FinancePy 0.168

Dominic O'Kane

July 11, 2020

Contents

Chapter 1

Introduction to FinancePy

FinancePy

FinancePy is a library of native Python functions which covers the following functionality:

- Valuation and risk models for a wide range of equity, FX, interest rate and credit derivatives.
- Portfolio asset allocation using Markovitz and other methods.

As the library is written entirely in Python, the user has the ability to examine the underlying code and its logic.

The target audience for this library is intended to include:

- Students wishing to learn derivative pricing and Python.
- Professors wishing to teach derivative pricing and Python.
- Traders wishing to price or risk-manage a derivative.
- Quantitative analysts seeking to price or reverse engineer a price.
- Risk managers wishing to replicate and understand a price.
- Portfolio managers wishing to check prices or calculate risk measures
- Fund managers wanting to value a portfolio or examine a trading strategy
- Structurers or financial engineers seeking to examine the pricing of a derivative structure.

Users are expected to have a good, but not advanced, understanding of Python.

Up until now my main focus has been on financial derivatives. In general my approach has been:

1. To make the code as simple as possible so that 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 - I am OK with some code duplication, at least temporarily.

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 hope to reduce the size of such differences.

IF YOU HAVE ANY EXAMPLES YOU WOULD LIKE ME TO REPLICATE, SEND ME SCREENSHOTS OF ALL THE UNDERLYING DATA AND MODEL DETAILS

The Library Design

The underlying Python library is split into a number of major modules:

- Finutils - These are utility functions used to assist you with modelling a security. These include dates (FinDate), calendars, schedule generation, some finance-related mathematics functions and some helper functions.

- Market - These are modules that capture the market information used to value a security. These include interest rate and credit curves, volatility surfaces and prices.

- Models - These are the low-level models used to value derivative securities ranging from Black-Scholes to complex stochastic volatility models.

- Products - These are the actual securities and range from Government bonds to Bermudan swaptions.

Any price is the result of a PRODUCT + MODEL + MARKET. The interface to each product has a value() function that will take a model and market to produce a price.

There are also two other folders which are currently fairly empty: They are:

- Portfolio - This will be where portfolio allocation will go,

- Risk - This is for portfolio risk analysis

How to Use the Library

FinancePy can be installed using pip (see instructions below). I have provided a range of template Jupyter notebooks under the github repository called FinancePy-Examples. The link is as follows:

https://github.com/domokane/FinancePy-Examples

A pdf description of functions can be found at the same repository.

Help Needed

The current version of the code is a beta. If you have any questions or issues then please send them to me. Contact me via the github page.

Author

My name is Dr. Dominic O'Kane. I teach Finance at the EDHEC Business School in Nice, France. I have 12 years of industry experience and 10 years of academic experience.

Installation

FinancePy can be installed from pip using the command:
 pip install financepy
 To upgrade an existing installation type:
 pip install –upgrade financepy

Dependencies

FinancePy depends on Numpy, Numba and Scipy.

Changelog

See the changelog for a detailed history of changes

Contributions

Contributions are very welcome. There are a number of requirements:

- You should use CamelCase i.e. variables of the form optionPrice
- Comments are required for every class and function and they should be clear
- At least one test case must be provided for every function
- Follow the style of the code as currently written. This may change over time but please use the current style as your guide.

License

MIT

Chapter 2

financepy.finutils

2.1 Introduction

This is a collection of modules used across a wide range of FinancePy functions. Examples include date generation, special mathematical functions and useful helper functions for performing some repeated action.

- FinDate is a class for handling dates in a financial setting. Special functions are included for computing IMM dates and CDS dates and moving dates forward by tenors.
- FinCalendar is a class for determining which dates are not business dates in a specific region or country.
- FinDayCount is a class for determining accrued interest in bonds and also accrual factors in ISDA swap-like contracts.
- FinError is a class which handles errors in the calculations done within FinancePy
- FinFrequency takes in a frequency type and then returns the number of payments per year
- FinGlobalVariables holds the value of constants used across the whole of FinancePy
- FinHelperFunctions is a set of helpful functions that can be used in a number of places
- FinMath is a set of mathematical functions specific to finance which have been optimised for speed using Numba
- FinRateConverter converts rates for one compounding frequency to rates for a different frequency
- FinSchedule generates a sequence of cashflow payment dates in accordance with financial market standards
- FinStatistics calculates a number of statistical variables such as mean, standard deviation and variance
- FinTestCases is the code that underlies the test case framework used across FinancePy

2.2 FinCalendar

Enumerated Type: FinBusDayAdjustTypes

- NONE
- FOLLOWING
- MODIFIED_FOLLOWING
- PRECEDING
- MODIFIED_PRECEDING

Enumerated Type: FinCalendarTypes

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

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):
```

__repr__

s = self._type

```
def __repr__(self):
```

2.3 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 a day of month, month and year. The year must be a 4-digit number greater than or equal to 1900.

```
def __init__(self, d, m, y):
```

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__

return self._excelDate ¡ other._excelDate

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

__gt__

return self._excelDate ¿ other._excelDate

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

__le__

return self._excelDate ¡= other._excelDate

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

__ge__

return self._excelDate ¿= other._excelDate

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

__sub__

return self._excelDate - other._excelDate

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

__eq__

return self._excelDate == other._excelDate

```
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. If mm is an integer or float you get back a single date. If mm is a vector you get back a vector of dates.

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

addYears

Returns a new date that is yy years after the FinDate. If yy is an integer or float you get back a single date. If yy is a list you get back a vector of dates.

```
def addYears(self, yy):
```

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):
```

date

Returns a datetime of the date

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

__repr__

returns a formatted string of the date

```
def __repr__(self):
```

print

prints formatted string of the date.

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

dailyWorkingDaySchedule

Returns a list of working dates between startDate and endDate. This function should be replaced by dateRange once addTenor allows for working days.

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

datediff

Calculate the number of days between two dates.

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

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):
```

dateRange

Returns a list of dates between startDate (inclusive) and endDate (inclusive). The tenor represents the distance between two consecutive dates and is set to daily by default.

```
def dateRange(startDate, endDate, tenor="1D"):
```

2.4 FinDayCount

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):
```

__repr__

Returns the calendar type as a string.

```
def __repr__(self):
```

2.5 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

print("FinError:", self._message)

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

hide_traceback

etype, value, tb = sys.exc_info()

```
def hide_traceback(exc_tuple=None, filename=None, tb_offset=None,
                   exception_only=False, running_compiled_code=False):
```

func_name

```
def func_name():
```

isNotEqual

if abs(x - y) ¿ tol:

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

2.6 FinFrequency

Enumerated Type: FinFrequencyTypes

- CONTINUOUS
- SIMPLE
- 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):
```

2.7 FinGlobalVariables

2.8 FinHelperFunctions

checkVectorDifferences

Compare two vectors elementwise to see if they are more different than tolerance.

```
def checkVectorDifferences(x, y, tol):
```

checkDate

Check that input d is a FinDate.

```
def checkDate(d):
```

dump

Get a list of all of the attributes of a class (not built in ones)

```
def dump(obj):
```

printTree

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

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

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

fast range function that takes start value, stop value and step.

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

normaliseWeights

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

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

labelToString

Format label/value pairs for a unified formatting.

```
def labelToString(label, value, separator="\n", listFormat=False):
```

tableToString

Format a 2D array into a table-like string.

```
def tableToString(header, valueTable, floatPrecision="10.7f"):
```

toUsableType

Convert a type such that it can be used with ‘isinstance‘

```
def toUsableType(t):
```

checkArgumentTypes

Check that all values passed into a function are of the same type as the function annotations. If a value has not been annotated, it will not be checked.

```
def checkArgumentTypes(func, values):
```

2.9 FinMath

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):
```

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

x = []

```
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 currently is 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

return phi2(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):
```

2.10 FinOptionTypes

Enumerated Type: FinOptionTypes

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

2.11 FinRateConverter

Class: FinRateConverter(object)

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

Data Members

- name
- months

Functions

__init__

PLEASE ADD A FUNCTION DESCRIPTION

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

__repr__

s = self.name

```
def __repr__(self):
```

2.12 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 which calculates a sequence of dates in line with market convention for fixed income products.

```
def __init__(self,
             startDate,
             endDate,
             frequencyType=FinFrequencyTypes.ANNUAL,
             calendarType=FinCalendarTypes.WEEKEND,
             busDayAdjustType=FinBusDayAdjustTypes.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 specified business day convention and the specified calendar.

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

generate_alternative

This adjusts each date BEFORE generating the next date. 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_alternative(self):
```

__repr__

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

```
def __repr__(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):
```

2.13 FinSobol

getGaussianSobol

""""""

```
def getGaussianSobol(numPoints, dimension):
```

getUniformSobol

""""""

```
def getUniformSobol(numPoints, dimension):
```

2.14 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):
```

Chapter 3

financepy.market.curves

3.1 Introduction

Curves

Overview

These modules create a family of curve types related to the term structures of interest rates. There are two basic types of curve:

1. Best fit yield curves fitting to bond prices which are used for interpolation. A range of curve shapes from polynomials to B-Splines is available.

2. Discount curves that can be used to present value a future cash flow. These differ from best fits curves in that they exactly refit the prices of bonds or CDS. The different discount curves are created by calibrating to different instruments. They also differ in terms of the term structure shapes they can have. Different shapes have different impacts in terms of locality on risk management performed using these different curves. There is often a trade-off between smoothness and locality.

Best Fit Bond Curves

The first category are FinBondYieldCurves.

FinBondYieldCurve

This module describes a curve that is fitted to bond yields calculated from bond market prices supplied by the user. The curve is not guaranteed to fit all of the bond prices exactly and a least squares approach is used. A number of fitting forms are provided which consist of

- Polynomial
- Nelson-Siegel
- Nelson-Siegal-Svensson
- Cubic B-Splines

This fitted curve cannot be used for pricing as yields assume a flat term structure. It can be used for fitting and interpolating yields off a nicely constructed yield curve interpolation curve.

FinCurveFitMethod

This module sets out a range of curve forms that can be fitted to the bond yields. These includes a number of parametric curves that can be used to fit yield curves. These include:

- Polynomials of any degree
- Nelson-Siegel functional form.
- Nelson-Siegel-Svensson functional form.
- B-Splines

Discount Curves

These are curves which supply a discount factor that can be used to present-value future payments.

FinDiscountCurve

This is a curve made from a Numpy array of times and discount factor values that represents a discount curve. It also requires a specific interpolation scheme. A function is also provided to return a survival probability so that this class can also be used to handle term structures of survival probabilities. Other curves inherit from this in order to share common functionality.

FinDiscountCurveFlat

This is a class that takes in a single flat rate.

FinDiscountCurveNS

Implementation of the Nelson-Siegel curve parametrisation.

FinDiscountCurveNSS

Implementation of the Nelson-Siegel-Svensson curve parametrisation.

FinDiscountCurveZeros

This is a discount curve that is made from a vector of times and zero rates.

FinInterpolate

This module contains the interpolation function used throughout the discount curves when a discount factor needs to be interpolated. There are three interpolation methods:

1. PIECEWISE LINEAR - This assumes that a discount factor at a time between two other known discount factors is obtained by linear interpolation. This approach does not guarantee any smoothness but is local. It does not guarantee positive forwards (assuming positive zero rates).

2. PIECEWISE LOG LINEAR - This assumes that the log of the discount factor is interpolated linearly. The log of a discount factor to time T is T x R(T) where R(T) is the zero rate. So this is not linear interpolation of R(T) but of T x R(T).

3. FLAT FORWARDS - This interpolation assumes that the forward rate is constant between discount factor points. It is not smooth but is highly local and also ensures positive forward rates if the zero rates are positive.

3.2 FinDiscountCurve

Class: FinDiscountCurve()

This is a base discount curve which has an internal representation of a vector of times and discount factors and an interpolation scheme for interpolating between these fixed points.

Data Members

- _valuationDate
- _times
- _discountFactors
- _interpMethod

Functions

__init__

Create the discount curve from a vector of times and discount factors with an anchor date and specify an interpolation scheme. As we are explicity linking dates and discount factors, we do not need to specify any compounding convention or day count calculation since discount factors are pure prices. We do however need to specify a convention for interpolating the discount factors in time.

```
def __init__(self,
             valuationDate: FinDate,
```

zeroRate

Calculate the zero rate to maturity date. The compounding frequency of the rate defaults to continuous which is useful for supplying a rate to theoretical models such as Black-Scholes that require a continuously compounded zero rate as input.

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

parRate

Calculate the par rate to maturity date. This is the rate paid by a bond that has a price of par today.

```
def parRate(self,
            dt: FinDate,
```

_zeroRate

Calculate the zero rate to maturity date but with times as inputs. This function is used internally and should be discouraged for external use. The compounding frequency defaults to continuous.

```
def _zeroRate(self,
              times: list,
```

_parRate

Calculate the zero rate to maturity date but with times as inputs. This function is used internally and should be discouraged for external use. The compounding frequency defaults to continuous.

```
def _parRate(self,
             times: list,
```

df

Function to calculate a discount factor from a date or a vector of dates.

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

_df

Hidden function to calculate a discount factor from a time or a vector of times. Discourage usage in favour of passing in dates.

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

survProb

return self.df(dt)

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

fwd

Calculate the continuously compounded forward rate at the forward FinDate provided. This is done by perturbing the time by a small amount and measuring the change in the log of the discount factor divided by the time increment dt.

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

_fwd

Calculate the continuously compounded forward rate at the forward time provided. This is done by perturbing the time by a small amount and measuring the change in the log of the discount factor divided by the time increment dt.

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

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):
```

__repr__

header = ”TIMES,DISCOUNT FACTORS”

```
def __repr__(self):
```

print

Simple print function for backward compatibility.

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

timesFromDates

PLEASE ADD A FUNCTION DESCRIPTION

```
def timesFromDates(dt, valuationDate):
```

pv01Times

Calculate a bond style pv01 by calculating remaining coupon times for a bond with t years to maturity and a coupon frequency of f. The order of the list is reverse time order - it starts with the last coupon date and ends with the first coupon date.

```
def pv01Times(t, f):
```

3.3 FinDiscountCurveFlat

Class: FinDiscountCurveFlat(FinDiscountCurve)

A very simple discount curve based on a single zero rate with its own specified compounding method. Hence the curve is assumed to be flat. It is used for quick and dirty analysis and when limited information is available. It inherits several methods from FinDiscountCurve.

Data Members

- _valuationDate
- _flatRate
- _rateFrequencyType
- _dayCountType
- _times
- _discountFactors

Functions

__init__

Create a discount curve which is flat. This is very useful for quick testing and simply requires a curve date and a rate and also a frequency. As we have entered a rate, a corresponding day count convention must be used to specify how time periods are to be measured. As the curve is flat, no interpolation scheme is required.

```
def __init__(self,
             valuationDate: FinDate,
```

bump

Creates a new FinDiscountCurveFlat object with the entire curve bumped up by the bumpsize. All other parameters are preserved.

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

df

Function to calculate a discount factor from a date or a vector of dates. Times are calculated according to a specified convention.

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

_df

Return the discount factor given a single or vector of times. The discount factor depends on the rate and this in turn depends on its compounding frequency and it defaults to continuous compounding. It also depends on the day count convention. This was set in the construction of the curve to be ACT_ACT_ISDA.

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

__repr__

PLEASE ADD A FUNCTION DESCRIPTION

```
def __repr__(self):
```

print

Simple print function for backward compatibility.

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

timesFromDatesFlat

PLEASE ADD A FUNCTION DESCRIPTION

```
def timesFromDatesFlat(dt, valuationDate, dayCountType):
```

3.4 FinDiscountCurveNS

Class: FinDiscountCurveNS(FinDiscountCurve)

Implementation of Nelson-Siegel parametrisation of a discount curve. The internal rate is a continuously compounded rate but you can calculate alternative frequencies by providing a corresponding compounding frequency.

Data Members

- _curveDate

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):
```

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. Default frequency is a continuously compounded rate.

```
def zeroRate(self, dt, frequencyType=FinFrequencyTypes.CONTINUOUS):
```

fwd

Calculation of continuously compounded 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):
```

3.5 FinDiscountCurveNSS

Class: FinDiscountCurveNSS(FinDiscountCurve)

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

Data Members

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

Functions

__init__

PLEASE ADD A FUNCTION DESCRIPTION

```
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 returns 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):
```

3.6 FinDiscountCurvePiecewiseFlat

Class: FinDiscountCurvePWFlat(FinDiscountCurve)

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,
             frequencyType=FinFrequencyTypes.CONTINUOUS,
             interpolationMethod=FinInterpMethods.FLAT_FORWARDS):
```

zeroRate

PLEASE ADD A FUNCTION DESCRIPTION

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

fwd

NEED TODO THIS

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

df

PLEASE ADD A FUNCTION DESCRIPTION

```
def df(self,
       t,
       frequencyType=FinFrequencyTypes.CONTINUOUS,
       interpolationMethod=FinInterpMethods.FLAT_FORWARDS):
```

3.7 FinDiscountCurvePiecewiseLinear

Class: FinDiscountCurvePW(FinDiscountCurve)

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

PLEASE ADD A FUNCTION DESCRIPTION

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

fwd

NEED TODO THIS

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

df

PLEASE ADD A FUNCTION DESCRIPTION

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

3.8 FinDiscountCurvePoly

Class: FinDiscountCurvePoly(FinDiscountCurve)

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):
```

__repr__

Display internal parameters of curve.

```
def __repr__(self):
```

print

Simple print function for backward compatibility.

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

3.9 FinDiscountCurveZeros

Class: FinDiscountCurveZeros(FinDiscountCurve)

This is a curve calculated from a set of dates and zero rates. As we have rates as inputs, we need to specify the corresponding compounding frequency. Also to go from rates and dates to discount factors we need to compute the year fraction correctly and for this we require a day count convention. Finally, we need to interpolate the zero rate for the times between the zero rates given and for this we must specify an interpolation convention.

Data Members

- _valuationDate
- _frequencyType
- _dayCountType
- _interpMethod
- _times
- _zeroRates
- _discountFactors

Functions

__init__

Create the discount curve from a vector of dates and zero rates factors. The first date is the curve anchor. Then a vector of zero dates and then another same-length vector of rates. The rate is to the corresponding date. We must specify the compounding frequency of the zero rates and also a day count convention for calculating times which we must do to calculate discount factors. Finally we specify the interpolation scheme.

```
def __init__(self,
             valuationDate,
             dates,
             zeroRates,
             frequencyType=FinFrequencyTypes.ANNUAL,
             dayCountType=FinDayCountTypes.ACT_ACT_ISDA,
             interpMethod=FinInterpMethods.FLAT_FORWARDS):
```

_buildCurvePoints

Hidden function to extract discount factors from zero rates.

```
def _buildCurvePoints(self):
```

bump

Calculate the continuous forward rate at the forward date.

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

__repr__

PLEASE ADD A FUNCTION DESCRIPTION

```
def __repr__(self):
```

timesFromDatesFlat

PLEASE ADD A FUNCTION DESCRIPTION

```
def timesFromDatesFlat(dt, valuationDate, dayCountType):
```

3.10 FinInterpolate

Enumerated Type: FinInterpMethods

- LINEAR_ZERO_RATES
- FLAT_FORWARDS
- LINEAR_FORWARDS

interpolate

PLEASE ADD A FUNCTION DESCRIPTION

```
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):
```

Chapter 4

financepy.market.volatility

4.1 Introduction

Market Volatility

Overview

These modules create a family of curve types related to the market volatility. There are three types of class:

1. Term structures of volatility i.e. volatility as a function of option expiry date.
2. Volatility curves which are smile/skews so store volatility as a function of option strike.
3. Volatility surfaces which hold volatility as a function of option expiry date AND option strike.

The classes are as follows:

FinEquityVolCurve

Equity volatility as a function of option strike. This is usually a skew shape.

FinFXVolSurface

FX volatility as a function of option expiry and strike. This class constructs the surface from the ATM volatility and 25 delta strangles and risk reversals and does so for multiple expiry dates.

FinLiborCapFloorVol

Libor cap/floor volatility as a function of option expiry (cap/floor start date). Takes in cap (flat) volatility and boostraps the caplet volatility. This is assumed to be piecewise flat.

FinLiborCapFloorVolFn

Parametric function for storing the cap and caplet volatilities based on form proposed by Rebonato.

4.2 FinEquityVolCurve

Class: FinEquityVolCurve()

Class to manage a smile or skew in volatility at a single maturity horizon. It fits the volatility using a polynomial. Includes analytics to extract the implied pdf of the underyling at maturity.

Data Members

- _curveDate
- _strikes
- _volatilities
- _z
- _f

Functions

__init__

PLEASE ADD A FUNCTION DESCRIPTION

```
def __init__(self,
             curveDate,
             expiryDate,
             strikes,
             volatilities,
             polynomial=3):
```

volatility

Return the volatility for a strike using a given polynomial interpolation.

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

calculatePDF

calculate the probability density function of the underlying using the volatility smile or skew curve following the approach set out in Breedon and Litzenberger.

```
def calculatePDF():
```

4.3 FinFXVolSurface

Class: FinFXVolSurface()

Data Members

- _valueDate
- _spotFXRate
- _currencyPair
- _notionalCurrency
- _domDiscountCurve
- _forDiscountCurve
- _numVolCurves
- _tenors
- _atmVols
- _mktStrangle25DeltaVols
- _riskReversal25DeltaVols
- _atmMethod
- _deltaMethod
- _deltaMethodString
- _tenorIndex
- _F0T
- _K_25_D_C
- _K_25_D_P
- _K_25_D_C_MS
- _K_25_D_P_MS
- _K_ATM
- _V_25_D_MS
- _deltaATM
- _texp
- _curveDate

- _strikes
- _volatilities
- _z
- _f

Functions

__init__

PLEASE ADD A FUNCTION DESCRIPTION

```
def __init__(self,
             valueDate,
             spotFXRate,
             currencyPair,
             notionalCurrency,
             domDiscountCurve,
             forDiscountCurve,
             tenors,
             atmVols,
             mktStrangle25DeltaVols,
             riskReversal25DeltaVols,
             atmMethod=FinFXATMMethod.FWD_DELTA_NEUTRAL,
             deltaMethod=FinFXDeltaMethod.SPOT_DELTA):
```

volFunction

Return the volatility for a strike using a given polynomial interpolation following Section 3.9 of Iain Clark book.

```
def volFunction(self, K, tenorIndex):
```

buildVolSurface

PLEASE ADD A FUNCTION DESCRIPTION

```
def buildVolSurface(self):
```

solveForSmileStrike

Solve for the strike that sets the delta of the option equal to the target value of delta allowing the volatility to be a function of the strike

```
def solveForSmileStrike(self,
                        vanillaOption,
                        deltaTarget,
                        tenorIndex):
```

checkCalibration

PLEASE ADD A FUNCTION DESCRIPTION

```
def checkCalibration(self):
```

plotVolCurves

PLEASE ADD A FUNCTION DESCRIPTION

```
def plotVolCurves(self):
```

calculatePDF

calculate the probability density function of the underlying # using the volatility smile or skew curve following the approach set # out in Breedon and Litzenberger.

```
# def calculatePDF(self):
```

obj

Return a function that is minimised when the ATM, MS and RR vols have been best fitted using the parametric volatility curve respresented by cvec

```
def obj(cvec, *args):
```

deltaFit

```
def deltaFit(K, *args):
```

4.4 FinLiborCapVolCurve

Class: FinLiborCapVolCurve()

Class to manage a term structure of cap (flat) volatilities and to do the conversion to caplet (spot) volatilities. This does not manage a strike dependency, only a term structure. The cap and caplet volatilies are keyed off the cap and caplet maturity dates. However this volatility only applies to the evolution of the Libor rate out to the caplet start dates. Note also that this class also handles floor vols.

Data Members

- _curveDate
- _capSigmas
- _capMaturityDates
- _dayCountType
- _capletGammas

Functions

__init__

Create a cap/floor volatility curve given a curve date, a list of cap maturity dates and a vector of cap volatilities. To avoid confusion first date of the capDates must be equal to the curve date and first cap volatility for this date must equal zero. The internal times are calculated according to the provided day count convention. Note cap and floor volatilities are the same for the same strike and tenor, I just refer to cap volatilities in the code for code simplicity.

```
def __init__(self,
             curveDate,  # Valuation date for cap volatility
             capMaturityDates,  # curve date + maturity dates for caps
             capSigmas,  # Flat cap volatility for cap maturity dates
             dayCountType):
```

generateCapletVols

Bootstrap caplet volatilities from cap volatilities using similar notation to Hulls book (page 32.11). The first volatility in the vector of caplet vols is zero.

```
def generateCapletVols(self):
```

capletVol

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

capVol

Return the cap flat volatility for a specific cap maturity date for the last caplet/floorlet in the cap/floor. The volatility interpolation is piecewise flat.

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

__repr__

Output the contents of the FinCapVolCurve class object.

```
def __repr__(self):
```

4.5 FinLiborCapVolCurveFn

Class: FinLiborCapVolCurveFn()

Class to manage a term structure of caplet volatilities using the parametric form suggested by Rebonato (1999).

Data Members

- _curveDate
- _a
- _b
- _c
- _d

Functions

__init__

PLEASE ADD A FUNCTION DESCRIPTION

```
def __init__(self,
             curveDate,
             a,
             b,
             c,
             d):
```

capFloorletVol

Return the caplet volatility.

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

Chapter 5

financepy.products.equity

5.1 Introduction

This folder covers a range of equity derivative products. These range from simple Vanilla-style options to more complex payoffs and path-dependent options.

5.2 FinEquityAsianOption

Class: FinEquityAsianOption(FinEquityOption)

Class for an Equity Asian Option. This is an option with a final payoff linked to the average stock price. The valuation is done for both an arith- metic and geometric average.

Data Members

- _startAveragingDate
- _expiryDate
- _strikePrice
- _optionType
- _numObservations

Functions

__init__

Creat FinEquityAsian option.

```
def __init__(self,
             startAveragingDate: FinDate,
```

value

Calculate the value of an Asian option using one of the specified models.

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

valueGeometric

This option valuation is based on paper by Kemna and Vorst 1990. It calculates the Geometric Asian option price which is a lower bound on the Arithmetic option price. This should not be used as a valuation model for the Arithmetic Average option but can be used as a control variate for other approaches.

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

valueCurran

Valuation of an Asian option using the result by Vorst.

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

valueTurnbullWakeman

Asian option valuation based on paper by Turnbull and Wakeman 1991 which uses the edgeworth expansion to find the first two moments of the arithmetic average.

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

valueMC

Monte Carlo valuation of the Asian Average option.

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

valueMC_fast

Monte Carlo valuation of the Asian Average option.

```
def valueMC_fast(self, valueDate, stockPrice, discountCurve,
                 dividendYield,  # Yield
                 model,          # Model
                 numPaths,       # Numpaths integer
                 seed,
                 accruedAverage):
```

valueMC_fast_CV

Monte Carlo valuation of the Asian Average option using a control variate method.

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

valueMC_NUMBA

PLEASE ADD A FUNCTION DESCRIPTION

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

valueMC_fast_NUMBA

PLEASE ADD A FUNCTION DESCRIPTION

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

valueMC_fast_CV_NUMBA

PLEASE ADD A FUNCTION DESCRIPTION

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

5.3 FinEquityBarrierOption

Enumerated Type: FinEquityBarrierTypes

- 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: FinEquityBarrierOption(FinEquityOption)

Class to hold details of an Equity Barrier Option. It also calculates the option price using Black Scholes for 8 different variants on the Barrier structure in enum FinEquityBarrierTypes.

Data Members

- _expiryDate
- _strikePrice
- _barrierLevel
- _numObservationsPerYear
- _optionType
- _notional

Functions

__init__

strikePrice: float,

```
def __init__(self,
             expiryDate: FinDate,
```

value

This prices the option using the formulae given in the paper by Clewlow, Llanos and Strickland December 1994 which can be found at https://warwick.ac.uk/fac/soc/wbs/subjects/finance/research/wpaperseries/1994/94-54.pdf

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

valueMC

PLEASE ADD A FUNCTION DESCRIPTION

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

5.4 FinEquityBasketOption

Class: FinEquityBasketOption(FinEquityOption)

class FinEquityBasketOption(FinEquityOption):

Data Members

- _expiryDate
- _strikePrice
- _optionType
- _numAssets

Functions

__init__

strikePrice: float,

```
def __init__(self,
             expiryDate: FinDate,
```

validate

PLEASE ADD A FUNCTION DESCRIPTION

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

value

PLEASE ADD A FUNCTION DESCRIPTION

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

valueMC

PLEASE ADD A FUNCTION DESCRIPTION

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

5.5 FinEquityBinomialTree

Enumerated Type: FinEquityTreePayoffTypes

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

Enumerated Type: FinEquityTreeExerciseTypes

- EUROPEAN
- AMERICAN

Class: FinEquityBinomialTree()

class FinEquityBinomialTree():

Data Members

- m_optionValues
- m_stockValues
- m_upProbabilities
- m_numSteps
- m_numNodes

Functions

__init__

pass

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

value

PLEASE ADD A FUNCTION DESCRIPTION

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

validatePayoff

PLEASE ADD A FUNCTION DESCRIPTION

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

payoffValue

PLEASE ADD A FUNCTION DESCRIPTION

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

valueOnce

PLEASE ADD A FUNCTION DESCRIPTION

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

5.6 FinEquityCompoundOption

Class: FinEquityCompoundOption(FinEquityOption)

class FinEquityCompoundOption(FinEquityOption):

Data Members

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

Functions

__init__

expiryDate2: FinDate,

```
def __init__(self,
             expiryDate1: FinDate,
```

value

PLEASE ADD A FUNCTION DESCRIPTION

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

valueTree

PLEASE ADD A FUNCTION DESCRIPTION

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

impliedStockPrice

PLEASE ADD A FUNCTION DESCRIPTION

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

f

PLEASE ADD A FUNCTION DESCRIPTION

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

valueOnce

PLEASE ADD A FUNCTION DESCRIPTION

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

5.7 FinEquityDigitalOption

Class: FinEquityDigitalOption(FinEquityOption)

class FinEquityDigitalOption(FinEquityOption):

Data Members

- _expiryDate
- _strikePrice
- _optionType

Functions

__init__

strikePrice: float,

```
def __init__(self,
             expiryDate: FinDate,
```

value

PLEASE ADD A FUNCTION DESCRIPTION

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

valueMC

PLEASE ADD A FUNCTION DESCRIPTION

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

5.8 FinEquityFixedLookbackOption

Enumerated Type: FinEquityFixedLookbackOptionTypes

- FIXED_CALL
- FIXED_PUT

Class: FinEquityFixedLookbackOption(FinEquityOption)

class FinEquityFixedLookbackOption(FinEquityOption):

Data Members

- _expiryDate
- _optionType
- _optionStrike

Functions

__init__

optionType: FinEquityFixedLookbackOptionTypes,

```
def __init__(self,
             expiryDate: FinDate,
```

value

PLEASE ADD A FUNCTION DESCRIPTION

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

valueMC

PLEASE ADD A FUNCTION DESCRIPTION

```
def valueMC(
        self,
        valueDate,
        stockPrice,
        discountCurve,
        dividendYield,
        volatility,
        stockMinMax,
```

```
numPaths=10000,
numStepsPerYear=252,
seed=4242):
```

5.9 FinEquityFloatLookbackOption

Enumerated Type: FinEquityFloatLookbackOptionTypes

- FLOATING_CALL
- FLOATING_PUT

Class: FinEquityFloatLookbackOption(FinEquityOption)

class FinEquityFloatLookbackOption(FinEquityOption):

Data Members

- _expiryDate
- _optionType

Functions

__init__

optionType: FinEquityFloatLookbackOptionTypes):

```
def __init__(self,
             expiryDate: FinDate,
```

value

PLEASE ADD A FUNCTION DESCRIPTION

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

valueMC

PLEASE ADD A FUNCTION DESCRIPTION

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

```
seed=4242):
```

5.10 FinEquityModelTypes

Class: FinEquityModel(object)

Data Members

- _parentType
- _volatility
- _implementation

Functions

__init__

self._parentType = None

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

Class: FinEquityModelBlackScholes(FinEquityModel)

class FinEquityModelBlackScholes(FinEquityModel):

Data Members

- _parentType
- _volatility
- _numStepsPerYear
- _useTree

Functions

__init__

self._parentType = FinEquityModel

```
def __init__(self, volatility, numStepsPerYear=100, useTree=False):
```

Class: FinEquityModelHeston(FinEquityModel)

class FinEquityModelHeston(FinEquityModel):

Data Members

- _parentType
- _volatility

- _meanReversion
- _implementation

Functions

__init__

self._parentType = FinEquityModel

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

5.11 FinEquityOption

Enumerated Type: FinEquityOptionTypes

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

Enumerated Type: FinEquityOptionModelTypes

- BLACKSCHOLES
- ANOTHER

Class: FinEquityOption(object)

class FinEquityOption(object):

Data Members

No data members found.

Functions

delta

v = self.value(

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

gamma

v = self.delta(

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

vega

v = self.value(

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

theta

v = self.value(

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

rho

PLEASE ADD A FUNCTION DESCRIPTION

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

5.12 FinEquityRainbowOption

Enumerated Type: FinEquityRainbowOptionTypes

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

Class: FinEquityRainbowOption(FinEquityOption)

class FinEquityRainbowOption(FinEquityOption):

Data Members

- _expiryDate
- _payoffType
- _payoffParams
- _numAssets

Functions

__init__

payoffType: FinEquityRainbowOptionTypes,

```
def __init__(self,
             expiryDate: FinDate,
```

validate

PLEASE ADD A FUNCTION DESCRIPTION

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

validatePayoff

PLEASE ADD A FUNCTION DESCRIPTION

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

value

PLEASE ADD A FUNCTION DESCRIPTION

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

valueMC

PLEASE ADD A FUNCTION DESCRIPTION

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

payoffValue

PLEASE ADD A FUNCTION DESCRIPTION

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

valueMCFast

PLEASE ADD A FUNCTION DESCRIPTION

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

5.13 FinEquityVanillaOption

Class: FinEquityVanillaOption(FinEquityOption)

class FinEquityVanillaOption(FinEquityOption):

Data Members

- _expiryDate
- _strikePrice
- _optionType
- _numOptions

Functions

__init__

strikePrice: float,

```
def __init__(self,
             expiryDate: FinDate,
```

value

PLEASE ADD A FUNCTION DESCRIPTION

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

xdelta

PLEASE ADD A FUNCTION DESCRIPTION

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

xgamma

PLEASE ADD A FUNCTION DESCRIPTION

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

xvega

PLEASE ADD A FUNCTION DESCRIPTION

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

xtheta

PLEASE ADD A FUNCTION DESCRIPTION

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

impliedVolatility

PLEASE ADD A FUNCTION DESCRIPTION

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

valueMC

PLEASE ADD A FUNCTION DESCRIPTION

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

value_MC_OLD

PLEASE ADD A FUNCTION DESCRIPTION

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

f

PLEASE ADD A FUNCTION DESCRIPTION

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

fvega

PLEASE ADD A FUNCTION DESCRIPTION

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

5.14 FinEquityVarianceSwap

Class: FinEquityVarianceSwap(object)

Data Members

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

Functions

__init__

Create variance swap contract.

```
def __init__(self,
             startDate: FinDate,
```

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

PLEASE ADD A FUNCTION DESCRIPTION

```
    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

PLEASE ADD A FUNCTION DESCRIPTION

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

Chapter 6

financepy.products.credit

6.1 Introduction

This folder contains a set of credit-related assets ranging from CDS to CDS options, to CDS indices, CDS index options and then to CDS tranches. They are as follows:

- FinCDS is a credit default swap contract. It includes schedule generation, contract valuation and risk-management functionality.
- FinCDSBasket is a credit default basket such as a first-to-default basket. The class includes valuation according to the Gaussian copula.
- FinCDSIndexOption is an option on an index of CDS such as CDX or iTraxx. A full valuation model is included.
- FinCDSOption is an option on a single CDS. The strike is expressed in spread terms and the option is European style. It is different from an option on a CDS index option. A suitable pricing model is provided which adjusts for the risk that the reference credit defaults before the option expiry date.
- FinCDSTranche is a synthetic CDO tranche. This is a financial derivative which takes a loss if the total loss on the portfolio exceeds a lower threshold K1 and which is wiped out if it exceeds a higher threshold K2. The value depends on the default correlation between the assets in the portfolio of credits. This also includes a valuation model based on the Gaussian copula model.

FinCDSCurve

This is a curve that has been calibrated to fit the market term structure of CDS contracts given a recovery rate assumption and a FinLiborCurve discount curve. It also contains a LiborCurve object for discounting. It has methods for fitting the curve and also for extracting survival probabilities.

6.2 FinCDS

Class: FinCDS(object)

A class which manages a 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

Functions

__init__

Create a CDS from the step-in date, maturity date and coupon

```
def __init__(self,
             stepInDate: FinDate, #  FinDate is when protection starts (usually T
                 +1)
```

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 on a specific valuation date given an issuer curve and a contract recovery rate.

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

creditDV01

Calculation of the change in the value of the CDS contract for a one basis point change in the level of the CDS curve.

```
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

Value of the contract on the settlement date including accrued interest.

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

cleanPrice

Value of the CDS contract excluding accrued interest.

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

riskyPV01_OLD

RiskyPV01 of the contract using the OLD method.

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

accruedDays

Number of days between the previous coupon and the currrent step in date.

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

accruedInterest

Calculate the amount of accrued interest that has accrued from the previous coupon date (PCD) to the stepInDate 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

The riskyPV01 is the present value of a risky one dollar paid on the premium leg of a CDS contract.

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

premiumLegPV

Value of the premium leg of a CDS.

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

parSpread

Breakeven CDS coupon that would make the value of the CDS contract equal to zero.

```
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):
```

__repr__

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

```
def __repr__(self):
```

print

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

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

printFlows

PLEASE ADD A FUNCTION DESCRIPTION

```
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):
```

6.3 FinCDSBasket

Class: FinCDSBasket(object)

Data Members

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

Functions

__init__

maturityDate: FinDate,

```
def __init__(self,
             stepInDate: FinDate,
```

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. This depends on the issuer curves and correlation matrix.

```
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):
```

6.4 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__

PLEASE ADD A FUNCTION DESCRIPTION

```
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. This function supports vectorisation.

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

df

Extract the discount factor from the underlying Libor curve. This function supports vectorisation.

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

buildCurve

Construct the CDS survival curve from a set of CDS contracts

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

fwd

Calculate the instantaneous forward rate at the forward date dt using the numerical derivative.

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

fwdRate

Calculate the forward rate according between dates date1 and date2 according to the specified day count convention.

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

zeroRate

Calculate the zero rate to date dt in the chosen compounding frequency where -1 is continuous is the default.

```
def zeroRate(self,
             dt,
             frequencyType=FinFrequencyTypes.CONTINUOUS):
```

__repr__

Print out the details of the survival probability curve.

```
def __repr__(self):
```

print

Simple print function for backward compatibility.

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

uniformToDefaultTime

Fast mapping of a uniform random variable to a default time given a survival probability curve.

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

f

Function that returns zero when the survival probability that gives a zero value of the CDS has been determined.

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

6.5 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__

maturityDate: FinDate,

```
def __init__(self,
             expiryDate: FinDate,
```

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):
```

6.6 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 = FinFrequencyTypes.QUARTERLY,
```

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 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=1e-6):
```

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=1e-6,
                              maxIterations=100):
```

6.7 FinCDSOption

Class: FinCDSOption()

Data Members

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

Functions

__init__

maturityDate: FinDate,

```
def __init__(self,
             expiryDate: FinDate,
```

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):
```

6.8 FinCDSTranche

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__

maturityDate: FinDate,

```
def __init__(self,
             stepInDate: FinDate,
```

valueBC

PLEASE ADD A FUNCTION DESCRIPTION

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

Chapter 7

financepy.products.bonds

7.1 Introduction

This folder contains a suite of bond-related functionality across a set of files and classes. They are as follows:

- 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.

Conventions

- All interest rates are expressed as a fraction of 1. So 3
- All notionals of bond positions are given in terms of a notional amount.
- All bond prices are based on a notional of 100.0.
- The face of a derivatives position is the size of the underlying position.

Bond Curves

These modules create a family of curve types related to the term structures of interest rates. There are two basic types of curve:

1. Best fit yield curves fitting to bond prices which are used for interpolation. A range of curve shapes from polynomials to B-Splines is available.
2. Discount curves that can be used to present value a future cash flow. These differ from best fits curves in that they exactly refit the prices of bonds or CDS. The different discount curves are created by calibrating to different instruments. They also differ in terms of the term structure shapes they can have. Different shapes have different impacts in terms of locality on risk management performed using these different curves. There is often a trade-off between smoothness and locality.

FinBondYieldCurve

This module describes a curve that is fitted to bond yields calculated from bond market prices supplied by the user. The curve is not guaranteed to fit all of the bond prices exactly and a least squares approach is used. A number of fitting forms are provided which consist of

- Polynomial
- Nelson-Siegel
- Nelson-Siegal-Svensson
- Cubic B-Splines

This fitted curve cannot be used for pricing as yields assume a flat term structure. It can be used for fitting and interpolating yields off a nicely constructed yield curve interpolation curve.

FinCurveFitMethod

This module sets out a range of curve forms that can be fitted to the bond yields. These includes a number of parametric curves that can be used to fit yield curves. These include:

- Polynomials of any degree
- Nelson-Siegel functional form.
- Nelson-Siegel-Svensson functional form.
- B-Splines

7.2 FinBond

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
- _frequency
- _face
- _par
- _settlementDate
- _accruedInterest
- _accruedDays
- _alpha
- _flowDates

Functions

__init__

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

```
def __init__(self,
             maturityDate: FinDate,
```

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):
```

principal

Calculate the principal value of the bond based on the face amount from its discount margin and making assumptions about the future Libor rates.

```
def principal(self,
              settlementDate,
              y,
              convention):
```

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):
```

cleanValueFromDiscountCurve

Calculate the clean bond value using some discount curve to present-value the bonds cashflows back to the curve anchor date and not to the settlement date.

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

valueBondUsingDiscountCurve

Calculate the bond *value* using some discount curve to PV the bonds cashflows to the curve anchor date. The anchor of the discount curve should be on the valuation date and so be 0-3 days before the settlement of the bond. This is not the same as the full price which is only the correct price on the settlement date of the bond which may be in the future.

```
def valueBondUsingDiscountCurve(self,
                                settlementDate,
                                discountCurve,
                                verbose=False):
```

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):
```

calcAccruedInterest

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

```
def calcAccruedInterest(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=FinBusDayAdjustTypes.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):
```

__repr__

s = labelToString("MATURITY DATE", self._maturityDate)

```
def __repr__(self):
```

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):
```

7.3 FinBondAnnuity

Class: FinBondAnnuity(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 with no principal repayment. Dates are then adjusted according to a specified calendar.

Data Members

- _maturityDate
- _coupon
- _frequencyType
- _frequency
- _calendarType
- _busDayAdjustType
- _dateGenRuleType
- _dayCountConventionType
- _face
- _par
- _settlementDate
- _accruedInterest
- _accruedDays
- _alpha
- _flowDates

Functions

__init__

coupon: float,

```
def __init__(self,
             maturityDate: FinDate,
```

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):
```

calculateFlowDatesPayments

PLEASE ADD A FUNCTION DESCRIPTION

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

_calcAccruedInterest

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

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

printFlows

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

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

__repr__

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

```
def __repr__(self):
```

print

Simple print function for backward compatibility.

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

7.4 FinBondConvertible

Class: FinBondConvertible(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
- _accrued
- _alpha
- _accruedDays

Functions

__init__

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

```
def __init__(self,
             maturityDate: FinDate,  # bond maturity date
```

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):
```

__repr__

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

```
def __repr__(self):
```

print

Simple print function for backward compatibility.

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

valueConvertible

PLEASE ADD A FUNCTION DESCRIPTION

```
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

n1, n2 = array.shape

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

7.5 FinBondEmbeddedOption

Enumerated Type: FinBondModelTypes

- BLACK
- HO_LEE
- HULL_WHITE
- BLACK_KARASINSKI

Enumerated Type: FinBondOptionTypes

- EUROPEAN_CALL
- EUROPEAN_PUT
- AMERICAN_CALL
- AMERICAN_PUT

Class: FinBondEmbeddedOption(object)

Data Members

- _maturityDate
- _coupon
- _frequencyType
- _accrualType
- _bond
- _callDates
- _callPrices
- _putDates
- _putPrices
- _face

Functions

__init__

Create a FinBondEmbeddedOption object with a maturity date, coupon and all of the bond inputs.

```
def __init__(self,
             maturityDate: FinDate,  # FinDate
```

value

Value the bond that settles on the specified date that can have both embedded call and put options. This is done using the specified model and a discount curve.

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

__repr__

PLEASE ADD A FUNCTION DESCRIPTION

```
def __repr__(self):
```

print

print(self)

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

7.6 FinBondFRN

Class: FinBondFRN(object)

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

Data Members

- _maturityDate
- _quotedMargin
- _frequencyType
- _accrualType
- _frequency
- _face
- _par
- _settlementDate
- _accruedInterest
- _accruedDays
- _flowDates

Functions

__init__

Create FinFloatingRateNote object given its maturity date, its quoted margin, coupon frequency, accrual type. Face is the size of the position and par is the notional on which price is quoted.

```
def __init__(self,
             maturityDate: FinDate,
```

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,
                                resetLibor,
                                currentLibor,
                                futureLibor,
                                dm):
```

principal

Calculate the clean trade price of the bond based on the face amount from its discount margin and making assumptions about the future Libor rates.

```
def principal(self,
              settlementDate,
              resetLibor,
              currentLibor,
              futureLibor,
              dm):
```

dollarRateDuration

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

```
def dollarRateDuration(self,
                       settlementDate,
                       resetLibor,
                       currentLibor,
                       futureLibor,
                       dm):
```

dollarCreditDuration

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

```
def dollarCreditDuration(self,
                         settlementDate,
                         resetLibor,
                         currentLibor,
                         futureLibor,
                         dm):
```

macauleyRateDuration

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

```
def macauleyRateDuration(self,
                         settlementDate,
                         resetLibor,
                         currentLibor,
                         futureLibor,
                         dm):
```

modifiedRateDuration

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

```
def modifiedRateDuration(self,
                         settlementDate,
                         resetLibor,
                         currentLibor,
                         futureLibor,
                         dm):
```

modifiedCreditDuration

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

```
def modifiedCreditDuration(self,
                           settlementDate,
                           resetLibor,
                           currentLibor,
                           futureLibor,
                           dm):
```

convexityFromDiscountMargin

Calculate the bond convexity from the discount margin using a numerical bump of size 1 basis point and taking second differences.

```
def convexityFromDiscountMargin(self,
                                settlementDate,
                                resetLibor,
                                currentLibor,
                                futureLibor,
                                dm):
```

cleanPriceFromDiscountMargin

Calculate the bond clean price from the yield.

```
def cleanPriceFromDiscountMargin(self,
                                 settlementDate,
                                 resetLibor,
                                 currentLibor,
                                 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,
                   resetLibor,
                   currentLibor,
                   futureLibor,
                   cleanPrice):
```

calcAccruedInterest

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

```
def calcAccruedInterest(self, settlementDate, resetLibor):
```

printFlows

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

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

__repr__

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

```
def __repr__(self):
```

print

Simple print function for backward compatibility.

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

f

Function used to do solve root search in DM calculation

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

7.7 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__

firstDeliveryDate: FinDate,

```
def __init__(self,
             tickerName: str,
```

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

The total invoice amount paid to take delivery of bond.

```
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):
```

__repr__

PLEASE ADD A FUNCTION DESCRIPTION

```
def __repr__(self):
```

print

Simple print function for backward compatibility.

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

7.8 FinBondMarket

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):
```

7.9 FinBondMortgage

Enumerated Type: FinBondMortgageTypes

- REPAYMENT
- INTEREST_ONLY

Class: FinBondMortgage(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

Functions

__init__

Create the mortgage using start and end dates and principal.

```
def __init__(self,
             startDate: FinDate,
```

repaymentAmount

Determine monthly repayment amount based on current zero rate.

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

generateFlows

Generate the bond flow amounts.

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

printLeg

print("START DATE:", self._startDate)

```
def printLeg(self):
```

__repr__

s = labelToString("START DATE", self._startDate)

```
def __repr__(self):
```

print

Simple print function for backward compatibility.

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

7.10 FinBondOption

Enumerated Type: FinBondModelTypes

- BLACK
- HO_LEE
- HULL_WHITE
- BLACK_KARASINSKI

Enumerated Type: FinBondOptionTypes

- EUROPEAN_CALL
- EUROPEAN_PUT
- AMERICAN_CALL
- AMERICAN_PUT

Class: FinBondOption()

Data Members

- _expiryDate
- _strikePrice
- _bond
- _optionType
- _face

Functions

__init__

expiryDate: FinDate,

```
def __init__(self,
             bond: FinBond,
```

value

Value the bond option using the specified model.

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

__repr__

s = labelToString("EXPIRY DATE", self._expiryDate)

```
def __repr__(self):
```

print

Simple print function for backward compatibility.

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

7.11 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: FinDate,
```

interpolatedYield

PLEASE ADD A FUNCTION DESCRIPTION

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

plot

Display yield curve.

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

__repr__

s = labelToString("SETTLEMENT DATE", self._settlementDate)

```
def __repr__(self):
```

print

Simple print function for backward compatibility.

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

7.12 FinBondYieldCurveModel

Class: FinCurveFitMethod()

class FinCurveFitMethod():

Data Members

No data members found.

Functions

Class: FinCurveFitPolynomial()

class FinCurveFitPolynomial():

Data Members

- _parentType
- _power

Functions

__init__

self._parentType = FinCurveFitMethod

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

_interpolatedYield

yld = np.polyval(self._coeffs, t)

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

__repr__

s = labelToString("Power", self._power)

```
def __repr__(self):
```

print

Simple print function for backward compatibility.

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

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

PLEASE ADD A FUNCTION DESCRIPTION

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

__repr__

s = labelToString("Beta1", self._beta1)

```
def __repr__(self):
```

print

Simple print function for backward compatibility.

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

Class: FinCurveFitNelsonSiegelSvensson()

class FinCurveFitNelsonSiegelSvensson():

Data Members

- _parentType
- _beta1
- _beta2

- _beta3
- _beta4
- _tau1
- _tau2
- _bounds

Functions

__init__

Create object to store calibration and functional form of NSS parametric fit.

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

_interpolatedYield

PLEASE ADD A FUNCTION DESCRIPTION

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

__repr__

s = labelToString(”Beta1”, self._beta1)

```
def __repr__(self):
```

print

Simple print function for backward compatibility.

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

Class: FinCurveFitBSpline()

class FinCurveFitBSpline():

Data Members

- _parentType
- _power
- _knots
- _spline

Functions

__init__

self._parentType = FinCurveFitMethod

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

_interpolatedYield

t = np.maximum(t, 1e-10)

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

__repr__

s = labelToString("Power", self._power)

```
def __repr__(self):
```

print

Simple print function for backward compatibility.

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

7.13 FinBondZeroCurve

Class: FinBondZeroCurve(FinDiscountCurve)

Data Members

- _settlementDate
- _valuationDate
- _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,
             valuationDate,
             bonds,
             cleanPrices,
             interpMethod=FinInterpMethods.FLAT_FORWARDS):
```

bootstrapZeroRates

PLEASE ADD A FUNCTION DESCRIPTION

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

zeroRate

Calculate the zero rate to maturity date.

```
def zeroRate(self, dt, frequencyType=FinFrequencyTypes.CONTINUOUS):
```

df

t = inputTime(dt, self)

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

survProb

t = inputTime(dt, self)

```
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):
```

__repr__

header = "TIMES,DISCOUNT FACTORS"

```
def __repr__(self):
```

print

Simple print function for backward compatibility.

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

f

curve = args[0]

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

Chapter 8

financepy.products.libor

8.1 Introduction

Libor Products

This folder contains a set of Libor-related products. More recently with the demise of Libor these are known as Ibor products. It includes:

FinInterestRateFuture

This is a class to handle interest rate futures contracts. This is an exchange-traded contract to receive or pay Libor on a specified future date. It can be used to build the Liboir term structure.

FinLiborCapFloor

This is a contract to buy a sequence of calls or puts on Libor over a period at a strike agreed today.

FinLiborDeposit

This is the basic Libor instrument in which a party borrows an amount for a specified term and rate unsecured.

FinLiborFRA

This is a class to manage Forward Rate Agreements (FRAs) in which one party agrees to lock in a forward Libor rate.

FinLiborSwap

This is a contract to exchange fixed rate coupons for floating Libor rates. This class has functionality to value the swap contract and to calculate its risk.

FinLiborSwaption

This is a contract to buy or sell an option on a swap. The model includes code that prices a payer or receiver swaption.

FinOIS

This is a contract to exchange the daily compounded Overnight index swap rate for a fixed rate agreed at contract initiation.

FinLiborCurve

This is a discount curve that is extracted by bootstrapping a set of Libor deposits, Libor FRAs and Libor swap prices. The internal representation of the curve are discount factors on each of the deposit, FRA and swap maturity dates. Between these dates, discount factors are interpolated according to a specified scheme - see below.

8.2 FinLiborBermudanSwaption

8.3 FinLiborCallableSwap

8.4 FinLiborCapFloor

Enumerated Type: FinLiborCapFloorTypes

- CAP
- FLOOR

Enumerated Type: FinLiborCapFloorModelTypes

- BLACK
- SHIFTED_BLACK
- SABR

Class: FinLiborCapFloor()

Class for Caps and Floors. These are contracts which observe a Libor reset L on a future start date and then make a payoff at the end of the Libor period which is Max[L-K,0] for a cap and Max[K-L,0] for a floor. This is then day count adjusted for the Libor period and then scaled by the contract notional to produce a valuation. A number of models can be selected from.

Data Members

- _calendarType
- _busDayAdjustType
- _startDate
- _maturityDate
- _optionType
- _strikeRate
- _lastFixing
- _frequencyType
- _dayCountType
- _notional
- _dateGenRuleType
- _valuationDate
- _dayCounter
- _capFloorLetDates
- _capFloorLetValues

Functions

__init__

Initialise FinLiborCapFloor object.

```
def __init__(self,
             startDate: FinDate,
```

_generateDates

PLEASE ADD A FUNCTION DESCRIPTION

```
def _generateDates(self):
```

value

Value the cap or floor using the chosen model which specifies the volatility of the Libor rate to the cap start date.

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

valueCapletFloorLet

Value the caplet or floorlet using a specific model.

```
def valueCapletFloorLet(self,
                        valuationDate,
                        capletStartDate,
                        capletEndDate,
                        liborCurve,
                        model):
```

printLeg

Prints the cap floor amounts.

```
def printLeg(self):
```

__repr__

s = labelToString("START DATE", self._startDate)

```
def __repr__(self):
```

print

print(self)

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

8.5 FinLiborConventions

Class: FinLiborConventions()

class FinLiborConventions():

Data Members

No data members found.

Functions

__init__

indexName: str = ”LIBOR”):

```
def __init__(self,
             currencyName: str,
```

8.6 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 1$paidhasapresentvalueof$1. This class inherits from FinDiscCurve so has all of the methods that class has.

Data Members

- _name
- _valuationDate
- _interpMethod
- _usedDeposits
- _usedFRAs
- _usedSwaps
- _times
- _discountFactors

Functions

__init__

PLEASE ADD A FUNCTION DESCRIPTION

```
def __init__(self,
             name,
             valuationDate,
             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):
```

checkRefits

PLEASE ADD A FUNCTION DESCRIPTION

```
def checkRefits(self):
```

__repr__

Print out the details of the Libor curve.

```
def __repr__(self):
```

print

Simple print function for backward compatibility.

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

f

Root search objective function for swaps

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

g

Root search objective function for swaps

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

8.7 FinLiborDeposit

Class: FinLiborDeposit(object)

A Libor deposit is an agreement to borrow money interbank at the Libor fixing rate starting on the settlement date and repaid on the maturity date with the interest amount calculated according to a day count convention and dates calculated according to a calendar and business day adjustment rule.

Data Members

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

Functions

__init__

Create a Libor deposit object which takes the settlement date when the amount of notional is borrowed, the deposit rate, the day count convention used to calculate the interest paid and a calendar and a business day adjustment method if dates fall on holidays.

```
def __init__(self,
             settlementDate: FinDate,
```

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 an existing Libor Deposit contract given a valuation date and a Libor curve. This is simply the PV of the future repayment plus interest discounted on the current Libor curve.

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

printFlows

Print the date and size of the future repayment.

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

__repr__

Print the contractual details of the Libor deposit.

```
def __repr__(self):
```

print

print(self)

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

8.8 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
- _busDayAdjustType
- _startDate
- _maturityDate
- _fraRate
- _payFixedRate
- _dayCountType
- _notional

Functions

__init__

Create a Forward Rate Agreeement object.

```
def __init__(self,
             startDate: FinDate,  # The date the FRA starts to accrue
```

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, valuationDate, 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):
```

printFlows

Determine the value of the Deposit given a Libor curve.

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

__repr__

s = labelToString("START ACCD DATE", self._startDate)

```
def __repr__(self):
```

print

print(self)

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

8.9 FinLiborFuture

Class: FinLiborFuture(object)

Data Members

- _deliveryDate
- _endOfInterestPeriod
- _lastTradingDate
- _accrualType
- _contractSize

Functions

__init__

Create an interest rate futures contract.

```
def __init__(self,
             todayDate: FinDate,
```

toFRA

Convert the futures contract to a FinLiborFRA object so it can be used to boostrap a Libor curve. For this we need to adjust the futures rate using the convexity correction.

```
def toFRA(self, futuresPrice, convexity):
```

futuresRate

Calculate implied futures rate from the futures price.

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

FRARate

Convert futures price and convexity to a FRA rate using the BBG negative convexity (in percent). This is then divided by 100 before being added to the futures rate.

```
def FRARate(self, futuresPrice, convexity):
```

convexity

Calculation of the convexity adjustment between FRAs and interest rate futures using the Hull-White model as described in technical note in link below: http://www-2.rotman.utoronto.ca/ hull/TechnicalNotes/TechnicalNote1.pdf NOTE THIS DOES NOT APPEAR TO AGREE WITH BLOOMBERG!! INVESTIGATE.

```
def convexity(self, valuationDate, volatility, meanReversion):
```

__repr__

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

```
def __repr__(self):
```

8.10 FinLiborLMMProducts

Class: FinLiborLMMProducts()

Data Members

- _startDate
- _gridDates
- _floatDayCountType
- _accrualFactors
- _numForwards
- _fwds
- _numPaths
- _numeraireIndex
- _useSobol
- _forwardCurve
- _volCurves
- _correlationMatrix
- _modelType

Functions

__init__

Create a European-style swaption by defining the exercise date of the swaption, and all of the details of the underlying interest rate swap including the fixed coupon and the details of the fixed and the floating leg payment schedules.

```
def __init__(self,
             settlementDate: FinDate,
```

simulate1F

Run the one-factor simulation of the evolution of the forward Libors to generate and store all of the Libor forward rate paths.

```
def simulate1F(self,
               discountCurve,
               volCurve: FinLiborCapVolCurve,
```

simulateMF

Run the simulation to generate and store all of the Libor forward rate paths. This is a multi-factorial version so the user must input a numpy array consisting of a column for each factor and the number of rows must equal the number of grid times on the underlying simulation grid. CHECK THIS.

```
def simulateMF(self,
               discountCurve,
               numFactors: int,
```

simulateNF

Run the simulation to generate and store all of the Libor forward rate paths using a full factor reduction of the fwd-fwd correlation matrix using Cholesky decomposition.

```
def simulateNF(self,
               discountCurve,
               volCurves: list,
```

valueSwaption

Value a swaption in the LMM model using simulated paths of the forward curve. This relies on pricing the fixed leg of the swap and assuming that the floating leg will be worth par. As a result we only need simulate Libors with the frequency of the fixed leg.

```
def valueSwaption(self,
                  settlementDate: FinDate,
```

valueCapFloor

Value a cap or floor in the LMM.

```
def valueCapFloor(self,
                  settlementDate: FinDate,
```

__repr__

Function to allow us to print the swaption details.

```
def __repr__(self):
```

print

Alternative print method.

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

8.11 FinLiborSwap

Class: FinLiborSwap(object)

Data Members

- _startDate
- _maturityDate
- _notional
- _payFixedLeg
- _fixedCoupon
- _floatSpread
- _fixedFrequencyType
- _floatFrequencyType
- _fixedDayCountType
- _floatDayCountType
- _payFixedFlag
- _calendarType
- _busDayAdjustType
- _dateGenRuleType
- _lastPaymentDate
- _firstFixingRate
- _valuationDate
- _adjustedFixedDates
- _adjustedFloatDates
- _fixedStartIndex
- _floatStartIndex
- _fixedFlows
- _floatFlows

Functions

__init__

Create an interest rate swap contract.

```
def __init__(self,
             startDate: FinDate,  # This is typically T+2 on a new swap
```

value

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

```
def value(self,
          valuationDate,
          discountCurve,
          indexCurve,
          firstFixingRate=None,
          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):
```

fixedDates

return a vector of the fixed leg payment dates

```
def fixedDates(self):
```

floatDates

return a vector of the fixed leg payment dates

```
def floatDates(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,  # IS THIS THE SETTLEMENT DATE ????
                  discountCurve,
                  indexCurve,
                  firstFixingRate=None,
                  principal=0.0):
```

printFixedLeg

Prints the fixed leg amounts.

```
def printFixedLeg(self):
```

printFloatLeg

Prints the floating leg amounts.

```
def printFloatLeg(self):
```

__repr__

s = labelToString("START DATE", self._startDate)

```
def __repr__(self):
```

print

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

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

8.12 FinLiborSwaption

Enumerated Type: FinLiborSwaptionTypes

- PAYER
- RECEIVER

Enumerated Type: FinLiborSwaptionModelTypes

- BLACK
- SABR

Class: FinLiborSwaption()

Data Members

- _settlementDate
- _exerciseDate
- _maturityDate
- _swaptionType
- _fixedCoupon
- _fixedFrequencyType
- _fixedDayCountType
- _notional
- _floatFrequencyType
- _floatDayCountType
- _calendarType
- _busDayAdjustType
- _dateGenRuleType
- _pv01
- _fwdSwapRate
- _forwardDf
- _underlyingSwap

Functions

__init__

Create a European-style swaption by defining the exercise date of the swaption, and all of the details of the underlying interest rate swap including the fixed coupon and the details of the fixed and the floating leg payment schedules.

```
def __init__(self,
             settlementDate: FinDate,
```

value

Valuation of a Libor European-style swaption using a choice of models on a specified valuation date.

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

cashSettledValue

Valuation of a Libor European-style swaption using a cash settled approach which is a market convention that used Blacks model and that discounts the future payments at a specified swap rate.

```
def cashSettledValue(self,
                     valuationDate,
                     discountCurve,
                     model):
```

printSwapFixedLeg

PLEASE ADD A FUNCTION DESCRIPTION

```
def printSwapFixedLeg(self):
```

printSwapFloatLeg

PLEASE ADD A FUNCTION DESCRIPTION

```
def printSwapFloatLeg(self):
```

__repr__

Function to allow us to print the swaption details.

```
def __repr__(self):
```

print

Alternative print method.

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

8.13 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
- _adjustedFixedDates
- _adjustedFloatDates

Functions

__init__

Create OIS object.

```
def __init__(self,
             startDate: FinDate,
```

generatePaymentDates

PLEASE ADD A FUNCTION DESCRIPTION

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

generateFixedLegFlows

PLEASE ADD A FUNCTION DESCRIPTION

```
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

PLEASE ADD A FUNCTION DESCRIPTION

```
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

print(”StartDate:”, self._startDate)

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

Chapter 9

financepy.products.fx

9.1 Introduction

FX Derivatives

Overview

These modules price and produce the sensitivity measures needed to hedge a range of FX Options and other derivatives with an FX underlying.

FX Forwards

Calculate the price and breakeven forward FX Rate of an FX Forward contract.

FX Vanilla Option

FX Option

This is a class from which other classes inherit and is used to perform simple perturbatory calculation of option Greeks.

FX Barrier Options

FX Basket Options

FX Digital Options

FX Fixed Lookback Option

FX Float Lookback Option

FX Rainbow Option

FX Variance Swap

9.2 FinFXBarrierOption

Enumerated Type: FinFXBarrierTypes

- 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: FinFXBarrierOption(FinFXOption)

class FinFXBarrierOption(FinFXOption):

Data Members

- _expiryDate
- _strikeFXRate
- _barrierLevel
- _numObservationsPerYear
- _optionType
- _notional
- _notionalCurrency

Functions

__init__

Create FX Barrier option product. This is an option that cancels if the FX rate crosses a barrier during the life of the option.

```
def __init__(self,
             expiryDate: FinDate,
```

value

Value FX Barrier Option using Black-Scholes model with closed-form analytical models.

```
def value(self,
          valueDate,
          spotFXRate,
          domDiscountCurve,
          forDiscountCurve,
          model):
```

valueMC

Value the FX Barrier Option using Monte Carlo.

```
def valueMC(self,
            valueDate,
            spotFXRate,
            domInterestRate,
            processType,
            modelParams,
            numAnnSteps=552,
            numPaths=5000,
            seed=4242):
```

9.3 FinFXBasketOption

Class: FinFXBasketOption(FinFXOption)

Class to manage FX Basket Option which is an option on a portfolio of FX rates.

Data Members

- _expiryDate
- _strikePrice
- _optionType
- _numAssets
- _notional

Functions

__init__

Create FX Basket Option with expiry date, strike price, option type, number of assets and notional.

```
def __init__(self,
             expiryDate: FinDate,
```

validate

Check that there is an input for each asset in the basket.

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

value

Value an FX Basket Option using Black-Scholes closed-form model which takes into account mean and variance of underlying.

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

valueMC

Value the FX Basket Option using Monte Carlo.

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

9.4 FinFXDigitalOption

Class: FinFXDigitalOption()

class FinFXDigitalOption():

Data Members

- _expiryDate
- _strikePrice
- _currencyPair
- _optionType
- _forName
- _domName

Functions

__init__

Create the FX Digital Option object. Inputs include expiry date, strike, currency pair, option type (call or put), notional and the currency of the notional. And adjustment for spot days is enabled. All currency rates must be entered in the price in domestic currency of one unit of foreign. And the currency pair should be in the form FORDOM where FOR is the foreign currency pair currency code and DOM is the same for the domestic currency.

```
def __init__(self,
             expiryDate: FinDate,
```

value

Valuation of a digital option using Black-Scholes model. This allows for 4 cases - first upper barriers that when crossed pay out cash (calls) and lower barriers than when crossed from above cause a cash payout (puts) PLUS the fact that the cash payment can be in domestic or foreign currency.

```
def value(self,
          valueDate,
          spotFXRate,  # ONE UNIT OF FOREIGN IN DOMESTIC CCY
          domDiscountCurve,
          forDiscountCurve,
          model):
```

9.5 FinFXFixedLookbackOption

Enumerated Type: FinFXFixedLookbackOptionTypes

- FIXED_CALL
- FIXED_PUT

Class: FinFXFixedLookbackOption()

Data Members

- _expiryDate
- _optionType
- _optionStrike

Functions

__init__

Create option with expiry date, option type and the option strike

```
def __init__(self,
             expiryDate: FinDate,
```

value

Value FX Fixed Lookback Option using Black Scholes model and analytical formulae.

```
def value(self,
          valueDate,
          stockPrice,
          domesticCurve,
          foreignCurve,
          volatility,
          stockMinMax):
```

valueMC

Value FX Fixed Lookback option using Monte Carlo.

```
def valueMC(self,
            valueDate,
            spotFXRate,  # FORDOM
            domesticCurve,
            foreignCurve,
            volatility,
            spotFXRateMinMax,
            numPaths=10000,
            numStepsPerYear=252,
            seed=4242):
```

9.6 FinFXFloatLookbackOption

Enumerated Type: FinFloatLookbackOptionTypes

- FLOATING_CALL
- FLOATING_PUT

Class: FinFloatLookbackOption(FinEquityOption)

class FinFloatLookbackOption(FinEquityOption):

Data Members

- _expiryDate
- _optionType

Functions

__init__

optionType: FinFloatLookbackOptionTypes):

```
def __init__(self,
             expiryDate: FinDate,
```

value

PLEASE ADD A FUNCTION DESCRIPTION

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

valueMC

PLEASE ADD A FUNCTION DESCRIPTION

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

```
        seed=4242):
```

9.7 FinFXForward

Class: FinFXForward()

Data Members

- _expiryDate
- _deliveryDate
- _strikeFXRate
- _currencyPair
- _notional
- _notionalCurrency
- _spotDays
- _notional_dom
- _notional_for
- _cash_dom
- _cash_for

Functions

__init__

Creates a FinFXForward which allows the owner to buy the FOR against the DOM currency at the strikeFXRate and to pay it in the notional currency.

```
def __init__(self,
             expiryDate: FinDate,
```

value

Calculate the value of an FX forward contract where the current FX rate is the spotFXRate.

```
def value(self,
          valueDate,
          spotFXRate,  # PRICE OF ONE UNIT OF FOREIGN IN DOMESTIC CCY
          domDiscountCurve,
          forDiscountCurve):
```

forward

Calculate the FX Forward rate that makes the value of the FX contract equal to zero.

```
def forward(self,
            valueDate,
            spotFXRate,  # PRICE OF ONE UNIT OF FOREIGN IN DOMESTIC CCY
            domDiscountCurve,
            forDiscountCurve):
```

9.8 FinFXMktConventions

Enumerated Type: FinFXATMMethod

- SPOT
- FWD
- FWD_DELTA_NEUTRAL
- FWD_DELTA_NEUTRAL_PREM_ADJ

Enumerated Type: FinFXDeltaMethod

- SPOT_DELTA
- FORWARD_DELTA
- SPOT_DELTA_PREM_ADJ
- FORWARD_DELTA_PREM_ADJ

Class: FinFXRate()

class FinFXRate():

Data Members

- _ccy1
- _ccy2

Functions

__init__

PLEASE ADD A FUNCTION DESCRIPTION

```
def __init__(self,
             ccy1,
             ccy2,
             rate):
```

9.9 FinFXModelTypes

Class: FinFXModel(object)

class FinFXModel(object):

Data Members

No data members found.

Functions

__init__

pass

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

Class: FinFXModelBlackScholes(FinFXModel)

Data Members

- _parentType
- _modelType
- _volatility
- _implementation

Functions

__init__

Create Black Scholes FX model object which holds volatility.

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

Class: FinFXModelHeston(FinFXModel)

Data Members

- _modelType
- _volatility
- _meanReversion
- _implementation

Functions

__init__

Create Heston FX Model which takes in volatility and mean reversion.

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

Class: FinFXModelSABR(FinFXModel)

Data Members

- _modelType
- _alpha
- _beta
- _rho
- _nu
- _implementation

Functions

__init__

Create FX Model SABR which takes alpha, beta, rho, nu and volatility as parameters.

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

9.10 FinFXOption

Class: FinFXOption(object)

Data Members

No data members found.

Functions

delta

Calculate the option delta (FX rate sensitivity) by adding on a small bump and calculating the change in the option price.

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

gamma

Calculate the option gamma (delta sensitivity) by adding on a small bump and calculating the change in the option delta.

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

vega

Calculate the option vega (volatility sensitivity) by adding on a small bump and calculating the change in the option price.

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

theta

Calculate the option theta (calendar time sensitivity) by moving forward one day and calculating the change in the option price.

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

rho

Calculate the option rho (interest rate sensitivity) by perturbing the discount curve and revaluing.

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

9.11 FinFXRainbowOption

Enumerated Type: FinFXRainbowOptionTypes

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

Class: FinRainbowOption(FinEquityOption)

class FinRainbowOption(FinEquityOption):

Data Members

- _expiryDate
- _payoffType
- _payoffParams
- _numAssets

Functions

__init__

payoffType: FinFXRainbowOptionTypes,

```
def __init__(self,
             expiryDate: FinDate,
```

validate

PLEASE ADD A FUNCTION DESCRIPTION

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

validatePayoff

PLEASE ADD A FUNCTION DESCRIPTION

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

value

PLEASE ADD A FUNCTION DESCRIPTION

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

valueMC

PLEASE ADD A FUNCTION DESCRIPTION

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

payoffValue

PLEASE ADD A FUNCTION DESCRIPTION

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

valueMCFast

PLEASE ADD A FUNCTION DESCRIPTION

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

9.12 FinFXVanillaOption

Class: FinFXVanillaOption()

This is a class for an FX Option trade. It permits the user to calculate the price of an FX Option trade which can be expressed in a number of ways depending on the investor or hedgers currency. It aslo allows the calculation of the options delta in a number of forms as well as the various Greek risk sensitivies.

Data Members

- _expiryDate
- _deliveryDate
- _strikeFXRate
- _currencyPair
- _premCurrency
- _notional
- _optionType
- _spotDays

Functions

__init__

Create the FX Vanilla Option object. Inputs include expiry date, strike, currency pair, option type (call or put), notional and the currency of the notional. And adjustment for spot days is enabled. All currency rates must be entered in the price in domestic currency of one unit of foreign. And the currency pair should be in the form FORDOM where FOR is the foreign currency pair currency code and DOM is the same for the domestic currency.

```
def __init__(self,
             expiryDate: FinDate,
```

value

This function calculates the value of the option using a specified model with the resulting value being in domestic i.e. ccy2 terms. Recall that Domestic = CCY2 and Foreign = CCY1 and FX rate is in price in domestic of one unit of foreign currency.

```
def value(self,
          valueDate,
          spotFXRate,  # ONE UNIT OF FOREIGN IN DOMESTIC CCY
          domDiscountCurve,
          forDiscountCurve,
          model):
```

delta_bump

Calculation of the FX option delta by bumping the spot FX rate by 1 cent of its value. This gives the FX spot delta. For speed we prefer to use the analytical calculation of the derivative given below.

```
def delta_bump(self,
               valueDate,
               spotFXRate,
               ccy1DiscountCurve,
               ccy2DiscountCurve,
               model):
```

delta

Calculation of the FX Option delta. There are several definitions of delta and so we are required to return a dictionary of values. The definitions can be found on Page 44 of Foreign Exchange Option Pricing by Iain Clark, published by Wiley Finance.

```
def delta(self,
          valueDate,
          spotFXRate,
          domDiscountCurve,
          forDiscountCurve,
          model):
```

gamma

This function calculates the FX Option Gamma using the spot delta.

```
def gamma(self,
          valueDate,
          spotFXRate,  # value of a unit of foreign in domestic currency
          domDiscountCurve,
          forDiscountCurve,
          model):
```

vega

This function calculates the FX Option Vega using the spot delta.

```
def vega(self,
         valueDate,
         spotFXRate,  # value of a unit of foreign in domestic currency
         domDiscountCurve,
         forDiscountCurve,
         model):
```

theta

This function calculates the time decay of the FX option.

```
def theta(self,
          valueDate,
```

```
            spotFXRate,  # value of a unit of foreign in domestic currency
            domDiscountCurve,
            forDiscountCurve,
            model):
```

impliedVolatility

This function determines the implied volatility of an FX option given a price and the other option details. It uses a one-dimensional Newton root search algorith to determine the implied volatility.

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

valueMC

Calculate the value of an FX Option using Monte Carlo methods. This function can be used to validate the risk measures calculated above or used as the starting code for a model exotic FX product that cannot be priced analytically. This function uses Numpy vectorisation for speed of execution.

```
def valueMC(self,
            valueDate,
            spotFXRate,
            domDiscountCurve,
            forDiscountCurve,
            model,
            numPaths=10000,
            seed=4242):
```

f

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

fvega

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

g

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

solveForStrike

This function determines the implied strike of an FX option given a delta and the other option details. It uses a one-dimensional Newton root search algorith to determine the strike that matches an input volatility.

```
def solveForStrike(valueDate,
                   vanillaOption,
                   spotFXRate,
                   domDiscountCurve,
```

```
forDiscountCurve,
delta,
deltaType,
volatility):
```

9.13 FinFXVarianceSwap

Class: FinFXVarianceSwap(object)

Data Members

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

Functions

__init__

Create variance swap contract.

```
def __init__(self,
             startDate: FinDate,
```

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

PLEASE ADD A FUNCTION DESCRIPTION

```
    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

PLEASE ADD A FUNCTION DESCRIPTION

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

9.14 FinFXVolatilitySmileDELETE

Chapter 10

financepy.models

10.1 Introduction

Models

Overview

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.

These modules hold all of the models used by FinancePy across asset classes.

The general philosophy is to separate where possible product and models so that these models have as little product knowledge as possible.

Also, Numba is used extensively, resulting in code speedups of between x 10 and x 100.

Generic Arbitrage-Free Models

There are the following arbitrage-free models:

- FinModelBlack is Black's model for pricing forward starting contracts (in the forward measure) assuming the forward is lognormally distributed.

- FinModelBlackShifted is Black's model for pricing forward starting contracts (in the forward measure) assuming the forward plus a shift is lognormally distributed. CHECK

- FinModelBachelier prices options assuming the underlying evolves according to a Gaussian (normal) process.

- FinSABR Model is a stochastic volatility model for forward values with 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.

- FinSABRShifted Model is a stochastic volatility model for forward value with 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.

The following asset-specific models have been implemented:

Equity Models

- FinHestonModel
- FinHestonModelProcess
- FinProcessSimulator

Interest Rate Models

Equilibrium Rate Models

There are two main short rate models.

- 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.

Arbitrage Free Rate Models

- 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.

Credit Models

- FinGaussianCopula1FModel is a Gaussian copula one-factor model. This class includes functions that calculate the portfolio loss distribution. This is numerical but deterministic.
- FinGaussianCopulaLHPModel is a Gaussian copula one-factor model in the limit that the number of credits tends to infinity. This is an asymptotic analytical solution.
- FinGaussianCopulaModel is a Gaussian copula model which is multifactor model. It has a Monte-Carlo implementation.
- FinLossDbnBuilder calculates the loss distribution.

- FinMertonCreditModel is a model of the firm as proposed by Merton (1974).

FX Models

10.2 FinGBMProcess

Class: FinGBMProcess()

class FinGBMProcess():

Data Members

No data members found.

Functions

getPaths

paths = getPaths(numPaths, numTimeSteps,

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

getPathsAssets

PLEASE ADD A FUNCTION DESCRIPTION

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

getPaths

PLEASE ADD A FUNCTION DESCRIPTION

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

getPathsAssets

PLEASE ADD A FUNCTION DESCRIPTION

```
def getPathsAssets(numAssets,
                   numPaths,
                   numTimeSteps,
                   t,
```

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

getAssets

PLEASE ADD A FUNCTION DESCRIPTION

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

10.3 FinMertonCreditModel

mertonCreditModelValues

PLEASE ADD A FUNCTION DESCRIPTION

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

10.4 FinModelBachelier

Class: FinModelBachelier()

Bacheliers Model which prices call and put options in the forward measure assuming the underlying rate follows a normal process.

Data Members

- _volatility

Functions

__init__

Create FinModel black using parameters.

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

value

Price a derivative using Bacheliers model which values in the forward measure following a change of measure.

```
def value(self,
          forwardRate,    # Forward rate F
          strikeRate,     # Strike Rate K
          timeToExpiry,   # Time to Expiry (years)
          df,             # Discount Factor to expiry date
          callOrPut):     # Call or put
```

__repr__

s = ”FINMODELBACHELIER”

```
def __repr__(self):
```

10.5 FinModelBlack

Class: FinModelBlack()

Blacks Model which prices call and put options in the forward measure according to the Black-Scholes equation.

Data Members

- _volatility
- _implementation
- _numSteps
- _seed
- _param1
- _param2

Functions

__init__

Create FinModel black using parameters.

```
def __init__(self, volatility, implementation=0):
```

value

Price a derivative using Blacks model which values in the forward measure following a change of measure.

```
def value(self,
          forwardRate,   # Forward rate F
          strikeRate,    # Strike Rate K
          timeToExpiry,  # Time to Expiry (years)
          df,            # Discount Factor to expiry date
          callOrPut):    # Call or put
```

__repr__

s = "FINMODELBLACK"

```
def __repr__(self):
```

10.6 FinModelBlackShifted

Class: FinModelBlackShifted()

Blacks Model which prices call and put options in the forward measure according to the Black-Scholes equation. This model also allows the distribution to be shifted to the negative in order to allow for negative interest rates.

Data Members

- _volatility
- _shift
- _implementation
- _numSteps
- _seed
- _param1
- _param2

Functions

__init__

Create FinModel black using parameters.

```
def __init__(self, volatility, shift, implementation=0):
```

value

Price a derivative using Blacks model which values in the forward measure following a change of measure.

```
def value(self,
          forwardRate,   # Forward rate
          strikeRate,    # Strike Rate
          timeToExpiry,  # time to expiry in years
          df,            # Discount Factor to expiry date
          callOrPut):    # Call or put
```

__repr__

s = "FINMODELBLACKSHIFTED"

```
def __repr__(self):
```

10.7 FinModelCRRTree

crrTreeVal

Value an American option using a Binomial Treee

```
def crrTreeVal(stockPrice,
               ccInterestRate,  # continuously compounded
               ccDividendRate,  # continuously compounded
               volatility,  # Black scholes volatility
               numStepsPerYear,
               timeToExpiry,
               optionType,
               strikePrice,
               isEven):
```

crrTreeValAvg

PLEASE ADD A FUNCTION DESCRIPTION

```
def crrTreeValAvg(stockPrice,
                  ccInterestRate,  # continuously compounded
                  ccDividendRate,  # continuously compounded
                  volatility,  # Black scholes volatility
                  numStepsPerYear,
                  timeToExpiry,
                  optionType,
                  strikePrice):
```

10.8 FinModelGaussianCopula

defaultTimesGC

Generate a matrix of default times by credit and trial using a Gaussian copula model using a full rank correlation matrix.

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

10.9 FinModelGaussianCopula1F

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

PLEASE ADD A FUNCTION DESCRIPTION

```
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):
```

10.10 FinModelGaussianCopulaLHP

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

PLEASE ADD A FUNCTION DESCRIPTION

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

expMinLK

PLEASE ADD A FUNCTION DESCRIPTION

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

LHPDensity

PLEASE ADD A FUNCTION DESCRIPTION

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

LHPAnalyticalDensityBaseCorr

PLEASE ADD A FUNCTION DESCRIPTION

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

LHPAnalyticalDensity

PLEASE ADD A FUNCTION DESCRIPTION

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

ExpMinLK

PLEASE ADD A FUNCTION DESCRIPTION

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

probLGreaterThanK

c = normpdf(P)

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

10.11 FinModelHeston

Enumerated Type: FinHestonNumericalScheme

- EULER
- EULERLOG
- QUADEXP

Class: FinModelHeston()

class FinModelHeston():

Data Members

- _v0
- _kappa
- _theta
- _sigma
- _rho

Functions

__init__

PLEASE ADD A FUNCTION DESCRIPTION

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

value_MC

PLEASE ADD A FUNCTION DESCRIPTION

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

value_Lewis

PLEASE ADD A FUNCTION DESCRIPTION

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

phi

k = k_in + 0.5 * 1j

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

phi_transform

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

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

integrand

k * x) * phi(k)) / (k**2 + 1.0 / 4.0)

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

value_Lewis_Rouah

PLEASE ADD A FUNCTION DESCRIPTION

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

f

k = k_in + 0.5 * 1j

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

value_Weber

PLEASE ADD A FUNCTION DESCRIPTION

```
def value_Weber(self,
                valueDate,
                option,
```

```
                    stockPrice,
                    interestRate,
                    dividendYield):
```

F

def integrand(u):

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

integrand

beta = b - 1j * rho * sigma * u

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

value_Gatheral

PLEASE ADD A FUNCTION DESCRIPTION

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

F

def integrand(u):

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

integrand

V = sigma * sigma

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

getPaths

PLEASE ADD A FUNCTION DESCRIPTION

```
def getPaths(
        s0,
        r,
        q,
        v0,
        kappa,
        theta,
        sigma,
        rho,
```

```
t,
dt,
numPaths,
seed,
scheme):
```

10.12 FinModelLHPlus

Class: LHPlusModel()

Large Homogenous Portfolio model with extra asset. Used for approximating full Gaussian copula.

Data Members

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

Functions

__init__

PLEASE ADD A FUNCTION DESCRIPTION

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

probLossGreaterThanK

Returns P(L¿K) where L is the portfolio loss given by model.

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

expMinLKIntegral

PLEASE ADD A FUNCTION DESCRIPTION

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

expMinLK

PLEASE ADD A FUNCTION DESCRIPTION

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

expMinLK2

PLEASE ADD A FUNCTION DESCRIPTION

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

trancheSurvivalProbability

PLEASE ADD A FUNCTION DESCRIPTION

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

10.13 FinModelLossDbnBuilder

indepLossDbnHeterogeneousAdjBinomial

PLEASE ADD A FUNCTION DESCRIPTION

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

portfolioGCD

PLEASE ADD A FUNCTION DESCRIPTION

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

indepLossDbnRecursionGCD

PLEASE ADD A FUNCTION DESCRIPTION

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

10.14 FinModelRatesBK

Class: FinModelRatesBK()

class FinModelRatesBK():

Data Members

- _a
- _sigma
- _numTimeSteps
- _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, sigma, a, numTimeSteps=100):
```

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, strike,
               face, couponTimes, couponFlows, americanExercise):
```

callablePuttableBond_Tree

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 callablePuttableBond_Tree(self,
                              couponTimes, couponFlows,
                              callTimes, callPrices,
                              putTimes, putPrices,
                              face):
```

buildTree

PLEASE ADD A FUNCTION DESCRIPTION

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

__repr__

Return string with class details.

```
def __repr__(self):
```

f

PLEASE ADD A FUNCTION DESCRIPTION

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

fprime

PLEASE ADD A FUNCTION DESCRIPTION

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

searchRoot

PLEASE ADD A FUNCTION DESCRIPTION

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

searchRootDeriv

PLEASE ADD A FUNCTION DESCRIPTION

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

americanBondOption_Tree_Fast

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

callablePuttableBond_Tree_Fast

Value a bond with embedded put and call options that can be exercised at any time over the specified list of put and call dates. Due to non-analytical bond price we need to extend tree out to bond maturity and take into account cash flows through time.

```
def callablePuttableBond_Tree_Fast(couponTimes, couponFlows,
                                   callTimes, callPrices,
                                   putTimes, putPrices, face,
                                   _sigma, _a, _Q,  # IS SIGMA USED ?
                                   _pu, _pm, _pd, _rt, _dt, _treeTimes,
                                   _dfTimes, _dfValues):
```

buildTreeFast

PLEASE ADD A FUNCTION DESCRIPTION

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

10.15 FinModelRatesCIR

Enumerated Type: FinCIRNumericalScheme

- EULER
- LOGNORMAL
- MILSTEIN
- KAHLJACKEL
- EXACT

Class: FinModelRatesCIR()

class FinModelRatesCIR():

Data Members

- _a
- _b
- _sigma

Functions

__init__

self._a = a

```
    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):
```

10.16 FinModelRatesHL

Class: FinModelRatesHL()

class FinModelRatesHL():

Data Members

- _sigma

Functions

__init__

Construct Ho-Lee model using single parameter of volatility. The dynamical equation is dr = theta(t) dt + sigma * dW. Any no-arbitrage fitting is done within functions below.

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

zcb

PLEASE ADD A FUNCTION DESCRIPTION

```
def zcb(self, rt1, t1, t2, discountCurve):
```

optionOnZCB

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

__repr__

Return string with class details.

```
def __repr__(self):
```

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 dollar of the discount factor.

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

10.17 FinModelRatesHW

Class: FinModelRatesHW()

class FinModelRatesHW():

Data Members

- _sigma
- _a
- _numTimeSteps
- _useJamshidian
- _Q
- _r
- _treeTimes
- _pu
- _pm
- _pd
- _discountCurve
- _treeBuilt
- _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. The model will switch to use Jamshidians approach where possible unless the useJamshidian flag is set to false in which case it uses the trinomial Tree.

```
def __init__(self, sigma, a, numTimeSteps=100, useJamshidian=True):
```

optionOnZCB

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 optionOnZCB(self, texp, tmat, strike, 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, strike, 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

Value an option on a bond with coupons that can have European or American exercise. Some minor issues to do with handling coupons on the option expiry date need to be solved.

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

callablePuttableBond_Tree

```
def callablePuttableBond_Tree(self,
                              couponTimes, couponAmounts,
                              callTimes, callPrices,
                              putTimes, putPrices,
                              face):
```

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

Build the trinomial tree.

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

__repr__

Return string with class details.

```
def __repr__(self):
```

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 dollar of the discount factor.

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

buildTree_Fast

Fast tree construction using Numba.

```
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):
```

callablePuttableBond_Tree_Fast

```
def callablePuttableBond_Tree_Fast(couponTimes, couponAmounts,
                                   callTimes, callPrices,
                                   putTimes, putPrices, face,
                                   _sigma, _a, _Q,  # IS SIGMA USED ?
                                   _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):
```

10.18 FinModelRatesLMM

Enumerated Type: FinRateModelLMMModelTypes

- LMM_ONE_FACTOR
- LMM_HW_M_FACTOR
- LMM_FULL_N_FACTOR

LMMPrintForwards

Helper function to display the simulated Libor rates.

```
def LMMPrintForwards(fwds):
```

LMMSwaptionVolApprox

Implements Rebonatos approximation for the swap rate volatility to be used when pricing a swaption that expires in period a for a swap maturing at the end of period b taking into account the forward volatility term structure (zetas) and the forward-forward correlation matrix rho..

```
def LMMSwaptionVolApprox(a, b, fwd0, taus, zetas, rho):
```

LMMSimSwaptionVol

Calculates the swap rate volatility using the forwards generated in the simulation to see how it compares to Rebonatto estimate.

```
def LMMSimSwaptionVol(a, b, fwd0, fwds, taus):
```

LMMFwdFwdCorrelation

Extract forward forward correlation matrix at some future time index from the simulated forward rates and return the matrix.

```
def LMMFwdFwdCorrelation(numForwards, numPaths, iTime, fwds):
```

LMMPriceCapsBlack

Price a strip of capfloorlets using Blacks model using the time grid of the LMM model. The prices can be compared with the LMM model prices.

```
def LMMPriceCapsBlack(fwd0, volCaplet, p, K, taus):
```

subMatrix

Returns a submatrix of correlation matrix at later time step in the LMM simulation which is then used to generate correlated Gaussian RVs.

```
def subMatrix(t, N):
```

CholeskyNP

Numba-compliant wrapper around Numpy cholesky function.

```
def CholeskyNP(rho):
```

LMMSimulateFwdsNF

Full N-Factor Arbitrage-free simulation of forward Libor curves in the spot measure given an initial forward curve, volatility term structure and full rank correlation structure. Cholesky decomposition is used to extract the factor weights. The number of forwards at time 0 is given. The 3D matrix of forward rates by path, time and forward point is returned. WARNING: NEED TO CHECK THAT CORRECT VOLATILITY IS BEING USED (OFF BY ONE BUG NEEDS TO BE RULED OUT)

```
def LMMSimulateFwdsNF(numForwards, numPaths, fwd0, zetas, correl, taus, seed):
```

LMMSimulateFwds1F

One factor Arbitrage-free simulation of forward Libor curves in the spot measure following Hull Page 768. Given an initial forward curve, volatility term structure. The 3D matrix of forward rates by path, time and forward point is returned. This function is kept mainly for its simplicity and speed. NB: The Gamma volatility has an initial entry of zero. This differs from Hulls indexing by one and so is why I do not subtract 1 from the index as Hull does in his equation 32.14. The Number of Forwards is the number of points on the initial curve to the trade maturity date. For example a cap that matures in 10 years with quarterly caplets has 40 forwards.

```
def LMMSimulateFwds1F(numForwards, numPaths, numeraireIndex, fwd0, gammas,
                      taus, useSobol, seed):
```

LMMSimulateFwdsMF

Multi-Factor Arbitrage-free simulation of forward Libor curves in the spot measure following Hull Page 768. Given an initial forward curve, volatility factor term structure. The 3D matrix of forward rates by path, time and forward point is returned.

```
def LMMSimulateFwdsMF(numForwards, numFactors, numPaths, numeraireIndex, fwd0,
                      lambdas, taus, useSobol, seed):
```

LMMCapFlrPricer

Function to price a strip of cap or floorlets in accordance with the simulated forward curve dynamics.

```
def LMMCapFlrPricer(numForwards, numPaths, K, fwd0, fwds, taus, isCap):
```

LMMSwapPricer

Function to reprice a basic swap using the simulated forward Libors.

```
def LMMSwapPricer(cpn, numPeriods, numPaths, fwd0, fwds, taus):
```

LMMSwaptionPricer

Function to price a European swaption using the simulated forward curves.

```
def LMMSwaptionPricer(strike, a, b, numPaths, fwd0, fwds, taus, isPayer):
```

LMMRatchetCapletPricer

Price a ratchet using the simulated Libor rates.

```
def LMMRatchetCapletPricer(spread, numPeriods, numPaths, fwd0, fwds, taus):
```

LMMFlexiCapPricer

Price a flexicap using the simulated Libor rates.

```
def LMMFlexiCapPricer(maxCaplets, K, numPeriods, numPaths, fwd0, fwds, taus):
```

LMMStickyCapletPricer

Price a sticky cap using the simulated Libor rates.

```
def LMMStickyCapletPricer(spread, numPeriods, numPaths, fwd0, fwds, taus):
```

10.19 FinModelRatesVasicek

Class: FinModelRatesVasicek()

class FinModelRatesVasicek():

Data Members

- _a
- _b
- _sigma

Functions

__init__

self._a = a

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

__repr__

s = labelToString("a", self._a)

```
    def __repr__(self):
```

meanr

mr = r0 * exp(-a * t) + b * (1 - exp(-a * t))

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

variancer

vr = sigma * sigma * (1.0 - exp(-2.0 * a * t)) / 2.0 / a

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

zeroPrice

B = (1.0 - exp(-a * t)) / a

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

ratePath_MC

PLEASE ADD A FUNCTION DESCRIPTION

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

zeroPrice_MC

PLEASE ADD A FUNCTION DESCRIPTION

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

10.20 FinModelSABR

Class: FinModelSABR()

Data Members

- _alpha
- _beta
- _rho
- _nu

Functions

__init__

Create FinModelSABR with model parameters.

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

blackVol

Black volatility from SABR model using Hagan et al. approx.

```
def blackVol(self, f, k, t):
```

value

Price an option using Blacks model which values in the forward measure following a change of measure.

```
def value(self,
          forwardRate,    # Forward rate
          strikeRate,     # Strike Rate
          timeToExpiry,   # time to expiry in years
          df,             # Discount Factor to expiry date
          callOrPut):     # Call or put
```

blackVolFromSABR

PLEASE ADD A FUNCTION DESCRIPTION

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

10.21 FinModelSABRShifted

Class: FinModelSABRShifted()

Data Members

- _alpha
- _beta
- _rho
- _nu
- _shift

Functions

__init__

self._alpha = alpha

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

blackVol

Black volatility from SABR model using Hagan et al. approx.

```
def blackVol(self, f, k, t):
```

value

Price an option using Blacks model which values in the forward measure following a change of measure.

```
def value(self,
          forwardRate,    # Forward rate F
          strikeRate,     # Strike Rate K
          timeToExpiry,   # Time to Expiry (years)
          df,             # Discount Factor to expiry date
          callOrPut):     # Call or put
```

blackVolFromShiftedSABR

PLEASE ADD A FUNCTION DESCRIPTION

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

10.22 FinModelStudentTCopula

Class: FinModelStudentTCopula()

class FinModelStudentTCopula():

Data Members

No data members found.

Functions

defaultTimes

PLEASE ADD A FUNCTION DESCRIPTION

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

10.23 FinProcessSimulator

Enumerated Type: FinProcessTypes

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

Enumerated Type: FinHestonNumericalScheme

- EULER
- EULERLOG
- QUADEXP

Enumerated Type: FinGBMNumericalScheme

- NORMAL
- ANTITHETIC

Enumerated Type: FinVasicekNumericalScheme

- NORMAL
- ANTITHETIC

Enumerated Type: FinCIRNumericalScheme

- EULER
- LOGNORMAL
- MILSTEIN
- KAHLJACKEL
- EXACT

Class: FinProcessSimulator()

class FinProcessSimulator():

Data Members

No data members found.

Functions

__init__

pass

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

getProcess

PLEASE ADD A FUNCTION DESCRIPTION

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

getHestonPaths

PLEASE ADD A FUNCTION DESCRIPTION

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

getGBMPaths

PLEASE ADD A FUNCTION DESCRIPTION

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

getVasicekPaths

PLEASE ADD A FUNCTION DESCRIPTION

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

getCIRPaths

PLEASE ADD A FUNCTION DESCRIPTION

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