Príloha č. 1: Zdrojový kód

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jan  5 10:44:08 2023

@author: michal
"""
locals().clear()
globals().clear()

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
from scipy.stats import norm
import quantstats as qs
qs.extend_pandas()
import time

start_time = time.time()

# data import via csv, that was downloaded from Bloomberg terminal
nasdaq_data =
pd.read_csv(r"C:\Users\micha\Desktop\Diplomovka\Code\NASDAQ_raw.csv")
nasdaq_data = nasdaq_data[["Date", "Adj Close"]]
nasdaq_returns = nasdaq_data["Adj Close"].pct_change()
arr = nasdaq_returns.values

nasdaq_returns = pd.DataFrame(nasdaq_returns)
bonds_data =
pd.read_csv(r"C:\Users\micha\Desktop\Diplomovka\Code\bonds_data.csv")


def GBM(mu, sigma, S0, dt, T, num_paths):
    """
    simulated stock price paths using the Geometric Brownian Motion
    model.

    Parameters:
    mu: expected return
    sigma: volatility
    S0: initial index value
    dt: time step
    T: total number of trading days
    num_paths: number of index value paths to generate
    """
    np.random.seed(0)
    wiener = np.random.normal(size=(T, num_paths))
    S = np.zeros((T+1, num_paths))
    S[0] = S0
    for t in range(1, T+1):
        S[t] = S[t-1] * np.exp((mu - 0.5*sigma**2)*dt +
sigma*np.sqrt(dt)*wiener[t-1])

    return S

# Simulation setup
mu = 0.127454 # expected return
sigma = 0.3 # volatility
S0 = 11196.2 # price at 14.11.2022
```

```
dt = 1/252 # daily
T = 252 # number of simulated days
num_paths = 10000 # number of simulated paths

Sim = GBM(mu, sigma, S0, dt, T, num_paths)

# Plot some results of the simulation
fig, ax = plt.subplots(figsize=(8, 5))
ax.plot(Sim[:, :100])
ax.set_title('100 simulated paths of NASDAQ')
ax.set_xlabel('Trading days')
ax.set_ylabel('Index value')
plt.show()

# normal prob plot
fig, ax = plt.subplots(figsize=(8, 5))
res = stats.probplot(arr, plot=ax)
ax.set_xlabel("Quantiles of normal distribution")
ax.set_ylabel("Quantiles of input sample")
ax.set_title("Normal probability plot of NASDAQ returns")
plt.show()

# histogram
plt.hist(arr, bins=70)
plt.axvline(0.000506, color='k', linestyle='dashed', linewidth=1)
plt.title('Frequency of NASDAQ returns')
plt.show()

# returns of NASDAQ in general
nasdaq_data["Date"]=pd.to_datetime(nasdaq_data["Date"])
plt.plot(nasdaq_data["Date"], nasdaq_data["Adj Close"])
plt.title('NASDAQ')
plt.show()

# index to returns
Sim = pd.DataFrame(Sim)
Sim_prices = Sim
Sim = Sim.pct_change(axis = 0)
Sim_returns = Sim
Sim = Sim.applymap(lambda x: x+1)

#####################################################################
'''
Mechanics of strategies
BH, CM, SL, CPPI, TIPP, SP
'''
inv_amount = 100 # initial amount we invest
riskfree_return = 0.01208516126 # bond yields per quarter

#BH (allocation = % of bonds allocation)
def BH(allocation):
    bh_t = 0
    bh_riskfree = inv_amount * allocation
    bh_risky = inv_amount - bh_riskfree
    bh_risky = np.full(num_paths, bh_risky)
    bh_risky = pd.DataFrame([bh_risky])

    while bh_t <= T-1:
        bh_t += 1
        bh_risky = bh_risky * Sim.loc[bh_t]
```

```
        if bh_t == 63 or bh_t == 126 or bh_t == 189 or bh_t == T:
            bh_riskfree = bh_riskfree * (1 + riskfree_return)

    bh_results = bh_risky.applymap(lambda x: x+bh_riskfree)
    return bh_results

#CM (allocation = % of bonds allocation)
def CM(allocation):
    allocation = 0.5
    cm_t = 0
    cm_riskfree = inv_amount * allocation
    cm_risky = inv_amount - cm_riskfree
    cm_risky = np.full(num_paths, cm_risky)
    cm_risky = pd.DataFrame([cm_risky])
    cm_riskfree = np.full(num_paths, cm_riskfree)
    cm_riskfree = pd.DataFrame([cm_riskfree])
    cm_portfolio_value = cm_riskfree + cm_risky

    while cm_t <= T-1:
        cm_t += 1
        cm_risky = cm_risky * Sim.loc[cm_t]
        if cm_t == 63 or cm_t == 126 or cm_t == 189 or cm_t == T:
           cm_riskfree = cm_riskfree * (1 + riskfree_return)

        cm_portfolio_value = cm_risky + cm_riskfree

        # rebalancing
        cm_riskfree = cm_portfolio_value * allocation
        cm_risky = cm_portfolio_value * (1-allocation)

    cm_results = cm_portfolio_value
    return cm_results

#SL (sl_floor_prct = % of the floor)
def SL(sl_floor_prct):
    allocation = 0
    sl_floor = inv_amount * sl_floor_prct
    sl_t = 0
    sl_riskfree = inv_amount * allocation
    sl_risky = inv_amount - sl_riskfree
    sl_risky = np.full(num_paths, sl_risky)
    sl_risky = pd.DataFrame([sl_risky])
    sl_portfolio_value = sl_risky
    popped_portfolio = pd.DataFrame()
    portfolio_new = pd.DataFrame()

    while sl_t <= T-1:
        sl_t += 1
        if sl_t == 1:
            sl_portfolio_value = sl_portfolio_value * Sim.loc[sl_t]
        if sl_t == 63 or sl_t == 126 or sl_t == 189 or sl_t == T:
            sl_floor = sl_floor * (1 + riskfree_return)

        #rebalancing
        mask = sl_portfolio_value.iloc[0] < sl_floor
        popped_portfolio = sl_portfolio_value.loc[:, mask].copy()
        portfolio_new = pd.concat([portfolio_new, popped_portfolio], axis
        = 1)
        columns_drop = sl_portfolio_value.columns[mask]
        columns_keep = sl_portfolio_value.columns[~mask]
        sl_portfolio_value.drop(columns_drop, axis=1, inplace=True)
```

```
        sl_portfolio_value = sl_portfolio_value *
Sim.loc[sl_t,columns_keep]
        if sl_t == 63 or sl_t == 126 or sl_t == 189 or sl_t == T:
            portfolio_new = portfolio_new * (1 + riskfree_return)


    sl_results = pd.concat([sl_portfolio_value, portfolio_new], axis =1)
    return sl_results

#CPPI (cppi_floor = % of the floor, cppi_multiplier = multiplier for
strategy)
def CPPI(cppi_floor, cppi_multiplier):
    cppi_t = 0
    cppi_risky = pd.DataFrame([np.full(num_paths,0)])
    cppi_riskfree = pd.DataFrame([np.full(num_paths,0)])
    cppi_portfolio_value = np.full(num_paths, 0)
    cppi_portfolio_value = pd.DataFrame([cppi_portfolio_value])
    cushion = pd.DataFrame([np.full(num_paths, (inv_amount - (inv_amount
    * cppi_floor)))])
    cppi_risky = cushion * cppi_multiplier
    cppi_riskfree = inv_amount - cppi_risky

    while cppi_t <= T-1:
        cppi_t += 1

        cppi_risky = cppi_risky * Sim.loc[cppi_t]
        if cppi_t == 63 or cppi_t == 126 or cppi_t == 189 or cppi_t == T:
           cppi_riskfree = cppi_riskfree * (1 + riskfree_return)
        cppi_portfolio_value = cppi_risky + cppi_riskfree

        # rebalancing of strategy
        cushion = cppi_portfolio_value - (inv_amount * cppi_floor)
        cppi_risky = cushion * cppi_multiplier
        mask = cppi_risky > cppi_portfolio_value
        cppi_risky = cppi_risky.where(~mask, cppi_portfolio_value)
        mask = mask.drop(index=mask.index)
        cppi_riskfree = cppi_portfolio_value - cppi_risky

    cppi_results = cppi_portfolio_value
    return cppi_results

#TIPP (tipp_floor_prctr = % of the floor, tipp_multiplier = multiplier
for strategy)
def TIPP(tipp_floor_prct, tipp_multiplier):
    tipp_t = 0
    tipp_risky = pd.DataFrame([np.full(num_paths,0)])
    tipp_riskfree = pd.DataFrame([np.full(num_paths,0)])
    tipp_portfolio_value = np.full(num_paths, inv_amount)
    tipp_portfolio_value = pd.DataFrame([tipp_portfolio_value])
    cushion = pd.DataFrame([np.full(num_paths, (inv_amount - (inv_amount
* tipp_floor_prct)))])
    tipp_risky = cushion * tipp_multiplier
    tipp_riskfree = inv_amount - tipp_risky
    tipp_floor =
pd.DataFrame([np.full(num_paths,inv_amount*tipp_floor_prct)])

    while tipp_t <= T-1:
        tipp_t += 1
```

```
        tipp_floor_old = tipp_portfolio_value * tipp_floor_prct # store
the old floor from t-1

        tipp_risky = tipp_risky * Sim.loc[tipp_t]
        if tipp_t == 63 or tipp_t == 126 or tipp_t == 189 or tipp_t == T:
           tipp_riskfree = tipp_riskfree * (1 + riskfree_return)
        tipp_portfolio_value = tipp_risky + tipp_riskfree

        # rebalancing of strategy
        tipp_floor = tipp_portfolio_value * tipp_floor_prct # new floor
        mask_floor = tipp_floor > tipp_floor_old #floor rebalancing
        tipp_floor = tipp_floor.where(~mask_floor, tipp_floor_old) #floor
rebalancing

        cushion = tipp_portfolio_value - tipp_floor
        tipp_risky = cushion * tipp_multiplier
        mask = tipp_risky > tipp_portfolio_value #no short sale
        tipp_risky = tipp_risky.where(~mask, tipp_portfolio_value) #no
short sale
        tipp_riskfree = tipp_portfolio_value - tipp_risky

    tipp_results = tipp_portfolio_value
    return tipp_results

#SP (sp_floor_prct = % of the floor)

def SP(sp_floor_prct):
    sp_floor = inv_amount * sp_floor_prct
    sp_t = 0
    sp_riskfree = pd.DataFrame([np.full(num_paths,0)])
    sp_risky = inv_amount - (inv_amount * sp_floor_prct)
    sp_risky = np.full(num_paths, sp_risky)
    sp_risky = pd.DataFrame([sp_risky])
    sp_riskfree = inv_amount - sp_risky
    while sp_t <= T-1:
        sp_t += 1
        sp_risky = sp_risky * Sim.loc[sp_t]
        if sp_t == 63 or sp_t == 126 or sp_t == 189 or sp_t == T:
            sp_riskfree = sp_riskfree * (1 + riskfree_return)

        sp_portfolio_value = sp_risky + sp_riskfree
        # rebalancing of strategy
        time_to_maturity = (T - sp_t)/252
        prices_row = Sim_prices.loc[sp_t]
        sigma = Sim_returns.std()

        d1 = (np.log(prices_row/sp_floor) + (riskfree_return + 0.5 *
        sigma**2) * time_to_maturity) / sigma *
        np.sqrt(time_to_maturity)
        d2 = d1 - sigma * np.sqrt(time_to_maturity)

        value = sp_floor * np.exp(-riskfree_return * time_to_maturity) *
        norm.cdf(-d2) - prices_row * norm.cdf(-d1)

        K = (sp_floor_prct * sp_portfolio_value) * (prices_row + value)

        sp_risky_prct = (prices_row * norm.cdf(d1)) / (prices_row *
       (norm.cdf(d1)) + K * (np.exp(-riskfree_return * time_to_maturity))
       * norm.cdf(sigma * np.sqrt(time_to_maturity)-d2))

        sp_riskfree_prct = 1 - sp_risky_prct
```

```
        sp_risky = sp_portfolio_value * sp_risky_prct
        sp_riskfree = sp_portfolio_value * sp_riskfree_prct
        sp_portfolio_value = sp_risky + sp_riskfree
    return sp_portfolio_value


########################################################################
'''
Functions for measures
SD, Omega, Sharpe, VaR
'''

def stochastic_dominance(ecdf_a, ecdf_b):
    ecdf_a = ecdf_a.transpose()
    ecdf_b = ecdf_b.transpose()
    ecdf_a = pd.Series(pd.Series(ecdf_a[0]).rank(pct=True).values,
name='ecdf')
    ecdf_b = pd.Series(pd.Series(ecdf_b[0]).rank(pct=True).values,
name='ecdf')
    # First-order stochastic dominance
    if (ecdf_a >= ecdf_b).all() and (ecdf_a > ecdf_b).any():
        order = 'A dominates B FSD'
        prob = (ecdf_a - ecdf_b).sum()
    elif (ecdf_b >= ecdf_a).all() and (ecdf_b > ecdf_a).any():
        order = 'B dominates A FSD'
        prob = (ecdf_b - ecdf_a).sum()
    # Second-order stochastic dominance
    elif (ecdf_a >= ecdf_b).all() and (ecdf_a == ecdf_b).all():
        order = 'A weakly dominates B SSD'
        prob = 0
    elif (ecdf_b >= ecdf_a).all() and (ecdf_a == ecdf_b).all():
        order = 'B weakly dominates A SSD'
        prob = 0
    # Third-order stochastic dominance
    else:
        p_ab = (ecdf_a - ecdf_b) * (ecdf_a > ecdf_b)
        p_ba = (ecdf_b - ecdf_a) * (ecdf_b > ecdf_a)
        prob = (p_ab.sum() + p_ba.sum()) / 2
        if prob > 0:
            order = 'A strictly dominates B TSD'
        elif prob < 0:
            order = 'B strictly dominates A TSD'
        else:
            order = 'no stochastic dominance'

    return order, prob

def omega_ratio(returns):
    returns = returns.transpose()
    threshold = inv_amount*(1+riskfree_return)**4
    positive_returns = returns[returns > threshold]
    positive_returns = positive_returns.dropna()
    negative_returns = returns[returns < threshold]
    negative_returns = negative_returns.dropna()
    p = len(positive_returns) / len(returns)
    n = len(negative_returns) / len(returns)
    omega = p / n
    return omega

def sharpe_ratio(returns):
```

```
    returns = returns.transpose()
    returns = (returns - inv_amount)/100
    calc = returns.mean()/returns.std()
    return calc

def var(returns, confidence_level=0.95):
    returns = returns.transpose()
    VaR = returns.quantile(1 - confidence_level)
    return VaR

########################################################################
'''
Functions for expected utility
exponential and quadratic
'''

def quadratic_eu(returns, A):
    returns = returns.transpose()
    avg = ((returns.mean())-100)/100
    stdv = returns.std()
    calc = avg-0.5*A*stdv**2
    return calc

def exponential_eu(returns, alpha):
    returns = returns.transpose()
    avg = ((returns.mean())-100)/100
    calc = np.exp(avg*(-alpha))
    return calc

########################################################################
bh_1 = BH(0)
bh_2 = BH(0.3)
cm_3 = CM(0.3)
sl_4 = SL(1)
cppi_5 = CPPI(1,1)
cppi_6 = CPPI(1,3)
cppi_7 = CPPI(1,5)
tipp_8 = TIPP(1,1)
tipp_9 = TIPP(1,3)
tipp_10 = TIPP(1,5)
SP_11 = SP(1)

# plot for strategy payoff function
gg = Sim_prices.loc[252] #NASDAQ at time T
fig, ax = plt.subplots()
ax.scatter(gg, cppi_5, label='CPPI, m1', color = 'green', s=10)
ax.scatter(gg, cppi_6, label='CPPI, m3', color = 'black', s=10)
ax.scatter(gg, cppi_7, label='CPPI, m5', color = 'blue',s=10)
ax.scatter(gg, bh_2, label='Bh (70/30)', color = 'yellow',s=10)
ax.set_xlabel('Cena NASDAQ v čase T')
ax.set_ylabel('Hodnota portfólia v čase T')
ax.set_title('Monte Carlo simulácia CPPI a BH stratégie, σ=30%, Floor
100%')
ax.axvline(x=11196.2, color='red', linestyle=':', label = 'NASDAQ
$t_{0}$')
ax.axhline(y=100, color = 'red', linestyle = '-', label = 'Floor 100%')
#ax.set_ylim(65,300)
ax.legend()
plt.show()

end_time = time.time()
```

```
elapsed_time = end_time - start_time
print(f"elapsed_time is {elapsed_time} seconds")
```