Vanilla Option Pricing

Release 0.1.0

Dec 07, 2020

Quick reference

A Python package implementing stochastic models to price financial options.

The theoretical background and a comprehensive explanation of models and their parameters can be found is the paper Fast calibration of two-factor models for energy option pricing by Emanuele Fabbiani, Andrea Marziali and Giuseppe De Nicolao, freely available on arXiv.

CHAPTER 1

Installation

The preferred way to install the package is using pip.

If your want to modify the code or contribute to the development, feel free to clone the repository and to set up the project on your own machine.

1.1 Install from PyPI

Install the package (or add it to your `requirements.txt` file):

```
pip install vanilla_option_pricing
```

1.2 Set up the project

Clone the repository:

```
git clone https://github.com/donlelef/vanilla-option-pricing.git
```

The project uses Poetry as a dependency manager. To get started, follow the official documentation. Then, simply run:

```
poetry install
```

CHAPTER 2

Getting started

This tutorial shows basic usage of this package.

2.1 Creating options

Let's create a sample call option

```
from datetime import date, datetime, timedelta
from vanilla_option_pricing.option import VanillaOption
from vanilla_option_pricing.models import GeometricBrownianMotion
from vanilla_option_pricing.calibration import ModelCalibration

option = VanillaOption(
    spot=100,
    strike=101,
    dividend=0,
    date=datetime.today(),
    maturity=datetime.today() + timedelta(days=30),
    option_type='c',
    price=1,
    instrument='TTF'
)
```

2.2 Implied volatility and option pricing

We can compute the implied volatility and create a Geometric Brownian Motion model with it. Of course, if now we ask price the option using the Black framework, we'll get back the initial price.

```
volatility = option.implied_volatility_of_undiscounted_price
model = GeometricBrownianMotion(volatility)
```

(continues on next page)

(continued from previous page)

```
model_price = model.price_option_black(option)
print(f'Actual price: {option.price}, model price: {model_price}')
```

2.3 Calibrating models

We can also try and calibrate the parameters of a model against listed options.

```
data_set = [
    VanillaOption('TTF', 'c', date(2018, 1, 1), 2, 101, 100, date(2018, 2, 1)),
    VanillaOption('TTF', 'p', date(2018, 1, 1), 2, 98, 100, date(2018, 2, 1)),
    VanillaOption('TTF', 'c', date(2018, 1, 1), 5, 101, 100, date(2018, 5, 31))
]

for o in data_set:
    print(f'Implied volatility: {o.implied_volatility_of_undiscounted_price}')

model = GeometricBrownianMotion(0.2)
calibration = ModelCalibration(data_set)

result, tuned_model = calibration.calibrate_model(model)
print(result)
print(f'Calibrated implied volatility: {tuned_model.parameters[0]}')
```

As we can see, the calibration process takes the implied volatility of the model close to the average of the options it has been trained on.

CHAPTER 3

Models

In the context of this package, a model is a stochastic process.

The package APIs offer a simple way of extracting the variance, the standard deviation and the volatility derived from a model at a given time instant.

There are three models currently implemented by this package: a detailed description and further references can be found in the paper Fast calibration of two-factor models for energy option pricing.

3.1 Geometric Brownian Motion

The celebrated Geometric Brownian Motion process, adopted in the Black and Black-Scholes-Merton frameworks for option pricing.

```
from datetime import datetime, timedelta
from vanilla_option_pricing.option import VanillaOption
from vanilla_option_pricing.models import GeometricBrownianMotion

option = VanillaOption(
    spot=100,
    strike=101,
    dividend=0,
    date=datetime.today(),
    maturity=datetime.today() + timedelta(days=30),
    option_type='c',
    price=1,
    instrument='TTF'
)

volatility = option.implied_volatility_of_undiscounted_price
print(f'Option volatility is {volatility}')
gbm_model = GeometricBrownianMotion(volatility)
t = 0.5
print(f'At time t={t} years, volatility is {gbm_model.volatility(t)}, '
```

(continues on next page)

(continued from previous page)

```
      f'variance is {gbm_model.variance(t)}, '
      f'standard deviation is {gbm_model.standard_deviation(t)}')
```

3.2 Ornstein-Uhlenbeck

The Ornstein-Uhlenbeck process, the simplest mean-reverting model, quite popular for energy commodities.

```
from vanilla_option_pricing.models import OrnsteinUhlenbeck

ou_model = OrnsteinUhlenbeck(
    p_0 = 1,
    l = 1,
    s = volatility
)
print(f'At time t={t} years, volatility is {ou_model.volatility(t)}, '
      f'variance is {ou_model.variance(t)}, '
      f'standard deviation is {ou_model.standard_deviation(t)}')
```

3.3 Log Mean-Reverting To Generalised Wiener Process

One of the most common two-factor, mean-reverting models.

```
import numpy as np
from vanilla_option_pricing.models import LogMeanRevertingToGeneralisedWienerProcess

lmrgw_model = LogMeanRevertingToGeneralisedWienerProcess(
    p_0 = np.eye(2),
    l = 100,
    s_x = 0.1,
    s_y = 0.3
)
print(f'At time t={t} years, volatility is {lmrgw_model.volatility(t)}, '
      f'variance is {lmrgw_model.variance(t)}, '
      f'standard deviation is {lmrgw_model.standard_deviation(t)}')
```

CHAPTER 4

Model calibration

The market calibration calibration is a procedure which takes an option pricing model and a set of listed vanilla options and tunes the parameters of the model so that the option price given by the model is as close as possible to the actual prices of listed options.

More rigorous details and a mathematical formulation can be found in the paper Fast calibration of two-factor models for energy option pricing.

4.1 Creating inputs

We'll suppose that the available dataset to tune our model contains has only three options. In a realistic scenario, tens to hundreds of options would be needed.

```
from datetime import date
from vanilla_option_pricing.option import VanillaOption
from vanilla_option_pricing.models import GeometricBrownianMotion, OrnsteinUhlenbeck
from vanilla_option_pricing.calibration import ModelCalibration

data_set = [
    VanillaOption('TTF', 'c', date(2018, 1, 1), 2, 101, 100, date(2018, 2, 1)),
    VanillaOption('TTF', 'p', date(2018, 1, 1), 2, 98, 100, date(2018, 2, 1)),
    VanillaOption('TTF', 'c', date(2018, 1, 1), 5, 101, 100, date(2018, 5, 31))
]
```

We want to calibrate both a Geometric Brownian motion and an Ornstein-Uhlenbeck model.

```
models = [
    GeometricBrownianMotion(0.2),
    OrnsteinUhlenbeck(p_0=0, l=100, s=2)
]
```

4.2 Calibrating models

We can now instantiate the calibration object, run the optimization algorithm and inspect the results.

```
calibration = ModelCalibration(data_set)

for model in models:
    result, trained_model = calibration.calibrate_model(model)
    print('Optimization results:')
    print(result)
    print(f'Calibrated parameters: {trained_model.parameters}\n\n')
```

CHAPTER 5

vanilla_option_pricing package

5.1 vanilla_option_pricing.calibration module

class vanilla_option_pricing.calibration.**ModelCalibration**(*options: List[vanilla_option_pricing.option.VanillaOptio*

Bases: object

Calibrate option pricing models with prices of listed options

Parameters **options** – a collection of VanillaOption

DEFAULT_PARAMETER_LOWER_BOUND = 0.0001

calibrate_model(*model: vanilla_option_pricing.option_pricing.OptionPricingModel*, *method: str = None*, *options: Dict[KT, VT] = None*, *bounds: Union[str, Sequence[Tuple[float, float]]] = 'default'*) → Tuple[scipy.optimize.optimize.OptimizeResult, vanilla_option_pricing.option_pricing.OptionPricingModel]

Tune model parameters and returns a tuned model. The algorithm tries to minimize the squared difference between the prices of listed options and the prices predicted by the model: the parameters of the model are the optimization variables. The numerical optimization is performed by minimize() in the scipy package.

Parameters

- **model** – the model to calibrate
- **method** – see minimize()
- **options** – see minimize()
- **bounds** – the bounds to apply to parameters. If none is specified, then the *DEFAULT_PARAMETER_LOWER_BOUND* is applied for all the parameters. Otherwise, a list of tuples (lower_bound, upper_bound) for each parameter shall be specified.

Returns a tuple (res, model), where res is the result of minimize(), while model a calibrated model

5.2 vanilla_option_pricing.models module

class vanilla_option_pricing.models.**GeometricBrownianMotion**(*s: float*)
Bases: *vanilla_option_pricing.option_pricing.OptionPricingModel*

The celebrated Geometric Brownian Motion model

Parameters **s** – the volatility, must be non-negative

name = 'Geometric Brownian Motion'

parameters
Model parameters, as a tuple of real numbers, in the order (s,).

variance(*t: float*) → float
The variance of the model output at a given time instant

Parameters **t** – the time when the variance is evaluated

Returns the variance at time t

class vanilla_option_pricing.models.**LogMeanRevertingToGeneralisedWienerProcess**(*p_0: numpy.array*, *l: float*, *s_x: float*, *s_y: float*)
Bases: *vanilla_option_pricing.option_pricing.OptionPricingModel*

The Log Mean-Reverting To Generalised Wiener Process model. It is a two-factor, mean reverting model, where the long-term behaviour is given by a Geometric Brownian motion, while the short-term mean-reverting tendency is modelled by an Ornstein-Uhlenbeck process.

Parameters

- **p_0** – the initial variance, that is the variance of the state at time t=0. Must be a 2x2 numpy array
- **l** – the strength of mean-reversion, must be non-negative
- **s_x** – volatility of the long-term process, must be non-negative
- **s_y** – volatility of the short-term process, must be non-negative

name = 'Log Mean-Reverting To Generalised Wiener Process'

parameters
Model parameters, as a tuple of real numbers, in the order l, s_x, s_y.

variance(*t: float*) → float
The variance of the model output at a given time instant

Parameters **t** – the time when the variance is evaluated

Returns the variance at time t

class vanilla_option_pricing.models.**NumericalLogMeanRevertingToGeneralisedWienerProcess**(*p_0*
num
l:
floa
s_x
floa
s_y
floa

Bases: *vanilla_option_pricing.option_pricing.OptionPricingModel*

This model relies on the same stochastic process as `LogMeanRevertingToGeneralisedWienerProcess`, but uses a numerical procedures based on a matrix exponential instead of the analytical formulas to compute the variance. As this approach is considerably slower, it is strongly suggested to adopt `LogMeanRevertingToGeneralisedWienerProcess` instead, using this class only for benchmarking

Parameters

- **p_0** – the initial variance, that is the variance of the state at time t=0. Must be a 2x2 numpy array, symmetric and positive semidefinite
- **l** – the strength of mean-reversion, must be non-negative
- **s_x** – volatility of the long-term process, must be non-negative
- **s_y** – volatility of the short-term process, must be non-negative

name = 'Numerical Log Mean-Reverting To Generalised Wiener Process'

parameters
Model parameters, as a tuple of real numbers, in the order l, s_x, s_y.

variance(*t: float*) → float
The variance of the model output at a given time instant

Parameters **t** – the time when the variance is evaluated

Returns the variance at time t

class vanilla_option_pricing.models.**NumericalModel**(*A: numpy.array*, *B: numpy.array*, *p_0: numpy.array*)

Bases: `object`

A general-purpose linear stochastic system. All the parameters must be matrices (as Numpy arrays) of suitable dimensions.

Parameters

- **A** – the dynamic matrix A of the system
- **B** – the input matrix B of the system
- **p_0** – the initial variance, that is the variance of the state at time t=0, must be symmetric and positive semidefinite

variance(*t: float*) → float
The variance of the model output at a given time instant

Parameters **t** – the time when the variance is evaluated

Returns the variance at time t

class vanilla_option_pricing.models.**OrnsteinUhlenbeck**(*p_0: float*, *l: float*, *s: float*)
Bases: *vanilla_option_pricing.option_pricing.OptionPricingModel*

The single-factor, mean-reverting Ornstein-Uhlenbeck process.

Parameters

- **p_0** – the initial variance, that is the variance of the state at time t=0, must be positive semidefinite and symmetric
- **l** – the strength of the mean-reversion, must be non-negative
- **s** – the volatility, must be non-negative

name = 'Ornstein-Uhlenbeck'

parameters
Model parameters, as a list of real numbers, in the order [l, s].

variance(*t: float*) → float
The variance of the model output at a given time instant

Parameters **t** – the time when the variance is evaluated

Returns the variance at time t

5.3 vanilla_option_pricing.option module

class vanilla_option_pricing.option.**VanillaOption**(*instrument: str*, *option_type: str*, *date: datetime.datetime*, *price: float*, *strike: float*, *spot: float*, *maturity: datetime.datetime*, *dividend=0*)

Bases: object

A European vanilla option. All the prices must share the same currency.

Parameters

- **instrument** – name of the underlying
- **option_type** – type of the option (c for call, p for put)
- **date** – the date when the option is traded
- **price** – option price
- **strike** – option strike price
- **spot** – spot price of the underlying
- **maturity** – the maturity date
- **dividend** – underlying dividend - if any, expressed as a decimal number

DAYS_IN_YEAR = 365.2425

implied_volatility_of_undiscounted_price
The implied volatility of the option, considering an undiscounted price. Returns zero if the implied volatility is negative.

to_dict()

Returns all the fields of the object in a dictionary

years_to_maturity
The years remaining to option maturity, as a decimal number.

vanilla_option_pricing.option.**check_option_type**(*option_type: str*)
A utility function to check the validity of the type of an option. Raises a ValueError if the type is invalid. :param option_type: the type of the option: valid types are "c" for call and "p" for put

vanilla_option_pricing.option.**option_list_to_pandas_dataframe**(*options: List[vanilla_option_pricing.option.Vanilla* → pandas.core.frame.DataFrame
A utility function to convert a list of `VanillaOption` to a pandas dataframe.

Parameters **options** – a list of `VanillaOption`

Returns a pandas dataframe, containing option data

vanilla_option_pricing.option.**pandas_dataframe_to_option_list**(*data_frame: pandas.core.frame.DataFrame*) → List[vanilla_option_pricing.option.VanillaO
A utility function to convert a pandas dataframe to a list of `VanillaOption`. For this function to work, the dataframe columns should be named as the parameters of `VanillaOption`'s constructor.

Parameters **data_frame** – a pandas dataframe, containing option data

Returns a list of `VanillaOption`

5.4 vanilla_option_pricing.option_pricing module

class vanilla_option_pricing.option_pricing.**OptionPricingModel**
Bases: `abc.ABC`

A model which can be used to price European vanilla options.

parameters
The model parameters, returned as a list of real numbers.

price_black(*option_type: str*, *spot: float*, *strike: float*, *years_to_maturity: float*) → float
Finds the no-arbitrage price of a European Vanilla option. Price is computed using the Black formulae, but the variance of the underlying is extracted from this model.

Parameters

- **option_type** – the type of the option (c for call, p for put)
- **spot** – the spot price of the underlying
- **strike** – the option strike price
- **years_to_maturity** – the years remaining before maturity - as a decimal number

Returns the no-arbitrage price of the option

price_black_scholes_merton(*option_type: str*, *spot: float*, *strike: float*, *years_to_maturity: float*, *risk_free_rate: float*, *dividend: float = 0*) → float
Finds the no-arbitrage price of a European Vanilla option. The price is computed using the Black-Scholes-Merton framework, but the variance of the underlying is extracted from this model.

Parameters

- **option_type** – the type of the option (c for call, p for put)
- **spot** – the spot price of the underlying
- **strike** – the option strike price

- **years_to_maturity** – the years remaining before maturity - as a decimal number
- **risk_free_rate** – the risk-free interest rate
- **dividend** – the dividend paid by the underlying - as a decimal number

Returns the no-arbitrage price of the option

price_option_black(*option: vanilla_option_pricing.option.VanillaOption*) → float
Same as `price_black()`, but the details of the vanilla option are provided by a `VanillaOption` object.

Parameters **option** – a `VanillaOption`

Returns the no-arbitrage price of the option

price_option_black_scholes_merton(*option: vanilla_option_pricing.option.VanillaOption*, *risk_free_rate: float*) → float
Same as `price_black_scholes_merton()`, but the details of the vanilla option are provided by a `VanillaOption` object.

Parameters

- **option** – a `VanillaOption`
- **risk_free_rate** – the risk-free interest rate

Returns the no-arbitrage price of the option

standard_deviation(*t: float*) → float
The standard deviation of the model output at a given instant, that is the squared root of the `variance()` at the same instant

Parameters **t** – the time when the standard deviation is evaluated

Returns the standard deviation at time t

variance(*t: float*) → float
The variance of the model output at a given time.

Parameters **t** – the time when the variance is evaluated

Returns the variance at time t

volatility(*t: float*) → float
The volatility of the model output at a certain instant, that is the `standard_deviation()` divided by the squared root of the time

Parameters **t** – the time when the volatility is evaluated

Returns the volatility at time t

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

v

Index

S

T

V

Y