## What needs to be thought of when developing an options pricer

When implementing an options pricer there are some very basic choices to make, then one has to build a set of required components and, eventually, has to fine-tune those components for optimum performance and result precision.

The choices we have made will be specified in quare brackets below.

The basic choices are:

* Types of products supported [American Options]
* Number of factors to consider [One factor – underlying value]
* Volatility model to use [Local Volatility]
* Pricing algorithm to use [PDE]

Given our basic choices, we found some extra components had to be built:

* Day count calculator
* Holiday calendars
* Volatility surface implementation
* Interest rates model
* Payoff model
* PDE grid
* PDE pricer

The importance, roles, and potential ways to improve these components are explained below.

### Day count calculator

Our very simple day count calculator is implemented in daycount.py. This component acts as a bridge between market data, normally available as numbers for particular date/datetime points, and the mathematical models’ world where time is usually expressed as a year fraction.

We implemented an ultra-simple approach where we assume each year consists of 365.25 days (mind the leap day, forget any other adjustments). Then we take a real number of days in a certain dates range and divide it by 365.25 to get a year fraction. An obvious disadvantage of this approach is that no year will have exact year fraction of 1.0, as leap years will be 366/365.25 and non-leap years 365/365.25. However, the chosen approach is very simple to implement and, most importantly, to reverse – we can easily calculate a date given a start date and a year fraction and we are 100% sure our year fraction by date and date by year fraction functions are 100% consistent.

There are many other possible approaches, some of which may be market conventions for particular asset classes and/or markets. For instance, one may use the real number of days from all years involved as their daycounts, so that every year’s year fraction is actually 1.0. There are some conventions which would use 365.0 days as a daycount for all years. Also, one could use logic which would only take business days into account.

### Holiday calendars

Out approach to business days is ultra-simple as well: we assume every week day is a working day and every weekend has 0 volatility. Implementation can be found in daycount.py.

In reality, one may want to store a list of holidays for all countries of interest, and to combine those using, sometimes not very trivial, market conventions. For example, for a EURUSD FX trade one would probably assume that a union of USA and Eurozone holidays should be used as a holiday calendar, although that may also depend on rules of a particular exchange or market conventions used in OTC markets. For cross-currency trades like EURTRY, USD calendars may still be involved, as an acknowledgement of the fact that cross FX rate is frequently calculated using two USD rates – USDTRY and EURUSD in our example.

### Volatility surface implementation

TODO: revise this once there is a real implementation

Our volatility surface uses implied volatilities we obtained from our source data, keyed on expiry date x strike. Our surface’s API provides access to interpolated volatilities in terms of expiry date (or, alternatively, year fraction) and a strike as well. To be able to use volatility at any point in time we use flat extrapolation in both time and strike spaces, and we use the following interpolation:

* We first linearly interpolate volatility smiles for every expiry date we have data for
* Given a strike we’re interested in, we pick two corresponding interpolated volatilities from smiles, adjacent to the expiry date of interest: V1 and V2
* We interpolate between V1 and V2 in time space.

If requested expiry date falls onto one of volatility smiles’ expiry dates, we do not interpolate in time space. Implementation can be found in vol\_surface.py.

In real-world volatility models, a lot of extra features may be considered, including:

* Non-flat extrapolation – for smiles one may want to define linear extrapolation slope or something more complex, time extrapolation may be at least linear, not constant
* Linear interpolation can be replaced with a smoother one
* Not all markets assume volatility is strike-sticky. With our approach we assume volatility for a particular strike and time stays the same regardless of underlying value. However e.g., FX markets, will assume volatility is delta-sticky, i.e., implied volatility stays the same for a particular time and option’s delta. Commodity markets would normally use sticky moneyness, meaning volatility will be the same for time and S/K ratio. Where S is underlying value and K is option’s strike.

### Interest rates model

We used a piecewise-constant interpolation for interest rates, which is better than a constant rate, but could be improved further for real-life applications.

Our rates model is based on some USD rates from [home.treasury.gov](http://home.treasury.gov), as we only deal with USD-denominated equities. Yield curve is implemented in yc.py with some helper code also available to fetch source data for interest rates.

Data sources exist for rates in other currencies, some of which have fees for data access. There are normally many ways to build a yield curve for any particular currencies, but basing it off various observed interest rates products and bootstrapping. In many cases pricing applications would also distinguish between pricing and discounting curves, where pricing curve may match some kind of market consensus on a rate and discounting curve may account for cost of borrowing for a particular market participant.

When a counterparty is known, market participants may use dedicated yield curves for the client in question, with rates accounting for default risk.

### Payoff model

Payoff model needs to be built in order to run PDE pricer and Monte-Carlo simulations. It can be either embedded into pricers themselves, but we found useful to have it as a separate entity with an added functionality of getting a discounted payoff for a given date. This is implemented in payoff.py.

For more complex payoffs, notably barriers, where price is not continuous as a function of underlying value, one may find it useful sometimes to amend payoffs slightly to improve pricing precision, e.g., move a barrier a bit, calculate a more stable price, and then adjust the resulting price, accounting for initial barrier shift.

### PDE Grid

We found it somewhat useful to have a PDE grid separated from PDE pricer. Our grid is parameterised by the number of time points T and spot points S. We use constant steps in both T and S, with the latter allowing us to involve no interpolation when getting t+1 values for the same spot value during PDE steps.

Our grid is centred around current spot value and its code is located in pde\_grid.py.

The extensions of this approach could include:

* Fewer points in the ITM part of grid where the option’s value may already be equal to that of a forward.
* Uneven grid, e.g., logarithmic
* More time points towards short end of the grid, where market data normally has more points (e.g., volatility tenors may be something like ON TN 1W 1M 3M 6M 1Y 2Y)

### PDE Pricer

We used an implicit PDE pricer, as suggested in [Hjelmberg, Lagerstrom]. We also calculated a maximum of so far estimated price and current discounted payoff at every point of time as suggested in [Hull, 2011], thus we haven’t calculated an early execution boundary. We have used NumPy linalg solver to solve the systems of linear equations at every time point, without any explicit optimisations to account for the matrix being 3-diagonal. Implementation for PDE pricer is located in pde\_pricer.py.

Every time step could potentially be parallelised, so that potentially slower queries like getting volatilities for strikes could be parallelised / vectorised.

An optimisation could be also added for non-dividend American calls, to avoid early expiry checks for those.

Speaking of dividends, taking them into account is also a way to make the existing algorithm more versatile.

### Bid/offer spread

We haven’t taken bid/offer spread into account at all, while it could be taken into account for underlying price, rates and volatilities.

### Multi-factor models

We are only considering a single factor – the underlying share price itself. However, experienced market participants may use multi-factor models, such as stochastic volatility models (underlying and its volatility are stochastic factors) or stochastic rates models, where non-rates underlying and interest rates are both considered stochastic, and others.