SABR calibration in Python

Giovanni Travaglini

January 31, 2016

Abstract

This work is to implement in Python the SABR volatility model for swaptions and its calibration. The attached script takes market data of forward rates and swaption volatilities, then calibrates the parameters in order to get the implied volatilities.

Formulas are given by Hagan et al. (2002), it is provided also the *"**shifted** model"* to overtake negative forward rates' problem.

You can download the script here (SABR.py), and the market data example here (MarketData.xlsx).

This paper consists in the explanation of the commands to have reason of computations and final results.

Keywords: SABR, Python, calibration, shift, implied volatility, swaptions

Contents

1 Brief introduction

1.1 Swaptions

A swaption is an option to enter into an interest rate swap. The time distance between "today" and the option's exercise date is called expiry, while the time distance between the exercise date and the swap's maturity is called tenor.

1.2 SABR model

The SABR (Stochastic Alpha Beta Rho) model (*Hagan et al. 2002*) is one of the simplest possible generalizations of the single-FRA rate Black model with stochastic volatility.

To compute the SABR implicit volatilities we need the following data (for a given tenor):

- Forward rate
- Expiry of the option
- Option strike (for example ATM + 50bps)
- Values for the parameters α, β, ρ, ν

1.3 Calibration of parameters

To calibrate the model with respect to market data the following function is minimized:

$$\sqrt{\sum_{i=-x}^{x} \left(\sigma_{SABR,i} - \sigma_{MKT,i} \right)^2}$$

$$s.t.: \ \alpha > 0 \ , \ 0 \leq \beta \leq 1 \ , \ -1 \leq \rho \leq 1 \ , \ \nu > 0$$

i is the strike index, so the sum is computed along the swaption's "smile".

1.4 Input file

Click here to download the market data.

The *input file* of market data has this structure:

			Market Volatilities for strike spreads in bps:								
Tenor	Expiry	Fwd	-150	-100	-50	-25	0	25	50	100	150
2	0,25	0,010763833	0,0000	1,0470	0,4812	0,4327	0,4268	0,4148	0,4253	0,4322	0,4495
2	0,5	0,011099189	0,0000	0,9647	0,5079	0,4637	0,4477	0,4390	0,4377	0,4452	0,4576
2	0,75	0,011602429	0,0000	0,8253	0,5033	0,4648	0,4494	0,4387	0,4348	0,4375	0,4463
2	1	0,012193564	0,0000	0,6796	0,4788	0,4474	0,4501	0,4435	0,4478	0,4611	0,4754
2	2	0,016195984	0,0000	0,9119	0,5417	0,4628	0,4529	0,4461	0,4386	0,4387	0,4442
2	5	0,028436364	0,4040	0,3541	0,3218	0,3107	0,3048	0,2975	0,2923	0,2873	0,2870
2	10	0,033873471	0,3026	0,2725	0,2510	0,2422	0,2343	0,2279	0,2228	0,2161	0,2128
5	0,25	0,01601749	1,1870	0,6027	0,4655	0,4278	0,4030	0,3879	0,3789	0,3710	0,3725
5	0,5	0,01680246	0,9568	0,5800	0,4661	0,4339	0,4125	0,3969	0,3888	0,3801	0,3785
5	0,75	0,017681947	0,8325	0,5562	0,4578	0,4288	0,4078	0,3914	0,3821	0,3719	0,3692
5	1	0,018623226	0,7242	0,5240	0,4446	0,4210	0,4042	0,3904	0,3807	0,3699	0,3668
5	2	0,022384186	0,5704	0,4686	0,4119	0,3925	0,3781	0,3656	0,3561	0,3438	0,3380
5	5	0,030538725	0,3720	0,3304	0,3016	0,2910	0,2816	0,2758	0,2700	0,2617	0,2572
5	10	0,033460207	0,3108	0,2749	0,2488	0,2387	0,2304	0,2237	0,2184	0,2115	0,2083

1.5 Output files

Click here to download the script.

The *output files* are the calibrated parameters, the SABR volatilities and its differences with respect to the market.

This is how the volatilities will be printed on screen:

SABR VOLATILITIES

strikes:		-150	-100	-50	-25	ATM	25	50	100	150
tenor	expiry									
2y	3m	2.4627	1.0223	0.5411	0.422	0.3898	0.398	0.4155	0.4507	0.4792
2y	6m	2.2568	0.9476	0.5512	0.4585	0.4246	0.4226	0.4317	0.4556	0.4768
2y	9m	1.8902	0.8166	0.5267	0.4637	0.4366	0.4295	0.4313	0.4428	0.4551
2y	1y	1.5033	0.678	0.4816	0.4503	0.4417	0.4434	0.4491	0.4628	0.475
2y	2y	1.8832	0.886	0.5779	0.4912	0.4404	0.4208	0.421	0.4421	0.4666
2y	5y	0.4042	0.3536	0.3222	0.3115	0.3033	0.2973	0.293	0.2881	0.2864
2y	10y	0.3031	0.2721	0.2501	0.2416	0.2345	0.2286	0.2237	0.2165	0.2119
5y	3m	1.1844	0.6101	0.4623	0.4259	0.4031	0.389	0.3806	0.3727	0.3702
5y	6m	0.9565	0.5819	0.4635	0.4327	0.4124	0.399	0.3902	0.3807	0.3767
5y	9m	0.8321	0.5583	0.4557	0.427	0.4072	0.3934	0.3838	0.3727	0.3674
5y	1y	0.7239	0.5253	0.4436	0.42	0.4031	0.391	0.3822	0.3712	0.3652
5y	2y	0.5702	0.4692	0.4119	0.3924	0.3773	0.3655	0.3565	0.3444	0.3376
5y	5y	0.3722	0.33	0.3015	0.291	0.2824	0.2754	0.2698	0.2619	0.2571
5y	10y	0.3112	0.2745	0.2485	0.2386	0.2305	0.2239	0.2187	0.2117	0.2079
10y	3m	0.6344	0.4886	0.4001	0.3698	0.3472	0.3313	0.321	0.3124	0.3127
10y	6m	0.6148	0.4856	0.4038	0.375	0.353	0.3373	0.3272	0.3193	0.3211
10y	9m	0.5858	0.4699	0.3955	0.3688	0.3481	0.3328	0.3222	0.3119	0.3109
10y	1y	0.5541	0.4536	0.3886	0.365	0.3461	0.3313	0.3201	0.3062	0.3003
10y	2y	0.4728	0.402	0.3551	0.3377	0.3234	0.3118	0.3025	0.2892	0.281
10y	5y	0.3601	0.3162	0.2858	0.2744	0.2651	0.2577	0.2518	0.2438	0.2396
10y	10y	0.3367	0.2953	0.2666	0.2558	0.247	0.2398	0.234	0.226	0.2214

2 Commented script

2.1 Modules

```
import xlrd
import math
import numpy
from scipy.optimize import minimize
```

"xlrd" is needed to read the input file, "math" to use logarithm and square root, "numpy" to use the numerical enviroment, "scipy.optimize" to minimize the objective function for the calibration.

2.2 Functions

For the explanation I skip the printing and writing commands in order to simplify the reading.

Each function can be called singularly, using the appropriate data (pay attention to variables' dimension).

Variables:

- alpha, beta, rho and nu are the parameters
- F is the forward rate
- K is the strike of the option
- time is the expiry of the option
- MKT is the market volatility
- i is the index for the tenor/expiry swaption
- j is the index for the moneyness of the option

2.2.1 SABR implied volatilities

```
def SABR(alpha,beta,rho,nu,F,K,time,MKT):
    if K <= 0:
        VOL = 0
        diff = 0
    elif F == K: # ATM formula
        V = (F*K)**((1-beta)/2.)
        logFK = math.log(F/K)
        A = 1 + ( ((1-beta)**2*alpha**2)/(24.*(V**2)) + (alpha*beta*nu*rho)/(4.*V) ...
            + ((nu**2)*(2-3*(rho**2))/24.) ) * time
        B = 1 + (1/24.)*(((1-beta)*logFK)**2) + (1/1920.)*(((1-beta)*logFK)**4)
        VOL = (alpha/V)*A
        diff = VOL - MKT
    elif F != K: # not-ATM formula
        V = (F*K)**((1-beta)/2.)
        logFK = math.log(F/K)
        z = (nu/alpha)*V*logFK
        x = math.log( ( math.sqrt(1-2*rho*z+z**2) + z - rho ) / (1-rho) )
        A = 1 + ( ((1-beta)**2*alpha**2)/(24.*(V**2)) + (alpha*beta*nu*rho)/(4.*V) ...
            + ((nu**2)*(2-3*(rho**2))/24.) ) * time
        B = 1 + (1/24.)*(((1-beta)*logFK)**2) + (1/1920.)*(((1-beta)*logFK)**4)
        VOL = (nu*logFK*A)/(x*B)
        diff = VOL - MKT
    if MKT==0:
        diff = 0
```

```
def smile(alpha,beta,rho,nu,F,K,time,MKT,i):
    for j in range(len(K)):
        if K[0] <= 0:
            shift(F,K)
        SABR(alpha,beta,rho,nu,F,K[j],time,MKT[j])


def SABR_vol_matrix(alpha,beta,rho,nu,F,K,time,MKT):
    for i in range(len(F)):
        smile(alpha[i],beta[i],rho[i],nu[i],F[i],K[i],time[i],MKT[i],i)
```

The *SABR* function computes the implied volatilities for a single swaption, so all variables are scalars. If the forward rate is negative the implied volatility cannot be computed, so I set the output values zero.

This is the core function of the script, all input data and iterations are modelled to fit this specification.

The *smile* function computes the implied volatilities for a given "smile" pointed out by the index i. F, time and the parameters are scalars; K and MKT are vectors.

The *matrix* function computes the implied volatilities for all combinations of swaptions. F, time and the parameters are vectors; K and MKT are matrices.

2.2.2 Shift

```
def shift(F,K):
    shift = 0.001 - K[0]
    for j in range(len(K)):
        K[j] = K[j] + shift
        F = F + shift
```

When the forward rate is close to zero, the strike can become negative. In this case the *shifted model* allows to calibrate the parameters and to compute the implied volatilities for the whole smile.

The size of the shift depends on the lowest strike, then its value is added to all the strikes.

2.2.3 Calibration

```
def objfunc(par,F,K,time,MKT):
    sum_sq_diff = 0
    if K[0]<=0:
        shift(F,K)
    for j in range(len(K)):
        if MKT[j] == 0:
            diff = 0
        elif F == K[j]:
            V = (F*K[j])**((1-par[1])/2.)
            logFK = math.log(F/K[j])
            A = 1 + ( ((1-par[1])**2*par[0]**2)/(24.*(V**2)) + ...
```

```
                + (par[0]*par[1]*par[3]*par[2])/(4.*V) + ...
                + ((par[3]**2)*(2-3*(par[2]**2))/24.) ) * time
            B = 1 + (1/24.)*(((1-par[1])*logFK)**2)+(1/1920.)*(((1-par[1])*logFK)**4)
            VOL = (par[0]/V)*A
            diff = VOL - MKT[j]
        elif F != K[j]:
            V = (F*K[j])**((1-par[1])/2.)
            logFK = math.log(F/K[j])
            z = (par[3]/par[0])*V*logFK
            x = math.log((math.sqrt(1-2*par[2]*z+z**2) + z - par[2])/(1-par[2]))
            A = 1 + (((1-par[1])**2*par[0]**2)/(24.*(V**2)) + ...
                + (par[0]*par[1]*par[3]*par[2])/(4.*V) + ...
                + ((par[3]**2)*(2-3*(par[2]**2))/24.) ) * time
            B = 1 + (1/24.)*(((1-par[1])*logFK)**2)+(1/1920.)*(((1-par[1])*logFK)**4)
            VOL = (par[3]*logFK*A)/(x*B)
            diff = VOL - MKT[j]
        sum_sq_diff = sum_sq_diff + diff**2
        obj = math.sqrt(sum_sq_diff)
    return obj


def calibration(starting_par,F,K,time,MKT):
    for i in range(len(F)):
        x0 = starting_par
        bnds = ( (0.001,None) , (0,1) , (-0.999,0.999) , (0.001,None)  )
        res = minimize(objfunc, x0 , (F[i],K[i],time[i],MKT[i]) , ...
                        bounds = bnds, method='SLSQP')
        alpha[i] = res.x[0]
        beta[i] = res.x[1]
        rho[i] = res.x[2]
        nu[i] = res.x[3]
```

Through the *calibration* we want to find the parameters (for each smile) that make model volatilities as close as possible to market volatilities.

The *objective function* computes, with arbitrary parameters, the volatilities for a single smile, then sums the squares of all market differences. Its square root is the final quantity that we want to minimize.

Notice that volatility differences for missing market data are set as zero, so that the minimization is not affected.

For a specification without the shift, negative strikes can be neglected setting the differences zero as before.

I used the method "SLSQP" for constrained minimization of multivariate scalar functions, where the objective function is minimized with starting values for the parameters. The bounds for the parameters are those specified before in the model.

Running the script, this warning message appears:

```
Warning (from warnings module):
  File "C:\Python27\lib\site-packages\scipy\optimize\slsqp.py", line 337
    bnderr = where(bnds[:, 0] > bnds[:, 1])[0]
RuntimeWarning: invalid value encountered in greater
```

It is simply due to the unlimited bounds for α and ν .

Lastly the starting guess is replaced by the calibrated parameters.

2.3 Data and variables

2.3.1 Output and input files

```
outvol = open('outvol.csv', 'w')
vol_diff = open('vol differences.csv', 'w')
parameters = open('parameters.csv', 'w')

while True:
    try:
        file_input = xlrd.open_workbook('market_data.xlsx')
    except:
        print 'Input file is not in the directory!'
    break
Market_data = file_input.sheet_by_name('Swaptions data')
```

We first create the output files, that will be overwritten every time you run the script.

"outvol.csv" will contain the volatilities that we are going to compute, "vol differences.csv" their differences with respect to the market, and "parameters.csv" the calibrated parameters.

All market data we need is in a single input file, obviously, it must be in the directory, otherwise the program can't continue.

2.3.2 Swaptions characteristics

```
strike_spreads=[]
j=0
while True:
    try:
        strike_spreads.append(int(Market_data.cell(1,3+j).value))
        j = j+1
    except:
        break
num_strikes = len(strike_spreads)

expiries=[]
i=0
while True:
        try:
            expiries.append(Market_data.cell(2+i,1).value)
            i = i + 1
        except:
            break
```

```
tenors=[]
i=0
while True:
    try:
        tenors.append(Market_data.cell(2+i,0).value)
        i = i + 1
    except:
        break

F = []
i=0
while True:
    try:
        F.append(Market_data.cell(2+i,2).value)
        i = i+1
    except:
        break

K = numpy.zeros((len(F),num_strikes))
for i in range(len(F)):
    for j in range(num_strikes):
        K[i][j] = F[i] + 0.0001*(strike_spreads[j])

MKT = numpy.zeros((len(F),num_strikes))
for i in range(len(F)):
    for j in range(num_strikes):
        MKT[i][j] = Market_data.cell(2+i,3+j).value


starting_guess = numpy.array([0.001,0.5,0,0.001])
alpha = len(F)*[starting_guess[0]]
beta = len(F)*[starting_guess[1]]
rho = len(F)*[starting_guess[2]]
nu = len(F)*[starting_guess[3]]
```

From the input file, we take the values of strike spreads, tenors and expiries, in order to set the swaptions characteristics.

The same for the forward rate, the strike grid and the market volatilities.

At the end, the parameters have to be initialized with the starting guess.

You can modify my example of market data as you want, indeed I put an odd last row to show how to customize the format.

If you don't have market data, remember to write "0" instead of leaving blank.

2.3.3 Labels

```
exp_dates = len(expiries)*[0]
for i in range(len(expiries)):
    if expiries[i] < 1:
        exp_dates[i] = str(int(round(12*expiries[i])))+'m'
```

```
    else:
        exp_dates[i] = str(int(round(expiries[i])))+'y'
        if expiries[i]-round(expiries[i]) > 0:
            exp_dates[i] = exp_dates[i]+str(int(round((12*(round(expiries[i],2) -...
            int(expiries[i])))))+'m'
        elif expiries[i]-round(expiries[i]) < 0:
            exp_dates[i] = str(int(round(tenors[i]))-1)+'y'
            exp_dates[i] = exp_dates[i]+str(int(round((12*(round(expiries[i],2) -...
            int(expiries[i])))))+'m'

ten_dates = len(tenors)*[0]
for i in range(len(tenors)):
    if tenors[i] < 1:
        ten_dates[i] = str(int(round(12*tenors[i])))+'m'
    else:
        ten_dates[i] = str(int(round(tenors[i])))+'y'
        if tenors[i]-round(tenors[i]) > 0:
            ten_dates[i] = ten_dates[i]+str(int(round((12*(round(tenors[i],2) - ...
            int(tenors[i])))))+'m'
        elif tenors[i]-round(tenors[i]) < 0:
            ten_dates[i] = str(int(round(tenors[i]))-1)+'y'
            ten_dates[i] = ten_dates[i]+str(int(round((12*(round(tenors[i],2) - ...
            int(tenors[i])))))+'m'

label_exp = exp_dates
label_ten = ten_dates
label_strikes = num_strikes*[0]
for i in range(num_strikes):
    if strike_spreads[i] == 0 :
        label_strikes[i] = 'ATM'
    else:
        label_strikes[i] = str(strike_spreads[i])
```

Theese commands set the labels to improve the reading of the output.

2.4 Function calls

```
calibration(starting_guess,F,K,expiries,MKT)

SABR_vol_matrix(alpha,beta,rho,nu,F,K,expiries,MKT)
```

After creating all the variables, we call the calibration function that calibrates the parameters.

Then we compute and print the SABR implied volatilities for all swaptions and strikes with the just calibrated parameters.

```
outvol.close()
vol_diff.close()
parameters.close()
```

In the end we need to close the output files.