PSTAT 222C Homework 3

Alex Bernstein

```python
import matplotlib.pyplot as plt
import numpy as np
from tqdm import tqdm,trange
from numba import jit
import copy
import pdb
import time
```

Problem 1: ...

We have the Heston Stochastic Volatility Model with:

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

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 from t to $T - t$ (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:

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

- Second Order Standard Derivatives:

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

- Second Order Mixed Derivatives:

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

Letting $S_k = k\Delta s$ and $V_j = j\Delta v$, Our numeric simulation therefore solves:

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

Letting $S_k = k\Delta s$ and $V_j = j\Delta v$, we find:

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

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

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

Problem 2: Two-Factor Markovian Path Dependent Volatility Model

```
#@jit(nopython=True)
def MC_put(M, S0 = 10.0):
    """
    Prices an European put option using Monte Carlo simulations.
    The follows a model where the volatility is a function of the past returns.
    """
    # Option parameters
    T, K = 1/4, 10
    # Model parameters
    beta0, beta1, beta2 = 0.08, -0.08, 0.5
    lambda1, lambda2 = 62, 40
    r = 0.05
    # Initial values
    R1, R2 = np.repeat(-0.044, M), np.repeat(0.007, M)
    dt = 1/(365*2)
```

```
    timesteps = round(T/dt)  # Time discretization
    # Initialize S and Z
    S = np.zeros((M,timesteps+1))
    print(S.shape)
    np.random.seed(1)
    Z = np.random.randn(M, timesteps)
    S[:,0] = S0
    # Monte Carlo
    for t in range(1,timesteps+1):
      # Compute sigma
      sigma = beta0 + beta1*R1 + beta2*(R2**(1/2))
      sigma = np.maximum(sigma, 0)  # Ensure positivity of sigma
      # Euler-Maruyama scheme
      S[:,t] = S[:,t-1] + S[:,t-1]*sigma*np.sqrt(dt)*Z[:,t-1]
      R1 = R1 + lambda1*(sigma*np.sqrt(dt)*Z[:,t-1] - R1)*dt
      R2 = R2 + lambda2*((sigma**2)*np.sqrt(dt)*Z[:,t-1] - R2)*dt
      S[:,t] = np.maximum(S[:,t], 0) # Ensure positivity of S
      R2 = np.maximum(R2, 0)         # Ensure positivity of R2
    # Discounted payoff
    return np.maximum(np.exp(-r*T)*(K-S[:,-1]), 0)
M = 1000
# Put option price at S0 = 10
np.mean(MC_put(M))
```

```
(1000, 183)
```

```
0.17263743814911536
```

```
sqrt = np.sqrt
#@jit(nopython=True)
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)
```

```python
    return( np.maximum( np.exp(-r_risk_free*T)*(K-S[:,-1]),0))
M = 25000
T = 1/4
dt = 1/(2*365)
Z = np.random.normal(size=(M,int(1/dt)))
print(np.mean(mc_samples(M,Z,s0=10,T=T,dt=dt)))
# print(np.mean(MC_put(M)))
```

```
0.2969982954635496
```

```python
#@jit(nopython=True)
def bump_eval_sim(s0,K,eps,T=1/4):
    npaths = 1000
    dt = T/(2*365)
    n_steps = int(1/dt)
    s0_check = [s0-eps,s0,s0+eps]
    prices = np.zeros(len(s0_check))
    dW = np.random.normal(0,1,size=(npaths,n_steps))
    for j in range(len(s0_check)):
        S0 = s0_check[j]
        prices[j] = np.mean(mc_samples(npaths,dW,S0,T,dt))
    return prices

K = 10
s0_vals =np.arange(9,11.1,0.1)
T = 1/4
dt = T/(2*365)
prices = []
prices_2 = []
for m in trange(len(s0_vals)):
    prices.append(bump_eval_sim(s0_vals[m],K,0.005))

print(prices)
```

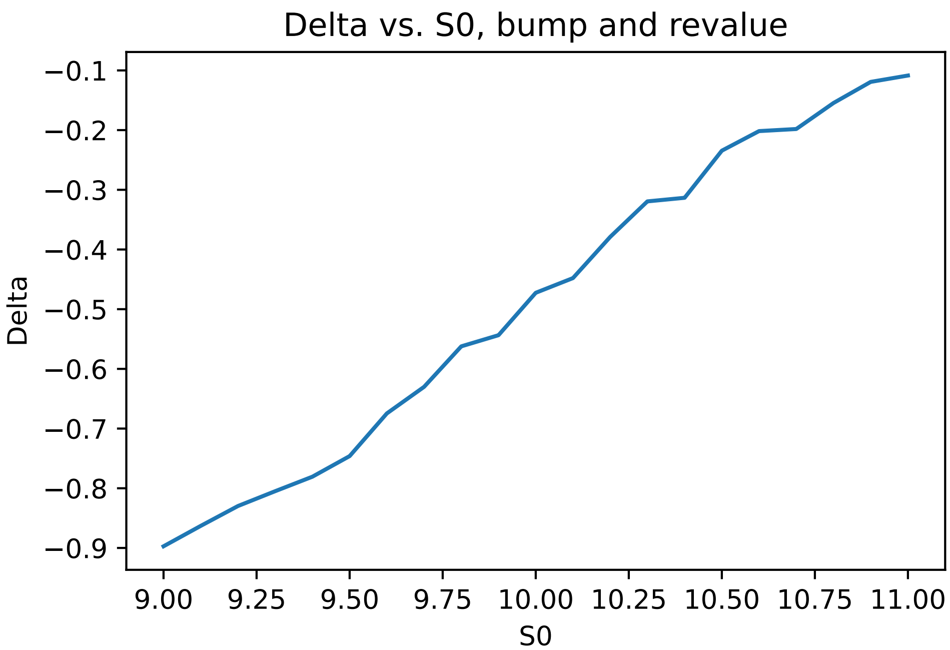
```
  0%|          | 0/21 [00:00<?, ?it/s]  5%|          | 1/21 [00:00<00:02,  9.82it/s] 10%|          | 2/21 [
```

```
[array([1.06991009, 1.0654212 , 1.06093799]), array([0.96164059, 0.95732259, 0.95301071]), array([0.8143568
```

```python
import pandas as pd



bump_sim_price = pd.DataFrame(prices)
bump_sim_price.index = s0_vals
bump_sim_price.columns = ["S0-eps","S0","S0+eps"]
```

```python
bump_sim_price["delta"] = (bump_sim_price["S0+eps"]-bump_sim_price["S0-eps"])/0.01

bump_sim_price
plt.plot(s0_vals,bump_sim_price["delta"])
plt.title("Delta vs. S0, bump and revalue")
plt.xlabel("S0")
plt.ylabel("Delta")
# df = pd.concat([bump_sim_price_old,bump_sim_price],axis=1)
# df
```

```
Text(0, 0.5, 'Delta')
```



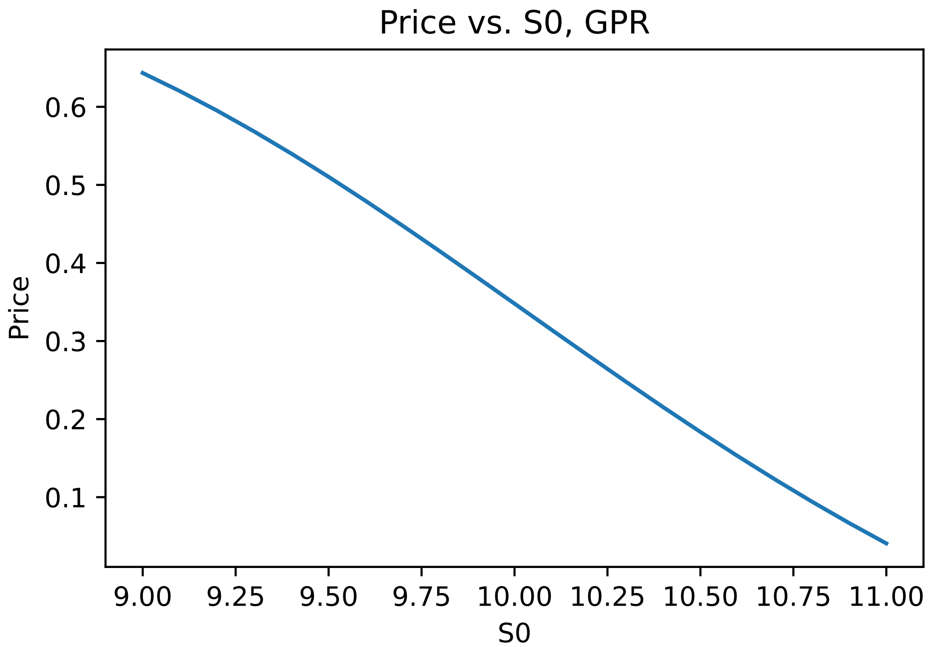
Let m_x be the posterior value at point x. The output for a gaussian process regression with N training points and kernel function $\kappa(\cdot,\cdot)$ is given by:

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

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 with respect to S_0 is straightforward:

$$\nabla_{S_0}\mathbb{E}\{m_x|X,\mathbf{y}\} = \sum_{i=1}^{n}\alpha_i\nabla_{S_0}\kappa(x_i,x)$$

```
import sklearn.gaussian_process as gp
from sklearn.gaussian_process.kernels import Matern
npaths = 1000
T = 1/4
dt = T/(2*365)
K = 100
## S0 values between 9 and 11 varying by 0.1
s0_vals = np.arange(9,11.1,0.1)
s0_vals = s0_vals.reshape(-1,1)
Y=[]
## Generate 1000 paths for each S0 value
M = 1000
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= np.append(Y,np.mean(sample_out))
    sd_Y=np.append(sd_Y,np.std(sample_out))
# Y.reshape(-1,1)
obs_var=np.diag(np.maximum(1e-4,sd_Y))
kernel = gp.kernels.Matern(length_scale=1,nu=2.5,length_scale_bounds=[10e-5,1000])
model = gp.GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=50,alpha=np.diag(obs_var), norma

#Fit Model
model.fit(s0_vals,Y)

# Predict on S0 and get standard deviations
fit_out,sigma = model.predict(s0_vals.reshape(-1,1),return_std=True)
plt.plot(s0_vals,fit_out)
plt.title("Price vs. S0, GPR")
plt.xlabel("S0")
plt.ylabel("Price")
plt.show()
```




```
out = 0
l = model.kernel_.get_params()['length_scale']
s_test = s0_vals[4].reshape(-1,1)
for key,a in enumerate(model.alpha_):
    kern = model.kernel_(s_test,s0_vals[key].reshape(-1,1),eval_gradient=False).ravel()
    out = out + a*kern
print(out)
print(model.predict(s_test))
```

```
[0.54005501]
[0.54005501]
```

The derivative of the Kernel Function is from the cited paper:

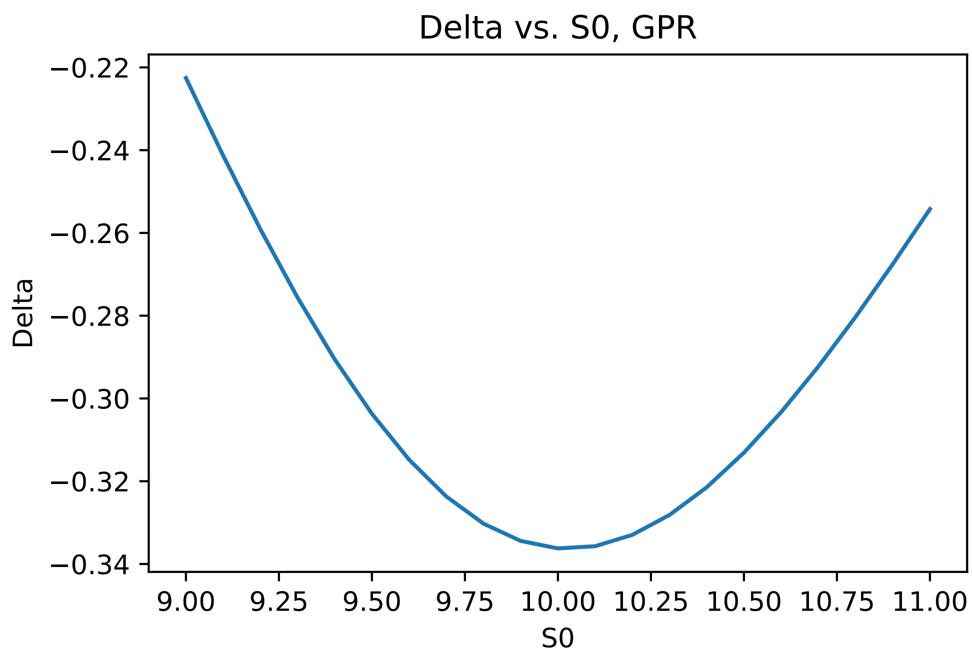
$$\frac{\partial \kappa_{M_{5,2}}}{\partial x_k}(x, x_k) = \left(\frac{-\frac{5}{3\ell^2}(x_k - x_k^{'}) - \frac{5^{3/2}}{3\ell^3}(x_k - x_k')|x_k - x_k'|}{1 + \frac{\sqrt{5}}{\ell}|x_k - x_k^{'}| + \frac{5}{3\ell^2}(x_k - x_k^{'})^2} \right)$$

```
# 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)
```

```python
    denom = 1 + sqrt(5)/l*np.abs(x-x_train)+5/(3*l**2)*(x-x_train)**2
    return (num/denom).ravel()


def matern_deriv_var(y,l):
    l = model.kernel_.get_params()['length_scale']
    covmat = model.kernel_(s0_vals,s0_vals) + obs_var
    mult_term = np.linalg.inv(covmat)
    d52 = matern_deriv_mult(x,y,l)
    return -5/(3*l**2)- (d52 @ mult_term) @ d52
```

```python
l = model.kernel_.get_params()['length_scale']

# Initialize 0 vector
grad_vals =0.0*fit_out

var_vals = 0.0*fit_out

# 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,s0_vals.reshape(-1,1),eval_gradient=False).ravel() # Vector of kernel Val
    kern_coef = matern_deriv_mult(x,s0_vals,l).ravel() #Vector ofderivative coefficents
    d_52_k = kern_coef * kern_val #multiplied elementwise from above
    grad_vals[key] = d_52_k @ model.alpha_ #inner product with regression coefficients

plt.plot(s0_vals,grad_vals)
plt.title("Delta vs. S0, GPR")
plt.xlabel("S0")
plt.ylabel("Delta")
```

```
Text(0, 0.5, 'Delta')
```



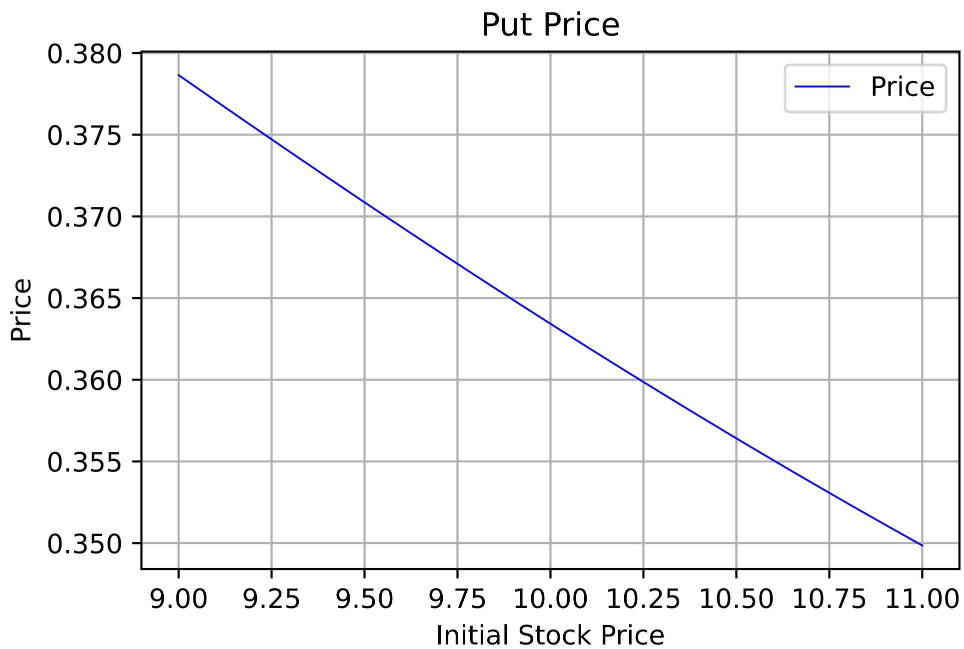
Part D

```
import torch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
torch.device('cuda')
T, dt, M = 1/4, 1/(2*360), 1000
sims = 1000
epsilon = 0.005
input = np.linspace(9,11,sims)*1.0
output = np.zeros(sims)
for i in range(sims):
  np.random.seed(1)
  Z = np.random.randn(M, round(T/dt))
  output[i] = np.mean(mc_samples(M,Z,input[i],T,dt))*1.0#(MC_put(M, Z, input[i]))
input = torch.tensor(input,dtype=torch.float32)
output = torch.tensor(output,dtype=torch.float32)
input = input.unsqueeze(1)
output = output.unsqueeze(1)
input.requires_grad = True
```

```python
output.requires_grad = True
# Implement simple feedforward NN
class NN(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Linear(1, 30)
        self.layer2 = nn.Linear(30, 1)
        self.activation = nn.Tanh()
    def forward(self, x):
        x = self.activation(self.layer1(x))
        x = self.layer2(x)
        return x
# Instantiate the neural network
#net = NeuralNetwork('ReLU')
net = NN()
# Define loss function
criterion = nn.MSELoss()
# Choose optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)
# Train the neural network
num_epochs = 1000
for epoch in range(num_epochs):
    optimizer.zero_grad() # Zeros the gradients
    # pdb.set_trace()
    predictions = net.forward(input) # Forward pass
    # Compute the loss
    loss = criterion(predictions, output)
    loss.backward()   # Backward pass
    optimizer.step()  # Update the weights
    # Print the loss every 10 epochs
    if (epoch + 1) % 100 == 0:
        print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item()}')
```

```
Epoch [100/1000], Loss: 0.08172357082366943
Epoch [200/1000], Loss: 0.07987432926893234
Epoch [300/1000], Loss: 0.07875601947307587
Epoch [400/1000], Loss: 0.07798078656196594
Epoch [500/1000], Loss: 0.07735644280910492
Epoch [600/1000], Loss: 0.07677006721496582
Epoch [700/1000], Loss: 0.07612478733062744
Epoch [800/1000], Loss: 0.07530706375837326
Epoch [900/1000], Loss: 0.0742117315530777
Epoch [1000/1000], Loss: 0.0729222223162651
```

```python
# Create the line plot
S0 = torch.arange(9,11.01,0.1)
plt.plot(S0, net(S0.unsqueeze(1)).detach(), label = 'Price', linewidth=0.7, color='blue')
# Adding labels and title
plt.xlabel('Initial Stock Price')
plt.ylabel('Price')
plt.title('Put Price')
plt.legend()
# Display the plot
plt.grid(True)
plt.show()
```




```python
x = S0.unsqueeze(1)
x.requires_grad = True
y = net(x)
gradx_of_y = torch.autograd.grad(sum(y), x,create_graph = True)[0]
plt.plot(S0, gradx_of_y.detach(), label = 'Delta', linewidth=0.7, color='blue')
# Adding labels and title
plt.xlabel('Initial Stock Price')
plt.ylabel('Delta')
plt.title('Delta')
```

```
plt.legend()
# Display the plot
plt.grid(True)
plt.show()
```