Toward Adjoint-Based Optimization in Computational Finance

Niloofar Safiran [corresponding author]
safiran@stce.rwth-aachen.de
Uwe Naumann
naumann@stce.rwth-aachen.de

LuFG Informatik 12: Software and Tools for Computational Engineering,
RWTH Aachen University, Germany.

1 Introduction

The LIBOR market model (LMM) belongs to the family of interest rate market models. LIBOR stands for the London Interbank Offered Rate and is the interest rate at which large banks can borrow funds from one another. These rates are published daily by the British Banker's Association and so are directly observable.

The LIBOR Market Model is an approach that uses forward LIBOR rates as a foundation. In a LIBOR Market Model, the pay-off of an interest rate derivative is expressed in terms of a finite number of forward LIBOR rates. These rates are then assumed to be log-normally distributed under an appropriate forward measure. Since each forward LIBOR rate can introduce another stochastic component to the model, the dimensionality of the LMM can be very high - for instance, pricing a ten year interest rate swap could require modeling ten forward LIBOR rates. Currently, pricing is achieved in practice by using simulation based 'Monte Carlo' methods.

In computational finance, Monte Carlo simulation is used to compute the correct prices for financial options. More important, however, is the ability to compute the so-called "Greeks", the first and second order derivatives of the prices with respect to input parameters such as the current asset price, interest rate and level of volatility.

Automatic Differentiation (AD), sometimes alternatively called algorithmic differentiation, is a method for computing derivatives of output of numerical programs with respect to its inputs both accurately (with machine precision) and efficiently. The two basic modes of AD – forward and reverse – and combinations thereof yield products of a vector with Jacobian, its transposed, or Hessian respectively.

Numerical simulation programs map potentially very large number of input parameters (let there be n) onto often much fewer outputs (say m of them, also refer to as objectives). The classical numerical approach to quantifying the sensitivities of those objectives with respect to the parameters through Finite Difference quotients increase the computational complexity by a factor of $O(n)$. The number of parameters may reach values of the order of $n=10^9$. Hence forward sensitivity analysis would require n runs of the simulation, which is simply not feasible.

Reverse (or Adjoint) methods and corresponding program transformation techniques have been developed to replace the dependence on n with that on the number of objectives m. Often the number of objectives is equal to one. In this case adjoint programs deliver the sensitivities of the objective with respect to all input parameters with an increase in the computational complexity of $O(1)$.

Adjoint codes can be generated from a given numerical simulation program by semantic program transformation technique. By applying Derivative Code Compiler (DCC) these transformations would be generated automatically.

The only problem with using adjoint is: Restoring the intermediate values computed by such a program in reverse order for a given upper bound on the available memory.

The goal of this abstract is to analyze the sensitivity of one of the financial market models, namely “LIBOR Market Model”. For this purpose, first of all we should simulate this model. The next step would be calculating the sensitivities. This paper discusses the three main approaches to computing Greeks: finite difference, Path-wise Forward and Path-wise adjoint calculation. The last of these has an adjoint implementation with a computational cost which is independent of the number of inputs. We will introduce a tool, namely “Derivative Code Compiler” (DCC), which is very useful and accurate in calculating the derivatives. DCC uses the principals of Automatic Differentiation (AD), which is described in section 3. What we should do is, to write a program which is adaptable by DCC, then apply it. It will automatically calculate the derivatives. The next step, which we are interested in, is to do some optimization using the first and second order derivatives, which are calculated by DCC and compare them with the results we get by applying Finite Difference (i.e., without using DCC).

2 LIBOR Market Model

The forward rates have a correlated log-normal evolution:

$$\frac{dL_i}{L_i}=\mu_i(t)dt+\sigma_i(t)dW_i(t)$$
$$\Rightarrow\ L_i(t+\Delta t)=L_i(t)\exp\left((\mu_i(t)-\frac{1}{2}\bar{\sigma}_i^2(t,t+\Delta t))\Delta t+\bar{\sigma}_i(t,t+\Delta t)\Delta W_i(t)\right)$$

where W_i is Brownian motion and $\sigma_i(t)$ is the volatility. We further assume that the Brownian motion W_i and W_j of different rates $L_i(t)$ and $L_j(t)$ are instantaneously correlated according to ρ_{ij} , $i,j=1,\dots,n$ i.e., $dW_i(t)dW_j(t)=\rho_{ij}dt$.

We fix the rolled over one period bond as numeraire.

$$\mu_i=\sum_{j=\gamma(t)}^{i}\frac{\delta_j L_j(t)}{1+\delta_j L_j(t)}\rho_{ij}(t)\sigma_i(t)\sigma_j(t),\quad \delta_i=T_{i+1}-T_i,\quad \gamma(t)=i\ \ if\ \ T_{i-1}\le t<T_i$$

The volatility and correlation are deterministic functions of time.

The functional form of the volatility:

$$\sigma_i(t)=(a+b(T_i-t))\exp(-c(T_i-t))+d \qquad \bar{\sigma}_i(t,t+\Delta t)=\sqrt{\frac{1}{\Delta t}\int_t^{t+\Delta t}\sigma_i^2(\tau)d\tau}$$

The correlation is exponentially decaying. Its functional form is:

$$\rho_{ij}=\exp(-\beta.|T_i-T_j|)$$

We are given a tenor structure T_i , $i=\alpha+1,\dots,\beta$, and a swap rate *K* at time *t*. The payer swaption is an option to enter into a swap at time T_α with swap rate *K*. It thus has time-t discounted pay-off:

$$P(T_\alpha;t)N.max((\sum_{i=\alpha+1}^{\beta}P(T_\alpha,T_i;t)\delta_i(L_i(T_\alpha)-K)),0)$$

where P(T;t) is the Zero-coupon bond.

3 Derivative Code Compiler

Manual implementation of the forward/reverse mode algorithms is possible but tedious. Fortunately, automated tools have been developed. One approach is Derivative Code Compiler (DCC), which uses the principals of Algorithmic Differentiation (AD) in order to calculate forward and backward derivatives.

Forward/Adjoint codes can be generated from a given numerical simulation program by semantic program transformation technique. By applying Derivative Code Compiler (DCC) these transformations would be generated automatically.

Programmer supplies code which takes u as input and produces $v = f(u)$ as output.

- In forward mode, AD tool generates new code which takes u and $\dot{u}$ as input, and produces v and $\dot{v}$ as output:

$$\dot{v} = \left(\frac{\partial f}{\partial u}\right)\dot{u}$$

- In reverse mode, AD tool generates new code which takes u and $\bar{v}$ as input, and produces v and $\bar{u}$ as output:

$$\bar{u} = \left(\frac{\partial f}{\partial u}\right)^T \bar{v}$$

The first step after installing dcc, is to define the function(s) in one file, e.g. *f.c* . For example in our LIBOR model, we defined 3 files, i.e., path_calc.c (to calculate the forward rates), Portfolio.c (to calculate the portfolio value) and libor_head.c (this function first apply path_calc function and then Portfolio function). We should gather all of them in *f.c*.

To apply dcc a configuration file is needed. It contains the information about the mode, which should be used.

Tangent-linear (or forward) code of *f.c* can be generated by dcc by running *dcc f.c < d1.conf* command. The first command line parameter is the name of the file containing the code to be transformed. The meaning and the content of the configuration of *d1.conf* will be explained later. Running this command, the file *d1_f.c* will be generated in the same folder.

To generate the Adjoint code of the first order the command *dcc f.c < b1.conf* should be used. By applying *b1.conf* to *f.c* in dcc, the file *b1_f.c* will be generated in the same folder.

To generate the second order derivative one has to apply dcc to the generated code *d1_f.c* or *b1_f.c.* After generating the first derivatives there are 4 different possibilities of generating the second order derivative: Forward over Forward (d2d1), Forward over Backward (d2b1), Backward over Forward (b2d1) and Backward over Backward (b2b1).

f.c < d1.conf ------> (generates) d1_f.c

------> d1_f.c < b2d1.conf ------> b2d1_f.c

------> d1_f.c < d2d1.conf ------> d2d1_f.c

f.c < b1.conf ------> (generates) b1_f.c

------> b1_f.c < d2b1.conf ------> d2b1_f.c

------> b1_f.c < b2b1.conf ------> b2b1_f.c

The contents of the configuration files are:

Name of the file	Content of the file
d1.conf	1 1 v
b1.conf	2 1 v 100000 cs_c fds 100000 fds_c ids 100000 ids_c
d2d1.conf	1 2 v
b2d1.conf	2 2 v 100000 cs_c fds 100000 fds_c ids 100000 ids_c
b2b1.conf	2 1 v 100000 cs_c2 fds2 100000 fds_c2 ids2 100000 ids_c2
d2b1.conf	1 2 v

Listing 1: Overview of used configuration files

The first entry denotes the mode (1: tangent-linear, 2: adjoint) and the second entry the order of the differentiation. Names of auxiliary variables are derived from the third entry by concatenation with automatically generated suffixes. The size of control stack whose name is derived from cs is set to 100000. A counter starting with the string cs_c is used to address the control stack. Similar information is provided for the floating-point and integer data stack.

The dependent and independent program variables need to be specified by the user via special pragmas. e.g. for our LIBOR model:

//$ad indep L
//$ad dep portfolio_val

After defining dependencies and generating d1_f.c (or b1_f.c), we should include this "d1_f.c" (or b1_f.c) to the driver. The driver is the code, in which we define our main function and initialize the variables, in order to get the forward or backward derivative.

3 Implementation

The function "libor_head" computes the forward rates with "path_calc" and apply those rates to calculate the discounted payoff with "Portfolio". This function is in file "f.c" So, in the file of main function, we simply include "f.c" and initialize all of the values and call "libor_head" which has the following signature:

```
void libor_head(int& N, int& N_mat, double& delta, int& N_opt, int* maturity, double*
                swaprates, int& curr_time, double* L, double* Z, double& portfolio_val,
                double* sigma, double& a, double& b, double& c, double& d, double&
                beta, double* rho, double* maturity_i)
```

N: The difference between the current time and the last maturity (N_mat).
N_mat: The last maturity, i.e., the greatest* β .
delta: The time step, δ , which we simulate the LIBOR forward rates for that interval.
N_opt: The number of options.
maturity: Each option has its own maturity (end life).
Swaprates: Each option has its own swap rate (i.e., fixed rate) K.
curr_time: Our current time, here $t=0$.
L: The LIBOR rates at t=0.
Z: $N(0,1)$ random variable.
portfolio_val: The discounted payoff V.

* Note that here we have N_opt options in which the maturity for all of them may not be the same.

sigma: The volatility σ .
a,b,c,d: values to calibrate σ .
beta: value to calibrate the correlation ρ .
maturity_i: The distance from current time until N per step. In other words:
maturity_i[i] = curr_time + i * δ , for i=0, 1, . . . , N-1.

3.1 Calculating Sensitivities

In the previous section, we simulated our LIBOR market model. Here we will show the procedure of calculating sensitivities with different methods.

All of the codes for this section can be found on DVD on folder “Simulate_Compare”.

3.1.1 First Order (Deriving Delta : $\partial V/\partial L(0)$)

A comparison of the duration of calculation with different methods is given in Figure 1.

3.1.1.1 Finite Difference

For deriving the Delta with one sided finite difference we use the formula:

$$\frac{\partial V}{\partial L}=\frac{V(L+h)-V(L)}{h}$$

which h is a very small perturbation. In our case, we set h=1e-8.

Procedure: Calculate the discounted payoff for the given initial LIBOR rate (e.g. L[i] = 0.05, i=0:N-1), so we will get V(L), then define a loop over j (j=0:N-1) and at each iteration set the forward rates to their initial rate (0.05) and just perturb one L with h (L[j] = 0.05 + h) and compute the discounted payoff for the perturbed L, i.e., V(L+h). Then use above formula to get Delta for each L.

The formula for calculating Delta with two sided finite difference is:

$$\partial V/\partial L=\frac{V(L+h)-V(L-h)}{2h}$$

Procedure: Define a loop over j (j=0:N-1) and at each iteration set the forward rates to their initial rate (L[i]=0.05, i=0:N-1) and just perturb one L with h (L[j] = 0.05 + h) and compute the discounted payoff for the perturbed L, i.e., V(L+h). Again, set the forward rates to their initial rate (L[i]=0.05, i=0:N-1) and just perturb one L with h (L[j] = 0.05 - h) and compute the discounted payoff for the perturbed L, i.e., V(L-h)Then use above formula to get Delta for each L.

3.1.1.2 Forward Method with DCC

The aim of this section is to calculate the derivatives with forward (or tangent linear) method using Derivative Code Compiler.

All of the functions are collected in file “f.c”. Note that there exist some limitations for the file which dcc should be applied on (in our case: “f.c”). For example, the functions should not return anything. They all should have *void* type, furthermore, all of the values should be initialized before defining the methods in the functions, etc. For more reading, please refer to [2].

At first, we define a file “d1.conf” (Listing 1).
Then on the shell console we write:

```
dcc  f.c < d1.conf
```

Applying d1.conf to f.c will generate file “d1_f.c”.

In this file all of the functions of “f.c” will be written, but with prefix “d1” , and also for all variables with *double* type, an auxiliary variables with the same type and dimension is defined.

Procedure: In the file that we defined the main function, we should include “f.c” and “d1_f.c”.
For calculating the derivative of payoff with respect to each LIBOR forward rate, in main function for each of the values which has type *double* (e.g. double a) define another variable which has also type *double* as an auxiliary variable (e.g. d1_a) with the same dimension and set them to zero. Define a loop over j (j=0:N-1) and at each iteration set all variables and also the forward rates to their initial value (L[i]=0.05, i=0:N-1) and all the auxiliary variables to zero except for d1_L[j] (the auxiliary variable for L which has N dimension like L). Set d1_L[j]=1 and apply “d1_libor_head” which has signature:

```
void d1_libor_head(int& N, int& N_mat, double& delta, double& d1_delta, int& N_opt, int*
                   maturity, double* swaprates, double* d1_swaprates, int& curr_time,
                   double* L, double* d1_L, double* Z, double* d1_Z, double&
                   portfolio_val, double& d1_portfolio_val, double* sigma, double*
                   d1_sigma, double& a, double& d1_a, double& b, double& d1_b,
                   double& c, double& d1_c, double& d, double& d1_d, double& beta,
                   double& d1_beta, double* rho, double* d1_rho, double* maturity_i,
                   double* d1_maturity_i)
```

At each iteration the derivative with respect to L[j] is stored in d1_portfolio_val. At each iteration, output this value. Then compile the file which contains the main function and see the results.

3.1.1.3 Adjoint Method with DCC

The aim of this section is to calculate the derivatives with Adjoint (or Reverse) method using Derivative Code Compiler.

At first, we define a file “b1.conf” (Listing 1).

Note that for using Adjoint method, it is very important to define checkpoints which store the intermediate values.

Then on the shell console we write:

```
dcc  f.c < b1.conf
```

Applying b1.conf to f.c will generate file “b1_f.c”.

In this file all of the functions of “f.c” will be written, but with prefix “b1” , and also for all variables with *double* type, an auxiliary variables with the same type and dimension is defined.
Additionally it has one extra *integer* type variable “bmode” in the signature of all functions defined in “f.c”, which in the main function should be set to 1.

Procedure: In the file that we defined the main function, we should include “b1_f.c”.
For calculating the derivative of payoff with respect to each LIBOR forward rate, in main function for each of the

values which has type *double* (e.g. double a) define another variable which has also type *double* as an auxiliary variable (e.g. b1_a) with the same dimension and set them to zero. Set b1_portfolio_val=1 and bmode=1. Then apply "b1_libor_head" which has signature:

```
void b1_libor_head(int& bmode, int& N, int& N_mat, double& delta, double& b1_delta,
                   int& N_opt, int* maturity, double* swaprates, double* b1_swaprates,
                   int& curr_time, double* L, double* b1_L, double* Z, double* b1_Z,
                   double& portfolio_val, double& b1_portfolio_val, double* sigma,
                   double* b1_sigma, double& a, double& b1_a, double& b, double&
                   b1_b, double& c, double& b1_c, double& d, double& b1_d, double&
                   beta, double& b1_beta, double* rho, double* b1_rho, double*
                   maturity_i, double* b1_maturity_i)
```

The derivative with respect to L is stored in b1_L. For i=0:N-1 , output this b1_L[i]. Then compile the file which contains the main function and see the results.

3.1.2 Second Order (Calculating Gamma : $\partial^2 V / \partial L^2$)

A comparison of the duration of calculation with different methods is given in Figure 2.

3.1.2.1 Finite Difference

For deriving the Gamma with finite difference we use the formula:

$$\frac{\partial^2 V}{\partial L_i^2} = \frac{-V(L+h_i e_i)+16V(L+h_i e_i)-30V(L)+16V(L-h_i e_i)-V(L-2h_i e_i)}{12h_i^2}$$

$$\frac{\partial^2 V}{\partial L_i \partial L_j} = \frac{V(L+h_i e_i+h_j e_j)-V(L+h_i e_i-h_j e_j)-V(L-h_i e_i+h_j e_j)+V(L-h_i e_i-h_j e_j)}{4h_i h_j}$$

In this case h1=h2=1e-6.

procedure: In the main function, calculate the discounted payoff for the given initial rates (L[i]=0.05, i=0:N-1) get V(L). Define a loop over i1 (i1=0:N). Set the LIBOR rates to their initial values (L[j]=0.05 , j=0:N-1) but perturb L[i1] with L[i1]+h1 and compute V for these forward rates, you will get V(L+h1). Do the same for L[i1]-h1 and get V(L-h1). Define another loop (in the i1 loop) over i2 (i2=0:N-1). Set the LIBOR rates to their initial values (L[j]=0.05 , j=0:N-1) but perturb L[i2] with L[i2]+h2 and compute V for these forward rates, you will get V(L+h2).Set the LIBOR rates to their initial values (L[j]=0.05 , j=0:N-1) but perturb L[i2] with L[i2]-h2 and compute V for these forward rates, you will get V(L-h2). Set the LIBOR rates to their initial values (L[j]=0.05 , j=0:N-1) but perturb L[i2] with L[i2]+h2 and L[i1] with L[i1]+h1 compute V for these forward rates, you will get V(L+h1+h2).Set the LIBOR rates to their initial values (L[j]=0.05 , j=0:N-1) but perturb L[i2] with L[i2]-h2 and L[i1] with L[i1]+h1 compute V for these forward rates, you will get V(L+h1-h2).Set the LIBOR rates to their initial values (L[j]=0.05 , j=0:N-1) but perturb L[i2] with L[i2]+h2 and L[i1] with L[i1]-h1 compute V for these forward rates, you will get V(L-h1+h2).Set the LIBOR rates to their initial values (L[j]=0.05 , j=0:N-1) but perturb L[i2] with L[i2]-h2 and L[i1] with L[i1]-h1 compute V for these forward rates, you will get V(L-h1-h2). Put these values in the above formula, you will get Gamma.

3.1.2.2 Forward over Forward Method with DCC

The aim of this section is to calculate the second order derivatives with forward over forward method using Derivative Code Compiler.

At first, we define “d1.conf” and “d2d1.conf”(Listing 1).

Then on the shell console we write:

```
dcc f.c < d1.conf
```

Applying d1.conf to f.c will generate file “d1_f.c”.

In this file all of the functions of “f.c” will be written, but with prefix “d1” , and also for all variables with *double* type, an auxiliary variables with the same type and dimension is defined.

For generating the appropriate code for the second derivative with forward over forward, we should apply d2d1.conf to “d1_f.c”.

```
dcc d1_f.c < d2d1.conf
```

Applying d2d1.conf to d1_f.c will generate file “d2_d1_f.c”.

In this file all of the functions of “d1_f.c” will be written, but with prefix “d2” , and also for all variables with *double* type, two auxiliary variables with the same type and dimension is defined.

Procedure: In the file that we defined the main function, we should include “d2_d1_f.c”.
For calculating the second derivative of payoff with respect to each LIBOR forward rate, in main function for each of the values which has type *double* (e.g. double a) define another three variable which has also type *double* as an auxiliary variable (e.g. d1_a, d2_a, d2_d1_a) with the same dimension and set them to zero. Define a loop over i1 (i1=0:N-1) and another loop over i2 (i2=0:N-1)(in loop i1). At each iteration set all variables and also the forward rates to their initial value (L[j]=0.05, j=0:N-1) and all the auxiliary variables to zero except for d1_L[i1] and d2_L[i2] (the auxiliary variables for L which has N dimension like L). Set d1_L[i1]=1 and d2_L[i2]=1. Then apply “d1_d2_libor_head” which has signature:

```
void d2_d1_libor_head(int& N, int& N_mat, double& delta, double& d2_delta, double&
                      d1_delta, double& d2_d1_delta, int& N_opt, int* maturity, double*
                      swaprates, double* d2_swaprates, double* d1_swaprates, double*
                      d2_d1_swaprates, int& curr_time, double* L, double* d2_L, double*
                      d1_L, double* d2_d1_L, double* Z, double* d2_Z, double* d1_Z,
                      double* d2_d1_Z, double& portfolio_val, double& d2_portfolio_val,
                      double& d1_portfolio_val, double& d2_d1_portfolio_val, double*
                      sigma, double* d2_sigma, double* d1_sigma, double* d2_d1_sigma,
                      double& a, double& d2_a, double& d1_a, double& d2_d1_a,
                      double& b, double& d2_b, double& d1_b, double& d2_d1_b,
                      double& c, double& d2_c, double& d1_c, double& d2_d1_c,
                      double& d, double& d2_d, double& d1_d, double& d2_d1_d,
                      double& beta, double& d2_beta, double& d1_beta, double&
                      d2_d1_beta, double* rho, double* d2_rho, double* d1_rho, double*
                      d2_d1_rho, double* maturity_i, double* d2_maturity_i, double*
                      d1_maturity_i, double* d2_d1_maturity_i)
```

The second derivative with respect to L is stored in d2_d1_portfolio_value. At each iteration over i2 output this d2_d1_portfolio_value. Then compile the file which contains the main function and see the results.

3.1.2.3 Forward over Adjoint Method with DCC

The aim of this section is to calculate the second order derivatives with forward over Adjoint method using

Derivative Code Compiler.

At first, we define “b1.conf” and “d2b1.conf” (Listing 1).
Then on the shell console we write:

```
dcc f.c < b1.conf
```

Applying b1.conf to f.c will generate file “b1_f.c”.

In this file all of the functions of “f.c” will be written, but with prefix “b1” , and also for all variables with double type, an auxiliary variables with the same type and dimension is defined. Additionally it has one extra integer type variable “bmode” in the signature of all functions defined in “f.c”, which in the main function should be set to 1.

For generating the appropriate code for the second derivative with forward over forward, we should apply d2b1.conf to “b1_f.c”.

```
cp b1_f.c b1_f_tmp.c
cpp -I. -P -C b1_f_tmp.c >b1_f.c
rm b1_f_tmp.c
dcc b1_f.c < d2b1.conf
```

Applying d2b1.conf to b1_f.c will generate file “d2_b1_f.c”.

In this file all of the functions of “b1_f.c” will be written, but with prefix “d2” , and also for all variables with *double* type, two auxiliary variables with the same type and dimension is defined.

Procedure: In the file that we defined the main function, we should include “d2_b1_f.c”.
For calculating the second derivative of payoff with respect to each LIBOR forward rate, in main function for each of the values which has type *double* (e.g. double a) define another three variable which has also type *double* as an auxiliary variable (e.g. b1_a, d2_a, d2_b1_a) with the same dimension and set them to zero. Define a loop over i (i=0:N-1). Set bmode=1. At each iteration set all variables and also the forward rates to their initial value (L[j]=0.05, j=0:N-1) and all the auxiliary variables to zero except for b1_portfolio_value and d2_L[i] (the auxiliary variables for L which has N dimension like L). Set b1_portfolio_value=1 and d2_L[i]=1. Then apply “d1_b2_libor_head” which has signature:

```
void d2_b1_libor_head(int& bmode_1, int& N, int& N_mat, double& delta, double&
                      d2_delta, double& b1_delta, double& d2_b1_delta, int& N_opt, int*
                      maturity, double* swaprates, double* d2_swaprates, double*
                      b1_swaprates, double* d2_b1_swaprates, int& curr_time, double* L,
                      double* d2_L, double* b1_L, double* d2_b1_L, double* Z, double*
                      d2_Z, double* b1_Z, double* d2_b1_Z, double& portfolio_val,
                      double& d2_portfolio_val, double& b1_portfolio_val, double&
                      d2_b1_portfolio_val, double* sigma, double* d2_sigma, double*
                      b1_sigma, double* d2_b1_sigma, double& a, double& d2_a, double&
                      b1_a, double& d2_b1_a, double& b, double& d2_b, double& b1_b,
                      double& d2_b1_b, double& c, double& d2_c, double& b1_c,
                      double& d2_b1_c, double& d, double& d2_d, double& b1_d,
                      double& d2_b1_d, double& beta, double& d2_beta, double&
                      b1_beta, double& d2_b1_beta, double* rho, double* d2_rho, double*
                      b1_rho, double* d2_b1_rho, double* maturity_i, double*
                      d2_maturity_i, double* b1_maturity_i, double* d2_b1_maturity_i)
```

The second derivative with respect to L is stored in d2_b1_L. At each iteration over i output this d2_d1_L[j], j=0:N-1. Then compile the file which contains the main function and see the results.

3.2 Optimization of the Forward Rates

Suppose that we have now the real values of the discounted payoff at each time until maturity, i.e., v[j] and j=0:Nmat-1, which we simulated before. We name these real payoffs "v_tilda". We want to find the best match. Set $J = \sum_{i=0}^{Nmat-1} (v[i] - v_{tilda}[i])^2$ and try to find the minimum of J with respect to the forward rates $L_i(0), i=0:N$.

3.2.1 Optimization with NAG

nag_opt_bounds_deriv (e04kbc) is a comprehensive quasi-Newton algorithm for finding:

– an unconstrained minimum of a function of several variables.
– a minimum of a function of several variables subject to fixed upper and/or lower bounds on the variables.
(Here we do not have any bound for our forward rates, therefore we set bound=Nag_Nobounds.)

First derivatives are required. nag_opt_bounds_deriv (e04kbc) is intended for objective functions which have continuous first and second derivatives (although it will usually work even if the derivatives have occasional discontinuities).

Specification:

```
#include <nag.h>
#include <nage04.h>
void nag_opt_bounds_deriv (Integer n,
                           void (*objfun)(Integer N, const double L[], double *J, double g[], Nag_Comm *comm),
                           Nag_BoundType bound, double bl[], double bu[], double L[], double *J, double g[],
                           Nag_E04_Opt *options, Nag_Comm *comm, NagError *fail)
```

For optimizing our model with this method, we need the first derivative. For this purpose we compute the derivatives with Forward and adjoint mode of AD and then will compare them.

Again functions path_calc and Portfolio are defined in f.c. The procedure to construct d1_f.c and b1_f.c have been mentioned before.

3.2.1.1. With Forward:

Procedure: Write the main function like the standard form of NAG for optimization with Quasi-Newton, i.e., nag_opt_bounds_deriv (e04kbc). In the file that the main function is defined, include "d1_f.c". Function objfun is supposed to take the initial forward rates and calculate the portfolio values and also the first derivatives. For calculating the derivative of payoff with respect to each LIBOR forward rate, in objfun function for each of the values which has type *double* (e.g. double a) define another variable which has also type *double* as an auxiliary variable (e.g. d1_a) with the same dimension and set them to zero. Define a loop over j (j=0:N-1) and at each iteration set all variables and also the forward rates to their initial value (e.g. L[i]=0.05, i=0:N-1) and all the auxiliary variables to zero except for d1_L[j] (the auxiliary variable for L which has N dimension like L). Set d1_L[j]=1 and apply "d1_libor_head" in objfun. Note that "d1_libor_head" has the same signature as described in 3.1.1.2.

At each iteration the derivative with respect to L[j] is stored in d1_portfolio_val. Therefore at each iteration set g[j]=d1_portfolio_val. Then compile the file which contains the main function and see the results.

3.2.1.2 With Adjoint:

*Procedure:*Write the main function like the standard form of NAG for optimization with Quasi-Newton, i.e., nag_opt_bounds_deriv (e04kbc). In the file that we defined the main function, we should include "b1_f.c". Function objfun is supposed to take the initial forward rates and calculate the portfolio values and also the first derivatives. For calculating the derivative of payoff with respect to each LIBOR forward rate, in objfun function for each of the values which has type *double* (e.g. double a) define another variable which has also type *double* as an auxiliary variable (e.g. b1_a) with the same dimension and set them to zero. Set b1_portfolio_val=1 and bmode=1. Then apply "b1_libor_head".Note that "b1_libor_head" has the same signature as described in 3.1.1.3.

The derivative with respect to L is stored in b1_L. For i=0:N-1 , output this g[i]=b1_L[i]. Then compile the file which contains the main function and see the results.

Note: For integer data type NAG uses "Integer", however DCC uses "int". We will face problem if we pass an argument with data type "Integer" to DCC. What we can do is: e.g. n should be set:

```
static void NAG_CALL objfun(Integer n, double l[], double* objf, double gc[], Nag_Comm *comm)
{
  …
  int N = n;
  …
   d1_libor_head(int& N, … )   // or b1_libor_head(int& bmode, int& N, … )
}
```

4 Experiments and Results

We Simulate LIBOR Market Model for *L(0)=0.05* and *Nmat=40* time-steps with a vector of *N=Nmat+40* forward rates for *Nopt=15* options, and computes the *N* deltas and *N*N* Gammas for a portfolio of swaptions with different methods.

$$V = P(t;T_\alpha)\sum_{n=1}^{Nopt}\{100.\left(\sum_{i=\alpha+1}^{maturity[n]} P(T_\alpha,T_i)\delta_i(L_i(T_\alpha)-K_n)\right)_+\}$$

$$L_i(t+\delta_i) = L_i(t)\exp\left((\mu_i(t)-\frac{1}{2}\bar{\sigma}_i^2(t,t+\delta_i))\delta_i+\bar{\sigma}_i(t,t+\delta_i)\Delta W_i(t)\right)$$

$$\mu_i = \sum_{j=\gamma(t)}^{i}\frac{\delta_j L_j(t)}{1+\delta_j L_j(t)}\rho_{ij}(t)\sigma_i(t)\sigma_j(t),\quad for\ i=1,\ldots,N,\quad \delta_i=T_{i+1}-T_i,\quad \gamma(t)=i\ \ if\ \ T_{i-1}\le t<T_i$$

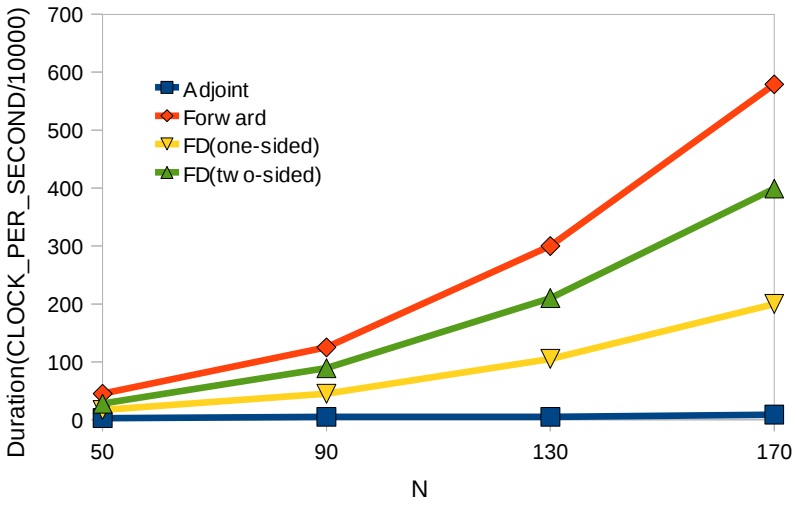

Figure 1: Duration of calculating Delta $\partial V/\partial L(0)$

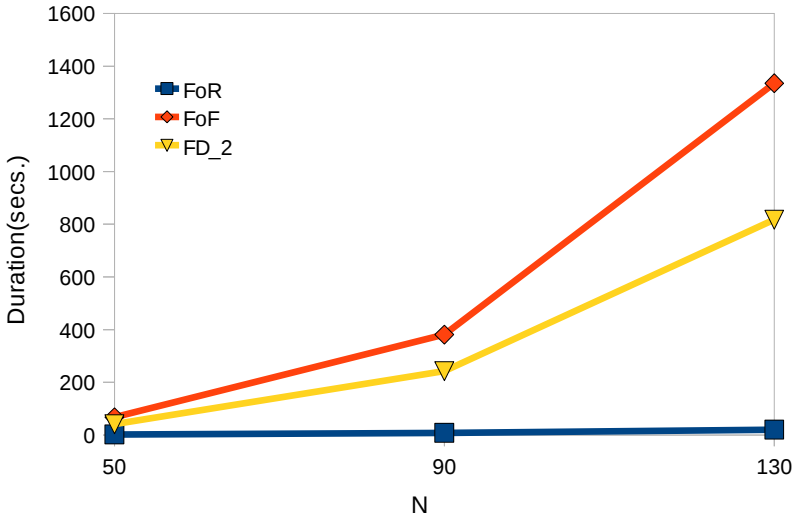

Figure 2: Duration of calculating Gamma $\partial^2 V/\partial L^2$

As it is shown, FoR is the fastest. It is 42.5 times faster than FoF and 27.5 times faster than FD_2.

In both calculations (Delta and Gamma) the results of forward and adjoint are the same, but they have a small difference with the results of Finite Difference.
The Norm of Difference between the Forward method and Finite Difference for different step-sizes is:

Step-Size (h)	**||Forward – Finite_Difference||**	**||FoF – Finite_Difference_2||**
h= 0.1	Difference= 65.0024	Difference= 0.961973
h= 0.01	Deference= 6.80839	Difference= 0.00961083
h= 0.001	Difference= 0.684175	Difference= 0.000210797
h= 0.0001	Difference= 0.0686059	Difference= 0.000206181

h= 1e-05	Difference= 0.00721036	Difference= 0.00172978
h= 1e-06	Difference= 0.00194603	Difference= 0.106025
h= 1e-07	Difference= 0.00177178	Difference= 15.9903
h= 1e-08	Difference= 0.00177574	Difference= 1116.5

Table 1: The importance of choosing appropriate step-size in Finite Difference method.

4.1 Optimizing LIBOR Model

Let *v_tilda* be the real values. We set $J(L)=\sum_{i=0}^{Nmat-1}(\text{v\_tilda[i]} - \text{V(L)[i]})^2$. Depending on the method of optimization, derivatives will be needed. In this section, we calculate the derivatives for optimization with different methods and compare the results.

- Optimizing with Steepest Descent: $L_{n+1}=L_n-\alpha_n\nabla J(L_n),\quad n\geq 0$ for $\alpha>0$ a small enough number, we have:

$$J(L_0)\geq J(L_1)\geq J(L_2)\geq\dots$$

We get optimal α_n with line search.

Optimization with Steepest Descent				
	Finite Difference h=1e-8	**Finite Difference h=1e-6**	**Forward**	**Adjoint**
Duration (sec.)	85	45	576	65
Precision	0.00147028	0.157431	9.43328e-11	9.53151e-11
Last J	4.4078e-10	4.27724e-06	2.50867e-24	2.7347e-24
Number of iteration	35183	18708	94942	94753

Table 3: Compare different methods for Optimizing with Steepest Descent.

The results of Adjoint and Forward are nearly the same, but they have different duration. This time Adjoint was 8.86 times faster than forward.

- Optimizing with Newton: $L_{n+1}=L_n-\alpha_n[H J(L_n)]^{-1}\nabla J(L_n),\quad n\geq 0$

Solving this part with LU Decomposition.

Optimization with Newton			
	Second Derivative Finite Difference	**Forward over Forward (FoF)**	**Forward over Adjoint (FoR)**
Duration (sec.)	22	183	5
Precision	0.0365909	2.1539e-11	2.33617e-11
Last J	2.15361e-09	1.3592e-27	1.69795e-27
Number of iteration	22	7	7

Table 3: Compare different methods for Optimizing with Newton.

The results of FoR and FoF are the same, but they have different duration. This time FoR was 36.6 times faster than FoF.

- Optimizing with Quasi-Newton using NAG: $L_{n+1} = L_n - \alpha_n B_n \nabla J(L_n), \quad n \geq 0$

  Where B_n is an approximation of the Hessian matrix.

Optimization with NAG			
	Finite Difference	**Forward**	**Adjoint**
Number of iteration	126	47	47
Last J	8.6167e-11	2.8413e-17	2.8413e-17
Norm g	3.6e-03	1.5e-06	1.5e-06
Norm L	4.5e-01	4.5e-01	4.5e-01
Norm(L(k-1)-L(k))	4.8e-08	1.5e-06	1.5e-06
Real time	0m19.490s	0m7.680s	0m0.287s
User time	0m19.379s	0m7.011s	0m0.235s
System time	0m0.085s	0m0.431s	0m0.027s

Table 4: Compare different methods for Optimizing with Quasi-Newton with NAG.

The results of Adjoint and Forward are nearly the same, but they have different duration. Here, Adjoint was 26.76 times faster than forward.

5 Conclusion

For all of those experiments, the time spent by the adjoint code was minimum. The only problem with using adjoint is: Restoring the intermediate values.

Care should be taken when choosing step-size h as an inappropriate h may effect the derivatives considerably.

We also optimized our LMM with 3 methods, Steepest Descent, Newton and Quasi-Newton (with NAG), and calculated the first and the second order of derivatives with different methods. We set our precision to 1e-10 for Steepest Descent and Newton, but for both optimizations, Finite Difference did not converge to that precision. Also in Table 4, the precision that we achieved by applying Forward and Adjoint to calculate the derivatives in NAG is much higher than the one that we got by using Finite Difference.

We have seen that: The adjoint method is advantageous for calculating the sensitivities of a small number of securities with respect to a large number of parameters. The forward method is advantageous for calculating the sensitivities of many securities with respect to a small number of parameters.

References

[1] *Exact First- and Second-Order Greeks by Algorithmic Differentiation.* NAG Technical Report, TR5/10, The Numerical Algorithm Group ltd. 2010.

[2] U. Naumann: *The Art of Differentiating Computer Programs, Introduction.* SIAM, 2011, to appear.

[3] Gill P. E. and Murray W. : *Quasi-Newton methods for unconstrained optimization.* J. Inst. Math. Appl. 9 91–108, 1972.

[4] Gill P. E. and Murray W. : *Safeguarded steplength algorithms for optimization using descent methods.* NPL Report NAC 37, National Physical Laboratory, 1973 .

[5] Gill P. E. and Murray W. : *Minimization subject to bounds on the variables.* NPL Report, NAC 72 , National Physical Laboratory , 1976.

[6] Gill P. E., Murray W. and Pitfield R. A. : *The implementation of two revised quasi-Newton algorithms for unconstrained optimization.* NPL Report NAC 11, National Physical Laboratory, 1972 .

[7] J. Nocedal and S. J. Wright: *Numerical Optimization.* Second Edition, Springer Series in Operations Research, 2006.

[8] M. Giles and P. Glasserman : *Smoking Adjoints: fast evaluation of Greeks in Monte Carlo calculations.* RISK, January 2006.

[9] M. Giles: *Monte Carlo evaluation of sensitivities in computational finance* Report NA-07/12, Oxford University Computing Laboratory, 2007.

[10] P. Glassermann and X. Zhao : *Arbitrage-free discretization of the lognormal forward libor and swap rate models,* in: Finance and stochastics, Vol. 4, No. 1, 2000.

[11] P. Glassermann: *Monte Carlo Methods in Financial Engineering,* Springer Verlag, 2003.

[12] A. Griewank and A. Walther: *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation.* Second Edition, SIAM, 2008.