Finite Difference Methods with Python

Vahan Hovhannisyan
vahan.hovhannisyan@gmail.com

2024

1 Introduction

This tutorial aims to give you practical experience implementing finite difference methods in Python, particularly for examples from finance and options theory. We will implement solutions for the Black-Scholes equation for a European put option. The theory in this tutorial is based on the material from *Numerical Methods in Finance with C++* by M. Capiński and T. Zastawniak [1], which is available from the library in electronic version.

Throughout this tutorial we will use `Python3` and assume basic knowledge of `Python`. We will use `Jupyter notebook`, which is available from `Software Hub` or can be downloaded from https://jupyter.org. For a detailed `Python` tutorial, please refer to https://docs.python.org/3/tutorial/

2 Getting started with Python and Jupyter

2.1 Functions

We will start with a function that computes and returns the payoff of a European put option. Functions are declared using the `def` keyword.

Listing 1: eu_put_payoff

```python
def eu_put_payoff(strike, spot):
    """ Computes the payoff of a European put option
    :param strike: the strike price
    :param spot: the spot price
    :return: payoff value
    """
    return max(strike - spot, 0)

# Take strike and spot price values as user input
# `input` returns a `str`, so we need to convert it to `float`
strike = float(input("Enter the strike price: "))
spot = float(input("Enter the spot price: "))
# Compute and print the payoff of the European put option
payoff = eu_put_payoff(strike, spot)
print("The payoff of the European put option is: {0}".format(payoff))
```

In this example, we defined and used a new function called `Payoff`, which takes two floating point numbers as input and returns one floating point number. Note that `float` and `str` are some of the predefined types in `Python`. Next we will work with user-defined types.

2.2 Object Oriented Programming

An important feature of `Python` is the support for Object Oriented Programming (OOP). Today we will look only at the basics of OOP: classes, objects and members.

- Classes are user defined types and are used to collect relevant data and functions together.
- Object is an instance of a class. Object compares to its class, as `2.5` compares to `float`.
- Each class has members: data and functions.
- Constructor is a special member function of any class. In `Python` `__init__()` is a reserved method and is the constructor for its class.

We will demonstrate the *class syntax* using a simple `EuPut` class. Classes are declared using the `class` keyword. To declare a member function, we need to pass a reference to the object as the first input argument to that function. It is standard practice to call that argument `self`.

Listing 2: The EuPut class

```python
class EuPut:
    """ Class for working with European put opions
    """
    def __init__(self, strike):
        """ Constructor
        :param strike: option's strike price
        """
        self.strike = strike # Assign the input argument to the object member

    def payoff(self, spot):
        """ Return the payoff at a given spot price
        :param spot: option's spot price
        :return: payoff value at the given spot price
        """
        return max(self.strike - spot, 0)
```

Remark. Member functions can directly access other members of the same object. However, every new object will have its own distinct members.

Now we can use the `EuPut` class to calculate payoffs for various spot prices.

Listing 3: Using the EuPut class

```python
# One example
option = EuPut(100) # Create a new EuPut object with 100 strike price
spot = 90
payoff = option.payoff(spot) # Compute payoff at 90 spot price
print(f"Strike price: {option.strike}, spot price: {spot}, payoff: {payoff}")

# More examples
# Make a list of 2 EuPut objects with 100 and 200 strike prices
options = [EuPut(100), EuPut(200)]
spots = [80, 110, 170, 220] # Make a list of 4 spot prices
for option in options: # iterate over all options and spot prices
    for spot in spots:
        payoff = option.payoff(spot) # compute payoff and print
        print(f"Strike price: {option.strike}, spot price: {spot}, payoff: {payoff}")
```

3 Parabolic partial differential equations

First, let us consider the parabolic differential equation:

$$\frac{\partial v(t,x)}{\partial t} = a(t,x)\frac{\partial^2 v(t,x)}{\partial x^2} + b(t,x)\frac{\partial v(t,x)}{\partial x} + c(t,x)v(t,x) + d(t,x) \tag{1}$$

for $(t, x) \in [0, T] \times [x_l, x_u]$. We are also given the following boundary conditions:

$$v(T, x) = f(x), \tag{2}$$

$$v(t, x_l) = f_l(t), \tag{3}$$

$$v(t, x_u) = f_u(t), \tag{4}$$

where $f : [x_l, x_u] \to \mathbb{R}$ and $f_l, f_u : [0, T] \to \mathbb{R}$ are given functions. f is called *terminal boundary condition*, f_l - *lower boundary condition* and f_u - *upper boundary condition.*

A typical example of (1) is the Black-Scholes equation:

$$\frac{\partial u(t,z)}{\partial t} + \frac{\sigma^2}{2} z^2 \frac{\partial^2 u(t,z)}{\partial z^2} + rz \frac{\partial u(t,z)}{\partial z} - ru(t,z) = 0, \tag{5}$$

where $u(t, S(t)) : [0, T] \times \mathbb{R} \to \mathbb{R}$ is a $C^{1,2}$ pricing function for an underlying asset with price $S(t)$ at time t. (5) is a special case of (1) with parameters

$$\begin{aligned} a(t,z) &= -\frac{\sigma^2}{2} z^2, \\ b(t,z) &= -rz, \\ c(t,z) &= r, \\ d(t,z) &= 0. \end{aligned} \tag{6}$$

The boundary conditions for a put option with expiry date T and strike price K will be

$$\begin{aligned} &f(z) = u(T,z) = \max\{K - z, 0\}, \\ &fu(t) = u(t, z_u) = 0, \\ &fl(t) = u(t, z_l) = e^{-r(T-t)} K. \end{aligned} \tag{7}$$

4 Black-Scholes for European Put Option

In this subsection we will create a simple class to model the Black-Scholes equation for a European put option. Our class will need data about the underlying asset:

- `rate` - risk free interest rate and
- `sigma` - standard deviation,

the option:

- `strike` - strike price and
- `t_up` - time to expiration,
- `x_low` and `x_up` - lower and upper bounds for the the spot price.

Note that we used rather generic variable names for boundary conditions - time to expiration and low and upper bound on the spot price. This is to reflect the fact that our solvers are not limited to the Black-Scholes equation and could solve any parabolic PDEs.

Then adding implementations for the coefficients and the boundary conditions we get the following `EuPutBlackScholes` class.

Listing 4: The EuPutBlackScholes class

```
import math # `math` contains basic mathematical function

class EuPutBlackScholes:
    """ Class containing data and methods to represent
```

```
the Black-Scholes PDE for European put option
"""
def __init__(self, rate, sigma, strike, t_up, x_low, x_up):
    """ Constructor
    :param rate: underlying asset's risk free interest rate
    :param sigma: underlying asset's standard deviation
    :param strike: option's strike price
    :param t_up: option's time to expiration
    :param x_low: lower bound for the option's spot price
    :param x_up: upper bound for the option's spot price
    """
    # Initialise `rate`, `sigma`, `sigma`, `strike`,
    # `t_low`, `t_up`, `x_low` and `x_up` members

def coeff_a(self, t, x):
    """ Compute coefficient a
    :param t: current time value
    :param x: current spot value
    """
    # Return the `a(t,x) coefficient

def coeff_b(self, t, x):
    """ Compute coefficient b
    :param t: current time value
    :param x: current spot value
    """
    # Return the `b(t,x) coefficient

def coeff_c(self, t, x):
    """ Compute coefficient c
    :param t: current time value
    :param x: current spot value
    """
    # # Return the `c(t,x) coefficient

def coeff_d(self, t, x):
    """ Compute coefficient d
    :param t: current time value
    :param x: current spot value
    """
    # Return the `d(t,x) coefficient

def bound_cond_tup(self, x):
    """ Compute upper boundary condition for time
    :param x: current spot price
    """
    # Return the `f(x)` boundary condition

def bound_cond_xlow(self, t):
    """ Compute lower boundary condition for spot price
    :param t: current time
    """
    # Return the `f_l(t)` boundary condition
def bound_cond_xup(self, t):
    """ Compute upper boundary condition for spot price
    :param t: current time
    """
    # Return the `f_u(t)` boundary condition
```

Then we can create a `pde` object as follows:

```
# Set values for the underlying asset and boundary conditions
spot_init = 100
rate = 0.05
sigma = 0.2
strike = 100
time = 1/12
spot_low = 0
spot_up = 2*spot_init
# Initialise a PDE object representing the Black-Scholes equation
pde = EuPutBlackScholes(rate, sigma, strike, time, spot_low, spot_up)
```

5 Explicit Scheme

In this section we will create a new class for solving parabolic PDEs, such as the Black-Scholes equation, using the explicit finite difference scheme. The method works on a finite set

$$\{(t_i, x_j) : i = 0, \dots, i_{\max} \text{ and } j = 0, \dots, j_{\max}\}, \tag{8}$$

then taking

$$\begin{array}{cc} t_i = i\Delta t, & x_j = x_l + j\Delta x, \\ \Delta t = \frac{T}{i_{\max}} & \Delta x = \frac{x_u - x_l}{j_{\max}}, \end{array} \tag{9}$$

we can use the following notation for $i = 0, \dots, i_{\max}$ and $j = 0, \dots, j_{\max}$:

$$v_{i,j} = v(t_i, x_j), \tag{10}$$

$$\begin{array}{cc} a_{i,j} = a(t_i, x_j), & b_{i,j} = b(t_i, x_j), \\ c_{i,j} = c(t_i, x_j), & d_{i,j} = d(t_i, x_j), \end{array} \tag{11}$$

$$f_j = f(x_j), \quad f_{l,i} = f_l(t_i), \quad f_{u,i} = f_u(t_i). \tag{12}$$

The Explicit finite difference method operates on the following recursive formula:

$$v_{i-1,j} = A_{i,j} v_{i,j-1} + B_{i,j} v_{i,j} + C_{i,j} v_{i,j+1} + D_{i,j}, \tag{13}$$

where

$$\begin{array}{cc} A_{i,j} = \frac{\Delta t}{\Delta x}\left(\frac{b_{i,j}}{2} - \frac{a_{i,j}}{\Delta x}\right), & B_{i,j} = 1 - \Delta t c_{i,j} + \frac{2\Delta t a_{i,j}}{\Delta x^2}, \\ C_{i,j} = -\frac{\Delta t}{\Delta x}\left(\frac{b_{i,j}}{2} + \frac{a_{i,j}}{\Delta x}\right), & D_{i,j} = -\Delta t d_{i,j}. \end{array} \tag{14}$$

Since the following $v_{i,j}$ can be computed from the boundary conditions, we start from $i = i_{\max}$ and finish with $i = 0$.

$$v_{i_{\max},j} = f_j, \forall j = 0, \dots, j_{\max}. \tag{15}$$

Then using the boundary conditions we can also show that

$$v_{i-1,0} = f_{l,i-1}, \tag{16}$$

$$v_{i-1,j_{\max}} = f_{u,i-1}. \tag{17}$$

We will use this procedure to compute the $v_{i,j}$ values for all grid points.

But first we will create an *abstract* `PDESolver` class to represent all PDE solvers. Here we are taking advantage of class inheritance - a very important concept in OOP. The basic idea is that all (desired) functionality of the base class will be inherited by the derived classes. Then we will create each specific solver (the Explicit Scheme and the Implicit Scheme) as derived classes from this base class. This will help us avoid code duplication, as well as make our code more extendable. The `PDESolver` class will take a PDE object and number of discretisations for `t` and `x` variable. It will also

- compute and store sizes of each cell on the gird;
- initialise a gird matrix to store solutions;
- add a number of handy functions to easily access coefficients, `t` and `x` values of the gird, and boundary values;
- have an `interpolate()` function to interpolate a solution between grid points

```python
import numpy as np
# `numpy` is used for matrix operations, see https://numpy.org/

class PDESolver:
    """ Abstract class to solve Black-Scholes PDEs.
    """
    def __init__(self, pde, imax, jmax):
        """ Constructor
        :param pde: The PDE to solve
        :param imax: last value of the first variable's discretisation
        :param jmax: last value of the second variable's discretisation
        """
        # 1. Initialise `pde`, `imax`, `jmax` members
        # 2. Pre-compute `dt` and `dx` members for the grid
        # 3. Initialise a grid matrix to contain solutions using `np.empty`

    def _t(self, i):
        """ Return the discretised value of t at index i """
        return self.dt * i

    def _x(self, j):
        """ Return the discretised value of x at index j """
        return self.dx * j

    def _a(self, i, j):
        """ Helper umbrella function to get coefficient
        a at discretised locations """
        # Return the value from pde.coeff_a

    def _b(self, i, j):
        """ Helper umbrella function to get coefficient
        b at discretised locations """
        # Return the value from pde.coeff_b

    def _c(self, i, j):
        """ Helper umbrella function to get coefficient
        c at discretised locations """
        # Return the value from pde.coeff_c
    def _d(self, i, j):
        """ Helper umbrella function to get coefficient d at discretised locations """
        # Return the value from pde.coeff_d

    def _tup(self, j):
        """ Helper umbrella function to get upper boundary condition
        for time at discretised j index
        :param j: discretised x index """
        # Return the value from pde.bound_cond_tup

    def _xlow(self, i):
        """ Helper umbrella function to get lower boundary condition
        for x at discretised i index
        :param i: discretised t index """
        # Return the value from pde.bound_cond_xlow

    def _xup(self, i):
        """ Helper umbrella function to get upper boundary condition
        for x at discretised i index
        :param i: discretised t index """
        # Return the value from pde.bound_cond_xup

    def interpolate(self, t, x):
        """ Get interpolated solution value at given time and space
        :param t: point in time
        :param x: point in space
        :return: interpolated solution value """
        # 1. Compute the `i` and `j` indices on the grid for the
        #    input `t` and `x` continues values
        # 2. Compute the `l_1`, `l_0`, `w_1` and `w_o` values
        # 3. Return the interpolated value using the 4 grid
        #    points around (t,x) wieghted by
        #    the `l0`, `l1`, `w0`, `w1` coefficients
```

Now we are ready to implement the algorithm for the Explicit Scheme. We will create a `ExplicitScheme` class derived from `PDESolver`, then add functionality to

- compute values of the `A`, `B`, `C` and `D` coefficients, and
- iteratively solve the PDE on the grid points.

Listing 5: The ExplicitScheme class

```
class ExplicitScheme(PDESolver):
    """ Black-Scholes PDE solver using the explicit scheme
    """

    def __init__(self, pde, imax, jmax):
        super().__init__(pde, imax, jmax)

    def _A(self, i, j):
        """
        Coefficient A_{i,j} for the explicit scheme
        :param i: index of x discretisation
        :param j: index of t discretisation
        """
        # Compute and return the `A_{i,j}` coefficient's value

    def _B(self, i, j):
        """
        Coefficient B_{i,j} for the explicit scheme
        :param i: index of x discretisation
        :param j: index of t discretisation
        """
        # Compute and return the `B_{i,j}` coefficient's value

    def _C(self, i, j):
        """
        Coefficient C_{i,j} for the explicit scheme
        :param i: index of x discretisation
        :param j: index of t discretisation
        """
        # Compute and return the `C_{i,j}` coefficient's value

    def _D(self, i, j):
        """
        Coefficient D_{i,j} for the explicit scheme
        :param i: index of x discretisation
        :param j: index of t discretisation
        """
        # Compute and return the `D_{i,j}` coefficient's value

    def solve_grid(self):
        """
        Solves the PDE and saves the values in the member matrix `grid`
        """
        def _grid_entry(i, j): # You can define a function inside a function
            """ Helper function to compute the grid entry at point (i, j) """
            # Compute and return the solution on the `(i-1,j)` grid point
            # Using the `(i-1,j)`, `(i,j)` and `(i,j+1)` grid points

        # 1. compute all grid points for the last row:
        # 2. Iterate for all i from self.imax to 0 inclusive and update grid points:
        #    2.1. (i-1, 0) and (i-1, self.jmax) using the boundary conditions
        #    2.2. (i-1, 1:self.jmax-1) using the `_grid_entry` function defined above
```

Now we are ready to create solve the `pde` we created earlier using the `ExplicitScheme` class.

```
# Create a solver object for the explicit scheme
# Larger than 200x200 grid can make the algorithm too slow
exp_scheme = ExplicitScheme(pde, 200, 200)
# Solve the PDE on grid points
exp_scheme.solve_grid()
# Return the interpolated solution
sol_exp = exp_scheme.interpolate(0, spot_init)
# print the solution
print("Solution from the explicit scheme: {0}".format(sol_exp))
```

6 Implicit Scheme

In this section we will implement the finite difference Implicit Scheme for solving parabolic PDEs. Implicit schemes use different approximations for the partial gradients of v so that there is no explicit formula for $v_{i-1,j}$. Specifically, in this case the iterative formula can be written in a matrix form as follows:

$$\mathbf{v}_{i-1} = \mathbf{B}_i^{-1}(\mathbf{A}_i\mathbf{v}_i + \mathbf{w}_i), \tag{18}$$

where

$$\mathbf{A}_i = \begin{pmatrix} B_{i,1} & C_{i,1} & 0 & 0 & \dots & 0 \\ A_{i,2} & B_{i,2} & C_{i,2} & 0 & \dots & 0 \\ 0 & A_{i,3} & B_{i,3} & C_{i,3} & \ddots & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & C_{i,j_{\max}-2} \\ 0 & 0 & \dots & 0 & A_{i,j_{\max}-1} & B_{i,j_{\max}-1} \end{pmatrix}, \tag{19}$$

$$\mathbf{B}_i = \begin{pmatrix} F_{i,1} & G_{i,1} & 0 & 0 & \dots & 0 \\ E_{i,2} & F_{i,2} & G_{i,2} & 0 & \dots & 0 \\ 0 & E_{i,3} & F_{i,3} & G_{i,3} & \ddots & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & G_{i,j_{\max}-2} \\ 0 & 0 & \dots & 0 & E_{i,j_{\max}-1} & F_{i,j_{\max}-1} \end{pmatrix}, \tag{20}$$

$$\mathbf{v}_i = \begin{pmatrix} v_{i,1} \\ \vdots \\ v_{i,j_{\max}-1} \end{pmatrix}, \tag{21}$$

$$\mathbf{w}_i = \begin{pmatrix} D_{i,1} + A_{i,1}f_{l,i} - E_{i,1}f_{l,i-1} \\ D_{i,2} \\ \vdots \\ D_{i,j_{\max}-2} \\ D_{i,j_{\max}-1} + C_{i,j_{\max}-1}f_{u,i} - G_{i,j_{\max}-1}f_{u,i-1} \end{pmatrix}. \tag{22}$$

The scalars A,...G are defined as follows:

$$\begin{gathered} A_{i,j} = 0, \quad B_{i,j} = 1, \quad C_{i,j} = 0, \\ D_{i,j} = -\Delta t d_{i-1,j}, \\ E_{i,j} = -\frac{\Delta t}{\Delta x}\left(\frac{b_{i-1,j}}{2} - \frac{a_{i-1,j}}{\Delta x}\right), \\ F_{i,j} = 1 + \Delta t c_{i-1,j} - \frac{2\Delta t a_{i-1,j}}{\Delta x^2}, \\ G_{i,j} = \frac{\Delta t}{\Delta x}\left(\frac{b_{i-1,j}}{2} + \frac{a_{i-1,j}}{\Delta x}\right). \end{gathered} \tag{23}$$

To implement the Implicit Scheme, we will

1. Create a new `ImplicitScheme` class derived from `PDESolver`;
2. implement a new member functions for coefficients $\mathbf{A}_{i,j}$ to $\mathbf{G}_{i,j}$
3. implement a new member function `get_w()` to compute the vector $\mathbf{w}$;
4. implement a new `compute_vi()` member function to compute $\mathbf{v}_{i-1}$ using 18
5. implement a new `solve_grid()` member function to iteratively compute $\mathbf{v}_i$s using boundary conditions and `compute_vi()`

Listing 6: The ImplicitScheme class

```
# `scipy` is a comprehensive module for scientific computing,
# see https://www.scipy.org/
# here we use its `sparse` submodule to work with sparse matrices
# used for the Implicit Scheme
import scipy.sparse

class ImplicitScheme(PDESolver):
    """ Black-Scholes PDE solver using the implicit scheme
    """

    def __init__(self, pde, imax, jmax):
        super().__init__(pde, imax, jmax)

    def _A(self, i, j):
        """
        Coefficient A_{i,j} for the implicit scheme
        :param i: index of x discretisation
        :param j: index of t discretisation
        """
        # Compute and return the `A_{i,j}` coefficient's value

    def _B(self, i, j):
        """
        Coefficient B_{i,j} for the implicit scheme
        :param i: index of x discretisation
        :param j: index of t discretisation
        """
        # Compute and return the `B_{i,j}` coefficient's value

    def _C(self, i, j):
        """
        Coefficient C_{i,j} for the implicit scheme
        :param i: index of x discretisation
        :param j: index of t discretisation
        """
        # Compute and return the `C_{i,j}` coefficient's value

    def _D(self, i, j):
        """
        Coefficient D_{i,j} for the implicit scheme
        :param i: index of x discretisation
        :param j: index of t discretisation
        """
        # Compute and return the `D_{i,j}` coefficient's value

    def _E(self, i, j):
        """
        Coefficient E_{i,j} for the implicit scheme
        :param i: index of x discretisation
        :param j: index of t discretisation
        """
        # Compute and return the `E_{i,j}` coefficient's value
    def _F(self, i, j):
        """
        Coefficient F_{i,j} for the implicit scheme
        :param i: index of x discretisation
        :param j: index of t discretisation
        """
        # Compute and return the `F_{i,j}` coefficient's value

    def _G(self, i, j):
        """
        Coefficient G_{i,j} for the implicit scheme
        :param i: index of x discretisation
        :param j: index of t discretisation
        """
        # Compute and return the `G_{i,j}` coefficient's value

    def get_w(self, i):
        """
        Compute the intermediate vector w_i used to compute the right-hand-side
        of the linear system of equation in the implicit scheme
        :param i: index of x discretisation
        :return: a numpy array of [w_1, ..., w_jmax-1] entries
        """
        # 1. Initialise `w` as a list with a single element
        # 2. Use Python's built-in `extend()` method to add
        #    the remaining values until and excluding the last one
        # 3. Use Python's built-in `append()` method to add the last value

    def compute_vi(self, i):
        """
        Compute the v_{i-1} vector solving the (18) inverse problem
        Uses the precomputed values of grid[i, :] to compute grid[i-1, :]
        :param i: i-th iteration
        :return: the v_{i-1} vector, numpy array of length `self.jmax+1`
                 Doesn't include the first and last entries
                 which are computed using the boundary conditions
        """
        # 1. Compute the 3 diagonals of the matrix A
        # 2. Use `scipy.sparse.diags` to create the A tri-diagonal matrix
        # 3. Compute the 3 diagonals of the matrix B
        # 4. Use `scipy.sparse.diags` to create the B tri-diagonal matrix
        # 5. Compute the right hand side of the linear system of equations
        # 6. Solve the linear system of equations using `np.linalg.solve`
        # 7. Return the solution

    def solve_grid(self):
        """ Iteratively solve the PDE for the grid values
        """
        # 1. Compute the last row of the grid using the boundary condition on `t`
        # 2. Iterate `i` from the last row to the the first (exclusive) and
        #    2.1. Set the elements on row `i-1` exluding the edges
        #    using `self.compute_vi()`
        #    2.2. Set the edges on row `i-1` using the boundary conditions
```

Now we can use the `ImplicitScheme` to solve the same PDE:

```
# Create a solver object for the implicit scheme
# Since we use the very effiicient `scipy` and `numpy` libraries,
# here we can experiment with more granular grids - try 1000x1000
# Solve the PDE on grid points
# Return the interpolated solution
# print the solution
```

7 Final Notes

For homework, consider implementing a new `CrankNicolsonScheme` derived from `ImplicitScheme` and override `A()`, `B()`, `C()`, `D()`, `E()` and `F()` member functions accordingly.

References

[1] Maciej J. Capiński and Tomasz Zastawniak. *Numerical Methods in Finance with C.* Mastering Mathematical Finance. Cambridge University Press, 2012.