PSTAT 222C Homework 3

Alex Bernstein

```python
import matplotlib.pyplot as plt
import numpy as np
from tqdm import tqdm,trange
from numba import njit,cuda
import copy
import pdb
import time
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import pandas as pd
import sklearn.gaussian_process as gp
from sklearn.gaussian_process.kernels import Matern
plt.rcParams['text.usetex'] = True
from IPython.display import display, Markdown
```

Problem 1

Problem 1a

Derivation of Finite Difference

We have the Heston Stochastic Volatility Model with:

$$dS_t = rS_t dt + S_t\sqrt{V_t}dW_t^1$$
$$dV_t = \kappa(\theta - V_t)dt + \eta\sqrt{V_t}dW_t^2$$

Applying the multivariate Ito's Lemma, to a function $f(t, V, S)$ (and excluding the $\frac{\partial f}{\partial t}$ term because it is 0) we have:

$$df(t, V, S) = \frac{\partial f}{\partial t}dt + \frac{\partial f}{\partial S}dS_t + \frac{\partial f}{\partial V}dV_t + \frac{1}{2}\frac{\partial^2 f}{\partial S^2}dS_t^2 + \frac{1}{2}\frac{\partial^2 f}{\partial V^2}dV_t^2 + \frac{\partial^2 f}{\partial S\partial V}dS_t dV_t$$

where we also have that, by the rules of Ito calculus:

$$dS_t^2 = S_t^2 V_t dt \quad dV_t^2 = \eta^2 V_t dt \quad dS_t dV_t = \rho\eta S_t V_t dt$$

and so we obtain the following differential:

$$\begin{aligned} df(t,V,S) &= \frac{\partial f}{\partial t}dt + \frac{\partial f}{\partial S}\left(rS_t dt + S_t\sqrt{V_t}dW_t^1\right) + \frac{\partial F}{\partial V}\left(\kappa(\theta - V_t)dt + \eta\sqrt{V_t}dW_t^2\right) \\ &\quad + \left(\frac{\partial^2 f}{\partial S^2}\frac{S_t^2 V_t}{2} + \frac{\partial^2 f}{\partial V^2}\frac{\eta^2 V_t}{2} + \rho\eta S_t V_t \frac{\partial f}{\partial S \partial V}\right)dt \\ &= \left(\frac{\partial f}{\partial t} + rS_t\frac{\partial f}{\partial S} + \kappa(\theta - V_t)\frac{\partial F}{\partial V} + \frac{S_t^2 V_t}{2}\frac{\partial^2 f}{\partial S^2} + \frac{\eta^2 V_t}{2}\frac{\partial^2 f}{\partial V^2} + \rho\eta S_t V_t\frac{\partial f}{\partial S\partial V}\right)dt \\ &\quad + S_t\sqrt{V_t}\frac{\partial F}{\partial S}dW_t^1 + \eta\sqrt{V_t}\frac{\partial F}{\partial V}dW_t^2 \end{aligned}$$

Taking the risk-neutral Expectation of both sides, we find that the martingale terms are 0, with the rest becoming:

$$\mathbb{E}[df(t,V,S)] = rf(t,V,S)dt = \left(\frac{\partial f}{\partial t} + rS_t\frac{\partial f}{\partial S} + \kappa(\theta - V_t)\frac{\partial F}{\partial V} + \frac{S_t^2 V_t}{2}\frac{\partial^2 f}{\partial S^2} + \frac{\eta^2 V_t}{2}\frac{\partial^2 f}{\partial V^2} + \rho\eta S_t V_t\frac{\partial f}{\partial S\partial V}\right)dt$$

After doing a time change to $t = T - t$ (which only flips the sign on $\partial_t f$ from the chain rule) we have a more familiar PDE:

$$\frac{\partial f}{\partial t} = rS_t\frac{\partial f}{\partial S} + \kappa(\theta - V_t)\frac{\partial f}{\partial V} + \frac{S_t^2 V_t}{2}\frac{\partial^2 f}{\partial S^2} + \frac{\eta^2 V_t}{2}\frac{\partial^2 f}{\partial V^2} + \rho\eta S_t V_t\frac{\partial f}{\partial S\partial V} - rf$$

Now, we have two spatial coordinates and one time coordinate, so we will use the notation that:

$f(t,V,S) = f(m\Delta t, j\Delta s, k\Delta v) = f_{j,k}^m$

We use the following approximations for the partial derivatives:

- First Order Derivatives:

$$\frac{\partial f}{\partial t} = \frac{f_{j,k}^{m+1} - f_{j,k}^m}{\Delta t} \qquad \frac{\partial f}{\partial V} = \frac{f_{j,k+1}^m - f_{j,k-1}^m}{2\Delta v} \qquad \frac{\partial f}{\partial S} = \frac{f_{j+1,k}^m - f_{j-1,k}^m}{2\Delta s}$$

- Second Order Standard Derivatives:

$$\frac{\partial^2 f}{\partial V^2} = \frac{f_{j,k+1}^m - 2f_{j,k}^m + f_{j,k-1}^m}{(\Delta s)^2} \qquad \frac{\partial^2 f}{\partial S^2} = \frac{f_{j+1,k}^m - 2f_{j,k}^m + f_{j-1,k}^m}{(\Delta v)^2}$$

- Second Order Mixed Derivatives:

$$\frac{\partial^2 f}{\partial S\partial V} = \frac{f_{j+1,k+1}^m + f_{j-1,k-1}^m - f_{j+1,k-1}^m - f_{j-1,k+1}^m}{4\Delta s\Delta v}$$

Our numeric simulation therefore solves:

$$\begin{aligned}\frac{f_{j,k}^{m+1}-f_{j,k}^{m}}{\Delta t} &= rS_j\frac{f_{j+1,k}^{m}-f_{j-1,k}^{m}}{2\Delta s}+\kappa(\theta-V_k)\frac{f_{j,k+1}^{m}-f_{j,k-1}^{m}}{2\Delta v}\\ &+\frac{1}{2}V_kS_j^2\left(\frac{f_{j+1,k}^{m}-2f_{j,k}^{m}+f_{j-1,k}^{m}}{(\Delta s)^2}\right)+\frac{1}{2}\eta^2V_k\left(\frac{f_{j,k+1}^{m}-2f_{j,k}^{m}+f_{j,k-1}^{m}}{(\Delta v)^2}\right)\\ &+\rho\eta S_jV_k\left(\frac{f_{j+1,k+1}^{m}+f_{j-1,k-1}^{m}-f_{j+1,k-1}^{m}-f_{j-1,k+1}^{m}}{4\Delta s\Delta v}\right)-rf_{j,k}^{m}\end{aligned}$$

Letting $S_j = j\Delta s$ and $V_k = k\Delta v$, we find:

$$\begin{aligned}\frac{f_{j,k}^{m+1}-f_{j,k}^{m}}{\Delta t} &= rj\frac{f_{j+1,k}^{m}-f_{j-1,k}^{m}}{2}+\kappa(\theta-k\Delta v)\frac{f_{j,k+1}^{m}-f_{j,k-1}^{m}}{2\Delta v}\\ &+\frac{1}{2}kj^2\Delta v\left(f_{j+1,k}^{m}-2f_{j,k}^{m}+f_{j-1,k}^{m}\right)+\frac{1}{2}\eta^2k\left(\frac{f_{j,k+1}^{m}-2f_{j,k}^{m}+f_{j,k-1}^{m}}{\Delta v}\right)\\ &+\rho\eta jk\left(\frac{f_{j+1,k+1}^{m}+f_{j-1,k-1}^{m}-f_{j+1,k-1}^{m}-f_{j-1,k+1}^{m}}{4}\right)-rf_{j,k}^{m}\end{aligned}$$

Collecting terms, solving for $f_{j,k}^{m+1}$ we find:

$$\begin{aligned}f_{j,k}^{m+1} &= f_{j,k}^{m}\left(1-r\Delta t-j^2k\Delta v\Delta t-\frac{\eta^2k\Delta t}{\Delta v}\right)\\ &+f_{j,k-1}^{m}\left(\frac{k\eta^2\Delta t}{2\Delta v}-\frac{\kappa(\theta-k\Delta v)\Delta t}{2\Delta v}\right)+f_{j,k+1}^{m}\left(\frac{k\eta^2\Delta t}{2\Delta v}+\frac{\kappa(\theta-k\Delta v)\Delta t}{2\Delta v}\right)\\ &+f_{j-1,k}^{m}\left(\frac{j^2k\Delta v\Delta t}{2}-\frac{rj\Delta t}{2}\right)+f_{j+1,k}^{m}\left(\frac{j^2k\Delta v\Delta t}{2}+\frac{rj\Delta t}{2}\right)\\ &+\frac{\rho\eta jk\Delta t}{4}\left(f_{j+1,k+1}^{m}+f_{j-1,k-1}^{m}-f_{j+1,k-1}^{m}-f_{j-1,k+1}^{m}\right)\end{aligned}$$

Boundary Conditions

At each time point, there exists a $N_s \times N_v$ grid of points. We need to solve for the values at the boundaries where $S = 0$, $S = N_S$ and $V = 0$, $V = N_V$.

- **Lower Stock Boundary** ($S_t = 0$): When $S = 0$, $(K - S_t)_+ \approx K$, and so we can simply discount the strike price to the start. Hence $f_{0,k}^m = e^{-rt}K$.
- **Upper Stock Boundary** ($S_t = N_S$): As the stock price gets larger, the value of the option approaches 0 as the option is less likely to be exercised at any point. Hence, we im pose that $f_{N_s,k}^m = 0$ at this boundary.
- **Lower Volatility Boundary** ($V_t = 0$): As V gets close to 0, the value of the stock becomes deterministic in the Heston model, and thus $\frac{\partial^2 f}{\partial V^2} \overset{V\to 0}{\to} 0$. Hence, solving the expression for $\frac{\partial^2 f}{\partial V^2}$ tells us that $f_{j,0}^m = 2f_{j,1}^m - f_{j,2}^m$
- **Upper Volatility Boundary** ($V_t = N_V$): Determining an appropriate boundary condition for high volatility is difficult, because volatility is only "bounded" due to computational constraints (we clearly cannot simulate

an infinite grid). Because there is no convenient approach for this, we will assume that the change in volatility near the upper boundary is low, and set $f^m_{j,N_V} = f^m_{j,N_V-1}$.

- **Terminal Condition**: At maturity index $I = T/\Delta t$ (corresponding to index 1 in our code because of the reversed time), the value of the put option is simply $f^1_{j,k} = (K - j\Delta s)_+$, for each value in the grid.

Code

Note that the code for Problem 1a and Problem b are nearly identical; we will describe the δ_{freq} (and equivalent `freq` code parameter) term in Problem 1b, but the value in the standard American Option setting is that $\delta_{freq} = dt$. This effectively means that every timestep is a valid stopping time.

Also note that, in order to allow compilation in a reasonable amount of time, the Numba Just-In-Time Compiler is used frequently throughout both problems. See Lam, Pitrou, and Seibert (2015).

```
@njit
def explicit_FD_put(dt,ds,dv,K,freq=None):
    if(freq==None):
        freq = dt
    # Heston Model
    r = 0.05
    kappa = 1
    theta = 0.2
    eta = 0.5
    rho = -0.4

    # Option
    T=1
    S_max = 2.5*K
    V_min = 0.05
    V_max = 0.6
    # Stop set defintion.  time between allowed stops is freq/dt; can stop at T/freq indicies
    # Default is freq = dt; this is American option
    stop_set = np.arange((round(T/freq)))*(freq/dt)

    # Discretization
    time_n = round(T/dt)
    asset_n = round(S_max/ds)+1
    vol_n = round((V_max-V_min)/dv)+1
    vetS = ds*np.arange(asset_n)

    payoff_exit = np.maximum(K-vetS,0).repeat(vol_n).reshape((-1,vol_n))
    f = np.copy(payoff_exit)*1.0
    for i in range(time_n):
        newf = np.zeros((asset_n,vol_n))
        for j in range(1,asset_n-1):
            for k in range(1,vol_n-1):
                A = np.zeros((3,3))
                A[2,2] = A[0,0] = rho*eta*j*(V_min/dv+k)*dt/4
                A[0,2] = A[2,0] = -rho*eta*j*(V_min/dv+k)*dt/4
                A[1,1] = 1-r*dt - j**2*(V_min/dv+k)*dv*dt - eta**2*(V_min/dv+k)*dt/dv
                A[0,1] = (j**2*(V_min/dv + k)*dv*dt/2-r*j*dt/2)
                A[2,1] = r*j*dt/2 + j**2*(V_min/dv + k)*dv*dt/2
                A[1,0] = (eta**2*(V_min/dv + k)*dt/(2*dv)- \
                    dt*kappa*(theta - (V_min/dv + k)*dv)/ (2*dv))
                A[1,2] = dt*kappa*(theta - (V_min/dv + k)*dv)/(2*dv) +\
                     eta**2*(V_min/dv + k)*dt/(2*dv)
                newf[j,k] = (
                    A[0,0]*f[j-1,k-1] + A[0,1]*f[j-1,k] + A[0,2]*f[j-1,k+1] +
                    A[1,0]*f[j,k-1] + A[1,1]*f[j,k] + A[1,2]*f[j,k+1] +
                    A[2,0]*f[j+1,k-1]+A[2,1]*f[j+1,k]+A[2,2]*f[j+1,k+1])

        # Lower V Boundary
        newf[1:asset_n-1,0] = 2*newf[1:asset_n-1,1]-newf[1:asset_n-1,2]

        # Upper V Boundary
        newf[1:asset_n-1,vol_n-1] = newf[1:asset_n-1,vol_n-2]

        # Lower S Boundary
        newf[0,:] = K*np.exp(-r*(i+1)*dt)

        if(i in stop_set):
            f = np.maximum(payoff_exit,newf)
        else:
            f = newf
    return f

# model param setup
K, dt, ds, dv, v_min = 100,1E-5,1,0.05,0.05
prices = explicit_FD_put(dt,ds,dv,K,dt)
s0,v0 = 100, 0.25
ind_s0,ind_v0 = round(s0/ds),round((v0-v_min)/dv)
price_out = prices[ind_s0,ind_v0]
```

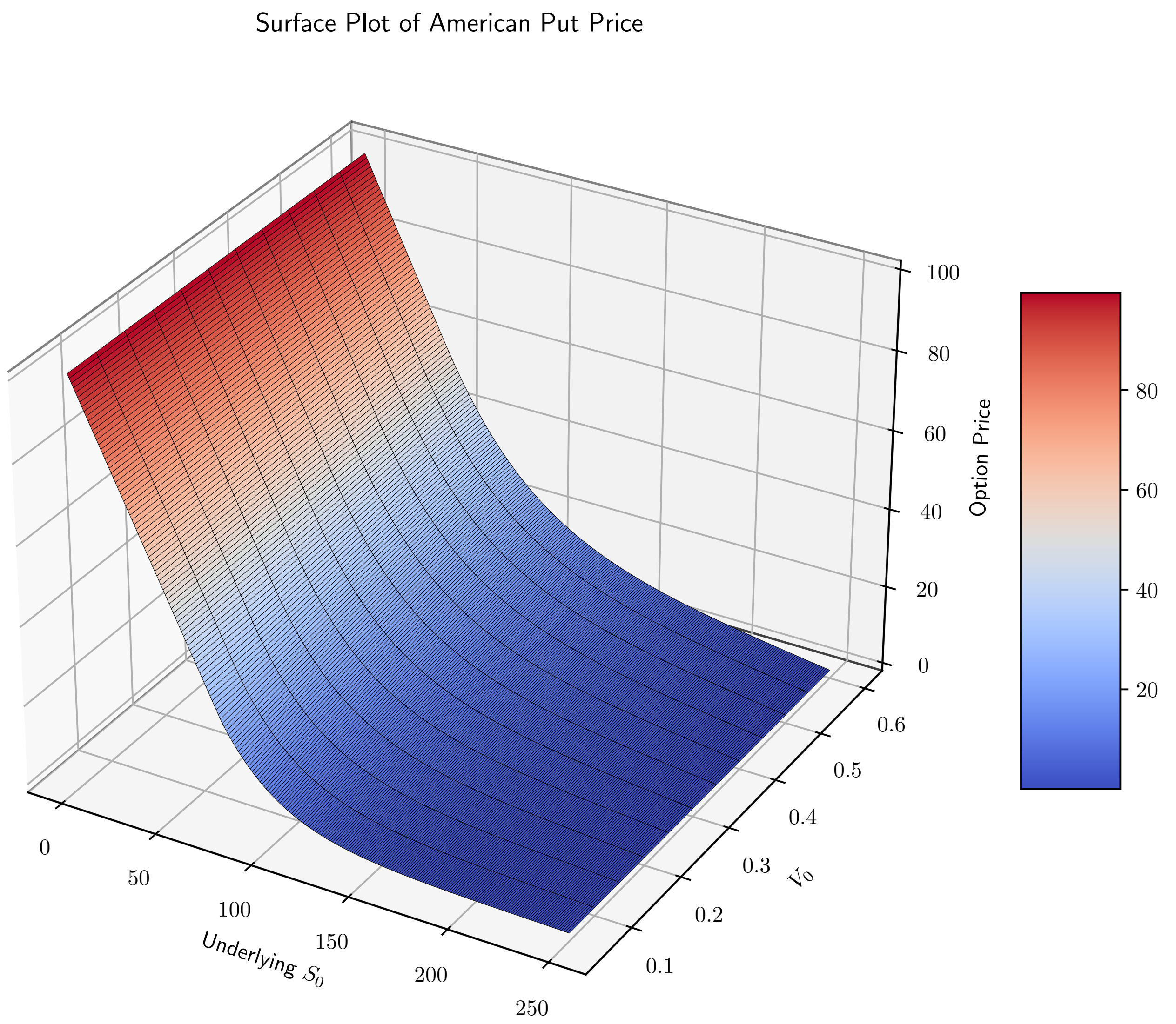
```
display(Markdown(f'We find that for the given parameters, the price of an American Option is ${round(pri
```

We find that for the given parameters, the price of an American Option is $16.121.

Plot

```
K = 100
s_max , v_max = 2.5*K,0.6
s_steps = round(s_max/ds)+1
v_steps = round((v_max-v_min)/dv)+1
vetS = ds*np.arange(s_steps)
vetV = dv*np.arange(v_steps)+v_min
s, v = np.meshgrid(vetV, vetS)
fig = plt.figure(figsize=(12,8))
plt1 = fig.add_subplot(111,projection='3d')
surf = plt1.plot_surface(v,s,prices, cmap = 'coolwarm',edgecolors='k',lw=0.1,rstride=1,cstride=1)
fig.colorbar(surf, shrink=0.5, aspect=5)
plt1.xaxis.pane.set_edgecolor('k')
plt1.yaxis.pane.set_edgecolor('k')
plt1.zaxis.pane.set_edgecolor('k')

plt.title("Surface Plot of American Put Price")
plt1.set_xlabel(r'Underlying $S_0$')
plt1.set_ylabel(r"$V_0$")
plt1.set_zlabel(r"Option Price")
plt.show()
```



Problem 1b

We compare the price of various Bermudan Options, along with the prices of European and American options for the model specified in the previous problem.

```
d_list = np.array([1,1/6,1/12,1/24,1/365,dt])
prices_list = [explicit_FD_put(dt,ds,dv,K,d)[ind_s0,ind_v0] for d in d_list]
berm_prices = pd.DataFrame(prices_list,index=d_list)
```

```python
berm_prices.index.name="Frequency"
berm_prices.columns=["Price"]
berm_prices.style.format(precision=2)
berm_prices
```

	Price
Frequency	
1.000000	15.535493
0.166667	15.535493
0.083333	15.535493
0.041667	15.535493
0.002740	15.989317
0.000010	16.121244

It appears that prices drop off somewhat between the American Option and the daily-exercise Bermudan Option, and drop off again between the daily-exercise Bermudan Option and less-frequently exercised ones. It also appears that there is effectively no difference between bi-monthly and less frequent exercise; even the European option (exercise frequency of `1`) has the same price.

Problem 1c

We examine the two dimensional GBM model:

$$dS_t^1 = (r-d)S_t^1 dt + \sigma_1 S_t^1 dW_t^1$$
$$dS_t^2 = (r-d)S_t^2 dt + \sigma_2 S_t^2 dW_t^2$$

Applying Ito's Lemma, we have:

$$df = \frac{\partial f}{\partial t}dt + \frac{\partial f}{\partial S^1}dS_t^1 + \frac{\partial f}{\partial S^2}dS_t^2 + \frac{1}{2}\frac{\partial^2 f}{\partial (S^1)^2}(dS_t^1)^2 + \frac{1}{2}\frac{\partial^2 f}{\partial (S^2)^2}(dS_t^2)^2 + +\frac{1}{2}\frac{\partial^2 f}{\partial S^1 \partial S^2}dS_t^1 dS_t^2$$

where we have $dt^2 = 0$, $dW_1 dW_2 = \rho dt$ with $\rho = 0$ and so:

$$(dS_t^1)^2 = \sigma_1^2 (S_t^1)^2 dt \qquad (dS_t^2)^2 = \sigma_2^2 (S_t^2)^2 dt \qquad dS_t^1 dS_t^2 = \rho\sigma_1\sigma_2 S_t^1 S_t^2 dt = 0$$

And so, we have:

$$df = \left(\frac{\partial f}{\partial t} + (r-d)S_t^1\frac{\partial f}{\partial S^1} + (r-d)S_t^2\frac{\partial f}{\partial S^2} + \frac{1}{2}\sigma_1^2(S_t^1)^2\frac{\partial^2 f}{\partial (S_1)^2} + \frac{1}{2}\sigma_2^2(S_t^2)^2\frac{\partial^2 f}{\partial (S_2)^2}\right)dt + \sigma_1 S_t^1 dW_t^1 + \sigma_2 S_t^2 dW_t^2$$

Taking the expectation under the risk neutral measure, we find $\mathbb{E}[df] = rf dt$. Because the martingale terms have 0 expectation, we generate the following PDE:

$$\frac{\partial f}{\partial t} + (r-d)S_t^1\frac{\partial f}{\partial S^1} + (r-d)S_t^2\frac{\partial f}{\partial S^2} + \frac{1}{2}\sigma_1^2(S_t^1)^2\frac{\partial^2 f}{\partial (S_1)^2} + \frac{1}{2}\sigma_2^2(S_t^2)^2\frac{\partial^2 f}{\partial (S_2)^2} - rf = 0$$

Similar to the previous problems, we can discretize this PDE. Define $f_{i,j,k} = f(i\Delta t, j\Delta S^1, k\Delta S^2)$ where i, j, k are integers. Our derivatives are approximated as follows:

$$\begin{aligned}
\frac{\partial f}{\partial t} &\approx \frac{f_{i,j,k} - f_{i-1,j,k}}{\Delta t} \\
\frac{\partial f}{\partial S^1} &\approx \frac{f_{i,j+1,k} - f_{i,j-1,j}}{2\Delta S^1} & \frac{\partial f}{\partial S^2} &\approx \frac{f_{i,j,k+1} - f_{i,j,k-1}}{2\Delta S^1} \\
\frac{\partial^2 f}{\partial (S^1)^2} &\approx \frac{f_{i,j+1,k} - 2f_{i,j,k} + f_{i,j-1,k}}{(\Delta S^1)^2} & \frac{\partial^2 f}{\partial (S^2)^2} &\approx \frac{f_{i,j,k+1} - 2f_{i,j,k} + f_{i,j,k-1}}{(\Delta S^2)^2}
\end{aligned}$$

Substituting these values and defining $S_j^1 = j\Delta S^1$ and $S_k^2 = k\Delta S^2$ we have:

$$\begin{aligned}
\frac{f_{i,j,k} - f_{i-1,j,k}}{\Delta t} &= -(r-d)\left(j\Delta S^1\frac{f_{i,j+1,k} - f_{i,j-1,k}}{2\Delta S^1} + k\Delta S^2\frac{f_{i,j,k+1} - f_{i,j,k-1}}{2\Delta S^2}\right) \\
&\quad - \frac{1}{2}\left(\sigma_1^2 j^2(\Delta S^1)^2\frac{f_{i,j+1,k} - 2f_{i,j,k} + f_{i,j-1,k}}{(\Delta S^1)^2} + \sigma_2^2 k^2(\Delta S^2)^2\frac{f_{i,j,k+1} - 2f_{i,j,k} + f_{i,j,k-1}}{(\Delta S^2)^2}\right) + rf_{i,j,k} \\
\frac{f_{i-1,j,k}}{\Delta t} &= f_{i,j,k}\left(\frac{1}{\Delta t} - r\right) + (r-d)\left(j\Delta S^1\frac{f_{i,j+1,k} - f_{i,j-1,k}}{2\Delta S^1} + k\Delta S^2\frac{f_{i,j,k+1} - f_{i,j,k-1}}{2\Delta S^2}\right) \\
&\quad + \frac{1}{2}\left(\sigma_1^2 j^2(\Delta S^1)^2\frac{f_{i,j+1,k} - 2f_{i,j,k} + f_{i,j-1,k}}{(\Delta S^1)^2} + \sigma_2^2 k^2(\Delta S^2)^2\frac{f_{i,j,k+1} - 2f_{i,j,k} + f_{i,j,k-1}}{(\Delta S^2)^2}\right)
\end{aligned}$$

Grouping terms, we have:

$$\begin{aligned}
f_{i-1,j,k} =& f_{i,j,k}\left(1 - r\Delta t - \sigma_1^2 j^2\Delta t - \sigma_2^2 k^2\Delta t\right) \\
&+ f_{i,j-1,k}\left(\frac{\sigma_1^2 j^2\Delta t}{2} - \frac{(r-d)j\Delta t}{2} +\right) + f_{i,j+1,k}\left(\frac{(r-d)j\Delta t}{2} + \frac{\sigma_1^2 j^2\Delta t}{2}\right) \\
&+ f_{i,j,k-1}\left(\frac{\sigma_2^2 k^2\Delta t}{2} - \frac{(r-d)k\Delta t}{2}\right) + f_{i,j,k+1}\left(\frac{(r-d)k\Delta t}{2} + \frac{\sigma_2^2 k^2\Delta t}{2}\right)
\end{aligned}$$

We are supposed to price a **2-D Max Call** with payoff $g(t, S^1, S^2) = e^{-rt}(\max(S^1, S^2) - K)_+$, with $K = 40$, $d = 0.1$. There are 25 exercise opportunities, so this is a Bermudan option with $\delta_f req = 0.04$. We also have that $S_0^1 = S_0^2 = 40$, $\sigma_1 = \sigma_2 = 0.2$, $r = 0.06$ and $T = 1$.

Boundary Conditions

At each time point, there exists a $N_{S^1} \times N_{S^2}$ grid of points. We need to solve for the values at the boundaries where $S^i = 0$, $S^i = N_{S^i}$.

- **Lower Stock Boundary** ($S_t^i = 0$): When $S^1 = 0$, $(max(S^1, S^2) - K)_+ = (S^2 - K)_+$ and $\frac{\partial^2 f}{\partial (S^1)^2} = 0$. Solving for our approximation gives us that

$$f_{i,0,k} = 2f_{i,1,k} - f_{i,2,k}$$

A similar computation for S^2 gives us:

$$f_{i,j,0} = 2f_{i,j,1} - f_{i,j,2}$$

If $S^1 = S^2 = 0$, then we can compute $f_{i,0,0}$ by substituting one of the above solutions into the other. Doing so either way yields the same result, that

$$f_{i,0,0} = 4f_{i,1,1} - 2f_{i,1,2} - 2f_{i,2,1} + f_{i,2,2}$$

- **Upper Stock Boundary** ($S_t = N_S$): As the stock price gets larger, value of the option is unlikely to change, either, as the boundary is typically set very far above the strike price. We approximate this the same way we approximate the lower boundary, namely by setting the second derivatives to 0 and solving. This yields us:

$$\begin{aligned}
f_{i,N_{S^1},k} &= 2f_{i,N_{S^1}-1,k} - f_{i,N_{S^1}-2,k} \\
f_{i,j,N_{S^2}} &= 2f_{i,j,N_{S^2}-1} - f_{i,j,N_{S^2}-2} \\
f_{i,N_{S^1},N_{S^2}} &= 4f_{i,N_{S^1}-1,N_{S^2}-1} - 2f_{i,j,N_{S^2}-1} - 2f_{i,N_{S^1}-1,k} + f_{i,N_{S^1}-2,N_{S^2}-2}
\end{aligned}$$

Code

Note: we use the same discretization for both dS^1 and dS^2 for convenience.

```
@njit
def GBM_2_d(dt,ds,freq,K):
    # Stock/gbm Parameters
    r, d = 0.06, 0.1
    s1, s2 = 0.2, 0.2
    s0 = 40
    # Option Parameters
    T = 1
    s_1_max, s_2_max = 2*K, 2*K

    # Discretization
    time_n = round(T/dt)
    asset_n_1 = round(s_1_max/ds)+1
    vetS_1 = ds*np.arange(asset_n_1)
    asset_n_2 = round(s_2_max/ds)+1
    vetS_2 = ds*np.arange(asset_n_2)

    # Default is freq = dt; this is American option
    stop_set = np.arange((round(T/freq)))*(freq/dt)

    # Construct comparison of all discretized S1 and S2 values
    comp_grid = np.array([(x,y) for x in vetS_1 for y in vetS_2])

    # Construct initial payoff function
    payoff_exit = np.maximum(np.array(
        [np.maximum(x1,x2) for (x1,x2) in comp_grid])-K,0).reshape((-1,asset_n_2))
    f = np.copy(payoff_exit)*1.0

    for i in range(time_n):
        newf = np.zeros((asset_n_1,asset_n_2))
        for j in range(1,asset_n_1-1): #S1 Loop in j (first param)
            for k in range(1,asset_n_2-1): #S2 Loop in k (second param)
                A = np.zeros((3,3))
                A[1,1] = 1 - r*dt - (s1**2)*(j**2)*dt - (s2**2)*(k**2)*dt
                A[0,1] = (s1**2)*(j**2)*dt/2 - (r-d)*j*dt/2
                A[2,1] = (r - d)*j*dt/2 +(s1**2)*(j**2)*dt/2
                A[1,0] = (s2**2)*(k**2)*dt/2 - (r-d)*k*dt/2
                A[1,2] = (r - d)*k*dt/2 +(s2**2)*(k**2)*dt/2

                newf[j,k] = (
                    A[1,1]*f[j,k] + A[0,1]*f[j-1,k] + A[2,1] * f[j+1,k] +
                    A[1,0]*f[j,k-1] + A[1,2]*f[j,k+1])
        ## Impose Boundaries
        # Lower S1 boundary
        newf[1:asset_n_1,0] = 2* newf[1:asset_n_1,1] - newf[1:asset_n_1,2]
        # Upper S1 boundary
        newf[1:asset_n_1,-1] = 2* newf[1:asset_n_1,-2] - newf[1:asset_n_1,-3]
        # Lower S0 Boundary including (0,0)
        newf[0,:] = 2*newf[1,:]-newf[2,:]
        # Upper S0 Boundary including (S1max,S2max)
        newf[-1,:] = 2*newf[-2,:] - newf[-3,:]
        if(i in stop_set):
            f = np.maximum(np.exp(-r*i*dt)*payoff_exit,newf)
        else:
            f = newf


    return f
K, dt, ds  = 40,1E-5,1
prices_GBM = GBM_2_d(dt,ds,0.04,K)
s0 = 40
ind_s0,ind_v0 = round(s0/ds),round(s0/ds)
berm_call_price = prices_GBM[ind_s0,ind_v0]
```

```python
display(Markdown(f'We find that the price of a Bermudan 2-D Max-Call is {round(berm_call_price,3)}.'))
```

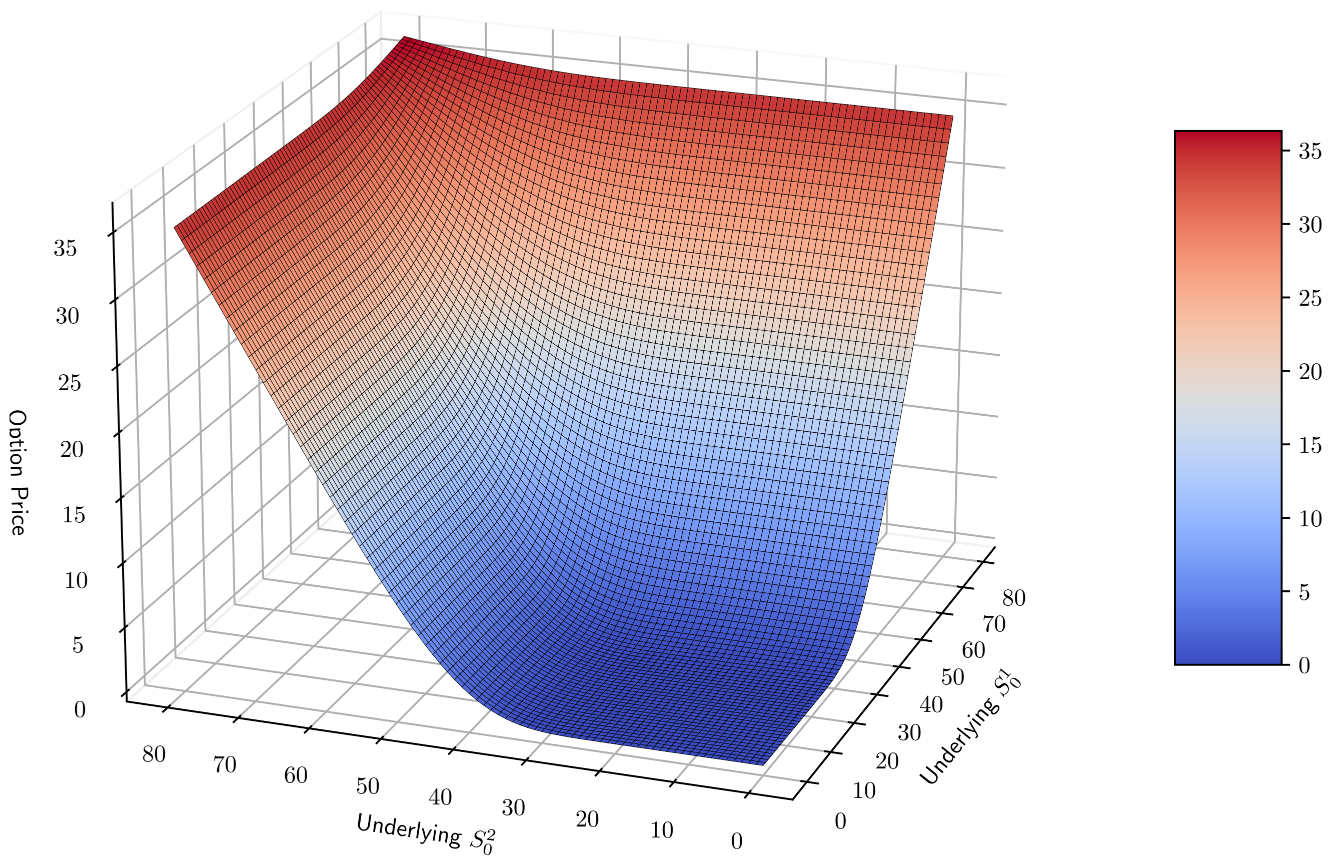
We find that the price of a Bermudan 2-D Max-Call is 4.044.

We also create a surface plot of the price of the 2-D Max Call as a function of initial prices of the underlying assets:

```python
K = 40
s_max = 2*K
s_steps = round(s_max/ds)+1
vetS = ds*np.arange(s_steps)

s1, s2 = np.meshgrid(vetS, vetS)
fig = plt.figure(figsize=(12,8))
plt1 = fig.add_subplot(111,projection='3d')
surf = plt1.plot_surface(s1,s2,prices_GBM, cmap = 'coolwarm',
    rstride=1,cstride=1,edgecolors='k',lw=0.1,)
fig.colorbar(surf, shrink=0.5, aspect=5)
plt.title(r"Surface Plot of Bermudan Max-Call with $\delta_{freq} = 0.04$")
plt1.set_xlabel(r'Underlying $S^1_0$')
plt1.set_ylabel(r"Underlying $S^2_0$")
plt1.set_zlabel(r"Option Price")
plt1.view_init(20,200)
plt1.xaxis.pane.fill=False
plt1.yaxis.pane.fill=False
plt1.zaxis.pane.fill=False
plt.show()
```

Surface Plot of Bermudan Max-Call with $\delta_{freq} = 0.04$



Problem 2

We consider the following model for returns:

$$\begin{aligned} dS_t &= S_t\sigma(R_{1,t},R_{2,t})dW_t \qquad \sigma(R_{1,t},R_{2,t}) = \beta_0+\beta_1R_1+\beta_2\sqrt{R_2} \\ dR_{1,t} &= \lambda_1(\sigma(R_{1,t},R_{2,t})dW_t - R_{1,t})dt \\ dR_{1,t} &= \lambda_1(\sigma(R_{1,t},R_{2,t}) - R_{1,t})dt \end{aligned}$$

Note that this is driven by a single Brownian Motion, so R_1 and S_t share the same random shocks.

Problem 2a

We price an option with maturity $T = 1/4$ via an Euler-Maruyama Discretization. We use a time-grid with step-sizes spaced such that two observations occur per day, i.e. $\Delta t = \frac{1}{2\times 365}$.

```
sqrt = np.sqrt
@njit
def mc_samples(M,Z,s0,T,dt):
    b0 = 0.08; b1 = -0.08; b2 = 0.5 ; K = 10
    l1 = 62; l2 = 40
    r1 = -0.044*np.ones(M); r2 = 0.007*np.ones(M)
    nsteps = int(T/dt)
    r_risk_free = 0.05
    S = np.ones((M,nsteps))*s0
    for j in range(1,nsteps):
        dW = sqrt(dt)*Z[:,j-1]
        s12 = np.maximum(b0 +r1*b1+sqrt(r2)*b2,0)
        S[:,j] = np.maximum(S[:,j-1]+S[:,j-1]*s12*dW,0)
        r1 = r1 + l1*(s12*dW - r1)*dt
        r2 = r2 + l2*( s12**2 -r2)*dt
        r2 = np.maximum(r2,0)

    return( np.maximum( np.exp(-r_risk_free*T)*(K-S[:,-1]),0))
M = 1000
T = 1/4
dt = 1/(2*365)
Z = np.random.normal(size=(M,int(1/dt)))
em_sample = mc_samples(M,Z,s0=10,T=T,dt=dt)
EM_val = np.mean(em_sample)
EM_sd = np.std(em_sample)
```

```
ci_band = 1.96*EM_sd/np.sqrt(M)
display(Markdown(rf"The option price is simply taken to be the mean of the discounted Euler-Maruyama rea
```

The option price is simply taken to be the mean of the discounted Euler-Maruyama realizations. We find this is: 0.296 ± 0.026.

Problem 2b

We will approximate Delta with a "bump-and-revalue" strategy by estimating P at $S_0 \pm \varepsilon$. Delta is approximated as:

$$\Delta \approx \frac{\hat{P}(S_0 + \varepsilon) - \hat{P}(S_0 - \varepsilon)}{2\varepsilon}$$

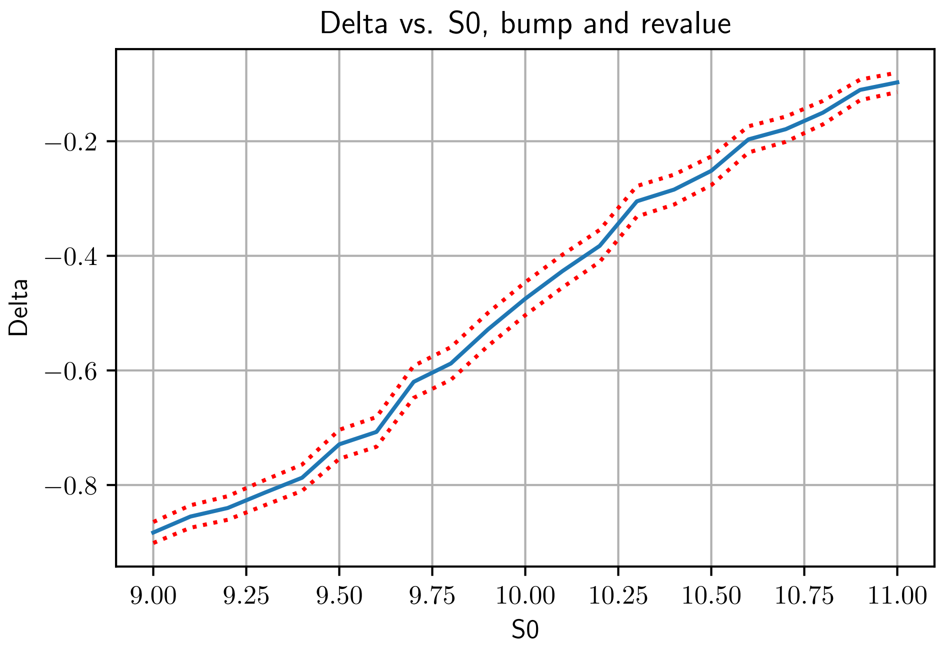
We will plot Delta as a function of S_0 along with 95% normal confidence interval bounds.

```
@njit
def bump_eval_sim(npaths,s0,K,eps,T=1/4):
    dt = 1/(2*365)
    n_steps = int(T/dt)
    dW = np.random.normal(0,1,size=(npaths,n_steps))
    p_up = mc_samples(npaths,dW,s0+eps,T,dt)
    p_down = mc_samples(npaths,dW,s0-eps,T,dt)
    prices = (p_up+p_down)/2
    deltas = (p_up-p_down)/(2*eps)
    r ={}
    r['price'] = np.mean(prices)
    r['price_std']=np.std(prices)
    r['delta'] = np.mean(deltas)
    r['delta_std']=np.std(deltas)
    return r

K = 10; npaths = 1000
s0_vals =np.arange(9,11.1,0.1)
T = 1/4
dt = 1/(2*365)
eps = 0.005
price = []
price_ci = []
delta = []
delta_ci = []
for m in range(len(s0_vals)):
    out = bump_eval_sim(npaths,s0_vals[m],K,eps)
    price.append(out['price'])
    delta.append(out['delta'])
    price_ci.append(1.96*(out['price_std']/sqrt(npaths)))
    delta_ci.append(1.96*(out['delta_std']/sqrt(npaths)))


data_all = pd.DataFrame(data = [price, price_ci,delta,delta_ci]).T
data_all.index = s0_vals
data_all.index.name = "S0"
data_all.columns = ["price","price ci","delta","delta ci"]

data_all
plt.plot(s0_vals,data_all["delta"])
plt.plot(s0_vals,data_all["delta"]-data_all["delta ci"],
    linestyle="dotted",color="red",label="95% CI")
plt.plot(s0_vals,data_all["delta"]+data_all["delta ci"],linestyle="dotted",color="red")
plt.title("Delta vs. S0, bump and revalue")
plt.xlabel("S0")
plt.ylabel("Delta")
plt.grid(True)
plt.show()
```



Problem 2c

We first fit a Gaussian Process Regression to the Option Price with a Matern-5/2 Kernel. Some noise independent of the Matern-5/2 Kernel is added:

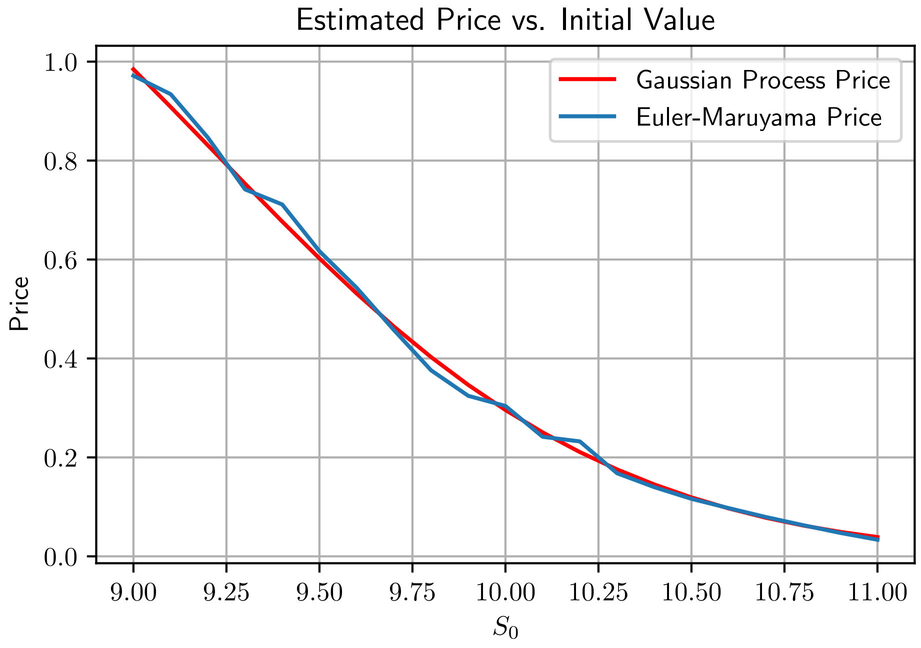
```
npaths, T , dt, K = 1000, 1/4, 1/(2*365), 100
## S0 values between 9 and 11 varying by 0.1
s0_vals = np.arange(9,11.01,0.1)
Y=[]
```

```
## Generate 1000 paths for each S0 value
sd_Y = []
for s in s0_vals:
    Z = np.random.normal(size=(npaths,int(1/dt)))
    sample_out = mc_samples(npaths,Z,s,T,dt)
    Y.append(np.mean(sample_out)*1.0)
    sd_Y.append(np.std(sample_out)*1.0)

kernel = gp.kernels.Matern(nu=2.5,length_scale_bounds=[1E-5,10000])
model = gp.GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=10,alpha=1e-3)
#Fit Model
model.fit(s0_vals.reshape(-1,1),Y)
# Predict on S0 and get standard deviations
fit_out,sigma = model.predict(s0_vals.reshape(-1,1),return_std=True)
data_all["price gp"] = fit_out
```

We plot the Options Prices from the Gaussian Process Regression against the Euler-Maruyama estimates:

```
plt.plot(s0_vals,data_all["price gp"],color="red",label="Gaussian Process Price")
plt.plot(s0_vals,data_all['price'],label="Euler-Maruyama Price")
# plt.plot(s0_vals,data_all["price"])
plt.xlabel(r"$S_0$")
plt.ylabel("Price")
plt.grid(True)
plt.title("Estimated Price vs. Initial Value")
plt.legend()
plt.show()
```



We now wish to compute Delta from this regrssion. Let m_x be the posterior value at point x. The output for a gaussian process regression with N training points and kernel function $\kappa(\cdot,\cdot)$ is given by:

$$m_x = \mathbb{E}\{m|X,\mathbf{y}\} = \sum_{i=1}^{N} \alpha_i \kappa(x_i, x)$$

where $\alpha = (\mathbf{K} + \sigma_n^2 \mathbf{I})^{-1}\mathbf{y}$, $K_{i,j} = \kappa(x_i, x_j)$, and σ_n^2 is the uncertainty of each observation. Differentiating this expression gives us:

$$\nabla \mathbb{E}\{m_x|X,\mathbf{y}\} = \sum_{i=1}^{n} \alpha_i \frac{\partial \kappa_{M_{5,2}}(x,x')}{\partial x_i}$$

The derivative of the Kernel Function is from Ludkovski and Saporito (2022):

$$\frac{\partial \kappa_{M_{5,2}}}{\partial x_k}(x,x') = \left(\frac{-\frac{5}{3\ell^2}(x_k - x_k^{'}) - \frac{5^{3/2}}{3\ell^3}(x_k - x_k')|x_k - x_k'|}{1 + \frac{\sqrt{5}}{\ell}|x_k - x_k^{'}| + \frac{5}{3\ell^2}(x_k - x_k^{'})^2} \right) \kappa_{M_{5,2}}(x,x')$$

where $\kappa_{M_{5,2}}(x,x')$ is the Matern-5/2 kernel.

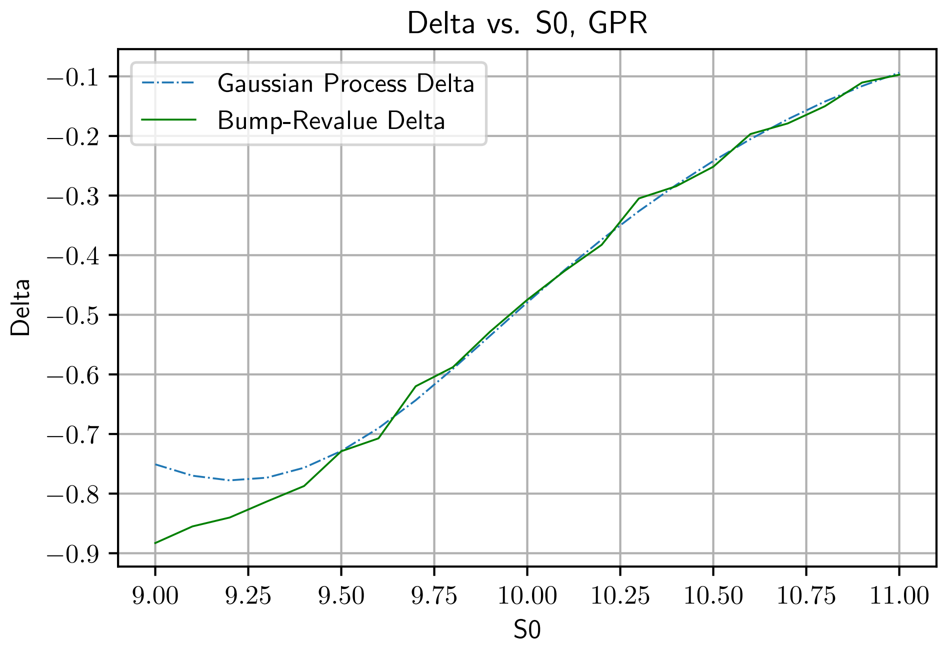
```
# Formula from Paper
def matern_deriv_mult(x,x_train,l):
    num = -5/(3*l**2)*(x-x_train)-(5**(1.5))/(3*l**3)*(x-x_train)*np.abs(x-x_train)
    denom = 1 + sqrt(5)/l*np.abs(x-x_train)+5/(3*l**2)*(x-x_train)**2
    return (num/denom).ravel()

ell = model.kernel_.get_params()['length_scale']
# Initialize 0 vector
delta_gp = np.zeros(len(s0_vals))*0.0

# x_a is point at which deriv is taken- in this case, we sweep across s0 values
for key,x in enumerate(s0_vals):
    kern_val = model.kernel_(x.reshape(-1,1),s0_vals.reshape(-1,1),eval_gradient=False).ravel()
    delta_gp[key] = sum(matern_deriv_mult(x,s0_vals,ell)* kern_val * model.alpha_)
```

We plot the Gaussian Process Delta against Bump-Revalue Delta:

```
plt.plot(s0_vals,delta_gp,label="Gaussian Process Delta",linewidth = 0.7,linestyle='dashdot')
plt.plot(s0_vals,data_all["delta"],color="green",label="Bump-Revalue Delta",linewidth = 0.7)
plt.title("Delta vs. S0, GPR")
plt.xlabel("S0")
plt.ylabel("Delta")
plt.grid(True)
plt.legend()
plt.show()
```



An interesting phenomenon is that at the Gaussian Process tends to diverge from the stochastically estimated values at the ends of our band of values checked. Nevertheless, away from the endpoints within the observed range, the Gaussian Process regression matches the bump-and-revalue strategy surprisingly well, and produces a much smoother output.

Problem 2d

We implement a Neural Network with one hidden layer to price the previously described option. We generate data from our earlier Monte Carlo simulation, and use it to a neural nework with each of the `ReLU`, `Tanh` and `ELU` activation functions. Our networks implement the ADAM Optimizer on our Mean-Square Error (MSE) loss function as it performed better than the classical Stochastic Gradient Descent.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

class NN(nn.Module):
    def __init__(self, activation):
        super().__init__()
        self.layer1 = nn.Linear(1, 30)
        self.layer2 = nn.Linear(30, 1)
        self.activation = activation
    def forward(self, x):
        x = self.activation(self.layer1(x))
        x = self.layer2(x)
        return x

@njit
def make_data(n):
    T,dt,M = 1/4,1/(2*365), 1000
    s0_vals = (np.random.rand(n)*2+9)*1.0
    price = np.zeros(n)

    for i in range(n):
        Z = np.random.normal(0,1,size=(npaths,round(T/dt)))
        price[i] = np.mean(mc_samples(M,Z,s0_vals[i],T,dt))*1.0

    return(s0_vals,price)
```

```
device = (
    "cuda"
    if torch.cuda.is_available()
    else "mps"
    if torch.backends.mps.is_available()
    else "cpu"
)
```

```
def train(net,x,y,epoch_num):
    mse = nn.MSELoss()
    optimizer = optim.Adam(net.parameters(),lr = 0.0001)
    print("Starting Training")
    for epoch in range(epoch_num):
        optimizer.zero_grad()
        predictions = net.forward(x)
        loss = mse(predictions,y)
        loss.backward()
        optimizer.step()
        if(epoch + 1 )%(round(epoch_num/4)) == 0:
            print(f'Epoch [{epoch +1}/{epoch_num}], Loss: {loss.item()}')

N = 10000
epochs=50000
```

```
display(Markdown(f'We generate {N} data paris and train over {epochs} epochs.'))
```

We generate 10000 data paris and train over 50000 epochs.

```
np.random.seed(1234567)
(x,y) = make_data(N)
x = torch.tensor(x, dtype = torch.float32).unsqueeze(1).requires_grad_().to(device)
y = torch.tensor(y, dtype = torch.float32).unsqueeze(1).requires_grad_().to(device)
```

```
torch.manual_seed(2)
net_relu = NN(nn.ReLU()).to(device)
train(net_relu,x,y,epochs )
# torch.save(net_relu.state_dict(),'relu_net')
```

```
Starting Training
Epoch [12500/50000], Loss: 0.0010729379719123244
Epoch [25000/50000], Loss: 0.00023725113715045154
Epoch [37500/50000], Loss: 0.00023327417147811502
Epoch [50000/50000], Loss: 0.000233182538067922
```

Using `tanh` instead of `relu`:

```
net_tanh = NN(nn.Tanh()).to(device)
train(net_tanh,x,y,epochs)
```

```
# torch.save(net_tanh.state_dict(),'tanh_net')
```

```
Starting Training
Epoch [12500/50000], Loss: 0.0023764765355736017
Epoch [25000/50000], Loss: 0.00035378802567720413
Epoch [37500/50000], Loss: 0.0002515829401090741
Epoch [50000/50000], Loss: 0.00023985000734683126
```

Using ELU

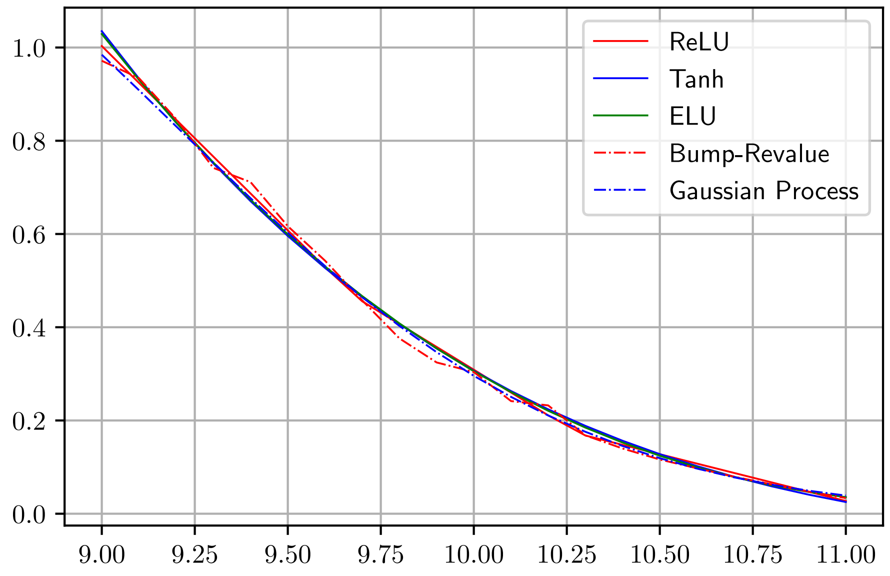
```
net_elu = NN(nn.ELU()).to(device)
train(net_elu,x,y,epochs)
# torch.save(net_elu.state_dict(),'elu_net')
```

```
Starting Training
Epoch [12500/50000], Loss: 0.004471060819923878
Epoch [25000/50000], Loss: 0.0017179179703816772
Epoch [37500/50000], Loss: 0.00027756314375437796
Epoch [50000/50000], Loss: 0.00020805129315704107
```

```
# Move models back onto cpu
if(device!='cpu'):
    device = torch.device('cpu')
    net_relu.to(device)
    net_tanh.to(device)
    net_elu.to(device)
    x.to(device)
    y.to(device)




s0 = torch.arange(9,11.1,0.1).to(device)
plt.plot(s0,net_relu(s0.unsqueeze(1)).detach(),color='red',label="ReLU",linewidth = 0.7)
plt.plot(s0,net_tanh(s0.unsqueeze(1)).detach(),color='blue',label="Tanh",linewidth = 0.7)
plt.plot(s0,net_elu(s0.unsqueeze(1)).detach(),color='green',label="ELU",linewidth = 0.7)
plt.plot(s0,data_all['price'],label="Bump-Revalue",color="red",linestyle="dashdot",linewidth = 0.7)
plt.plot(s0,data_all['price gp'],label="Gaussian Process",color="blue",linestyle="dashdot",linewidth = 
plt.legend()
plt.grid(True)
```



For prices, it appears that the both `Tanh` and `ELU` perform better than `ReLU`, which, due to its piecewise linearity, fails to capture a lot of the necessary complexity. It is difficult to tell which is superior between `Tanh` and `ELU`.

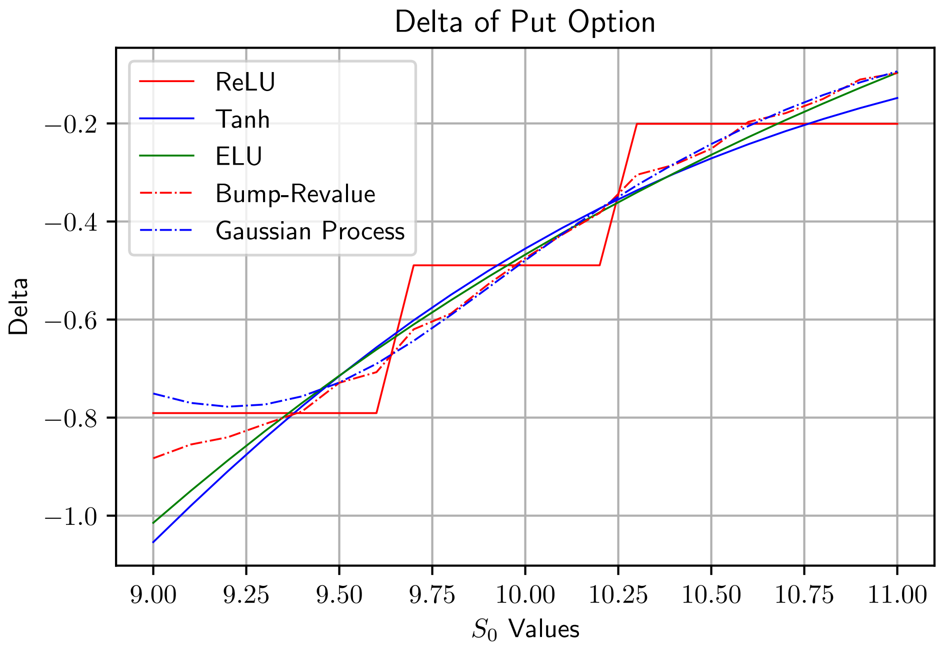
Compute Delta

We can compute the Delta of the put option by evaulating the gradients of our neural networks directly.

```
x = s0.unsqueeze(1)
x.requires_grad=True
price_ReLU = net_relu(x)
price_Tanh = net_tanh(x)
price_ELU = net_elu(x)

grad_relu = torch.autograd.grad(sum(price_ReLU),x,create_graph=True)[0]
grad_tanh = torch.autograd.grad(sum(price_Tanh),x,create_graph=True)[0]
grad_elu = torch.autograd.grad(sum(price_ELU),x,create_graph=True)[0]

plt.plot(s0,grad_relu.detach(),label="ReLU",color='red',linewidth = 0.7)
plt.plot(s0,grad_tanh.detach(),label="Tanh",color='blue',linewidth = 0.7)
plt.plot(s0,grad_elu.detach(),label="ELU",color='green',linewidth = 0.7)
plt.plot(s0,data_all['delta'],label="Bump-Revalue",color="red",linestyle="dashdot",linewidth = 0.7)
plt.plot(s0,delta_gp,label="Gaussian Process",color="blue",linestyle="dashdot",linewidth = 0.7)
plt.xlabel(r"$S_0$ Values")
plt.ylabel("Delta")
plt.title("Delta of Put Option")
plt.legend()
plt.grid(True)
plt.show()
```



The failure of `ReLU` is immediately apparent here, as Delta is very clearly not piecewise-constant (the apparentl linearity is just an artifact of `Python`'s plotting interpolation). `Tanh` appears unable to capture some of the complexity of Delta for the lower regions of our training area, which makes `ELU` seem somewhat superior. in this situation.

One other issue with `ReLU` is that I had a great deal of trouble capturing anything other than a purely linear function and constant delta with this model; it took generating a very large number of samples in order to capture any nonlinearity.

Lam, Siu Kwan, Antoine Pitrou, and Stanley Seibert. 2015. "Numba: A Llvm-Based Python Jit Compiler." In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 1–6.

Ludkovski, Mike, and Yuri Saporito. 2022. "KrigHedge: Gaussian Process Surrogates for Delta Hedging." https://arxiv.org/abs/2010.08407.