In [ ]:
```
import sys
sys.path.append('D:/Study/2023 spring/FINTECH545 Quant Risk/')
from risk_lib import bsm, covariance_estimation_techniques, non_psd_fixes, riskStat
```

Problem 1

1. Using the data in "problem1.csv" a. Calculate Log Returns (2pts) b. Calculate Pairwise Covariance (4pt) c. Is this Matrix PSD? If not, fix it with the "near_psd" method (2pt) d. Discuss when you might see data like this in the real world. (2pt)

In [ ]:
```
import pandas as pd
p1 = pd.read_csv('problem1.csv')
p1.head()
```

Out[ ]:

	Price1	Price2	Price3	Date
0	94.183334	100.328416	105.194707	2023-04-12
1	96.704466	106.165696	107.098697	2023-04-13
2	95.361978	102.318377	NaN	2023-04-14
3	96.169666	104.488549	106.306717	2023-04-15
4	96.819714	106.475325	107.043957	2023-04-16

In [ ]:
```
log_r = riskStats.return_calculate(p1, 'LOG')
log_r = log_r.iloc[:, 1:]
# This is the log return
log_r
```

Out[ ]:

	Price1	Price2	Price3
1	0.026416	0.056552	0.017938
2	-0.013980	-0.036912	NaN
3	0.008434	0.020988	NaN
4	0.006737	0.018836	0.006911
5	-0.002865	-0.008917	-0.002980
6	-0.004938	0.004920	0.002172
7	0.001848	-0.020292	-0.014717
8	-0.003057	0.005509	0.006038
9	-0.004712	-0.009511	0.003515
10	0.005624	0.016691	0.003364
11	0.006577	0.012261	0.005265
12	-0.001007	-0.009713	-0.005714
13	0.005900	0.026144	0.007212
14	NaN	NaN	-0.014589
15	NaN	NaN	0.013114
16	-0.004866	-0.009715	-0.009421
17	-0.002944	-0.014893	-0.000713
18	NaN	0.000600	-0.000688
19	NaN	NaN	NaN

In [ ]:

```
def pcov(df):
    vars =df.var()
    std = np.sqrt(vars)
    # Get the pearson correlation matrix
    corr = np.corrcoef(df,rowvar=False)
    cov = np.diag(std) @ corr @ np.diag(std)
    return cov

p_cov = covar.missing_cov(log_r,skipmiss=False, func = pcov)
# This is the pairwise correlation
p_cov
```

Out[ ]:

```
array([[8.46145916e-05, 1.89639683e-04, 4.70100075e-05],
       [1.89639683e-04, 4.89528203e-04, 1.42914435e-04],
       [4.70100075e-05, 1.42914435e-04, 8.19544084e-05]])
```

In [ ]:

```
import numpy as np

np.linalg.eig(p_cov)
# The eigenvalues are all positive, so this is a PSD matrix
```

Out[ ]:

```
(array([6.07412292e-04, 7.39275901e-06, 4.12921522e-05]),
 array([[-0.34961036, -0.87362902, -0.33844487],
        [-0.89566344,  0.4176416 , -0.15284795],
        [-0.27488106, -0.24969547,  0.92848941]]))
```

In [ ]:

```
# Price1 and Price2 have a high positive correlation of 0.900665, which means that t
# Price3 also has a positive correlation with both Price1 and Price2, but the correl
```

```
# This can be seen when one of the three is a security and the other two are its fin
```

Problem 2

1. “problem2.csv” contains data about a call option. Time to maturity is given in days. Assume 255 days in a year. a. Calculate the call price (1pt) b. Calculate Delta (1pt) c. Calculate Gamma (1pt) d. Calculate Vega (1pt) e. Calculate Rho (1pt) Assume you are long 1 share of underlying and are short 1 call option. Using Monte Carlo assuming a Normal distribution of arithmetic returns where the implied volatility is the annual volatility and 0 mean f. Calculate VaR at 5% (2pt) g. Calculate ES at 5% (2pt) h. This portfolio’s payoff structure most closely resembles what? (1pt)

In [ ]:
```
p2 = pd.read_csv('problem2.csv')
df = p2
```

In [ ]:
```
S = df['Underlying'].iloc[0]
K = df['Strike'].iloc[0]
sigma = df['IV'].iloc[0]
T = df['TTM'].iloc[0] / 255
r = df['RF'].iloc[0]
q = df['DivRate'].iloc[0]
```

In [ ]:
```
bsm.gbsm_greeks(S, K, 0, T, r, q, sigma, option_type='call')
```

Out[ ]:
```
(5.480608877402638,
 0.5789738538909803,
 0.03297181407321243,
 27.135022800729487,
 -4.439070911517648,
 27.038088730337417)
```

In [ ]:
```
# a. Calculate the call price: 5.480608877402638
# b. Calculate Delta (1pt): 0.5789738538909803
# c. Calculate Gamma (1pt): 0.03297181407321243
# d. Calculate Vega (1pt): 27.135022800729487
# e. Calculate Rho (1pt): -4.439070911517648
```

In [ ]:
```
import pandas as pd
import numpy as np
from scipy.stats import norm

num_scenarios = 10000

# Generate random scenarios for future price
scenarios = S * np.exp((r - q - 0.5 * sigma ** 2) * T + sigma * np.sqrt(T) * np.ra

# Calculate portfolio value for each scenario
portfolio_values = -np.maximum(scenarios - K, 0) + scenarios

# Calculate VaR at 5%
var = riskStats.VAR(portfolio_values)

# Calculate ES at 5%
es = riskStats.ES(portfolio_values)

print(f'VaR at 5%: {-var}')
print(f'ES at 5%: {-es}')
```

```
VaR at 5%: 81.99313948083511
ES at 5%: 78.20025352028976
```

In [ ]:
```
from scipy.stats import skew, kurtosis
skew(portfolio_values), kurtosis(portfolio_values)
```

Out[ ]:
```
(-1.6602361215309842, 2.2506286032826104)
```

In [ ]:
```
# Negative Skew makes this a risky investment because most of portfolio values fall
# This is seen with the large VaR and ES numbers.
```

Problem 3

1. Data in “problem2_cov.csv” is the covariance for 3 assets. “problem3_ER.csv” is the expected return for each asset as well as the risk free rate. a. Calculate the Maximum Sharpe Ratio Portfolio (4pt) b. Calculate the Risk Parity Portfolio (4pt) c. Compare the differences between the portfolio and explain why. (2pt)

In [ ]:
```
cov = pd.read_csv('problem3_cov.csv')
er3 = pd.read_csv('problem3_ER.csv')
```

In [ ]:
```
cov.values
```

Out[ ]:
```
array([[0.03847047, 0.03556668, 0.03726546],
       [0.03556668, 0.03567933, 0.03588815],
       [0.03726546, 0.03588815, 0.03916904]])
```

In [ ]:
```
er3
```

Out[ ]:

	RF	Expected_Value_1	Expected_Value_2	Expected_Value_3
0	0.045	0.141188	0.137633	0.142058

In [ ]:
```
from scipy.optimize import minimize

er = [0.141188372907701, 0.137633309103119, 0.142057758369346]
rf = 0.045

def max_sharpe_ratio_weights(exp_returns, rf, cov_matrix, restrict="True"):
    num_stocks = len(exp_returns)

    # Define the Sharpe Ratio objective function to be minimized
    def neg_sharpe_ratio(weights):
        port_return = np.dot(weights, exp_returns)
        port_volatility = np.sqrt(np.dot(weights.T, np.dot(cov_matrix, weights)))
        sharpe_ratio = (port_return - rf) / port_volatility
        return -sharpe_ratio

    # Define the constraints
    constraints = ({'type': 'eq', 'fun': lambda x: np.sum(x) - 1}) # The sum of the
    if restrict == "True":
        bounds = tuple([(0, 1) for i in range(num_stocks)]) # The weights must be b
        initial_weights = np.ones(num_stocks) / num_stocks # Start with equal weight
        opt_results = minimize(neg_sharpe_ratio, initial_weights, method='SLSQP', bc
    elif restrict == "False":
        bounds = tuple([(-1, 1) for i in range(num_stocks)]) # The weights must be
        initial_weights = np.ones(num_stocks) / num_stocks # Start with equal weight
        opt_results = minimize(neg_sharpe_ratio, initial_weights, method='SLSQP', cc
```

```python
    # Find the portfolio weights that maximize the Sharpe Ratio

    return opt_results.x.round(4), -opt_results.fun

max_sharpe_ratio_weights(er, rf, cov.values)
```

Out[ ]:

```
(array([0.3333, 0.3333, 0.3333]), 0.4970805979987282)
```

In [ ]:

```python
# The maximum sharpe portfolio is [0.3333, 0.3333, 0.3333].
```

In [ ]:

```python
def risk_parity_weights(covar):
    n = covar.shape[1]
    def pvol(x):
        return np.sqrt(x.T @ covar @ x)

    def pCSD(x):
        p_vol = pvol(x)
        csd = x * (covar @ x) / p_vol
        return csd

    def sseCSD(x):
        csd = pCSD(x)
        mCSD = np.sum(csd) / n
        dCsd = csd - mCSD
        se = dCsd * dCsd
        return 1.0e5 * np.sum(se)

    # Constraints
    cons = ({'type': 'eq', 'fun': lambda w: np.sum(w) - 1})

    # Bounds
    bnds = [(0, None) for _ in range(n)]

    # Initial guess
    x0 = np.array([1/n] * n)
    res = minimize(sseCSD, x0, method='SLSQP', bounds=bnds, constraints=cons)
    return np.round(res.x, decimals=4)

risk_parity_weights(cov.values)
```

Out[ ]:

```
array([0.3301, 0.3428, 0.3271])
```

In [ ]:

```python
# The risk parity portfolio is [0.3301, 0.3428, 0.3271].
```

In [ ]:

```python
# The risk parity portfolio has more weights on the second asset and less weight on
# This is because the second asset has a lower return and lower volatility.
# On the other hand, the third asset has the highest return and volatility.
```

Problem 4

1. Data in “problem4_returns.csv” is a series of returns for 3 assets. “problem4_startWeight.csv” is the starting weights of a portfolio of these assets as of the first day in the return series. a. Calculate the new weights for the start of each time period (2pt) b. Calculate the ex-post return attribution of the portfolio on each asset (4pt) c. Calculate the ex-post risk attribution of the portfolio on each asset (2pt)

In [ ]:
```
p4_r = pd.read_csv('problem4_returns.csv')
p4_r.head()
```

Out[ ]:

	Asset1	Asset2	Asset3	Date
0	-0.046684	0.041869	-0.004892	2023-04-12
1	-0.090208	0.015082	-0.031945	2023-04-13
2	-0.047394	-0.145864	0.053473	2023-04-14
3	-0.053670	0.010180	-0.042510	2023-04-15
4	0.048936	-0.034171	-0.141175	2023-04-16

In [ ]:
```
p4_w = pd.read_csv('problem4_startWeight.csv')
p4_w
```

Out[ ]:

	weight1	weight2	weight3
0	0.521223	0.360809	0.117968

In [ ]:
```
# Load data
returns = p4_r.iloc[:, :3]
start_weight = pd.read_csv('problem4_startWeight.csv')

# Calculate new weights
w = start_weight.copy()
for i in range(1, len(returns)):
    w.loc[i] = w.loc[i-1] * (1 + returns.iloc[i-1].values)
    w.loc[i] /= w.loc[i].sum()

# These are the new weights
w
```

Out[ ]:

	weight1	weight2	weight3
0	0.521223	0.360809	0.117968
1	0.501810	0.379637	0.118553
2	0.477220	0.402817	0.119964
3	0.491440	0.371940	0.136619
4	0.478657	0.386708	0.134635
5	0.506537	0.376809	0.116654
6	0.496037	0.387226	0.116736
7	0.490277	0.396832	0.112891
8	0.466052	0.421569	0.112380
9	0.463343	0.413444	0.123212
10	0.441196	0.429086	0.129718
11	0.444068	0.427784	0.128148
12	0.460563	0.400102	0.139335
13	0.467940	0.398165	0.133895
14	0.500517	0.367254	0.132229
15	0.488969	0.375246	0.135785
16	0.452390	0.390544	0.157066
17	0.437223	0.409081	0.153696
18	0.451568	0.413136	0.135296
19	0.423560	0.435019	0.141421

In [ ]:

```
stocks = ['Asset1', 'Asset2', 'Asset3']
optimal_weights = w.iloc[-1]

# Calculate portfolio return and updated weights for each day
n = p4_r.shape[0]
m = len(stocks)

pReturn = np.empty(n)
weights = np.empty((n, len(optimal_weights)))
lastW = optimal_weights.copy()
matReturns = p4_r[stocks].values

for i in range(n):
    # Save Current Weights in Matrix
    weights[i, :] = lastW

    # Update Weights by return
    lastW = lastW * (1.0 + matReturns[i, :])

    # Portfolio return is the sum of the updated weights
    pR = lastW.sum()

    # Normalize the weights back so sum = 1
    lastW = lastW / pR

    # Store the return
```

```python
    pReturn[i] = pR - 1

# Set the portfolio return in the Update Return DataFrame
p4_r["Portfolio"] = pReturn

# Calculate the total return
totalRet = np.exp(np.sum(np.log(pReturn + 1))) - 1

# Calculate the Carino K
k = np.log(totalRet + 1) / totalRet

# Carino k_t is the ratio scaled by 1/K
carinoK = np.log(1.0 + pReturn) / pReturn / k

# Calculate the return attribution
attrib = pd.DataFrame(matReturns * weights * carinoK[:, np.newaxis], columns=stocks

# Set up a DataFrame for output
Attribution = pd.DataFrame({"Value": ["TotalReturn", "Return Attribution"]})

# Loop over the stocks
for s in stocks + ["Portfolio"]:
    # Total Stock return over the period
    tr = np.exp(np.sum(np.log(p4_r[s] + 1))) - 1

    # Attribution Return (total portfolio return if we are updating the portfolio co
    atr = tr if s == "Portfolio" else attrib[s].sum()

    # Set the values
    Attribution[s] = [tr, atr]

# Check that the attribution sums back to the total Portfolio return
assert np.isclose(Attribution.iloc[1, 1:len(stocks) + 1].sum(), totalRet)

# Realized Volatility Attribution

# Y is our stock returns scaled by their weight at each time
Y = matReturns * weights

# Set up X with the Portfolio Return
X = np.column_stack((np.ones(n), pReturn))

# Calculate the Beta and discard the intercept
B = np.linalg.inv(X.T @ X) @ X.T @ Y
B = B[1, :]

# Component SD is Beta times the standard deviation of the portfolio
cSD = B * np.std(pReturn)

# Check that the sum of component SD is equal to the portfolio SD
assert np.isclose(cSD.sum(), np.std(pReturn))

# Add the Vol attribution to the output
vol_attrib = pd.DataFrame({"Value": ["Vol Attribution"], **{stocks[i]: [cSD[i]] for

Attribution = pd.concat([Attribution, vol_attrib], ignore_index=True)

print(Attribution)
```

```
                Value    Asset1    Asset2    Asset3  Portfolio
0         TotalReturn -0.426018 -0.104639 -0.128839  -0.244185
1  Return Attribution -0.181572 -0.042268 -0.020345  -0.244185
2     Vol Attribution  0.012737  0.017603  0.002870   0.033210
```

In [ ]:
```
# These are the return attribution and risk attribution
```

Problem 5

In [ ]:
```
p5 = pd.read_csv('problem5.csv')
prices = p5.iloc[:, :4]
returns = prices.pct_change().dropna(how='all')
returns = returns - np.mean(returns)
returns.head()
```

```
d:\Anaconda\lib\site-packages\numpy\core\fromnumeric.py:3438: FutureWarning: In a fu
ture version, DataFrame.mean(axis=None) will return a scalar mean over the entire Da
taFrame. To retain the old behavior, use 'frame.mean(axis=0)' or just 'frame.mean()'
  return mean(axis=axis, dtype=dtype, out=out, **kwargs)
```

Out[ ]:

	Price1	Price2	Price3	Price4
1	-3.385251e-07	-0.000078	-0.000060	-0.000180
2	-9.558203e-05	0.000082	-0.000652	-0.000029
3	6.247609e-04	0.000088	0.000634	0.000166
4	-5.667665e-04	-0.000287	-0.000018	-0.000133
5	2.681479e-04	0.000253	0.000115	0.000275

In [ ]:
```
from scipy.stats import t, norm, spearmanr, multivariate_normal

Y = returns.values
corsp = np.cov(Y)

nSim = 5000

models = [t.fit(Y[:, i]) for i in range(4)]
U = np.column_stack([(Y[:, i] - loc) / scale for i, (df, loc, scale) in enumerate(m

corsp = spearmanr(U).correlation
_simU = norm.cdf(multivariate_normal.rvs(mean=np.zeros(4), cov=corsp, size=nSim)).T
simReturn = np.empty_like(_simU)

for i in range(4):
    df, loc, scale = models[i]
    simReturn[:, i] = t.ppf(_simU[:, i], df=df, loc=loc, scale=scale)

def _VAR(w):
    x = np.array(w)
    r = np.sum(simReturn * x[:, np.newaxis], axis=0)
    return riskStats.VAR(r)
```

In [ ]:
```
price1 = prices[['Price1']]
price2 = prices[['Price2']]
price3 = prices[['Price3']]
price4 = prices[['Price4']]
```

In [ ]:
```
# VAR of asset1
np.mean(_VAR(price1))
```

Out[ ]:
```
-2314.051569110883
```

In [ ]:
```
# VAR of asset2
np.mean(_VAR(price2))
```
Out[ ]:
```
-2953.6580562080417
```
In [ ]:
```
# VAR of asset3
np.mean(_VAR(price3))
```
Out[ ]:
```
-3316.7546082236804
```
In [ ]:
```
# VAR of asset4
np.mean(_VAR(price4))
```
Out[ ]:
```
-3214.5226492329225
```
In [ ]:
```
# VAR of asset1&2
np.mean(_VAR(price4))
```
Out[ ]:
```
-3214.5226492329225
```