Pricing Engine Architecture v1.1

Current State of the Architecture

- The first draft has been reorganized to the structure detailed in the previous presentation.

- Now, we are at v1.1, where we have addressed the challenges/bottlenecks laid out in the previous presentation

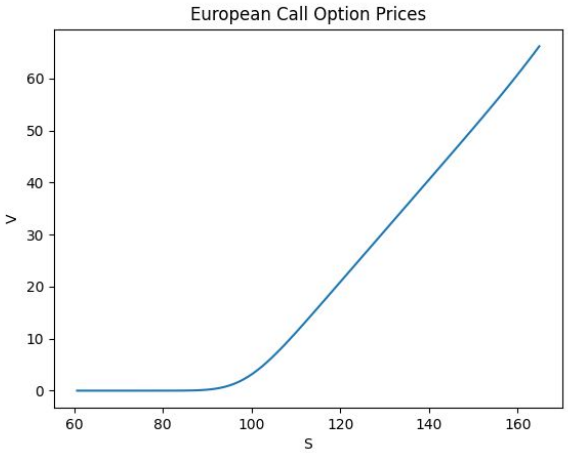
- Among the options currently reproducible are:
  - American/European Vanilla
  - American/European Binary

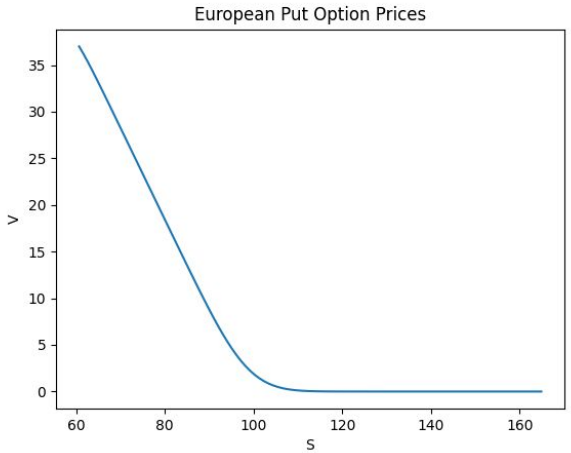
- Knockout barrier options are possible, but currently only European-style (Exercise only at maturity) for now. To implement American-style Barriers eventually.

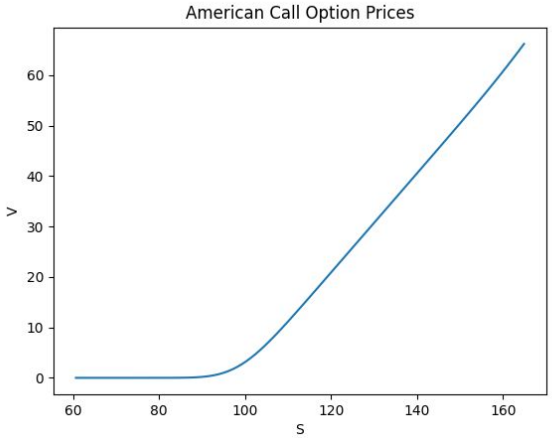
Vanilla Prices

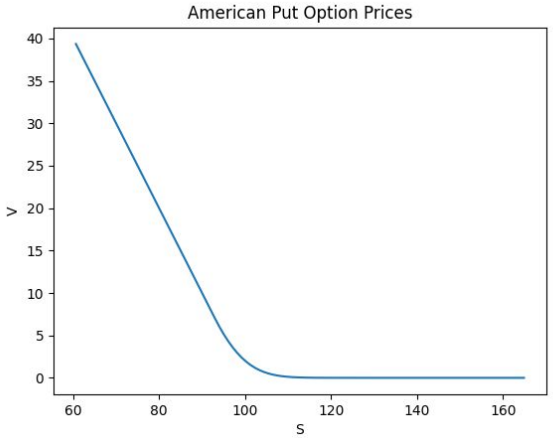
K = 100; r = 0.03,
q = 0.01; vol = 0.1;
T = 1 year

$$Payoff = \max(S - K, 0)$$





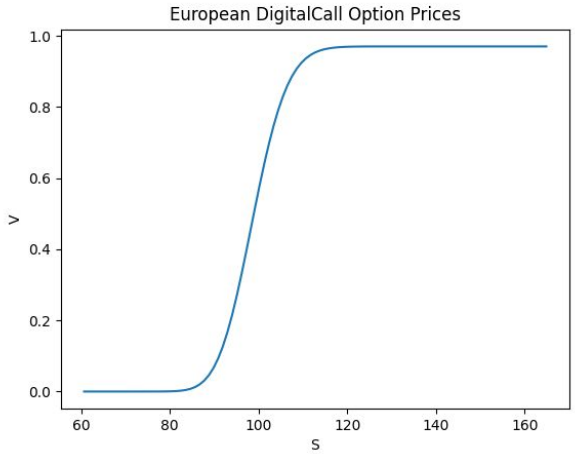


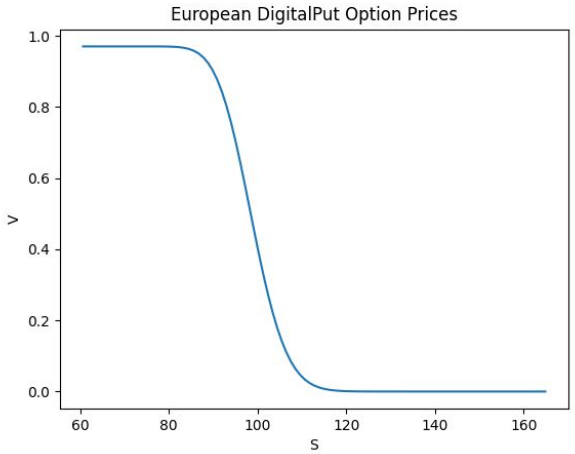


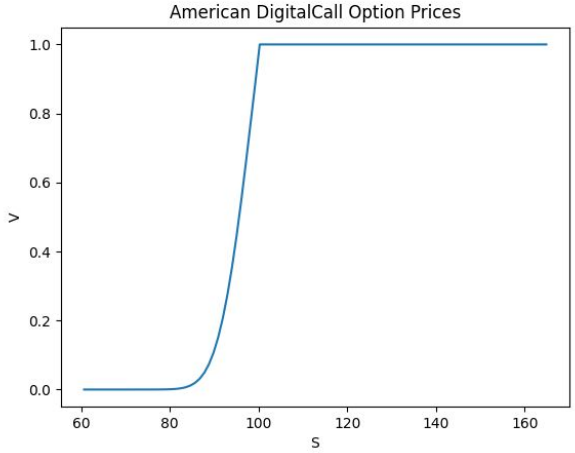
Digital Prices

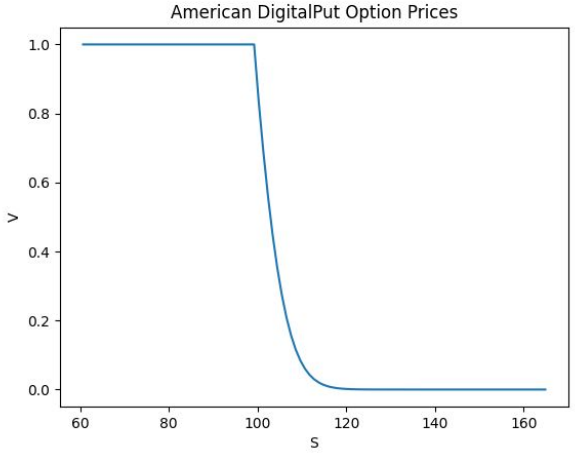
K = 100; r = 0.03,
q = 0.01; vol = 0.1;
T = 1 year

$$Payoff = \begin{cases} 1 & \text{if } S > K \\ 0 & \text{otherwise} \end{cases}$$









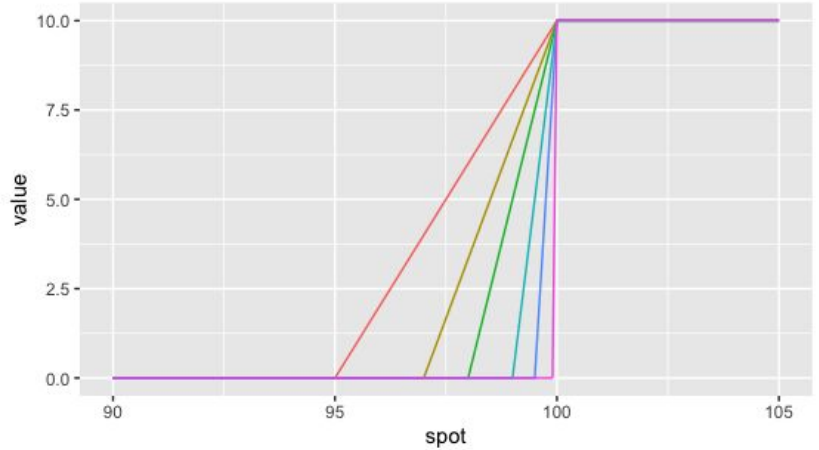
State of bottlenecks after applying the Open-Closed Principle

- **Overly-Conditional/Interdependent Structure**
  - Significantly reduced
  - Each class has only one job
  - Each implementation is always based on an Interface/Abstraction

- **Extensibility of Code**
  - Extending code requires only adding the necessary classes, e.g:
  - *New European-Style Option (Barrier, shark-fin etc.)*:
    - Add new PayOff and Boundary classes inheriting from the resp. classes.
  - *Stochastic Volatility Pricer (Heston, SABR, etc.):*
    - Define a new abstract class PDEStochVol
    - Define PDEHeston and PDESABR etc. as children of PDEStochVol

Example - Pricing a Digital Put

- Say we want to price an ATM European digital option, DP(S,t) with the initial parameters:
  - S = 100
  - K = 100
  - r = 0.03
  - q = 0.01
  - Vol = 0.1
  - T = 1



Pricing a Digital Put - Classes to Import

```python
# Import Option classes
from .options.vanilla import VanillaOption
from .options.payoff.payoff import PayOffDigitalPut
from .options.payoff.PayOffFactory import PayOffFactory

# Import boundary classes
from .options.boundary.BoundaryFactory import BoundaryFactory, BoundaryDigitalPut

# Import PDE class
from .pde_params.pdebs import PDEBlackScholes

# Import FDM pricer classes
from .engine.scheme.cranknicolson import FDMEulerCrankNicolson
```

The only classes necessary to import:

- **Option information** -
  - PayOffXXX/PayOffFactory
  - VanillaOption
  - BoundaryXXX/ BoundaryFactory
- **FDM Scheme** -
  - Explicit
  - Implicit/Crank-Nicolson
- **The actual solver class** -
  - PDEBlackScholes

*Note: There are (abstract) classes under the hood like PDEConvectionDiffusion, PayOff and Boundary, but these should **never** be imported when pricing*

Pricing a Digital Put - Parameters to define

```
S = 100 # Base spot price
K = 100 # Strike
r = 0.03 # Risk-free rate
q = 0.01 # Dividend rate
T = 1 # Time to expiry
vol = 0.1 # Volatility
American = False # American or European option

N_x = 100 # Number of space grid points
N_t = 200 # Number of time grid points

FD_class = FDMEulerCrankNicolson # FD scheme to use
mode = "fast" # Matrix inversion scheme to use

# Register boundary and payoff
PayOffFactory.RegisterPayOff("PayOffDigitalPut", PayOffDigitalPut)
BoundaryFactory.RegisterBoundary("PayOffDigitalPut", BoundaryDigitalPut)
```

Pricing a Digital Put - Initializing the Option

```
# Initialize Payoff
payoffdigitalput = PayOffFactory.CreatePayoff("PayOffDigitalPut", Strike=K)

# Initializing Option
vanillacall = VanillaOption(
    OptionPayoff = payoffdigitalput,
    Expiry       = T,
    RiskFreeRate = r,
    DividendRate = q, # OPTIONAL
    Volatility   = vol
)
```

- We initialize the PayOff from the Factory
- We can just key in the option parameters
- Note the dividend rate is optional

Pricing a Digital Put - Pricing the Option

```
# Initializing and calculating PDE solutions
pde_bs = PDEBlackScholes(
    Option=vanillacall,
    Current_Spot=S,
    N_x=N_x,
    N_t=N_t,
    American=False,
    FDMclass=FDMEulerCrankNicolson,
    mode="fast" # OPTIONAL - Defaults to "fast"
)
```

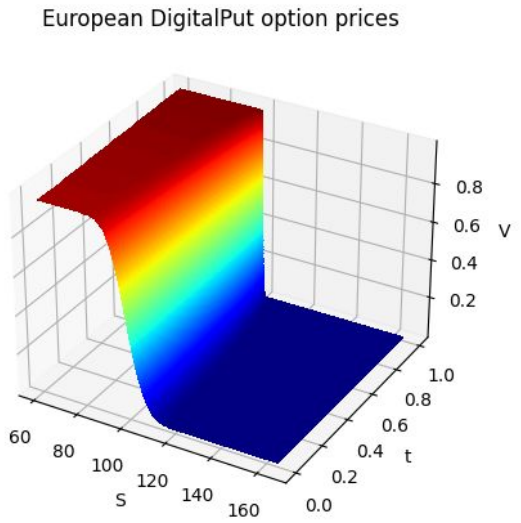
- Just call PDEBlackScholes with our parameters

Pricing a Digital Put - Retrieving the solution

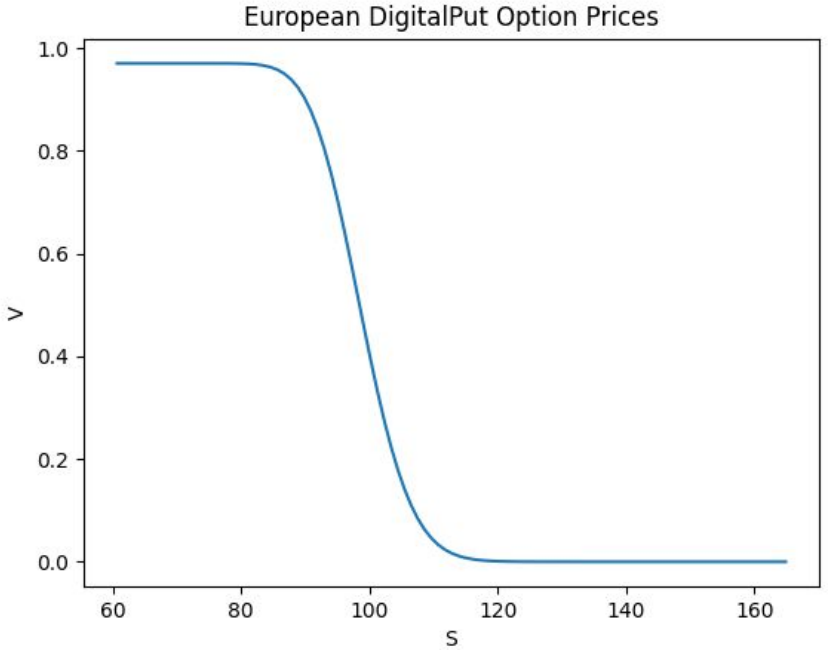
```
# Obtaining solution
solution = pde_bs.get_solution()

# Obtaining entire grid of solutions
grid = pde_bs.get_grid()
```

- get_solution():
  - Retrieves an array of solutions at t=0, varying between spot price
- get_grid():
  - Retrieves the entire grid of solutions (V(x,t))

Pricing a Digital Put - The Results



Price surface for a range of (S,t)



Prices for various S at t=0

Choice of Finite Difference Methods

We have a choice of 3 finite difference methods to choose from:

- **Explicit** (FDMEulerExplicit) -
  - Fastest (About O(n) where n is number of time-points)
  - Conditionally Stable (Requires Nt >> Nx for accurate solutions)
  - Accuracy is O(m^2, n) where m is number of space-points

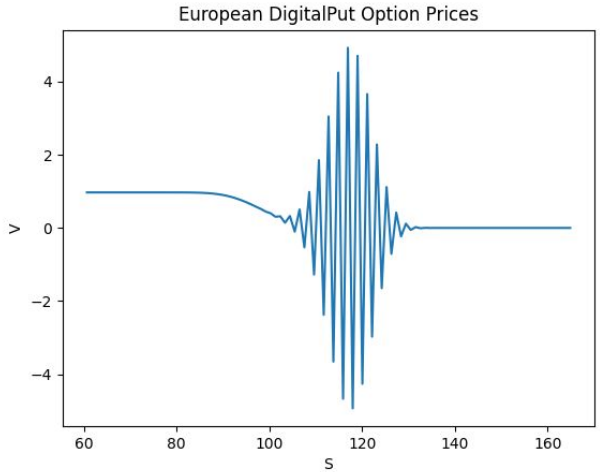
- **Implicit** (FDMEulerImplicit) -
  - Slow (Requires matrix inversion which is O(n^3))
  - Accuracy is O(m^2, n^2)
  - Unconditionally Stable

- **Crank-Nicolson** (FDMEulerCrankNicolson) -
  - Slow (Requires matrix inversion, about O(n^3))
  - Accuracy is O(m^2, n^2)
  - Unconditionally Stable

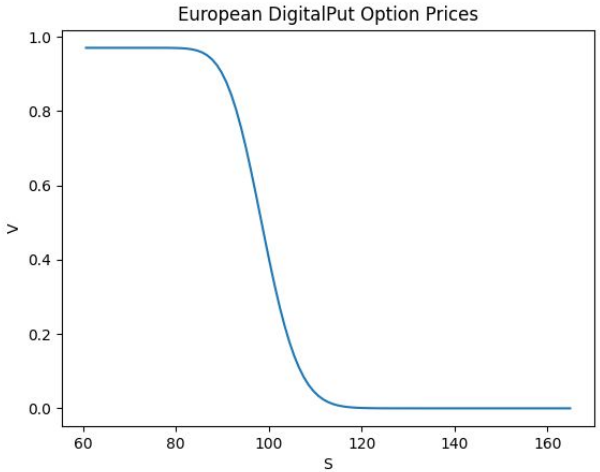
Speed/Efficiency

- As mentioned earlier, the *explicit* method is the fastest at about O(n), **but** it gets quite unstable if there aren't enough time-points.

- In fact, for a 10x increase in space-points, we'd need a 100x increase in time points.



$$N_t < CN_x^2$$



$$N_t \geq CN_x^2$$

Stability

The implicit and Crank-Nicolson methods are unconditionally stable (provided the payoff is nice enough).

This means that it needs less points than the explicit method.

However, it is much slower as it requires inverting a matrix of size N_x × N_t for all time-steps t

```
Option payoff: PayOffDigitalPut
Scheme: FDMEulerExplicit
Mode: fast

N_x=100
N_t=200

Elapsed time: 0.008129889000000001 seconds
```

```
Option payoff: PayOffDigitalPut
Scheme: FDMEulerCrankNicolson
Mode: gaussian

N_x=100
N_t=50

Elapsed time: 1.202766457 seconds
```

Circumventing the tradeoff

However, it turns out that this isn't an issue:

- The matrix to invert is a tri-diagonal matrix (non-zero entries on only the diagonal, super-diagonal and sub-diagonal)
- There is an algorithm to invert it in O(n) time (Tridiagonal matrix algorithm)

This drastically speeds up the implicit/Crank-Nicolson methods

```
Option payoff: PayOffDigitalPut
Scheme: FDMEulerCrankNicolson
Mode: fast

N_x=100
N_t=50

Elapsed time: 0.04317614800000004 seconds
```

$$\begin{pmatrix} 1 & 4 & 0 & 0 \\ 3 & 4 & 1 & 0 \\ 0 & 2 & 3 & 4 \\ 0 & 0 & 1 & 3 \end{pmatrix}.$$

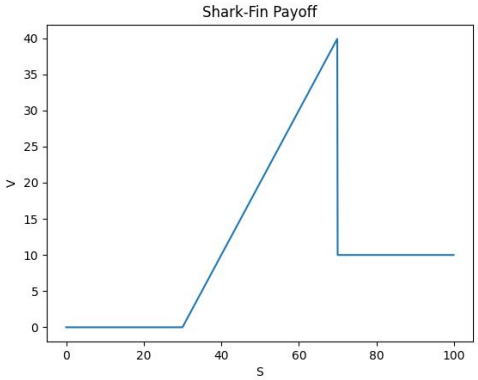
TODO

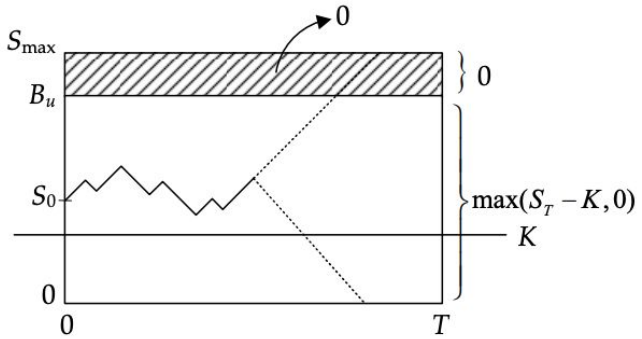
- Implement testing against exact pricers whenever applicable
- New contracts
- Numerical Greek calculator
- Implementing new PDEs

Future Plans - Implementing New Contracts

- Can immediately implement:
  - European-style Barrier Knockout options
  - Any other European (Exercise at expiry) contracts

- Requires new abstract classes:
  - American-Style Barrier Knockout Options (shark-fin etc)
    - This will require a grid class, new boundary classes and a modification of FDM as the boundary conditions include a portion of the interior of the grid





Future Plans - Numerical Greek Calculator

- To obtain a grid of option Greek approximations (Theta, Delta and Gamma), we may apply finite differences to the pricing grid we receive.

- Pros:
  - We do not need a closed-form formula.
  - We get greeks ranging over multiple (S,t) in one go.

- Cons:
  - To obtain more accurate readings, Nx and Nt must be increased
  - If the payoff function is not smooth, there may be jumps and spurious oscillations
  - Other greeks like Rho, Vega will require multiple FDM grids to evaluate

Future Plans - Implementing New PDEs

- Recall that the Black-Scholes Model has constant volatility
  - Not representative of the market

- If we consider stochastic volatility, we can consider options under said model
- Consider the Heston model:

$$\begin{cases} dS_t & = rS_t dt + S_t\sqrt{V_t}dW_t^1, \\ dV_t & = \kappa(\eta - V_t)dt + \sigma\sqrt{V_t}dW_t^2, \\ dW_t^1 dW_t^2 & = \rho dt, \end{cases}$$

- We can derive a 2-space-dimensional PDE with respect to time and (S,V):

$$\frac{1}{2}\sigma^2 v\frac{\partial^2 u}{\partial v^2} + \sigma sv\rho\frac{\partial^2 u}{\partial s\partial v} + \frac{1}{2}vs^2\frac{\partial^2 u}{\partial s^2} + \kappa(\eta - v)\frac{\partial u}{\partial v} + rs\frac{\partial u}{\partial s} + \frac{\partial u}{\partial t} - ru = 0.$$

Stochastic Volatility Model - Challenges

- We now have to iterate over matrix inversion of order N_t * (N_x * N_v) which can get huge really quickly.
- To handle them, we need to be clever and use FD schemes that work implicitly on parts of the matrix that are quick to invert and explicit schemes for the parts that are hard to invert.
- One of the boundary conditions may involve solving a 1-space-dimensional PDE.
- There are schemes, such as the Alternating Direction Implicit (ADI) scheme or the Douglas scheme that help make solving said PDEs more efficient.
- Will require a new abstract PDE module and its own boundary condition files

  For more information: Finite Difference Methods in Derivatives Pricing under Stochastic Volatility Models.

Thank you