CVA Calculation for an Interest Rate Swap

- Calculate the credit valuation adjustment to the price of an interest rate swap using the credit spreads for Counterparty B.
- Plot MtM values (a good plot will show results from many simulations)
- Produce smoothed Expected Exposure prodile using the mean of the exposure distribution - distribution of Forward LIBORs at each time $T_i + 1$.
- Produce Potential Future Exposure with the simulated L_6M taken from the 97.5^{th} percentile.

The details for the IRS are as follows:

Recovery Rate = 40%

Tenor = 5Y

Payments Frequency = 6M

MtM Position = Floating Leg - Fixed Leg

Counterparty A = Sovereign UK (http://data.cnbc.com/quotes/GBCD5 (http://data.cnbc.com/quotes/GBCD5)) -> 43.00

Counterparty B = Sovereign Germany (http://data.cnbc.com/quotes/DECD5 (http://data.cnbc.com/quotes/DECD5)) -> 20.245

Credit spread value as per CNBC = 22.755 basis points (0.2275%, 0.002275)

We will need to compute:

- Fwd LIBOR rates (via monte carlo)
- Discount Factors
- Exposure
- Expected Exposure

And once we have all of those parts we will be able to determine the CVA (Credit Valuation Adjustment) for the interest rate swap outlined above.

Forward LIBOR

To provide the L_6M structure, we will generate the Forward LIBOR using *One Factor Libor Market Model*, described in 'Advanced Quantative Finance', Alonso Peña.

I have a particular interest in parallel and GPU based computing so I took it as an opportunity to rewrite the reference material in Python to aid integration with the CUDA GPU library provided by nVidia (https://developer.nvidia.com/cuda-toolkit (https://developer.nvidia.com/cuda-toolkit)) and also the Intel MKL libraries for optimized Math functions on intel processors. These are conveniently provided by the default install of Anaconda (http://www.continuum.io (http://www.continuum.io)) and are utilitised under the thirty day free trial.

In [2]:
```
import numpy as np
import accelerate as acc
from app.gpuCheck import getGpuCount

from accelerate.cuda.rand import PRNG, QRNG

gpuEnabled = False # gpu acceleration is not available in jupyter..
debug = False # code here is pretty much commented so set to false here..

# print 'Checking for compatible GPU ...'
# if getGpuCount() < 1:
#     print 'no compatible GPU installed, will revert to using numpy libraries fo
r RNG'
# else:
#     print 'nVidia GPU(s) found: ', getGpuCount()
#     print 'Enabling GPU for RNG'
#     gpuEnabled = True
```

The notebook itself runs the normal versions of this, if you are interested in running the GPU enabled versions. You will need an NVidia GFX card with compute capability of greater than 2.0 and have installed the latest cuda drivers. Once installed you can run the python files from disk. Cuda code is included but commented out in the notebook, it can cause kernel panics and is somewhat unstable when running in an iPython environment.

Calculate Credit Default Swap Spreads

In order to infer the correct lambda to use, we need to calculate the spread value for the CDS.

So for the value of the default leg, we will use:

$$PV(default) = \sum_{i=1}^{T_i} N.(1-R).DF_i.PD(T_i,T_{i-1})$$

$$PV(premium) = \sum_{i=1}^{T_i} \pi.N.\Delta t.DF_i.PD(T_i)$$

Bootstrapping the hazard rates using the formula below:

$$\lambda_k = \frac{-1}{\Delta t} ln(\frac{P(T_{k-1}D(0,T_k)(1-R)+\sum n=1^{k-1}}{})$$

Discount Factors

The following formula has been used to derive the discount factors:

$$DF_i = exp(-S_iT_i)$$

Default Probabilities

The following formula has been used to derive the default probablities:

$$PD_i = exp(-\lambda T_{i-1}) - exp(-\lambda T_i) \quad \forall i = 1,2,3,4$$

Forward Rates

next we need the forward rates and the discount factors.

$$L_I = S_I$$

$$L_1 = \frac{S_iT_i - S_{i-1}T_{i-1}}{T_i - T_{i-1}}$$

Forward rates can be derived from the spot rates in continous time.

To encapsulate this I have created the CDS class below:

In [1]:
```
from numpy import array, ones, zeros, linspace, exp, put, sum, log

"""
above are the imports form the NumPy packages www.numpy.org
they come as part of the Anaconda distribution from continuum.io
"""

class CreditDefaultSwap(object):
    def __init__(self, N=float, timesteps=int, discountFactors=list, lamda=float,
 seed=float):
        self.N = N  # notional
        self.__lamda = lamda
        self.__seed = seed
        self.recoveryRate = 0.4
        self.Dt = 0.25
        self.timesteps = linspace(0, 1, timesteps + 1)
        self.discountFactors = discountFactors
        self.pT = self.generateProbSurvival()
        self.premiumLegsSpread = self.calculatePremiumLegSpreads()
        self.__premiumLegSum = sum(self.premiumLegsSpread)
        self.pD = self.generateProbDefault()
        self.defaultLegsSpread = self.calculateDefaultLegSpreads()
        self.__defaultLegSum = sum(self.defaultLegsSpread)
        self.fairSpread = self.premiumLegsSpread / self.defaultLegsSpread

    @property
    def premiumLegSum(self):
        return self.__premiumLegSum

    @property
    def defaultLegSum(self):
        return self.__defaultLegSum

    @property
    def markToMarket(self):
        return self.__premiumLegSum - self.__defaultLegSum

    @property
    def lamda(self):
        return self.__lamda

    @property
    def seed(self):
        return self.__seed

    def generateProbSurvival(self):
        """
        using $\exp^{-\lambda*T_i}$
        :return:
        """
        pt = ones(len(self.timesteps))
        for index, t in enumerate(self.timesteps):
            if t > 0:
                ps = exp(self.lamda * -1 * t)
                put(pt, index, ps)
        return pt

    def generateProbDefault(self):
        """
        using $P(T,0) = P_{i-1} - P_i$
        :return:
        """
```

```
Before optimisation:  -7772.71007138

/Users/Admin/anaconda/lib/python2.7/site-packages/ipykernel/__main__.py:23: Runt
imeWarning: invalid value encountered in divide
```

We need to now compute what is the value that should be used. So by using an optimising algorithm we can minimise the value of markToMarket to 0. This will then give us the value that we should use for the hazard rate λ.

In [2]:
```
def calibrateCDS(lamda=float, seed=0.01):
    global CreditDefaultSwap
    c = CreditDefaultSwap(N=notional, timesteps=payments, discountFactors=df, lam
da=lamda, seed=seed)
    return c.markToMarket

from scipy.optimize import fsolve

calibratedLambda = fsolve(calibrateCDS, lamda)

print 'Calibrated Lambda: ', calibratedLambda

c = CreditDefaultSwap(N=notional, timesteps=payments, discountFactors=df, lamda=c
alibratedLambda, seed=seed)
```

```
Calibrated Lambda:  [ 0.01663204]

/Users/Admin/anaconda/lib/python2.7/site-packages/ipykernel/__main__.py:23: Runt
imeWarning: invalid value encountered in divide
```

The optimisation routine is provided by parts of the SciPy Optimize library, so we will need to import this. it needs to be wrapped in function so that the optimiser is able to call it with different values.

In [5]:
```
print 'After optimisation: ', c.markToMarket
```

```
After optimisation:  -1.81898940355e-12
```

Which is pretty close to zero so we can go ahead and use this value for our lambda in our simulation. Of course that was using some test data to validate that our implementation is correct.

Random Number Generation

In order to facilitate simplified random number generation, it seemed appropriate to setup a utilty class to do this. It has also given the opportunity to explore the ability to use the GPU for both Pseudo Random Number Generation and Quasi Random Number Generation.

Additional libraries:

ghalton - https://github.com/fmder/ghalton (https://github.com/fmder/ghalton)

sobol_seq - https://github.com/naught101/sobol_seq (https://github.com/naught101/sobol_seq)

conda (package manager for Anaconda) does not install these properly so it is best to just clone and install it in the normal python way:

```
cd itemToInstall/
python setup.py build
python setup.py test #(sometimes this fails, when no tests have been written)
python setup.py install #(sometimes requires admin privs depending on way python has b
een installed)]
```

these items are included on the usb drives.

In [1]:
```
import ghalton
import sobol_seq
from accelerate.cuda.rand import PRNG, QRNG
from numpy import array, empty, random, square, log, sqrt


def getPseudoRandomNumbers_Uniform(length=int):
    """

    generates a an array of psuedo random numbers from uniform distribution using
 numpy

    :param length:
    :return:
    """
    return random.uniform(size=length)


def getPseudoRandomNumbers_Uniform_cuda(length=int):
    # type: (object) -> object
    """

    generates a an array of psuedo random numbers from uniform distribution using
 CUDA

    :rtype: ndarray
    :param length:
    :return:
    """
    prng = PRNG(rndtype=PRNG.XORWOW)
    rand = empty(length)
    prng.uniform(rand)

    return rand


def getPseudoRandomNumbers_Standard(shape=tuple):
    """

    generates a an array of psuedo random numbers from standard normal distributi
on using numpy

    :param length:
    :return:
    """
    return random.normal(size=shape)


def getPseudoRandomNumbers_Standard_cuda(shape=tuple):
    # type: (object) -> object
    """

    generates a an array of psuedo random numbers from standard normal distributi
on using CUDA

    :rtype: ndarray
    :param length:
    :return:
    """
    prng = PRNG(rndtype=PRNG.XORWOW)
    rand = empty(shape)
```

Monte Carlo Simulation

We will now use a Monte Carlo simulation to compute the expected < DL > and expected < PL >. With the expectation of the fair spread being:

$$E[s] = \frac{\langle DL\rangle}{\langle PL\rangle}$$

To bootstrap the LIBOR curve we will use the One Factor Libor Model [1]. Originally implemented in C++, this has been recoded in Python. It could be argued that this would imply a slight loss in performance from C++, with C++ being as low level as possible without becoming assembler. An optimisation for python here would be to use a Cython wrapper to get as close as possible to the original C. One of the the languages advantages here is not to have to manage memory and pointers manually.

```
In [2]: import app.rng as rng

        from numpy import ndarray, zeros, sqrt, put, exp, linspace, array, append, square
        , mean, percentile, insert

        gpuEnabled = False
        # pushing the size of rng generated above 100000 causes GPU to run out of space
        # possible optization is to load it into a 3d vector shape instead of flat struct
        ure.

        def initRandSource():
            randN = 100000
            randSourceGPU = rng.getPseudoRandomNumbers_Uniform_cuda(randN) if gpuEnabled
        else []
            randSourceCPU = rng.getPseudoRandomNumbers_Uniform(randN)
            return randSourceGPU if gpuEnabled else randSourceCPU

        randSource = initRandSource()

        # 5Y tenor
        noOfYears = 5.
        # 6M payments
        paymentFrequency = 0.5
        yearFraction = paymentFrequency / noOfYears
        noOfPayments = noOfYears / paymentFrequency

        # no of timesteps
        timesteps = linspace(0, 1, noOfPayments + 1)

        print 'Timesteps / yearFraction: ', timesteps

        # taken from BOE spot curve data
        initRates_BOE_6m = [0.423546874, 0.425980925, 0.45950348, 0.501875772, 0.55147301
        1, 0.585741857, 0.626315731,
                            0.667316554, 0.709477279, 0.753122018]

        # to simulate the L6M
```

```
Timesteps / yearFraction:  [ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1
.  ]
```

In [3]:
```
from numpy import array

class lSimulation(object):
    """
    this class is used a container for a single simulation
    providing convenenience methods to retrieve
    markToMarket values
    eeA = expected exposure from the simulation for Counterparty A
    eeB = expected exposure from the simulation for Counterparty B
    """
    def __init__(self, liborTable=array, dfTable=array,
                 notional=1000000, dt=0.25, k=0.04):
        """


        :type k: float
        :type dt: float
        :type notional: float
        :type dfTable: ndarray
        :type liborTable: ndarray
        """
        self.__liborTable = liborTable
        self.__dfTable = dfTable

        # calculate payments for each timestep using the given notional, tenor, f
ixed rate,
        # floating(simulated) and discount factors (simulated)
        self.payments = self.calcPayments(notional, dt, k)

        self.mtm = array([flt - fxd for flt, fxd in self.payments])

        # expected exposure for counterParty A (using positive mtm)
        self.eeA = [max(L - K, 0) for L, K in self.payments]

        # expected exposure for counterParty B (using negative mtm)
        self.eeB = [min(L - K, 0) for L, K in self.payments]


    def liborTable(self):
        return self.__liborTable

    def dfTable(self):
        return self.__dfTable

    def calcPayments(self, notional=float, dt=float, fixed=-1.0):
        """
        calculate payments for the simulation of the Fwd rates and discount facto
rs
        given notional and tenor

        if fixed is set it will use a fixed rate
        there is the possibility here of a negative interest rate but that is out
side the
        scope of this exercise
        :param notional:
        :param dt:
        :param fixed:
        :return: float
        """
        payments = []

        for index in range(0, len(self.__liborTable)):
```

In [4]:
```
class LMM1F:
    def __init__(self, strike=0.05, alpha=0.5, sigma=0.15, dT=0.5, nSims=10,
                 initialSpotRates=ndarray, notional=float):
        """

        :param strike:  caplet
        :param alpha: daycount factor
        :param sigma: fwd rates volatility
        :param dT: 6M (year fraction of 0.5 is default)
        :param nSims: no of simulations to run
        """
        self.K = strike
        self.alpha = alpha
        self.sigma = sigma
        self.dT = dT
        self.N = len(initialSpotRates) - 1
        self.M = nSims
        self.initialSpotRates = initialSpotRates
        self.notional = notional

    def simulateLMMviaMC(self):

        l = zeros(shape=(self.N + 1, self.N + 1), dtype=float)
        d = zeros(shape=(self.N + 2, self.N + 2), dtype=float)

        # init zero based tables

        # init spot rates
        for index, s in enumerate(self.initialSpotRates):
            l[index][0] = s

        simulations = []

        for i in xrange(self.M):
            # setup brownian motion multipliers
            gbm_multipliers = self.initWeinerProcess(self.N + 1)

            # computeFwdRatesTableau
            l, d = self.computeTableaus(self.N, self.alpha, self.sigma, l, self.d
T,
                                        gbm_multipliers, d)

            # computeDiscountRatesTableau
            # d = self.computeDiscountRatesTableau(self.N, l, d, self.alpha)

            storeValue = lSimulation(l, d, self.notional, self.dT)
            simulations.append(storeValue)
        return array(simulations)

    def getSimulationData(self):
        simulations = self.simulateLMMviaMC()
        return simulations

    def initWeinerProcess(self, length=int):
        seq = zeros(self.N + 1)
        for dWi in xrange(length):
            dW = sqrt(self.dT) * rng.getBoxMullerSample(randSource)
            put(seq, dWi, dW)

        if debug:
            print 'Discount Factors', seq
```

Now that we have defined the two classes above lets run 1000 simulations. Timing is included to provide performance impact.

In [10]:
```
import time
from app.utils import printTime

debug = False
n = 10
notional = 1000000
initRates = array([0.01, 0.03, 0.04, 0.05, 0.07])
irEx = LMM1F(nSims=n, initialSpotRates=initRates, dT=yearFraction, notional=notio
nal)
start_time = time.clock()
a = irEx.getSimulationData()
printTime('CPU: generating 10 simulations', start_time)

n = 100
irEx = LMM1F(nSims=n, initialSpotRates=initRates, dT=yearFraction, notional=notio
nal)
start_time = time.clock()
a = irEx.getSimulationData()
printTime('CPU: generating 100 simulations', start_time)

n = 1000
irEx = LMM1F(nSims=n, initialSpotRates=initRates, dT=yearFraction, notional=notio
nal)
start_time = time.clock()
a = irEx.getSimulationData()
printTime('CPU: generating 1000 simulations', start_time)

n = 10000
irEx = LMM1F(nSims=n, initialSpotRates=initRates, dT=yearFraction, notional=notio
nal)
start_time = time.clock()
a = irEx.getSimulationData()
printTime('CPU: generating 10000 simulations', start_time)
```

```
CPU: generating 10 simulations took: 0.015847000000000833 seconds
CPU: generating 100 simulations took: 0.07040799999999692 seconds
CPU: generating 1000 simulations took: 1.1697109999999995 seconds
CPU: generating 10000 simulations took: 8.890600999999997 seconds
```

CPU: generating simulation data took: 0.03408100000000047 seconds CPU: generating simulation data took: 0.13604000000000038 seconds CPU: generating simulation data took: 0.9945149999999998 seconds CPU: generating simulation data took: 7.875406 seconds

next we need the forward rates and the discount factors.

$L_I = S_I$

$L1 = \frac{S_i T_i - S_{i-1} T_{i-1}}{T_i - T_{i-1}}$

Forward rates can be derived from the spot rates in continous time.

Discount Factors

$$DF_i = exp(-S_i T_i)$$

Default Probabilities

$$PD_i = exp(-\lambda T_{i-1}) - exp(-\lambda T_i) \quad \forall i = 1, 2, 3, 4$$

Conclusions

Credit Model pricing is particularly suited to computational methods since it can be a laborious process to get all the mechanics of the models in place.

```
In [ ]:
```

References

[1] '*The One Factor Libor Market Model Using Monte Carlo Simulation: An Empirical Investigation*', Pena, di Sabatino,Ligato,Ventura,Bertagna. Dec 2010

[2] '*Advanced Quantitative Finance with C++*', Pena. 2014

[3] '*Python for Finance*', Yves Hilpisch. 2014

[4] '*emscriptem*', http://kripken.github.io/emscripten-site/ (http://kripken.github.io/emscripten-site/). - C++ to JavaScript compiler.

[5] '*VisPy*', http://cyrille.rossant.net/compiler-data-visualization/ (http://cyrille.rossant.net/compiler-data-visualization/)

[3] '*Title*', Author, (Mon YYYY)

```
In [ ]:
```

```
In [ ]:
```