient algorithms to avoid unnecessary recomputations when calculating Greeks; support for a large number of trade types; extensive set of tools to orchestrate recomputations and post-process their results (e.g. P&L explain, VaR, PFE, XVA); tools to manage life-cycle of trades; connections to external databases and systems of various nature; client-server pricing APIs; connections to cash flow management systems and payment systems; support for parallel computations (thread safety).

All these features can be implemented as modifications or extensions of the code provided. You are supposed to carry out various improvements and extensions as specified below.

1 License

The license for the source code is in the main directory. Please read the license file.

2 Source code organization

All source code files are in the subdirectory *src*. Source files with names matching the patterns *Demo*.cpp* or *Test*.cpp* contain the *main* routine for separate programs. The other cpp files are library components used by the various programs.

The *src* subdirectory also contains a *Makefile* for *gcc* compilation. If you use Visual Studio, you need to: create a new solution; in the solution add one console project for each cpp file with name *Demo*.cpp* or *Test*.cpp*; in each project also add all other cpp files (except the ones

with name *Demo*.cpp* or *Test*.cpp*), so that in each project you have one and only one cpp file containing the function *main*.

A number of txt files are given. *portfolio_0.txt* and *risk_factors_0.txt* work with the program in its current state. *portfolio_i.txt* and *risk_factors_i.txt* work after completion of the first i tasks. If a file with appropriate numeration is missing, you can use the last one available (e.g. in task 11, you can use *risk_factors_5.txt*)

To run the program type from the command line:
DemoRisk -p portfolio.txt -f risk_factors.txt

To run the program directly from inside Visual Studio, you can specify the working directory where your txt files are located and the command line arguments (*portfolio.txt risk_factors.txt*) in the *Debugging* section of the project configuration.

3 What to do

You have to complete sequentially all tasks mentioned in section 5. After completing each task you should test correctness of your work by successfully running the command
DemoRisk portfolio_i.txt risk_factors_i.txt
and comparing the output of your program with the output given in *output_i.txt*. To compare files you can use the free program *WinMerge*, or *sdiff* on Linux. Your goal is to generate an identical output. To compare if files are identical, on Windows you can use WinMerge.

After completing all tasks you need to zip and submit all files with extension *.h and *.cpp. Do not submit any other file (e.g. txt files, project files, object files, executable files).

4 Marking criteria

Marking is going to be based on the following criterias.

- Correctness (60%):
  - I will run your *DemoRisk* with my trade portfolio and market data files and compare your output with mine. This may include expired trades (i.e. trades where the last payment date is before the pricing date, which should generate an error), missing market data (e.g. a missing FX spot), missing fixings.
  - I will run my *TestDate* program against your *Date* class.
  - I will run your *DemoRisk* with my trade portfolio and market data files and compare your output with mine, using a base currency different from USD.
  - I will run your *DemoRisk* with my trade portfolio and market data files and compare your output with mine.
  - I will check the total computation time of your executable against mine for a large portfolio with multiple currencies. Any major performance difference ($>$ 20% slower) implies a sub-optimal

design (inappropriate choice of data structures and/or algorithms).

- Style (40%):
  - correct use of indentation (configure your editor to convert tab to spaces and to use tabs of 4 characters)
  - proper choices of variable and function names
  - appropriate use of source code comments
  - avoid code duplication (do not copy and paste, introduce new functions as appropriate)
  - code robustness (e.g. assert function arguments are valid)
  - code conciseness (do not do in 10 lines what you could do in one line)
  - do not reinvent the weel (use library functions when available)
  - minimal diff: only modify the lines of code strictly necessary, do not insert unnecessary whitespaces (you can use *WinMerge* to view the difference between your files and the original ones, or use *git*, which is excellent for checking differences). I will use git to check differences.

5 Tasks

5.1 Improve the *Date* class

The most common operations performed with the *Date* class when pricing are comparison (e.g. $<$, $==$, $>$) and computation of distance between two dates. The current internal representation of the date class is not optimal for these operations.

Refactor the *Date* class by modifying the current internal representation, which entails of the 4 data members *day*, *month*, *year* and *is_leap*, to a single data member of type *unsigned* with name *serial*, representing the number of days elapsed since 1-Jan-1900. This allows to speed up the operations mentioned above and reduce memory footprint.

Change the serialization format to be based directly on *serial*, so that no extra work is necessary when saving or loading from a file.

When the *Date* class is constructed from the arguments *day*, *month* and *year*, you need to generate the equivalent serial *number*. When converting to a string in calendar format for display to the screen, you need to convert *serial* on-the-fly into day, month and year.

5.2 Write a test for the *Date* class

Complete the program in *TestDate.cpp*, so that it tests the correctness of the *Date* class. It should perform the following tests:

1. Construction of an invalid date should generate an error: generate intentionally 1000 random invalid dates (e.g. 31-Apr-2010) and verify that the *Date* class constructor throws an error (use *try...catch*).

2. Verify that converting a date in calendar format (day, month, year) to serial format and then converting back to calendar format yields the original date. Repeat for all dates in the valid range (1-Jan-1900, 31-Dec-2199).
3. Verify that the serial number generated for 2 contiguous dates are contiguous. For instance 31-Jan-2012 and 1-Feb-2012 are contiguous dates, hence the serial numbers they generate should only differ by 1. Repeat for all pairs of contiguous dates in the valid range (1-Jan-1900, 31-Dec-2199).

At the begin of the test, randomize the random number generator. If the test fails, the program should throw an exception, if it succeeds it should just print the message "SUCCESS".

Print out the random seed you used for the random number generator, so that, if the test fails, a programmer can re-use the same seed and reproduce the error, which is essential to be able to debug it.

5.3 Perfect streaming for type *double*

Currently when a floating point number in double precision is saved to a file it is transformed from IEEE-754 format to a decimal representation with a finite number of decimals in scientific notation. The conversion from IEEE-754 to decimal (when saving) and then back to IEEE-754 (when reading) involves rounding and can cause accuracy loss. In other words, if we start from a value of type *double*, we save it as a *string* in decimal format, then we read back the *string* and convert it to *double*, we may not get back exactly the same value we started with.

To overcome the problem, change the streaming representation of a *double* to the integer interpretation of its binary representation in hexadecimal format with letters in lowercase.

If x is a variable of type *double*, it binary representation can be reinterpreted as a 64 bits integer, which can be represented exactly. To do that, you can either get the memory address to the variable of type *double* and *reinterpret_cast* it to a 64-bit *int* pointer or use a *union*. To make the textual representation more compact, we use base 16 (hexadecimal). For example, the double number -0.15625 should be saved to file as the sequence of 16 characters bfc4000000000000 (see example in *DemoHex.cpp*, which use a *union*). When reading, you need to read the integer number saved in hexadecimal format and re-interpret it as a double.

You need to modify the implementation of the *operator<<* for *double* and implement the overload of the *operator>>* (you may draw inspiration from the ones implemented for the *Date* class).

5.4 Improve the *CurveDiscount* class

The current *CurveDiscount* curve assumes that the interest rate curve is constant.

Modify it so that it takes a set of rates related to different tenors and modify it so that when querying for a discount factor, the value returned is correctly interpolated.

For example, given the data points IR.1W.USD, IR.2W.USD, IR.1M.USD, IR.2M.USD, IR.3M.USD, IR.6M.USD, IR.9M.USD, IR.1Y.USD, IR.18M.USD, IR.2Y.USD, ..., you need to resolve each of them to an actual date with respect to the anchor date, construct a stepwise constant interpolation scheme, modify the function which returns discount factor. For sake of clarity, D, W, M and Y mean respectively days, weeks, months and years. Although this is not consistent with market conventions, for simplicity assumes that 1M=30 days and 1Y=365 days, and ignore the distinction between weekdays and weekends when computing tenor dates.

Since you do not know in advance which points are available in the market data server, if you want all the available points for the currency EUR, you need to modify the market data server to be able to return the array of risk factors with pattern IR.*.EUR. Use the <regex> STL 正则表达式 library to search for a string pattern and implement the method *MarketDataServer::matching*, whose header is already defined in *MarketDataServer.h*. Note that "IR.*.EUR" is not the correct regular expression string to be used here, you need to read about regular expressions and figure out what is the most appropriate string to use, i.e. it should match "just" what needed. For instance, "IR.*EUR" would match "IR.1W.EUR", but also "IRwhateverEUR", which is undesirable.

The interpolation scheme requires computation of the local rates. If r_i and r_{i+1} are the absolute rates for tenor T_i and T_{i+1}, as returned from the market data server, the local rate $r_{i,i+1}$ is the one that solves the following equation:

$$e^{-r_i T_i - r_{i,i+1}(T_{i+1}-T_i)} = e^{-r_{i+1}T_{i+1}}$$

Then the discount factor for any date $t \in [T_i, T_{i+1}]$ is

$$df(t) = e^{-r_i T_i - r_{i,i+1}(t-T_i)}$$

In an efficient implementation as much calculation as possible should be done in the constructor, so that the function *df*, which will be invoked millions of times, is as fast as possible. For instance, the values of the products $r_i T_i$ could be precomputed for all i and cached in a data member of the class.

In the *df* function use binary search (see *lower_bound*) to determine the interval i such that $T_i \leq t < T_{i+1}$.

If the function *df* is called with a date beyond the last tenor or before the anchor date (today), it should generate an error.

5.4.1 PV01 with tenors

The *PV01* risk function needs to be modified into 2 different risk functions: *PV01Parallel* that computes risk with respect to parallel shift of the yield curve (all risk factor move simultaneously); and *PV01Bucketed* that computes risk with respect to individual yield curves (the yield curve for each tenor T_i change, with all the rest remaining constant). In both cases use central differences, with the same bump size as currently defined

in *Demo.cpp*. The 2 new functions must have the same arguments and return type as the existing one.

5.5 Recover from pricing failures

The function *IPricer::price* fails and throws an exception when an error occurs. For instance, if the settlement date of a *Payment* is in the past, pricing of the entire portfolio fails. We would like instead that only problematic trades fail to price while all remaining trades price succesfully.

Modify the typedef *portfolio_values_t* to *vector<pair<double,string>>*. In the *compute_prices* function use *try{}catch{}* to catch exceptions. If there is no error set the double to the price and the string to an empty string. If there is an error set the double to NaN (see header $< limits >$) and the string to the error message embedded in the exception.

When computing thet total for the book (function *portfolio_total*), you should return a pair containing the total for trades which price succesfully and an array of pairs with the index of the trades which price to fail and the associated error message.
$pair < double, vector < pair < size\_t, string >>>$
To identify trades with errors, check if the value is a NaN, not if the error message string is empty, because a trade could potentially fail and still have an empty error message.

When computing finite differences, if either the up or down bump are NaN, then set the result to NaN and set the error message to the correspnding error message (if both up and down states have error, you can ignore one of the two error messages).

Modify *DemoRisk* to display both the total for the book and the number of trades which fail to price.

Compare the output of your program with the one given.

5.6 Add a new market object *CurveFXSpot*

The new market object should implement the interface *ICurveFXSpot*.

Note that the risk factors contained in the market data server only include spot currencies in the format CCYUSD, i.e. the prices of CCY in USD. The class must support direct, inverse and cross currency pairs (e.g. EURUSD, USDJPY, EURGBP). It is up to you to make the correct queries to the market data server and construct the *CurveFXSpot* appropriately.

You need to modity the *Market* class as appropriate.

Note that according with market conventions the spot price observed at time T is the forward price for an exchange of money at $T + 2$. Ignore this difference and assume that the spot price observed at time T is defined as the exchange rate for an instantaneous exchange of money at time T. Thanks to this simplification we can simply use the spot directly to convert the present value of a trade price from one currency to another. A real risk management system however, should take this difference into

account.

Modify the *PricerPayment* class to use the new class *CurveFXSpot*.

5.7 Add pricer configuration

Modify *DemoRisk* to read another optional argument form the command line, the base currency (argument name *-b CCY*). If not explicitly passed, this argument takes the value USD. Pass this to the *run* function. Modify the function *ITrade::Pricer* adding an argument *configuration* of type *string*.

In a real risk management system the configuration argument could be very complex and allow for instance to select a pricing model (e.g. Black Scholes or Heston) or a numerical framework (e.g. PDE or Monte Carlo). In this project the configuration string is simply the base currency in which all risk is to be computed. At the moment, by default everything is computed in USD. After your changes, *pricer("EUR")* should return a pricer that computes the price in EUR.

You need to modify all pricers accordingly. It may be convenient adding a base class *Pricer* which all pricers inherit from and take care of this conversion.

5.8 Add a new market object *CurveFXForward*

The new market object should implement the interface *ICurveFXForward*.

The forward price observed at time T_0 of currency ccy_1 expressed in currency ccy_2 for delivery at time T can be computed as

$$F(T_0, T) = \mathbb{E}[S(T)|\mathcal{F}_{T_0}] = S(T_0)\frac{B_1(T_0, T)}{B_2(T_0, T)}$$

where T_0 is the pricing date, $S(T_0)$ is the current spot price of currency ccy_1 expressed in currency ccy_2, $B_i(T_0, T)$ is the discount factor in currency ccy_i for time T observed at time T_0.

This class should use the *CurveFXSpot* class to get $S(T_0)$ and the *CurveDiscount* classes to get the discount factors.

Modify the *Market* class as appropriate.

5.9 Add fixing data server

To compute the price of some trades we need information in the past as well as in the future. For example, if today is the 15th of November and we want to compute the price of an Asian option that pays the average of the daily closing price in the month of November, some of these closing prices are in the past and need to be accounted for.

Past observations are called fixings. Introduce a *FixingDataServer* class, which implements the following methods:
// return the fixing if available, otherwise trigger an error
double get(const string& name, const Date& t) const;
// return the fixing if available, NaN otherwise, and set the flag if found
pair < double, bool > lookup(const string& name, const Date& t) const;

Populate the fixing data server with the data provided in the file *fixings.txt*.

Add an extra command line argument (*-x*) to *DemoRisk.cpp* and use it to communicate to the program the name of the data file containing the fixings.

Create a *FixingDataServer* object in *DemoRisk.cpp* and pass it as an extra argument to the *IPricer::price* function. Only the pricers of trades requiring fixings will do something with it, other pricers will ignore it (note that *TradePayment* does not have any fixing, hence it will not use it). In particular: fixings occurring in the past should be obtained from this argument and if not available an exception should be thrown; fixings occurring exactly on the pricing date should be taken from this argument if available, resolved through the Market otherwise; fixings occurring in the future should be resolved through the Market (e.g. forward prices).

Do not change the format of dates to serial and of double to hexadecimal in the data file.

5.10 Add FX Spot Greek function

Add another function for Greeks calculation, named *fx_delta*, which computes the risk of the book with respect to each of the FX spot rates quoted against USD in the market data server. It should use central difference with a relative bump of 0.1%.

It should have the same return value and argument types as the PV01 functions.

Add calculation of FXDelta at the end of *DemoRisk.cpp* and display the cumulative book value for each fx spot rate.

5.11 Add new trade type *TradeFXForward*

Given two dates T_1 and T_2, at time T_2 it pays the following amount of ccy_2

$$payoff = N[S(T_1) - K]$$

where $S(T_1)$ is the spot price of currency ccy_1 expressed in currency ccy_2 observed at the fixing date T_1, K is a predefined strike price, N is the notional amount and T_2 is the settlement date (it must be $T_1 < T_2$ for the trade to be valid).

Assign to the trade ID=3 (arbitrarily chosen) and the following serialized representation:
ID;NOTIONAL;CCY1;CCY2;STRIKE;FIXINGDATE;SETTLEDATE;

Example:
3;EUR;USD;notional in hex;strike in hex;date in serial;date in serial;

The currency pair $ccy_1 ccy_2$ can be any direct, inverse or cross pair.

Implement the corresponding pricer object. The price in ccy_2 can be computed as:

$$price = B_2(T_0, T_2)[F(T_0, T_1) - K]$$

where $B_2(T_0, T_2)$ is the discount factor in currency 2 for settlement date T_2 and $F(T_0, T_1)$ is the currency forward price for date T_1. Do not forget to convert this price into USD (or any other specified base currency).

5.11.1 Historical fixings

Note that when the pricing date T_0 is between T_1 and T_2, you need to know the value of S_{T_1}, which is in the past, i.e. you need to know its fixed value.

Note that there can be various situations with respect to the pricing date T_0:

- $T_0 < T_1$: both fixing and settlement are in the future
- $T_0 = T_1$ and fixing for T_1 not available: there is still full delta risk and PV01 risk
- $T_1 \leq T_0 < T_2$ and fixing for T_1 available: there is no more delta risk with respect to the underlying, but we still have PV01 risk and there may be delta risk for conversion to base currency
- $T_0 = T_2$: no delta with respect to the underlying or PV01 risk, we assume the payment has not been settled yet, hence we can still have a non-zero price, there may be still delta risk with respect to conversion to base currency.
- $T_0 > T_2$: trade is expired, an error should be generated