```
'''
Author:      Weiyi Chen
Copyright:   Copyright (C) 2015 Baruch College, Modeling and Market Making in
    Forex Exchange
Description: This programming question will try to determine whether using a
    factor-based approach to reducing
dimensionality is better than an ad hoc method. The result will show that setting
    hedge notionals based on the
true factor shocks should provide a better hedge performance than based on the ad
    hoc triangle shocks.
Test:        hedger_test.py
'''

from numpy import *
from lazy import lazy

class Hedger(object):
    """
    In the toy market, we assume that we know the dynamics of the asset currency
        interest rate:

        dQ=σ_1 e^(-β_1 T) dz_1+σ_2 e^(-β_2 T) dz_2
        E[dz_1 dz_2 ]=ρ dt

    where σ_1=1%/sqrt(yr), σ_2=0.8%/sqrt(yr), β_1=0.5/yr, β_2=0.1/yr, and ρ=-0.4.
    """
    def __init__(self):
        super(Hedger, self).__init__()

    @lazy
    def Spot(self):
        ''' We start by assuming a toy market: spot = 1 '''
        return 1.

    @lazy
    def Q(self):
        ''' asset currency interest rate curve = Q(T) = flat at 3% '''
        return .03

    @lazy
    def R(self):
        ''' denominated currency interest rate curve = R(T) = flat at 0% '''
        return 0.

    @lazy
    def Sigma1(self):
        ''' vol of the 1st parameter '''
        return .01

    @lazy
    def Sigma2(self):
        ''' vol of the 2nd parameter '''
        return .008

    @lazy
    def Beta1(self):
```

```
        ''' mean reversion of 1st parameter '''
        return .5

    @lazy
    def Beta2(self):
        ''' mean reversion of 1st parameter '''
        return .1

    @lazy
    def Rho(self):
        ''' correlation between two brownian motions '''
        return -.4

    @lazy
    def T1(self):
        '''
        1st benchmark date, we will use forwards to settlement date to hedge the
            forward rate risk (or equivalently,
        the risk to moves in the asset currency interest rate) of our portfolio.
        '''
        return .25

    @lazy
    def T2(self):
        '''
        2nd benchmark date, we will use forwards to settlement date to hedge the
            forward rate risk (or equivalently,
        the risk to moves in the asset currency interest rate) of our portfolio.
        '''
        return 1.

    @lazy
    def Dt(self):
        '''
        the length of the time step the simulation advances over

        Simulate the portfolio forward a time dt=0.001y. That will result in the
            asset-currency rates moving according
        to the factor model described above, which shocks the benchmark rates for
            tenors T1 and T2, and for the portfolio's
        risk tenor T.
        '''
        return 1e-3

    @lazy
    def SqrtDt(self):
        ''' square root of dt '''
        return sqrt(self.Dt)

    @lazy
    def Nruns(self):
        ''' the number of Monte Carlo runs '''
        return 1e5

    @lazy
    def Tenor(self):
```

```
        '''
        The portfolio to hedge has one position: a unit asset-currency notional
            of a forward contract settling at time T.
        You'll try this for values of T in [0.1,0.25,0.5,0.75,1,2] to see how
            performance changes for portfolios with
        risk to different tenors.
        '''
        return .1

    @lazy
    def HedgingStrategy(self):
        '''
        You will try three different hedging strategies:

        Value 1: one where you choose the hedge notionals (of forwards settling
            at times T1 and T2) based on the triangle
        shock we discussed in class (though as there are only two benchmarks here
            , the T1 shock will be flat before T1 and
        the T2 shock will be flat after T2);
        Value 2: one where the notionals are set to hedge the actual two shocks
            from the factors described above;
        Value 3: lastly, one where you don't hedge at all.
        '''
        return 0

    @lazy
    def Dz1s(self):
        ''' first brownian motion '''
        return random.normal(0, self.SqrtDt, self.Nruns)

    @lazy
    def Dz2s(self):
        ''' second browian motion '''
        return self.Rho * self.Dz1s + sqrt(1 - self.Rho ** 2) * random.normal(0,
            self.SqrtDt, self.Nruns)

    @lazy
    def DQTs(self):
        ''' the dynamics of the asset currency interest rate '''
        return self.Sigma1 * exp(-self.Beta1 * self.Tenor) * self.Dz1s + self.
            Sigma2 * exp(-self.Beta2 * self.Tenor) * self.Dz2s

    @lazy
    def DQ1s(self):
        ''' rate shocks for T1 '''
        return self.Sigma1 * exp(-self.Beta1 * self.T1) * self.Dz1s + self.Sigma2
            * exp(-self.Beta2 * self.T1) * self.Dz2s

    @lazy
    def DQ2s(self):
        ''' rate shocks for T2 '''
        return self.Sigma1 * exp(-self.Beta1 * self.T2) * self.Dz1s + self.Sigma2
            * exp(-self.Beta2 * self.T2) * self.Dz2s

    @lazy
    def DQT_dQ1(self):
```

```
        ''' the resulting shock to Q(T); i.e. dQ(T)/dQ1 '''
        dz1 = -1./self.Sigma1 * exp(self.Beta1*self.T2-self.Beta2*(self.T2-self.
            T1)) / (1.-exp((self.Beta1-self.Beta2)*(self.T2-self.T1)))
        dz2 =  1./self.Sigma2 * exp(self.Beta2*self.T1) / (1.-exp((self.Beta1-
            self.Beta2)*(self.T2-self.T1)))
        return self.Sigma1 * exp(-self.Beta1*self.Tenor)*dz1 + self.Sigma2 * exp
            (-self.Beta2*self.Tenor)*dz2

    @lazy
    def DQT_dQ2(self):
        ''' second shock, i.e. dQ(T)/dQ2 '''
        dz1 = -1./self.Sigma1 * exp(self.Beta1*self.T1+self.Beta2*(self.T2-self.
            T1)) / (1.-exp((self.Beta2-self.Beta1)*(self.T2-self.T1)))
        dz2 =  1./self.Sigma2 * exp(self.Beta2*self.T2) / (1.-exp((self.Beta2-
            self.Beta1)*(self.T2-self.T1)))
        return self.Sigma1 * exp(-self.Beta1*self.Tenor)*dz1 + self.Sigma2*exp(-
            self.Beta2*self.Tenor)*dz2

    @lazy
    def HedgingNotional1(self):
        '''
        Add in the hedges: two forwards, settling at times T1 and T2, with
            notionals set to hedge the portfolio (either
        against the two triangle shocks or against the two factor shocks).
        '''
        if self.HedgingStrategy == 0:
            return 0.
        elif self.HedgingStrategy == 1:
            if self.Tenor <= self.T1:
                return self.Tenor / self.T1 * exp(-self.Q*(self.Tenor-self.T1))
            elif self.Tenor >= self.T2:
                return 0.
            else:
                return (self.T2-self.Tenor) / (self.T2-self.T1) * self.Tenor /
                    self.T1 * exp(-self.Q*(self.Tenor-self.T1))
        elif self.HedgingStrategy == 2:
            return self.DQT_dQ1 * self.Tenor / self.T1 * exp(-self.Q*(self.Tenor-
                self.T1))
        else:
            raise TypeError('Hedging Strategy can only be 0, 1 or 2')

    @lazy
    def HedgingNotional2(self):
        '''
        Add in the hedges: two forwards, settling at times T1 and T2, with
            notionals set to hedge the portfolio (either
        against the two triangle shocks or against the two factor shocks).
        '''
        if self.HedgingStrategy == 0:
            return 0.
        elif self.HedgingStrategy == 1:
            if self.Tenor <= self.T1:
                return 0.
            elif self.Tenor >= self.T2:
                return self.Tenor/self.T2*exp(-self.Q*(self.Tenor-self.T2))
            else:
```

```
            return (self.Tenor-self.T1)/(self.T2-self.T1)*self.Tenor/self.T2*
                exp(self.Q*(self.T2-self.Tenor))
        elif self.HedgingStrategy == 2:
            return self.DQT_dQ2 * self.Tenor / self.T2 * exp(self.Q*(self.T2-self
                .Tenor))
        else:
            raise TypeError('Hedging Strategy can only be 0, 1 or 2')

    @lazy
    def PNLs(self):
        ''' Construct the PNL distributions for the three hedging approaches.
            Determine the PNL realized. '''
        pnls = self.Spot * (exp(-(self.Q+self.DQTs)*self.Tenor) - exp(-self.Q*
            self.Tenor))
        pnls -= self.HedgingNotional1 * self.Spot * (exp(-(self.Q+self.DQ1s)*self
            .T1) - exp(-self.Q*self.T1))
        pnls -= self.HedgingNotional2 * self.Spot * (exp(-(self.Q+self.DQ2s)*self
            .T2) - exp(-self.Q*self.T2))
        return pnls

    @lazy
    def PNL_std(self):
        ''' standard deviation of simulation PNL '''
        return self.PNLs.std()
```