```
import abc

from numpy import array, identity, linalg, matrix, random
import numpy as np
import random as rn

class SampleCreator(object):
    '''
    Abstract base class for a simulator of one path each for an ensemble of ra
ndom variables (eg several Brownian motions W1, W2, etc)
    Attributes:
        size (int): the number of random variables
    '''
    # indicate that this is an abstract base class (abc)
    __metaclass__ = abc.ABCMeta

    def __init__(self, size):
        self.size = size

    @abc.abstractmethod
    def create_sample(self, n_samples=1, time_step=1, *args):
        '''
        Generate a single path for each random variable (number of random vari
ables, eg number of Brownian motions, is self.size). One path consists of n_sa
mples steps
        Args:
            n_samples (int): the number of steps along a single path
            time_step (float): the size of a mini time step
        Returns:
            ndarray: a 2D array (num rows: self.size, num cols: n_samples). Ea
ch row is one single path for one random variable
        '''

class SimpleGaussianSampleCreator(SampleCreator):

    '''
    Creates IID samples distributed in Gaussian fashion. Does not use numpy
    because numpy seeding does not play well with multiprocessing
    '''

    def create_sample(self, n_samples=1, time_step=1, *args):
        '''
        Generate a single path for each random variable (number of random vari
ables, eg number of Brownian motions, is self.size). One path consists of n_sa
mples steps
        Args:
            n_samples (int): the number of steps along a single path
            time_step (float): the size of a mini time step. For Brownian moti
on, time_step is variance of the normal dist
        Returns:
            ndarray: a 2D array (num rows: self.size, num cols: n_samples). Ea
ch row is one single path for one random variable
        '''
```

```
        # each sample has a mean = 0 and variance = time_step
        sigma = time_step ** 0.5
        return array([
            array([rn.gauss(0, sigma) for _ in xrange(n_samples)])
            for _ in xrange(self.size)
        ])

class IIDSampleCreator(SampleCreator):
    '''
    Simulator of one path each for an ensemble of independent random variables
    Attributes:
        size (int): the number of independent random variables (eg the number
of independent Brownian motions W1, W2, etc)
        distro (method of numpy.random): the distribution from which to take o
ne step and form one random path
    '''
    def __init__(self, size, distro=random.normal):
        self.distro = distro
        super(IIDSampleCreator, self).__init__(size)

    def create_sample(self, n_samples=1, time_step=1, *args):
        '''
        Generate a single path for each random variable (number of random vari
ables, eg number of Brownian motions, is self.size). One path consists of n_sa
mples steps
        Args:
            n_samples (int): the number of steps along a single path
            time_step (float): the size of a mini time step. For Brownian moti
on, time_step is variance of the normal dist
        Returns:
            ndarray: a 2D array (num rows: self.size, num cols: n_samples). Ea
ch row is one single path for one random variable
        '''
        # each sample has a mean = 0 and variance = time_step
        return array([
            self.distro(scale=time_step**0.5, size=n_samples)
            for _ in xrange(self.size)
        ])


class CorrelatedSampleCreator(IIDSampleCreator):
    '''
    Simulator of one path each for an ensemble of correlated random variables
(eg correlated Brownian motions). When the variables together take one time st
ep, their steps are correlated by a given correlation matrix
    Attributes:
        size (int): the number of correlated random variables (eg the number o
f correlated Brownian motions W1, W2, etc)
        distro (method of numpy.random): the distribution from which to take o
ne step and form one random path
        _corr_matrix (ndarray): correlation matrix of the steps being taken al
ong correlated paths by the random variables
        _ts_transforms (dict): a dict mapping one time_step to a transformatio
```

```
n matrix C where C * C.T = the covariance matrix corresponding to the size of
that time_step
    '''

    def __init__(self, corr_matrix, scales=None, distro=random.normal):
        self._set_corr_matrix(corr_matrix)
        size = len(corr_matrix) # len(ndarray) returns num of rows
        self._ts_transforms = {}

        super(CorrelatedSampleCreator, self).__init__(size=size,
                                                      distro=distro)

    def _set_corr_matrix(self, cm):
        '''
        Set the corr matrix of self to cm after checking conditions
        Args:
            cm (ndarray): a symmetrix positive definite correlation matrix
        Returns:
            Set self._corr_matrix to cm
        '''
        r, c = cm.shape

        if r != c:
            raise ValueError('Correlation matrix %s is not square' % cm)

        for i in xrange(r):
            for j in xrange(i+1):
                if cm.item(i, j) != cm.item(j, i):
                    raise ValueError('Correlation matrix %s is not symmetric'
% cm)

        if not np.all(linalg.eigvalsh(cm) > 0):
            raise ValueError('Correlation matrix %s is not positive definite'
% cm)

        self._corr_matrix = cm

    def create_sample(self, n_samples=1, time_step=1, *args):
        '''
        Generate a single path for each random variable (number of random vari
ables, eg number of Brownian motions, is self.size). One path consists of n_sa
mples steps
        Args:
            n_samples (int): the number of steps along a single path
            time_step (float): the size of a mini time step. For Brownian moti
on, time_step is variance of the normal dist
        Returns:
            ndarray: a 2D array (num rows: self.size, num cols: n_samples). Ea
ch row is one single path for one random variable. The steps are correlated by
 self._corr_matrix
        '''
        # try to see if a transform matrix C is already made for this time_ste
p
```

```
        transform = self._ts_transforms.get(time_step)
        # if there is no matrix C yet then create one for this time_step
        if transform is None:
            transform = self._create_transform(time_step)
            self._ts_transforms[time_step] = transform

        # first create a set of independent paths for the group of variables
        # these iid samples must be standard normal before transforming
        # here iid is n_samples columns of Z, where Z is a column of std norma
l
        iid = super(CorrelatedSampleCreator, self).create_sample(n_samples, ti
me_step=1)
        # matrix transform is C, and Y = C*Z so covar(Y) = C*C.T
        # matrix.A returns self as ndarray
        return (transform * matrix(iid)).A

    def _create_transform(self, time_step):
        '''
        From a time_step dt, calculate covar matrix and the transform matrix C
 where C * C.T = the target covar matrix
        Args:
            time_step: variance of each step of the Brownian motion
        Returns:
            matrix: the transformation matrix C
        '''
        # first create a diag matrix where the diag is sqrt(dt)
        ts = time_step ** 0.5
        I = matrix(ts*identity(self.size))

        # make the covar matrix based on dt and corr matrix
        covar = I * self._corr_matrix * I

        # if Z is a column of std normal independent variables, then Y = C*Z
        # then covar(Y) = C * C.T
        # here we return the transform matrix C
        return matrix(linalg.cholesky(covar))
```

```
import abc
import itertools
import math

class Stock(object):
    '''
    Abstract base class for a stock object
    Attributes:
        spot (float): the current spot price
        post_walk_price (float): the price at T after walking one full path
    '''

    __metaclass__ = abc.ABCMeta

    def __init__(self, spot):
        self.spot = spot
        self.post_walk_price = spot

    @abc.abstractmethod
    def find_volatilities(self, time_step, vol_steps):
        '''
        Return the volatility values over multiple time steps
        This is a generator function that returns a generator of vol
        Args:
            time_step (float): size of a mini time step (dt)
            vol_steps (list): a 1D iterable of the Brownian increments driving
 stochastic variance in the Heston model
        Returns:
            generator: a generator to generate vol on the fly, over time steps
        '''

    def walk_price(self, risk_free, time_step, price_steps, vol_steps=None):
        '''
        Simulate the stock to walk one full path over multiple steps using geo
metric Brownian motion
        Args:
            risk_free (float): the rate in the deterministic term of dS
            time_step (float): size of a mini time step (dt)
            price_steps (list): a 1D iterable of the Brownian increments drivi
ng the diffusion term in dS
            vol_steps (list): a 1D iterable of the Brownian increments driving
 stochastic variance in the Heston model. If not provided, default behavior is
 to use the price_steps.

        Returns:
            float: the price S(T) after walking one full simulated path.
        '''
        vol_steps = price_steps if vol_steps is None else vol_steps
        vols = self.find_volatilities(time_step, vol_steps)

        lprice = math.log(self.post_walk_price)

        # for dW, sigma in itertools.izip(price_steps, vols):
```

```
        #     t1 = (risk_free - (0.5 * sigma**2)) * time_step
        #     t2 = sigma * dW
        #     lprice += (t1 + t2)
        # self.post_walk_price = math.exp(lprice)

        price = self.post_walk_price
        for dW, sigma in itertools.izip(price_steps, vols):
            price += time_step*risk_free*price + sigma*price*dW

        self.post_walk_price = price
        return self.post_walk_price


class ConstantVolatilityStock(Stock):
    '''
    A stock with constant volatility
    Attributes:
        spot (float): the current spot price
        post_walk_price (float): the price at T after walking one full path
        vol (float): the constant vol in the diffusion term
    '''

    def __init__(self, spot, vol):
        super(ConstantVolatilityStock, self).__init__(spot)
        self.vol = vol

    def find_volatilities(self, time_step, vol_steps):
        '''
        Return the volatility values over multiple time steps
        This is a generator function that returns a Generator of vol
        Args:
            time_step (float): size of a mini time step
            vol_steps (list): here the vol is constant so vol_steps contain th
e marks of time steps that vol will undertake
        Returns:
            generator: returns the stock's volatility n times, where n is the
length of vol_steps
        '''
        return itertools.repeat(self.vol, len(vol_steps))


class VariableVolatilityStock(Stock):
    '''
    A stock with stochastic volatility based on Heston model
    Attributes:
        spot (float): the current spot price
        post_walk_price (float): the price at T after walking one full path
        _base_vol (float): the starting point of the vol at t = 0. The square
of _base_vol is the starting variance V(0)
        _kappa (float): the mean reversion speed in Heston SDE for dV(t)
        _theta (float): the mean reversion level in Heston SDE for dV(t)
        _gamma (float): the constant diffusion term in Heston SDE for dV(t)
    '''
```

```
    def __init__(self, spot, base_vol, kappa, theta, gamma):
        super(VariableVolatilityStock, self).__init__(spot)

        self.vol = base_vol
        self._base_vol = base_vol
        self._kappa = kappa
        self._theta = theta
        self._gamma = gamma

    def find_volatilities(self, time_step, vol_steps):
        '''
        Return the volatility values over multiple time steps
        This is a generator function that returns a Generator of vol
        Args:
            time_step (float): size of a mini time step
            vol_steps (list): a 1D iterable of the Brownian increments driving
 stochastic variance in the Heston model
        Returns:
            generator: a random walk of the volatility using the full truncati
on method. Thus, if we are left in a situation where the next step would lead
us to a negative variance, the step instead goes to zero.
        '''
        volatility = max(0, self.vol)
        variance = self._base_vol**2

        for dZ in vol_steps:
            drift = self._kappa * (self._theta - variance) * time_step
            diffusion = self._gamma * volatility * dZ
            variance = variance + drift + diffusion
            volatility = max(0, variance) ** 0.5
            self.vol = volatility
            yield volatility
```

```
import copy
import numpy

import mlmc.random_numbers as random

def create_simple_path(stocks,
                       risk_free,
                       T,
                       n_steps,
                       rng_creator=None,
                       chunk_size=100000):
    '''
    Get the post walk price of each of the inputted stocks
    Each stock walks one path to final time T
    The post walk price is returned without changing the stock itself
    This path simulation is vanilla Monte Carlo for Euler Maruyama
    To address memory issues, the simulation is run in chunks.
    Args:
        stocks (iterable): list of stocks that will walk
        risk_free (float): risk free rate driving stock drift
        T (float): the final time at end of a walk
        n_steps (long): number of steps along one path
        rng_creator (callable): a no-arg function that will return an
            object implementing the SampleCreator interface. Default
            is a function that returns an IIDSampleCreator that outputs
            both dW and dZ.
        chunk_size (long): a walk of n_steps is done in chunks to address
            memory issues; chunk_size is the size of one chunk

    Returns:
        list: the post walk price of each stock
    '''

    stocks = [copy.deepcopy(s) for s in stocks]
    rng = rng_creator() if rng_creator else random.SimpleGaussianSampleCreator
(2*len(stocks))

    dt = float(T) / n_steps

    chunks = [chunk_size for _ in xrange(n_steps/chunk_size)]
    chunks.append(n_steps % chunk_size)

    for c in chunks:
        if not c:
            continue

        samples = rng.create_sample(n_samples=c, time_step=dt)
        interval = rng.size / len(stocks)

        for i, s in enumerate(stocks):
            # samples[i*interval] are the dW
            # samples[i*interval + 1] are the dZ
            s.walk_price(risk_free,
```

```
                         dt,
                         *samples[i*interval:(i+1)*interval])

    return [s.post_walk_price for s in stocks]


def create_layer_path(stocks,
                      risk_free,
                      T,
                      n_steps,
                      rng_creator=None,
                      chunk_size=100000,
                      K=2):
    '''
    Each stock walks one path in the finer level, and one path on the coarser
level, to final time T
    The post walk price is returned without changing the stock itself
    This path simulation is MultiLevel Monte Carlo for Euler Maruyama
    To address memory issues, the simulation is run in chunks.
    Args:
        stocks (iterable): list of stocks that will walk
        risk_free (float): risk free rate driving stock drift
        T (float): the final time at end of a walk
        n_steps (long): number of steps along one path
        rng_creator (callable): a no-arg function that will return an
            object implementing the SampleCreator interface. Default
            is a function that returns an IIDSampleCreator that outputs
            both dW and dZ.
        chunk_size (long): a walk of n_steps is done in chunks to address
            memory issues; chunk_size is the size of one chunk
        K (int): for level L in MLMC, the interval [0,T] is partitioned into K
**L intermediate time steps

    Returns:
        list: the post walk price of each stock
    '''

    stocks = [
        (copy.deepcopy(s), copy.deepcopy(s))
        for s in stocks
    ]

    rng = rng_creator() if rng_creator else random.SimpleGaussianSampleCreator
(2*len(stocks))

    # dt is for the coarser level, dt_sub for finer level
    dt = float(T) / n_steps
    dt_sub = dt / K

    chunks = [chunk_size for _ in xrange(n_steps/chunk_size)]
    chunks.append(n_steps % chunk_size)

    for c in chunks:
```

```
        if not c:
            continue

        # first, samples := dW and dZ are the finer level
        samples = rng.create_sample(n_samples=c*K,
                                    time_step=dt_sub)
        interval = rng.size / len(stocks)

        # s1 walks the coarse level, s2 walks the fine level
        for i, (s1, s2) in enumerate(stocks):
            subs = samples[i*interval:(i+1)*interval] # finer level
            fulls = numpy.array([
                numpy.array([s[i:i+K].sum() for i in xrange(0, c*K, K)])
                for s in subs
            ]) # coarser level

            s1.walk_price(risk_free, dt, *fulls)
            s2.walk_price(risk_free, dt_sub, *subs)


    return [
        (s1.post_walk_price, s2.post_walk_price)
        for s1, s2 in stocks
    ]


def calculate(task):
    return task[0](*task[1:])


def main():
    import multiprocessing
    from mlmc.stock import ConstantVolatilityStock
    pool = multiprocessing.Pool(4)
    stock = ConstantVolatilityStock(10, 0.1)

    x = pool.map(
        calculate,
        [
            [create_simple_path] + [[stock], 0.01, 1, 100]
            for _ in xrange(100)
        ]
    )

    import pprint
    pprint.pprint(sorted(xx[0] for xx in x))


if __name__ == '__main__':
    main()
```

```
from __future__ import division

import abc
import collections
import datetime
import functools
import itertools
import math
import numpy as np
import scipy.stats as ss

from mlmc import path, stock

class Option(object):

    __metaclass__ = abc.ABCMeta

    ''' A general representation of an option. '''

    def __init__(self, assets, risk_free, expiry, is_call):
        '''
        assets: list of underlying assets. Will probably be stocks in this fra
mework.
        risk_free: the risk-free interest rate
        expiry: days until expiration of the option
        is_call: boolean. Whether the option is a call option or a put option
        '''
        self.assets = assets
        self.risk_free = risk_free
        self.expiry = expiry
        self.is_call = is_call

    @abc.abstractmethod
    def determine_payoff(self, *args, **kwargs):
        ''' Figure out the valuation of the option '''


class EuropeanStockOption(Option):

    ''' A stock option with a European payout '''

    def __init__(self, assets, risk_free, expiry, is_call, strike):
        '''
        assets: list of underlying assets. Will probably be stocks in this fra
mework.
        risk_free: the risk-free interest rate
        expiry: days until expiration of the option
        is_call: boolean. Whether the option is a call option or a put option
        strike: the strike price of the option
        '''
        if isinstance(assets, collections.Iterable):
            assets = assets[:1]
            if not isinstance(assets[0], stock.Stock):
```

```
                raise TypeError("Requires an underlying stock")
        elif isinstance(assets, stock.Stock):
            assets = [assets]
        else:
            raise TypeError("Requires an underlying stock")

        super(EuropeanStockOption, self).__init__(assets, risk_free, expiry, i
s_call)
        self.strike = strike

    def determine_payoff(self, final_spot, *args, **kwargs):
        v1, v2 = (final_spot, self.strike) if self.is_call else (self.strike,
final_spot)
        return max(v1 - v2, 0)


class EuropeanSwaption(Option):

    ''' An exchange option with a European payout.'''

    def __init__(self, assets, risk_free, expiry, is_call):
        '''
        assets: list of underlying assets. Will probably be stocks in this fra
mework.
        risk_free: the risk-free interest rate
        expiry: days until expiration of the option
        is_call: boolean. Whether the option is a call option or a put option
        '''

        if len(assets) != 2:
            raise ValueError('Requires two underlying assets')

        super(EuropeanSwaption, self).__init__(assets, risk_free, expiry, is_c
all)

    def determine_payoff(self, s1_final_spot, s2_final_spot, *args, **kwargs):
        v1, v2 = (s1_final_spot, s2_final_spot) if self.is_call else (s2_final
_spot, s1_final_spot)
        return max(v1 - v2, 0)


class OptionSolver(object):

    ''' Given an option, will solve for the 'correct' price of the option '''

    __metaclass__ = abc.ABCMeta

    @abc.abstractmethod
    def solve_option_price(self, option, return_stats=False):
        '''
        Actually solve the option price.
        option: an Option object. May need to be a specific type of option
        return_stats: boolean. Return not only the option price, but also asso
```

```
ciate statistics
        '''


class AnalyticEuropeanStockOptionSolver(OptionSolver):

    ''' A Black-Scholes stock option pricer. Only works for European stock opt
ions '''

    def solve_option_price(self, option):
        '''
        Actually solve the option price.
        option: an Option object. May need to be a specific type of option
        '''
        underlying = option.assets[0]
        spot = underlying.spot
        vol = underlying.vol
        risk_free = option.risk_free
        expiry = option.expiry
        strike = option.strike

        log_diff = math.log(spot / strike)
        vt = 0.5 * vol**2
        denom = vol * math.sqrt(expiry)

        d1 = (log_diff + (risk_free + vt)*expiry) / denom
        d2 = (log_diff + (risk_free - vt)*expiry) / denom

        discount = math.exp(-risk_free * expiry)

        if option.is_call:
            S, d1, K, d2 = spot, d1, -strike, d2
        else:
            S, d1, K, d2 = -spot, -d1, strike, -d2

        return S * ss.norm.cdf(d1) + K * ss.norm.cdf(d2) * discount


class StatTracker(object):

    ''' Keeps track of running means and variances '''

    def __init__(self, discount):
        '''
        discount: the discount value that we will use to weigh all stats.
        '''
        self.discount = discount
        self.count = 0
        self.total = 0
        self.sum_of_squares = 0
        self.initial_val = None

    @property
```

```
    def variance(self):
        '''
        The running variance of the samples so far added
        '''
        if self.count in (0, 1):
            return float('inf')

        square_of_sum = self.total**2 / self.count
        variance = (self.sum_of_squares - square_of_sum) / (self.count - 1)
        return (self.discount * variance)

    @property
    def stdev(self):
        '''
        The running standard deviation of the samples so far added
        '''
        if self.count in (0, 1):
            return float('inf')

        return self.variance ** 0.5

    @property
    def mean(self):
        '''
        The running arithmetic mean of the samples so far added
        '''
        if self.count == 0:
            return float('nan')

        return self.discount * (self.total + self.initial_val*self.count) / se
lf.count

    def add_sample(self, s):
        '''
        Add a sample to our set of samples for use in the
        running statistics
        '''
        if self.initial_val is None:
            self.initial_val = s

        self.count += 1
        diff = s - self.initial_val
        self.total += diff
        self.sum_of_squares += diff**2

    def get_interval_length(self, z_score):
        '''
        Determine the size of the confidence interval given a specific z-score
        z_score: float. The number of standard deviations away from the sample
 mean
                 we expect the population mean to fall into. 1.96 for 95% conf
idence.
        '''
```

```
        if self.count == 0:
            return float('inf')

        return self.stdev * self.count**(-0.5) * z_score


class NaiveMCOptionSolver(OptionSolver):

    '''
    Solve an option price using a simple Monte Carlo strategy
    of continued Euler-Maruyama paths until the resultant
    mean has an associated confidence interval shorter
    than the max_interval_length
    '''

    def __init__(self,
                 max_interval_length,
                 confidence_level=0.95,
                 rng_creator=None,
                 n_steps=None):
        '''
        max_interval_length: float. The longest the confidence interval may be
        confidence_level: float. The % chance the population mean falls within
                          the calculated confidence interval
        rng_creator: fn. No-arg function that returns a SampleCreator object
        n_steps: int. Number of steps per Euler-Maruyama path. Defaults to
                 option expiry normalized by the max_interval_length
        '''
        self.max_interval_length = max_interval_length
        self.confidence_level = confidence_level
        self.rng_creator = rng_creator
        self.n_steps = n_steps

    @property
    def confidence_level(self):
        ''' The confidence level of the solver '''
        return self._confidence_level

    @confidence_level.setter
    def confidence_level(self, value):
        self._confidence_level = value
        self._z_score = ss.norm.ppf(1 - 0.5*(1-self.confidence_level))

    @property
    def z_score(self):
        ''' The z score associated with the confidence level '''
        return self._z_score

    def _simulate_paths(self, option, n_steps, discount):
        stat_tracker = StatTracker(discount)
        cnt = itertools.count()

        while next(cnt) < 10 or stat_tracker.get_interval_length(self.z_score)
```

```
 > self.max_interval_length:
            result = path.create_simple_path(option.assets,
                                             option.risk_free,
                                             option.expiry,
                                             n_steps,
                                             self.rng_creator)
            payoff = option.determine_payoff(*result)
            stat_tracker.add_sample(payoff)
        return stat_tracker

    def solve_option_price(self, option, return_stats=False):
        '''
        Actually solve the option price.
        option: an Option object. May need to be a specific type of option
        return_stats: boolean. Return not only the option price, but also asso
ciate statistics
        '''
        expiry = option.expiry
        risk_free = option.risk_free
        discount = math.exp(-risk_free * expiry)

        n_steps = self.n_steps or int(math.floor(expiry / self.max_interval_le
ngth))

        tracker = self._simulate_paths(option, n_steps, discount)

        if return_stats:
            return tracker.mean, tracker.stdev, tracker.count, n_steps
        else:
            return tracker.mean


class LayeredMCOptionSolver(OptionSolver):

    '''
    Solve option price using a multi-level Monte Carlo (MLMC)
    strategy. There are multiple ways of doing this
    '''

    @abc.abstractmethod
    def run_levels(self, option, discount):
        '''
        Run the multiple levels of E-M paths

        option: Option. What we're looking to price
        discount: float. Discount factor of money in the future.
        '''

    def run_bottom_level(self, option, steps):
        '''
        Run the bottom level of E-M paths. Each of the E-M paths
        will have only one step
```

```
        option: Option. What we're looking to price
        steps: int. Totally irrelevant, used only because run_upper_levels
               requires it as part of the signature.
        '''
        result = path.create_simple_path(option.assets,
                                         option.risk_free,
                                         option.expiry,
                                         1,
                                         self.rng_creator)
        return option.determine_payoff(*result),

    def run_upper_level(self, option, steps):
        '''
        Run a non-bottom level of E-M paths. This comes out to
        running two paths, one with K times as many steps as the other

        option: Option. What we're looking to price
        steps: int. the number of steps for the path with fewer steps.
        '''
        result = path.create_layer_path(option.assets,
                                        option.risk_free,
                                        option.expiry,
                                        steps,
                                        self.rng_creator,
                                        K=self.level_scaling_factor)
        coarse, fine = zip(*result)
        payoff_coarse = option.determine_payoff(*coarse)
        payoff_fine = option.determine_payoff(*fine)

        return (payoff_fine - payoff_coarse),

    def run_level(self, option, L, n, *trackers):
        '''
        Run an individual level

        option: Option. What we're looking to price.
        L: int. The level we wish to run.
        n: int. The number of times we wish to run the level.
        *trackers: iterable of StatTrackers. Will be used to keep track of
                   price, and possibly also time to run path.
        '''
        if L == 0:
            fn = self.run_bottom_level
            steps = 1
        else:
            fn = self.run_upper_level
            steps = self.level_scaling_factor ** (L - 1)
        for _ in xrange(n):
            for s, t in zip(fn(option, steps), trackers):
                t.add_sample(s)

    def solve_option_price(self, option, return_stats=False):
        '''
```

```
        Actually solve the option price.
        option: an Option object. May need to be a specific type of option
        return_stats: boolean. Return not only the option price, but also asso
ciate statistics
        '''
        expiry = option.expiry
        risk_free = option.risk_free
        discount = math.exp(-risk_free * expiry)
        trackers = self.run_levels(option, discount)

        if return_stats:
            means = [t.mean for t in trackers]
            variances = [t.variance for t in trackers]
            counts = [t.count for t in trackers]
            price = sum([t.mean for t in trackers])
            return (price, means, variances, counts)

        else:
            return sum([t.mean for t in trackers])


class SimpleLayeredMCOptionSolver(LayeredMCOptionSolver):

    '''
    The MLMC strategy should use a simple system for determining
    whether or not to continue, based on the empirical size of the
    highest level in comparison to the second-highest level. The number
    of paths run should simply be initially assumed as the same for all
    levels
    '''

    def __init__(self,
                 max_interval_length,
                 level_scaling_factor=4,
                 base_steps=1000,
                 rng_creator=None,
                 min_L=3):
        '''
        max_interval_length: float. Size of the error of the price
        level_scaling_factor: int. Ratio of steps in level l+1 to steps in lev
el l
        base_steps: int. Number of paths to initially run per level
        rng_creator: no-arg function returning SampleCreator
        min_L: int. Starting number of levels to run
        '''
        self.max_interval_length = max_interval_length
        self.level_scaling_factor = max(level_scaling_factor, 2)
        self.base_steps = base_steps
        self.rng_creator = rng_creator
        self.min_L = min_L

    def _determine_additional_steps(self, option, trackers):
        find_dt = lambda l:  option.expiry / (self.level_scaling_factor ** l)
```

```
        tot = 2 * (self.max_interval_length ** -2) * sum(
            (t.variance/find_dt(L)) ** 0.5
            for L, t in enumerate(trackers)
        )
        ideal_ns = (
            tot * (t.variance * find_dt(L)) ** 0.5
            for L, t in enumerate(trackers)
        )
        return [
            int(math.ceil(max((ideal_n - t.count), 0)))
            for ideal_n, t in zip(ideal_ns, trackers)
        ]

    def _is_error_too_high(self, trackers):
        t1, t2 = trackers[-2:]

        empirical = max(abs(t1.mean) / self.level_scaling_factor, abs(t2.mean)
)
        estimated = ((self.level_scaling_factor - 1) * self.max_interval_lengt
h / (2**0.5))
        return estimated < empirical

    def run_levels(self, option, discount):
        trackers = [
            (self.base_steps, StatTracker(discount))
            for _ in xrange(self.min_L)
        ]

        while sum(n for n, _ in trackers) > 0:
            for L, (n, t) in enumerate(trackers):
                self.run_level(option, L, n, t)

            addl_steps = self._determine_additional_steps(
                option,
                [x[1] for x in trackers]
                )

            nt = []
            for L, (addl, (n, t)) in enumerate(zip(addl_steps, trackers)):
                self.run_level(option, L, addl, t)
                nt.append((0, t))

            trackers = nt

            if self._is_error_too_high([x[1] for x in trackers]):
                trackers.append((self.base_steps, StatTracker(discount)))

        return [t[1] for t in trackers]


class HeuristicLayeredMCOptionSolver(LayeredMCOptionSolver):

    '''
```

```
    The MLMC strategy should use a system for determining
    whether or not to continue based on the empirically-determined
    decay factors of the size of the layer means, layer variances and
    layer costs.
    '''

    def __init__(self,
                 target_mse,
                 rng_creator=None,
                 initial_n_levels=3,
                 level_scaling_factor=4,
                 initial_n_paths=5000,
                 alpha=None,
                 beta=None,
                 gamma=None):
        '''
        target_mse: float. Target mean standard error
        rng_cretor: no-arg function returning SampleCreator
        initial_n_levels: int. Number of levels to run initially. Must be >2
        level_scaling_factor: int. Ratio of steps in level l+1 to steps in lev
el l
        initial_n_paths: int. Number of paths to run initially on the base lev
el
        alpha: float. decay factor of the means of the level
        beta: float. decay factor of the variances of the level
        gamma: float. growth factor of the cost of the level
        '''
        self.target_mse = target_mse
        self.rng_creator = rng_creator
        self.initial_n_levels = max(initial_n_levels, 3)
        self.level_scaling_factor = max(level_scaling_factor, 2)
        self.initial_n_paths = initial_n_paths

        self._alpha = alpha
        self._beta = beta
        self._gamma = gamma

    def cost_determined(fn):
        @functools.wraps(fn)
        def wrapper(self, *args, **kwargs):
            d1 = datetime.datetime.now()
            res = fn(self, *args, **kwargs)
            d2 = datetime.datetime.now()

            delta = d2 - d1
            delta = delta.seconds + delta.microseconds*1e-6
            return delta, res[0]

        return wrapper

    run_bottom_level = cost_determined(LayeredMCOptionSolver.run_bottom_level)
    run_upper_level = cost_determined(LayeredMCOptionSolver.run_upper_level)
```

```
    def _determine_additional_n_values(self, trackers):
        overall = int(math.ceil(sum(
            (p.variance * c.mean)**0.5
            for _, p, c in trackers
        ) / (self.target_mse**2)))

        return [
            max(0, int(math.ceil(overall * (p.variance * c.mean)**0.5)) - p.co
unt)
            for _, p, c in trackers
        ]

    def _find_coefficients(self, payoff_trackers, cost_trackers):
        A = np.array([[i, 1] for i, _ in enumerate(payoff_trackers, 1)])

        if self._alpha:
            alpha = self._alpha
        else:
            x = np.array([[np.log2(p.mean)] for p in payoff_trackers])
            alpha = max(0.5, -np.linalg.lstsq(A, x)[0][0])

        if self._beta:
            beta = self._beta
        else:
            x = np.array([[np.log2(p.variance)] for p in payoff_trackers])
            beta = max(0.5, -np.linalg.lstsq(A, x)[0][0])

        if self._gamma:
            gamma = self._gamma
        else:
            x = np.array([[np.log2(p.mean)] for p in cost_trackers])
            gamma = np.linalg.lstsq(A, x)[0][0]

        return alpha, beta, gamma

    def run_levels(self, option, discount):
        n_levels = self.initial_n_levels
        trackers = [
            (self.initial_n_paths, StatTracker(discount), StatTracker(1))
            for _ in xrange(n_levels)
        ]

        while sum(n for n, _, _ in trackers):
            for i, (n, payoff_tracker, cost_tracker) in enumerate(trackers):
                self.run_level(option, i, n, cost_tracker, payoff_tracker)

            addl_n_values = self._determine_additional_n_values(trackers)
            alpha, beta, gamma = self._find_coefficients(*zip(*((p, c) for (_,
 p, c) in trackers[1:])))

            trackers = [
                (addl_n, p, c)
                for addl_n, (_, p, c) in
```

```
                itertools.izip(addl_n_values, trackers)
            ]

            if all(n <= 0.01*p.count for n, p, _ in trackers):
                remaining_error = max(
                    (t.mean * 2**(alpha*i)) / (2**alpha - 1)
                    for i, (_, t, _) in enumerate(trackers[-2:], start=-2)
                )

                if remaining_error > (0.5**0.5) * self.target_mse:
                    guess_v = trackers[-1][1].variance  / (2^beta)
                    guess_c = trackers[-1][2].mean * (2 ** gamma)
                    term = (guess_v/guess_c) ** 0.5

                    base = sum((t.variance/c.cost)**0.5 for _, t, c in tracker
s)
                    base += term
                    guess_n = 2 * term * base / (self.target_mse**2)
                    trackers.append((guess_n, StatTracker(discount), StatTrack
er(1)))

        return [p for _, p, _ in trackers]
```