UNIVERSITÁ DEGLI STUDI DI MILANO-BICOCCA

Facoltá di Economia

Corso di Laurea Magistrale in Economia e Finanza

Abcd +

Relatore: Prof. Ferdinando M. AMETRANO

Tesi di Laurea di
GABRIELE GIUDICI
Nr. Matricola: 783898

Anno Accademico 2017-2018

abcdabcdabcd

Contents

Introduction **iv**

1 Interest rate **1**
1.1 Interest Rates and its components 1
1.2 Year fraction 2
1.3 Type and form of Interest Rate 4
1.3.1 Type 4
1.3.2 Form 6
1.4 Relations 7

2 Multiple curve bootstrapping **9**
2.1 Rationale and features 9
2.2 The algorithm 12

3 The ABCD of interest Rate Basis Spread **14**
3.1 Rationale for abcd modelling 14
3.2 Relation between continuous and simple basis 15
3.3 Generalized basis modelling 17

4 Abcd Framework **19**
4.1 Tenor Basis 19
4.2 Abcd Tenor Basis 26
4.3 Discount Corrected Term Structure 28

5 Abcd Reparameterization **31**
5.1 Parameters conversion 31
5.2 Idea adjustment 33

6 Abcd Framework+ **34**
6.1 Nested Calibration 34
6.2 Calibration data 34
6.3 Calibrate data 37

6.4 Global error 38
6.5 Calibrate 41

7 Fixing of c: an empirical analysis 44
7.1 Globally shared c 44
7.1.1 Not incremental approach 44
7.1.2 Incremental approach 48
7.2 Basis shifting 51
7.2.1 Not incremental approach 52
7.2.2 Incremental approach 55

8 Fixing of time: an empirical analysis 59

Introduction

Given the ABCD model [2], the aim of this research is to reparameterize and modify the calibration process in accordance with the following explanation and to empirically verify the robustness of our methodology.

Common c

Looking at the abcd continuous form of the basis and considering the following statements, it is possible to understand the reasons for the introduction of a new inputs form: "If the exponential term is the same, then a difference between abcd/ABCD basis will still be an abcd/ABCD function". This is clear from a mathematical point of view indicating:

- x as the first generic tenor;
- y as the second generic tenor, where: $x > y$;
- x, y as the difference of the above mentioned tenor: $x - y$.

It follows that, because of the tenor basis dominance explained in [2], given two absolute basis:

$$s_x(t) > s_y(t), \forall t > 0$$

and

$$s_x(t) - s_y(t), \forall t > 0$$

such that their respective abcd forms are:

$$s_x(t) = (a_x + b_x t)e^{-c_x t} + d_x$$

and

$$s_y(t) = (a_y + b_y t)e^{-c_y t} + d_y$$

if $c_x = c_y = c_{x,y}$ then the relative basis is still an abcd function:

$$\begin{aligned} s_{x,y}(t) &= s_x(t) - s_y(t) \\ &= (a_x + b_x t)e^{-c_x t} + d_x - ((a_y + b_y t)e^{-c_y t} + d_y) \\ &= (a_x - a_y + (b_x - b_y)t)e^{-c_{x,y} t} + d_x - d_y \\ &= (a_{x,y} + b_{x,y} t)e^{-c_{x,y} t} + d_{x,y} \end{aligned} \tag{1}$$

Where the above reported equation is obtained using the below reported notation:

$$\begin{aligned} a_{x,y} &= a_x - a_y; \\ b_{x,y} &= b_x - b_y; \\ d_{x,y} &= d_x - d_y; \end{aligned} \tag{2}$$

This result has been shown for continuous basis, but obviously it is equally valid for simple ones.

Considering f_x for a generic tenors x, the above reported relation can be exploited. Given that:

$$f_x(t) = s_x(t) + f_{ON}(t) \tag{3}$$

Then:

$$\begin{aligned} s_{x,y}(t) &= f_x(t) - f_y(t) \\ &= s_x(t) + f_{ON}(t) - s_y(t) - f_{ON}(t) \\ &= s_x(t) - s_y(t) \end{aligned} \tag{4}$$

is still an abcd basis, if the two basis are sharing the common exponential term.

For this reason, it is possible to use the abcd framework from [2], where the calibration of the model has been made with respect to a tenor which differs from the f_{ON} .

However, is it true that, given two generic tenors x and y, if:

$$c_x \neq c_y \tag{5}$$

then the relative basis $s_{x,y}$ is not an abcd function? Empirical results [1] show how it would seem that $s_{x,y}$ is still an abcd.

[1] *Abcd_Double_Hump_Research.xlsx* from the sheet *impact_time_dependent_d_lab*

Before going through algebraic calculus to proof this guess, it's preferred to proceed with a "reductio per absurdum". If the statement: "an abcd function shows only one t_{max}" is contradicted, then $s_{x,y}$ won't be an abcd function.

Given:

$$\begin{aligned} s_{x,y}(t) &= s_x(t) - s_y(t) \\ &= (a_x + b_x t)e^{-c_x t} + d_x - ((a_y + b_y t)e^{-c_y t} + d_y) \end{aligned} \tag{6}$$

considering $t_{max}(x)$ equation for s_x :

$$[-c_x(a_x + b_x t_{max}(x)) + b_x]e^{-c_x t_{max}(x)} = 0 \tag{7}$$

then $t_{max}(x, y)$ equation for $s_{x,y}$ is:

$$\begin{aligned} &[-c_x(a_x + b_x t_{max}(x, y)) + b_x]e^{-c_x t_{max}(x,y)} \\ -&[-c_y(a_y + b_y t_{max}(x, y)) + b_y]e^{-c_y t_{max}(x,y)} = 0 \end{aligned} \tag{8}$$

Unfortunately, it's impossible to retrieve $t_{max}(x, y)$ because of the problematic form:

$$y = b(e^b) \tag{9}$$

Because of this result it's impossible to study the problem with a "reductio per absurdum" and it's necessary to make an attempt retrieving an abcd from $s_{x,y}$:

$$\begin{aligned} s_{x,y}(t) &= s_x(t) - s_y(t) \\ &= (a_x + b_x t)e^{-c_x t} + d_x - ((a_y + b_y t)e^{-c_y t} + d_y) \\ &= (a_x + b_x t)e^{-c_x t} + d_{x,y} \end{aligned} \tag{10}$$

Where: $d_{x,y} = d_x - ((a_y + b_y t)e^{-c_y t} + d_y)$. This means that $s_{x,y}$ is still an abcd basis, but d is time dependent, this is the reason why looking at the empirical results it seemed an abcd one. Therefore, it makes no sense to talk about relative abcd retrieved from absolute abcd with different c.

Given this findings, what did it happen when $s_{x,y}$ was treated as an abcd in [2]? They obtained by construction $s_{x,6M}$ as abcd, for doing this they calibrated the basis on s_{6M} according to (4). Given that s_{6M} and $s_{x,6M}$ were abcd by construction, but with different c, it follows that s_x was not abcd. The only drawback in the adopted scheme is that is not possible to swap from relative to absolute basis. Acting in this way [2] created two different models, one for the absolute basis and one for the relative ones, that can be reconciled guaranteeing that c is the same for both the absolute and relative basis. Obviously, according to the calibration that will be performed (incremental or not), different parameters will be estimated, but with a common c will be possible to swap from a basis to the other indepentendly from the calibration type.

Maximum time

Independently from the considered tenor, the market uncertainty is expected to be on the same point of time t_{max}. In fact,there are no financial reasons why the market should forecast different uncertainty horizons for different tenors. Theoretically, this means that:

$$t_{max}(x) = \frac{1}{c_x} - \frac{a_x}{b_x} \tag{11}$$

is not a function of tenor, but an implied bound instead:

$$t_{max} = \frac{1}{c_x} - \frac{a_x}{b_x} \tag{12}$$

The above equation shows how the t_{max} needs to be the same for each tenor, thanks to the relation (tenor dependent) amongst a,b and c.

Recalling the importance of c, its value is strictly related with the maximum time, because given the maximum time functional (continuous time) form (11) and considering that empirically the magnitude of:

$$\frac{1}{c_x}$$

dominates the ratio between a_x and b_x, the value of c explains the value of t_{max}. Therefore, in order to fix t_{max} it makes sense to fix c. If c is fixed then (12) becomes:

$$t_{max} = \frac{1}{c} - \frac{a_x}{b_x} \tag{13}$$

This means that, in order to have the same t_{max} for different tenors, there should be an implied relation between a and b that holds for each tenor, so that, given two generic tenor x and y:

$$\frac{a_x}{b_x} = \frac{a_y}{b_y} \tag{14}$$

that need to be empirically evaluated.

In the end, strictly related with t_{max} is the corresponding value of the absolute abcd:

$$s_x(t_{max}) = \frac{b_x}{c_x} e^{(\frac{a_x c_x}{b_x} - 1)} + d_x \tag{15}$$

Aim

For these above mentioned reasons, the ideas that need to be investigated are:

1. reparameterization of the model, in its continuous form, relying on the following parameters:

$$a_x, d_x, t_{max}, s_x(t_{max}) \tag{16}$$

   This new parametric form makes sense, because:

   - $a_x + d_x$ is the value of the abcd corresponding to $t = 0$;
   - d_x is the long run value of the abcd basis;
   - t_{max} represents the peak of uncertainty in terms of time;
   - $s_x(t_{max})$ is the value of the abcd at the peak of uncertainty;

   Therefore, with this view is possible to give financial meaning to model parameters. Remarkable feature should be that, after retrieving the new parametric form, all the analytical findings from [2] could be exploited, just shifting from the new parametric form to the old abcd one. In this way, the users are given an interface, which allows them to fix the maxima of the basis as observed on the markets.

2. modification of the framework and testing the fitting globally sharing c, ensuring the property of moving from absolute to relative basis and vice versa;

3. modification of the framework and testing the fitting globally sharing t_{max}, studying the financial intuition behind it;

4. modification of the framework and testing the fitting globally sharing t_{max} and c, in order to completely validate the model and the idea.

Chapter 1

Interest rate

1.1 Interest Rates and its components

In a world based upon financial markets the fundamental rule is that "to collaborate with someone, people want to be remunerated". The basic financial action that agents can do is to lend their money, but, given the above mentioned rule, this action has to be supported by a reward: the interest (I). There are multiple reasons why interest exists and all of them are linked with the time-value of money, the implied benefits deriving from the possession of legal tender currency. Therefore interest rate should be higher enough for compensating :

1. Preference for liquidity, the value that an agent attributes on keeping cash in his bank account;
2. Alternative risk free investment, the value that an agent may have investing in less risky assets;
3. Inflation effects, the eroding power of inflation;
4. Borrower's creditworthiness, the risk embedded in the borrower itself.

For instance, considering a lender L and a borrower B, when B borrows from L a notional N expressed in the legal tender from t_1 up to t_2, L accepts if and only if at the maturity B gives back:

$$N + I = N + N(h(R, \tau)) = N(1 + h(R, \tau(t_1, t_2))) \tag{1.1}$$

Where:

- $I = N(h(R, \tau))$;

- $h(R, \tau)$ is the capitalization factor: percentage value applied to a notional today which gives the final amount to reimburse at the maturity. Its inverse is the discount factor, the value of the final amount today.
- τ is the year fraction: time from the starting of the contract to the maturity, expressed with year as tenure (time/ year), in accordance with business day convention (BDC) and day count convention (DCC).
- R is the applied interest rate;

The introduction of $h(R, \tau)$ and its components requires the explanation of these new objects.

1.2 Year fraction

Roughly speaking, year fraction indicates the time between two generic point of time t_1, t_2 expressed in term of years :

$$RawYearFraction = \frac{t_2 - t_1}{year} \tag{1.2}$$

In the real world there is no homogeneous date framework (for instance, there are different calendars for different stock exchanges) so that, in order to consider a generic time t it's necessary to deal with:

1. Business day convention: convention for managing a fixing date which falls on a vacation day;
2. Day count convention: convention for counting days.

Business day conventions can be grouped in:

- Following (preceding): if t is a vacation day, then the fixing date falls in the first next (previous) business day;
- Modified following (preceding): if t is a vacation day and if the first next (previous) business day belongs to the considered month then the following (preceding) convention is applied, otherwise the preceding (following) one is applied.
- End of month: when the start date of a period is on the final business day of a particular calendar month, the end date is on the final business day of the end month.

Day count conventions depend on the considered markets:

- Money market:

$$\frac{Actual}{Actual};\frac{Actual}{360};\frac{Actual}{365};\frac{Actual}{365Fixed};$$

- Bond market:

$$\frac{30}{360};$$

- Treasury market:

$$\frac{Actual}{360};$$

Where:

1. Numerator
   - "Actual" means that we are counting days according with the considered calendar;
   - "30" indicates that each month has 30 days.
2. Denominator:
   - "360/365" means that the considered year has 360/365 days;
   - "365Fixed" indicates that years normally have 365 days; leap years are not considered.

Given the above reported explanation, it should be now clear that the year fraction can't be simply seen as (1.2), but it is:

$$\tau(t_1 t_2, BDC, DCC) = \frac{t_2(BDC, DCC) - t_1(BDC, DCC)}{f(year)} \tag{1.3}$$

For readability purposes, in the following explanation, the above equation is written as $\tau(t_1, t_2)$.

1.3 Type and form of Interest Rate

1.3.1 Type

The type of interest depends on the rule of interest accruing.

The simple one is the " simple interest", it means that a certain interest is applied for a certain time to a certain notional (that is the same of the initial example). Therefore, mathematically it is possible to describe the accrued interest as:

$$I = NR\tau(t_1, t_2) \tag{1.4}$$

The capitalization factor, which indicates how much it grows an unit of currency in the specific interest regime, is:

$$h(R, \tau(t_1, t_2)) = 1 + R\tau(t_1, t_2) \tag{1.5}$$

while, remembering that exists an inverse relation between discount factor and capitalization factor, the former is:

$$D(R, \tau(t_1, t_2)) = \frac{1}{1 + R\tau(t_1, t_2)} \tag{1.6}$$

Example: investing 100$ of notional for one year (in a world where one year is one year, therefore your year fraction is 1) with 0.02 interest rate, then your final amount of $ will be:

$$100 * (1 + 0.02 * 1) = 102$$

Instead of investing 100$ for one year, it is possible to invest 100$ for n year , reinvesting each year the accrued interest :

$$x = 100 * (1 + R * 1)(1 + R * 1)..(1 + R * 1)$$

that can be rewritten as:

$$x = 100 * (1 + R * 1)^n$$

Investing each month instead of each year leads to:

$$x = 100\left(1 + \frac{R}{m}\right)^m$$

because the rate is applied for 1/m year for m times. Generalizing the notion, it is possible to write:

$$x = 100\left(1+\frac{R}{m}\right)^{mn}$$

that is what is obtained investing for n years reinvesting m times for year, the years can be rewritten as year fraction: $\tau(0, n)$. With a further generalization the spanning period can be simpy represented as an investment that goes from t_1 up to t_2, therefore the previous formula becomes:
Therefore, the capitalization factor is:

$$h(R, \tau(t_1, t_2)) = \left(1+\frac{R}{m}\right)^{m\tau(t_1,t_2)} \tag{1.7}$$

while the corresponding discount factor is:

$$D(R, \tau(t_1, t_2)) = \left(1+\frac{R}{m}\right)^{-m\tau(t_1,t_2)} \tag{1.8}$$

In the end, reinvesting infinite times, stressing the (1.7) equation, for m that goes to infinite:

$$\lim_{m\to\infty}\left(1+\frac{R}{m}\right)^{m\tau(t_1,t_2)}$$

because of Nepero limit:

$$\lim_{m\to\infty}\left(1+\frac{1}{m}\right)^{m} = e$$

changing variable and posing $t = \dfrac{m}{R}$, such that $m = tR$ and t is still pointing to infinite:

$$\lim_{t\to\infty}\left[\left(1+\frac{1}{t}\right)^{t}\right]^{\tau(t_1,t_2)R}$$

can be rewritten as:

$$h(R, \tau(t_1, t_2)) = e^{\tau(t_1,t_2)*R} \tag{1.9}$$

that is the continuously compounded capitalization factor. While its corresponding discount factor is:

$$D(R, \tau(t_1, t_2)) = e^{-\tau(t_1,t_2)R} \tag{1.10}$$

1.3.2 Form

The same rate can have different forms:

1. spot rate: rate that starts accruing interest today (the spot date, that for interest rate deriva is today plus 2 days) until a certain time T;
2. forward rate: rate that starts accruing interest at a future date (greater the the spot date) and until a certain time T;
3. instantaneous forward rate: rate that accrues interest from a certain t to $t + \Delta t$, where $\lim_{\Delta t \to 0}$, for such reason is called instantaneous.

Considering the rate $R(t_0, t)$, specifying it just for this section as $R(t)$ and $\tau(t_0, t)$ as t, given $R(t)$ and $R(t + \Delta t)$, taking the relative increment from t to $t + \Delta t$ with respect to time, it is obtained :

$$f(t, t + \Delta t) = \frac{R(t + \Delta t)(t + \Delta t) - R(t)t}{\Delta t}$$

taking $\lim_{\Delta t \to 0}$:

$$\begin{aligned} \lim_{\Delta t \to 0} f(t, t + \Delta t) &= \lim_{\Delta t \to 0} \frac{R(t + \Delta t)(t + \Delta t) - R(t)t}{\Delta t} \\ &= \lim_{\Delta t \to 0} \frac{R(t + \Delta t)\Delta t}{\Delta t} + \frac{t(R(t + \Delta t) - R(t))}{\Delta t} \end{aligned}$$

solving the limit:

$$\begin{aligned} f(t, t + \Delta t) &= R(t) + t\frac{\partial R(t)}{\partial t} \\ &= \frac{\partial t}{\partial t}R(t) + t\frac{\partial R(t)}{\partial t} \\ &= \frac{\partial R(t)t}{\partial t} \end{aligned}$$

integrating both sides:

$$\int_{t_0}^{t} f(u, u + \Delta u)\mathrm{d}u = R(t)t$$

Therefore, the $R(t)$ can be seen as the average of instantaneous forward rate between t_0 and t:

$$R(t) = \frac{\int_{t_0}^{t} f(u, u + \Delta u)\mathrm{d}u}{t} \tag{1.11}$$

Remembering (1.10) and indicating t as $\tau(t_0, t)$, it follows that in continuous time the discount factor can be seen as:

$$e^{-\int_{\tau(0,t_0)}^{\tau(0,t)} f(u,u+\Delta u)\mathrm{d}u} \tag{1.12}$$

Pay attention: this integral makes sense if and only if the adopted DCC is strictly monotone!

1.4 Relations

Beyond the above reported relation, particularly interesting is the no arbitrage relation that follows: in an arbitrage free world investing today t_0 until T or investing from t_0 to t_1 and from t_1 to T should be equivalent, otherwise all the investor would investing in one or the other strategy until when, because of the market adjustments, they are indifferent between the two. Therefore, it is valid that:

$$e^{R_{(t_0,t_1)}\tau(t_0,t_1)} e^{F_{(t_1,T)}\tau(t_1,T)} = e^{R_{(t_0,T)}\tau(t_0,T)}$$

therefore the forward rate can be retrieved as:

$$F_{(t_1,T)} = \frac{R_{(t_0,T)}\tau(t_0, T) - R_{(t_0,t_1)}\tau(t_0, t_1)}{\tau(t_1, T)} \tag{1.13}$$

Moreover, considering the simple capitalization factor, it is possible to obtain a similar approximated relation:

$$(1 + R_{(t_0,t_1)}\tau(t_0, t_1))(1 + F_{(t_1,T)}\tau(t_1, T)) = (1 + R_{(t_0,T)}\tau(t_0, T))$$

that leads to :

$$F_{(t_1,T)} = \frac{1}{\tau(t_1, T)} \left(\frac{D(t_0, t_1)}{D(t_1, T)} - 1 \right) \tag{1.14}$$

which is essential because allows retrieving F, as discount factor ratio, independently from what form of D has been employed.

Then, considering (1.12), it can be rewritten as:

$$F_{(t_1,T)} = \frac{1}{\tau(t_1, T)} \left(e^{-\int_{\tau(0,t_1)}^{\tau(0,T)} f(u,u+\Delta u)\mathrm{d}u} - 1 \right) \tag{1.15}$$

These relations are fundamental for two reasons, firstly (1.14) allows to model F starting from a instantaneous forward rate f and it is the crucial point in abcd framework and its revision. Secondly, during the financial crisis this relation stopped to be valid on the standard curve and today it holds only on each curve of the multiple curves world where "investing today t_0 until T or investing from t_0 to t_1 and from t_1 to T it's equivalent if and only if there is a basis on the rate that spans from 0 to T in order to compensate the impossibility to get out from the investment before than T, which represents the risk of credit and maturity".

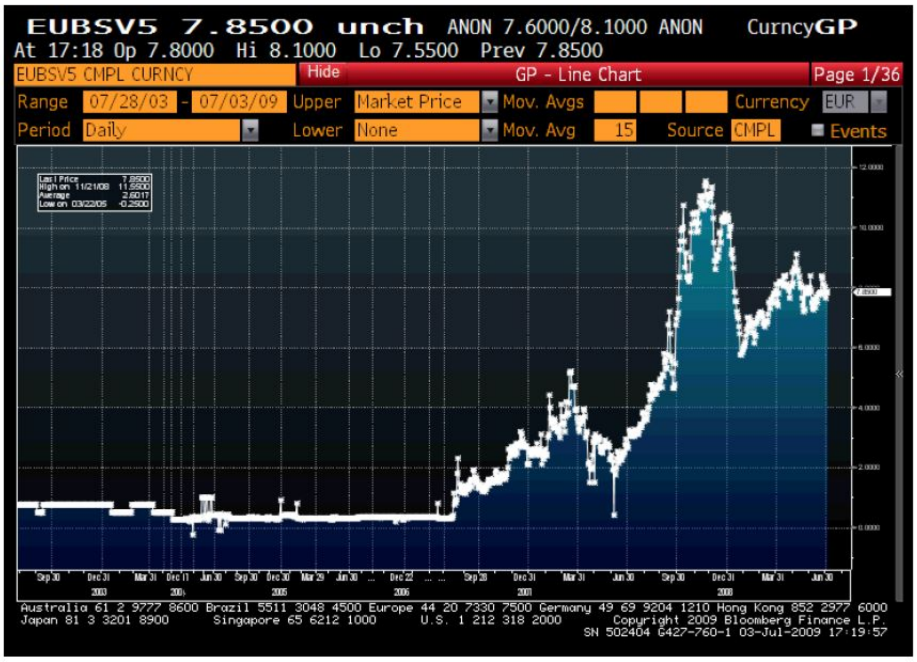
Chapter 2

Multiple curve bootstrapping

2.1 Rationale and features

This section aims to explain how to deal with multiple curves world and bootstrapping, linking these concepts with the abcd framework.

As already mentioned above, during the financial crisis a premium between different tenor grew :


```
EUBSV5 7.8500 unch  ANON 7.6000/8.1000 ANON     CurncyGP
At 17:18 Op 7.8000  Hi 8.1000  Lo 7.5500  Prev 7.8500
EUBSV5 CMPL CURNCY          Hide        GP - Line Chart        Page 1/36
Range  07/28/03 - 07/03/09  Upper  Market Price  Mov. Avgs              Currency EUR
Period Daily                Lower  None          Mov. Avg   15  Source CMPL   Events

Last Price        7.8500
High on 11/21/08 11.5500
Average           2.6017
Low on 03/22/05  -0.2500

Australia 61 2 9777 8600 Brazil 5511 3048 4500 Europe 44 20 7330 7500 Germany 49 69 9204 1210 Hong Kong 852 2977 6000
Japan 81 3 3201 8900      Singapore 65 6212 1000      U.S. 1 212 318 2000       Copyright 2009 Bloomberg Finance L.P.
                                                                   SN 502404 G427-760-1 03-Jul-2009 17:19:57
```

Figure 2.1: Basis swap 3M vs 6M premium

a change of structure, but not correlation among different tenor appeared, which led to phenomenon like the difference between implied forward rate and market forward rate and therefore Forward Rate Agreement mispricing:

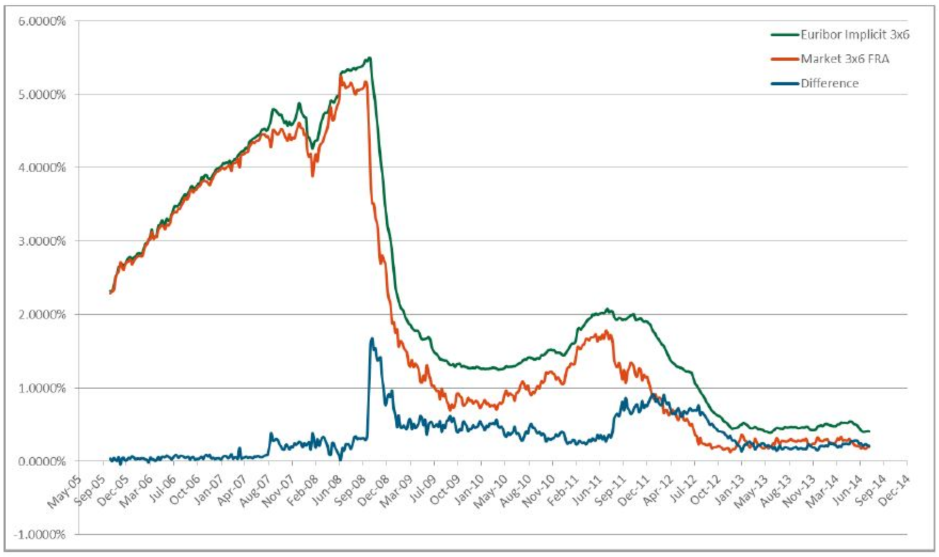


Figure 2.2: Implied FRA vs Market FRA

Given these considerations, a framework for estimating these new curves was required. The chosen method is the bootstrapping, a numerical procedure that allows, for instance, to retrieve a term structure such that the market quotes used in the process are exactly or with the minimum error repriced. The bootstrap vademecum core principles are:

- smoothness as term structure quality indicator, the most granular representation of the curve is its instantaneous forward one, that in the real world corresponds to look at overnight forward rate applying (1.15) with $t1 = today$ and $T = today + 1day$.

- homogeneous instruments: it means that in order to bootstrap an x tenor curve the algorithm has to be fed with the corresponding instruments which incorporate the maturity and credit premia;

- unique discounting curve. Before crises discounting and forwarding curve were the same thing, after rising of multiple curves it needed to be chosen a discounting curve. Initially each curve was discounted using itself (endogenous discounting), but clearly this was not reasonable. Indeed, given that 1$ tomorrow is worth the same independently from what curve it is considered, it should be found a unique discounting curve. Given that the interest rate derivative world it is based on collateralization and given that on collateral account is paid the ON rate, it makes sense that the cost of money and therefore the discounting curve should be retrieved on the overnight one (exogenous discounting).

- not inclusion of deposit in bootstrap procedure: according to the previous considerations and given that deposits are not collateralized it makes no sense including them in the procedure.
- choice of bootstrap method: exact fit or best fit. While the former lead to an exact repricing of market quote, the second just try to achieve the best fit as possible. The main idea is that "a term structure that can't perfectly repricing the instruments which is looking at during the bootstrap it is fragile and seems not reliable for further uses", therefore an exact fit approach is preferred.
- choice of interpolation scheme. A global interpolator allows to preserve curve shape, while a local one exchanges a better shape with easiness of implementation and interpretation. The former, although bringing complexity, is preferred.

Understood which are the best practices on the market, before proceeding toward the implementation, there are common problems that need to be addressed in order to perfectly get the idea of how a bootstrapping algorithm works.

- information short: if the aim is to bootstrap a curve with a tenor greater than ON, it means that in the first part of the curve up to the first market quote there will be a short of information, therefore the quant needs to understand how to deal with this. Different solution have been implemented, among those it was spread the use of synthetic deposit, which roughly speaking mimic the shape of ON curve adding a basis in order to match the x tenor modelled curve magnitude.
- liquidity necessity: given a liquidity need, that arises because of regulation constraints, in the end of year, but in general each time that liquidity requirement is requested, a leap on the ON curve is observable, which is trasmitted to a jump in the respective x tenor modelled curve. This matter required a particular estimation technique which can easily lead to an overall estimation error.
- missing pillar: given n market quotes which help to model their respective curve, how is it possible to deal with the curve pillar those are not

reflected in them, but can be useful to evaluate particular instruments? The answer is: interpolating. This opens new debating, that can be divided in two main groups. The first is about local interpolation scheme, it means that the searched point is fully described by 2 points which lay next to it, for example the linear interpolation belongs to this family. Unfortunately this approach do not preserve the curve shape, therefore advanced interpolations scheme should be employed and they are called global, because the interpolated point depends on the global shape of the curve, dependencies exist among each point of the curve. Among them, the one which leads to optimal results is the monotonic cubic, but this algorithm may have problem of spurious convexity or lack of monotonicity, that can be erased applying a filter as the Hyman one.

2.2 The algorithm

Given the above explained background where the bootstrap algorithm takes roots, it is possible proceed with the explanation relying on the QuantLib implementation.

The main elements that compete in bootstrapping procedure are :

1. a vector of no overlapping market quotes (deposit,FRA, futures,Swap) $\mathbf{Q} = \{q_1, q_2..q_n\}$;
2. trait T, it is the form of the rate used in the interpolation(discount factor, forward rate, instantaneous forward rate);
3. interpolation type I (linear, cubic spline ecc) ;
4. type of bootstrapping B (iterative bootstrap, local bootstrap);
5. type of fit (exact or best fit).

In the following the iterative bootstrap algorithm is sketched. Firstly, the process is recursively and incremental: it means that starts from the first pillar and evolves pillar by pillar up to the last one, corresponding to the last market quote date, and if the exit condition is still unsatisfied it restarts recursively up to the satisfaction of a given condition (for instance: the maximum difference between the curve quotes before and after the bootstrap cycle needs to be lower than the accuracy or the reaching of the maximum number of iterations). Given that each q_i instrument refers to a precise ordered dates, the algorithm starts cycling from the first available date, i.d the first pillar, which corresponds to the first element of Q: q_1. First time an instrument enters in the process, for its corresponding pillar, a guess and an error function are provided (in the recursive loop the guess becomes the value from the previous iteration) and subsequently passed as inputs to the solver which tries to minimize such error function of the q_i quote and the r_i quote, which is implied (instrument dependent) in the term structure. For instance, given a quoted q_i that corresponds to 3x6 market FRA rate, that is quoted in term of its contractual fixed rate, the corresponding implied fixing r_i from the currently bootstrapped term structure has to be the same of it. This means zeroing the error function as below reported:

$$f(q_i, r_i) = 0$$

In order to achieve this result the solver changes the guess/ previous value, the algorithm puts the new value in curve (at the height of the considered pillar), the curve it's interpolated with the specified type of interpolation I on the decided form of rate T and then r_i it is computed and an error retrieved through $f(q_i, r_i)$. Once the pillar has been changed then the algorithm steps ahead to the next pillar and it re-proposes the same solution. Note: in this process the evaluation of each instrument depends on the I that has been chosen, because for evaluating points outside the quoted grid an interpolation is required. In particular case of global type, changing the pillar means to change the bootstrapped term structure. Because of this continuous mutant behavior the algorithm should evaluate recursively each pillar more than once in order to satisfy the already mentioned condition. Therefore, in case of global scheme, until the satisfaction of the exit condition the algorithm cycles, while for a local scheme one run is enough for obtaining the searched solution.

The above reported was QuantLib idea of bootstrapping, anyway, instead of repricing the quotes, it is possible to work on the term structure that allows zeroing of the instruments net present value, the rationale is that in order to buy and sell today a given instrument its net present value should be zero, otherwise can't there be a deal counterpart.

Concluding, the multiple curve bootstrap it is fundamental not only as a finance principle, but also in the continuation, because has been used in abcd framework for building the legacy curves that are one of the metric with respect to evaluate the goodness of the new curve construction scheme.

Chapter 3

The ABCD of interest Rate Basis Spread

3.1 Rationale for abcd modelling

Given the problems and practical difficulties shown in chapter 2 along with the inconsistency of (1.14) from chapter 1, [2] thought to model the basis between an x tenor simple forward and its corresponding overnight spanning forward as:

$$S_x(t) = F_x(t) - F_x^{ON}(t) \tag{3.1}$$

that empirically can be parameterized as:

$$S_x(t) = (A_x + B_x t)e^{-C_x t} + D_x \tag{3.2}$$

The parameters have a particular meaning:

- A + D indicates the value of the basis in t equals to 0;
- D is the long run function value;
- C indicates how fastly the function tends to D;
- B indicates the additional boost to t in order get the maximum value of function.

As it is clear, A and D have clear financial meaning while the other two cannot be so easily interpreted. This difficult in interpreting them, will lead to a modification in the following of this dissertation.

This basis is calibrated only using a subset of the instruments refereed to the x modelled curve. The calibration of the basis is performed exploiting a best fit calibration algorithm such that is obtained $F_x^{calib}(t) = S_x(t) + F_x^{ON}(t)$. Subsequently, through an exact fit bootstrapping procedure, which exploits linear interpolation and therefore local scheme, the Rebonato correction factors are obtained:

$$K_x(t_i) = \frac{F_x^{mkt}(t_i)}{F_x^{calib}(t_i)} \tag{3.3}$$

where $F_x^{mkt}(t_i)$ is the value of the contract quoted on the market and fixing at time t_i. The corrector factors play a huge role, because allow to shape the information around an abcd basis while leaving the possibility to perfectly reproducing the observed curve and the embedded noise that belongs to the market.

3.2 Relation between continuous and simple basis

However, as already mentioned, in derivative world, the best practice is to model the quantity in continuous time, and it allows to model the instantaneous forward rate when retrieving the pseudo discount factor as in (1.15) therefore it is introduced:

$$s_x(t) = f_x(t) - f_{ON}(t) \tag{3.4}$$

where $f_x(t)$ is the instantaneous forward rate on the x curve and $f_{ON}(t)$ is the instantaneous forward rate on the ON curve.

Egregious results are that the meaning of parameters is preserved and that exists an relation between continuous and simple basis:

$$1 + F_x(t)\tau_x(t) = e^{\int_t^{t+x} f_x(u)\mathrm{d}u} \tag{3.5}$$

Since $f_x(t) = s_x(t) + f_{ON}(t)$, we have:

$$\begin{aligned} 1 + F_x(t)\tau_x(t) &= e^{\int_t^{t+x}[s_x(u)+f_{ON}(u)]\mathrm{d}u} \\ &= [1 + F_x^{ON}(t)\tau_x(t)]e^{\int_t^{t+x} s_x(u)\mathrm{d}u}. \end{aligned} \tag{3.6}$$

Therefore:

$$e^{\int_t^{t+x} s_x(u)\mathrm{d}u} = \frac{1 + F_x(t)\tau_x(t)}{1 + F_x^{ON}(t)\tau_x(t)}. \tag{3.7}$$

Taking the logarithms and using a first-order approximation, we have:

$$\begin{aligned}\int_t^{t+x} s_x(u)\mathrm{d}u &= \ln\left[\frac{1+F_x(t)\tau_x(t)}{1+F_x^{ON}(t)\tau_x(t)}\right] \\ &\approx F_x(t)\tau_x(t) - F_x^{ON}(t) \\ &= S_x(t)\tau_x(t).\end{aligned} \tag{3.8}$$

Therefore it is possible to express the parameters of the simple basis A, B, C, D as parameters of the continuous one a,b,c,d as it follows:

$$\begin{aligned}\int_t^{t+\tau_x} s_x(u)\mathrm{d}u &= \int_t^{t+\tau_x} \left[(a_x + b_x u)e^{-c_x u} + d_x\right]\mathrm{d}u \\ &= \left[\hat{A}_x(\tau_x) + \hat{B}_x(\tau_x)t\right] e^{-\hat{C}_x t} + \hat{D}_x(\tau_x)\end{aligned} \tag{3.9}$$

with

$$\begin{aligned}\hat{A}_x(\tau_x) &= -\left(\frac{a_x}{c_x} + \frac{b_x}{c_x^2} + \frac{b_x}{c_x}\tau_x\right)e^{-c_x\tau_x} + \left(\frac{a_x}{c_x} + \frac{b_x}{c_x^2}\right) \\ \hat{B}_x(\tau_x) &= \frac{b_x}{c_x}\left(1 - e^{-c_x\tau_x}\right) \\ \hat{C}_x &= c_x \\ \hat{D}_x(\tau_x) &= d_x\tau_x.\end{aligned} \tag{3.10}$$

Substituting the above into equation 3.8, we obtain

$$\left[\hat{A}_x(\tau_x) + \hat{B}_x(\tau_x)t\right] e^{-\hat{C}_x t} + \hat{D}_x(\tau_x) \approx S_x(t)\tau_x \tag{3.11}$$

Given that also with this new framework a best fit calibration was employed, in order to exact reprice the quote the following corrector factor has been locally bootstrapped according to:

$$D_x^{adj}(t) = k_x(t)\cdot D_x^{calib}(t) = k_x(t)\cdot D_{ON}(t)\cdot e^{\int_0^t s_x(u)\mathrm{d}u} \tag{3.12}$$

in order to get:

$$F_x(t) = \frac{1}{\tau_x}\left[\frac{D_x^{adj}(t)}{D_x^{adj}(t+x)} - 1\right] \tag{3.13}$$

Note: differently from basis, it is not possible to goes from continuous to simple correction factors.

3.3 Generalized basis modelling

In order to achieve better results the previous structure was applied but with incremental calibration, exploiting the generalized equation:

$$\begin{aligned} S_{x,y}(t) &= F_x(t) - F_x^y(t) \\ s_{x,y}(t) &= f_x(t) - f_y(t) \end{aligned} \tag{3.14}$$

At this point the authors, always relying on the idea of modelling a basis as abcd, created two different framework one for the relative basi from incremental calibration and one for absolute basis with respect to the overnight rate in a not incremental calibration scheme.

Moreover, they noticed how the t_{max} values were closed for each guessing a common value for it. Furthermore, they noticed how the simple basis peak anticipates the continuous one, but it is correctly, because :

$$\int_t^{t+x} s_x(u)\mathrm{d}u = S_x(t)\tau_x(t). \tag{3.15}$$

therefore the basis $S_x(t)$ in t incorporates the maximum value of $s_x(t_{max})$.

Both the approaches led to extremely positive results, but the second allowed also to overcome the problematic shape of the retrieved 1M curve as it is possible to appreciate:

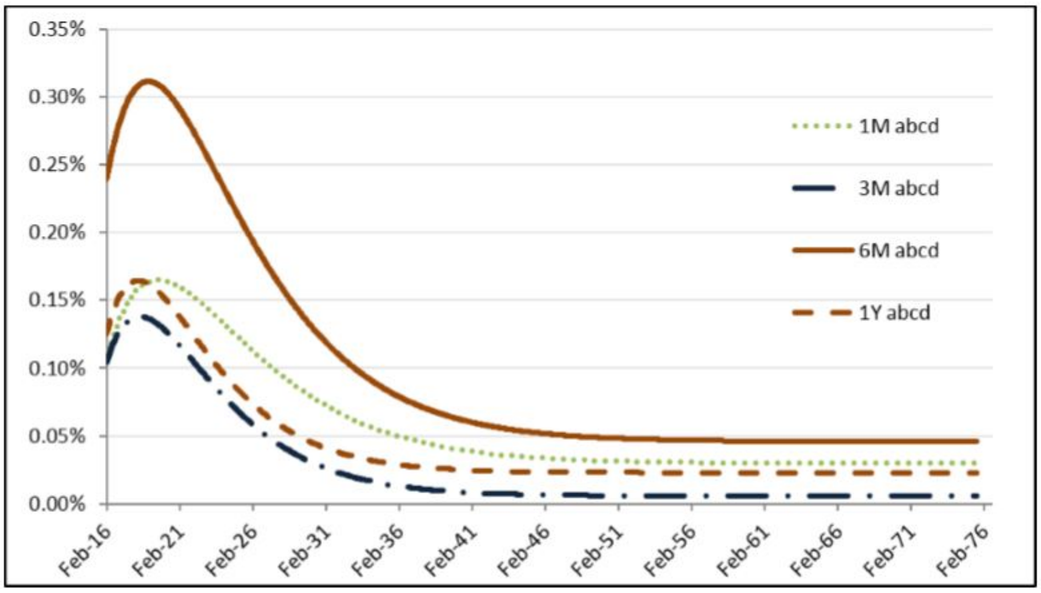


Figure 3.1: Basis from incremental calibration

Looking at the empirical results a common feature that seems to be interested is that all the basis share quite the same time of maximum and it makes sense, because ther is no reason why a basis should forecast different uncertainty horizon from another one.

Unfortunately, it is not possible to shift from relative to absolute approach because difference of abcd with different exponential term do not lead to another abcd. In order to fix that matter allowing the model to manage the shifting and reconciling the absolute and relative ones, in order to evaluate the intuition of a financial common t_{max} value and looking for parameters that embeds a greater financial meaning, the abcd framework needs to be reviewed and tuned.

Chapter 4

Abcd Framework

In order to improve the [2] framework it is essential to understand the previous implementation, that is written in C++ language and available in the QuantLib library in the "experimental" folder. The abcd implementation relies on the QuantLib modularity which acts as a base framework from which to extend all the further derivations.

4.1 Tenor Basis

Tenor basis is the super class from which the abcd scheme derives. It is modelled as a *CalibratedModel*:

```
class TenorBasis : public CalibratedModel {
     public:
          TenorBasis(Size nArguments,
                      boost::shared_ptr<IborIndex> iborIndex,
                      boost::shared_ptr<IborIndex> baseIborIndex,
                      Date referenceDate = Date())
                      ... }
```

Therefore it inherits its structure and calibrate methods. The most important methods, for continuous basis calibration, are:

```
Rate accrualFactor(Date d1,Date d2) const;
```

and

```
void calibrate(
    const std::vector<boost::shared_ptr<RateHelper> >&,
    OptimizationMethod& method,
    const EndCriteria& endCriteria
    = EndCriteria(1000, 100, 1.0e-8, 0.3e-4, 0.3e-4),
    const std::vector<Real>& weights = std::vector<Real>(),
    const std::vector<bool>& fixParameters = std::vector<bool>());
```

For better understanding these two functions, it needs to be explained:

```
class TenorBasisYieldTermStructure : public YieldTermStructure {
  public:
    TenorBasisYieldTermStructure
    (const boost::shared_ptr<TenorBasis>& basis);
    const Date& referenceDate() const;
    Calendar calendar() const;
    Natural settlementDays() const;
    Date maxDate() const;
  private:
    DiscountFactor discountImpl(Time) const;
    boost::shared_ptr<TenorBasis> basis_;
    };
```

and its constructor:

```
    TenorBasisYieldTermStructure::TenorBasisYieldTermStructure(
                    const boost::shared_ptr<TenorBasis>& basis)
    : YieldTermStructure(basis->dayCounter()), basis_(basis) {}
```

that simply builds an *YieldTermStructure* object and stores a *TenorBasis* object called basis in *basis_*. Moreover, it designs a *discountImpl* which will be fundamental and therefore later explained. The core of Tenor is the *calibrate* method which follows:

```
    void TenorBasis::calibrate(
    const std::vector<boost::shared_ptr<RateHelper> >& helpers,
    OptimizationMethod& method,
    const EndCriteria& endCriteria,
    const std::vector<Real>& weights,
    const std::vector<bool>& fixParameters) {
    TenorBasisYieldTermStructure
     yts(boost::shared_ptr<TenorBasis>(this, no_deletion));
    std::vector<boost::shared_ptr<CalibrationHelperBase> >
                                    cHelpers(helpers.size());
    for (Size i = 0; i<helpers.size(); ++i) {
         helpers[i]->setTermStructure(&yts);
         cHelpers[i] = helpers[i];
    }
    CalibratedModel::calibrate(cHelpers, method, endCriteria,
                         constraint(), weights, fixParameters);
    }
```

A *TenorBasisYieldTermStructure*, which stores the *TenorBasis* object itself as *basis_*, is built and called yts. Then a vector of *CalibrationHelperBase* is instantiated. It is fundamental the chain which links *CalibrationHelper* and *RateHelper*: indeed the former it is child of *CalibrationHelperBase*, while the latter inherits from *BootstrapHelper*, which is child of *CalibrationHelperBase* too. Given these chain of properties it is possible to write $cHelpers[i] =$

helpers[*i*]. In the end the method from the parent class *CalibratedModel* is exploited:

```
   void CalibratedModel::calibrate(
                  const vector<shared_ptr<CalibrationHelper> >& h,
                  OptimizationMethod& method,
                  const EndCriteria& endCriteria,
                  const Constraint& additionalConstraint,
                  const vector<Real>& weights,
                  const vector<bool>& fixParameters) {
    vector<shared_ptr<CalibrationHelperBase> > tmp(h.size());
    for (Size i=0; i<h.size(); ++i)
    tmp[i] = static_pointer_cast<CalibrationHelperBase>(h[i]);
    calibrate(tmp, method, endCriteria, additionalConstraint,
    weights, fixParameters);
    }
```

This piece of code simply casts the helpers in *CalibrationHelperBase* objects and then calls the overloaded polymorphic method *calibrate*, which is the core of the calibration process:

```
void CalibratedModel::calibrate(
        const vector<shared_ptr<CalibrationHelperBase> >& h,
        OptimizationMethod& method,
        const EndCriteria& endCriteria,
        const Constraint& additionalConstraint,
        const vector<Real>& w,
        const vector<bool>& fixParameters) {

    QL_REQUIRE(!h.empty(), "no helpers provided");

    Array prms = params();
    if (fixParameters.empty()) {
        fixedParameters_.resize(prms.size());
        std::fill(fixedParameters_.begin(),
        fixedParameters_.end(), false);
    } else {
        QL_REQUIRE(fixParameters.size() == prms.size(),
               "mismatch between number of parametrs (" <<
               h.size() << ") and fixed parameters booleans
               (" <<fixParameters.size() << ")");
        fixedParameters_ = fixParameters;
    }
    Projection proj(prms, fixedParameters_);

    if (w.empty()) {
        weights_.resize(h.size());
        std::fill(weights_.begin(), weights_.end(), 1.0);
    } else {
       //checks
    }
    CalibrationFunction f(this, h, weights_, proj);

    Constraint c;
    if (additionalConstraint.empty())
     c = *constraint_;
    else
    c = CompositeConstraint(*constraint_, additionalConstraint);
    ProjectedConstraint pc(c, proj);

    Problem prob(f, pc, proj.project(prms));
    shortRateEndCriteria_ = method.minimize(prob, endCriteria);
    Array result(prob.currentValue());
    setParams(proj.include(result));
    problemValues_ = prob.values(result);

    notifyObservers();
    }
```

Firstly, the fixing of parameters is managed: if *fixParameters* is empty by default the values are not fixed, instead if some value is fixed, it is required that the number of fixing indicator should equals the number of parameters.

Afterwards, a *Projection* object is instantiated. There a bit of the machinery is repeated (about checking the number of fixing indicator) and then the number of free parameters is set. Furthermore, there is a fundamental method *Projection::project* which returns only the not fixed values, i.e. those parameters that should be calibrated during the process, for this reason called *projectedParameters*.

After checking the presence of weights and assigned a default values of 1 for each entry of the *weights_* vector, it builds the *CalibrationFunction*, this is possible because it is an inner class of *CalibratedModel*. This function is essential, because allows to retrieve the errors in order to calibrate the model, but will be later better explained.

Moreover, a *ProjectedConstraint* object is built and it allows to manage the additional constraint along with the fixed parameters.

Therefore, all the ingredients necessary for the problem are provided, the problem set and the solver method can minimize the problem request with a specific *EndCriteria*. Here the trick is performed. Considering that the selected method for abcd framework is *LevenbergMarquardt*:

```
EndCriteria::Type LevenbergMarquardt::minimize(Problem& P,
                       const EndCriteria& endCriteria) {
        EndCriteria::Type ecType = EndCriteria::None;
       P.reset();
        Array x_ = P.currentValue();
        currentProblem_ = &P;
        initCostValues_ = P.costFunction().values(x_);
        int m = initCostValues_.size();
        int n = x_.size();
        if(useCostFunctionsJacobian_) {
            initJacobian_ = Matrix(m,n);
            P.costFunction().jacobian(initJacobian_, x_);
        }
        boost::scoped_array<Real> xx(new Real[n]);
        std::copy(x_.begin(), x_.end(), xx.get());
        boost::scoped_array<Real> fvec(new Real[m]);
        boost::scoped_array<Real> diag(new Real[n]);
        int mode = 1;
        Real factor = 1;
        int nprint = 0;
        int info = 0;
        int nfev =0;
        boost::scoped_array<Real> fjac(new Real[m*n]);
        int ldfjac = m;
        boost::scoped_array<int> ipvt(new int[n]);
        boost::scoped_array<Real> qtf(new Real[n]);
        boost::scoped_array<Real> wa1(new Real[n]);
        boost::scoped_array<Real> wa2(new Real[n]);
        boost::scoped_array<Real> wa3(new Real[n]);
        boost::scoped_array<Real> wa4(new Real[m]);
        //error messages
        QL_REQUIRE(n > 0, "no␣variables␣given");
        QL_REQUIRE(m >= n,
                   "less␣functions␣(" << m <<
                   ")␣than␣available␣variables␣(" << n << ")");
        QL_REQUIRE(endCriteria.functionEpsilon() >= 0.0,
                   "negative␣f␣tolerance");
        QL_REQUIRE(xtol_ >= 0.0, "negative␣x␣tolerance");
        QL_REQUIRE(gtol_ >= 0.0, "negative␣g␣tolerance");
        QL_REQUIRE(endCriteria.maxIterations() > 0,
                   "null␣number␣of␣evaluations");

        MINPACK::LmdifCostFunction lmdifCostFunction =
        boost::bind(&LevenbergMarquardt::fcn, this,
        _1, _2, _3, _4, _5);
        MINPACK::LmdifCostFunction lmdifJacFunction =
        useCostFunctionsJacobian_
        ? boost::bind(&LevenbergMarquardt::jacFcn, this,
        _1, _2, _3, _4, _5)
            : MINPACK::LmdifCostFunction(NULL);
        MINPACK::lmdif(m, n, xx.get(), fvec.get(),
                           endCriteria.functionEpsilon(),
                           xtol_,
                           gtol_,
                           endCriteria.maxIterations(),
                         epsfcn_,
                         diag.get(), mode, factor,
                         nprint, &info, &nfev, fjac.get(),
                         ldfjac, ipvt.get(), qtf.get(),
                         wa1.get(), wa2.get(), wa3.get(), wa4.get(),
                         lmdifCostFunction,
                         lmdifJacFunction);
          info_ = info;
          // check requirements & endCriteria evaluation
          //QL_REQUIRE checks
          std::copy(xx.get(), xx.get()+n, x_.begin());
          P.setCurrentValue(x_);
          P.setFunctionValue(P.costFunction().value(x_));

          return ecType;
     }
```

It resets the problem and sets the variables, it performs a series of checks, it sets a particular cost function *LevenbergMarquardt::fcn* and then *lmdif* performs the minimization, after that the current problem value and the cost function one are set. Without being too specific, *lmdif* logic acts as follows: the error that should be minimized are retrieved through the values function embedded in :

```
void LevenbergMarquardt::fcn(int, int n, Real* x,
Real* fvec, int*) {
    Array xt(n);
    std::copy(x, x+n, xt.begin());
    if (currentProblem_->constraint().test(xt)) {
        const Array& tmp = currentProblem_->values(xt);
        std::copy(tmp.begin(), tmp.end(), fvec);
    } else {
    std::copy(initCostValues_.begin(),
    initCostValues_.end(), fvec);
    }
    }
        }
```

and written in *fvec.* The implementation of *values*, from the *model* class, follows:

```
      virtual Disposable<Array>values(const Array& params)const {
            model_->setParams(projection_.include(params));
            Array values(helpers_.size());
            for (Size i=0; i<helpers_.size(); ++i) {
                values[i] = helpers_[i]->calibrationError() *
                std::sqrt(weights_[i]);
            }
            return values;
        }
```

Here a fundamental role is played by :

```
    void CalibratedModel::setParams(const Array& params) {
        Array::const_iterator p = params.begin();
        for (Size i=0; i<arguments_.size(); ++i) {
            for (Size j=0; j<arguments_[i].size(); ++j, ++p) {
            ...
                arguments_[i].setParam(j, *p);
            }
        }
        QL_REQUIRE(p==params.end(),"parameter array too big!");
        generateArguments();
        notifyObservers();
    }
```

This function sets the *params* chosen by the algorithm in the *arguments_* container and then generates the new parameters of the function that should be evaluated in order to get the errors. This feature will be fundamental in modifying the current framework.

The overall error it is retrieved exploiting *calibrationError()*:

```
 Real calibrationError() const {
 return quote_->value() - impliedQuote(); }
```

and there it is clear the mimic of the bootstrapping *"Ã la QuantLib"* which creates an error, exploiting the polymorphic feature of the particular instantiated rate helper. This is not all: when the *impliedQuote* method is called, then it asks at the *termStructure_* to return a *discount*. Besides, it is woth to know that the *zeroRate* and *forwardRate* functions in QuantLib are all implemented as functions of *discount*, that it is a function of *disocuntImpl*, therefore they are linked. Remembering what anticipated above about the importance of *accrualFactor*, the *TenorBasisYieldTermStructure::discountImpl* is shown:

```
  DiscountFactor TenorBasisYieldTermStructure::
  discountImpl(Time t) const {
        Date ref = referenceDate();
        Date d = basis_->dateFromTime(t);
        Real accrFactor = basis_->accrualFactor(ref, d);
        return 1.0 / accrFactor;
    }
```

this is the key: the error depends on *discountImpl* that it depends on *accrualFactor*, which models the pseudo discount with the abcd framework :

```
    Rate TenorBasis::accrualFactor(Date d1,
                                   Date d2) const {
    QL_REQUIRE(d1 <= d2,
               "d2 (" << d2 << ") cannot be
```

```
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣before␣d1␣(" << d1 << ")");
    // baseCurve must be a discounting curve...
    // otherwise it could not provide fwd(t1, t2) with t2-t1!=tau
    Handle<YieldTermStructure> baseCurve =
    baseIborIndex_->forwardingTermStructure();
    Real accrFactor =
    baseCurve->discount(d1) / baseCurve->discount(d2);
    Real instContBasisIntegral = sign_ * integrate_(d1, d2);
    accrFactor *= std::exp(instContBasisIntegral);
    return accrFactor;
    }
```

Therefore, the *accrFactor* is initially retrieved from the *baseCurve* through the *discount* method and then multiplied by the compounding factor, where the exponential factor is the integrated instantaneous basis. Note : the sign of the abcd factor depends on whether or not we are calibrating with respect to a *baseCurve* with a greater *tenor* with respect to the benchmark curve whom base is searched.

4.2 Abcd Tenor Basis

AbcdTenorBasis is a child class of the above presented *TenorBasis* class. The main feaure exploited in the excel framework is its constructor:

```
AbcdTenorBasis::AbcdTenorBasis(shared_ptr<IborIndex> iborIndex,
                      boost::shared_ptr<IborIndex> baseIborIndex,
                      Date referenceDate,
                      bool isSimple,
                      const std::vector<Real>& coeff)
    : TenorBasis(4, iborIndex, baseIborIndex, referenceDate) {
        //std::vector<Real> y = inverse(coeff);
        std::vector<Real> y = coeff;
        arguments_[0] = ConstantParameter(y[0], NoConstraint());
        arguments_[1] = ConstantParameter(y[1], NoConstraint());
        arguments_[2] = ConstantParameter(y[2], NoConstraint());
        arguments_[3] = ConstantParameter(y[3], NoConstraint());
        isSimple_ = isSimple;
        generateArguments();
    }
```

It takes a vector of guess *coeff* and it stores them in an vector of object: *arguments_*. Then, in order to choose the correct algorithm, it asks whether or not the calibration is on simple basis and then it generates the problem parameters:

```
void AbcdTenorBasis::generateArguments() {
    std::vector<Real> x(4);
    x[0] = arguments_[0](0.0);
    x[1] = arguments_[1](0.0);
    x[2] = arguments_[2](0.0);
    x[3] = arguments_[3](0.0);
    //std::vector<Real> y = direct(x);
    std::vector<Real> y = x;
    if (isSimple_) {
        basis_ = shared_ptr<AbcdMathFunction>(
            new AbcdMathFunction(y[0], y[1], y[2], y[3]));
        vector<Real> c =
        basis_->definiteDerivativeCoefficients(0.0, tau_);
        c[0] *= tau_;
        c[1] *= tau_;
        // unaltered c[2] (the c in abcd)
        c[3] *= tau_;
        instBasis_ =
        shared_ptr<AbcdMathFunction>(new AbcdMathFunction(c));
    } else {
        instBasis_ = shared_ptr<AbcdMathFunction>(
            new AbcdMathFunction(y[0], y[1], y[2], y[3]));
        vector<Real> c =
        instBasis_->definiteIntegralCoefficients(0.0, tau_);
        c[0] /= tau_;
        c[1] /= tau_;
        // unaltered c[2] (the c in abcd)
        c[3] /= tau_;
        basis_ =
        shared_ptr<AbcdMathFunction>(new AbcdMathFunction(c));
    }
    }
```

All the *arguments_* entries are written in the vector x, then according to the type of searched basis (*isSimple* or not) it creates, in the continuous basis specific case, an *instBasis_* and *AbcdMathFunction* with the given parameters:

```
AbcdMathFunction::AbcdMathFunction(Real aa, Real bb,
Real cc, Real dd)
    : a_(aa), b_(bb), c_(cc), d_(dd), abcd_(4), dabcd_(4) {
        abcd_[0]=a_;
        abcd_[1]=b_;
        abcd_[2]=c_;
        abcd_[3]=d_;
        initialize_();
    }
```

It creates *abcd_* and *dabcd_* vectors, where *dabcd_* is the vector of derivative coefficients, it sets *abcd_* and calls *initialize*:

```
  void AbcdMathFunction::initialize_() {
        validate(a_, b_, c_, d_);
        da_ = b_ - c_*a_;
        db_ = -c_*b_;
        dabcd_[0]=da_;
        dabcd_[1]=db_;
        dabcd_[2]=c_;
        dabcd_[3]=0.0;

        pa_ = -(a_ + b_/c_)/c_;
        pb_ = -b_/c_;
        K_ = 0.0;

        dibc_ = b_/c_;
        diacplusbcc_ = a_/c_ + dibc_/c_;
    }
```

Before defining a series of variables that will be exploited in the algorithm, it calls *AbcdMathFunction::validate* that simply checks that specific abcd framework features are matched. Going back to *generateArguments*, another abcd, but simple one, is instantiated transforming the *instBasis_* parameters.

Given this framework, the *AbcdTenorBasis* can be calibrated according to the parent class above explained method.

4.3 Discount Corrected Term Structure

Differently from the above shown framework, the *DiscountCorrectedTermStructure* one has been already too well explained in [1] (Thank you for this book Luigi), because it is a particular case of the famous *PiecewiseYieldCurve* scheme. However, it is worth to further explain some details of this specific implementation:

```
  class DiscountCorrectedTermStructure :
          public YieldTermStructure,
          protected InterpolatedCurve<Linear>,
          public LazyObject {
    public:
    typedef Discount traits_type;
    typedef Linear interpolator_type;
    DiscountCorrectedTermStructure(
    const Handle<YieldTermStructure>& bestFitCurve,
    const std::vector<boost::shared_ptr<RateHelper> >& instruments,
    Real accuracy = 1.0e-12);
    const Date& referenceDate() const;
    DayCounter dayCounter() const;
    Calendar calendar() const;
    Natural settlementDays() const;
    Date maxDate() const;
    const std::vector<Time>& times() const;
    const std::vector<Date>& dates() const;
    const std::vector<Real>& data() const;
        void update();
    private:
    DiscountFactor discountImpl(Time) const;
    void performCalculations() const;
    // data members
    Handle<YieldTermStructure> bestFitCurve_;
    std::vector<boost::shared_ptr<RateHelper> > instruments_;
    Real accuracy_;
    mutable std::vector<Date> dates_;

    // bootstrapper classes are declared as friend to manipulate
    // the curve data. They might be passed the data instead, but
    // it would increase the complexity---which is high enough
    // already.
    friend class IterativeBootstrap<DiscountCorrectedTermStructure>;
    friend class BootstrapError<DiscountCorrectedTermStructure>;
    IterativeBootstrap<DiscountCorrectedTermStructure> bootstrap_;
    };
```

Firstly, the *traits_type* is a discount and the *interpolator_type* is linear, it means that the bootstrapping is performed on the pseudo discount factor with a linear interpolation. Furthermore, the other method that matter is:

```
    DiscountFactor DiscountCorrectedTermStructure:
    discountImpl(Time t) const {
        calculate();
        DiscountFactor d = bestFitCurve_->discount(t, true);
        Real k = interpolation_(t, true);
        return k*d;
    }
```

The two of this feature together means that: the algorithm starts the bootstrap process from a pillar guess that is close to a possible value of the discount factor (1, also the optimal correction factor value). Then, inside the *IterativeBootstrap* code, through a *BootstrapError:: operator*:

```
        template <class Curve>
    Real BootstrapError<Curve>::operator()(Real guess) const {
        Traits::updateGuess(curve_->data_, guess, segment_);
        curve_->interpolation_.update();
        return helper_->quoteError();
    }
```

it updates the curve with the new guess, it interpolates with a linear interpolator, updates the observers and returns the error, but the *quoteError()* interface is:

```
    Real BootstrapHelper::quoteError()
    const { return quote_->value() - impliedQuote(); }
```

Therefore, as in the previous framework an *impliedQuote* is retrieved, that leads to a call to the above shown *discountImpl* method, that interpolates the correction factors that compose the curve that is currently bootstrapped. To better explain the problem that has been solved with this algorithm, it is possible to think in this way: "given a curve that returns a certain fixing value for a certain curve pillar (the base one with respect to the basis is searched)), what is the correction that needs to be applied in order to perfectly repricing the observed quotes?". Or better: when *discountImpl* is invoked, it doesn't know that there is already a discount from $bestFitCurve\_$, therefore it will just try to solve its problem and, given the presence of this particular *discountImpl* implementation that provide a basis value, this will lead it implicitly retrieving a bootstrapped curve of correction factors.

Note: *QuantLib* is **The QuantLib**.

Chapter 5

Abcd Reparameterization

5.1 Parameters conversion

Going through the practical side of the exhibited idea, problems are encountered retrieving b and c functional forms.

Given that the choice of how to specify the parameters is ambiguous, the equations of $s_x(t_{max})$ and t_{max} have been plugged into a system, in order to explicit c and b:

$$\begin{cases} s_x(t_{max}) = \frac{b_x}{c_x} e^{(\frac{a_x c_x}{b_x} - 1)} + d_x & (15) \\ t_{max} = \dfrac{1}{c_x} - \dfrac{a_x}{b_x} & (12) \end{cases}$$

Note: to achieve better readability during this mathematical steps:

- $s_x(t_{max})$ will be written as: s;
- t_{max} will be written as: t;
- all the parameters will be expressed without considering the tenor, in a generic way;

Starting from (12):

$$t = \frac{1}{c} - \frac{a}{b}$$

$$\frac{1}{c} = t + \frac{a}{b}$$

This equation is obtained:

$$c = \frac{b}{a + tb} \tag{5.1}$$

Then, working on (15):

$$s = \frac{b}{c} e^{(\frac{ac}{b} - 1)} + d$$

and plugging (5.1), the following is obtained:

$$s = (bt + a) e^{(\frac{a}{bt+a} - 1)} + d$$

but b cannot be retrieved because of the presence of a form such as (9).
A further solution can be to retrieve b from (12):

$$b = \frac{ac}{1 - tc} \tag{5.2}$$

and then substitute in (15), obtaining:

$$s = \frac{a}{1 - tc} e^{(-tc)} + d$$

However, the above mentioned problem persists (9).

Another solution can be to consider that because of the nested calibration t_{max} is a constant, therefore (15) simply becomes:

$$s = (a + bt) e^{(-ct)} + d \tag{5.3}$$

Rewriting the above equation, b is obtained:

$$b = \frac{s - d}{te^{(-ct)}} - \frac{a}{t}$$

Plugging (5.1):

$$b = \frac{s - d}{te^{(-\frac{bt}{a+tb})}} - \frac{a}{t}$$

Unfortunately, once again, the problem remains.
In the end, even when plugging (5.2) in (5.3) the problem still remains:

$$s = (a + \frac{act}{1 - tc}) e^{(-ct)} + d$$

These attempts lead to a mandatory adjustment of the main idea previously exhibited and are exposed in the following paragraphs.

5.2 Idea adjustment

Given the above mentioned considerations, it is necessary to rethink the main idea of the model reparameterization that has to be shifted from the recalibration based on (16) to an input form such:

$$a_x, c_x, t_{max}, d_x \tag{5.4}$$

This allows to work on the parameters that has been spotted that are essential for the new framework, i.e. c and t_{max} along with a and d that have a clear financial meaning, without facing problem as the one in the previous section.

Chapter 6

Abcd Framework+

6.1 Nested Calibration

The general idea at the basis of nested calibration is that the minimization problem can be split in two problems. The first minimization is only with respect to a certain variable k. Once fixed, it can act as an implied bound for the other parameters of the model. The second step is to internally minimize with respect to other model parameters. In the specific case of this research, it is interesting to fix either t_{max} or c or both. therefore the designed framework needs to have the flexibility in order to manage these user requests.

6.2 Calibration data

In order to exploit the $AbcdTenorBasis :: calibrate$ method, already shown in chapter 4, in a multiple model framework it is necessary to wrap its parameters in a repository object (in the end should be clear why). For this reason the object that follows has been created:

```
class CalibratedModel::CalibrationData {
public:
//some common methods
virtual const std::vector < boost::shared_ptr<RateHelper>> rateHelper()= 0;
virtual OptimizationMethod method()= 0;
virtual const EndCriteria endCriteria() = 0;
virtual const std::vector<Real> weights() = 0;
virtual const std::vector<bool> fixParameters()) = 0;
virtual int globalParameter(int innerErrorNumber) = 0;
virtual void globalParameterOrdering(std::vector<int> globalParameter)= 0;
virtual Real globalErrorWeight() = 0;
```

```
//attributes
protected:
const std::vector<boost::shared_ptr<RateHelper> > helpers_,
OptimizationMethod method_,
const EndCriteria endCriteria_,
const std::vector<Real> weights_,
const std::vector<bool> fixParameters_;
std::vector<int> globalParameter_;
Real globalErrorWeight_;
};
```

as it is possible to see this class just stores some attributes, that are protected such that derived classes can access them, and it designs virtual functions in order to access them.

Moreover, this function has been thought to be really general being an inner class of the CalibratedModel. Therefore may be some doubts about the correctness of storing such virtual functions, that can be too much specific for the abcd case, anyway the author still think that can work.

Being more specific, this class is implemented by :

```
class AbcdCalibrationData:public CalibratedModel::CalibrationData {
public:
AbcdCalibrationData(const std::vector<int> & globalParameter;
Real globalErrorWeight;
const std::vector<boost::shared_ptr<RateHelper> >& helpers,
OptimizationMethod& method,
const EndCriteria& endCriteria,
const std::vector<Real>& weights,
const std::vector<bool>& fixParameters);

//methods
const std::vector < boost::shared_ptr<RateHelper> rateHelper();
OptimizationMethod method();
const EndCriteria endCriteria();
const std::vector<Real> weights();
const std::vector<bool> fixParameters());
int globalParameter(int innerErrorNumber);
void globalParameterOrdering(std::vector<int> globalParameter);
Real globalErrorWeight();
};
```

the methods are:

```

const std::vector < boost::shared_ptr<RateHelper>> AbcdTenorBasis::AbcdCalibrationI
return rateHelper_;
};
OptimizationMethod AbcdTenorBasis::AbcdCalibrationData  method() {
return method_;
```

```
};
const EndCriteria  AbcdTenorBasis::AbcdCalibrationData  endCriteria() {
return endCriteria_;
};
const std::vector<Real>  AbcdTenorBasis::AbcdCalibrationData  weights()
{
return weights_;
};
const std::vector<bool>  AbcdTenorBasis::AbcdCalibrationData  fixParameters() {
return fixParameters_;
};
const std::vector<int> AbcdTenorBasis::AbcdCalibrationData  globalParameter(int inn
{
return globalParameter_[innerErrorNumber];
};
Real AbcdTenorBasis::AbcdCalibrationData  globalErrorWeight()  {
return globalErrorWeight_;
};
```

The only one that merits to be commented is the *globalParameter* method. Each time that a new global error is created it needs to know what is the parameter with respect to the calibration process is fixed, therefore when it asks for *globalParameter_* it wants information about the global parameter specific for the computed error:

```
const std::vector<int> AbcdTenorBasis::AbcdCalibrationData  globalParameter(int inn
{
        return globalParameter_[innerErrorNumber];
};
```

Therefore, the algorithm in order to assits the user accepts the *globalParameter* inputs in an opposite order with respect to the one that it needs (later will be clearer why), there is a bit of manipulation encapsulated in the method:

```
void globalParameterOrdering(std::vector<int> globalParameter) {

int j = globalParameter.end();

globalParameter_.resize(globalParameter.size());

for (std::vector<int>::iterator  i = globalParameter.begin();i != globalParameter.en
{
        globalParameter_[i] = globalParameter[j];
        j--;
}

}
```

Therefore, the constructor works as follows:

```
AbcdTenorBasis::AbcdCalibrationData(std::vector<int> & globalParameter;
Real globalErrorWeight;
const std::vector<boost::shared_ptr<RateHelper> >& helpers,
OptimizationMethod& method,
const EndCriteria& endCriteria,
const std::vector<Real>& weights,
const std::vector<bool>& fixParameters)
: globalErrorWeight_(globalErrorWeight),helpers_(helpers), method_(method),
endCriteria_(endCriteria), weights_(weights), fixParameters_(fixParameters) {

globalParameterOrdering(std::vector<int> globalParameter);

}
```

as above mentioned, it reorders the globalParameter, for matching the ”chinese boxes” scheme that will be later explained.

6.3 Calibrate data

While *CalibrationData* guards the inputs for the *CalibratedModel*, this class contains the inputs in order to make the global calibrator method *calibrate* working:

```
class CalibratedModel::CalibrateData {
public:
CalibrateData(
boost::shared_ptr<Solver1D> & solver,
Real & accuracy,
Real & guess,
Real & min,
Real & max);

boost::shared_ptr<Solver1D> solver()const;
Real accuracy()const;
Real guess()const;
Real min()const;
Real max()const;

private:

boost::shared_ptr<Solver1D> solver_;
Real accuracy_;
Real guess_;
Real min_;
Real max_;
};
```

Given that its functionality is strictly related with the *calibrate* method, it needs to be included in the *CalibratedModel* object as an inner class too. Should be noted that also the solver has been stored as parameter, not too nice choice, but it is useful in the extend of giving the user the possibility to change optimization method.

As for the *globalParameter_* also this object refers to the number of global errors, therefore there will be a *CalibrateData* object for any error, indeed *GlobalError* takes a strip of them.

It is possible to divide the implementation in two main idea:

- the objects that work for the specific *calibrate* of each *CalibratedModel* , as *CalibrationData*, that will be invoked with its referred method and they are the same for each inner error.
- the objects that work for the global *calibrate*, therefore they will be invoked with its referred method and they can change for each inner error.

6.4 Global error

As for *CalibrationData* and *CalibrateData*, *GlobalError* has been designed in order to be part of *CalibratedModel* object:

```
class CalibratedModel::GlobalError {

public:
GlobalError(
//innerErrorNumber plays it's fundamental because helps to select data according to
int & innerErrorNumber,
//it needed a vector of CalibratedModel, because we are retrieving the error from m
        std::vector<boost::shared_ptr<CalibratedModel>>& calibratedModel,
//different CalibratedModel need different calibration data
        std::vector<boost::shared_ptr<CalibrationData>>& calibrationData,
//if there is more than one global error: more error needs more calibrate and there
        std::vector<boost::shared_ptr<CalibrateData>>& calibrateData)
//methods

virtual void error()= 0;
virtual void updateGuess(Real guess) const = 0;
virtual void loadCalibrationData() const = 0;
virtual void innerError()= 0;
void errorFork()const;
void calibrationDataOrdering(std::vector<boost::shared_ptr<CalibrateData>> calibrat
std::vector<boost::shared_ptr<CalibratedModel>> calibratedModel()const;
std::vector<boost::shared_ptr<CalibrationData>> calibrationData()const;
std::vector<boost::shared_ptr<CalibrateData>> calibrateData()const;

Real GlobalError::operator()(Real guess);

private:
std::vector<boost::shared_ptr<CalibratedModel>> calibratedModel_;
std::vector<boost::shared_ptr<CalibrationData>> calibrationData_;
std::vector<boost::shared_ptr<CalibrateData>> calibrateData_;
/*if calibrationData_ makes sense as wrapped in GlobalError, I think that calibrate
rather than the right approach, anyway.. it should work*/
Real error_;
int innerErrorNumber_ ;
}
```

The constructor simply forwards its inputs to the GlobalError attributes:

```
CalibratedModel::GlobalError(int &innerErrorNumber,
std::vector<boost::shared_ptr<CalibratedModel>>& calibratedModel,
std::vector<boost::shared_ptr<CalibrationData>>& calibrationData,
std::vector<boost::shared_ptr<CalibrateData>>& calibrateData)
:innerErrorNumber_(innerErrorNumber), calibratedModel_(calibratedModel), calibratio
```

Crucial role is played by $GlobalError :: operator()$:

```
Real CalibratedModel::GlobalError::operator()(Real guess) const {

//put the global guess in the global parameter position ( that has to be fixed in
this->updateGuess(guess);

//decide whether to generate another global error or directly calibrate
errorFork();

//get the global error
error_ = 0;

this->error()

return error_;
}
```

it is the core: this structure mimics the famous $IterativeBootstrapp ::$ $calculate()$ method, because the QuantLib solver uses the *operator()* for retrieving the error, therefore it is the same. The operator updates the guess with *updateGuess* according to the specific *GlobalError*. Then *errorFork* evaluates if it needs to create another *GlobalError* or if it is possible to calibrate:

```
void errorFork() const {

if innerErrorNumber_ > 1{
```

```
innerError();
}
else
{
loadCalibrationData();
}

};
```

Here is where it is possible to work in order to create other inner errors (as will be clearer later).

Looking at the *AbcdGlobalError* object, it simply implements all the specific methods designed in *GlobalError*. The constructor is:

```
AbcdGlobalError::AbcdGlobalError(int &innerErrorNumber,
std::vector<boost::shared_ptr<CalibratedModel>>& calibratedModel,
std::vector<boost::shared_ptr<CalibrationData>>& calibrationData,
std::vector<boost::shared_ptr<CalibrateData>>& calibrateData)
:GlobalError(innerErrorNumber, calibratedModel, calibrationData, calibrateData) {
calibrateDataOrdering(calibrateData)
};
```

that forwards to *GlobalError* its required inputs and it orders, as already seen for *globalParameters_* and according to the same ratio, *calibrateData*:

```
void calibrateDataOrdering(std::vector<boost::shared_ptr<CalibrateData>> calibrateD
{

int j = calibrateData.end();

calibrateData_.resize(calibrateData.size());

for (std::vector<int>::iterator i = calibrateData.begin();i != calibrateData.end();
{
calibrateData_[i] = calibrateData[j];
j--;
}

}
```

Then, it implements the methods embedded in *operator()*:

```
void AbcdGlobalError::updateGuess( Real guess) const {

for (std::vector<int>::iterator i = calibratedModel_.begin();i != calibratedModel_
{
calibratedModel_[i]->arguments_[calibrationData_[i].globalParameter(this->innerErro

}
```

for each calibrated model, that needs to be globally calibrated, it poses the *CalibratedModel::arguments_* in the position indicated by the *globalParameter_* entry, specific of the researched global error (*innerErrorNumber_*), equals to the global guess. It means that for each inner error is possible to change the *globalParameter_[i]* according to the considered error, this information is embedded in *innerErrorNumber_* that is univocal for each inner *GlobalError* created.

The *loadCalibrationData* method both loads data for calibrating and calibrates:

```
void AbcdGlobalError::loadCalibrationData() const{

for (std::vector<int>::iterator i = calibratedModel_.begin();i != calibratedModel_
{
//here the model is AbcdtenorBasis therefore it uses AbcdTenorBasis::calibrate
calibratedModel_[i]->calibrate(calibrationData_[i]->rateHelper(),
calibrationData_[i]->method(),
calibrationData_[i]->endCriteria(),
calibrationData_[i]->weights(),
calibrationData_[i]->fixParameters())
}
```

In the end an error is retrieved exploiting:

```
void AbcdGlobalError::error() {

for (std::vector<int>::iterator i = calibratedModel_.begin();i != calibratedModel_
{
error_++ = calibrationData_[i]->globalErrorWeight()*calibratedModel_[i]->value(cali
params(), calibrationData_[i].rateHelper());
}
}
```

that exploits the *value* method of *CalibratedModel* in order to get the error and it allows to manipulate it through *globalErrorWeight*.

6.5 Calibrate

The *calibrate* method is the user most important function: once created the *GlobalError*, it passes the object to the *calibrate* method that follows:

```
void CalibratedModel::calibrate(
boost::shared_ptr<GlobalError> globalError
) {

//create the error
//solver
int i = globalError->innerErrorNumber_;
globalError->calibrateData_[i]->solver_->solve(*globalError, globalError->calibrate
globalError->calibrateData_[i]->min(), globalError->calibrateData_[i]->max());
}
```

Here the code looks awful, but there is a reason why to do this.

It asks *globalError* the number of global errors that need to be created, it is extremely important *innerErrorNumber_* because of its unique relation with the current error. Then, it uses this number for surfing among the different errors and corresponding methods with an inverse index structure. For instance: if there are 4 errors, the user passes 4 strips of parameter. Because of the ordering method, the first parameter will correspond to the fourth entry of the specific parameter (*globalParameter_* or *calibrateData_*). Given *innerErrorNumber_*, remembering that each time inner errors are created with a less one unit of it, it means that when *innerErrorNumber_* is 4 the algorithm is dealing with the first error, so it has to takes the parameters in position 4 . Probably there is a clever idea for exploit this information, like, instead of reordering, to ask the size of *calibrateData_* to make the difference between it (that is always the same) and the current *innerErrorNumber_*, that it means " if the size is 4, *innerErrorNumber_* 4 then takes the first value of parameter" and so on. Maybe was cleaver, but always based on the same ratio that *innerErrorNumber_* is the light in the dark "error" night.

The reason why was created *calibrateData_* should be now clear,because of the specific implementation of *innerError* :

```
void errorFork() const {

if innerErrorNumber_ > 1{
innerError();
}
else
{
loadCalibrationData();
}

};
```

i.e.:

```
void innerError() {
int innerErrorNumberCounter = this->innerErrorNumber_ - 1;
AbcdGlobalError Error(innerErrorNumberCounter,this->calibratedModel_, this->calibra
calibrate(Error);

}
```

was necessary to guard all the global *calibrate* inputs in a mobile object in order to exploit a "chinese boxes" scheme it. Indeed, in *errorFork* it is possible to fork the current worked error and create through *innerError* another *AbcdGlobalError*. This new global error has one less unit of $innerErrorNumber\_$, therefore it is possible to uniquely define it (fundamental!). Moreover, in order to create the error is necessary accessing the required data, therefore all the machinery was built to come at this point, i.e : the construction of n global errors that each time allows to calibrate the model fixing such parameter that is externally changed, understanding what error is dealt with thanks to the specific value of the n $innerErrorNumber\_$, reaching the end of the algorithm only when $innerErrorNumber\_$ equals to 1.

A review of code should simplyfing what is possible anf maybe also better design the above explained classes

Therefore this scheme allows to nested calibrate t_{max} but also nested calibrate t_{max} while internally nested calibrate c and viceversa.

The main cost in order to achieve these goals was to accumulate a lot of information inside the *GlobalError*, maybe also missing the coherence between the modelled object and its theoretical function (always bad), and a bit of complexity.

Chapter 7

Fixing of c: an empirical analysis

7.1 Globally shared c

This section tests effects of c fixing, which grants that all relative basis are abcd, on the model fitting quality, for the continuous calibration with both incremental and not incremental calibration method. Given the spreadsheets used to get the results exhibited in the chapter 3, with a little modification it is possible to design the algorithm that allows to globally fix c.

7.1.1 Not incremental approach

Starting from the not incremental method available in the spreadsheet "BasisCalibration", in the spreadsheet named "$BasisCalibration\_fixed\_c$" the idea is to create a control panel which allows to work on all the other sheets. Therefore, the cells which contain c_x parameters, that feed each calibrator, take their value from a common cell, which is the guessed global c. Moreover, have been also fixed their boolean values which indicates whether or not the respective parameters should be calibrated during the calibration process. Successively, starting from the guessed c the Excel Solver has been requested to change c in order to minimize the quantity:

$$\sum_{j=1}^{4} \sqrt{\frac{\sum_{i=1}^{n} (\hat{q_{j,i}} - q_{j,i})^2}{n}} \tag{7.1}$$

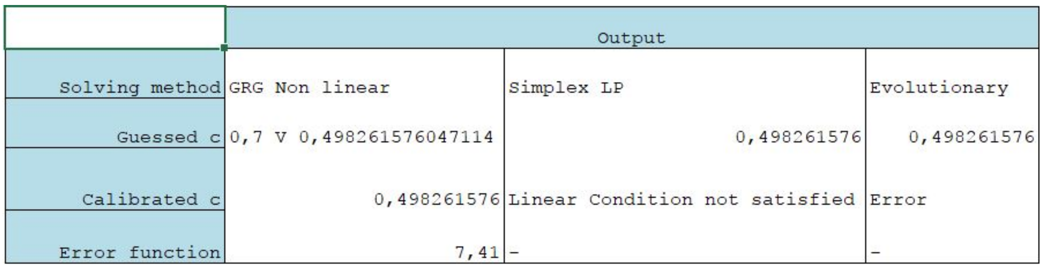
where:

- q is the market quote;
- $\hat{q}$ is the repriced quote from model;
- j indicates the basis;
- i indicates the $i - basis$;

The implementation choices are:

1. guesses of the parameters have been chosen picking from the results of the previous paper, that exhibits the best fit as possible according with this framework, while for c have been manually tempted different values and in the end the empirical attempts lead to a value of 0.7, which avoid the abortion of the output;
2. if a curve can't be calibrated, the algorithm fails;
3. the algorithm has been tried for each available excel solver method;

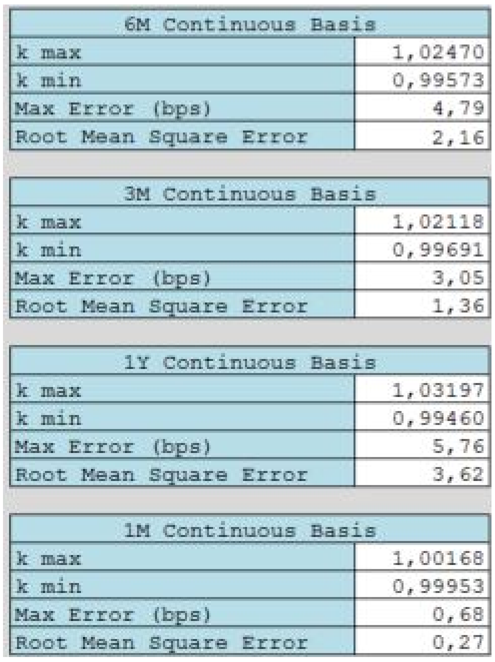
The outputs follow:



	Output		
Solving method	GRG Non linear	Simplex LP	Evolutionary
Guessed c	0,7 V 0,498261576047114	0,498261576	0,498261576
Calibrated c	0,498261576	Linear Condition not satisfied	Error
Error function	7,41	-	-

Figure 7.1: c fixing output

As it is possible to see, the Solver works only with the "GRG non linear" algorithm, probably because the problem is highly not linear (yeah, not so smart consideration). However, considering that the distribution of the correction factor is the parameter that represents the goodness of the calibration in the abcd framework, the results are excellent:



6M Continuous Basis	
k max	1,02470
k min	0,99573
Max Error (bps)	4,79
Root Mean Square Error	2,16

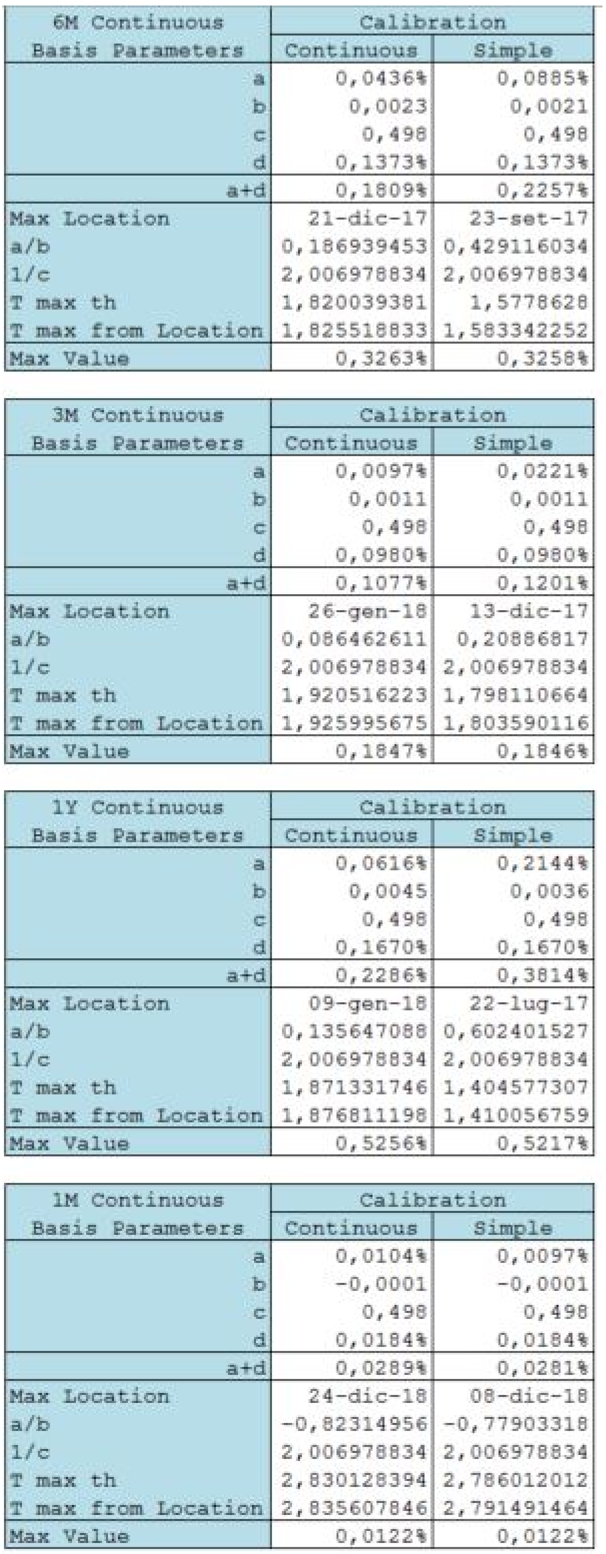
3M Continuous Basis	
k max	1,02118
k min	0,99691
Max Error (bps)	3,05
Root Mean Square Error	1,36

1Y Continuous Basis	
k max	1,03197
k min	0,99460
Max Error (bps)	5,76
Root Mean Square Error	3,62

1M Continuous Basis	
k max	1,00168
k min	0,99953
Max Error (bps)	0,68
Root Mean Square Error	0,27

Figure 7.2: Correction factors k from not incremental calibration with global c

Range of k values is closed to 1 that is the best value because indicates that no correction are needed. Moreover, the statistics shows an interesting features for the following of the research:



6M Continuous Basis Parameters	Calibration	
	Continuous	Simple
a	0,0436%	0,0885%
b	0,0023	0,0021
c	0,498	0,498
d	0,1373%	0,1373%
a+d	0,1809%	0,2257%
Max Location	21-dic-17	23-set-17
a/b	0,186939453	0,429116034
1/c	2,006978834	2,006978834
T max th	1,820039381	1,5778628
T max from Location	1,825518833	1,583342252
Max Value	0,3263%	0,3258%

3M Continuous Basis Parameters	Calibration	
	Continuous	Simple
a	0,0097%	0,0221%
b	0,0011	0,0011
c	0,498	0,498
d	0,0980%	0,0980%
a+d	0,1077%	0,1201%
Max Location	26-gen-18	13-dic-17
a/b	0,086462611	0,20886817
1/c	2,006978834	2,006978834
T max th	1,920516223	1,798110664
T max from Location	1,925995675	1,803590116
Max Value	0,1847%	0,1846%

1Y Continuous Basis Parameters	Calibration	
	Continuous	Simple
a	0,0616%	0,2144%
b	0,0045	0,0036
c	0,498	0,498
d	0,1670%	0,1670%
a+d	0,2286%	0,3814%
Max Location	09-gen-18	22-lug-17
a/b	0,135647088	0,602401527
1/c	2,006978834	2,006978834
T max th	1,871331746	1,404577307
T max from Location	1,876811198	1,410056759
Max Value	0,5256%	0,5217%

1M Continuous Basis Parameters	Calibration	
	Continuous	Simple
a	0,0104%	0,0097%
b	-0,0001	-0,0001
c	0,498	0,498
d	0,0184%	0,0184%
a+d	0,0289%	0,0281%
Max Location	24-dic-18	08-dic-18
a/b	-0,82314956	-0,77903318
1/c	2,006978834	2,006978834
T max th	2,830128394	2,786012012
T max from Location	2,835607846	2,791491464
Max Value	0,0122%	0,0122%

Figure 7.3: Parameters from not incremental calibration with global c

for 6M, 3M and 1Y the $t_{max}(x)$ values are closed to the other and this may mean that a global shared value of t_{max} is a sane idea.

The only problem that arises is that with respect to the legacy curve, the new basis seems losing fitting on the legacy one. Anyway it should not be a problem in the extend that is valid the idea that legacy curve brings with itself more noise than signal with respect to an abcd basis.

Graphically, it appears that the matter, already encountered in [2], about the shape of 1M remains:

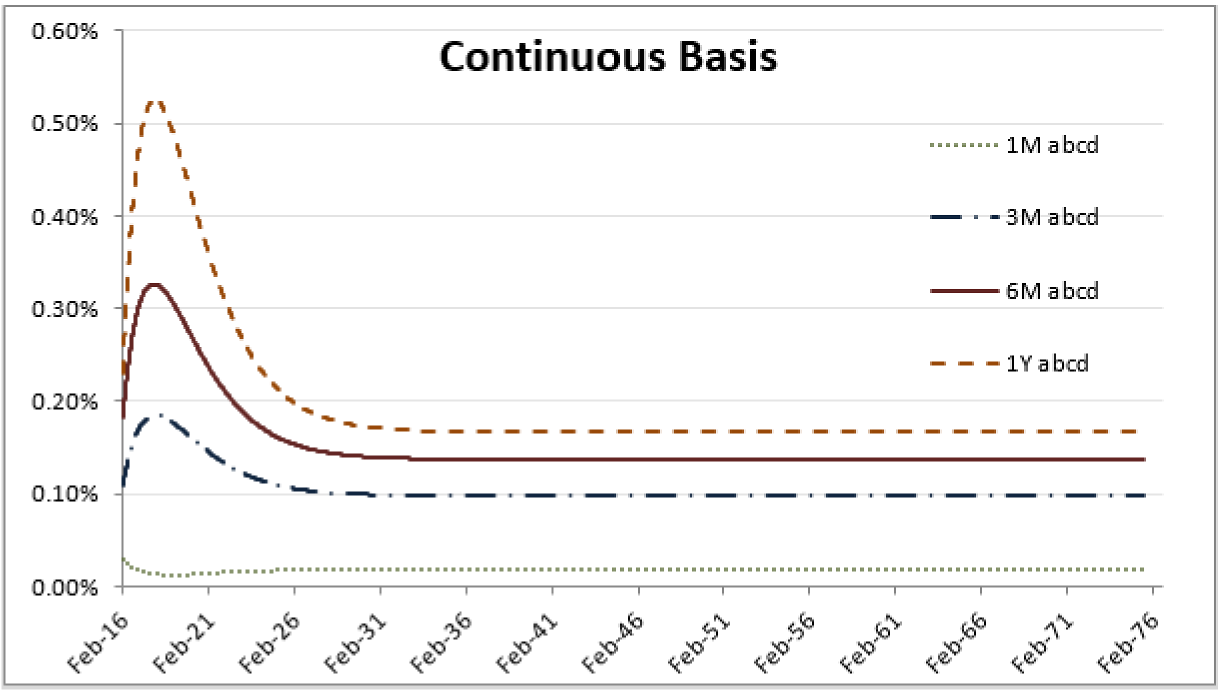
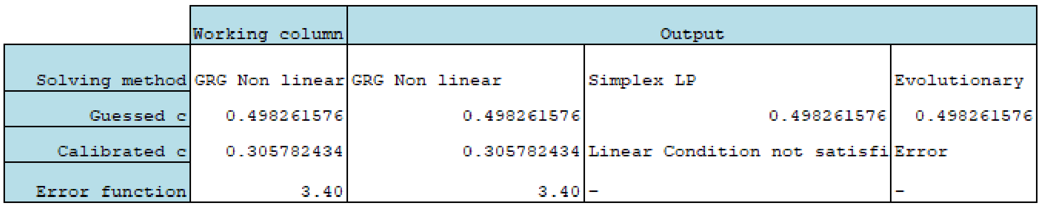


Figure 7.4: Absolute basis s_x from not incremental calibration

7.1.2 Incremental approach

Swapping to an incremental approach, the same modifications above reported are employed in order to get the searched result starting from the spreadsheet "*BasisCalibration_Incremental_fixed_c*".

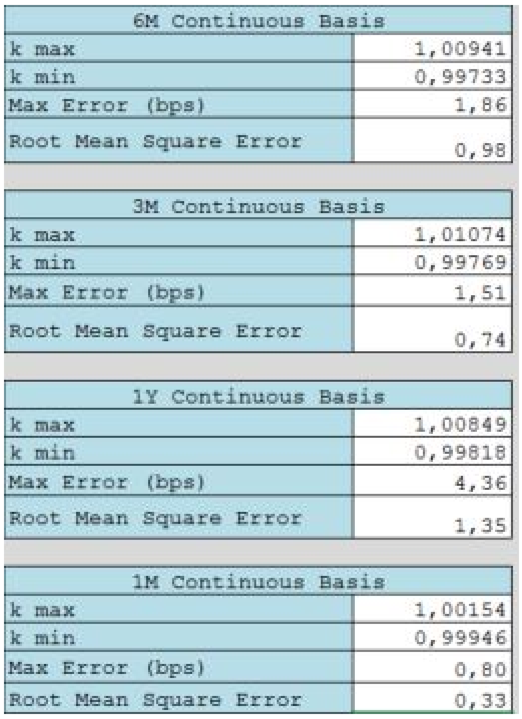
The outputs follow:



	Working column	Output		
Solving method	GRG Non linear	GRG Non linear	Simplex LP	Evolutionary
Guessed c	0.498261576	0.498261576	0.498261576	0.498261576
Calibrated c	0.305782434	0.305782434	Linear Condition not satisfi	Error
Error function	3.40	3.40	-	-

Figure 7.5: c fixing output

As it is possible to appreciate, results come once again only from the "GRG non linear" excel optimization. However, considering that the distribution of the correction factor is what was looking for, the results are still excellent:



6M Continuous Basis	
k max	1,00941
k min	0,99733
Max Error (bps)	1,86
Root Mean Square Error	0,98

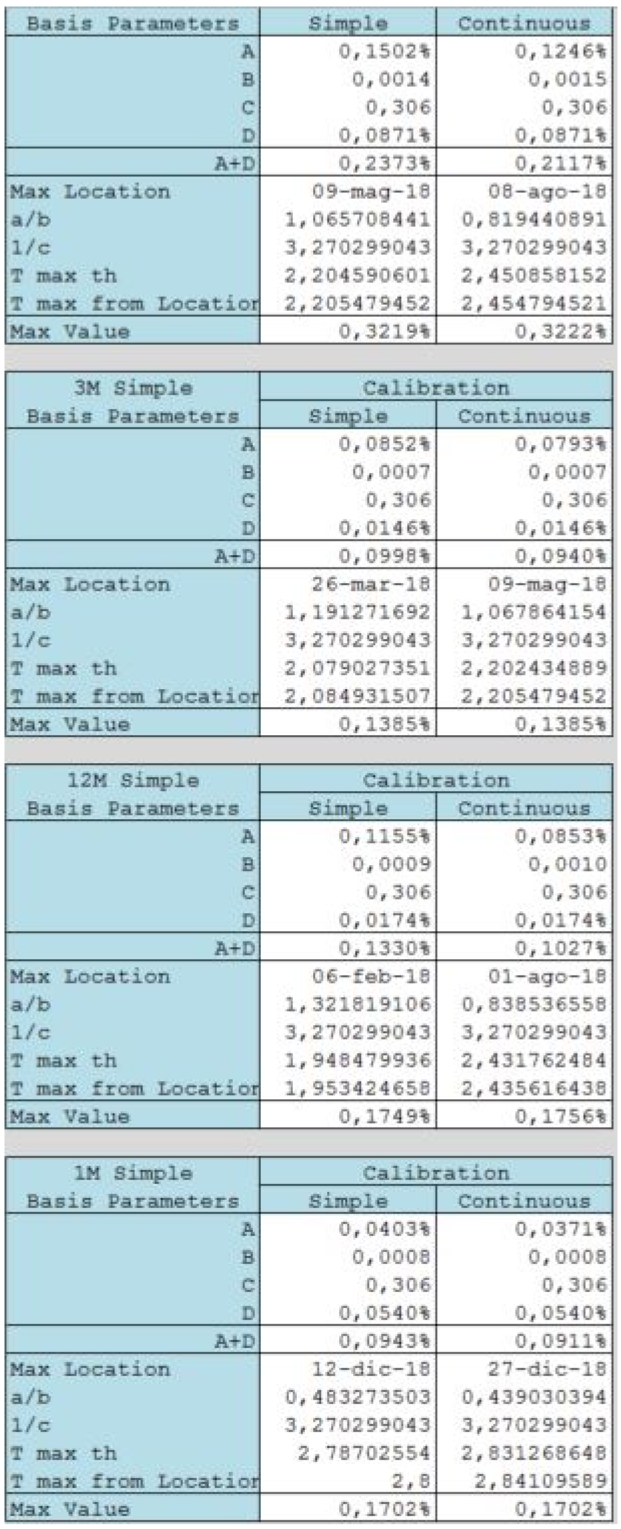
3M Continuous Basis	
k max	1,01074
k min	0,99769
Max Error (bps)	1,51
Root Mean Square Error	0,74

1Y Continuous Basis	
k max	1,00849
k min	0,99818
Max Error (bps)	4,36
Root Mean Square Error	1,35

1M Continuous Basis	
k max	1,00154
k min	0,99946
Max Error (bps)	0,80
Root Mean Square Error	0,33

Figure 7.6: Correction factors k from incremental calibration with global c

Range of k values is closed to 1 that is the best value for the calibration. Unfortunately, the statistics don't are as good as before:



Basis Parameters	Simple	Continuous
A	0,1502%	0,1246%
B	0,0014	0,0015
C	0,306	0,306
D	0,0871%	0,0871%
A+D	0,2373%	0,2117%
Max Location	09-mag-18	08-ago-18
a/b	1,065708441	0,819440891
1/c	3,270299043	3,270299043
T max th	2,204590601	2,450858152
T max from Locatior	2,205479452	2,454794521
Max Value	0,3219%	0,3222%

3M Simple	Calibration	
Basis Parameters	Simple	Continuous
A	0,0852%	0,0793%
B	0,0007	0,0007
C	0,306	0,306
D	0,0146%	0,0146%
A+D	0,0998%	0,0940%
Max Location	26-mar-18	09-mag-18
a/b	1,191271692	1,067864154
1/c	3,270299043	3,270299043
T max th	2,079027351	2,202434889
T max from Locatior	2,084931507	2,205479452
Max Value	0,1385%	0,1385%

12M Simple	Calibration	
Basis Parameters	Simple	Continuous
A	0,1155%	0,0853%
B	0,0009	0,0010
C	0,306	0,306
D	0,0174%	0,0174%
A+D	0,1330%	0,1027%
Max Location	06-feb-18	01-ago-18
a/b	1,321819106	0,838536558
1/c	3,270299043	3,270299043
T max th	1,948479936	2,431762484
T max from Locatior	1,953424658	2,435616438
Max Value	0,1749%	0,1756%

1M Simple	Calibration	
Basis Parameters	Simple	Continuous
A	0,0403%	0,0371%
B	0,0008	0,0008
C	0,306	0,306
D	0,0540%	0,0540%
A+D	0,0943%	0,0911%
Max Location	12-dic-18	27-dic-18
a/b	0,483273503	0,439030394
1/c	3,270299043	3,270299043
T max th	2,78702554	2,831268648
T max from Locatior	2,8	2,84109589
Max Value	0,1702%	0,1702%

Figure 7.7: Parameters from incremental calibration with global c

for 6M, 3M and 1Y the $t_{max}(x)$ values are not so closed to the other, but the $t_{max}(1M)$ is nearer to them , in the next chapter this issue will be addressed. Graphically, the curves shapes:

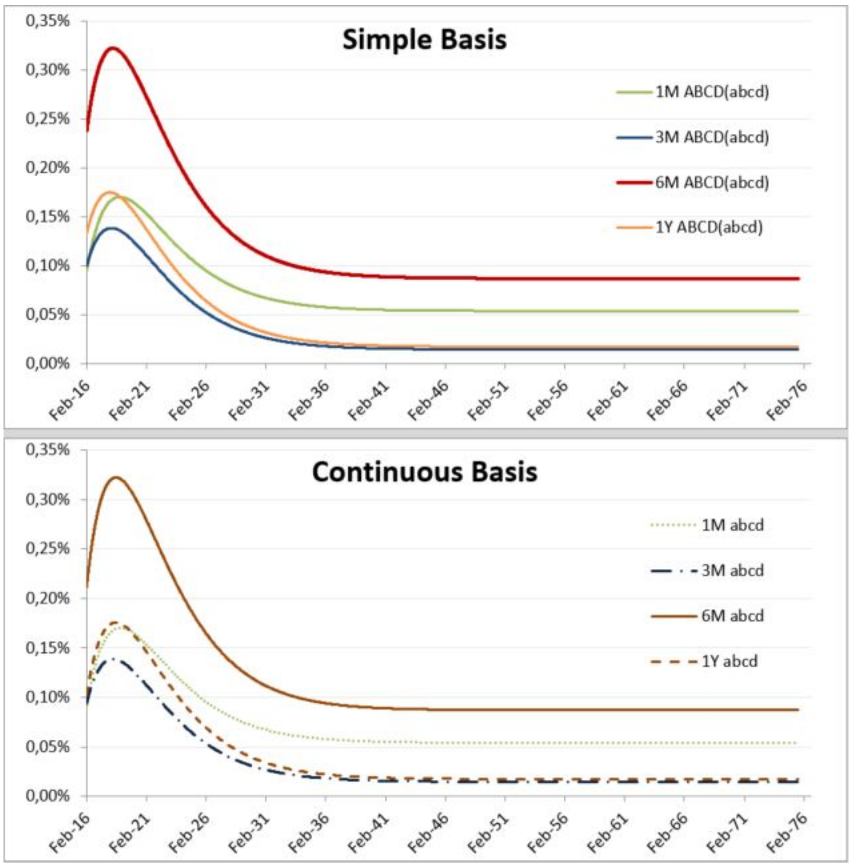


Figure 7.8: Relative basis from incremental calibration

Both the approach bring with them excellent results, because show that a relative abcd is not losing fitting power and is qualifying the entire scheme.

7.2 Basis shifting

Certified that also with a fixed value of c the model works, in the following is tested the capabilitie of repricing market quote exploiting basis retrieved thanks to (4).

7.2.1 Not incremental approach

After globally calibrated the model, where relative basis for 3M, 12M and 1M on ON have been built according to the abcd framework as (3), the relative basis can be retrieved thanks to (4). In particular have been retrieved:

$$s_{x,6M} = s_y - s_{6M} \tag{7.2}$$

where: $x \in (3M, 12M, 1M)$. The parameters of $s_{x,6M}$ have been retrieved exploiting (2).

If $x > 6M$:

$$\begin{aligned} a_{x,6M} &= a_x - a_{6M}; \\ b_{x,6M} &= b_x - b_{6M}; \\ d_{x,6M} &= d_x - d_{6M}; \end{aligned}$$

Otherwise:

$$\begin{aligned} a_{x,6M} &= a_{6M} - a_x; \\ b_{x,6M} &= b_{6M} - b_x; \\ d_{x,6M} &= d_{6M} - d_x; \end{aligned}$$

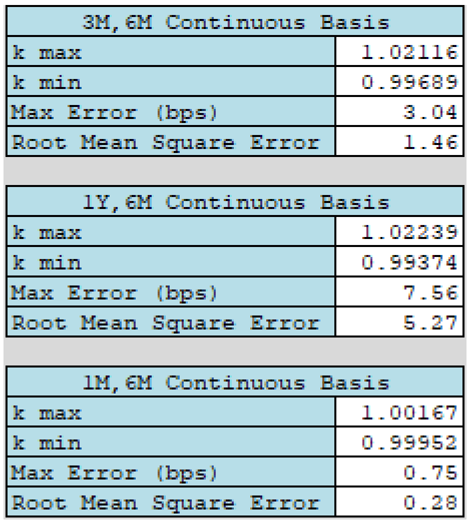
Given the basis, for each one has been made the repricing of the market quotes, modelling the considered instantaneous forward rate as:

$$f_x = f_{ON} + s_{6M} \pm s_{x,6M} \tag{7.3}$$

where the sign of the relative basis depends on whether or not the considered x tenor is greater or smaller than 6M.

Moreover, according to the incremental principle, the 1M it is also repriced as a basis on 3M.

The results for the basis with respect to 6M are:



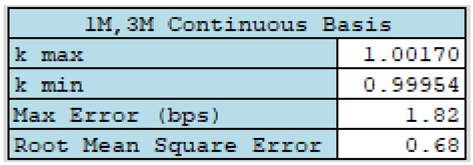
3M, 6M Continuous Basis	
k max	1.02116
k min	0.99689
Max Error (bps)	3.04
Root Mean Square Error	1.46

1Y, 6M Continuous Basis	
k max	1.02239
k min	0.99374
Max Error (bps)	7.56
Root Mean Square Error	5.27

1M, 6M Continuous Basis	
k max	1.00167
k min	0.99952
Max Error (bps)	0.75
Root Mean Square Error	0.28

Figure 7.9: Correction factors k from not incremental calibration with global c and relative basis

while for 1M from $s_{1M,3M}$ are:

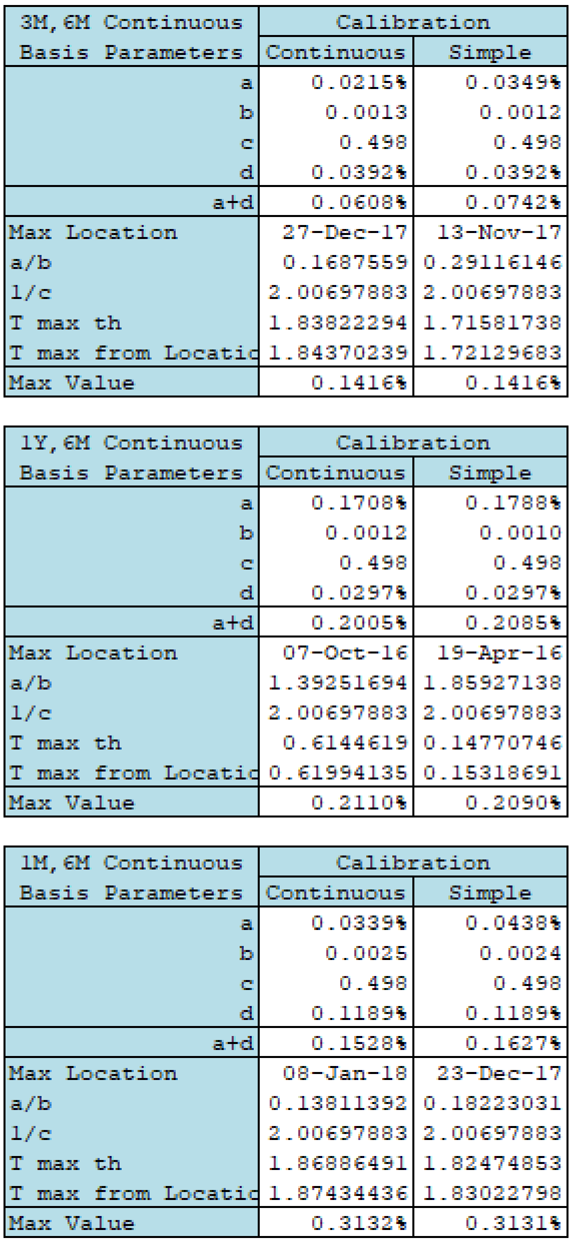


1M, 3M Continuous Basis	
k max	1.00170
k min	0.99954
Max Error (bps)	1.82
Root Mean Square Error	0.68

Figure 7.10: 1M correction factors k from not incremental calibration with global c and relative basis $s_{1M,3M}$

The k are generally closed to 1, but the root mean square error and the maximum error in basis points for 12M are high. Therefore, it seems that the k factor range is not a good metric for understanding how much the repricing is good, then a better metric is needed to summarize and interpret the correction factors.

Looking at the parameters and statistics:



3M,6M Continuous Basis Parameters	Calibration Continuous	Calibration Simple
a	0.0215%	0.0349%
b	0.0013	0.0012
c	0.498	0.498
d	0.0392%	0.0392%
a+d	0.0608%	0.0742%
Max Location	27-Dec-17	13-Nov-17
a/b	0.1687559	0.29116146
1/c	2.00697883	2.00697883
T max th	1.83822294	1.71581738
T max from Locatic	1.84370239	1.72129683
Max Value	0.1416%	0.1416%

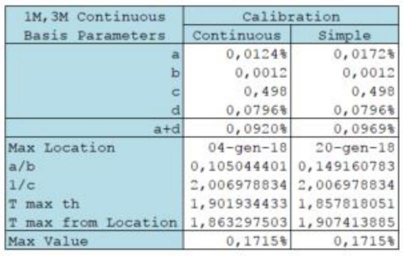
1Y,6M Continuous Basis Parameters	Calibration Continuous	Calibration Simple
a	0.1708%	0.1788%
b	0.0012	0.0010
c	0.498	0.498
d	0.0297%	0.0297%
a+d	0.2005%	0.2085%
Max Location	07-Oct-16	19-Apr-16
a/b	1.39251694	1.85927138
1/c	2.00697883	2.00697883
T max th	0.6144619	0.14770746
T max from Locatic	0.61994135	0.15318691
Max Value	0.2110%	0.2090%

1M,6M Continuous Basis Parameters	Calibration Continuous	Calibration Simple
a	0.0339%	0.0438%
b	0.0025	0.0024
c	0.498	0.498
d	0.1189%	0.1189%
a+d	0.1528%	0.1627%
Max Location	08-Jan-18	23-Dec-17
a/b	0.13811392	0.18223031
1/c	2.00697883	2.00697883
T max th	1.86886491	1.82474853
T max from Locatic	1.87434436	1.83022798
Max Value	0.3132%	0.3131%

Figure 7.11: Parameters from not incremental calibration with global c and relative basis

the parameters make sense, the time of maximum are closed, the only problem appears for the 12M that seems to act in a different way w.r.t. the other (it is really likely that there is an implementation error that the author can't currently see).

Moreover for $s_{1M,3M}$:



1M,3M Continuous Basis Parameters	Calibration	
	Continuous	Simple
a	0,0124%	0,0172%
b	0,0012	0,0012
c	0,498	0,498
d	0,0796%	0,0796%
a+d	0,0920%	0,0969%
Max Location	04-gen-18	20-gen-18
a/b	0,105044401	0,149160783
1/c	2,006978834	2,006978834
T max th	1,901934433	1,857818051
T max from Location	1,863297503	1,907413885
Max Value	0,1715%	0,1715%

Figure 7.12: Parameters from not incremental calibration with global c and relative basis $s_{1M,3M}$

Also looking at graphical comparison (excluding $s_{1M,3M}$):

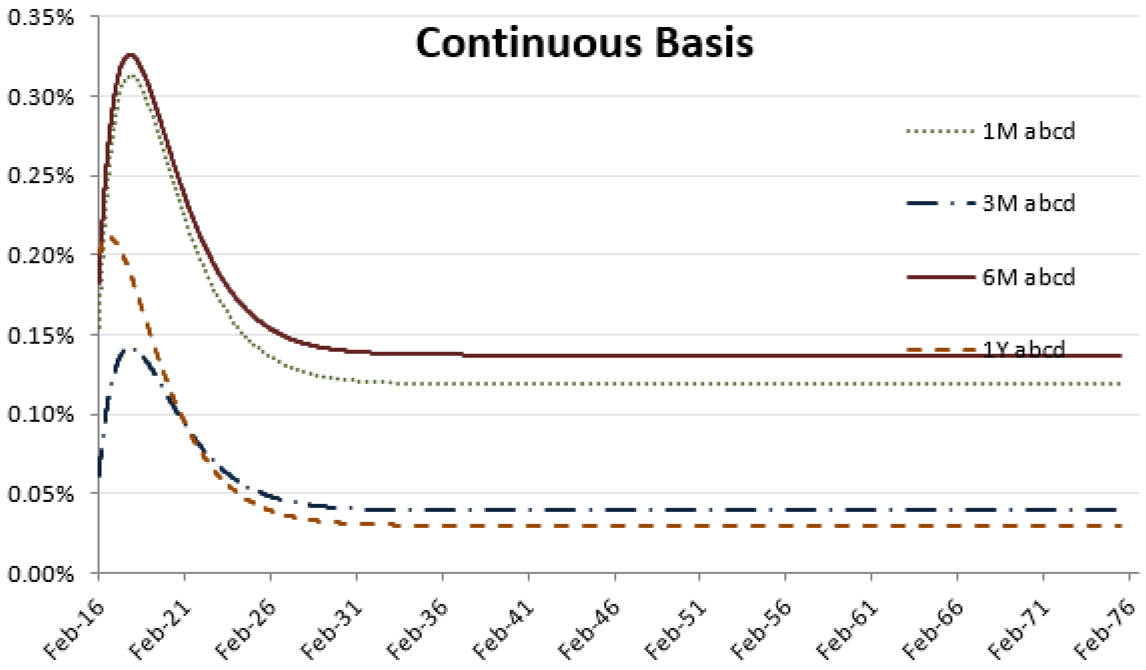


Figure 7.13: Absolute s_{6M} and its respective $s_{x,6M}$ relative basis

the basis dominance is respected and it makes sense that the $s_{1M,6M}$ is next to the s_{6M}.

7.2.2 Incremental approach

After globally calibrated the model, where relative basis for 3M, 12M and 1M have been built according to the abcd incremental framework as (4), the absolute basis can be retrieved. In particular have been retrieved:

$$s_x = s_{x,6M} + s_{6M} \tag{7.4}$$

where: $x \in (3M, 12M)$.

While for 1M because of incremental principle:

$$s_x = s_{x,3M} + s_{3M} \tag{7.5}$$

Where: $s_{3M} = s_{6M} - s_{3M,6M}$

The parameters of s_x have been retrieved exploiting (2). If $x > y$:

$$\begin{aligned} a_x &= a_{x,y} + a_y; \\ b_x &= b_{x,y} + b_y; \\ d_x &= d_{x,y} + d_y; \end{aligned}$$

Otherwise:

$$\begin{aligned} a_x &= a_y - a_{x,y}; \\ b_x &= b_y - b_{x,y}; \\ d_x &= d_y - d_{x,y}; \end{aligned}$$

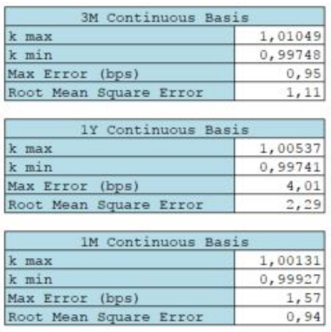
Where: $y \in (6M, 3M)$ in this specific case.

Given the basis, for each one has been made the repricing of the market quotes, modelling the considered instantaneous forward rate as:

$$f_x = f_{ON} + s_y \pm s_{x,y} \tag{7.6}$$

where the sign of the relative basis depends on whether or not the considered x tenor is greater or smaller than y.

The results are:



3M Continuous Basis	
k max	1,01049
k min	0,99748
Max Error (bps)	0,95
Root Mean Square Error	1,11

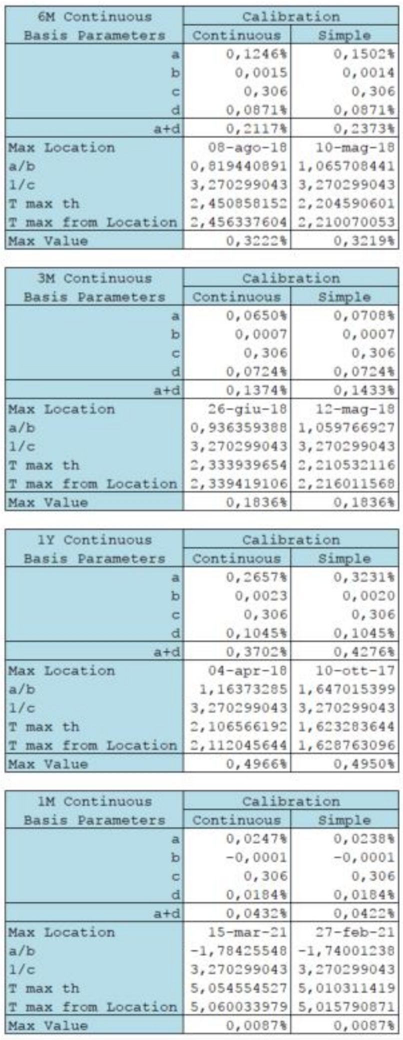
1Y Continuous Basis	
k max	1,00537
k min	0,99741
Max Error (bps)	4,01
Root Mean Square Error	2,29

1M Continuous Basis	
k max	1,00131
k min	0,99927
Max Error (bps)	1,57
Root Mean Square Error	0,94

Figure 7.14: Correction factors k from incremental calibration with global c and absolute basis

The k are generally closed to 1, but the root mean square error and the maximum error in basis points for 12M are still high. Therefore, it seems that the k factor range is not a good metric for understanding how much the repricing is good, then a better metric is needed to summarize and interpret the correction factors.

Looking at the parameters and statistics:



6M Continuous Basis Parameters	Calibration	
	Continuous	Simple
a	0,1246%	0,1502%
b	0,0015	0,0014
c	0,306	0,306
d	0,0871%	0,0871%
a+d	0,2117%	0,2373%
Max Location	08-ago-18	10-mag-18
a/b	0,819440891	1,065708441
1/c	3,270299043	3,270299043
T max th	2,450858152	2,204590601
T max from Location	2,456337604	2,210070053
Max Value	0,3222%	0,3219%

3M Continuous Basis Parameters	Calibration	
	Continuous	Simple
a	0,0650%	0,0708%
b	0,0007	0,0007
c	0,306	0,306
d	0,0724%	0,0724%
a+d	0,1374%	0,1433%
Max Location	26-giu-18	12-mag-18
a/b	0,936359388	1,059766927
1/c	3,270299043	3,270299043
T max th	2,333939654	2,210532116
T max from Location	2,339419106	2,216011568
Max Value	0,1836%	0,1836%

1Y Continuous Basis Parameters	Calibration	
	Continuous	Simple
a	0,2657%	0,3231%
b	0,0023	0,0020
c	0,306	0,306
d	0,1045%	0,1045%
a+d	0,3702%	0,4276%
Max Location	04-apr-18	10-ott-17
a/b	1,16373285	1,647015399
1/c	3,270299043	3,270299043
T max th	2,106566192	1,623283644
T max from Location	2,112045644	1,628763096
Max Value	0,4966%	0,4950%

1M Continuous Basis Parameters	Calibration	
	Continuous	Simple
a	0,0247%	0,0238%
b	-0,0001	-0,0001
c	0,306	0,306
d	0,0184%	0,0184%
a+d	0,0432%	0,0422%
Max Location	15-mar-21	27-feb-21
a/b	-1,78425548	-1,74001238
1/c	3,270299043	3,270299043
T max th	5,054554527	5,010311419
T max from Location	5,060033979	5,015790871
Max Value	0,0087%	0,0087%

Figure 7.15: Parameters from incremental calibration with global c and absolute basis

the time of maximum are not closed and the problem with 1M still appears.

Also looking at graphical comparison:

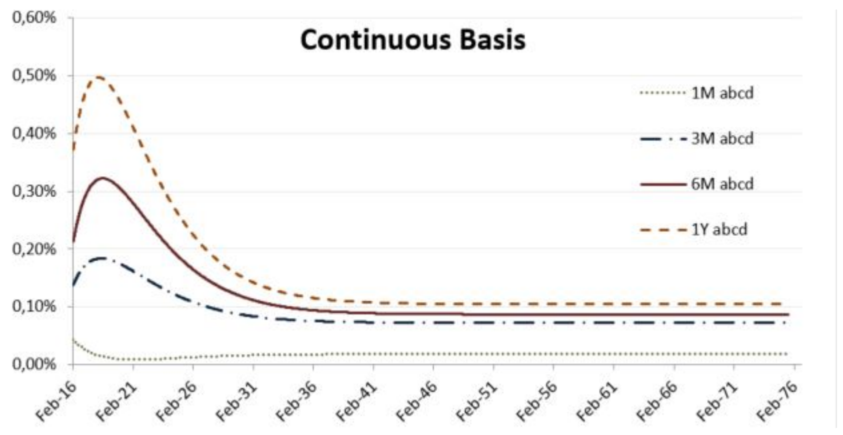


Figure 7.16: Absolute basis from relative ones

the basis dominance is respected and it is coherent that problem with 1M still remains for the absolute basis with both the approach.

What has been discovered in this chapter qualifies the model and allows to close the second point of the research schedule. Moreover,until now the previous framework has been exploited as much as possible, but given the goodness of the outputs it makes sense to implement in source code a specific solution in order to give a tool to perform this calibration without recurring to the Solver help.

Chapter 8

Fixing of time: an empirical analysis

Before to implement the approach, the parametric form of b, for a generic tenor x, needs to be retrieved:

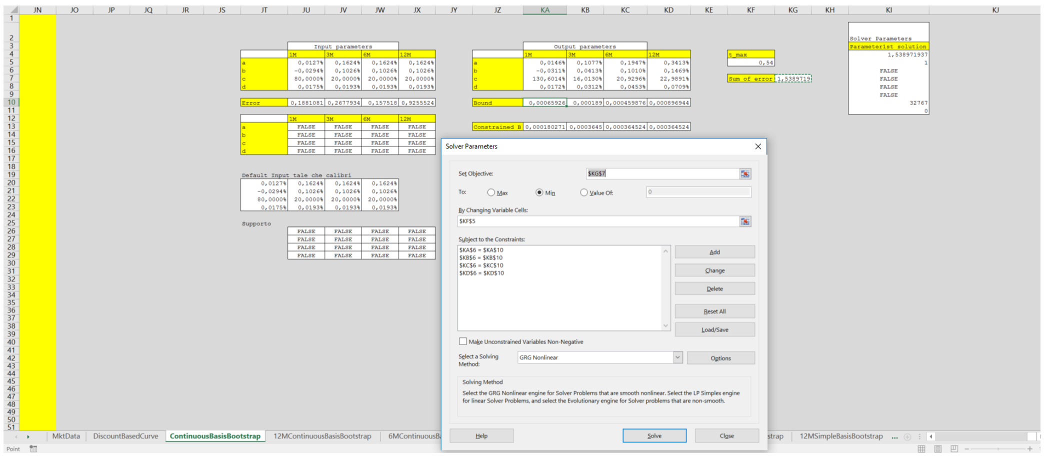
$$b_x = \frac{a_x c_x}{1 - t_{max} c_x} \tag{8.1}$$

As seen in the chapter chapter 7, the first idea is to exploit excel capabilities in order to perform our analysis. The first idea is to choose t_{max} through the Solver, while leaving the choice of a, c and d to the Calibrator and posing a condition on the outputs. The algorithm works as follows:

1. the Solver minimize (7.1) with respect to the time;
2. guesses enters the calibrator allowing it to change them (therefore the respective indicator has been set as "FALSE" in the framework);
3. multiple conditions on the Solver's output parameter have been posed such that $\hat{b}_x$, for each $x \in (1M, 3M, 6M, 12M)$ it's equal to:

$$\hat{b}_x = \frac{\hat{a}_x \hat{c}_x}{1 - t_{max} \hat{c}_x} \tag{8.2}$$

The solver is set as follows:



Input parameters

	1M	3M	6M	12M
a	0,0127%	0,1624%	0,1624%	0,1624%
b	-0,0294%	0,1026%	0,1026%	0,1026%
c	80,0000%	20,0000%	20,0000%	20,0000%
d	0,0175%	0,0193%	0,0193%	0,0193%
Error	0,1881081	0,2677934	0,157518	0,9255524

	1M	3M	6M	12M
a	FALSE	FALSE	FALSE	FALSE
b	FALSE	FALSE	FALSE	FALSE
c	FALSE	FALSE	FALSE	FALSE
d	FALSE	FALSE	FALSE	FALSE

Default Input tale che calibri

0,0127%	0,1624%	0,1624%	0,1624%
-0,0294%	0,1026%	0,1026%	0,1026%
80,0000%	20,0000%	20,0000%	20,0000%
0,0175%	0,0193%	0,0193%	0,0193%

Supporto

FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	FALSE	FALSE

Output parameters

	1M	3M	6M	12M
a	0,0146%	0,1077%	0,1947%	0,3413%
b	-0,0311%	0,0413%	0,1010%	0,1469%
c	130,6014%	16,0130%	20,9296%	22,9891%
d	0,0172%	0,0312%	0,0453%	0,0709%
Bound	0,00065926	0,000189	0,000459876	0,000896944
Constrained B	0,000180271	0,0003645	0,000364524	0,000364524

t_max	
0,54	
Sum of error	1,5389719

Solver Parameters

Parameter1st solution
1,538971937
1
FALSE
FALSE
FALSE
FALSE
32767
0

Solver Parameters

Set Objective: KG7

To: ○ Max [● Min] ○ Value Of: 0

By Changing Variable Cells: KF5

Subject to the Constraints:

KA6 = KA10
KB6 = KB10
KC6 = KC10
KD6 = KD10

☐ Make Unconstrained Variables Non-Negative

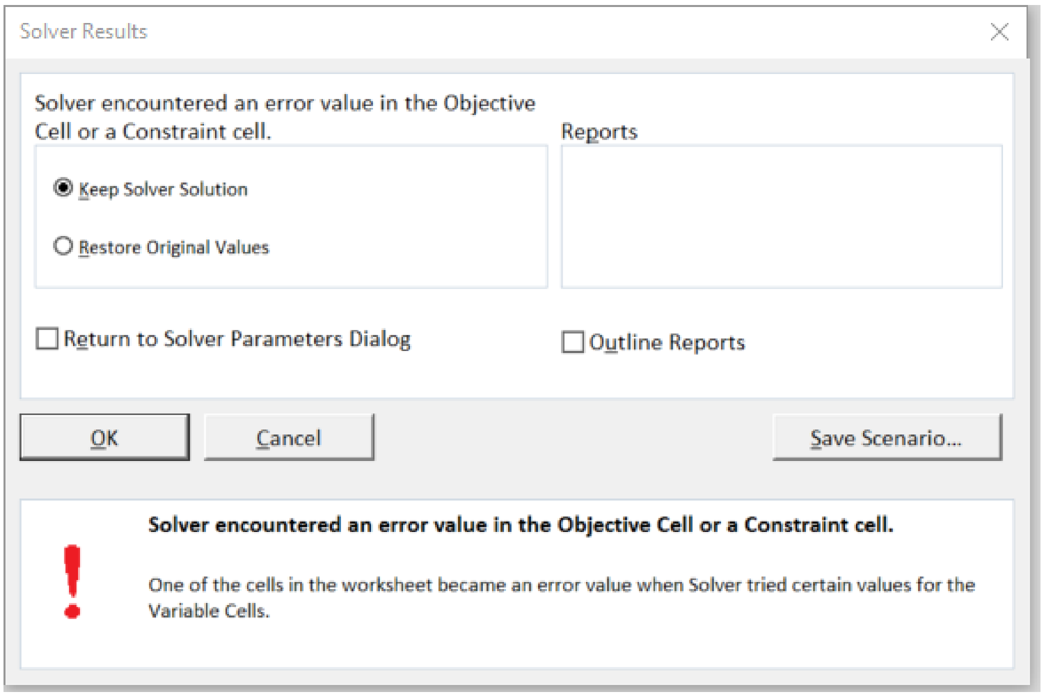
Select a Solving Method: GRG Nonlinear

Solving Method

Select the GRG Nonlinear engine for Solver Problems that are smooth nonlinear. Select the LP Simplex engine for linear Solver Problems, and select the Evolutionary engine for Solver problems that are non-smooth.

Figure 8.1: Fixed T- First Problem

and the output is:



Solver Results

Solver encountered an error value in the Objective Cell or a Constraint cell.

[● Keep Solver Solution]

○ Restore Original Values

☐ Return to Solver Parameters Dialog

Reports

☐ Outline Reports

Solver encountered an error value in the Objective Cell or a Constraint cell.

One of the cells in the worksheet became an error value when Solver tried certain values for the Variable Cells.

Figure 8.2: Fixed T-First Output

These results come from a not incremental approach, but are valid for both, and the problems are commonly shared too:

- given that the calibrator ignores the bounds that the solver has to respect, each time that the Solver refuses the calibrator's solution, the calibrator comes once again to the same solution and they conflict.
- the algorithm doesn't converge to a solution, therefore the input points and the algorithm itself need to be changed, the matter is how to deal with this changes;

Given these results, the nested calibration can't be put on practice through the solver, therefore the decided strategy aims to modify the source code.

Considering the will to built a global calibrator such that is possible also to calibrate with respect to c, in order to guarantee the future extensibility of the framework, the best solution it is to create a framework which allows to fix or not the parameters internally an globally fixing a variable, that , can change also dynamically in the framework for example with t_{max} that is a function of the other parameters concurring in specifying the model.

Bibliography

[1] Luigi Ballabio. *Implementing QuantLib.* Leanpub, 2017-08-21.

[2] Paolo Mazzocchi Ferdinando M. Ametrano, Luigi Ballabio. The abcd of interest rate basis spread. *SSRN*, July 2016.

[3] D.Brigo F.Mercudio. *Interest Rate Models - Theory and Practice.* Springer Finance, 2006.

[4] Martin Fowler. *UML Distilled. A Brief Guide to the Standard Object Modeling Language.* Pearson Education (US), 2003.