Krupa Homework 4

May 3, 2020

[2]:
```
import pandas as pd
import numpy as np
from numpy.random import Generator, SFC64
import math
import scipy.stats as sp
import time
from scipy.linalg import cholesky
import plotly
from plotly import graph_objs as go
plotly.offline.init_notebook_mode(connected = True)
rg = Generator(SFC64())
```

1 Problem 1: Considering Different Monte Carlo Schemes

[3]:
```
r = 0.06
q = 0.03
sig = 0.2
S = 100
K = 100
T = 1
def MC(S,K,r,sig,T,n,m,q=0,opt_type='c',print_df=False):
    # start time
    if print_df:
        start = time.time()
    # define dt and discount rate
    dt = T/n
    disc = np.exp(-r*T)
    c=1 if opt_type=='c' else -1
    # generate m arrays of random intervals of BM of length n with mean 0␣
↪variance dt
    dWs = np.array(np.split(rg.normal(0,np.sqrt(dt),n*m),m))
    # create arrays of stock paths begnning arrays of the differentials
    S_arr = np.cumsum((r-q - .5*sig**2)*dt+sig*dWs,axis=1)
    # add the log of the stock price then input into exponential function for␣
↪array of stock values
```

```python
    S_arr += np.log(S)
    S_arr = np.exp(S_arr)
    # array of m stock values at maturity T
    S_T = S_arr[:,n-1]
    opt_values = disc*np.maximum(c*(S_T - K),0)
    if print_df:
        run_time = time.time() - start
        return pd.DataFrame({'Option Type': opt_type, 'Sims (m)': m, 'Steps␣
↪(n)': n,
                            'Option Value': opt_values.mean(), 'SE':␣
↪opt_values.std()/np.sqrt(m),
                            'Time': run_time}, index = [1])
    else:
        return opt_values.mean()
```

[3]:
```python
ns = [300,600,300,600]
ms = [1000000,1000000,2000000,2000000]
sum_table = pd.DataFrame(columns = ['Option Type','Sims (m)','Steps␣
↪(n)','Option Value','SE','Time'])
# for i in range(0,4):
#     temp = MC(S,K,r,sig,T,ns[i],ms[i],q,'c',True)
#     sum_table = sum_table.append(temp, ignore_index=True)

# sum_table
```

[4]:
```python
temp = MC(S,K,r,sig,T,ns[0],ms[0],q,'c',True)
sum_table = sum_table.append(temp, ignore_index=True)
```

[5]:
```python
temp = MC(S,K,r,sig,T,ns[1],ms[1],q,'c',True)
sum_table = sum_table.append(temp, ignore_index=True)
```

[6]:
```python
temp = MC(S,K,r,sig,T,ns[2],ms[2],q,'c',True)
sum_table = sum_table.append(temp, ignore_index=True)
sum_table
```

[6]:

	Option Type	Sims (m)	Steps (n)	Option Value	SE	Time
0	c	1000000	300	9.133143	0.013683	18.037584
1	c	1000000	600	9.133740	0.013720	54.236782
2	c	2000000	300	9.134332	0.009688	45.858708

[ ]:
```python
# This last simulation crashed the kernel and forced it to restart so I've left␣
↪it out

# temp = MC(S,K,r,sig,T,ns[3],ms[3],q,'c',True)
# sum_table = sum_table.append(temp, ignore_index=True)
# sum_table
```

[7]:
```python
def␣
↪MC_antithetic(S,K,r,sig,T,n,m,q=0,opt_type='c',print_df=False,antithetic=True):
↪
```

```
    # start time
    if print_df:
        start = time.time()
    # define dt and discount rate
    dt = T/n
    disc = np.exp(-r*T)
    c=1 if opt_type=='c' else -1
    # generate m arrays of random intervals of BM of length n with mean 0␣
↪variance dt
    if antithetic:
        m_half = int(m/2)
        dWs = np.array(np.split(rg.normal(0,np.sqrt(dt),n*m_half),m_half))
        dWs = np.concatenate((dWs,-dWs),axis=0)
    else:
        dWs = np.array(np.split(rg.normal(0,np.sqrt(dt),n*m),m))
    # create arrays of stock paths begnning arrays of the differentials
    S_arr = np.cumsum((r-q - .5*sig**2)*dt+sig*dWs,axis=1)
    # add the log of the stock price then input into exponential function for␣
↪array of stock values
    S_arr += np.log(S)
    S_arr = np.exp(S_arr)
    # array of m stock values at maturity T
    S_T = S_arr[:,n-1]
    opt_values = disc*np.maximum(c*(S_T - K),0)
    if print_df:
        run_time = time.time() - start
        return pd.DataFrame({'Option Type': opt_type, 'Sims (m)': m, 'Steps␣
↪(n)': n,
                             'Option Value': opt_values.mean(), 'SE':␣
↪opt_values.std()/np.sqrt(m), 'Time': run_time},
                           index = [1])
    else:
        return opt_values.mean()
```

[8]:
```
N = sp.norm.pdf
def␣
↪MC_DC(S,K,r,sig,T,n,m,q=0,opt_type='c',print_df=False,antithetic=True,CV=True):
↪
    if CV == False:
        return MC_antithetic(S,K,r,sig,T,n,m,q,opt_type,print_df,antithetic)
    if print_df:
        start = time.time()
    c=1 if opt_type=='c' else -1
    dt = T/n
    nu = (r-q-.5*sig**2)*dt
    disc = np.exp((r-q)*dt)
```

```
    beta = -1

    S0 = S*np.ones(m)
    St = S0
    cv = np.zeros(m)

    # generate random numbers
    if antithetic:
        m_half = int(m/2)
        dWs = np.array(np.split(rg.normal(0,np.sqrt(dt),n*m_half),m_half))
        dWs = np.concatenate((dWs,-dWs),axis=0)
    else:
        dWs = np.array(np.split(rg.normal(0,np.sqrt(dt),n*m),m))

    # generate array of prices and delta for each step and add to array cv of␣
↪sums for each simulation
    for i in range(n):
        tt = i*dt
        d1 = (np.log(St/K)+(r-q+.5*sig**2)*(T-tt))/(sig*np.sqrt(T-tt))
        if c == 1:
            delta = N(d1)*np.exp(-q*(T-tt))
        else:
            delta = N(-d1)*np.exp(-q*(T-tt))
        rand = dWs[:,i]
        Stn = St*np.exp(nu+sig*rand)
        cv = cv + delta*(Stn-St*disc)
        St = Stn

    opt_values = np.maximum(0, c*(St-K)) + beta*cv
    opt_values = np.exp(-r*T)*opt_values
    if print_df:
        run_time = time.time() - start
        return pd.DataFrame({'Option Type': opt_type, 'Sims (m)': m, 'Steps␣
↪(n)': n,
                             'Option Value': opt_values.mean(), 'SE':␣
↪opt_values.std()/np.sqrt(m), 'Time': run_time},
                            index = [1])
    else:
        return opt_values.mean()
```

[9]:
```
n = 600
m = 1000000
AVs = [False,True,False,True]
DCs = [False,False,True,True]
sum_table2 = pd.DataFrame(columns = ['Option Type','Sims (m)','Steps␣
↪(n)','Option Value','SE','Time'])
```

```
for i in range(4):
    temp = MC_DC(S,K,r,sig,T,n,m,q,'c',True,AVs[i],DCs[i])
    sum_table2 = sum_table2.append(temp,ignore_index=True)

inds = ['MC','Antithetic','Delta Control','AV and DC']
sum_table2.rename(index = lambda x: inds[x], inplace = True)
sum_table2
```

[9]:

```
                Option Type Sims (m) Steps (n)  Option Value        SE  \
MC                        c  1000000       600      9.123243  0.013688
Antithetic                c  1000000       600      9.143607  0.013688
Delta Control             c  1000000       600      9.156537  0.010480
AV and DC                 c  1000000       600      9.145822  0.010475

                    Time
MC             46.085002
Antithetic     38.690917
Delta Control  83.875791
AV and DC      76.736579
```

The original Monte Carlo method is slower than the Antithetic variation as expected since the latter requires half as many randomly generated numbers while the Delta-based Control Variate methods are the slowest since I could not implement them without a for loop, which is computationally inefficient. Combining the Antithetic and Delta-based Control Variates decreased the run time slightly but not too significantly; however what is more notable is the decrease in standard error from implementing the Delta-based Control Variate. This is because this method incorporates the delta in the calculation to add information to the calculation of the option value since we know the effect from changes in the underlying at each step. The Antithetic Variate on the other hand, while it does add information by including the negative paths, truly independent paths add more information to the calculation so the value in this method is mainly that half as many intervals of Brownian Motion are required to be generated. This is seen in the table above where the standard error does not change significantly from the implementation of the Antithetic Variate but the run time decreases.

2 Problem 2: Finding the Value of a Mixed Asian Barrier Option Using Monte Carlo

[1]:

```
T = 60
B = 110
def ABO_MC(S,K,r,sig,T,B,m,q=0,antithetic=True):
    tau = T/360
    n = T*24
    dt = tau/n
    disc = np.exp(-r*tau)

    # generate m arrays of random intervals of BM of length n with mean 0␣
 ↪variance dt
```

```python
    if antithetic:
        m_half = int(m/2)
        dWs = np.array(np.split(rg.normal(0,np.sqrt(dt),n*m_half),m_half))
        dWs = np.concatenate((dWs,-dWs),axis=0)
    else:
        dWs = np.array(np.split(rg.normal(0,np.sqrt(dt),n*m),m))

    # create arrays of stock paths begnning arrays of the differentials
    S_arr = np.cumsum((r-q - .5*sig**2)*dt+sig*dWs,axis=1)
    # add the log of the stock price then input into exponential function for␣
↪array of stock values
    S_arr += np.log(S)
    S_arr = np.exp(S_arr)

    # set all paths that breach the boundary to 0 and remove them
    S_arr = np.where(S_arr > B,0,S_arr)
    S_arr = S_arr[np.apply_along_axis(min,1,S_arr)!=0]

    # define array of indices that represent the last hour of the day
    day_close = np.linspace(0,T*24,T+1)-1
    day_close = day_close[1:]
    day_sums = np.zeros(len(S_arr))

    # sum the stock value at the end of each day
    for i in range(len(day_close)):
        ind = int(day_close[i])
        day_sums += S_arr[:,ind]

    # divide
    A_T = day_sums/T
    opt_values = disc*np.maximum(A_T - K,0)
    return opt_values.sum()/m
```

Again the 2000000 sims crashed my computer so I had did 2 runs of 1000000 and averaged the results

[10]:
```python
run1 = ABO_MC(S,K,r,sig,T,B,1000000,q)
```

[11]:
```python
run2 = ABO_MC(S,K,r,sig,T,B,1000000,q)
```

[12]:
```python
(run1+run2)/2
```

[12]: 0.4249898566342799

3 Problem 3: A Futures Portfolio

[14]:
```
T = 2/12
X1 = 1000
X2 = 1000
V = 0.16

def gold_oil_spread(X1,X2,V,T,n,m):
    dt = T/n

    # Antithetic generation of Browninan motions
    m_half = int(m/2)
    dWs = np.split(rg.standard_normal(3*n*m_half),3)
    X1_arr = np.array(np.split(dWs[0],m_half))
    X1_arr = np.concatenate((X1_arr,-X1_arr),axis=0)
    X2_arr = np.array(np.split(dWs[1],m_half))
    X2_arr = np.concatenate((X2_arr,-X2_arr),axis=0)
    V_arr = np.array(np.split(dWs[2],m_half))
    V_arr = np.concatenate((V_arr,-V_arr),axis=0)

    # create Vt paths
    V_arr[:,0] = V + 10*(0.16-V)*dt + 0.3*np.sqrt(V*dt)*V_arr[:,0]
    for i in range(1,n):
        V_arr[:,i] = V_arr[:,i-1] + 10*(0.16-V_arr[:,i-1])*dt + 0.3*np.
↪sqrt(V_arr[:,i-1]*dt)*V_arr[:,i]

    # create arrays of X1 and X2 paths
    X1_arr = np.cumsum(0.001*dt + 0.1*np.sqrt(dt)*X1_arr,axis=1)
    X2_arr = np.cumsum(0.01*dt + np.sqrt(V_arr*dt)*X2_arr,axis=1)
    # add the log of the price then input into exponential function for array␣
↪of futures values
    X1_arr += np.log(X1)
    X2_arr += np.log(X2)
    X1_arr = np.exp(X1_arr)
    X2_arr = np.exp(X2_arr)
    # array of m stock values at maturity T
    F_T = X1_arr[:,n-1] - X2_arr[:,n-1]
    return F_T.mean()
```

[15]:
```
run1=gold_oil_spread(X1,X2,V,T,200,1000000)
```

[16]:
```
run2=gold_oil_spread(X1,X2,V,T,200,1000000)
```

[17]:
```
run3=gold_oil_spread(X1,X2,V,T,200,1000000)
```

[18]:
```
(run1+run2+run3)/3
```

[18]:
```
-14.094171305372605
```

4 Problem 3.1: A Futures Portfolio: Correlated Motions

[54]:
```
p1 = 0.4
p2 = -0.6
def gold_oil_spread_corr(X1,X2,V,T,p1,p2,n,m):
    # p1 is correlation between W1 and W2
    # p2 is correlation between W2 and Z
    dt = T/n
    # Antithetic generation of Browninan motions
    m_half = int(m/2)
    dWs = np.split(rg.standard_normal(3*n*m_half),3)
    X1_arr = np.array(np.split(dWs[0],m_half))
    X1_arr = np.concatenate((X1_arr,-X1_arr),axis=0)
    X2_arr = np.array(np.split(dWs[1],m_half))
    X2_arr = np.concatenate((X2_arr,-X2_arr),axis=0)
    X2_arr = p1*X1_arr + np.sqrt(1-p1**2)*X2_arr
    V_arr = np.array(np.split(dWs[2],m_half))
    V_arr = np.concatenate((V_arr,-V_arr),axis=0)
    V_arr = p2*X2_arr + np.sqrt(1-p2**2)*V_arr

    # create Vt paths
    V_arr[:,0] = V + 10*(0.16-V)*dt + 0.3*np.sqrt(V*dt)*V_arr[:,0]
    for i in range(1,n):
        V_arr[:,i] = V_arr[:,i-1] + 10*(0.16-V_arr[:,i-1])*dt + 0.3*np.
↪sqrt(V_arr[:,i-1]*dt)*V_arr[:,i]

    # create arrays of X1 and X2 paths
    X1_arr = np.cumsum(0.001*dt + 0.1*np.sqrt(dt)*X1_arr,axis=1)
    X2_arr = np.cumsum(0.01*dt + np.sqrt(V_arr*dt)*X2_arr,axis=1)
    # add the log of the price then input into exponential function for array␣
↪of futures values
    X1_arr += np.log(X1)
    X2_arr += np.log(X2)
    X1_arr = np.exp(X1_arr)
    X2_arr = np.exp(X2_arr)
    # array of m stock values at maturity T
    F_T = X1_arr[:,n-1] - X2_arr[:,n-1]
    return F_T.mean()
```

[55]:
```
run1 = gold_oil_spread_corr(X1,X2,V,T,p1,p2,200,1000000)
```

[56]:
```
run2 = gold_oil_spread_corr(X1,X2,V,T,p1,p2,200,1000000)
```

[57]:
```
run3 = gold_oil_spread_corr(X1,X2,V,T,p1,p2,200,1000000)
```

[58]:
```
(run1+run2+run3)/3
```

[58]:
```
1.1131399820992611
```

[66]:
```
.5*.15*.3
```

[66]: 0.0225

[81]:
```
def gold_oil_spread_corr_EM(X1,X2,V,T,p1,p2,n,m):
    # p1 is correlation between W1 and W2
    # p2 is correlation between W2 and Z
    dt = T/n
    # Antithetic generation of Browninan motions
    m_half = int(m/2)
    dWs = np.split(rg.standard_normal(3*n*m_half),3)
    X1_arr = np.array(np.split(dWs[0],m_half))
    X1_arr = np.concatenate((X1_arr,-X1_arr),axis=0)
    X2_arr = np.array(np.split(dWs[1],m_half))
    X2_arr = np.concatenate((X2_arr,-X2_arr),axis=0)
    X2_arr = p1*X1_arr + np.sqrt(1-p1**2)*X2_arr
    V_arr = np.array(np.split(dWs[2],m_half))
    V_arr = np.concatenate((V_arr,-V_arr),axis=0)
    V_arr = p2*X2_arr + np.sqrt(1-p2**2)*V_arr

    # create arrays of X1 paths (same as Euler method since we use the log of␣
 ↪the model)
    X1_arr = np.cumsum(0.001*dt + 0.1*np.sqrt(dt)*X1_arr,axis=1)
    # add the log of the price then input into exponential function for array␣
 ↪of futures values
    X1_arr += np.log(X1)
    X1_arr = np.exp(X1_arr)

    # create Vt and X2 paths using Euler-Milstein Method
    V_arr[:,0] = V + 10*(0.16-V)*dt + 0.3*np.sqrt(V*dt)*V_arr[:,0] + 0.
 ↪0225*(-dt + V_arr[:,0]**2)
    X2_arr[:,0] = X2 + 0.01*X2*dt + np.sqrt(V_arr[:,0]*dt)*X2*X2_arr[:,0] + .
 ↪5*V_arr[:,0]*X2*(-dt + X2_arr[:,0]**2)
    for i in range(1,n):
        V_arr[:,i] = V_arr[:,i-1] + 10*(0.16-V_arr[:,i-1])*dt + 0.3*np.
 ↪sqrt(V_arr[:,i-1]*dt)*V_arr[:,i] + \
        0.0225*(-dt + V_arr[:,i]**2)

        X2_arr[:,i] = X2_arr[:,i-1] + 0.01*X2_arr[:,i-1]*dt + np.sqrt(V_arr[:
 ↪,i]*dt)*X2_arr[:,i-1]*X2_arr[:,i] + \
        .5*V_arr[:,i]*X2_arr[:,i-1]*(-dt + X2_arr[:,i]**2)


    # array of m stock values at maturity T
    F_T = X1_arr[:,n-1] - X2_arr[:,n-1]
    return F_T.mean()
```

[83]:
```
gold_oil_spread_corr_EM(X1,X2,V,T,p1,p2,200,500000)
```

[83]: -4.51051268534234e+56

I was unable to finish the Euler-Midstein method; however, the main difference between the two is that the Euler-Midstein method incorporates a higher order term to improve accuracy and make a better approximation. The main difference from the previous problem is that there is some positive correlation between the two contracts that allows the gold contract to perform more closely to the oil contract despite the gold contract having a significantly lower drift coefficient. Moreover, the negative correlation between the oil contract and its stochastic volatilty cause oil to perform worse than in the previous example when its volatility increases. These correlations shrink the expected spread between these two contracts and even allows the gold contract to outperform to make the spread positive.

Problem 4: Parameter Estimation

```
setwd("/Users/Brendon/Documents/FE 621/HW 4")
df <- read.csv("2020SpringHW4SampleData.csv")
dt <- 0.0001
df <- ts(df[,2:4],deltat = dt)
head(df)

##       mydata1  mydata2  mydata3
## [1,] 2.000000 2.000000 2.000000
## [2,] 2.001132 2.001922 2.006852
## [3,] 2.003141 2.002217 2.023367
## [4,] 2.000863 2.000621 2.043867
## [5,] 2.000994 2.001031 2.045202
## [6,] 1.997924 1.999490 2.050986
```

Model 1:

dSt = theta1 * St * dt + theta2 * St^theta3 * dWt

```
library(Sim.DiffProc)

## Package 'Sim.DiffProc', version 4.5
## browseVignettes('Sim.DiffProc') for more informations.

f1 <- expression(theta[1]*x) # drift
g1 <- expression(theta[2]*x^theta[3]) # diffusion
aics1 <- c()
bics1 <- c()
logLs1 <- c()
for (i in 1:3) {
  fit <- fitsde(data = df[,i], drift = f1, diffusion = g1, start =
list(theta1=1,theta2=1,theta3=1))
  aics1 <- c(aics1,AIC(fit))
  bics1 <- c(bics1,BIC(fit))
  logLs1 <- c(logLs1,logLik(fit))
}

aic_sum_table <-
data.frame(mydata1=aics1[1],mydata2=aics1[2],mydata3=aics1[3],row.names =
c("Model 1"))
bic_sum_table <-
data.frame(mydata1=bics1[1],mydata2=bics1[2],mydata3=bics1[3],row.names =
c("Model 1"))
logL_sum_table <-
data.frame(mydata1=logLs1[1],mydata2=logLs1[2],mydata3=logLs1[3],row.names =
c("Model 1"))
```

Model 2:

dSt = (theta1 + theta2 * St)*dt* + theta3 * St^theta4 * dWt

```
f2 <- expression(theta[1] + theta[2]*x)
g2 <- expression(theta[3]*x^theta[4])
aics2 <- c()
bics2 <- c()
logLs2 <- c()
for (i in 1:3) {
  fit <- fitsde(data = df[,1], drift = f2, diffusion = g2, start =
list(theta1=1,theta2=1,theta3=1,theta4=1))
  aics2 <- c(aics2,AIC(fit))
  bics2 <- c(bics2,BIC(fit))
  logLs2 <- c(logLs2,logLik(fit))
}
aic_sum_table <-
rbind(aic_sum_table,data.frame(mydata1=aics2[1],mydata2=aics2[2],mydata3=aics
2[3],row.names = c("Model 2")))
bic_sum_table <-
rbind(bic_sum_table,data.frame(mydata1=bics2[1],mydata2=bics2[2],mydata3=bics
2[3],row.names = c("Model 2")))
logL_sum_table <-
rbind(logL_sum_table,data.frame(mydata1=logLs2[1],mydata2=logLs2[2],mydata3=l
ogLs2[3],row.names = c("Model 2")))
```

Model 3:

dSt = (theta1 + theta2 * St)*dt* + theta3 * sqrt(St)*dWt

```
f3 <- expression(theta[1] + theta[2]*x)
g3 <- expression(theta[3]*sqrt(x))
aics3 <- c()
bics3 <- c()
logLs3 <- c()
for (i in 1:3) {
  fit <- fitsde(data = df[,i], drift = f3, diffusion = g3, start =
list(theta1=1,theta2=1,theta3=1))
  aics3 <- c(aics3,AIC(fit))
  bics3 <- c(bics3,BIC(fit))
  logLs3 <- c(logLs3,logLik(fit))
}

aic_sum_table <-
rbind(aic_sum_table,data.frame(mydata1=aics3[1],mydata2=aics3[2],mydata3=aics
3[3],row.names = c("Model 3")))
bic_sum_table <-
rbind(bic_sum_table,data.frame(mydata1=bics3[1],mydata2=bics3[2],mydata3=bics
3[3],row.names = c("Model 3")))
logL_sum_table <-
```

```
rbind(logL_sum_table,data.frame(mydata1=logLs3[1],mydata2=logLs3[2],mydata3=l
ogLs3[3],row.names = c("Model 3")))
```

Model 4:

dSt = theta1 * dt + theta2 * St^theta3*dWt

```
f4 <- expression(theta[1])
g4 <- expression(theta[2]*x^theta[3])
aics4 <- c()
bics4 <- c()
logLs4 <- c()
for (i in 1:3) {
  fit <- fitsde(data = df[,i], drift = f4, diffusion = g4, start =
list(theta1=1,theta2=1,theta3=1))
  aics4 <- c(aics4,AIC(fit))
  bics4 <- c(bics4,BIC(fit))
  logLs4 <- c(logLs4,logLik(fit))
}

aic_sum_table <-
rbind(aic_sum_table,data.frame(mydata1=aics4[1],mydata2=aics4[2],mydata3=aics
4[3],row.names = c("Model 4")))
bic_sum_table <-
rbind(bic_sum_table,data.frame(mydata1=bics4[1],mydata2=bics4[2],mydata3=bics
4[3],row.names = c("Model 4")))
logL_sum_table <-
rbind(logL_sum_table,data.frame(mydata1=logLs4[1],mydata2=logLs4[2],mydata3=l
ogLs4[3],row.names = c("Model 4")))
```

Model 5

dSt = theta1 * St*dt + (theta2 + theta3 * St^theta4)*dWt

```
f5 <- expression(theta[1]*x)
g5 <- expression(theta[2] + theta[3]*x^theta[4])
aics5 <- c()
bics5 <- c()
logLs5 <- c()
for (i in 1:3) {
  fit <- fitsde(data = df[,i], drift = f5, diffusion = g5, start =
list(theta1=1,theta2=1,theta3=1,theta4=1))
  aics5 <- c(aics5,AIC(fit))
  bics5 <- c(bics5,BIC(fit))
  logLs5 <- c(logLs5,logLik(fit))
}

aic_sum_table <-
rbind(aic_sum_table,data.frame(mydata1=aics5[1],mydata2=aics5[2],mydata3=aics
5[3],row.names = c("Model 5")))
bic_sum_table <-
```

```
rbind(bic_sum_table,data.frame(mydata1=bics5[1],mydata2=bics5[2],mydata3=bics
5[3],row.names = c("Model 5")))
logL_sum_table <-
rbind(logL_sum_table,data.frame(mydata1=logLs5[1],mydata2=logLs5[2],mydata3=l
ogLs5[3],row.names = c("Model 5")))
```

Results Tables

```
print(paste("AIC Table"))

## [1] "AIC Table"

aic_sum_table

##           mydata1   mydata2   mydata3
## Model 1 -79765.23 -99826.96 -52764.89
## Model 2 -79763.01 -79763.01 -79763.01
## Model 3 -79699.26 -99765.15 -52765.02
## Model 4 -79762.43 -99882.82 -52766.73
## Model 5 -79755.84 -99821.34 -52766.95

print(paste("BIC Table"))

## [1] "BIC Table"

bic_sum_table

##           mydata1   mydata2   mydata3
## Model 1 -79752.81 -99814.54 -52752.46
## Model 2 -79752.58 -79752.58 -79752.58
## Model 3 -79686.84 -99752.72 -52752.60
## Model 4 -79750.01 -99870.40 -52754.31
## Model 5 -79745.42 -99810.92 -52756.53

print(paste("Extract Log-Likelihood Table"))

## [1] "Extract Log-Likelihood Table"

logL_sum_table

##          mydata1  mydata2  mydata3
## Model 1 39885.61 49916.48 26385.44
## Model 2 39885.50 39885.50 39885.50
## Model 3 39852.63 49885.57 26385.51
## Model 4 39884.22 49944.41 26386.37
## Model 5 39881.92 49914.67 26387.48
```

After applying the Euler method to each model and each data set the results show the first column of data, mydata1, most closely resembles model 1 since the model yields the smallest AIC and BIC while also having the largest Log-Likelihood. Model 4 can be assumed to be the source of the second column for the same reasons and model 2 is is signifcantly the best fit for the third column with respect to all three metrics.

Comparing Parameters From Each Method

Fitting Model 1 to Mydata1

```
pmle <- c("euler","ozaki","shoji","kessler")
fits1_1 <- lapply(1:4, function(i)
fitsde(df[,1],drift=f1,diffusion=g1,pmle=pmle[i],start =
list(theta1=1,theta2=1,theta3=1)))
coef_table1 <-
rbind(coef(fits1_1[[1]]),coef(fits1_1[[2]]),coef(fits1_1[[3]]),coef(fits1_1[[
4]]))
rownames(coef_table1) <- pmle

perf_table1 <-
rbind(cbind(AIC(fits1_1[[1]]),BIC(fits1_1[[1]]),logLik(fits1_1[[1]])),

cbind(AIC(fits1_1[[2]]),BIC(fits1_1[[2]]),logLik(fits1_1[[2]])),

cbind(AIC(fits1_1[[3]]),BIC(fits1_1[[3]]),logLik(fits1_1[[3]])),

cbind(AIC(fits1_1[[4]]),BIC(fits1_1[[4]]),logLik(fits1_1[[4]])))
rownames(perf_table1) <- pmle
colnames(perf_table1) <- c("AIC", "BIC", "Log-Like")

se_table1 <- cbind((confint(fits1_1[[1]])[,2]-
confint(fits1_1[[1]])[,1])/(2*1.96),
                   (confint(fits1_1[[2]])[,2]-
confint(fits1_1[[2]])[,1])/(2*1.96),
                   (confint(fits1_1[[3]])[,2]-
confint(fits1_1[[3]])[,1])/(2*1.96),
                   (confint(fits1_1[[4]])[,2]-
confint(fits1_1[[4]])[,1])/(2*1.96))
rownames(se_table1) <- c("theta1", "theta2", "theta3")
colnames(se_table1) <- pmle

print(paste("Table of coefficients from each method"))

## [1] "Table of coefficients from each method"

coef_table1

##            theta1    theta2    theta3
## euler   0.9338093 0.1983745 0.7110316
## ozaki   0.9344310 0.1983893 0.7109212
## shoji   0.9344310 0.1983894 0.7109212
## kessler 0.9339963 0.1983380 0.7111459

print(paste("Table of standard error for each coefficient from each method"))

## [1] "Table of standard error for each coefficient from each method"
```

```
se_table1

##              euler       ozaki       shoji     kessler
## theta1 0.141549363 0.141534742 0.141534776 0.141544637
## theta2 0.006095535 0.006095721 0.006095736 0.006096309
## theta3 0.026082024 0.026080576 0.026080630 0.026089117

print(paste("Table of performance metrics from each method"))

## [1] "Table of performance metrics from each method"

perf_table1

##               AIC       BIC Log-Like
## euler   -79765.23 -79752.81 39885.61
## ozaki   -79765.23 -79752.81 39885.61
## shoji   -79765.23 -79752.81 39885.61
## kessler -79765.23 -79752.81 39885.61
```

Fitting Model 4 to Mydata2

```
fits4_2 <- lapply(1:4, function(i)
fitsde(df[,2],drift=f4,diffusion=g4,pmle=pmle[i],start =
list(theta1=1,theta2=1,theta3=1)))

coef_table2 <-
rbind(coef(fits4_2[[1]]),coef(fits4_2[[2]]),coef(fits4_2[[3]]),coef(fits4_2[[
4]]))
rownames(coef_table2) <- pmle

perf_table2 <-
rbind(cbind(AIC(fits4_2[[1]]),BIC(fits4_2[[1]]),logLik(fits4_2[[1]])),

cbind(AIC(fits4_2[[2]]),BIC(fits4_2[[2]]),logLik(fits4_2[[2]])),

cbind(AIC(fits4_2[[3]]),BIC(fits4_2[[3]]),logLik(fits4_2[[3]])),

cbind(AIC(fits4_2[[4]]),BIC(fits4_2[[4]]),logLik(fits4_2[[4]])))
rownames(perf_table2) <- pmle
colnames(perf_table2) <- c("AIC", "BIC", "Log-Like")

se_table2 <- cbind((confint(fits4_2[[1]])[,2]-
confint(fits4_2[[1]])[,1])/(2*1.96),
                   (confint(fits4_2[[4]])[,2]-
confint(fits4_2[[4]])[,1])/(2*1.96))
rownames(se_table2) <- c("theta1", "theta2", "theta3")
colnames(se_table2) <- c(pmle[1],pmle[4])

print(paste("Table of coefficients from each method"))

## [1] "Table of coefficients from each method"
```

```
coef_table2

##            theta1    theta2    theta3
## euler   8.437439 0.1029783 0.2770873
## ozaki   1.000000 1.0000000 1.0000000
## shoji   1.000000 1.0000000 1.0000000
## kessler 8.437180 0.1029737 0.2771065

print(paste("Table of standard error for each coefficient from each method"))

## [1] "Table of standard error for each coefficient from each method"

se_table2

##               euler     kessler
## theta1 0.162871919 0.163129626
## theta2 0.002695944 0.002696522
## theta3 0.015024290 0.015027530

print(paste("Table of performance metrics from each method"))

## [1] "Table of performance metrics from each method"

perf_table2

##                AIC          BIC Log-Like
## euler   -99882.82 -99870.40219 49944.41
## ozaki         6.00     18.42088     0.00
## shoji         6.00     18.42088     0.00
## kessler -99882.82 -99870.40269 49944.41
```

Fitting Model 2 to Mydata3

```
fits2_3 <- lapply(1:4, function(i)
fitsde(df[,3],drift=f2,diffusion=g2,pmle=pmle[i],start =
list(theta1=1,theta2=1,theta3=1,theta4=1)))
coef_table3 <-
rbind(coef(fits2_3[[1]]),coef(fits2_3[[2]]),coef(fits2_3[[3]]),coef(fits2_3[[
4]]))
rownames(coef_table3) <- pmle

perf_table3 <-
rbind(cbind(AIC(fits2_3[[1]]),BIC(fits2_3[[1]]),logLik(fits2_3[[1]])),

cbind(AIC(fits2_3[[2]]),BIC(fits2_3[[2]]),logLik(fits2_3[[2]])),

cbind(AIC(fits2_3[[3]]),BIC(fits2_3[[3]]),logLik(fits2_3[[3]])),

cbind(AIC(fits2_3[[4]]),BIC(fits2_3[[4]]),logLik(fits2_3[[4]])))
rownames(perf_table3) <- pmle
colnames(perf_table3) <- c("AIC", "BIC", "Log-Like")
```

```
se_table3 <- cbind((confint(fits2_3[[1]])[,2]-
confint(fits2_3[[1]])[,1])/(2*1.96),
                   (confint(fits2_3[[2]])[,2]-
confint(fits2_3[[2]])[,1])/(2*1.96),
                   (confint(fits2_3[[3]])[,2]-
confint(fits2_3[[3]])[,1])/(2*1.96),
                   (confint(fits2_3[[4]])[,2]-
confint(fits2_3[[4]])[,1])/(2*1.96))
rownames(se_table3) <- c("theta1", "theta2", "theta3", "theta4")
colnames(se_table3) <- pmle

print(paste("Table of coefficients from each method"))

## [1] "Table of coefficients from each method"

coef_table3

##              theta1     theta2    theta3    theta4
## euler     0.9393161  0.5388178 0.8755603 0.5170062
## ozaki     0.9387800  0.5389677 0.8755002 0.5170301
## shoji    12.8988675 -2.5727091 0.8752303 0.5172636
## kessler   0.9388430  0.5390427 0.8755055 0.5170194

print(paste("Table of standard error for each coefficient from each method"))

## [1] "Table of standard error for each coefficient from each method"

se_table3

##              euler      ozaki      shoji    kessler
## theta1 6.37551091 6.37584253 6.37327462 6.42076835
## theta2 1.71620334 1.71619816 1.71577700 1.72748031
## theta3 0.03034120 0.03034312 0.03033087 0.03034243
## theta4 0.02576978 0.02577222 0.02577266 0.02577194

print(paste("Table of performance metrics from each method"))

## [1] "Table of performance metrics from each method"

perf_table3

##                AIC       BIC Log-Like
## euler   -52763.46 -52753.04 26385.73
## ozaki   -52763.46 -52753.04 26385.73
## shoji   -52766.98 -52756.56 26387.49
## kessler -52763.46 -52753.04 26385.73
```

This leads me to the conclusion that the Euler method yields the best estimates. The tables show the coefficients and standard errors from the Euler and Kessler methods are very close in all three cases. The Ozaki and Shoji-Ozaki methods were unable to output meaningful results for the second data set suggesting that they are not as versatile as the others, and seeing that they do not provide a signifcantly better fit in the other sets, I am reluctant to choose either method as the one to provide the best estimates. The Shoji-Ozaki method did find slightly lower information criteria and standard error for the last data set with signifcantly different estimates for theta1 and theta2 from the other methods; however, the difference in standard error is not signifcant enough to warant selection over the Euler method. The Kessler method employs a more complicated high order Taylor expansion approach to parameterization; however, this fails to yield a signifcant benefit over the Euler method in terms of information criteria or standard error and actually underperforms in some cases. Since the added complexity of this method adds significantly to the run time without any significant benefit in performance, I have come to the conclusion that the Euler method provides the best estimates since its simplicity relative to the alternatives allows it to run faster and be more flexible, and these benefits do not come at the expense of accuracy.