FinancePy

Dominic O'Kane

December 23, 2019

Contents

FinancePy FinancePy is a library of native Python functions for valuing, selecting and risk-managing financial assets.

The aim of this library for me has been to provide a comprehensive and accessible Python library for financial calculations that can be used by students to learn about financial derivatives. It can also be used by academics and practitioners to perform the pricing and risk-management of complex financial products, albeit without any warranties. Users should perform their own testing. See the license for the full disclaimer.

I intend that subsequent versions will also include asset selection, portfolio-level risk management, regulatory calculations and market analysis tools.

In general my objectives have been: 1) To make the code as simple as possible so that students and those with a basic Python fluency can understand and check the code. 2) To keep all the code in Python so users can look through the code to the lowest level. 3) To offset the performance impact of (2) by leveraging Numba to make the code as fast as possible without resorting to Cython. 4) To make the design product-based rather than model-based so someone wanting to price a specific exotic option can easily find that without having to worry too much about the model just use the default unless they want to. 5) To make the library as complete as possible so a user can find all their required finance-related functionality in one place. This is better for the user as they only have to learn one interface. 6) To avoid complex designs as I do not want to make it too hard for unskilled Python programmers to use the library. 7) To have good documentation and easy-to-follow examples. 8) To make it easy for interested parties to contribute.

In many cases the valuations should be close to if not identical to those produced by financial systems such as Bloomberg. However for some products, larger value differences may arise due to differences in date generation and interpolation schemes. Over time I expect to reduce the size of such differences.

Dependencies FinancePy depends on Numpy and Numba and Scipy.

Installation FinancePy can be installed via pip as ¿¿¿¿ pip install financepy

Changelog See the changelog for a detailed history of changes

Contributions Contributions are welcome, as long as you don’t mind camel case ;-)

License MIT

Chapter 1

..//finutils

1.1 Introduction

1.2 FinCalendar

1.2.0.1 Enumerated Type: FinDayAdjustTypes

- NONE
- FOLLOWING
- MODIFIED_FOLLOWING
- PRECEDING
- MODIFIED_PRECEDING

1.2.0.2 Enumerated Type: FinCalendarTypes

- TARGET
- US
- UK
- WEEKEND
- JAPAN
- NONE

1.2.0.3 Enumerated Type: FinDateGenRuleTypes

- FORWARD
- BACKWARD

Class: FinCalendar(object)

Class to manage designation of payment dates as holidays according to a regional or country-specific calendar convention specified by the user.

Data Members

- _type

Functions

__init__

Create a calendar based on a specified calendar type.

```
def __init__(self, calendarType):
```

adjust

Adjust a payment date if it falls on a holiday according to the specified business day convention.

```
def adjust(self, dt, busDayConventionType):
```

isBusinessDay

Determines if a date is a business day according to the specified calendar. If it is it returns True, otherwise False.

```
def isBusinessDay(self, dt):
```

getHolidayList

generates a list of holidays in a specific year for the specified calendar. Useful for diagnostics.

```
def getHolidayList(self, year):
```

easterMonday

Get the day in a given year that is Easter Monday. This is not easy to compute so we rely on a pre-calculated array.

```
def easterMonday(self, y):
```

str

def str(self):

```
def str(self):
```

1.3 FinDate

Class: FinDate()

Date class to manage dates that is simple to use and includes a number of useful date functions used frequently in Finance.

Data Members

- _y
- _m
- _d
- _excelDate
- _weekday

Functions

__init__

Create a date given year, month and day of month. The order is not enforced so 4th July 2019 can be created as FinDate(4,7,2019) or as FinDate(2019,7,4) so long as the middle number is the month. The year must be a 4-digit number greater than or equal to 1900.

```
def __init__(self, y_or_d, m, d_or_y):
```

fromDatetime

Construct a FinDate from a datetime as this is often needed if we receive inputs from other Python objects such as Pandas dataframes.

```
def fromDatetime(dt):
```

refresh

Update internal representation of date as number of days since the 1st Jan 1900. This is same as Excel convention.

```
def refresh(self):
```

__lt__

def __lt__(self, other):

```
def __lt__(self, other):
```

__gt__

def __gt__(self, other):

```
def __gt__(self, other):
```

__le__

def __le__(self, other):

```
def __le__(self, other):
```

__ge__

def __ge__(self, other):

```
def __ge__(self, other):
```

__sub__

def __sub__(self, other):

```
def __sub__(self, other):
```

__eq__

def __eq__(self, other):

```
def __eq__(self, other):
```

isWeekend

returns True if the date falls on a weekend.

```
def isWeekend(self):
```

addDays

Returns a new date that is numDays after the FinDate.

```
def addDays(self, numDays):
```

addWorkDays

Returns a new date that is numDays working days after FinDate.

```
def addWorkDays(self, numDays):
```

addMonths

Returns a new date that is mm months after the FinDate.

```
def addMonths(self, mm):
```

nextCDSDate

Returns a CDS date that is mm months after the FinDate. If no argument is supplied then the next CDS date after today is returned.

```
def nextCDSDate(self, mm=0):
```

thirdWednesdayOfMonth

For a specific month and year this returns the day number of the 3rd Wednesday by scanning through dates in the third week.

```
def thirdWednesdayOfMonth(self, m, y):
```

nextIMMDate

This function returns the next IMM date after the current date This is a 3rd Wednesday of Jun, March, Sep or December

```
def nextIMMDate(self):
```

addTenor

Return the date following the FinDate by a period given by the tenor which is a string consisting of a number and a letter, the letter being d, w, m , y for day, week, month or year. This is case independent. For example 10Y means 10 years while 120m also means 10 years.

```
def addTenor(self, tenor):
```

datediff

Calculate the number of dates between two dates.

```
def datediff(d1, d2):
```

date

def date(self):

```
def date(self):
```

__str__

def __str__(self):

```
def __str__(self):
```

print

def print(self):

```
def print(self):
```

dailyWorkingDaySchedule

def dailyWorkingDaySchedule(self, startDate, endDate):

```
def dailyWorkingDaySchedule(self, startDate, endDate):
```

1.4 FinDayCount

1.4.0.1 Enumerated Type: FinDayCountTypes

- THIRTY_E_360_ISDA
- THIRTY_E_360_PLUS_ISDA
- ACT_ACT_ISDA
- ACT_ACT_ICMA
- ACT_365_ISDA
- THIRTY_360
- THIRTY_360_BOND
- THIRTY_E_360
- ACT_360
- ACT_365_FIXED
- ACT_365_LEAP

Class: FinDayCount(object)

Calculate the fractional day count between two dates according to a specified day count convention.

Data Members

- _type

Functions

__init__

Create Day Count convention by passing in the Day Count Type.

```
def __init__(self, dccType):
```

yearFrac

Calculate the year fraction between dates dt1 and dt2 using the specified day count convention.

```
def yearFrac(self, dt1, dt2, dt3=None):
```

__str__

Returns the calendar type as a string.

```
def __str__(self):
```

1.5 FinError

Class: FinError(Exception)

Simple error class specific to FinPy. Need to decide how to handle FinancePy errors. Work in progress.

Data Members

- _message

Functions

__init__

Create FinError object by passing a message string.

```
def __init__(self, message):
```

print

def print(self):

```
def print(self):
```

func_name

def func_name():

```
def func_name():
```

isNotEqual

def isNotEqual(x, y, tol=1e-6):

```
def isNotEqual(x, y, tol=1e-6):
```

1.6 FinFrequency

1.6.0.1 Enumerated Type: FinFrequencyTypes

- ANNUAL
- SEMI_ANNUAL
- QUARTERLY
- MONTHLY

FinFrequency

This is a function that takes in a Frequency Type and returns an integer for the number of times a year a payment occurs.

```
def FinFrequency(frequencyType):
```

1.7 FinGlobalVariables

1.8 FinHelperFunctions

printTree

Function that prints a binomial or trinonial tree to screen for the purpose of debugging.

```
def printTree(array, depth=None):
```

inputFrequency

Function takes a frequency number and checks if it is valid.

```
def inputFrequency(f):
```

inputTime

Validates a time input in relation to a curve. If it is a float then it returns a float as long as it is positive. If it is a FinDate then it converts it to a float. If it is a Numpy array then it returns the array as long as it is all positive.

```
def inputTime(dt, curve):
```

listdiff

Calculate a vector of differences between two equal sized vectors.

```
def listdiff(a, b):
```

dotproduct

Fast calculation of dot product using Numba.

```
def dotproduct(xVector, yVector):
```

frange

def frange(start, stop, step):

```
def frange(start, stop, step):
```

normaliseWeights

Normalise a vector of weights so that they sum up to 1.0.

```
def normaliseWeights(wtVector):
```

1.9 FinMath

isLeapYear

Test whether year y is a leap year - if so return True, else False

```
def isLeapYear(y):
```

scale

Scale all of the elements of an array by the same amount factor.

```
def scale(x, factor):
```

testMonotonicity

Check that an array of doubles is monotonic and strictly increasing.

```
def testMonotonicity(x):
```

testRange

Check that all of the values of an array fall between a lower and upper bound.

```
def testRange(x, lower, upper):
```

maximum

Determine the array in which each element is the maximum of the corresponding element in two equally length arrays a and b.

```
def maximum(a, b):
```

maxaxis

Perform a search for the vector of maximum values over an axis of a 2D Numpy Array

```
def maxaxis(s):
```

minaxis

Perform a search for the vector of minimum values over an axis of a 2D Numpy Array

```
def minaxis(s):
```

covar

Calculate the Covariance of two arrays of numbers. TODO: check that this works well for Numpy Arrays and add NUMBA function signature to code. Do test of timings against Numpy.

```
def covar(a, b):
```

pairGCD

Determine the Greatest Common Divisor of two integers using Euclids algorithm. TODO - compare this with math.gcd(a,b) for speed. Also examine to see if I should not be declaring inputs as integers for NUMBA.

```
def pairGCD(v1, v2):
```

nprime

Calculate the first derivative of the Cumulative Normal CDF which is simply the PDF of the Normal Distribution

```
def nprime(x):
```

heaviside

Calculate the Heaviside function for x

```
def heaviside(x):
```

frange

def frange(start, stop, step):

```
def frange(start, stop, step):
```

normpdf

Calculate the probability density function for a Gaussian (Normal) function at value x

```
def normpdf(x):
```

normcdf_fast

Fast Normal CDF function based on XXX

```
def normcdf_fast(x):
```

normcdf_integrate

Calculation of Normal Distribution CDF by simple integration which can become exact in the limit of the number of steps tending towards infinity. This function is used for checking as it is slow since the number of integration steps is currently hardcoded to 10,000.

```
def normcdf_integrate(x):
```

normcdf_slow

Calculation of Normal Distribution CDF accurate to 1d-15. This method is faster than integration but slower than other approximations. Reference: J.L. Schonfelder, Math Comp 32(1978), pp 1232-1240.

```
def normcdf_slow(z):
```

normcdf

This is the Normal CDF function which forks to one of three of the implemented approximations. This is based on the choice of the fast flag variable. A value of 1 is the fast routine, 2 is the slow and 3 is the even slower integration scheme.

```
def normcdf(x, fastFlag):
```

N

This is the shortcut to the default Normal CDF function and is currently hardcoded to the fastest of the implemented routines. This is the most widely used way to access the Normal CDF.

```
def N(x):
```

phi3

Bivariate Normal CDF function to upper limits $b1$ and $b2$ which uses integration to perform the innermost integral. This may need further refinement to ensure it is optimal as the current range of integration is from -7 and the integration steps are dx = 0.001. This may be excessive.

```
def phi3(b1, b2, b3, r12, r13, r23):
```

norminvcdf

This algorithm computes the inverse Normal CDF and is based on the algorithm found at (http:#home.online.no/ pjacklam/notes/invnorm/) which is by John Herrero (3-Jan-03)

```
def norminvcdf(p):
```

M

def M(a, b, c):

```
def M(a, b, c):
```

phi2

Drezner and Wesolowsky implementation of bi-variate normal

```
def phi2(h1, hk, r):
```

corrMatrixGenerator

Utility function to generate a full rank n x n correlation matrix with a flat correlation structure and value rho.

```
def corrMatrixGenerator(rho, n):
```

1.10 FinRateConverter

Class: FinRateConverter(object)

Convert rates between different compounding conventions. This is not used.

Data Members

- name
- months

Functions

__init__

def __init__(self):

```
def __init__(self):
```

str

def str(self):

```
def str(self):
```

1.11 FinSchedule

Class: FinSchedule(object)

A Schedule is a vector of dates generated according to ISDA standard rules which starts on the next date after the start date and runs up to an end date. Dates are adjusted to a provided calendar. The zeroth element is the PCD and the first element is the NCD

Data Members

- _startDate
- _endDate
- _frequencyType
- _calendarType
- _busDayAdjustType
- _dateGenRuleType
- _adjustedDates

Functions

__init__

Create FinSchedule object.

```
def __init__(self,
             startDate,
             endDate,
             frequencyType=FinFrequencyTypes.ANNUAL,
             calendarType=FinCalendarTypes.WEEKEND,
             busDayAdjustType=FinDayAdjustTypes.FOLLOWING,
             dateGenRuleType=FinDateGenRuleTypes.BACKWARD):
```

flows

Returns a list of the schedule of dates.

```
def flows(self):
```

generate

Generate schedule of dates according to specified date generation rules and also adjust these dates for holidays according to the business day convention and the specified calendar.

```
def generate(self):
```

print

Print out the details of the schedule and the actual dates. This can be used for providing transparency on schedule calculations.

```
def print(self):
```

1.12 FinStatistics

mean

Calculate the arithmetic mean of a vector of numbers x.

```
def mean(x):
```

stdev

Calculate the standard deviation of a vector of numbers x.

```
def stdev(x):
```

stderr

Calculate the standard error estimate of a vector of numbers x.

```
def stderr(x):
```

var

Calculate the variance of a vector of numbers x.

```
def var(x):
```

moment

Calculate the m-th moment of a vector of numbers x.

```
def moment(x, m):
```

correlation

Calculate the correlation between two series x1 and x2.

```
def correlation(x1, x2):
```

1.13 FinTenor

1.14 FinTestCases

1.14.0.1 Enumerated Type: FinTestCaseMode

- SAVE_TEST_CASES
- ANALYSE_TEST_CASES
- DEBUG_TEST_CASES

Class: FinTestCases()

Test case framework for FinancePy. - The basic step is that we generate a GOLDEN folder that creates an output file for each testcase which is assumed to be correct. This can be done by running the test cases Python file with the globalTestCaseMode flag set to FinTestCaseMode.SAVE_TEST_CASES. - The second step is that we change the value of globalTestCaseMode to FinTestCaseMode.ANALYSE_TEST_CASES and then run the test scripts. This time they save a copy of the output to the COMPARE folder. Finally, a function called compareTestCases() is used to compare the new output with the GOLDEN output and states whether anything has changed. - The output of a test case has three forms each with its own method: 1) print - this outputs comma separated values 2) header - this must precede any print statement and labels the output columns 3) banner - this is any single string line separator Note that the header TIME is special as it tells the analysis that the value in the corresponding column is a timing and so its value is allowed to change without triggering an error.

Data Members

- _carefulMode
- _verbose
- _mode
- _moduleName
- _foldersExist
- _rootFolder
- _headerFields
- _goldenFolder
- _compareFolder
- _goldenFilename
- _compareFilename
- _headerFields[colNum]

Functions

__init__

Create the TestCase given the module name and whether we are in GOLDEN or COMPARE mode.

```
def __init__(self, moduleName, mode):
```

print

Print comma separated output to GOLDEN or COMPARE directory.

```
def print(self, *args):
```

banner

Print a banner on a line to the GOLDEN or COMPARE directory.

```
def banner(self, txt):
```

header

Print a header on a line to the GOLDEN or COMPARE directory.

```
def header(self, *args):
```

compareRows

Compare the contents of two rows in GOLDEN and COMPARE folders.

```
def compareRows(self, goldenRow, compareRow, rowNum):
```

compareTestCases

Compare output of COMPARE mode to GOLDEN output

```
def compareTestCases(self):
```

Chapter 2

..//products//equities

2.1 Introduction

2.2 FinAmericanOption

2.2.0.1 Enumerated Type: FinImplementations

- CRR_TREE
- BARONE_ADESI_APPROX

Class: FinAmericanOption()

Class that performs the valuation of an American style option on a dividend paying stock. Can easily be extended to price American style FX options.

Data Members

- _expiryDate
- _strikePrice
- _optionType

Functions

__init__

optionType):

```
def __init__(self,
             expiryDate,
             strikePrice,
             optionType):
```

value

numStepsPerYear=100):

```
def value(self,
          valueDate,
          stockPrice,
          discountCurve,
          dividendYield,
          model,
          numStepsPerYear=100):
```

delta

model):

```
def delta(self,
          valueDate,
          stockPrice,
          discountCurve,
          dividendYield,
          model):
```

gamma

model):

```
def gamma(self,
          valueDate,
          stockPrice,
          discountCurve,
          dividendYield,
          model):
```

vega

model):

```
def vega(self,
         valueDate,
         stockPrice,
         discountCurve,
         dividendYield,
         model):
```

theta

model):

```
def theta(self,
          valueDate,
          stockPrice,
          discountCurve,
          dividendYield,
          model):
```

rho

model):

```
def rho(self,
        valueDate,
        stockPrice,
        discountCurve,
        dividendYield,
        model):
```

crrTreeVal

Value an American option using a Binomial Treee

```
def crrTreeVal(stockPrice,
               riskFreeRate,
               dividendYield,
               volatility,
               numStepsPerYear,
               timeToExpiry,
               optionType,
               strikePrice,
               isEven):
```

2.3 FinAsianOption

Class: FinAsianOption(FinOption)

class FinAsianOption(FinOption):

Data Members

- _startAveragingDate
- _expiryDate
- _strikePrice
- _optionType
- _numObservations

Functions

__init__

numberOfObservations=0):

```
def __init__(self,
             startAveragingDate,
             expiryDate,
             strikePrice,
             optionType,
             numberOfObservations=0):
```

value

accruedAverage=None):

```
def value(self,
          valueDate,
          stockPrice,
          discountCurve,
          dividendYield,
          model,
          valuationMethod,
          accruedAverage=None):
```

valueGeometric

accruedAverage):

```
def valueGeometric(self,
                   valueDate,
                   stockPrice,
                   discountCurve,
                   dividendYield,
                   model,
                   accruedAverage):
```

valueCurran

accruedAverage):

```
def valueCurran(self,
                valueDate,
                stockPrice,
                discountCurve,
                dividendYield,
                model,
                accruedAverage):
```

valueTurnbullWakeman

accruedAverage):

```
def valueTurnbullWakeman(self,
                         valueDate,
                         stockPrice,
                         discountCurve,
                         dividendYield,
                         model,
                         accruedAverage):
```

valueMC

accruedAverage):

```
def valueMC(self,
            valueDate,
            stockPrice,
            discountCurve,
            dividendYield,
            model,
            numPaths,
            seed,
            accruedAverage):
```

valueMC_fast

accruedAverage):

```
    def valueMC_fast(self,
                     valueDate,
                     stockPrice,
                     discountCurve,
                     dividendYield,
                     model,
                     numPaths,
                     seed,
                     accruedAverage):
```

valueMC_fast_CV

accruedAverage):

```
    def valueMC_fast_CV(self,
                        valueDate,
                        stockPrice,
                        discountCurve,
                        dividendYield,
                        model,
                        numPaths,
                        seed,
                        accruedAverage):
```

valueMC_NUMBA

accruedAverage):

```
def valueMC_NUMBA(t0, t, tau, K, n, optionType,
                  stockPrice,
                  interestRate,
                  dividendYield,
                  volatility,
                  numPaths,
                  seed,
                  accruedAverage):
```

valueMC_fast_NUMBA

accruedAverage):

```
def valueMC_fast_NUMBA(t0, t, tau, K, n, optionType,
                       stockPrice,
                       interestRate,
                       dividendYield,
                       volatility,
                       numPaths,
                       seed,
                       accruedAverage):
```

valueMC_fast_CV_NUMBA

v_g_exact):

```
def valueMC_fast_CV_NUMBA(t0, t, tau, K, n, optionType,
                          stockPrice,
                          interestRate,
                          dividendYield,
                          volatility,
                          numPaths,
                          seed,
                          accruedAverage,
                          v_g_exact):
```

2.4 FinBarrierOption

2.4.0.1 Enumerated Type: FinBarrierTypes

- DOWN_AND_OUT_CALL
- DOWN_AND_IN_CALL
- UP_AND_OUT_CALL
- UP_AND_IN_CALL
- UP_AND_OUT_PUT
- UP_AND_IN_PUT
- DOWN_AND_OUT_PUT
- DOWN_AND_IN_PUT

Class: FinBarrierOption(FinOption)

class FinBarrierOption(FinOption):

Data Members

- _expiryDate
- _strikePrice
- _barrierLevel
- _numObservationsPerYear
- _optionType

Functions

__init__

numObservationsPerYear):

```
def __init__(self,
             expiryDate,
             strikePrice,
             optionType,
             barrierLevel,
             numObservationsPerYear):
```

value

model):

```
def value(
        self,
        valueDate,
        stockPrice,
        discountCurve,
        dividendYield,
        model):
```

valueMC

seed=4242):

```
def valueMC(
        self,
        valueDate,
        stockPrice,
        discountCurve,
        processType,
        modelParams,
        numAnnSteps=252,
        numPaths=10000,
        seed=4242):
```

2.5 FinBasketOption

Class: FinBasketOption(FinOption)

class FinBasketOption(FinOption):

Data Members

- _expiryDate
- _strikePrice
- _optionType
- _numAssets

Functions

__init__

numAssets):

```
def __init__(self,
             expiryDate,
             strikePrice,
             optionType,
             numAssets):
```

validate

betas):

```
def validate(self,
             stockPrices,
             dividendYields,
             volatilities,
             betas):
```

value

betas):

```
def value(self,
          valueDate,
          stockPrices,
          discountCurve,
          dividendYields,
          volatilities,
          betas):
```

valueMC

seed=4242):

```
def valueMC(self,
            valueDate,
            stockPrices,
            discountCurve,
            dividendYields,
            volatilities,
            betas,
            numPaths=10000,
            seed=4242):
```

2.6 FinBinomialTree

2.6.0.1 Enumerated Type: FinTreePayoffTypes

- FWD_CONTRACT
- VANILLA_OPTION
- DIGITAL_OPTION
- POWER_CONTRACT
- POWER_OPTION
- LOG_CONTRACT
- LOG_OPTION

2.6.0.2 Enumerated Type: FinTreeExerciseTypes

- EUROPEAN
- AMERICAN

Class: FinBinomialTree()

class FinBinomialTree():

Data Members

- m_optionValues
- m_stockValues
- m_upProbabilities
- m_numSteps
- m_numNodes

Functions

__init__

def __init__(self):

```
def __init__(self):
```

value

payoffParams):

```
def value(self,
          stockPrice,
          discountCurve,
          dividendYield,
          volatility,
          numSteps,
          valueDate,
          payoff,
          expiryDate,
          payoffType,
          exerciseType,
          payoffParams):
```

validatePayoff

def validatePayoff(payoffType, payoffParams):

```
def validatePayoff(payoffType, payoffParams):
```

payoffValue

def payoffValue(s, payoffType, payoffParams):

```
def payoffValue(s, payoffType, payoffParams):
```

valueOnce

payoffParams):

```
def valueOnce(stockPrice,
              r,
              dividendYield,
              volatility,
              numSteps,
              timeToExpiry,
              payoffType,
              exerciseType,
              payoffParams):
```

2.7 FinBlack

Class: BlackModel()

class BlackModel():

Data Members

No data members found.

Functions

value

callOrPut):

```
def value(self,
          forwardRate,
          strikeRate,
          timeToExpiry,
          sigma,
          callOrPut):
```

2.8 FinCompoundOption

Class: FinCompoundOption(FinOption)

class FinCompoundOption(FinOption):

Data Members

- _expiryDate1
- _expiryDate2
- _strikePrice1
- _strikePrice2
- _optionType1
- _optionType2

Functions

__init__

optionType2):

```
def __init__(self,
             expiryDate1,
             expiryDate2,
             strikePrice1,
             strikePrice2,
             optionType1,
             optionType2):
```

value

model):

```
def value(self,
          valueDate,
          stockPrice,
          discountCurve,
          dividendYield,
          model):
```

valueTree

numSteps=200):

```
def valueTree(self,
              valueDate,
              stockPrice,
              discountCurve,
              dividendYield,
              model,
              numSteps=200):
```

impliedStockPrice

model):

```
def impliedStockPrice(self,
                      stockPrice,
                      expiryDate1,
                      expiryDate2,
                      strikePrice1,
                      strikePrice2,
                      optionType2,
                      interestRate,
                      dividendYield,
                      model):
```

f

def f(s0, *args):

```
def f(s0, *args):
```

valueOnce

numSteps):

```
def valueOnce(stockPrice,
              riskFreeRate,
              dividendYield,
              volatility,
              t1,
              t2,
              optionType1,
              optionType2,
              k1,
              k2,
              numSteps):
```

2.9 FinDigitalOption

Class: FinDigitalOption(FinOption)

class FinDigitalOption(FinOption):

Data Members

- _expiryDate
- _strikePrice
- _optionType

Functions

__init__

optionType):

```
def __init__(self,
             expiryDate,
             strikePrice,
             optionType):
```

value

model):

```
def value(self,
          valueDate,
          stockPrice,
          discountCurve,
          dividendYield,
          model):
```

valueMC

seed=4242):

```
def valueMC(self,
            valueDate,
            stockPrice,
            discountCurve,
            dividendYield,
            model,
            numPaths=10000,
            seed=4242):
```

2.10 FinEquityModelTypes

Class: FinEquityModel(object)

Data Members

- _parentType
- _volatility
- _implementation

Functions

__init__

def __init__(self):

```
def __init__(self):
```

Class: FinEquityModelBlackScholes(FinEquityModel)

class FinEquityModelBlackScholes(FinEquityModel):

Data Members

- _parentType
- _volatility
- _implementation

Functions

__init__

def __init__(self, volatility):

```
def __init__(self, volatility):
```

Class: FinEquityModelHeston(FinEquityModel)

class FinEquityModelHeston(FinEquityModel):

Data Members

- _parentType
- _volatility
- _meanReversion
- _implementation

Functions

__init__

def __init__(self, volatility, meanReversion):

```
def __init__(self, volatility, meanReversion):
```

2.11 FinFixedLookbackOption

2.11.0.1 Enumerated Type: FinFixedLookbackOptionTypes

- FIXED_CALL
- FIXED_PUT

Class: FinFixedLookbackOption(FinOption)

class FinFixedLookbackOption(FinOption):

Data Members

- _expiryDate
- _optionType
- _optionStrike

Functions

__init__

optionStrike):

```
def __init__(self,
             expiryDate,
             optionType,
             optionStrike):
```

value

stockMinMax):

```
def value(self,
          valueDate,
          stockPrice,
          discountCurve,
          dividendYield,
          volatility,
          stockMinMax):
```

valueMC

seed=4242):

```
def valueMC(
        self,
        valueDate,
        stockPrice,
        discountCurve,
        dividendYield,
        volatility,
        stockMinMax,
        numPaths=10000,
        numStepsPerYear=252,
        seed=4242):
```

2.12 FinFloatLookbackOption

2.12.0.1 Enumerated Type: FinFloatLookbackOptionTypes

- FLOATING_CALL
- FLOATING_PUT

Class: FinFloatLookbackOption(FinOption)

class FinFloatLookbackOption(FinOption):

Data Members

- _expiryDate
- _optionType

Functions

__init__

optionType):

```
def __init__(self,
             expiryDate,
             optionType):
```

value

stockMinMax):

```
def value(self,
          valueDate,
          stockPrice,
          discountCurve,
          dividendYield,
          volatility,
          stockMinMax):
```

valueMC

seed=4242):

```
def valueMC(
        self,
        valueDate,
        stockPrice,
```

```
            discountCurve,
            dividendYield,
            volatility,
            stockMinMax,
            numPaths=10000,
            numStepsPerYear=252,
            seed=4242):
```

2.13 FinOption

2.13.0.1 Enumerated Type: FinOptionTypes

- EUROPEAN_CALL
- EUROPEAN_PUT
- AMERICAN_CALL
- AMERICAN_PUT
- DIGITAL_CALL
- DIGITAL_PUT
- ASIAN_CALL
- ASIAN_PUT
- COMPOUND_CALL
- COMPOUND_PUT

2.13.0.2 Enumerated Type: FinOptionModelTypes

- BLACKSCHOLES
- ANOTHER

Class: FinOption(object)

class FinOption(object):

Data Members

No data members found.

Functions

delta

model):

```
def delta(
        self,
        valueDate,
        stockPrice,
        discountCurve,
```

```
        dividendYield,
        model):
```

gamma

model):

```
def gamma(
        self,
        valueDate,
        stockPrice,
        discountCurve,
        dividendYield,
        model):
```

vega

model):

```
def vega(
        self,
        valueDate,
        stockPrice,
        discountCurve,
        dividendYield,
        model):
```

theta

model):

```
def theta(
        self,
        valueDate,
        stockPrice,
        discountCurve,
        dividendYield,
        model):
```

rho

model):

```
def rho(
        self,
        valueDate,
        stockPrice,
        discountCurve,
        dividendYield,
        model):
```

2.14 FinRainbowOption

2.14.0.1 Enumerated Type: FinRainbowOptionTypes

- CALL_ON_MAXIMUM
- PUT_ON_MAXIMUM
- CALL_ON_MINIMUM
- PUT_ON_MINIMUM
- CALL_ON_NTH
- PUT_ON_NTH

Class: FinRainbowOption(FinOption)

class FinRainbowOption(FinOption):

Data Members

- _expiryDate
- _payoffType
- _payoffParams
- _numAssets

Functions

__init__

numAssets):

```
def __init__(self,
             expiryDate,
             payoffType,
             payoffParams,
             numAssets):
```

validate

betas):

```
def validate(self,
             stockPrices,
             dividendYields,
```

```
                volatilities,
                betas):
```

validatePayoff

def validatePayoff(self, payoffType, payoffParams, numAssets):

```
def validatePayoff(self, payoffType, payoffParams, numAssets):
```

value

betas):

```
def value(self,
          valueDate,
          expiryDate,
          stockPrices,
          discountCurve,
          dividendYields,
          volatilities,
          betas):
```

valueMC

seed=4242):

```
def valueMC(self,
            valueDate,
            expiryDate,
            stockPrices,
            discountCurve,
            dividendYields,
            volatilities,
            betas,
            numPaths=10000,
            seed=4242):
```

payoffValue

def payoffValue(s, payoffTypeValue, payoffParams):

```
def payoffValue(s, payoffTypeValue, payoffParams):
```

valueMCFast

seed=4242):

```
def valueMCFast(t,
```

```
                stockPrices,
                discountCurve,
                dividendYields,
                volatilities,
                betas,
                numAssets,
                payoffType,
                payoffParams,
                numPaths=10000,
                seed=4242):
```

2.15 FinVanillaOption

Class: FinVanillaOption(FinOption)

class FinVanillaOption(FinOption):

Data Members

- _expiryDate
- _strikePrice
- _optionType

Functions

__init__

optionType):

```
def __init__(self,
             expiryDate,
             strikePrice,
             optionType):
```

value

model):

```
def value(self,
          valueDate,
          stockPrice,
          discountCurve,
          dividendYield,
          model):
```

xdelta

model):

```
def xdelta(self,
           valueDate,
           stockPrice,
           discountCurve,
           dividendYield,
           model):
```

xgamma

model):

```
def xgamma(self,
           valueDate,
           stockPrice,
           discountCurve,
           dividendYield,
           model):
```

xvega

model):

```
def xvega(self,
          valueDate,
          stockPrice,
          discountCurve,
          dividendYield,
          model):
```

xtheta

model):

```
def xtheta(self,
           valueDate,
           stockPrice,
           discountCurve,
           dividendYield,
           model):
```

impliedVolatility

price):

```
def impliedVolatility(self,
                      valueDate,
                      stockPrice,
                      discountCurve,
                      dividendYield,
                      price):
```

valueMC

seed=4242):

```
def valueMC(self,
            valueDate,
```

```
                stockPrice,
                discountCurve,
                dividendYield,
                model,
                numPaths=10000,
                seed=4242):
```

value_MC_OLD

seed=4242):

```
    def value_MC_OLD(self,
                     valueDate,
                     stockPrice,
                     discountCurve,
                     dividendYield,
                     terminalS,
                     seed=4242):
```

f

def f(volatility, *args):

```
def f(volatility, *args):
```

fvega

def fvega(volatility, *args):

```
def fvega(volatility, *args):
```

2.16 FinVarianceSwap

Class: FinVarianceSwap(object)

Data Members

- _startDate
- _maturityDate
- _strikeVariance
- _notional
- _payStrike
- _numPutOptions
- _numCallOptions
- _putWts
- _putStrikes
- _callWts
- _callStrikes
- _putWts[n]
- _callWts[n]

Functions

__init__

Create variance swap contract.

```
def __init__(self,
             startDate,
             maturityDateOrTenor,
             strikeVariance,
             notional=ONE_MILLION,
             payStrikeFlag=True):
```

value

Calculate the value of the variance swap based on the realised volatility to the valuation date, the forward looking implied volatility to the maturity date using the libor discount curve.

```
def value(self,
          valuationDate,
          realisedVar,
          fairStrikeVar,
          liborCurve):
```

fairStrikeApprox

This is an approximation of the fair strike variance by Demeterfi et al. (1999) which assumes that sigma(K) = sigma(F) - b(K-F)/F where F is the forward stock price and sigma(F) is the ATM forward vol.

```
def fairStrikeApprox(self,
                     valuationDate,
                     fwdStockPrice,
                     strikes,
                     volatilities):
```

fairStrike

Calculate the implied variance according to the volatility surface using a static replication methodology with a specially weighted portfolio of put and call options across a range of strikes using the approximate method set out by Demeterfi et al. 1999.

```
def fairStrike(self,
               valuationDate,
               stockPrice,
               dividendYield,
               volatilityCurve,
               numCallOptions,
               numPutOptions,
               strikeSpacing,
               discountCurve,
               useForward=True):
```

f

def f(x): return (2.0/tmat)*((x-sstar)/sstar-log(x/sstar))

```
    def f(x): return (2.0/tmat)*((x-sstar)/sstar-log(x/sstar))
```

realisedVariance

Calculate the realised variance according to market standard calculations which can either use log or percentage returns.

```
def realisedVariance(self, closePrices, useLogs=True):
```

print

def print(self):

```
def print(self):
```

Chapter 3

..//products//credit

3.1 Introduction

3.2 FinCDS

Class: FinCDS(object)

A class which manages Credit Default Swap. It performs schedule generation and the valuation and risk management of CDS.

Data Members

- _stepInDate
- _maturityDate
- _coupon
- _notional
- _longProtection
- _dayCountType
- _dateGenRuleType
- _calendarType
- _frequencyType
- _busDayAdjustType
- _adjustedDates
- _adjustedDates[i]
- _adjustedDates[flowNum-1]
- _accrualFactors
- _flows

Functions

__init__

stepInDate - FinDate that is the date protection starts (usually T+1) runningCoupon - Size of coupon on premium leg

```
def __init__(self,
             stepInDate,
             maturityDateOrTenor,
             runningCoupon,
```

```
                notional=ONE_MILLION,
                longProtection=True,
                frequencyType=FinFrequencyTypes.QUARTERLY,
                dayCountType=FinDayCountTypes.ACT_360,
                calendarType=FinCalendarTypes.WEEKEND,
                busDayAdjustType=FinDayAdjustTypes.FOLLOWING,
                dateGenRuleType=FinDateGenRuleTypes.BACKWARD):
```

generateAdjustedCDSPaymentDates

Generate CDS payment dates which have been holiday adjusted.

```
def generateAdjustedCDSPaymentDates(self):
```

calcFlows

Calculate cash flow amounts on premium leg.

```
def calcFlows(self):
```

value

Valuation of a CDS contract

```
def value(self,
          valuationDate,
          issuerCurve,
          contractRecovery=standardRecovery,
          pv01Method=0,
          prot_method=0,
          numStepsPerYear=25):
```

creditDV01

Valuation of a CDS contract

```
def creditDV01(self,
               valuationDate,
               issuerCurve,
               contractRecovery=standardRecovery,
               pv01Method=0,
               prot_method=0,
               numStepsPerYear=25):
```

interestDV01

Calculation of the interest DV01 based on a simple bump of the discount factors and reconstruction of the CDS curve.

```
def interestDV01(self,
                 valuationDate,
                 issuerCurve,
                 contractRecovery=standardRecovery,
                 pv01Method=0,
                 prot_method=0,
                 numStepsPerYear=25):
```

cashSettlementAmount

numStepsPerYear=25):

```
def cashSettlementAmount(self,
                         valuationDate,
                         settlementDate,
                         issuerCurve,
                         contractRecovery=standardRecovery,
                         pv01Method=0,
                         prot_method=0,
                         numStepsPerYear=25):
```

cleanPrice

numStepsPerYear=52):

```
def cleanPrice(self,
               valuationDate,
               issuerCurve,
               contractRecovery=standardRecovery,
               pv01Method=0,
               prot_method=0,
               numStepsPerYear=52):
```

riskyPV01_OLD

pv01Method=0):

```
def riskyPV01_OLD(self,
                  valuationDate,
                  issuerCurve,
                  pv01Method=0):
```

accruedDays

def accruedDays(self):

```
def accruedDays(self):
```

accruedInterest

Calculate the amount of accrued interest that has accrued from the previous coupon date (PCD) to the stepIn-Date of the CDS contract.

```
def accruedInterest(self):
```

protectionLegPV

Calculates the protection leg PV of the CDS by calling into the fast NUMBA code that has been defined above.

```
def protectionLegPV(self,
                    valuationDate,
                    issuerCurve,
                    contractRecovery=standardRecovery,
                    numStepsPerYear=25,
                    protMethod=0):
```

riskyPV01

pv01Method=0):

```
def riskyPV01(self,
              valuationDate,
              issuerCurve,
              pv01Method=0):
```

premiumLegPV

pv01Method=0):

```
def premiumLegPV(self,
                 valuationDate,
                 issuerCurve,
                 pv01Method=0):
```

parSpread

protMethod=0):

```
def parSpread(self,
              valuationDate,
              issuerCurve,
              contractRecovery=standardRecovery,
              numStepsPerYear=25,
              pv01Method=0,
              protMethod=0):
```

valueFastApprox

Implementation of fast valuation of the CDS contract using an accurate approximation that avoids curve building.

```
def valueFastApprox(self,
                    valuationDate,
                    flatContinuousInterestRate,
                    flatCDSCurveSpread,
                    curveRecovery=standardRecovery,
                    contractRecovery=standardRecovery):
```

print

print out details of the CDS contract and all of the calculated cashflows

```
def print(self, valuationDate):
```

printFlows

def printFlows(self, issuerCurve):

```
def printFlows(self, issuerCurve):
```

riskyPV01_NUMBA

Fast calculation of the risky PV01 of a CDS using NUMBA. The output is a numpy array of the full and clean risky PV01.

```
def riskyPV01_NUMBA(teff,
                    accrualFactorPCDToNow,
                    paymentTimes,
                    yearFracs,
                    npLiborTimes,
                    npLiborValues,
                    npSurvTimes,
                    npSurvValues,
                    pv01Method):
```

protectionLegPV_NUMBA

Fast calculation of the CDS protection leg PV using NUMBA to speed up the numerical integration over time.

```
def protectionLegPV_NUMBA(teff,
                          tmat,
                          npLiborTimes,
                          npLiborValues,
                          npSurvTimes,
```

```
npSurvValues,
contractRecovery,
numStepsPerYear,
protMethod):
```

3.3 FinCDSBasket

Class: FinCDSBasket(object)

Data Members

- _stepInDate
- _maturityDate
- _notional
- _coupon
- _longProtection
- _dayCountType
- _dateGenRuleType
- _calendarType
- _frequencyType
- _busDayAdjustType
- _cdsContract

Functions

__init__

dateGenRuleType=FinDateGenRuleTypes.BACKWARD):

```
def __init__(self,
             stepInDate,
             maturityDate,
             notional=ONE_MILLION,
             coupon=0.0,
             longProtection=True,
             frequencyType=FinFrequencyTypes.QUARTERLY,
             dayCountType=FinDayCountTypes.ACT_360,
             calendarType=FinCalendarTypes.WEEKEND,
             busDayAdjustType=FinDayAdjustTypes.FOLLOWING,
             dateGenRuleType=FinDateGenRuleTypes.BACKWARD):
```

valueLegs_MC

Value the legs of the default basket using Monte Carlo. The default times are an input so this valuation is not model dependent.

```
def valueLegs_MC(self,
                 valuationDate,
                 nToDefault,
                 defaultTimes,
                 issuerCurves,
                 liborCurve):
```

valueGaussian_MC

Value the default basket using a Gaussian copula model.

```
def valueGaussian_MC(self,
                     valuationDate,
                     nToDefault,
                     issuerCurves,
                     correlationMatrix,
                     liborCurve,
                     numTrials,
                     seed):
```

valueStudentT_MC

Value the default basket using the Student-T copula.

```
def valueStudentT_MC(self,
                     valuationDate,
                     nToDefault,
                     issuerCurves,
                     correlationMatrix,
                     degreesOfFreedom,
                     liborCurve,
                     numTrials,
                     seed):
```

value1FGaussian_Homo

Value default basket using 1 factor Gaussian copula and analytical approach which is only exact when all recovery rates are the same.

```
def value1FGaussian_Homo(self,
                         valuationDate,
                         nToDefault,
                         issuerCurves,
                         betaVector,
                         liborCurve,
```

```
numPoints=50):
```

3.4 FinCDSIndexOption

Class: FinCDSIndexOption(object)

Class to manage the pricing and risk management of an option to enter into a CDS index. Different pricing algorithms are presented.

Data Members

- _expiryDate
- _maturityDate
- _indexCoupon
- _strikeCoupon
- _notional
- _longProtection
- _dayCountType
- _dateGenRuleType
- _calendarType
- _frequencyType
- _businessDateAdjustType
- _cdsContract

Functions

__init__

dateGenRuleType=FinDateGenRuleTypes.BACKWARD):

```
def __init__(self,
             expiryDate,
             maturityDate,
             indexCoupon,
             strikeCoupon,
             notional=ONE_MILLION,
             longProtection=True,
             frequencyType=FinFrequencyTypes.QUARTERLY,
             dayCountType=FinDayCountTypes.ACT_360,
             calendarType=FinCalendarTypes.WEEKEND,
             busDayAdjustType=FinDayAdjustTypes.FOLLOWING,
             dateGenRuleType=FinDateGenRuleTypes.BACKWARD):
```

valueAdjustedBlack

This approach uses two adjustments to Blacks option pricing model to value an option on a CDS index.

```
def valueAdjustedBlack(self,
                       valuationDate,
                       indexCurve,
                       indexRecovery,
                       liborCurve,
                       sigma):
```

valueAnderson

This function values a CDS index option following approach by Anderson (2006). This ensures that the no-arbitrage relationship between the consituent CDS contract and the CDS index is enforced. It models the forward spread as a log-normally distributed quantity and uses the credit triangle to compute the forward RPV01.

```
def valueAnderson(self,
                  valuationDate,
                  issuerCurves,
                  indexRecovery,
                  sigma):
```

solveForX

Function to solve for the arbitrage free

```
def solveForX(self,
              valuationDate,
              sigma,
              indexCoupon,
              indexRecovery,
              liborCurve,
              expH):
```

calcObjFunc

An internal function used in the Anderson valuation.

```
def calcObjFunc(self,
                x,
                valuationDate,
                sigma,
                indexCoupon,
                indexRecovery,
                liborCurve):
```

calcIndexPayerOptionPrice

Calculates the intrinsic value of the index payer swap and the value of the index payer option which are both returned in an array.

```
def calcIndexPayerOptionPrice(self,
                              valuationDate,
                              x,
                              sigma,
                              indexCoupon,
                              strikeValue,
                              liborCurve,
                              indexRecovery):
```

3.5 FinCDSIndexPortfolio

Class: FinCDSIndexPortfolio()

This class manages the calculations associated with an equally weighted portfolio of CDS contracts with the same maturity date.

Data Members

- _dayCountType
- _dateGenRuleType
- _calendarType
- _frequencyType
- _businessDateAdjustType

Functions

__init__

Create FinCDSIndexPortfolio object. Note that all of the inputs have a default value which reflects the CDS market standard.

```
def __init__(self,
             frequencyType=FinFrequencyTypes.QUARTERLY,
             dayCountType=FinDayCountTypes.ACT_360,
             calendarType=FinCalendarTypes.WEEKEND,
             busDayAdjustType=FinDayAdjustTypes.FOLLOWING,
             dateGenRuleType=FinDateGenRuleTypes.BACKWARD):
```

intrinsicRPV01

Calculation of the risky PV01 of the CDS porfolio by taking the average of the risky PV01s of each contract.

```
def intrinsicRPV01(self,
                   valuationDate,
                   stepInDate,
                   maturityDate,
                   issuerCurves):
```

intrinsicProtectionLegPV

Calculation of the intrinsic protection leg value of the CDS porfolio by taking the average sum the protection legs of each contract.

```
def intrinsicProtectionLegPV(self,
                             valuationDate,
                             stepInDate,
                             maturityDate,
                             issuerCurves):
```

intrinsicSpread

Calculation of the intrinsic spread of the CDS portfolio as the one which would make the value of the protection legs equal to the value of the premium legs if all premium legs paid the same spread.

```
def intrinsicSpread(self,
                    valuationDate,
                    stepInDate,
                    maturityDate,
                    issuerCurves):
```

averageSpread

Calculates the average par CDS spread of the CDS portfolio.

```
def averageSpread(self,
                  valuationDate,
                  stepInDate,
                  maturityDate,
                  issuerCurves):
```

totalSpread

Calculates the total CDS spread of the CDS portfolio by summing over all of the issuers and adding the spread with no weights.

```
def totalSpread(self,
                valuationDate,
                stepInDate,
                maturityDate,
                issuerCurves):
```

minSpread

Calculates the minimum par CDS spread across all of the issuers in the CDS portfolio.

```
def minSpread(self,
              valuationDate,
              stepInDate,
              maturityDate,
              issuerCurves):
```

maxSpread

Calculates the maximum par CDS spread across all of the issuers in the CDS portfolio.

```
def maxSpread(self,
              valuationDate,
              stepInDate,
              maturityDate,
              issuerCurves):
```

spreadAdjustIntrinsic

Adjust individual CDS curves to reprice CDS index prices. This approach uses an iterative scheme but is slow as it has to use a CDS curve bootstrap required when each trial spread adjustment is made.

```
def spreadAdjustIntrinsic(valuationDate,
                          issuerCurves,
                          indexCoupons,
                          indexUpfronts,
                          indexMaturityDates,
                          indexRecoveryRate,
                          tolerance):
```

hazardRateAdjustIntrinsic

Adjust individual CDS curves to reprice CDS index prices. This approach adjusts the hazard rates and so avoids the slowish CDS curve bootstrap required when a spread adjustment is made.

```
def hazardRateAdjustIntrinsic(valuationDate,
                              issuerCurves,
                              indexCoupons,
                              indexUpfronts,
                              indexMaturityDates,
                              indexRecoveryRate,
                              tolerance,
                              maxIterations=100):
```

3.6 FinCDSOption

Class: FinCDSOption()

Data Members

- _expiryDate
- _maturityDate
- _strikeCoupon
- _longProtection
- _knockoutFlag
- _notional
- _frequencyType
- _dayCountType
- _calendarType
- _businessDateAdjustType
- _dateGenRuleType

Functions

__init__

dateGenRuleType=FinDateGenRuleTypes.BACKWARD):

```
def __init__(self,
             expiryDate,
             maturityDate,
             strikeCoupon,
             notional=ONE_MILLION,
             longProtection=True,
             knockoutFlag=True,
             frequencyType=FinFrequencyTypes.QUARTERLY,
             dayCountType=FinDayCountTypes.ACT_360,
             calendarType=FinCalendarTypes.WEEKEND,
             busDayAdjustType=FinDayAdjustTypes.FOLLOWING,
             dateGenRuleType=FinDateGenRuleTypes.BACKWARD):
```

value

Value the CDS option using Blacks model with an adjustment for any Front End Protection. TODO - Should the CDS be created in the init method ?

```
def value(self,
          valuationDate,
          issuerCurve,
          volatility):
```

impliedVolatility

Calculate the implied CDS option volatility from a price.

```
def impliedVolatility(self,
                      valuationDate,
                      issuerCurve,
                      optionValue):
```

fvol

Root searching function in the calculation of the CDS implied volatility.

```
def fvol(volatility, *args):
```

3.7 FinCDSTranche

3.7.0.1 Enumerated Type: FinLossDistributionBuilder

- RECURSION
- ADJUSTED_BINOMIAL
- GAUSSIAN
- LHP

Class: FinCDSTranche(object)

class FinCDSTranche(object):

Data Members

- _k1
- _k2
- _stepInDate
- _maturityDate
- _notional
- _coupon
- _longProtection
- _dayCountType
- _dateGenRuleType
- _calendarType
- _frequencyType
- _busDayAdjustType
- _cdsContract

Functions

__init__

dateGenRuleType=FinDateGenRuleTypes.BACKWARD):

```
def __init__(self,
             stepInDate,
             maturityDate,
             k1,
             k2,
             notional=ONE_MILLION,
             coupon=0.0,
             longProtection=True,
             frequencyType=FinFrequencyTypes.QUARTERLY,
             dayCountType=FinDayCountTypes.ACT_360,
             calendarType=FinCalendarTypes.WEEKEND,
             busDayAdjustType=FinDayAdjustTypes.FOLLOWING,
             dateGenRuleType=FinDateGenRuleTypes.BACKWARD):
```

valueBC

model=FinLossDistributionBuilder.RECURSION):

```
def valueBC(self,
            valuationDate,
            issuerCurves,
            upfront,
            coupon,
            corr1,
            corr2,
            numPoints=50,
            model=FinLossDistributionBuilder.RECURSION):
```

Chapter 4

..//products//bonds

4.1 Introduction

This folder contains a suite of bond-related functionality across a set of files and classes.

- FinAnnuity is a stream of cashflows that is generated and can be priced.
- FinBond is a basic fixed coupon bond with all of the associated duration and convexity measures. It also includes some common spread measures such as the asset swap spread and the option adjusted spread.
- FinBondCallable is a bond that has an embedded call and put option. A number of rate models pricing functions have been included to allow such bonds to be priced and risk-managed.
- FinBondFuture is a bond future that has functionality around determination of the conversion factor and calculation of the invoice price and determination of the cheapest to deliver.
- FinBondMarket is a database of country-specific bond market conventions that can be referenced. These include settlement days and accrued interest conventions.
- FinBondOption is a bond option class that includes a number of valuation models for pricing both European and American style bond options. Models for European options include a Lognormal Price, Hull-White (HW) and Black-Karasinski (BK). The HW valuation is fast as it uses Jamshidians decomposition trick. American options can also be priced using a HW and BK trinomial tree. The details are abstracted away making it easy to use.
- FinConvertibleBond enables the pricing and risk-management of convertible bonds. The model is a binomial tree implementation of Black-Scholes which allows for discrete dividends, embedded puts and calls, and a delayed start of the conversion option.
- FinFloatingNote enables the pricing and risk-management of a bond with floating rate coupons. Discount margin calculations are provided.
- FinMortgage generates the periodic cashflows for an interest-only and a repayment mortgage.

4.2 FinAnnuity

Class: FinAnnuity(object)

An annuity is a vector of dates and flows generated according to ISDA standard rules which starts on the next date after the start date (effective date) and runs up to an end date. Dates are then adjusted according to a specified calendar.

Data Members

- _startDate
- _endDate
- _frequencyType
- _calendarType
- _busDayAdjustType
- _dateGenRuleType
- _dayCountConventionType
- _schedule
- _flows
- _yearFractions

Functions

__init__

dayCountConventionType=FinDayCountTypes.ACT_360):

```
def __init__(self,
             startDate,
             endDate,
             frequencyType=FinFrequencyTypes.ANNUAL,
             calendarType=FinCalendarTypes.WEEKEND,
             busDayAdjustType=FinDayAdjustTypes.FOLLOWING,
             dateGenRuleType=FinDateGenRuleTypes.BACKWARD,
             dayCountConventionType=FinDayCountTypes.ACT_360):
```

generate

def generate(self, startDate):

```
def generate(self, startDate):
```

dump

def dump(self):

```
def dump(self):
```

4.3 FinBond

4.3.0.1 Enumerated Type: FinYieldConventions

- UK_DMO
- US_STREET
- US_TREASURY

Class: FinBond(object)

Class for fixed coupon bonds and performing related analytics. These are bullet bonds which means they have regular coupon payments of a known size that are paid on known dates plus a payment of par at maturity.

Data Members

- _maturityDate
- _coupon
- _frequencyType
- _accrualType
- _flowDates
- _frequency
- _face
- _settlementDate
- _accrued
- _accruedDays
- _alpha
- _pcd
- _ncd

Functions

__init__

Create FinBond object by providing Maturity Date, Frequency, coupon and the accrual convention type.

```
def __init__(self,
             maturityDate,
             coupon,
             frequencyType,
             accrualType,
             face=100.0):
```

calculateFlowDates

Determine the bond cashflow payment dates.

```
def calculateFlowDates(self, settlementDate):
```

fullPriceFromYield

Calculate the full price of bond from its yield to maturity. This function is vectorised with respect to the yield input.

```
def fullPriceFromYield(self, settlementDate, y,
                       convention=FinYieldConventions.UK_DMO):
```

dollarDuration

Calculate the risk or dP/dy of the bond by bumping.

```
def dollarDuration(self, settlementDate, ytm,
                   convention=FinYieldConventions.UK_DMO):
```

macauleyDuration

Calculate the Macauley duration of the bond on a settlement date given its yield to maturity.

```
def macauleyDuration(self, settlementDate, ytm,
                     convention=FinYieldConventions.UK_DMO):
```

modifiedDuration

Calculate the modified duration of the bondon a settlement date given its yield to maturity.

```
def modifiedDuration(self, settlementDate, ytm,
                     convention=FinYieldConventions.UK_DMO):
```

convexityFromYield

Calculate the bond convexity from the yield to maturity. This function is vectorised with respect to the yield input.

```
def convexityFromYield(self, settlementDate, ytm,
                       convention=FinYieldConventions.UK_DMO):
```

cleanPriceFromYield

Calculate the bond clean price from the yield to maturity. This function is vectorised with respect to the yield input.

```
def cleanPriceFromYield(self, settlementDate, ytm,
                        convention=FinYieldConventions.UK_DMO):
```

cleanPriceFromDiscountCurve

Calculate the bond price using some discount curve to present-value the bonds cashflows.

```
def cleanPriceFromDiscountCurve(self, settlementDate, discountCurve):
```

fullPriceFromDiscountCurve

Calculate the bond price using some discount curve to present-value the bonds cashflows.

```
def fullPriceFromDiscountCurve(self, settlementDate, discountCurve):
```

currentYield

Calculate the current yield of the bond which is the coupon divided by the clean price (not the full price)

```
def currentYield(self, cleanPrice):
```

yieldToMaturity

Calculate the bonds yield to maturity by solving the price yield relationship using a one-dimensional root solver.

```
def yieldToMaturity(self,
                    settlementDate,
                    cleanPrice,
                    convention=FinYieldConventions.US_TREASURY):
```

_accruedInterest

Calculate the amount of coupon that has accrued between the previous coupon date and the settlement date.

```
def _accruedInterest(self, settlementDate):
```

assetSwapSpread

Calculate the par asset swap spread of the bond. The discount curve is a Libor curve that is passed in. This function is vectorised with respect to the clean price.

```
def assetSwapSpread(
        self,
        settlementDate,
        cleanPrice,
        discountCurve,
        swapFloatDayCountConventionType=FinDayCountTypes.ACT_360,
        swapFloatFrequencyType=FinFrequencyTypes.SEMI_ANNUAL,
        swapFloatCalendarType=FinCalendarTypes.WEEKEND,
        swapFloatBusDayAdjustRuleType=FinDayAdjustTypes.FOLLOWING,
        swapFloatDateGenRuleType=FinDateGenRuleTypes.BACKWARD):
```

fullPriceFromOAS

Calculate the full price of the bond from its OAS given the bond settlement date, a discount curve and the oas as a number.

```
def fullPriceFromOAS(self,
                     settlementDate,
                     discountCurve,
                     oas):
```

optionAdjustedSpread

Return OAS for bullet bond given settlement date, clean bond price and the discount relative to which the spread is to be computed.

```
def optionAdjustedSpread(self,
                         settlementDate,
                         cleanPrice,
                         discountCurve):
```

printFlows

Print a list of the unadjusted coupon payment dates used in analytic calculations for the bond.

```
def printFlows(self, settlementDate):
```

priceFromSurvivalCurve

Calculate discounted present value of flows assuming default model. This has not been completed.

```
def priceFromSurvivalCurve(self,
                           discountCurve,
                           survivalCurve,
                           recoveryRate):
```

print

Print a list of the unadjusted coupon payment dates used in analytic calculations for the bond.

```
def print(self):
```

f

Function used to do root search in price to yield calculation.

```
def f(y, *args):
```

g

Function used to do root search in price to OAS calculation.

```
def g(oas, *args):
```

4.4 FinBondCallable

4.5 FinBondFuture

Class: FinBondFuture(object)

Class for managing futures contracts on government bonds that follows CME conventions and related analytics.

Data Members

- _tickerName
- _firstDeliveryDate
- _lastDeliveryDate
- _contractSize
- _coupon

Functions

__init__

coupon):

```
def __init__(self,
             tickerName,
             firstDeliveryDate,
             lastDeliveryDate,
             contractSize,
             coupon):
```

conversionFactor

Determine the conversion factor for a specific bond using CME convention. To do this we need to know the contract standard coupon and must round the bond maturity (starting its life on the first delivery date) to the nearest 3 month multiple and then calculate the bond clean price.

```
def conversionFactor(self, bond):
```

principalInvoicePrice

```
def principalInvoicePrice(self,
                          bond,
                          futuresPrice):
```

totalInvoiceAmount

futuresPrice):

```
def totalInvoiceAmount(self,
                       settlementDate,
                       bond,
                       futuresPrice):
```

cheapestToDeliver

Determination of CTD as deliverable bond with lowest cost to buy versus what is received when the bond is delivered.

```
def cheapestToDeliver(self,
                      bonds,
                      bondCleanPrices,
                      futuresPrice):
```

deliveryGainLoss

Determination of what is received when the bond is delivered.

```
def deliveryGainLoss(self,
                     bond,
                     bondCleanPrice,
                     futuresPrice):
```

4.6 FinBondMarket

4.6.0.1 Enumerated Type: FinBondMarkets

- AUSTRIA
- BELGIUM
- CYPRUS
- ESTONIA
- FINLAND
- FRANCE
- GERMANY
- GREECE
- IRELAND
- ITALY
- LATVIA
- LITHUANIA
- LUXEMBOURG
- MALTA
- NETHERLANDS
- PORTUGAL
- SLOVAKIA
- SLOVENIA
- SPAIN
- ESM
- EFSF
- BULGARIA
- CROATIA
- CZECH_REPUBLIC

- DENMARK
- HUNGARY
- POLAND
- ROMANIA
- SWEDEN
- JAPAN
- SWITZERLAND
- UNITED_KINGDOM
- UNITED_STATES

getTreasuryBondMarketConventions

Returns the day count convention for accrued interest, the frequency and the number of days from trade date to settlement date. This is for Treasury markets. And for secondary bond markets.

```
def getTreasuryBondMarketConventions(country):
```

4.7 FinBondOption

4.7.0.1 Enumerated Type: FinBondModelTypes

- BLACK
- HO_LEE
- HULL_WHITE
- BLACK_KARASINSKI

4.7.0.2 Enumerated Type: FinBondOptionTypes

- EUROPEAN_CALL
- EUROPEAN_PUT
- AMERICAN_CALL
- AMERICAN_PUT

Class: FinBondOption()

Data Members

- _expiryDate
- _strikePrice
- _bond
- _optionType
- _face

Functions

__init__

optionType):

```
def __init__(self,
             bond,
             expiryDate,
             strikePrice,
             face,
             optionType):
```

value

Value the bond option using the specified model.

```
def value(self,
          valueDate,
          discountCurve,
          model):
```

4.8 FinConvertibleBond

Class: FinConvertibleBond(object)

Class for convertible bonds. These bonds embed rights to call and put the bond in return for equity. Until then they are bullet bonds which means they have regular coupon payments of a known size that are paid on known dates plus a payment of par at maturity. As the options are price based, the decision to convert to equity depends on the stock price, the credit quality of the issuer and the level of interest rates.

Data Members

- _maturityDate
- _coupon
- _accrualType
- _frequency
- _frequencyType
- _callDates
- _callPrices
- _putDates
- _putPrices
- _startConvertDate
- _conversionRatio
- _face
- _settlementDate
- _flowDates
- _pcd
- _ncd
- _accrued
- _alpha
- _accruedDays

Functions

__init__

Create FinBond object by providing Maturity Date, Frequency, coupon and the accrual convention type.

```
def __init__(self,
             maturityDate,  # bond maturity date
             coupon,  # annual coupon
             frequencyType,  # coupon frequency type
             startConvertDate,  # date after which conversion is possible
             conversionRatio,  # number of shares per face of notional
             callDates,  # list of call dates
             callPrices,  # list of call prices
             putDates,  # list of put dates
             putPrices,  # list of put prices
             accrualType,  # day count type for accrued interest
             face=100.0  # face amount
             ):
```

calculateFlowDates

Determine the bond cashflow payment dates.

```
def calculateFlowDates(self, settlementDate):
```

value

A binomial tree valuation model for a convertible bond that captures the embedded equity option due to the existence of a conversion option which can be invoked after a specific date. The model allows the user to enter a schedule of dividend payment dates but the size of the payments must be in yield terms i.e. a known percentage of currently unknown future stock price is paid. Not a fixed amount. A fixed yield. Following this payment the stock is assumed to drop by the size of the dividend payment. The model also captures the stock dependent credit risk of the cash flows in which the bond price can default at any time with a hazard rate implied by the credit spread and an associated recovery rate. This is the model proposed by Hull (OFODS 6th edition,.page 522). The model captures both the issuers call schedule which is assumed to apply on a list of dates provided by the user, along with a call price. It also captures the embedded owners put schedule of prices.

```
def value(self,
          settlementDate,
          stockPrice,
          stockVolatility,
          dividendDates,
          dividendYields,
          discountCurve,
          creditSpread,
          recoveryRate = 0.40,
          numStepsPerYear = 100):
```

accruedDays

Calculate number days from previous coupon date to settlement.

```
def accruedDays(self, settlementDate):
```

_accruedInterest

Calculate the amount of coupon that has accrued between the previous coupon date and the settlement date.

```
def _accruedInterest(self, settlementDate):
```

currentYield

Calculate the current yield of the bond which is the coupon divided by the clean price (not the full price)

```
def currentYield(self, cleanPrice):
```

print

Print a list of the unadjusted coupon payment dates used in analytic calculations for the bond.

```
def print(self):
```

valueConvertible

numStepsPerYear):

```
def valueConvertible(tmat,
                     face,
                     couponTimes,
                     couponFlows,
                     callTimes,
                     callPrices,
                     putTimes,
                     putPrices,
                     convRatio,
                     startConvertTime,
                     # Market inputs
                     stockPrice,
                     dfTimes,
                     dfValues,
                     dividendTimes,
                     dividendYields,
                     stockVolatility,
                     creditSpread,
                     recRate,
                     # Tree details
                     numStepsPerYear):
```

printTree

def printTree(array):

```
def printTree(array):
```

4.9 FinFloatingRateNote

Class: FinFloatingRateNote(object)

Class for managing floating rate notes that pay a floating index plus a quoted margin.

Data Members

- _maturityDate
- _quotedMargin
- _frequencyType
- _accrualType
- _flowDates
- _frequency
- _face
- _redemption
- _settlementDate

Functions

__init__

Create FinFloatingRateNote object.

```
def __init__(self,
             maturityDate,
             quotedMargin,
             frequencyType,
             accrualType,
             face=100.0,
             redemption=1.0):
```

calculateFlowDates

Determine the bond cashflow payment dates.

```
def calculateFlowDates(self, settlementDate):
```

fullPriceFromDiscountMargin

Calculate the full price of the bond from its discount margin and # making assumptions about the future Libor rates.

```
def fullPriceFromDiscountMargin(self,
                                settlementDate,
                                nextCoupon,
                                futureLibor,
                                dm):
```

dollarDuration

Calculate the risk or dP/dy of the bond by bumping.

```
def dollarDuration(self,
                   settlementDate,
                   nextCoupon,
                   futureLibor,
                   dm):
```

macauleyDuration

Calculate the Macauley duration of the bond on a settlement date given its yield to maturity.

```
def macauleyDuration(self,
                     settlementDate,
                     nextCoupon,
                     futureLibor,
                     dm):
```

modifiedDuration

Calculate the modified duration of the bondon a settlement date given its yield to maturity.

```
def modifiedDuration(self,
                     settlementDate,
                     nextCoupon,
                     futureLibor,
                     dm):
```

convexityFromDiscountMargin

Calculate the bond convexity from the yield to maturity.

```
def convexityFromDiscountMargin(self,
                                settlementDate,
                                nextCoupon,
                                futureLibor,
                                dm):
```

cleanPriceFromDiscountMargin

Calculate the bond clean price from the yield.

```
def cleanPriceFromDiscountMargin(self,
                                 settlementDate,
                                 nextCoupon,
                                 futureLibor,
                                 dm):
```

fullPriceFromDiscountCurve

Calculate the bond price using some discount curve to present-value the bonds cashflows. THIS IS NOT COMPLETE.

```
def fullPriceFromDiscountCurve(self,
                               settlementDate,
                               indexCurve,
                               discountCurve):
```

discountMargin

Calculate the bonds yield to maturity by solving the price yield relationship using a one-dimensional root solver.

```
def discountMargin(self,
                   settlementDate,
                   nextCoupon,
                   futureLibor,
                   cleanPrice):
```

accruedDays

Calculate number of days from previous coupon date to settlement.

```
def accruedDays(self, settlementDate):
```

pcd

Determine the previous coupon date before the settlement date.

```
def pcd(self, settlementDate):
```

accruedInterest

Calculate the amount of coupon that has accrued between the previous coupon date and the settlement date.

```
    def accruedInterest(self,
                        settlementDate,
                        nextCoupon):
```

f

Function used to do solve root search in discount margin calculation.

```
def f(dm, *args):
```

4.10 FinMortgage

4.10.0.1 Enumerated Type: FinMortgageType

- REPAYMENT
- INTEREST_ONLY

Class: FinMortgage(object)

A mortgage is a vector of dates and flows generated in order to repay a fixed amount given a known interest rate. Payments are all the same amount but with a varying mixture of interest and repayment of principal.

Data Members

- _startDate
- _endDate
- _principal
- _frequencyType
- _calendarType
- _busDayAdjustType
- _dateGenRuleType
- _dayCountConventionType
- _schedule
- _mortgageType
- _interestFlows
- _principalFlows
- _principalRemaining
- _totalFlows

Functions

__init__

dayCountConventionType=FinDayCountTypes.ACT_360):

```
def __init__(self,
             startDate,
             endDate,
             principal,
             frequencyType=FinFrequencyTypes.MONTHLY,
             calendarType=FinCalendarTypes.WEEKEND,
             busDayAdjustType=FinDayAdjustTypes.FOLLOWING,
             dateGenRuleType=FinDateGenRuleTypes.BACKWARD,
             dayCountConventionType=FinDayCountTypes.ACT_360):
```

repaymentAmount

def repaymentAmount(self, zeroRate):

```
def repaymentAmount(self, zeroRate):
```

generateFlows

def generateFlows(self, zeroRate, mortgageType):

```
def generateFlows(self, zeroRate, mortgageType):
```

print

def print(self):

```
def print(self):
```

Chapter 5

..//products//libor

5.1 Introduction

5.2 FinInterestRateFuture

Class: FinInterestRateFuture(object)

Data Members

- _lastTradingDate
- _dayCountType
- _contractSize
- _lastSettlementDate
- _endOfInterestRatePeriod

Functions

__init__

Create an interest rate futures contract.

```
def __init__(self,
             lastTradingDate,
             dayCountType,
             contractSize):
```

futuresRate

Calculate implied futures rate from the futures price.

```
def futuresRate(self,
                settlementDate,
                futuresPrice):
```

convexity

Calculation of the convexity adjustment between FRAs and interest rate futures using the Hull-White model as described in technical note.

```
def convexity(self,
              settlementDate,
              volatility,
              a):
```

print

def print(self):

```
def print(self):
```

5.3 FinLiborCapFloor

5.3.0.1 Enumerated Type: FinLiborCapFloorType

- CAP
- FLOOR

5.3.0.2 Enumerated Type: FinLiborCapFloorModelTypes

- BLACK
- SHIFTED_BLACK
- SABR

Class: FinLiborCapFloor()

class FinLiborCapFloor():

Data Members

- _startDate
- _maturityDate
- _optionType
- _strikeRate
- _lastFixing
- _frequencyType
- _dayCountType
- _notional
- _calendarType
- _busDayAdjustType
- _dateGenRuleType
- _capFloorDates

Functions

__init__

dateGenRuleType=FinDateGenRuleTypes.BACKWARD):

```
def __init__(self,
             startDate,
             maturityDate,
             optionType,
             strikeRate,
             lastFixing=None,
             frequencyType=FinFrequencyTypes.QUARTERLY,
             dayCountType=FinDayCountTypes.THIRTY_E_360_ISDA,
             notional=ONE_MILLION,
             calendarType=FinCalendarTypes.WEEKEND,
             busDayAdjustType=FinDayAdjustTypes.FOLLOWING,
             dateGenRuleType=FinDateGenRuleTypes.BACKWARD):
```

value

model):

```
def value(self,
          valuationDate,
          liborCurve,
          model):
```

valueCapletFloorlet

model):

```
def valueCapletFloorlet(self,
                        valuationDate,
                        startDate,
                        endDate,
                        liborCurve,
                        model):
```

print

def print(self):

```
def print(self):
```

5.4 FinLiborDeposit

Class: FinLiborDeposit(object)

class FinLiborDeposit(object):

Data Members

- _calendarType
- _settlementDate
- _dayCountType
- _depositRate
- _notional
- _maturityDate

Functions

__init__

Create a Libor deposit object.

```
def __init__(self,
             settlementDate,
             maturityDateOrTenor,
             depositRate,
             dayCountType,
             notional=ONE_MILLION,
             calendarType=FinCalendarTypes.WEEKEND,
             busDayAdjustType=FinDayAdjustTypes.MODIFIED_FOLLOWING):
```

maturityDf

Returns the maturity date discount factor that would allow the Libor curve to reprice the contractual market deposit rate. Note that this is a forward discount factor that starts on settlement date.

```
def maturityDf(self):
```

value

Determine the value of the Deposit given a Libor curve.

```
def value(self, valueDate, liborCurve):
```

print

Print the contractual details of the Libor deposit.

```
def print(self):
```

5.5 FinLiborFRA

Class: FinLiborFRA(object)

Class for managing LIBOR forward rate agreements. A forward rate agreement is an agreement to exchange a fixed pre-agreed rate for a floating rate linked to LIBOR that is not known until some specified future fixing date. The FRA payment occurs on or soon after this date on the FRA settlement date. Typically the timing gap is two days. A FRA is used to hedge a Libor quality loan or lend of some agreed notional amount. This period starts on the settlement date of the FRA and ends on the maturity date of the FRA. For example a 1x4 FRA relates to a Libor starting in 1 month for a loan period ending in 4 months. Hence it linkes to 3-month Libor rate. The amount received by a payer of fixed rate at settlement is acc(1,2) * (Libor(1,2) - FRA RATE) / (1 + acc(0,1) x Libor(0,1)) So the value at time 0 is acc(1,2) * (FWD Libor(1,2) - FRA RATE) x df(0,2) If the base date of the curve is before the value date then we forward adjust this amount to that value date. For simplicity I have assumed that the fixing date and the settlement date are the same date. This should be amended later.

Data Members

- _calendarType
- _settlementDate
- _maturityDate
- _fraRate
- _payFixedRate
- _dayCountType
- _notional

Functions

__init__

Create FRA object.

```
def __init__(self,
             settlementDate,  # The date on which the floating rate fixes
             maturityDateOrTenor,  # The end of the Libor rate period
             fraRate,  # The fixed contractual FRA rate
             payFixedRate,  # True if the FRA rate is being paid
             dayCountType,  # For interest period
             notional=ONE_MILLION,
             calendarType=FinCalendarTypes.WEEKEND,
             busDayAdjustType=FinDayAdjustTypes.MODIFIED_FOLLOWING):
```

value

Determine mark to market value of a FRA contract based on the market FRA rate. The same curve is used for calculating the forward Libor and for doing discounting on the expected forward payment.

```
def value(self, valueDate, liborCurve):
```

maturityDf

Determine the maturity date discount factor needed to refit the FRA given the libor curve anbd the contract FRA rate.

```
def maturityDf(self, liborCurve):
```

print

def print(self):

```
def print(self):
```

5.6 FinLiborModelTypes

Class: FinLiborModel(object)

Data Members

- _parentType
- _volatility
- _implementation

Functions

__init__

def __init__(self):

```
def __init__(self):
```

Class: FinLiborModelBlack(FinLiborModel)

class FinLiborModelBlack(FinLiborModel):

Data Members

- _parentType
- _volatility
- _implementation

Functions

__init__

def __init__(self, volatility):

```
def __init__(self, volatility):
```

Class: FinLiborModelShiftedBlack(FinLiborModel)

class FinLiborModelShiftedBlack(FinLiborModel):

Data Members

- _parentType
- _volatility
- _shift
- _implementation

Functions

__init__

def __init__(self, volatility, shift):

```
def __init__(self, volatility, shift):
```

Class: FinLiborModelSABR(FinLiborModel)

class FinLiborModelSABR(FinLiborModel):

Data Members

- _parentType
- _alpha
- _beta
- _rho
- _nu

Functions

__init__

def __init__(self, alpha, beta, rho, nu):

```
def __init__(self, alpha, beta, rho, nu):
```

5.7 FinLiborProducts

Class: FinLiborSwap(object)

class FinLiborSwap(object):

Data Members

- payFixedLeg
- fixedLeg
- floatLeg
- payFixedFlag

Functions

__init__

dateGenRule="BACKWARD"):

```
def __init__(self, startDate, endDate,
             fixedCoupon, fixedFreq, fixedBasis,
             floatSpread, floatFreq, floatBasis,
             firstFixing=None,
             payFixedFlag=True,
             calendarName="WEEKEND",
             businessDateAdjust="FOLLOWING",
             dateGenRule="BACKWARD"):
```

value

def value(self, valueDate, discountCurve, indexCurve):

```
def value(self, valueDate, discountCurve, indexCurve):
```

dump

def dump(self):

```
def dump(self):
```

Class: FinLiborSwapFixedLeg(object)

class FinLiborSwapFixedLeg(object):

Data Members

- startDate
- maturityDate
- coupon
- freq
- basis
- flows
- schedule
- flows[numFlows-1].amount+

Functions

__init__

dateGenRule="BACKWARD"):

```
def __init__(self,
             startDate,
             maturityDate,
             coupon,
             freq,
             basis,
             calendarName="WEEKEND",
             businessDateAdjust="MODIFIED_FOLLOWING",
             dateGenRule="BACKWARD"):
```

value

def value(self, valueDate, discountCurve):

```
def value(self, valueDate, discountCurve):
```

generateFlows

def generateFlows(self, fixedBasis):

```
def generateFlows(self, fixedBasis):
```

dump

def dump(self):

```
def dump(self):
```

Class: FinLiborSwapFloatLeg(object)

class FinLiborSwapFloatLeg(object):

Data Members

- startDate
- endDate
- floatSpread
- freq
- basis
- firstFixing
- schedule
- flows
- flows[numFlows-1].amount+

Functions

__init__

dateGenRule):

```
def __init__(self,
             startDate,
             endDate,
             floatSpread,
             floatFreq,
             floatBasis,
             firstFixing,
             calendarName,
             businessDateAdjust,
             dateGenRule):
```

value

def value(self, valueDate, discountCurve, indexCurve):

```
def value(self, valueDate, discountCurve, indexCurve):
```

generateFlows

def generateFlows(self, indexCurve):

```
def generateFlows(self, indexCurve):
```

dump

def dump(self):

```
def dump(self):
```

5.8 FinLiborSwap

Class: FinLiborSwap(object)

Data Members

- _maturityDate
- _payFixedLeg
- _notional
- _startDate
- _fixedCoupon
- _floatSpread
- _fixedFrequencyType
- _floatFrequencyType
- _fixedDayCountType
- _floatDayCountType
- _payFixedFlag
- _calendarType
- _busDayAdjustType
- _dateGenRuleType
- _adjustedFixedDates
- _adjustedFloatDates
- _fixedYearFracs
- _fixedFlows
- _fixedDfs
- _fixedFlowPVs
- _fixedStartIndex
- _fixedFlowPVs[-1]+
- _fixedFlows[-1]+

- _floatYearFracs
- _floatFlows
- _floatFlowPVs
- _floatDfs
- _floatStartIndex
- _floatFlows[-1]+
- _floatFlowPVs[-1]+

Functions

__init__

Create an interest rate swap contract.

```
def __init__(self,
             startDate,
             maturityDateOrTenor,
             fixedCoupon,
             fixedFreqType,
             fixedDayCountType,
             notional=ONE_MILLION,
             floatSpread=0.0,
             floatFreqType=FinFrequencyTypes.QUARTERLY,
             floatDayCountType=FinDayCountTypes.THIRTY_360,
             payFixedFlag=True,
             calendarType=FinCalendarTypes.WEEKEND,
             busDayAdjustType=FinDayAdjustTypes.FOLLOWING,
             dateGenRuleType=FinDateGenRuleTypes.BACKWARD):
```

value

Value the interest rate swap on a value date given a single Libor discount curve.

```
def value(self,
          valuationDate,
          discountCurve,
          indexCurve,
          firstFixingRate,
          principal=0.0):
```

generateFixedLegPaymentDates

Generate the fixed leg payment dates all the way back to the start date of the swap which may precede the valuation date

```
def generateFixedLegPaymentDates(self):
```

generateFloatLegPaymentDates

Generate the floating leg payment dates all the way back to the start date of the swap which may precede the valuation date

```
def generateFloatLegPaymentDates(self):
```

pv01

Calculate the value of 1 basis point coupon on the fixed leg.

```
def pv01(self, valuationDate, discountCurve):
```

parCoupon

Calculate the fixed leg coupon that makes the swap worth zero. If the valuation date is before the swap payments start then this is the forward swap rate as it starts in the future. The swap rate is then a forward swap rate and so we use a forward discount factor. If the swap fixed leg has begun then we have a spot starting swap.

```
def parCoupon(self, valuationDate, discountCurve):
```

fixedLegValue

The swap may have started in the past but we can only value payments that have occurred after the valuation date.

```
def fixedLegValue(self, valuationDate, discountCurve, principal=0.0):
```

floatLegValue

Value the floating leg with payments from an index curve and discounting based on a supplied discount curve.

```
def floatLegValue(self,
                  valuationDate,
                  discountCurve,
                  indexCurve,
                  firstFixingRate=None,
                  principal=0.0):
```

printFixedLeg

Prints the fixed leg amounts.

```
def printFixedLeg(self, valuationDate):
```

printFloatLeg

Prints the floating leg amounts.

```
def printFloatLeg(self, valuationDate):
```

5.9 FinLiborSwaption

5.9.0.1 Enumerated Type: FinLiborSwaptionType

- PAYER
- RECEIVER

5.9.0.2 Enumerated Type: FinLiborSwaptionModelTypes

- BLACK
- SABR

Class: FinLiborSwaption()

class FinLiborSwaption():

Data Members

- _exerciseDate
- _maturityDate
- _swaptionType
- _swapFixedCoupon
- _swapFixedFrequencyType
- _swapFixedDayCountType
- _swapNotional
- _calendarType
- _busDayAdjustType
- _dateGenRuleType
- _pv01
- _fwdSwapRate
- _forwardDf

Functions

__init__

dateGenRuleType=FinDateGenRuleTypes.BACKWARD):

```
def __init__(self,
             exerciseDate,
             swapMaturityDate,
             swaptionType,
             swapFixedCoupon,
             swapFixedFrequencyType,
             swapFixedDayCountType,
             swapNotional=ONE_MILLION,
             calendarType=FinCalendarTypes.WEEKEND,
             busDayAdjustType=FinDayAdjustTypes.FOLLOWING,
             dateGenRuleType=FinDateGenRuleTypes.BACKWARD):
```

value

model):

```
def value(self,
          valuationDate,
          liborCurve,
          model):
```

print

def print(self):

```
def print(self):
```

5.10 FinOIS

Class: FinOIS(object)

Class for managing overnight index swaps. This is a swap contract in which a fixed payment leg is exchanged for a floating coupon leg. There is no exchange of par. The contract lasts from a start date to a specified maturity date. The fixed coupon is the OIS fixed rate which is set at contract initiation. The floating rate is not known until the end of each payment period. It is calculated at the end of the period as it is based on daily observations of the overnight index rate which are compounded according to a specific convention. Hence the OIS floating rate is determined by the history of the OIS rates. In its simplest form, there is just one fixed rate payment and one floating rate payment at contract maturity. However when the contract becomes longer than one year the floating and fixed payments become periodic. The value of the contract is the NPV of the two coupon streams. Discounting is done on a supplied OIS curve which is itself implied by the term structure of market OIS rates.

Data Members

- _startDate
- _maturityDate
- _payFixedLeg
- _notional
- _fixedRate
- _fixedFrequencyType
- _floatFrequencyType
- _fixedDayCountType
- _floatDayCountType
- _calendarType
- _busDayAdjustType
- _dateGenRuleType
- _fixedFlows
- _floatFlows
- _adjustedFixedDates
- _adjustedFloatDates

Functions

__init__

Create OIS object.

```
def __init__(self,
             startDate,
             maturityDate,
             fixedRate,
             fixedFrequencyType,
             fixedDayCountType,
             floatFrequencyType=FinFrequencyTypes.ANNUAL,
             floatDayCountType=FinDayCountTypes.ACT_360,
             payFixedLeg=True,
             notional=ONE_MILLION,
             calendarType=FinCalendarTypes.WEEKEND,
             busDayAdjustType=FinDayAdjustTypes.FOLLOWING,
             dateGenRuleType=FinDateGenRuleTypes.BACKWARD):
```

generatePaymentDates

def generatePaymentDates(self, valueDate):

```
def generatePaymentDates(self, valueDate):
```

generateFixedLegFlows

def generateFixedLegFlows(self, valueDate):

```
def generateFixedLegFlows(self, valueDate):
```

generateFloatLegFlows

Generate the payment amounts on floating leg implied by index curve

```
def generateFloatLegFlows(self, valueDate, indexCurve):
```

rate

Calculate the OIS rate implied rate from the history of fixings.

```
def rate(self, oisDates, oisFixings):
```

value

Value the interest rate swap on a value date given a single Libor discount curve.

```
def value(self, valueDate, discountCurve):
```

fixedLegValue

def fixedLegValue(self, valueDate, discountCurve, principal=0.0):

```
def fixedLegValue(self, valueDate, discountCurve, principal=0.0):
```

floatLegValue

Value the floating leg with payments from an index curve and discounting based on a supplied discount curve.

```
def floatLegValue(self,
                  valueDate,
                  discountCurve,
                  indexCurve,
                  principal=0.0):
```

df

Calculate the OIS rate implied discount factor.

```
def df(self,
       oisRate,
       startDate,
       endDate):
```

print

def print(self, valueDate, indexCurve):

```
def print(self, valueDate, indexCurve):
```

Chapter 6

..//products//fx

6.1 Introduction

Chapter 7

..//models

7.1 Introduction

This folder contains a range of models used in the various derivative pricing models implemented in the product folder. These include credit models for valuing portfolio credit products such as CDS Tranches, Monte-Carlo based models of stochastics processes used to value equity, FX and interest rate derivatives, and some generic implementations of models such as a tree-based Hull White model. Because the models are useful across a range of products, it is better to factor them out of the product/asset class categorisation as it avoids any unnecessary duplication. In addition we seek to make the interface to these models rely only on fast types such as floats and integers and Numpy arrays.

- FinCIRRateModel is a short rate model where the randomness component is proportional to the square root of the short rate. This model implementation is not arbitrage-free across the term structure.
- FinVasicekRateModel is a short rate model that assumes mean-reversion and normal volatility. It has a closed form solution for bond prices. It does not have the flexibility to fit a term structure of interest rates. For that you need to use the more flexible Hull-White model.
- FinBlackKaraskinskiRateModel is a short rate model in which the log of the short rate follows a mean-reverting normal process. It refits the interest rate term structure. It is implemented as a trinomial tree and allows valuation of European and American-style rate-based options.
- FinHullWhiteRateModel is a short rate model in which the short rate follows a mean-reverting normal process. It fits the interest rate term structure. It is implemented as a trinomial tree and allows valuation of European and American-style rate-based options. It also implements Jamshidian's decomposition of the bond option for European options.
- FinSABR Model is a stochastic volatility model for forward interest rates that has a closed form approximate solution for the implied volatility. It is widely used for pricing European style interest rate options, specifically caps and floors and also swaptions.

7.2 FinBlackKarasinskiRateModel

Class: FinBlackKarasinskiRateModel()

class FinBlackKarasinskiRateModel():

Data Members

- _a
- _sigma
- _Q
- _rt
- _treeTimes
- _pu
- _pm
- _pd
- _discountCurve
- _dfTimes
- _dfValues

Functions

__init__

Constructs the Black Karasinski rate model. The speed of mean reversion a and volatility are passed in. The short rate process is given by d(log(r)) = (theta(t) - a*log(r)) * dt + sigma * dW

```
def __init__(self, a, sigma):
```

bondOption

Option that can be exercised at any time over the exercise period. Due to non-analytical bond price we need to extend tree out to bond maturity and take into account cash flows through time.

```
def bondOption(self, texp, strikePrice,
               face, couponTimes, couponFlows, americanExercise):
```

buildTree

def buildTree(self, tmat, numTimeSteps, dfTimes, dfValues):

```
    def buildTree(self, tmat, numTimeSteps, dfTimes, dfValues):
```

f

def f(alpha, nm, Q, P, dX, dt, N):

```
def f(alpha, nm, Q, P, dX, dt, N):
```

fprime

def fprime(alpha, nm, Q, P, dX, dt, N):

```
def fprime(alpha, nm, Q, P, dX, dt, N):
```

searchRoot

def searchRoot(x0, nm, Q, P, dX, dt, N):

```
def searchRoot(x0, nm, Q, P, dX, dt, N):
```

searchRootDeriv

def searchRootDeriv(x0, nm, Q, P, dX, dt, N):

```
def searchRootDeriv(x0, nm, Q, P, dX, dt, N):
```

bondOptionFast

Option that can be exercised at any time over the exercise period. Due to non-analytical bond price we need to extend tree out to bond maturity and take into account cash flows through time.

```
def bondOptionFast(texp, tmat, strikePrice,  face, couponTimes, couponFlows,
                   americanExercise, _dfTimes, _dfValues,
                   _treeTimes, _Q, _pu, _pm, _pd, _rt, _dt, _a):
```

buildTreeFast

def buildTreeFast(a, sigma, treeTimes, numTimeSteps, discountFactors):

```
def buildTreeFast(a, sigma, treeTimes, numTimeSteps, discountFactors):
```

7.3 FinCIRRateModel

7.3.0.1 Enumerated Type: FinCIRNumericalScheme

- EULER
- LOGNORMAL
- MILSTEIN
- KAHLJACKEL
- EXACT

Class: FinCIRRateModel()

class FinCIRRateModel():

Data Members

- _a
- _b
- _sigma

Functions

__init__

def __init__(self, a, b, sigma):

```
def __init__(self, a, b, sigma):
```

meanr

Mean value of a CIR process after time t

```
def meanr(r0, a, b, t):
```

variancer

Variance of a CIR process after time t

```
def variancer(r0, a, b, sigma, t):
```

zeroPrice

Price of a zero coupon bond in CIR model.

```
def zeroPrice(r0, a, b, sigma, t):
```

draw

Draw a next rate from the CIR model in Monte Carlo.

```
def draw(rt, a, b, sigma, dt):
```

ratePath_MC

Generate a path of CIR rates using a number of numerical schemes.

```
def ratePath_MC(r0, a, b, sigma, t, dt, seed, scheme):
```

zeroPrice_MC

```
def zeroPrice_MC(r0, a, b, sigma, t, dt, numPaths, seed, scheme):
```

7.4 FinGaussianCopula1FModel

lossDbnRecursionGCD

Full construction of the loss distribution of a portfolio of credits where losses have been calculate as number of units based on the GCD.

```
def lossDbnRecursionGCD(numCredits,
                        defaultProbs,
                        lossUnits,
                        betaVector,
                        numIntegrationSteps):
```

homogeneousBasketLossDbn

Calculate the loss distribution of a CDS default basket where the portfolio is equally weighted and the losses in the portfolio are homo- geneous i.e. the credits have the same recovery rates.

```
def homogeneousBasketLossDbn(survivalProbabilities,
                             recoveryRates,
                             betaVector,
                             numIntegrationSteps):
```

trSurvProbRecursion

Get the tranche survival probability of a portfolio of credits in the one-factor GC model using a full recursion calculation of the loss distribution and survival probabilities to some time horizon.

```
def trSurvProbRecursion(k1,
                        k2,
                        numCredits,
                        survivalProbabilities,
                        recoveryRates,
                        betaVector,
                        numIntegrationSteps):
```

gaussApproxTrancheLoss

def gaussApproxTrancheLoss(k1, k2, mu, sigma):

```
def gaussApproxTrancheLoss(k1, k2, mu, sigma):
```

trSurvProbGaussian

Get the approximated tranche survival probability of a portfolio of credits in the one-factor GC model using a Gaussian fit of the conditional loss distribution and survival probabilities to some time horizon. Note that the losses in this fit are allowed to be negative.

```
def trSurvProbGaussian(k1,
                       k2,
                       numCredits,
                       survivalProbabilities,
                       recoveryRates,
                       betaVector,
                       numIntegrationSteps):
```

lossDbnHeterogeneousAdjBinomial

Get the portfolio loss distribution using the adjusted binomial approximation to the conditional loss distribution.

```
def lossDbnHeterogeneousAdjBinomial(numCredits,
                                    defaultProbs,
                                    lossRatio,
                                    betaVector,
                                    numIntegrationSteps):
```

trSurvProbAdjBinomial

Get the approximated tranche survival probability of a portfolio of credits in the one-factor GC model using the adjusted binomial fit of the conditional loss distribution and survival probabilities to some time horizon. This approach is both fast and highly accurate.

```
def trSurvProbAdjBinomial(k1,
                          k2,
                          numCredits,
                          survivalProbabilities,
                          recoveryRates,
                          betaVector,
                          numIntegrationSteps):
```

7.5 FinGaussianCopulaLHPModel

trSurvProbLHP

Get the approximated tranche survival probability of a portfolio of credits in the one-factor GC model using the large portfolio limit which assumes a homogenous portfolio with an infinite number of credits. This approach is very fast but not so as accurate as the adjusted binomial.

```
def trSurvProbLHP(k1,
                  k2,
                  numCredits,
                  survivalProbabilities,
                  recoveryRates,
                  beta):
```

portfolioCDF_LHP

def portfolioCDF_LHP(k, numCredits, qvector, recoveryRates, beta, numPoints):

```
def portfolioCDF_LHP(k, numCredits, qvector, recoveryRates, beta, numPoints):
```

expMinLK

def expMinLK(k, p, r, n, beta):

```
def expMinLK(k, p, r, n, beta):
```

LHPDensity

def LHPDensity(k, p, r, beta):

```
def LHPDensity(k, p, r, beta):
```

LHPAnalyticalDensityBaseCorr

def LHPAnalyticalDensityBaseCorr(k, p, r, beta, dbeta_dk):

```
def LHPAnalyticalDensityBaseCorr(k, p, r, beta, dbeta_dk):
```

LHPAnalyticalDensity

def LHPAnalyticalDensity(k, p, r, beta):

```
def LHPAnalyticalDensity(k, p, r, beta):
```

ExpMinLK

def ExpMinLK(k, p, r, n, beta):

```
def ExpMinLK(k, p, r, n, beta):
```

probLGreaterThanK

def probLGreaterThanK(K, P, R, beta):

```
def probLGreaterThanK(K, P, R, beta):
```

7.6 FinGaussianCopulaModel

defaultTimesGC

seed):

```
def defaultTimesGC(issuerCurves,
                   correlationMatrix,
                   numTrials,
                   seed):
```

7.7 FinGBMProcess

Class: FinGBMProcess()

class FinGBMProcess():

Data Members

No data members found.

Functions

getPaths

seed):

```
def getPaths(
        self,
        numPaths,
        numTimeSteps,
        t,
        mu,
        stockPrice,
        volatility,
        seed):
```

getPathsAssets

t, mus, stockPrices, volatilities, betas, seed):

```
def getPathsAssets(self, numAssets, numPaths, numTimeSteps,
                   t, mus, stockPrices, volatilities, betas, seed):
```

getPaths

seed):

```
def getPaths(numPaths,
             numTimeSteps,
             t,
             mu,
             stockPrice,
             volatility,
             seed):
```

getPathsAssets

seed):

```
def getPathsAssets(numAssets,
                   numPaths,
                   numTimeSteps,
                   t,
                   mus,
                   stockPrices,
                   volatilities,
                   betas,
                   seed):
```

getAssets

seed):

```
def getAssets(numAssets,
              numPaths,
              t,
              mus,
              stockPrices,
              volatilities,
              betas,
              seed):
```

7.8 FinHestonModel

7.8.0.1 Enumerated Type: FinHestonNumericalScheme

- EULER
- EULERLOG
- QUADEXP

Class: FinHestonModel()

class FinHestonModel():

Data Members

- _v0
- _kappa
- _theta
- _sigma
- _rho

Functions

__init__

def __init__(self, v0, kappa, theta, sigma, rho):

```
def __init__(self, v0, kappa, theta, sigma, rho):
```

value_MC

scheme=FinHestonNumericalScheme.EULERLOG):

```
def value_MC(self,
             valueDate,
             option,
             stockPrice,
             interestRate,
             dividendYield,
             numPaths,
             numStepsPerYear,
             seed,
             scheme=FinHestonNumericalScheme.EULERLOG):
```

value_Lewis

dividendYield):

```
def value_Lewis(self,
                valueDate,
                option,
                stockPrice,
                interestRate,
                dividendYield):
```

phi

def phi(k_in,):

```
def phi(k_in,):
```

phi_transform

def phi_transform(x):

```
def phi_transform(x):
```

integrand

def integrand(k): return 2.0 * np.real(np.exp(-1j *

```
def integrand(k): return 2.0 * np.real(np.exp(-1j * \
```

value_Lewis_Rouah

dividendYield):

```
def value_Lewis_Rouah(self,
                      valueDate,
                      option,
                      stockPrice,
                      interestRate,
                      dividendYield):
```

f

def f(k_in):

```
def f(k_in):
```

value_Weber

dividendYield):

```
def value_Weber(self,
                valueDate,
                option,
                stockPrice,
                interestRate,
                dividendYield):
```

F

def F(s, b):

```
def F(s, b):
```

integrand

def integrand(u):

```
def integrand(u):
```

value_Gatheral

dividendYield):

```
def value_Gatheral(self,
                   valueDate,
                   option,
                   stockPrice,
                   interestRate,
                   dividendYield):
```

F

def F(j):

```
def F(j):
```

integrand

def integrand(u):

```
def integrand(u):
```

getPaths

scheme):

```
def getPaths(
        s0,
        r,
        q,
        v0,
        kappa,
        theta,
        sigma,
        rho,
        t,
        dt,
        numPaths,
        seed,
        scheme):
```

7.9 FinHestonProcess

7.9.0.1 Enumerated Type: FinHestonScheme

- EULER
- EULERLOG
- QUADEXP

Class: FinHestonProcess(FinProcess)

class FinHestonProcess(FinProcess):

Data Members

- _numTimeSteps

Functions

getPathsAssets

fast = FinFastNumericalApproach.NUMBA):

```
def getPathsAssets(self,
                   t,
                   mus,
                   stockPrices,
                   volatilities,
                   betas,
                   seed,
                   fast = FinFastNumericalApproach.NUMBA):
```

getPaths

def getPaths(s0,r,q,v0,kappa,theta,sigma,rho,t,dt,numPaths,seed,scheme):

```
def getPaths(s0,r,q,v0,kappa,theta,sigma,rho,t,dt,numPaths,seed,scheme):
```

7.10 FinHoLeeRateModel

Class: FinHoLeeModel()

class FinHoLeeModel():

Data Members

- _discountCurve
- _sigma

Functions

__init__

def __init__(self, discountCurve, sigma):

```
def __init__(self, discountCurve, sigma):
```

P

t2): # forward maturity t2

```
def P(self,
      r1,  # short rate at time t1
      t1,  # foward start time t1
      t2):  # forward maturity t2
```

7.11 FinHullWhiteRateModel

Class: FinHullWhiteRateModel()

class FinHullWhiteRateModel():

Data Members

- _a
- _sigma
- _Q
- _r
- _treeTimes
- _pu
- _pm
- _pd
- _discountCurve
- _treeBuilt
- _bondValues
- _callOptionValues
- _putOptionValues
- _dfTimes
- _dfValues

Functions

__init__

Constructs the Hull-White rate model. The speed of mean reversion a and volatility are passed in. The short rate process is given by dr = (theta(t) - ar) * dt + sigma * dW

```
def __init__(self, a, sigma):
```

optionOnZeroCouponBond

Price an option on a zero coupon bond using analytical solution of Hull-White model. User provides bond face and option strike and expiry date and maturity date.

```
def optionOnZeroCouponBond(self, texp, tmat, strikePrice, face,
                           dfTimes, dfValues):
```

europeanBondOption_Jamshidian

Valuation of a European bond option using the Jamshidian deconstruction of the bond into a strip of zero coupon bonds with the short rate that would make the bond option be at the money forward.

```
def europeanBondOption_Jamshidian(self, texp, strikePrice, face, cpnTimes,
                                  cpnAmounts, dfTimes, dfValues):
```

europeanBondOption_Tree

Price an option on a coupon-paying bond using tree to generate short rates at the expiry date and then to analytical solution of zero coupon bond in HW model to calculate the corresponding bond price. User provides bond object and option details.

```
def europeanBondOption_Tree(self, texp, strikePrice, face, cpnTimes,
                            cpnAmounts):
```

optionOnZeroCouponBond_Tree

Price an option on a zero coupon bond using a HW trinomial tree. The discount curve was already supplied to the tree build.

```
def optionOnZeroCouponBond_Tree(self, texp, tmat, strikePrice, face):
```

americanBondOption_Tree

```
def americanBondOption_Tree(self, texp, strikePrice, face,
                    couponTimes, couponAmounts, americanExercise):
```

americanBondOption_Tree_OLD

```
def americanBondOption_Tree_OLD(self, texp, strikePrice, face,
                    couponTimes, couponAmounts, americanExercise):
```

df_Tree

Discount factor as seen from now to time tmat as long as the time is on the tree grid.

```
def df_Tree(self, tmat):
```

buildTree

def buildTree(self, treeMat, numTimeSteps, dfTimes, dfValues):

```
    def buildTree(self, treeMat, numTimeSteps, dfTimes, dfValues):
```

accruedInterpolator

Fast calulation of accrued interest using an Actual/Actual type of convention. This does not calculate according to other conventions.

```
def accruedInterpolator(tset, couponTimes, couponAmounts):
```

P_Fast

Forward discount factor as seen at some time t which may be in the future for payment at time T where Rt is the delta-period short rate seen at time t and pt is the discount factor to time t, ptd is the one period discount factor to time t+dt and pT is the discount factor from now until the payment of the $1 of the discount factor$.

```
def P_Fast(t, T, Rt, delta, pt, ptd, pT, _sigma, _a):
```

buildTree_Fast

def buildTree_Fast(a, sigma, treeTimes, numTimeSteps, discountFactors):

```
def buildTree_Fast(a, sigma, treeTimes, numTimeSteps, discountFactors):
```

americanBondOption_Tree_Fast

```
def americanBondOption_Tree_Fast(texp, strikePrice, face,
                    couponTimes, couponAmounts, americanExercise,
                    _sigma, _a, _Q, _pu, _pm, _pd, _rt, _dt, _treeTimes,
                    _dfTimes, _dfValues):
```

fwdFullBondPrice

Price a coupon bearing bond on the option expiry date and return the difference from a strike price. This is used in a root search to find the future expiry time short rate that makes the bond price equal to the option strike price. It is a key step in the Jamshidian bond decomposition approach. The strike is a clean price.

```
def fwdFullBondPrice(rt, *args):
```

7.12 FinLHPlusModel

Class: LHPlusModel()

class LHPlusModel():

Data Members

- _P
- _R
- _H
- _beta
- _P0
- _R0
- _H0
- _beta0

Functions

__init__

def __init__(self, P, R, H, beta, P0, R0, H0, beta0):

```
def __init__(self, P, R, H, beta, P0, R0, H0, beta0):
```

probLossGreaterThanK

def probLossGreaterThanK(self, K):

```
def probLossGreaterThanK(self, K):
```

expMinLKIntegral

def expMinLKIntegral(self, K, dK):

```
def expMinLKIntegral(self, K, dK):
```

expMinLK

def expMinLK(self, K):

```
def expMinLK(self, K):
```

expMinLK2

def expMinLK2(self, K):

```
def expMinLK2(self, K):
```

trancheSurvivalProbability

def trancheSurvivalProbability(self, k1, k2):

```
def trancheSurvivalProbability(self, k1, k2):
```

7.13 FinLossDbnBuilder

indepLossDbnHeterogeneousAdjBinomial

lossRatio):

```
def indepLossDbnHeterogeneousAdjBinomial(numCredits,
                                         condProbs,
                                         lossRatio):
```

portfolioGCD

def portfolioGCD(actualLosses):

```
def portfolioGCD(actualLosses):
```

indepLossDbnRecursionGCD

lossUnits):

```
def indepLossDbnRecursionGCD(numCredits,
                             condDefaultProbs,
                             lossUnits):
```

7.14 FinMertonCreditModel

mertonCreditModelValues

volatility):

```
def mertonCreditModelValues(assetValue,
                            bondFace,
                            timeToMaturity,
                            riskFreeRate,
                            assetGrowthRate,
                            volatility):
```

7.15 FinProcessSimulator

7.15.0.1 Enumerated Type: FinProcessTypes

- GBM
- CIR
- HESTON
- VASICEK
- CEV
- JUMP_DIFFUSION

7.15.0.2 Enumerated Type: FinHestonNumericalScheme

- EULER
- EULERLOG
- QUADEXP

7.15.0.3 Enumerated Type: FinGBMNumericalScheme

- NORMAL
- ANTITHETIC

7.15.0.4 Enumerated Type: FinVasicekNumericalScheme

- NORMAL
- ANTITHETIC

7.15.0.5 Enumerated Type: FinCIRNumericalScheme

- EULER
- LOGNORMAL
- MILSTEIN
- KAHLJACKEL
- EXACT

Class: FinProcessSimulator()

class FinProcessSimulator():

Data Members

No data members found.

Functions

__init__

def __init__(self):

```
def __init__(self):
```

getProcess

seed):

```
def getProcess(
        self,
        processType,
        t,
        modelParams,
        numAnnSteps,
        numPaths,
        seed):
```

getHestonPaths

seed):

```
def getHestonPaths(
        numPaths,
        numAnnSteps,
        t,
        drift,
        s0,
        v0,
        kappa,
        theta,
        sigma,
        rho,
        scheme,
        seed):
```

getGBMPaths

def getGBMPaths(numPaths, numAnnSteps, t, mu, stockPrice, sigma, scheme, seed):

```
def getGBMPaths(numPaths, numAnnSteps, t, mu, stockPrice, sigma, scheme, seed):
```

getVasicekPaths

seed):

```
def getVasicekPaths(
        numPaths,
        numAnnSteps,
        t,
        r0,
        kappa,
        theta,
        sigma,
        scheme,
        seed):
```

getCIRPaths

seed):

```
def getCIRPaths(
        numPaths,
        numAnnSteps,
        t,
        r0,
        kappa,
        theta,
        sigma,
        scheme,
        seed):
```

7.16 FinSABRModel

blackVolFromSABR

def blackVolFromSABR(alpha, beta, rho, nu, f, k, t):

```
def blackVolFromSABR(alpha, beta, rho, nu, f, k, t):
```

7.17 FinStudentTCopulaModel

Class: FinStudentTCopulaModel()

class FinStudentTCopulaModel():

Data Members

No data members found.

Functions

defaultTimes

seed):

```
def defaultTimes(self,
                 issuerCurves,
                 correlationMatrix,
                 degreesOfFreedom,
                 numTrials,
                 seed):
```

7.18 FinVasicekRateModel

Class: FinVasicekModel()

class FinVasicekModel():

Data Members

- _a
- _b
- _sigma

Functions

__init__

def __init__(self, a, b, sigma):

```
def __init__(self, a, b, sigma):
```

meanr

def meanr(r0, a, b, t):

```
def meanr(r0, a, b, t):
```

variancer

def variancer(a, b, sigma, t):

```
def variancer(a, b, sigma, t):
```

zeroPrice

def zeroPrice(r0, a, b, sigma, t):

```
def zeroPrice(r0, a, b, sigma, t):
```

ratePath_MC

def ratePath_MC(r0, a, b, sigma, t, dt, seed):

```
def ratePath_MC(r0, a, b, sigma, t, dt, seed):
```

zeroPrice_MC

def zeroPrice_MC(r0, a, b, sigma, t, dt, numPaths, seed):

```
def zeroPrice_MC(r0, a, b, sigma, t, dt, numPaths, seed):
```

Chapter 8

..//portfolio

8.1 Introduction

8.2 FinBondPortfolio

Class: FinBondPortfolio(object)

Class for fixed coupon bonds and performing related analytics. These are bullet bonds which means they have regular coupon payments of a known size that are paid on known dates plus a payment of par at maturity.

Data Members

- _numBonds
- _settlementDate

Functions

__init__

Create FinBondPortfolio object with a list of bond objects.

```
def __init__(self, settlementDate, bondList):
```

8.3 FinMeanVariancePortfolio

Chapter 9

..//risk

9.1 Introduction

9.2 FinPortfolioCreditDefaultMode

Class: FinPortfolioCreditDefaultMode(object)

class FinPortfolioCreditDefaultMode(object):

Data Members

- _numCredits
- _weights
- _hazardRates
- _recoveryRates
- _betaValues
- _support
- _lossDbn

Functions

__init__

weights):

```
def __init__(self,
             weights):
```

lossDistribution

numPoints):

```
def lossDistribution(self,
                     tmat,
                     hazardRates,
                     recoveryRates,
                     betaValues,
                     numPoints):
```

9.3 FinPortfolioRiskMetrics

expectedLoss

lossProbabilityVector):

```
def expectedLoss(lossSizeVector,
                 lossProbabilityVector):
```

valueAtRisk

confidenceLevel):

```
def valueAtRisk(lossSizeVector,
                lossProbabilityVector,
                confidenceLevel):
```

expectedShortfall

confidenceLevel):

```
def expectedShortfall(lossSizeVector,
                      lossProbabilityVector,
                      confidenceLevel):
```

Chapter 10

..//market//curves

10.1 Introduction

10.2 FinBondYieldCurve

Class: FinBondYieldCurve()

Class to do fitting of the yield curve and to enable interpolation of yields. Because yields assume a flat term structure for each bond, this class does not allow discounting to be done and so does not inherit from FinDiscountCurve. It should only be used for visualisation and simple interpolation but not for full term-structure-consistent pricing.

Data Members

- _settlementDate
- _bonds
- _ylds
- _curveFit
- _yearsToMaturity

Functions

__init__

Fit the curve to a set of bond yields using the type of curve specified. Bounds can be provided if you wish to enforce lower and upper limits on the respective model parameters.

```
def __init__(self, settlementDate, bonds, ylds, curveFit):
```

interpolatedYield

def interpolatedYield(self, maturityDate):

```
def interpolatedYield(self, maturityDate):
```

plot

Display yield curve.

```
def plot(self, title):
```

10.3 FinBondYieldCurveModel

Class: FinCurveFitMethod()

class FinCurveFitMethod():

Data Members

No data members found.

Functions

Class: FinCurveFitPolynomial()

class FinCurveFitPolynomial():

Data Members

- _parentType
- _power
- _coeffs

Functions

__init__

def __init__(self, power=3):

```
def __init__(self, power=3):
```

_interpolatedYield

def _interpolatedYield(self, t):

```
def _interpolatedYield(self, t):
```

Class: FinCurveFitNelsonSiegel()

class FinCurveFitNelsonSiegel():

Data Members

- _parentType

- _beta1
- _beta2
- _beta3
- _tau
- _bounds

Functions

__init__

Fairly permissive bounds. Only tau1 is 1-100

```
def __init__(self, tau=None, bounds=[(-1, -1, -1, 0.5), (1, 1, 1, 100)]):
```

_interpolatedYield

def _interpolatedYield(self, t, beta1=None, beta2=None, beta3=None, tau=None):

```
def _interpolatedYield(self, t, beta1=None, beta2=None, beta3=None, tau=None):
```

Class: FinCurveFitNelsonSiegelSvensson()

class FinCurveFitNelsonSiegelSvensson():

Data Members

- _parentType
- _beta1
- _beta2
- _beta3
- _beta4
- _tau1
- _tau2
- _bounds

Functions

__init__

I impose some bounds to help ensure a sensible result if the user does not provide any bounds. Especially for tau2.

```
def __init__(self, tau1=None,
             tau2=None,
             bounds = [(0, -1, -1, -1, 0, 1), (1, 1, 1, 1, 10, 100)]):
```

_interpolatedYield

beta4=None, tau1=None, tau2=None):

```
def _interpolatedYield(self, t, beta1=None, beta2=None, beta3=None,
                       beta4=None, tau1=None, tau2=None):
```

Class: FinCurveFitBSpline()

class FinCurveFitBSpline():

Data Members

- _parentType
- _power
- _knots

Functions

__init__

def __init__(self, power=3, knots=[1, 3, 5, 10]):

```
def __init__(self, power=3, knots=[1, 3, 5, 10]):
```

_interpolatedYield

def _interpolatedYield(self, t):

```
def _interpolatedYield(self, t):
```

10.4 FinBondZeroCurve

Class: FinBondZeroCurve()

Data Members

- _settlementDate
- _curveDate
- _bonds
- _cleanPrices
- _discountCurve
- _interpMethod
- _yearsToMaturity
- _times
- _values

Functions

__init__

Fit a discount curve to a set of bond yields using the type of curve specified.

```
def __init__(self, settlementDate, bonds, cleanPrices,
             interpMethod=FinInterpMethods.FLAT_FORWARDS):
```

bootstrapZeroRates

def bootstrapZeroRates(self):

```
def bootstrapZeroRates(self):
```

zeroRate

Calculate the zero rate to maturity date.

```
def zeroRate(self, dt, compoundingFreq=-1):
```

df

def df(self, dt):

```
def df(self, dt):
```

survProb

def survProb(self, dt):

```
def survProb(self, dt):
```

fwd

Calculate the continuous forward rate at the forward date.

```
def fwd(self, dt):
```

fwdRate

Calculate the forward rate according to the specified day count convention.

```
def fwdRate(self, date1, date2, dayCountType):
```

plot

Display yield curve.

```
def plot(self, title):
```

print

def print(self):

```
def print(self):
```

f

def f(df, *args):

```
def f(df, *args):
```

10.5 FinCDSCurve

Class: FinCDSCurve()

Generate a survival probability curve implied by the value of CDS contracts given a Libor curve and an assumed recovery rate. A scheme for the interpolation of the survival probabilities is also required.

Data Members

- _curveDate
- _cdsContracts
- _recoveryRate
- _liborCurve
- _interpolationMethod
- _builtOK
- _times
- _values

Functions

__init__

interpolationMethod=FinInterpMethods.FLAT_FORWARDS):

```
def __init__(self,
             curveDate,
             cdsContracts,
             liborCurve,
             recoveryRate=0.40,
             useCache=False,
             interpolationMethod=FinInterpMethods.FLAT_FORWARDS):
```

validate

Ensure that contracts are in increasinbg maturity.

```
def validate(self, cdsContracts):
```

survProb

Extract the survival probability to date dt.

```
def survProb(self, dt):
```

df

Extract the discount factor from the underlying Libor curve.

```
def df(self, t):
```

buildCurve

def buildCurve(self):

```
def buildCurve(self):
```

fwd

Calculate the instantaneous forward rate at the forward date.

```
def fwd(self, dt):
```

fwdRate

Calculate the forward rate according to the specified day count convention.

```
def fwdRate(self, date1, date2, dayCountType):
```

zeroRate

Calculate the zero rate to maturity date.

```
def zeroRate(self, dt, compoundingFreq=-1):
```

print

def print(self):

```
def print(self):
```

uniformToDefaultTime

def uniformToDefaultTime(u, t, v):

```
def uniformToDefaultTime(u, t, v):
```

f

def f(q, *args):

```
def f(q, *args):
```

10.6 FinCurve

inputFrequency

def inputFrequency(f):

```
def inputFrequency(f):
```

inputTime

def inputTime(dt, curve):

```
def inputTime(dt, curve):
```

10.7 FinDiscountCurve

Class: FinDiscountCurve()

class FinDiscountCurve():

Data Members

- _curveDate
- _times
- _values
- _interpMethod

Functions

__init__

interpMethod=FinInterpMethods.FLAT_FORWARDS):

```
def __init__(self, curveDate, times, values,
             interpMethod=FinInterpMethods.FLAT_FORWARDS):
```

zeroRate

Calculate the zero rate to maturity date.

```
def zeroRate(self, dt, compoundingFreq=-1):
```

df

def df(self, dt):

```
def df(self, dt):
```

survProb

def survProb(self, dt):

```
def survProb(self, dt):
```

fwd

Calculate the continuous forward rate at the forward date.

```
def fwd(self, dt):
```

bump

Calculate the continuous forward rate at the forward date.

```
def bump(self, bumpSize):
```

fwdRate

Calculate the forward rate according to the specified day count convention.

```
def fwdRate(self, date1, date2, dayCountType):
```

print

def print(self):

```
def print(self):
```

10.8 FinFlatCurve

Class: FinFlatCurve(FinDiscountCurve)

A trivally simple curve based on a single zero rate with its own specified compounding method. Hence the curve is assumed to be flat.

Data Members

- _curveDate
- _rate
- _cmpdFreq

Functions

__init__

Create a FinFlatCurve which requires a curve date.

```
def __init__(self, curveDate, rate, compoundingFreq=-1):
```

zeroRate

Return the zero rate which is simply the curve rate.

```
def zeroRate(self, dt, compoundingFreq):
```

bump

Calculate the continuous forward rate at the forward date.

```
def bump(self, bumpSize):
```

fwd

Return the fwd rate which is simply the zero rate.

```
def fwd(self, dt):
```

df

Return the discount factor based on the compounding approach.

```
def df(self, dt):
```

fwdRate

Calculate the forward rate according to the specified day count convention.

```
def fwdRate(self, date1, date2, dayCountType):
```

10.9 FinInterpolate

10.9.0.1 Enumerated Type: FinInterpMethods

- LINEAR_ZERO_RATES
- FLAT_FORWARDS
- LINEAR_FORWARDS

interpolate

method):

```
def interpolate(x,
                times,
                dfs,
                method):
```

uinterpolate

Return the interpolated value of y given x and a vector of x and y. The values of x must be monotonic and increasing. The different schemes for interpolation are linear in y (as a function of x), linear in log(y) and piecewise flat in the continuously compounded forward y rate.

```
def uinterpolate(t,
                 times,
                 dfs,
                 method):
```

vinterpolate

Return the interpolated values of y given x and a vector of x and y. The values of x must be monotonic and increasing. The different schemes for interpolation are linear in y (as a function of x), linear in log(y) and piecewise flat in the continuously compounded forward y rate.

```
def vinterpolate(xValues,
                 xvector,
                 dfs,
                 method):
```

10.10 FinLiborCurve

Class: FinLiborCurve(FinDiscountCurve)

Constructs a discount curve as implied by the prices of Libor deposits, FRAs and IRS. The curve date is the date on which we are performing the valuation based on the information available on the curve date. Typically it is the date on which an amount of $1paidhasapresentvalueof1. This class inherits from FinDiscountCurve so has all of the methods that class has.

Data Members

- _name
- _times
- _values
- _curveDate
- _interpMethod
- _usedDeposits
- _usedFRAs
- _usedSwaps

Functions

__init__

interpMethod=FinInterpMethods.FLAT_FORWARDS):

```
def __init__(self,
             name,
             curveDate,
             liborDeposits,
             liborFRAs,
             liborSwaps,
             interpMethod=FinInterpMethods.FLAT_FORWARDS):
```

validateInputs

Construct the discount curve using a bootstrap approach.

```
def validateInputs(self,
                   liborDeposits,
                   liborFRAs,
                   liborSwaps):
```

buildCurve

Construct the discount curve using a bootstrap approach.

```
def buildCurve(self):
```

f

def f(df, *args):

```
def f(df, *args):
```

10.11 FinNelsonSiegelCurve

Class: FinNelsonSiegelCurve()

Implementation of Nelson-Siegel parametrisation of a rate curve. The default is a continuously compounded rate but you can override this by providing a corresponding compounding frequency.

Data Members

- _curveDate
- _beta1
- _beta2
- _beta3
- _tau

Functions

__init__

Creation of a Nelson-Siegel curve. Parameters are provided as a list or vector of 4 values for beta1, beta2, beta3 and tau.

```
def __init__(self, curveDate, params, cmpdFreq=-1):
```

zeroRate

Calculation of zero rates with specified frequency. This function can return a vector of zero rates given a vector of times so must use Numpy functions.

```
def zeroRate(self, dt, compoundingFreq=-1):
```

fwd

Calculation of forward rates. This function can return a vector of instantaneous forward rates given a vector of times.

```
def fwd(self, dt):
```

df

Discount factor for Nelson-Siegel curve parametrisation.

```
def df(self, dt):
```

Class: FinNelsonSiegelSvenssonCurve()

Implementation of Nelson-Siegel-Svensson parametrisation of the zero rate curve

Data Members

- _beta1
- _beta2
- _beta3
- _beta4
- _tau1
- _tau2

Functions

__init__

def __init__(self, beta1, beta2, beta3, beta4, tau1, tau2):

```
def __init__(self, beta1, beta2, beta3, beta4, tau1, tau2):
```

zero

Calculation of zero rates. This function can return a vector of zero rates given a vector of times.

```
def zero(self, t):
```

fwd

Calculation of forward rates. This function uses Numpy so can return a vector of forward rates given a Numpy array vector of times.

```
def fwd(self, t):
```

df

Discount factor for Nelson-Siegel-Svensson curve parametrisation.

```
def df(self, t):
```

10.12 FinPiecewiseFlatCurve

Class: FinPiecewiseCurve()

Curve is made up of a series of zero rates assumed to each have a piecewise flat constant shape OR a piecewise linear shape.

Data Members

- _times
- _zeroRates
- _cmpdFreq
- _interpMethod

Functions

__init__

Curve is a vector of increasing times and zero rates.

```
def __init__(self,
             curveDate,
             times,
             zeroRates,
             compoundingFreq=-1,
             interpolationMethod=FinInterpMethods.FLAT_FORWARDS):
```

zeroRate

def zeroRate(self, t, compoundingFreq):

```
def zeroRate(self, t, compoundingFreq):
```

fwd

def fwd(self, t):

```
def fwd(self, t):
```

df

interpolationMethod=FinInterpMethods.FLAT_FORWARDS):

```
def df(self,
       t,
```

```
freq=0,  # This corresponds to continuous compounding
interpolationMethod=FinInterpMethods.FLAT_FORWARDS):
```

10.13 FinPiecewiseLinearCurve

Class: FinPiecewiseLinearCurve()

Curve is made up of a series of sections assumed to each have a constant forward rate. This class needs to be checked carefully.

Data Members

- _times
- _values

Functions

__init__

Curve is defined by a vector of increasing times and zero rates.

```
def __init__(self, curveDate, times, values):
```

zero

def zero(self, t, interpolationMethod=FinInterpMethods.FLAT_FORWARDS):

```
def zero(self, t, interpolationMethod=FinInterpMethods.FLAT_FORWARDS):
```

fwd

def fwd(self, t):

```
def fwd(self, t):
```

df

interpolationMethod=FinInterpMethods.FLAT_FORWARDS):

```
def df(self,
       t,
       freq=0,  # This corresponds to continuous compounding
       interpolationMethod=FinInterpMethods.FLAT_FORWARDS):
```

10.14 FinPolynomialCurve

Class: FinPolynomialCurve()

Curve with zero rate of specified frequency parametrised as a cubic polynomial.

Data Members

- _curveDate
- _coefficients
- _power

Functions

__init__

Create cubic curve from coefficients

```
def __init__(self, curveDate, coefficients,
             compoundingType=-1):
```

zeroRate

Zero rate from polynomial zero curve.

```
def zeroRate(self, dt):
```

df

Discount factor from polynomial zero curve.

```
def df(self, dt):
```

fwd

Continuously compounded forward rate.

```
def fwd(self, dt):
```

fwdRate

Calculate the forward rate according to the specified day count convention.

```
def fwdRate(self, date1, date2, dayCountType):
```

print

def print(self):

```
def print(self):
```