Portfolio optimization

Portfolio allocation vector

In this example we show how to do portfolio optimization using CVXPY. We begin with the basic definitions. In portfolio optimization we have some amount of money to invest in any of n different assets. We choose what fraction w_i of our money to invest in each asset i, $i = 1, \ldots, n$.

We call $w \in \mathbf{R}^n$ the *portfolio allocation vector*. We of course have the constraint that $\mathbf{1}^T w = 1$. The allocation $w_i < 0$ means a *short position* in asset i, or that we borrow shares to sell now that we must replace later. The allocation $w \geq 0$ is a *long only* portfolio. The quantity

$$\|w\|_1 = \mathbf{1}^T w_+ + \mathbf{1}^T w_-$$

is known as *leverage*.

Asset returns

We will only model investments held for one period. The initial prices are $p_i > 0$. The end of period prices are $p_i^+ > 0$. The asset (fractional) returns are $r_i = (p_i^+ - p_i)/p_i$. The porfolio (fractional) return is $R = r^T w$.

A common model is that r is a random variable with mean $\mathbf{E}r = \mu$ and covariance $\mathbf{E}(\mathbf{r} - \mu)(\mathbf{r} - \mu)^\mathbf{T} = \Sigma$. It follows that R is a random variable with $\mathbf{E}R = \mu^T w$ and $\mathbf{var}(R) = w^T \Sigma w$. $\mathbf{E}R$ is the (mean) *return* of the portfolio. $\mathbf{var}(R)$ is the *risk* of the portfolio. (Risk is also sometimes given as $\mathbf{std}(R) = \sqrt{\mathbf{var}(R)}$.)

Portfolio optimization has two competing objectives: high return and low risk.

Classical (Markowitz) portfolio optimization

Classical (Markowitz) portfolio optimization solves the optimization problem

$$\begin{array}{ll} \text{maximize} & \mu^T w - \gamma w^T \Sigma w \\ \text{subject to} & \mathbf{1}^T w = 1, \quad w \in \mathcal{W}, \end{array}$$

where $w \in \mathbf{R}^n$ is the optimization variable, $\mathcal{W}$ is a set of allowed portfolios (e.g., $\mathcal{W} = \mathbf{R}^n_+$ for a long only portfolio), and $\gamma > 0$ is the *risk aversion parameter*.

The objective $\mu^T w - \gamma w^T \Sigma w$ is the *risk-adjusted return*. Varying γ gives the optimal *risk-return trade-off*. We can get the same risk-return trade-off by fixing return and minimizing risk.

Example

In the following code we compute and plot the optimal risk-return trade-off for 10 assets, restricting ourselves to a long only portfolio.

```
# Generate data for long only portfolio optimization.
import numpy as np
import scipy.sparse as sp
np.random.seed(1)
n = 10
mu = np.abs(np.random.randn(n, 1))
Sigma = np.random.randn(n, n)
Sigma = Sigma.T.dot(Sigma)
```

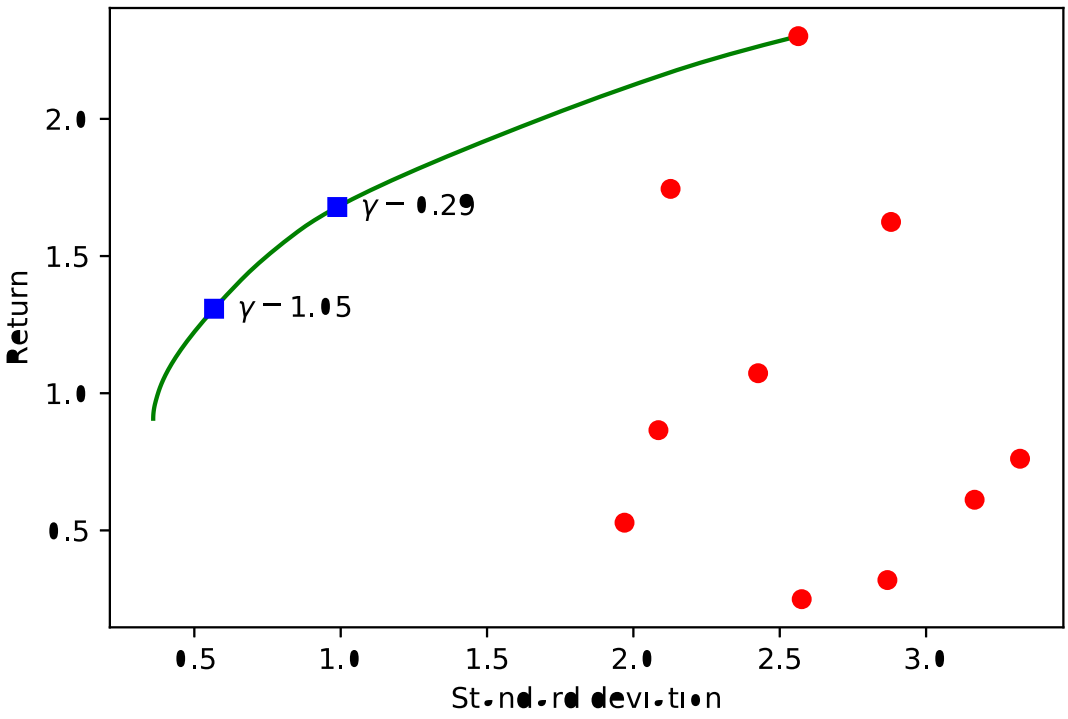
```
# Long only portfolio optimization.
import cvxpy as cp
```

```
w = cp.Variable(n)
gamma = cp.Parameter(nonneg=True)
ret = mu.T@w
risk = cp.quad_form(w, Sigma)
prob = cp.Problem(cp.Maximize(ret - gamma*risk),
                  [cp.sum(w) == 1,
                   w >= 0])
```

```
# Compute trade-off curve.
SAMPLES = 100
risk_data = np.zeros(SAMPLES)
ret_data = np.zeros(SAMPLES)
gamma_vals = np.logspace(-2, 3, num=SAMPLES)
for i in range(SAMPLES):
    gamma.value = gamma_vals[i]
    prob.solve()
    risk_data[i] = cp.sqrt(risk).value
    ret_data[i] = ret.value
```

```
# Plot long only trade-off curve.
import matplotlib.pyplot as plt
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

markers_on = [29, 40]
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(risk_data, ret_data, 'g-')
for marker in markers_on:
    plt.plot(risk_data[marker], ret_data[marker], 'bs')
    ax.annotate(r"$\gamma = %.2f$" % gamma_vals[marker], xy=(risk_data[marker]+.
for i in range(n):
    plt.plot(cp.sqrt(Sigma[i,i]).value, mu[i], 'ro')
plt.xlabel('Standard deviation')
plt.ylabel('Return')
plt.show()
```



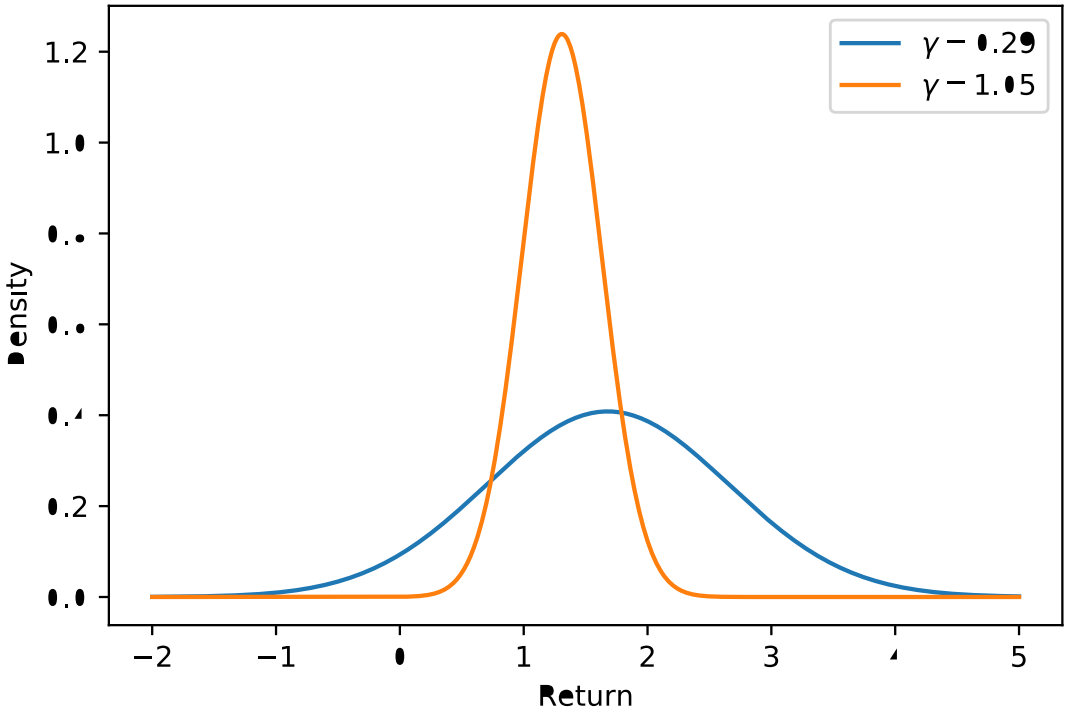

```
'''
```

We plot below the return distributions for the two risk aversion values marked on the trade-off curve. Notice that the probability of a loss is near 0 for the low risk value and far above 0 for the high risk value.

```
# Plot return distributions for two points on the trade-off curve.
import scipy.stats as spstats


plt.figure()
for midx, idx in enumerate(markers_on):
    gamma.value = gamma_vals[idx]
    prob.solve()
    x = np.linspace(-2, 5, 1000)
    plt.plot(x, spstats.norm.pdf(x, ret.value, risk.value), label=r"$\gamma = %.

plt.xlabel('Return')
plt.ylabel('Density')
plt.legend(loc='upper right')
plt.show()
```




```
'''
```

Portfolio constraints

There are many other possible portfolio constraints besides the long only constraint. With no constraint ($\mathcal{W} = \mathbf{R}^n$), the optimization problem has a simple analytical solution. We will look in detail at a *leverage limit*, or the constraint that $\|w\|_1 \leq L^{\max}$.

Another interesting constraint is the *market neutral* constraint $m^T \Sigma w = 0$, where m_i is the capitalization of asset i. $M = m^T r$ is the *market return*, and $m^T \Sigma w = \mathbf{cov}(M, R)$. The market neutral constraint ensures that the portfolio return is uncorrelated with the market return.

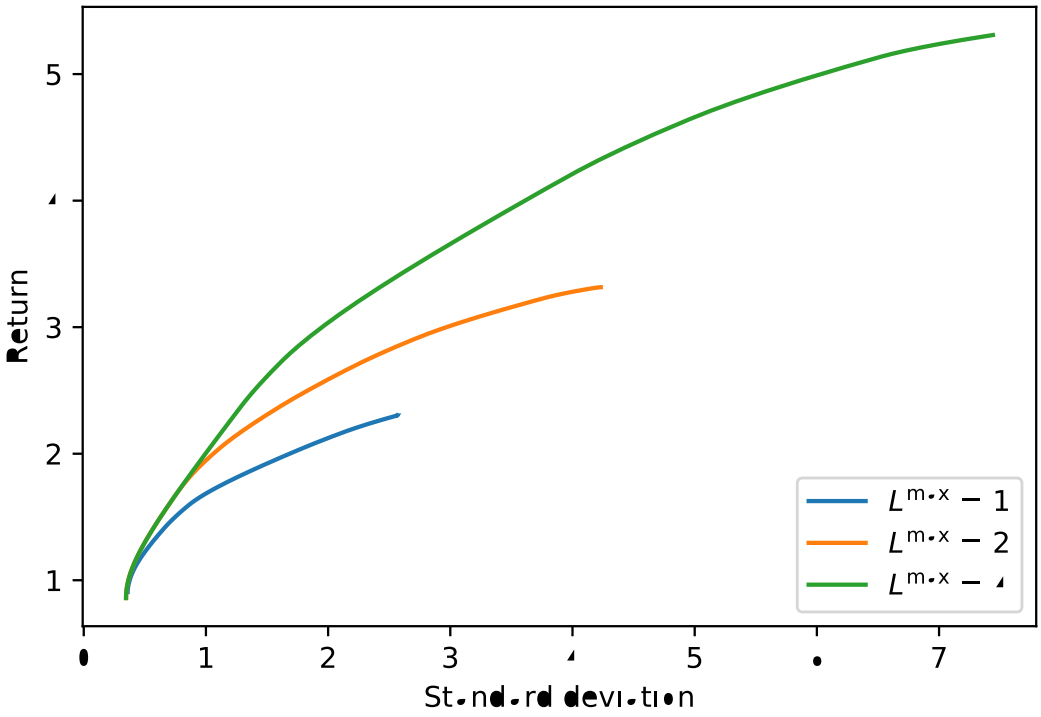
Example

In the following code we compute and plot optimal risk-return trade-off curves for leverage limits of 1, 2, and 4. Notice that more leverage increases returns and allows greater risk.

```
# Portfolio optimization with leverage limit.
Lmax = cp.Parameter()
prob = cp.Problem(cp.Maximize(ret - gamma*risk),
                  [cp.sum(w) == 1,
                   cp.norm(w, 1) <= Lmax])


# Compute trade-off curve for each leverage limit.
L_vals = [1, 2, 4]
SAMPLES = 100
risk_data = np.zeros((len(L_vals), SAMPLES))
ret_data = np.zeros((len(L_vals), SAMPLES))
gamma_vals = np.logspace(-2, 3, num=SAMPLES)
w_vals = []
for k, L_val in enumerate(L_vals):
    for i in range(SAMPLES):
        Lmax.value = L_val
        gamma.value = gamma_vals[i]
        prob.solve(solver=cp.SCS)
        risk_data[k, i] = cp.sqrt(risk).value
        ret_data[k, i] = ret.value
```

```
# Plot trade-off curves for each leverage limit.
for idx, L_val in enumerate(L_vals):
    plt.plot(risk_data[idx,:], ret_data[idx,:], label=r"$L^{\max}$ = %d" % L_val
for w_val in w_vals:
    w.value = w_val
    plt.plot(cp.sqrt(risk).value, ret.value, 'bs')
plt.xlabel('Standard deviation')
plt.ylabel('Return')
plt.legend(loc='lower right')
plt.show()
```



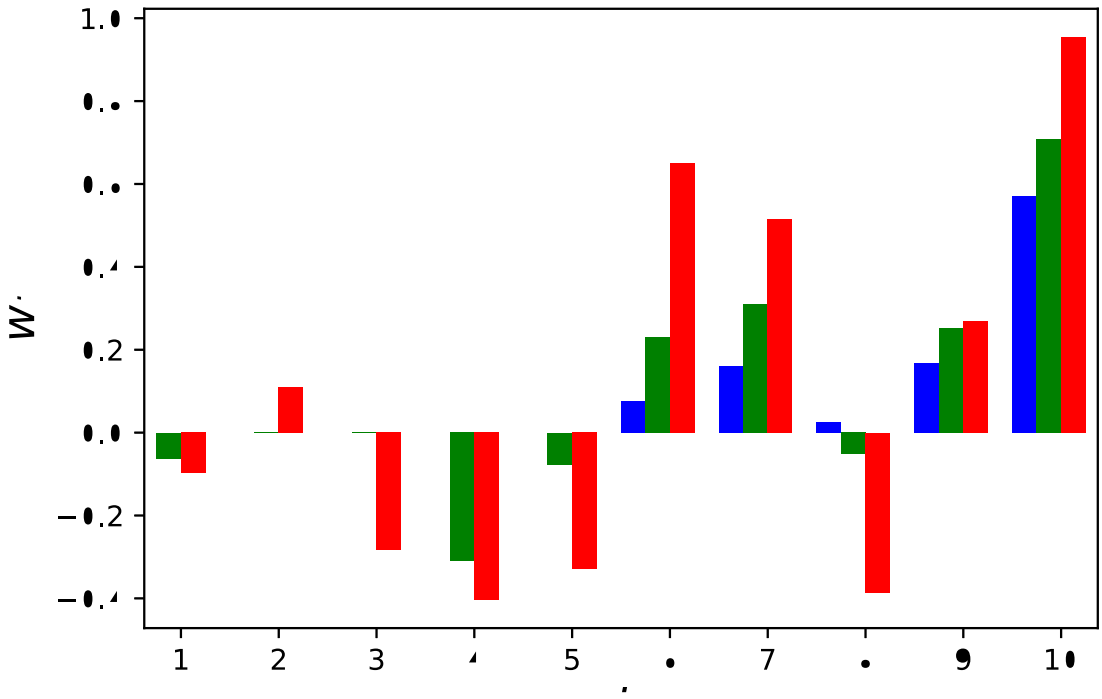

```
'''
```

We next examine the points on each trade-off curve where $w^T \Sigma w = 2$. We plot the amount of each asset held in each portfolio as bar graphs. (Negative holdings indicate a short position.) Notice that some assets are held in a long position for the low leverage portfolio but in a short position in the higher leverage portfolios.

```
# Portfolio optimization with a leverage limit and a bound on risk.
prob = cp.Problem(cp.Maximize(ret),
                  [cp.sum(w) == 1,
                   cp.norm(w, 1) <= Lmax,
                   risk <= 2])
```

```
# Compute solution for different leverage limits.
for k, L_val in enumerate(L_vals):
    Lmax.value = L_val
    prob.solve()
    w_vals.append( w.value )


# Plot bar graph of holdings for different leverage limits.
colors = ['b', 'g', 'r']
indices = np.argsort(mu.flatten())
for idx, L_val in enumerate(L_vals):
     plt.bar(np.arange(1,n+1) + 0.25*idx - 0.375, w_vals[idx][indices], color=co
             label=r"$L^{\max}$ = %d" % L_val, width = 0.25)
plt.ylabel(r"$w_i$", fontsize=16)
plt.xlabel(r"$i$", fontsize=16)
plt.xlim([1-0.375, 10+.375])
plt.xticks(np.arange(1,n+1))
plt.show()
```




```
'''
```

Variations

There are many more variations of classical portfolio optimization. We might require that $\mu^T w \geq R^{\min}$ and minimize $w^T \Sigma w$ or $\|\Sigma^{1/2} w\|_2$. We could include the (broker) cost of short positions as the penalty $s^T (w)_-$ for some $s \geq 0$. We could include transaction costs (from a previous portfolio w^{prev}) as the penalty

$$\kappa^T |w - w^{\text{prev}}|^\eta, \quad \kappa \geq 0.$$

Common values of η are $\eta = 1,\ 3/2,\ 2$.

Factor covariance model

A particularly common and useful variation is to model the covariance matrix Σ as a factor model

$$\Sigma = F \tilde{\Sigma} F^T + D,$$

where $F \in \mathbf{R}^{n \times k}$, $k \ll n$ is the *factor loading matrix*. k is the number of factors (or sectors) (typically 10s). F_{ij} is the loading of asset i to factor j. D is a diagonal matrix; $D_{ii} > 0$ is the *idiosyncratic risk*. $\tilde{\Sigma} > 0$ is the *factor covariance matrix*.

$F^T w \in \mathbf{R}^k$ gives the portfolio *factor exposures*. A portfolio is *factor j neutral* if $(F^T w)_j = 0$.

Portfolio optimization with factor covariance model

Using the factor covariance model, we frame the portfolio optimization problem as

$$\begin{array}{ll} \text{maximize} & \mu^T w - \gamma \left(f^T \tilde{\Sigma} f + w^T D w \right) \\ \text{subject to} & \mathbf{1}^T w = 1, \quad f = F^T w \\ & w \in \mathcal{W}, \quad f \in \mathcal{F}, \end{array}$$

where the variables are the allocations $w \in \mathbf{R}^n$ and factor exposures $f \in \mathbf{R}^k$ and $\mathcal{F}$ gives the factor exposure constraints.

Using the factor covariance model in the optimization problem has a computational advantage. The solve time is $O(nk^2)$ versus $O(n^3)$ for the standard problem.

Example

In the following code we generate and solve a portfolio optimization problem with 50 factors and 3000 assets. We set the leverage limit $= 2$ and $\gamma = 0.1$.

We solve the problem both with the covariance given as a single matrix and as a factor model. Using CVXPY with the OSQP solver running in a single thread, the solve time was 173.30 seconds for the single matrix formulation and 0.85 seconds for the factor model formulation. We collected the timings on a MacBook Air with an Intel Core i7 processor.

```
# Generate data for factor model.
n = 3000
m = 50
np.random.seed(1)
mu = np.abs(np.random.randn(n, 1))
Sigma_tilde = np.random.randn(m, m)
Sigma_tilde = Sigma_tilde.T.dot(Sigma_tilde)
D = sp.diags(np.random.uniform(0, 0.9, size=n))
F = np.random.randn(n, m)
```

```
# Factor model portfolio optimization.
w = cp.Variable(n)
f = cp.Variable(m)
gamma = cp.Parameter(nonneg=True)
Lmax = cp.Parameter()
ret = mu.T@w
risk = cp.quad_form(f, Sigma_tilde) + cp.sum_squares(np.sqrt(D) @ w)
prob_factor = cp.Problem(cp.Maximize(ret - gamma*risk),
                    [cp.sum(w) == 1,
                     f == F.T@w,
                     cp.norm(w, 1) <= Lmax])

# Solve the factor model problem.
Lmax.value = 2
gamma.value = 0.1
prob_factor.solve(verbose=True)
```

```
-----------------------------------------------------------------
           OSQP v0.5.0  -  Operator Splitting QP Solver
              (c) Bartolomeo Stellato,  Goran Banjac
        University of Oxford  -  Stanford University 2018
-----------------------------------------------------------------
problem:  variables n = 6050, constraints m = 6052
          nnz(P) + nnz(A) = 172325
settings: linear system solver = qdldl,
          eps_abs = 1.0e-04, eps_rel = 1.0e-04,
          eps_prim_inf = 1.0e-04, eps_dual_inf = 1.0e-04,
          rho = 1.00e-01 (adaptive),
          sigma = 1.00e-06, alpha = 1.60, max_iter = 10000
          check_termination: on (interval 25),
          scaling: on, scaled_termination: off
          warm start: on, polish: on

iter   objective    pri res    dua res    rho        time
   1  -2.1359e+03   7.63e+00   3.73e+02   1.00e-01   8.41e-02s
 200  -4.1947e+00   1.61e-03   7.86e-03   3.60e-01   3.24e-01s
 400  -4.6288e+00   3.02e-04   6.01e-04   3.60e-01   6.26e-01s
 500  -4.6491e+00   2.49e-04   4.66e-04   3.60e-01   7.69e-01s

status:               solved
solution polish:      unsuccessful
number of iterations: 500
optimal objective:    -4.6491
run time:             8.46e-01s
optimal rho estimate: 3.31e-01

4.649145900085359
```

```
# Standard portfolio optimization with data from factor model.
risk = cp.quad_form(w, F.dot(Sigma_tilde).dot(F.T) + D)
```

```
prob = cp.Problem(cp.Maximize(ret - gamma*risk),
                  [cp.sum(w) == 1,
                   cp.norm(w, 1) <= Lmax])

# Uncomment to solve the problem.
# WARNING: this will take many minutes to run.
prob.solve(verbose=True)
```

```
-----------------------------------------------------------------
           OSQP v0.5.0  -  Operator Splitting QP Solver
              (c) Bartolomeo Stellato,  Goran Banjac
        University of Oxford  -  Stanford University 2018
-----------------------------------------------------------------
problem:  variables n = 6000, constraints m = 6002
          nnz(P) + nnz(A) = 4519500
settings: linear system solver = qdldl,
          eps_abs = 1.0e-04, eps_rel = 1.0e-04,
          eps_prim_inf = 1.0e-04, eps_dual_inf = 1.0e-04,
          rho = 1.00e-01 (adaptive),
          sigma = 1.00e-06, alpha = 1.60, max_iter = 10000
          check_termination: on (interval 25),
          scaling: on, scaled_termination: off
          warm start: on, polish: on

iter   objective    pri res    dua res    rho        time
   1  -1.1774e+04   2.65e+02   1.51e+04   1.00e-01   7.43e+00s
 200  -4.1080e+02   2.42e-01   8.86e-04   1.00e-01   1.03e+01s
 400  -1.9413e+02   1.13e-01   2.51e-04   1.00e-01   1.30e+01s
 600  -1.2345e+02   6.40e-02   1.09e-04   1.00e-01   1.59e+01s
 800  -8.7560e+01   4.67e-02   5.29e-05   1.00e-01   1.90e+01s
1000  -6.5202e+01   3.49e-02   2.99e-05   1.00e-01   2.20e+01s
1200  -5.0118e+01   2.68e-02   1.91e-05   1.00e-01   2.52e+01s
1400  -3.9737e+01   2.09e-02   1.41e-05   1.00e-01   2.81e+01s
1600  -3.2445e+01   1.72e-02   1.06e-05   1.00e-01   3.10e+01s
1800  -2.6947e+01   1.42e-02   8.27e-06   1.00e-01   3.41e+01s
2000  -2.2700e+01   1.17e-02   6.57e-06   1.00e-01   3.72e+01s
2200  -1.9294e+01   9.74e-03   5.29e-06   1.00e-01   4.06e+01s
2400  -1.6616e+01   8.26e-03   4.32e-06   1.00e-01   4.41e+01s
2600  -1.4460e+01   7.01e-03   3.56e-06   1.00e-01   4.69e+01s
2800  -1.2704e+01   5.95e-03   2.93e-06   1.00e-01   4.99e+01s
3000  -1.1267e+01   5.06e-03   2.43e-06   1.00e-01   5.41e+01s
3200  -1.0092e+01   4.25e-03   2.00e-06   1.00e-01   5.73e+01s
3400  -9.1244e+00   3.58e-03   1.66e-06   1.00e-01   6.14e+01s
3600  -8.3286e+00   3.04e-03   1.38e-06   1.00e-01   6.50e+01s
3800  -7.6760e+00   2.60e-03   1.14e-06   1.00e-01   6.84e+01s
4000  -7.1409e+00   2.26e-03   9.40e-07   1.00e-01   7.18e+01s
4200  -6.7000e+00   2.04e-03   7.81e-07   1.00e-01   7.53e+01s
4400  -6.3366e+00   1.85e-03   6.50e-07   1.00e-01   7.87e+01s
4600  -6.0382e+00   1.69e-03   5.41e-07   1.00e-01   8.31e+01s
4800  -5.7969e+00   1.58e-03   4.54e-07   1.00e-01   8.65e+01s
5000  -5.5953e+00   1.46e-03   3.83e-07   1.00e-01   8.97e+01s
5200  -5.4277e+00   1.37e-03   3.24e-07   1.00e-01   9.30e+01s
5400  -5.2885e+00   1.28e-03   2.73e-07   1.00e-01   9.62e+01s
```

```
5600  -5.1729e+00   1.20e-03   2.30e-07   1.00e-01   1.00e+02s
5800  -5.0768e+00   1.13e-03   1.94e-07   1.00e-01   1.04e+02s
6000  -4.9968e+00   1.08e-03   1.63e-07   1.00e-01   1.07e+02s
6200  -4.9301e+00   1.02e-03   1.37e-07   1.00e-01   1.10e+02s
6400  -4.8746e+00   9.80e-04   1.18e-07   1.00e-01   1.14e+02s
6600  -4.8281e+00   9.40e-04   1.09e-07   1.00e-01   1.18e+02s
6800  -4.7893e+00   9.04e-04   1.01e-07   1.00e-01   1.22e+02s
7000  -4.7568e+00   8.72e-04   9.40e-08   1.00e-01   1.25e+02s
7200  -4.7295e+00   8.44e-04   8.75e-08   1.00e-01   1.28e+02s
7400  -4.7372e+00   8.63e-04   2.54e-07   1.00e-01   1.32e+02s
7600  -4.7339e+00   8.57e-04   1.41e-07   1.00e-01   1.35e+02s
7800  -4.7278e+00   8.25e-04   8.93e-08   1.00e-01   1.38e+02s
8000  -4.7195e+00   7.99e-04   5.47e-08   1.00e-01   1.41e+02s
8200  -4.7100e+00   7.75e-04   4.25e-08   1.00e-01   1.45e+02s
8400  -4.7002e+00   7.59e-04   3.67e-08   1.00e-01   1.48e+02s
```

```
print('Factor model solve time = {}'.format(prob_factor.solver_stats.solve_time)
print('Single model solve time = {}'.format(prob.solver_stats.solve_time))
```

```
Factor model solve time = 0.845742191
Single model solve time = 173.303122558
```