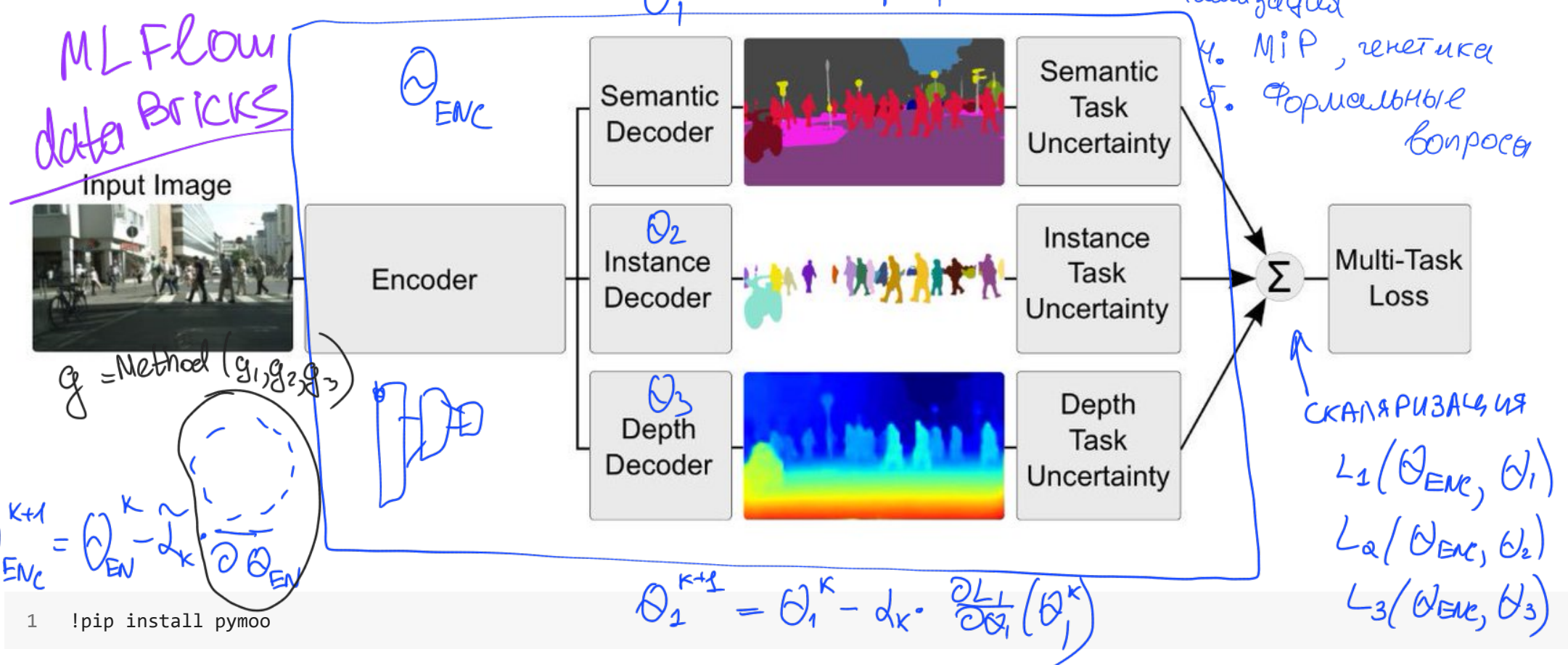
Multi task learning

[Multi-Task Learning as Multi-Objective Optimization](#)



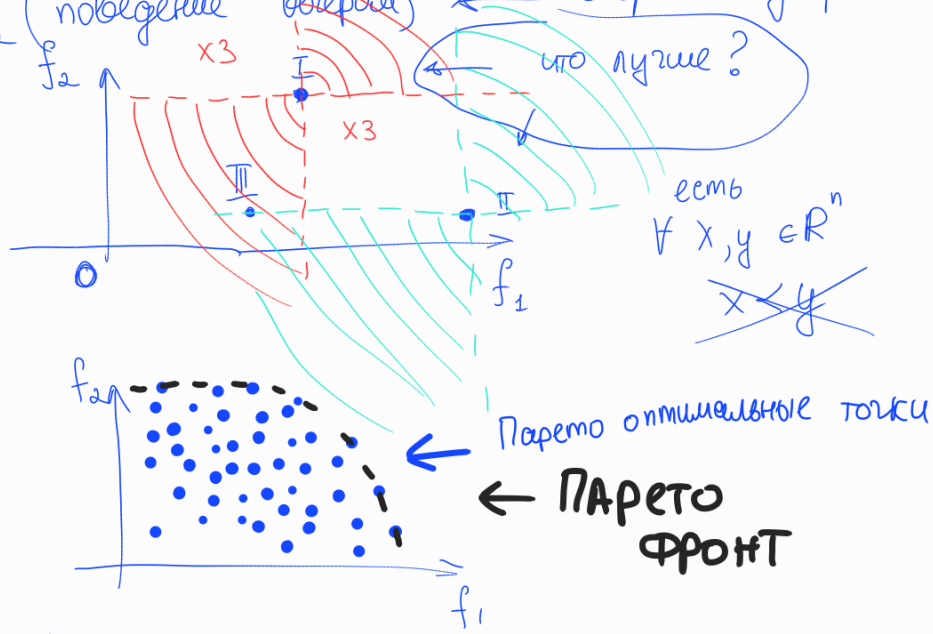

```
!pip install pymoo
```

```
Collecting pymoo
  Downloading https://files.pythonhosted.org/packages/07/11/0591960f255d55325e516c2babe3586385dc67c4fae074d75498f535a703/pymoo-0.4.2.1.ta
     |████████████████████████████████| 3.7MB 5.1MB/s
Requirement already satisfied: numpy>=1.15 in /usr/local/lib/python3.6/dist-packages (from pymoo) (1.19.5)
Requirement already satisfied: scipy>=1.1 in /usr/local/lib/python3.6/dist-packages (from pymoo) (1.4.1)
Requirement already satisfied: matplotlib>=3 in /usr/local/lib/python3.6/dist-packages (from pymoo) (3.2.2)
Requirement already satisfied: autograd>=1.3 in /usr/local/lib/python3.6/dist-packages (from pymoo) (1.3)
Collecting cma==2.7
  Downloading https://files.pythonhosted.org/packages/b9/3b/87a4efbcfeaf3172d81ef843f0b0c34c3ba60ec884aa6777f34f68b57418/cma-2.7.0-py2.py
     |████████████████████████████████| 245kB 30.2MB/s
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/lib/python3.6/dist-packages (from matplotlib>=3->py
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.6/dist-packages (from matplotlib>=3->pymoo) (2.8.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.6/dist-packages (from matplotlib>=3->pymoo) (0.10.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.6/dist-packages (from matplotlib>=3->pymoo) (1.3.1)
Requirement already satisfied: future>=0.15.2 in /usr/local/lib/python3.6/dist-packages (from autograd>=1.3->pymoo) (0.16.0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.6/dist-packages (from python-dateutil>=2.1->matplotlib>=3->pymoo) (1.15
Building wheels for collected packages: pymoo
  Building wheel for pymoo (setup.py) ... done
  Created wheel for pymoo: filename=pymoo-0.4.2.1-cp36-cp36m-linux_x86_64.whl size=1344733 sha256=425b1c87818ec5d7a09ba61e381d56ba7bb5eac
  Stored in directory: /root/.cache/pip/wheels/24/1a/15/c95ef6c978305899c8b374fd3a36dd494a46a63b2b32c6c5c5
Successfully built pymoo
Installing collected packages: cma, pymoo
Successfully installed cma-2.7.0 pymoo-0.4.2.1
```

```
from pymoo.algorithms.nsga2 import NSGA2
from pymoo.factory import get_problem
from pymoo.optimize import minimize
from pymoo.visualization.scatter import Scatter

problem = get_problem("bnh")

algorithm = NSGA2(pop_size=100)

res = minimize(problem,
               algorithm,
               ('n_gen', 200),
               seed=1,
               verbose=False)

plot = Scatter()
plot.add(problem.pareto_front(), plot_type="line", color="black", alpha=0.7)
plot.add(res.F, color="red")
plot.show()
```

f_1 (поведение вечером) ← весёлость вечера → max

f_2 (поведение вечером) ← спорт по утрам → max



Парето оптимальные точки

← ПАРЕТО ФРОНТ

Multiobjective optimization

Конкретный выбор (λ) приводит к конкретной точке Парето ФРОНТА

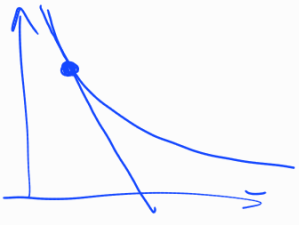
$$\min_{x \in \mathbb{R}^n} f_0(x, \lambda)$$

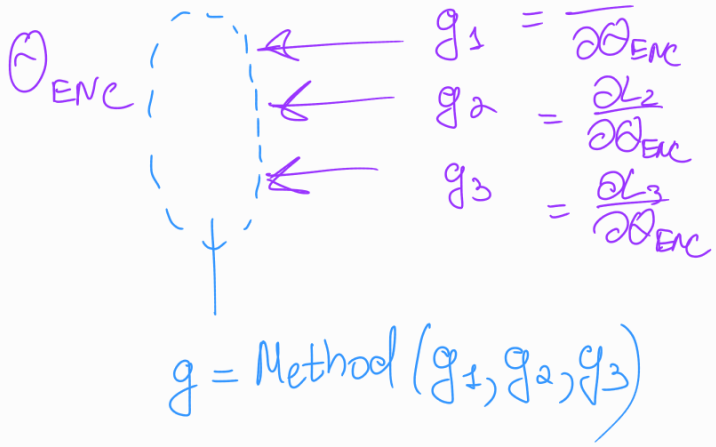
$$f_0(\lambda, x) = \min_{x \in \mathbb{R}^n} \begin{pmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_T(x) \end{pmatrix}$$

$\lambda_1, \lambda_2, \lambda_T$

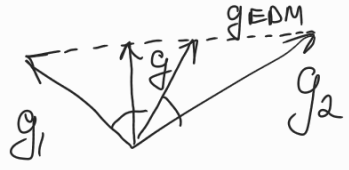
$$\sum_{i=1}^{T} \lambda_i f_i(x) \to \min_{x \in \mathbb{R}}$$

СКАЛЯРИЗАЦИЯ





Например:



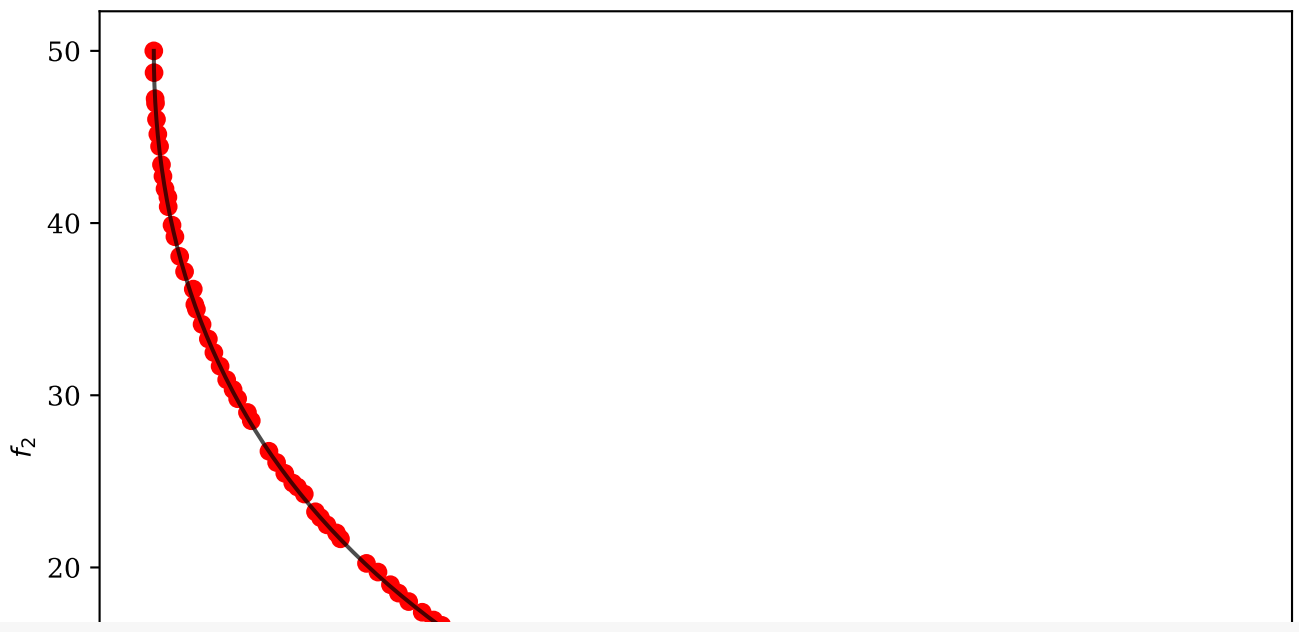
EDM

MGDA

multiple gradient descent algorithm

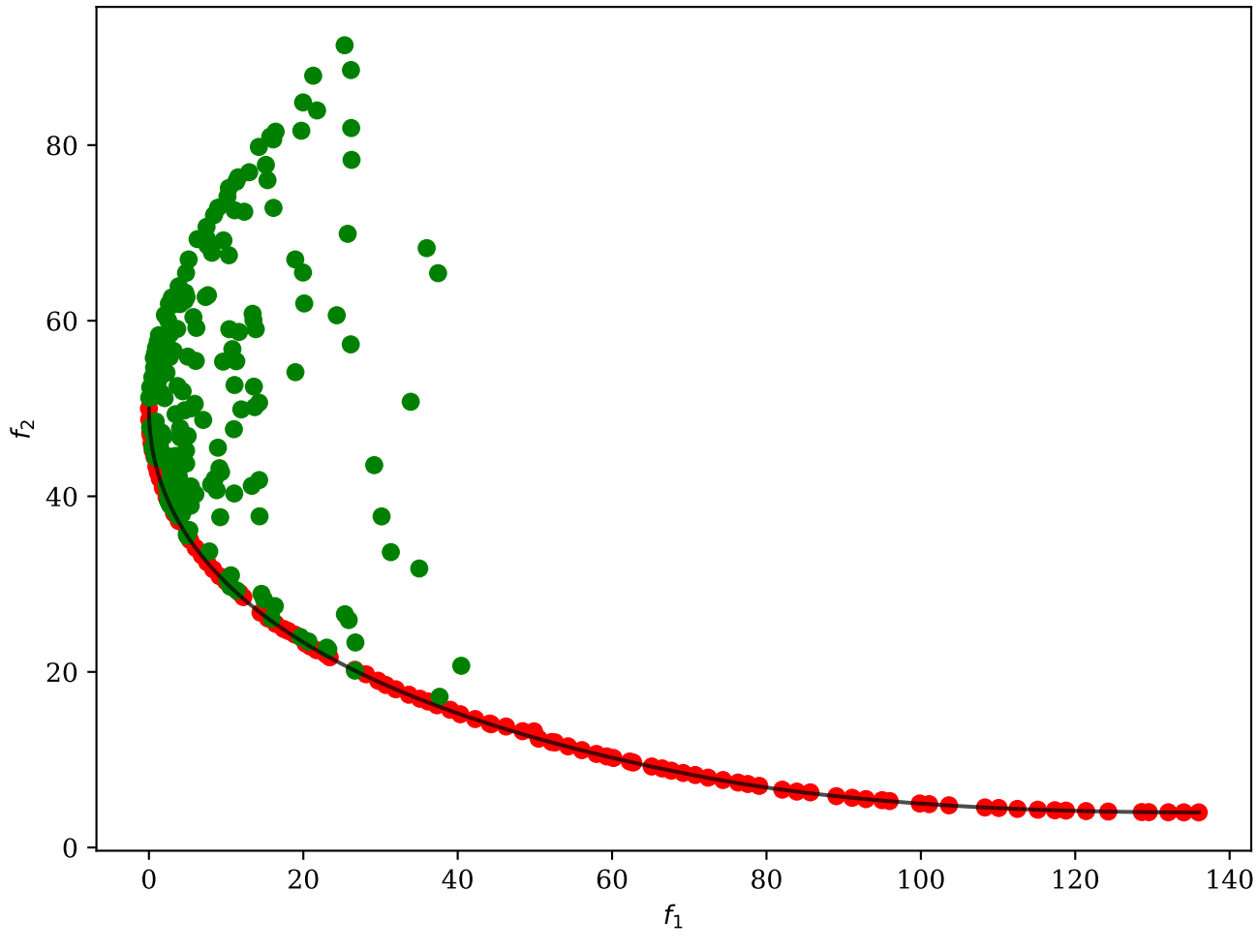
PcGrad

```
<pymoo.visualization.scatter.Scatter at 0x7fa66b3f0588>
```




```
plot = Scatter()
plot.add(problem.pareto_front(), plot_type="line", color="black", alpha=0.7)
plot.add(res.F, color="red")
for i in range(200):
    x = np.random.randn(2)
    plot.add(np.array(problem.evaluate(x)[0]), color="green")
plot.show()
```

```
<pymoo.visualization.scatter.Scatter at 0x7fa659c0b668>
```



Portfolio optimization

source

Задача:
разбогатеть
N финансовых инстр.
Вектор $w \in \mathbb{R}^N$
$1^T w = 1$
$\sum_{i=1}^{N} w_i = 1$
Историческая
статистика
ОФЗ
Сбер
Тинькоff
TESLA
GameStop
1
T
N
$P \in \mathbb{R}^{N \times T}$
$y = kx$
Вводим понятие fractional return
$r(t) = \frac{P(t) - P(0)}{P(0)}$
$\mu^T w \to \max_{w \in \mathbb{R}^N}$
$1^T w = 1$
$\Rightarrow R \in \mathbb{R}^{N \times T}$
μ_1
μ_N
$\mu \in \mathbb{R}^N$

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.

$\mu^T w$ $\mu_k < 0$ $w_k < 0$ $\mu_k w_k > 0$
$w \geq 0$? long
$w < 0$? short
прод
Δ>0
пок
ЗАНИМАЕМ АКЦИЮ ПРОДАЕМ
ПОКУПАЕМ И ВОЗВРАЩАЕ
100р
10р

Classical (Markowitz) portfolio optimization

средний рост
скалярный гиперпараметр
дисперсию

Classical (Markowitz) portfolio optimization solves the optimization problem

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

$= f(w) = \mu w - \gamma \sigma \cdot w^2$

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*.

$\max f(x)$
$g(x) = 0$

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.

1) Задача выпуклая
2) $L(w, \lambda) = f(w) + \lambda \cdot g(w) = \mu^T w - \gamma w^T \Sigma w + \lambda \cdot (1^T w - 1)$
$\nabla_w L = \mu - \gamma 2 \Sigma w + \lambda \cdot 1 = 0 \Rightarrow (\mu + \lambda \cdot 1) = 2\gamma \cdot \Sigma w \to w^* = \frac{1}{2\gamma} \Sigma^{-1} (\mu + \lambda \cdot 1)$
$\nabla_\lambda L = 1^T w - 1 = 0 \Rightarrow 1^T \cdot \frac{1}{2\gamma} \Sigma^{-1} (\mu + \lambda \cdot 1) - 1 = 0 \Rightarrow \frac{1}{2\gamma} 1^T \Sigma^{-1} \mu + \frac{1 \cdot \lambda}{2\gamma} 1^T \Sigma^{-1} 1 - 1 = 0$
$1^T \Sigma^{-1} \mu + \lambda \cdot S = 2\gamma \Rightarrow \lambda = \frac{2\gamma - 1^T \Sigma^{-1} \mu}{S}$
$w^* = \frac{1}{2\gamma} \Sigma^{-1} \left(\mu + \frac{2\gamma - 1^T \Sigma^{-1} \mu}{S} \cdot 1 \right)$
$S = \sum_{i,j=1}^{n} \sigma_{ij}$
$\sigma_{ij} = (\Sigma^{-1})_{ij}$

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.

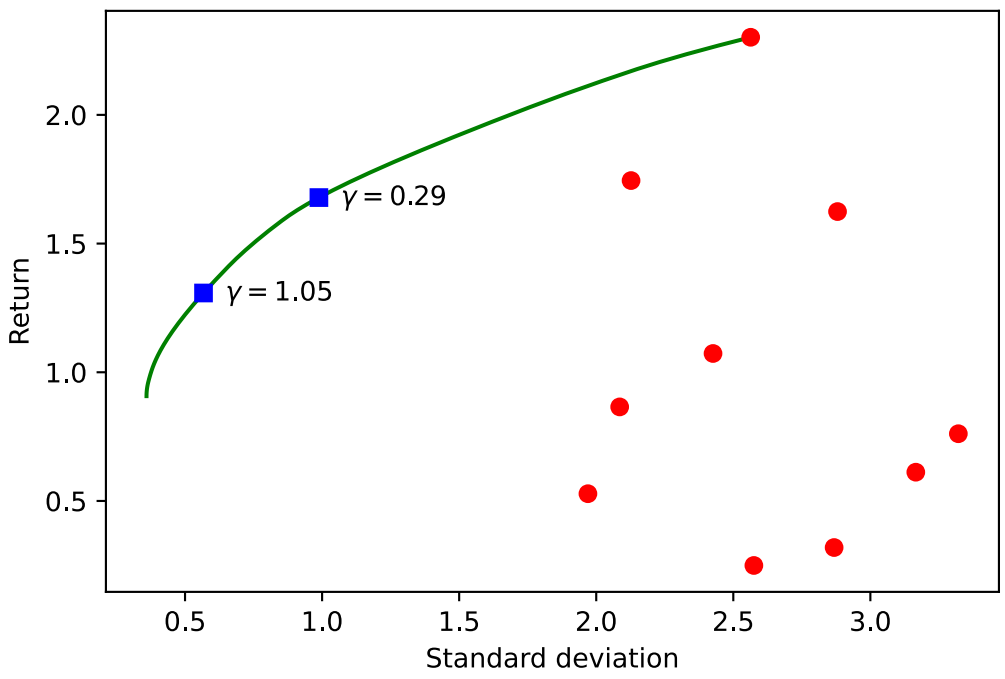
```
1  # Generate data for long only portfolio optimization.
2  import numpy as np
```

```
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
```

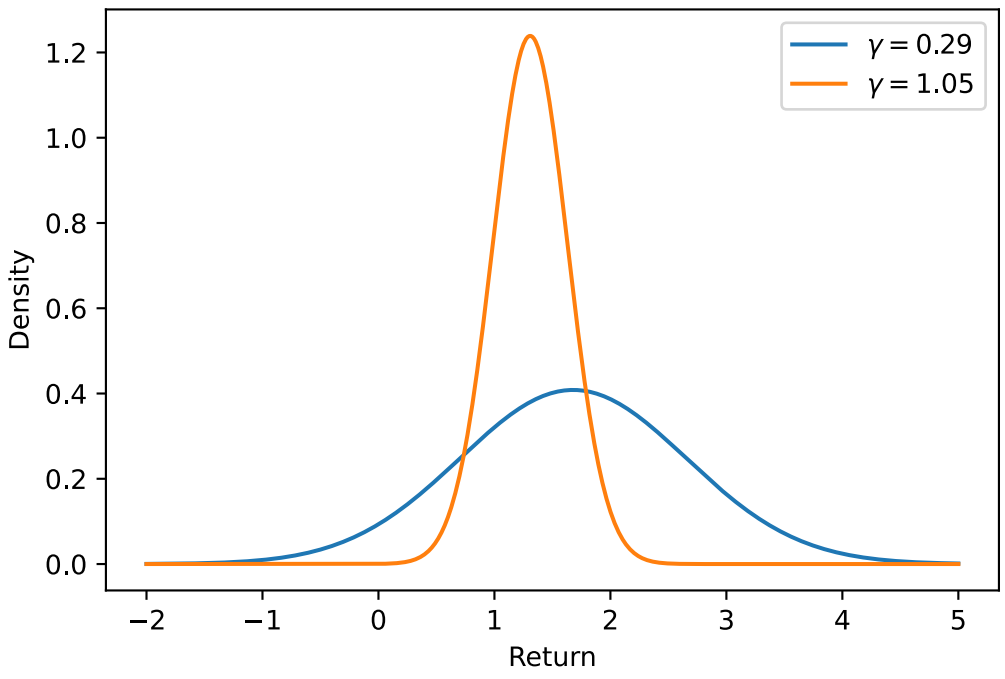
```
# Plot long only trade-off curve.
import matplotlib.pyplot as plt
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

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]+.08, ret_data[marker]-.03))
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"$\gamma = %.2f$" % gamma.value)
```

```
11
12  plt.xlabel('Return')
13  plt.ylabel('Density')
14  plt.legend(loc='upper right')
15  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.

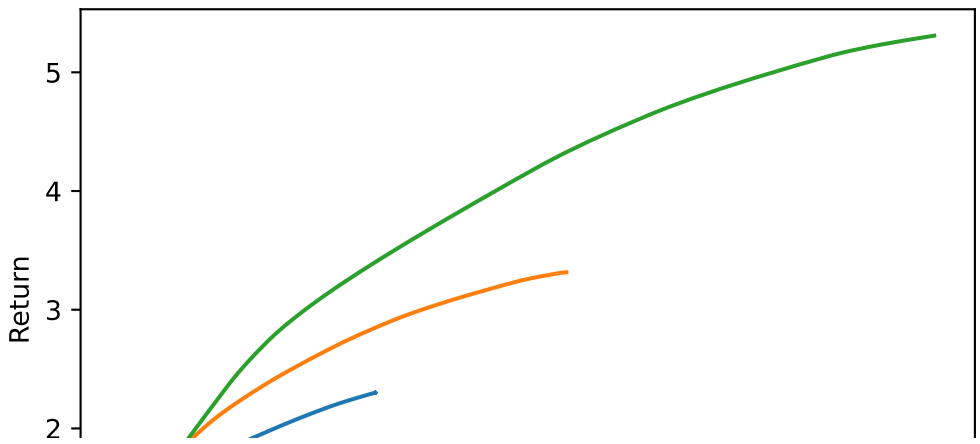
Example

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.

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

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

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

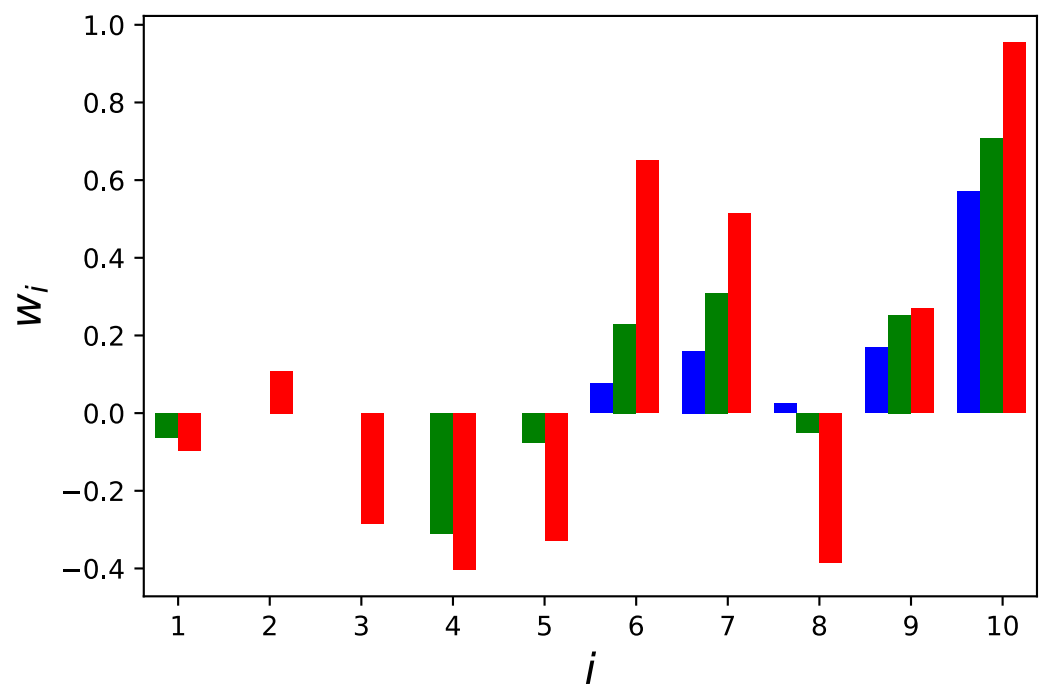


We next examine the points on each trade-off curve where $w^T\Sigma w = 2$. We plot the amount of each asset held in each portfolio as bar graphs. (Negative holdings indicate a short position.) Notice that some assets are held in a long position for the low leverage portfolio but in a short position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

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 = np.diag(np.random.uniform(0, 0.9, size=n))
F = np.random.randn(n, m)
```

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

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

```
-----------------------------------------------------------------
           OSQP v0.6.2  -  Operator Splitting QP Solver
              (c) Bartolomeo Stellato,  Goran Banjac
        University of Oxford  -  Stanford University 2021
-----------------------------------------------------------------
problem:  variables n = 6050, constraints m = 6052
          nnz(P) + nnz(A) = 172325
settings: linear system solver = qdldl,
          eps_abs = 1.0e-05, eps_rel = 1.0e-05,
          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, time_limit: off

iter   objective    pri res    dua res    rho        time
   1  -2.1359e+03   7.63e+00   3.73e+02   1.00e-01   7.33e-02s
 200  -4.1946e+00   1.59e-03   7.86e-03   3.60e-01   4.82e-01s
 400  -4.6288e+00   3.02e-04   6.01e-04   3.60e-01   8.18e-01s
 600  -4.6444e+00   2.20e-04   7.87e-04   3.60e-01   1.20e+00s
 800  -4.6230e+00   1.09e-04   3.70e-04   3.60e-01   1.60e+00s
1000  -4.6223e+00   8.59e-05   1.04e-04   3.60e-01   1.97e+00s
1200  -4.6205e+00   8.56e-05   9.35e-06   3.60e-01   2.31e+00s
1400  -4.6123e+00   6.44e-05   1.54e-04   3.60e-01   2.67e+00s
1575  -4.6064e+00   2.97e-05   4.06e-05   3.60e-01   2.99e+00s

status:               solved
solution polish:      unsuccessful
number of iterations: 1575
```

```
optimal objective:    -4.6064
run time:             3.03e+00s
optimal rho estimate: 3.87e-01

4.606413081938858
```

```
# 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, max_iter=30000)
```

```
19800  -4.6236e+00   4.14e-04   3.39e-08   5.04e-01   3.23e+02s
20000  -4.6236e+00   4.13e-04   2.83e-08   5.04e-01   3.26e+02s
20200  -4.6237e+00   4.13e-04   2.40e-08   5.04e-01   3.29e+02s
20400  -4.6279e+00   4.19e-04   1.10e-06   5.04e-01   3.32e+02s
20600  -4.6328e+00   4.14e-04   4.43e-07   5.04e-01   3.35e+02s
20800  -4.6348e+00   4.09e-04   3.19e-07   5.04e-01   3.38e+02s
21000  -4.6360e+00   4.06e-04   2.50e-07   5.04e-01   3.41e+02s
21200  -4.6368e+00   4.03e-04   2.00e-07   5.04e-01   3.45e+02s
21400  -4.6375e+00   4.00e-04   1.62e-07   5.04e-01   3.48e+02s
21600  -4.6380e+00   3.99e-04   1.40e-07   5.04e-01   3.51e+02s
21800  -4.6386e+00   3.98e-04   1.18e-07   5.04e-01   3.54e+02s
22000  -4.6392e+00   3.98e-04   1.00e-07   5.04e-01   3.57e+02s
22200  -4.6396e+00   3.97e-04   9.12e-08   5.04e-01   3.60e+02s
22400  -4.6402e+00   3.88e-04   1.72e-06   5.04e-01   3.63e+02s
22600  -4.6481e+00   3.69e-04   6.62e-07   5.04e-01   3.66e+02s
22800  -4.6513e+00   3.68e-04   3.75e-07   5.04e-01   3.70e+02s
23000  -4.6578e+00   3.65e-04   1.26e-06   5.04e-01   3.73e+02s
23200  -4.6606e+00   3.71e-04   3.81e-07   5.04e-01   3.76e+02s
23400  -4.6574e+00   3.69e-04   1.74e-07   5.04e-01   3.79e+02s
23600  -4.6563e+00   3.55e-04   3.46e-07   5.04e-01   3.82e+02s
23800  -4.6543e+00   3.48e-04   1.62e-07   5.04e-01   3.85e+02s
24000  -4.6534e+00   3.46e-04   9.43e-08   5.04e-01   3.89e+02s
24200  -4.6530e+00   3.45e-04   7.96e-08   5.04e-01   3.92e+02s
24400  -4.6575e+00   3.44e-04   6.80e-07   5.04e-01   3.95e+02s
24600  -4.6604e+00   3.40e-04   4.12e-07   5.04e-01   3.98e+02s
24800  -4.6596e+00   3.36e-04   2.92e-07   5.04e-01   4.02e+02s
25000  -4.6582e+00   3.30e-04   2.22e-07   5.04e-01   4.05e+02s
25200  -4.6570e+00   3.26e-04   1.74e-07   5.04e-01   4.08e+02s
25400  -4.6560e+00   3.21e-04   1.40e-07   5.04e-01   4.11e+02s
25600  -4.6552e+00   3.18e-04   1.21e-07   5.04e-01   4.14e+02s
25800  -4.6545e+00   3.15e-04   1.09e-07   5.04e-01   4.17e+02s
26000  -4.6540e+00   3.12e-04   9.83e-08   5.04e-01   4.20e+02s
26200  -4.6536e+00   3.10e-04   8.83e-08   5.04e-01   4.23e+02s
26400  -4.6644e+00   2.98e-04   8.13e-07   5.04e-01   4.26e+02s
26600  -4.6618e+00   2.91e-04   3.47e-07   5.04e-01   4.29e+02s
26800  -4.6579e+00   2.86e-04   2.42e-07   5.04e-01   4.33e+02s
27000  -4.6555e+00   2.84e-04   2.09e-07   5.04e-01   4.36e+02s
27200  -4.6539e+00   2.85e-04   1.89e-07   5.04e-01   4.39e+02s
27400  -4.6529e+00   2.85e-04   1.70e-07   5.04e-01   4.42e+02s
27600  -4.6522e+00   2.85e-04   1.53e-07   5.04e-01   4.45e+02s
27800  -4.6517e+00   2.84e-04   1.38e-07   5.04e-01   4.48e+02s
28000  -4.6513e+00   2.84e-04   1.24e-07   5.04e-01   4.51e+02s
28200  -4.6510e+00   2.84e-04   1.12e-07   5.04e-01   4.54e+02s
28400  -4.6507e+00   2.83e-04   1.01e-07   5.04e-01   4.57e+02s
28600  -4.6504e+00   2.82e-04   9.15e-08   5.04e-01   4.61e+02s
28800  -4.6502e+00   2.82e-04   8.31e-08   5.04e-01   4.64e+02s
29000  -4.6499e+00   2.81e-04   7.57e-08   5.04e-01   4.67e+02s
29200  -4.6497e+00   2.80e-04   6.92e-08   5.04e-01   4.70e+02s
29400  -4.6495e+00   2.80e-04   6.34e-08   5.04e-01   4.73e+02s
29600  -4.6493e+00   2.79e-04   5.82e-08   5.04e-01   4.77e+02s
29800  -4.6491e+00   2.79e-04   5.36e-08   5.04e-01   4.80e+02s
30000  -4.6489e+00   2.78e-04   4.94e-08   5.04e-01   4.83e+02s

status:               solved inaccurate
number of iterations: 30000
optimal objective:    -4.6489
run time:             4.83e+02s
optimal rho estimate: 9.34e-01

4.64886481797565
```

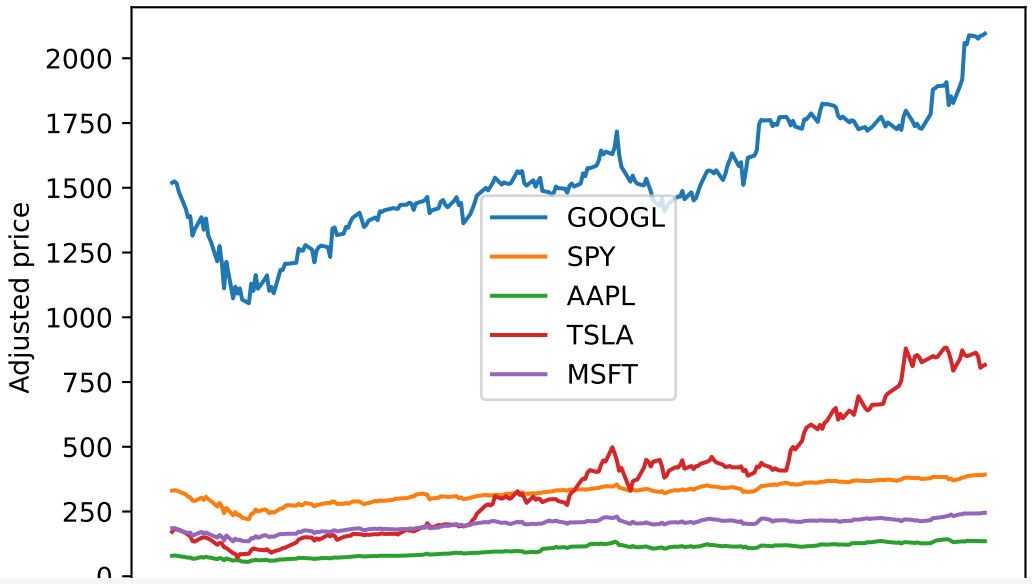
```
print('Factor model solve time = {}'.format(prob_factor.solver_stats.solve_time))
print('Single model solve time = {}'.format(prob.solver_stats.solve_time))
```

```
Factor model solve time = 3.0333686589999997
Single model solve time = 483.077024319
```

Как с деньгами обстоит вопрос

What about real data?

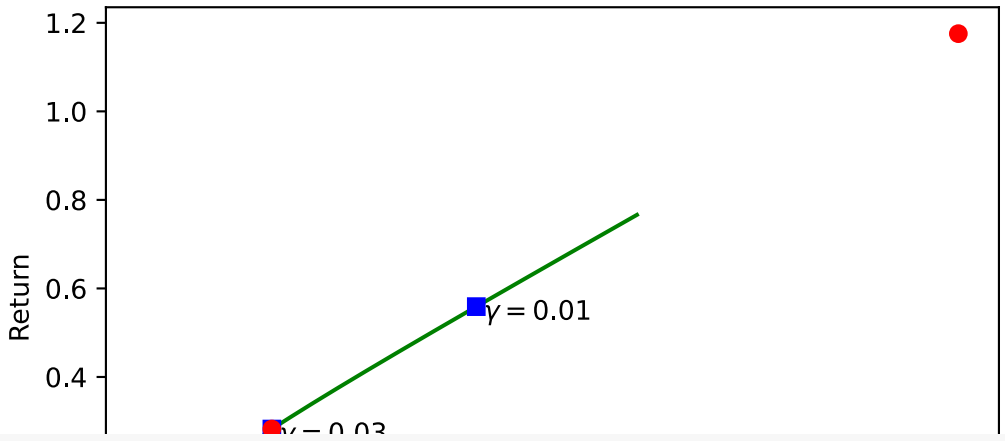

```
from pandas_datareader import data as web
import datetime
import matplotlib.pyplot as plt

stocks = ['GOOGL', 'SPY', 'AAPL', 'TSLA', 'MSFT']

start_date = (datetime.datetime.now() - datetime.timedelta(days=365)).strftime("%m-%d-%Y")
df = web.DataReader(stocks, data_source='yahoo', start=start_date)

# Adjusted price
for stock in stocks:
    plt.plot(df['Adj Close'][stock], label=stock)
plt.xlabel('date')
plt.ylabel('Adjusted price')
plt.legend()
plt.show()

# Fractional return
frac_return = {}
for stock in stocks:
    frac_return[stock] = [(price - df['Adj Close'][stock][0])/df['Adj Close'][stock][0] for price in df['Adj Close'][stock]]
    plt.plot(frac_return[stock], label=stock)
plt.xlabel('date')
plt.ylabel('Fractional return')
plt.legend()
plt.show()
```




```
# Calculating mean and covariance of fractional return
# number of stocks
N = len(frac_return)

# number of historical values per stock
M = len(frac_return[stocks[0]])

mu = np.zeros(N)
Sigma = np.zeros((N,N))
Prices = np.zeros((M,N))

for i_asset, (stock, return_array) in enumerate(frac_return.items()):
    mu[i_asset] = np.array(return_array).mean()
    Prices[:, i_asset] = return_array

Sigma = 1/N*(Prices - mu).T @ (Prices - mu)
```

```
# 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 = [3, 10, 70]
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]+.08, ret_data[marker]-.03))
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()
```




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


w = cp.Variable(N)
gamma = 10
ret = mu.T@w
risk = cp.quad_form(w, Sigma)
prob = cp.Problem(cp.Maximize(ret - gamma*risk),
                  [cp.sum(w) == 1,
                   w >= 0])
prob.solve(verbose=True)

print(f'{[(stock, x) for stock, x in zip(stocks, w.value)]}')
```

```
-----------------------------------------------------------------
          OSQP v0.6.2  -  Operator Splitting QP Solver
             (c) Bartolomeo Stellato,  Goran Banjac
        University of Oxford  -  Stanford University 2021
-----------------------------------------------------------------
problem:  variables n = 5, constraints m = 6
          nnz(P) + nnz(A) = 25
settings: linear system solver = qdldl,
          eps_abs = 1.0e-05, eps_rel = 1.0e-05,
          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, time_limit: off

iter   objective    pri res    dua res    rho        time
   1  -2.0945e-03   1.00e+00   1.36e+03   1.00e-01   2.08e-04s
  50   6.8270e+00   5.75e-06   1.96e-04   8.09e-01   5.78e-04s
plsh   6.8271e+00   2.14e-22   0.00e+00   --------   9.09e-04s

status:               solved
solution polish:      successful
number of iterations: 50
optimal objective:    6.8271
run time:             9.09e-04s
optimal rho estimate: 5.42e-01

[('GOOGL', -2.140038590828118e-22), ('SPY', 0.9999999999999993), ('AAPL', 4.435850852504018e-23), ('TSLA', 1.0105310029397988e-23), ('MSF
```

```
[(stock, x) for (stock, x) in zip(stocks, w.value)]
```

```
[('GOOGL', -1.2310333739669261e-23),
 ('SPY', 1.0705162474990867e-22),
 ('AAPL', 0.5995001807644814),
 ('TSLA', -1.5763444272763734e-23),
 ('MSFT', 0.4004998192355186)]
```

Materials

- Portfolio Optimization Algo Trading colab notebook
- Multi objective portfolio optimization

$$f(x) \to \min_{x \in \mathbb{R}^n}$$

$$f \to \min_{x,y,z}$$

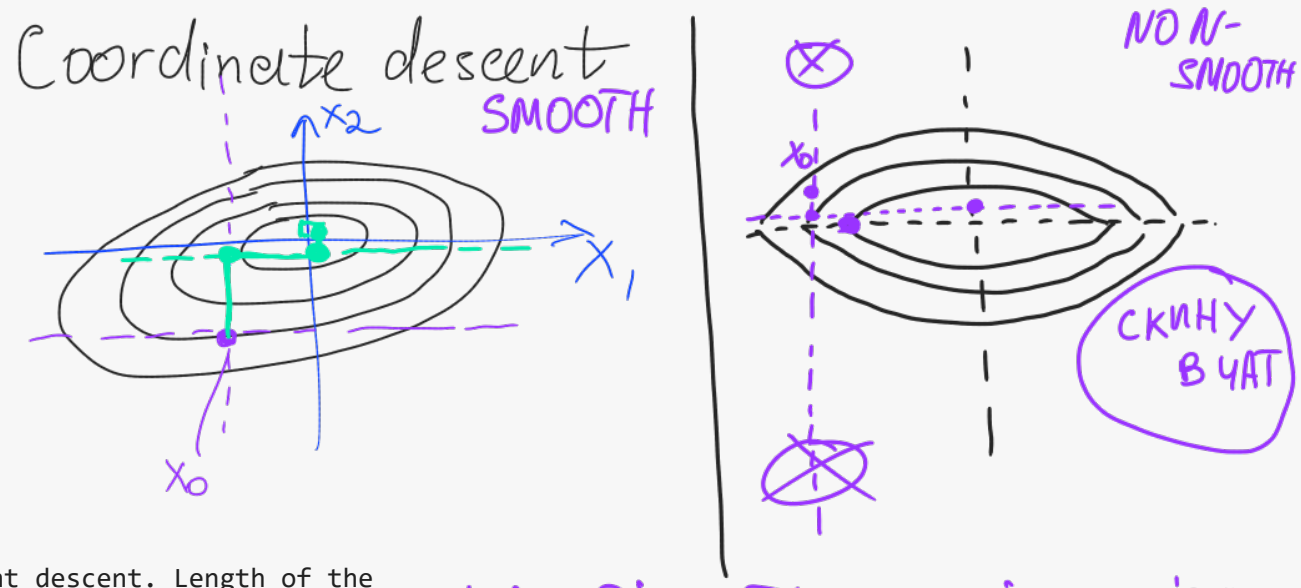
1. $x_{k+1} = \arg\min_x f(x, y_k, z_k)$
2. $y_{k+1} = \arg\min_y f(x_k, y, z_k)$
3. $z_{k+1} = \arg\min_z f(x_k, y_k, z)$

Alternating minimization

source

```
import numpy as np


def gradient_descent(init, steps, grad, proj=lambda x: x, num_to_keep=None):
    """Projected gradient descent.

```

```
    Parameters
    ----------
        initial : array
            starting point
        steps : list of floats
            step size schedule for the algorithm
        grad : function
            mapping arrays to arrays of same shape
        proj : function, optional
            mapping arrays to arrays of same shape
        num_to_keep : integer, optional
            number of points to keep

    Returns
    -------
        List of points computed by projected gradient descent. Length of the
        list is determined by `num_to_keep`.
    """
    xs = [init]
    for step in steps:
        xs.append(proj(xs[-1] - step * grad(xs[-1])))
        if num_to_keep:
          xs = xs[-num_to_keep:]
    return xs


def conditional_gradient(initial, steps, oracle, num_to_keep=None):
    """Conditional gradient.

        Conditional grdient (Frank-Wolfe) for first-order optimization.

    Parameters:
    -----------
        initial: array,
            initial starting point
        steps: list of numbers,
            step size schedule
        oracle: function,
            mapping points to points, implements linear optimization
            oracle for the objective.

    Returns:
    --------
        List of points computed by the algorithm.
    """
    xs = [initial]
    for step in steps:
        xs.append(xs[-1] + step*(oracle(xs[-1])-xs[-1]))
        if num_to_keep:
          xs = xs[-num_to_keep:]
    return xs


def gss(f, a, b, tol=1e-5):
    """Golden section search.
        Source: https://en.wikipedia.org/wiki/Golden-section_search
    Find the minimum of f on [a,b]
    Parameters:
    -----------
        f: a strictly unimodal function on [a,b]
        a: lower interval boundary
        b: uper interval boundary
    Returns:
    --------
        Point in the interval [a, b]
    """
    gr = 1.6180339887498949
    c = b - (b - a) / gr
    d = a + (b - a) / gr
    while abs(c - d) > tol:
        if f(c) < f(d):
            b = d
        else:
            a = c
        # we recompute both c and d here to avoid loss of precision
        # which may lead to incorrect results or infinite loop
        c = b - (b - a) / gr
        d = a + (b - a) / gr
    return (b + a) / 2


def random_search(oracle, init, num_steps, line_search=gss):
    """Implements random search.
    Parameters:
```



```
    -----------
        oracle: Function.
        init: Point in domain of oracle.
        num_steps: Number of iterations.
        line_search: Line search method (defaults to golden section.)

    Returns:
    --------
        List of iterates.
    """

    iterates = [init]
    for _ in range(num_steps):
        d = np.random.normal(0, 100, init.shape)
        d /= np.linalg.norm(d)
        x = iterates[-1]
        eta = line_search(lambda step: oracle(x + step * d), -1, 1)
        iterates.append(x + eta * d)
    return iterates
```

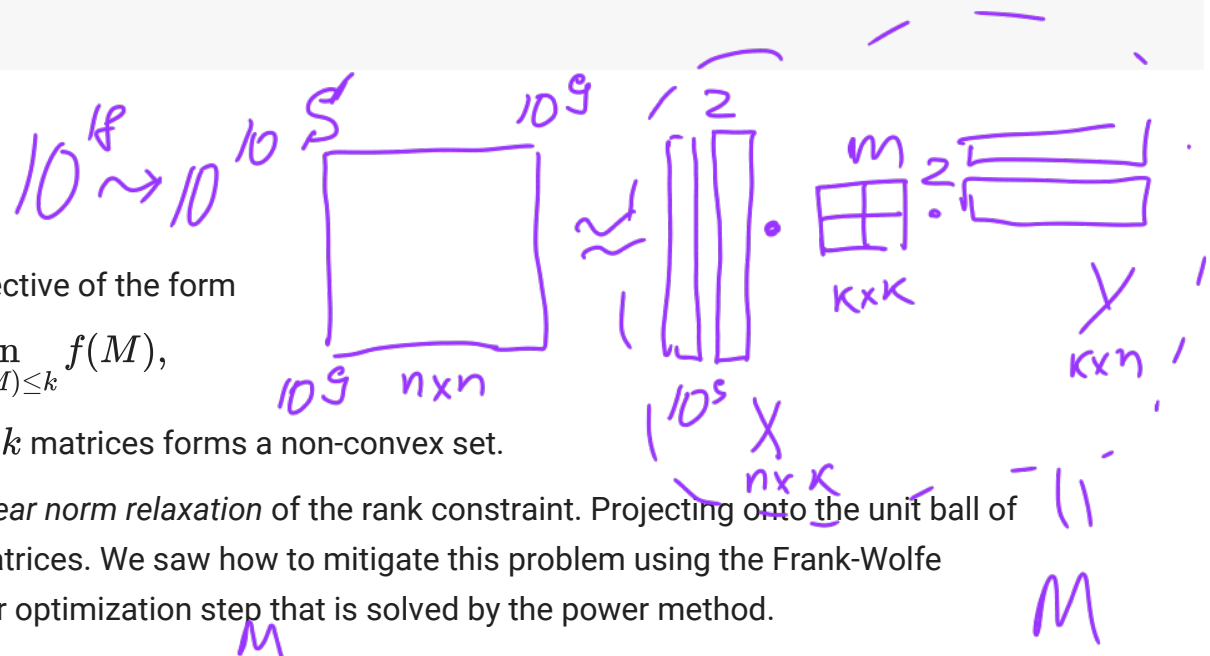
```
import numpy as np


def simplex_projection(vector):
    """Projection onto the unit simplex.

        Source: https://gist.github.com/daien/1272551
    Parameters:
    -----------
        vector: array
            Vector to be projected onto simplex.
    Returns:
    --------
        Vector in the unit simplex
    """
    if np.sum(vector) <=1 and np.alltrue(vector >= 0):
        return vector
    # get the array of cumulative sums of a sorted (decreasing) copy of v
    u = np.sort(vector)[::-1]
    cssv = np.cumsum(u)
    # get the number of > 0 components of the optimal solution
    rho = np.nonzero(u * np.arange(1, len(u)+1) > (cssv - 1))[0][-1]
    # compute the Lagrange multiplier associated to the simplex constraint
    theta = (cssv[rho] - 1) / (rho + 1.0)
    # compute the projection by thresholding v using theta
    return np.maximum(vector-theta, 0)


def nuclear_projection(matrix):
    """Projection onto nuclear norm unit ball.
    Parameters:
        matrix: two-dimensional array
            Matrix to be projected onto nuclear norm unit ball.
    Returns:
        Matrix in the unit ball of the nuclear norm.
    """
    U, s, V = np.linalg.svd(matrix, full_matrices=False)
    s = simplex_projection(s)
    return U.dot(np.diag(s).dot(V))
```

```

import matplotlib
import matplotlib.pyplot as plt
from IPython.core.display import display, HTML


kwargs = {'linewidth' : 3.5}
font = {'weight' : 'normal', 'size'   : 24}


def error_plot(ys, yscale='log'):
    plt.figure(figsize=(8, 8))
    plt.xlabel('Step')
    plt.ylabel('Error')
    plt.yscale(yscale)
    plt.plot(range(len(ys)), ys, **kwargs)


def convergence_plot(fs, gs):
    plt.figure(figsize=(14,6))
    plt.subplot(121)
    plt.title('Convergence in objective')
```

```
    plt.xlabel('Step')
    plt.ylabel('Error')
    plt.yscale('log')
    plt.plot(range(len(fs)), fs, **kwargs)
    plt.subplot(122)
    plt.title('Convergence in domain')
    plt.xlabel('Step')
    plt.yscale('log')
    plt.plot(range(len(gs)), gs, **kwargs)
    plt.tight_layout()


def setup_layout():
    matplotlib.rc('font', **font)
```

```
%matplotlib inline

#import numpy as np
import autograd.numpy as np
from autograd import grad
import matplotlib
import matplotlib.pyplot as plt
from matplotlib import colors

np.random.seed(228)

setup_layout()
```

Low-rank matrix factorization



In low-rank matrix factorization we're generally trying to solve an objective of the form

$$\min_{\text{rank}(M) \leq k} f(M),$$

where $f: \mathbb{R}^{m \times n} \to \mathbb{R}$ is a convex function. Note that the set of rank k matrices forms a non-convex set.

In lecture 5, we saw that this problem can be attacked using the *nuclear norm relaxation* of the rank constraint. Projecting onto the unit ball of the nuclear norm was a costly operation that does not scale large matrices. We saw how to mitigate this problem using the Frank-Wolfe algorithm, in which the nuclear norm projection is replaced by a linear optimization step that is solved by the power method.

Here we'll see a natural approach to solve the non-convex formulation directly without any relaxation via *alternating minimization*.

Alternating minimization

The idea behind alternating minimization is that a rank k matrix M can be written in factored form as $M = XY^\top$, where $X \in \mathbb{R}^{m \times k}$ and $Y \in \mathbb{R}^{n \times k}$.

Given initial guesses X_0, Y_0, we can then alternate between optimizing X and optimizing Y separately as follows:

For $t = 1, \ldots, T$:

- $X_t = \arg\min_X f(XY_{t-1}^\top)$
- $Y_t = \arg\min_Y f(X_t Y^\top)$

Since matrix multiplication is bilinear, the function $f(XY^\top)$ is convex in its argument X and also convex in its argument Y.

```
def alternating_minimization(left, right, update_left, update_right,
                             num_updates):
    """Alternating minimization."""
    iterates = [(left, right)]
    for _ in range(num_updates):
        left = update_left(right)
        right = update_right(left)
        iterates.append((left, right))
    return iterates
```

Check out this monograph by Jain and Kar for a survey of convergence results for alternating minimization.

Matrix completion

A common instance of this general problem is known as *matrix completion*. Here we have a partially observed matrix $m \times n$ matrix and we try to fill in its missing entries. This is generally impossible unless we make additional assumptions on the target matrix. A natural assumption is that the target matrix is close to low rank. In other words, the matrix is specified by far fewer than mn parameters.

To set up some notation:

- We observe coordinates of an unknown matrix $A \in \mathbb{R}^{m \times n}$ specified by a set Ω.
- We denote by P_Ω the coordinate projection of a matrix onto the set of entries in Ω.
- We will denote by $\|\cdot\|_F$ the Frobenius norm.

The matrix completion objective can then be written as:

$$\min_{X \in \mathbb{R}^{m \times k}, Y \in \mathbb{R}^{n \times k}} \frac{1}{2} \|P_\Omega(A - XY^\top)\|_F^2$$

Assumptions

To make the problem tractable, researchers rely on primarily two assumptions:

- Uniformly random samples: The entries of Ω are chosen independently at random.
- Incoherence: The entries of A are "spread out" so that a random samples picks up a propertional share of A with good probability. Formally, this can be achieved by assuming that the singular vectors of A have small ℓ_∞-norm.

Alternating updates for matrix completion

Here we'll compute the updates needed for alternating minimization via a naive direct solve in each row using the pseudoinverse.

In other words, this approach takes advantage of the explicit form of f as the Frobenius norm. While simple and slow when naively implemented, this approach, known as Alternating Least Squares is popular because it is possible to create fairly efficient distributed versions of the algorithm.

Since both sides of the alternation are equivalent up to transposition, let's consider our problem for fixed X, solving for the least squares solution Y in

$$\min_{Y \in \mathbb{R}^{n \times k}} \frac{1}{2} \left\| P_\Omega(A - XY^\top) \right\|_F^2 .$$

Since the i-th row $\mathbf{y}_i$ of Y is the only component of Y that appears in the i-th column of $P_\Omega(A - XY^\top)$, and the Frobenius norm is additive in matrix entries, we can optimize each row separately and combine them later to recover the solution to the joint problem over matrices Y; so fix $i \in [n]$ and consider

$$\min_{\mathbf{y} \in \mathbb{R}^k} \|\mathbf{s}_i \times (\mathbf{a}_i - X\mathbf{y})\|_2^2 ,$$

where $\mathbf{a}_i$ is the i-th column vector of A and $\mathbf{s}_i$ is a binary vector corresponding to projection onto known entries $(\cdot, i) \in \Omega$ (where above we use pointwise multiplication, but rewriting this as multiplication with a diagonal binary matrix we recover a cannonical least squares problem).

This means we need to solve n k-dimensional-input m-dimensional-output least-squares problems. Since $\mathbf{s}_i$ is binary, we can reduce the size of the output dimension of the least-squares problem by ignoring the entries that are zeroed out (reducing output dimension to $\|\mathbf{s}_i\|_1$).

This approach is pretty slow when performed serially!. We could instead use any of the convex solvers we already saw. The advantage of a direct solve is that we have no additional hyperparameters to worry about. However, we can see that the independence of the n subproblems would be amenable to a distributed implementation.

```
def update_right(A, S, X):
    """Update right factor for matrix completion objective."""
    m, n = A.shape
    _, k = X.shape
    Y = np.zeros((n, k))
    # For each row, solve a k-dimensional regression problem
    # only over the nonzero projection entries. Note that the
    # projection changes the least-squares matrix siX so we
    # cannot vectorize the outer loop.
    for i in range(n):
        si = S[:, i]
        sia = A[si, i]
        siX = X[si]
        Y[i,:] = np.linalg.lstsq(siX, sia)[0]
    return Y

def update_left(A, S, Y):
    return update_right(A.T, S.T, Y)
```

We can now instantiate the general algorithm for our problem.

```
def altmin(A, S, rank, num_updates):
    """Toy implementation of alternating minimization."""
    m, n = A.shape
    X = np.random.normal(0, 1, (m, rank))
    Y = np.random.normal(0, 1, (n, rank))
    return alternating_minimization(X, Y,
                                    lambda Y: update_left(A, S, Y),
                                    lambda X: update_right(A, S, X),
                                    num_updates)
```

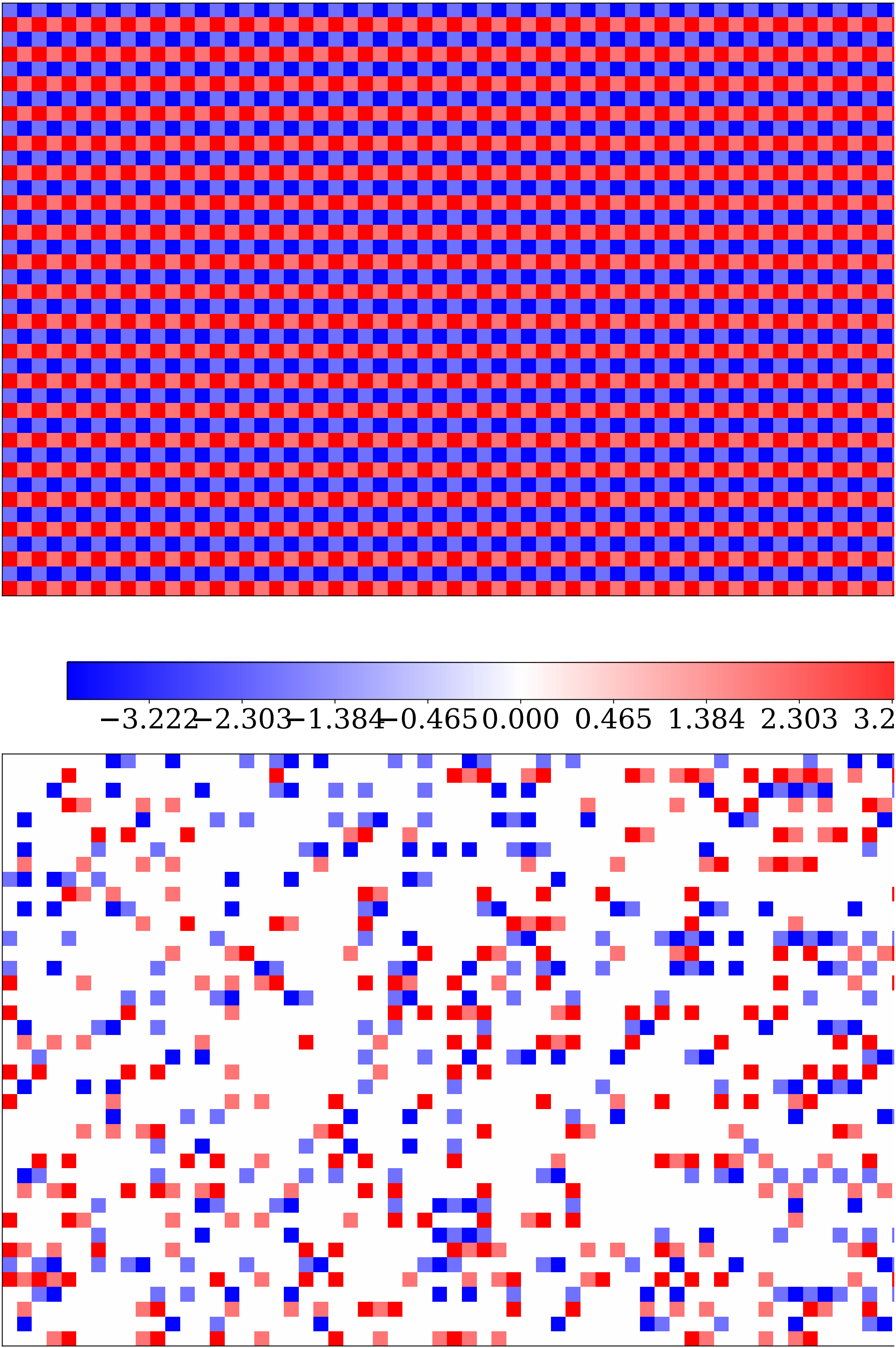
Below is code to plot a matrix decomposition in a neat manner. Ignore this for now, unless you're really into matplotlib.

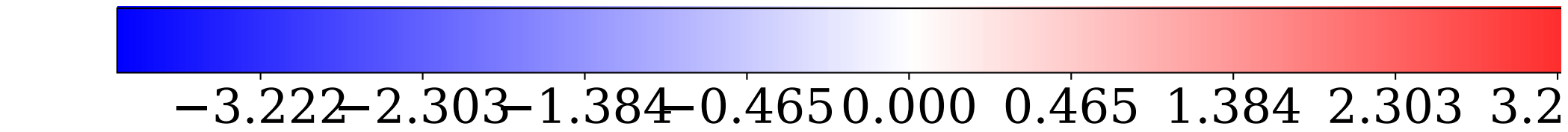
```
def plot_decomposition(A, U=None, V=None):
    """Plot matrix decomposition."""
    m, n = A.shape
    fig_height = 9
    fig_width = float(n)*fig_height/m

    cmap=plt.get_cmap('bwr')
    bounds=np.concatenate([np.linspace(-100,-4,1), np.linspace(-4,-0.005,114),
        np.linspace(-0.005,0.005,25), np.linspace(0.005,4,114),np.linspace(4,100,1)])
    norm = colors.BoundaryNorm(bounds, cmap.N)

    fig = plt.figure(figsize=(fig_width, fig_height))

    rects = [[0.05,0.15,0.8,0.8],[0.825,0.15,0.1,0.8],[0.05,0.05,0.8,0.1]]

    ims = []
    for (rect, mat) in zip(rects, [A, U, V]):
        if type(mat)==type(None):
            break
        ax = fig.add_axes(rect)
        ims.append(
            ax.imshow(mat, cmap=cmap,norm=norm,interpolation='none'))
        ax.set_xticklabels([])
        ax.set_yticklabels([])
        ax.xaxis.set_tick_params(size=0)
        ax.yaxis.set_tick_params(size=0)

    cbaxes = fig.add_axes([0.1, 0.01, 0.7, 0.05])
    plt.colorbar(ims[0], orientation='horizontal', cax=cbaxes)
    plt.show()
```

Let's see how alternating minimization works in a toy example.

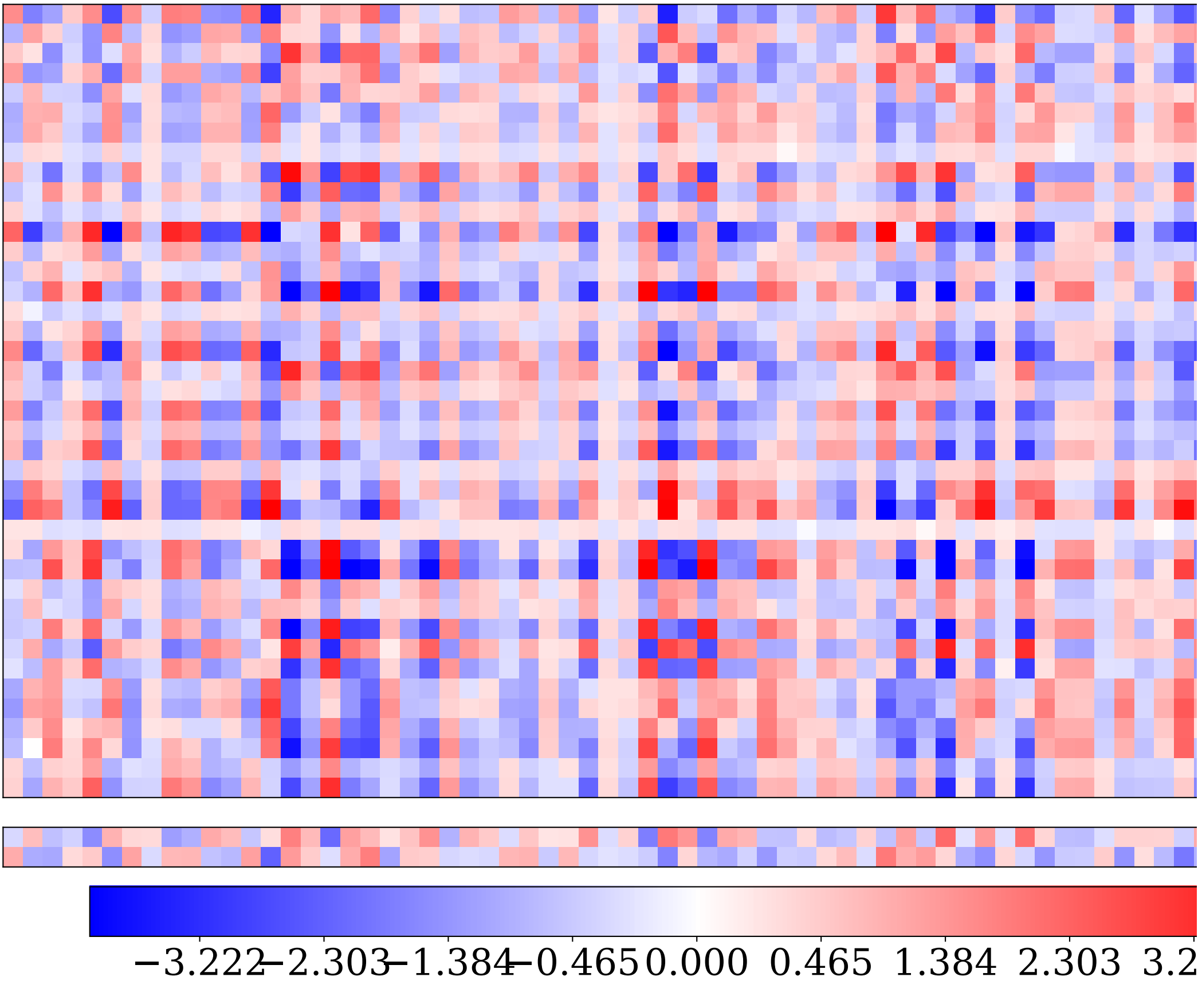
```
def subsample(A, density):
    """Randomly zero out entries of the input matrix."""
    C = np.matrix.copy(A)
    B = np.random.uniform(0, 1, C.shape)
    C[B > density] = 0
    return C, B <= density

def example1():
    """Run alternating minimization on subsample of rank 2 matrix."""

    A = np.zeros((40, 70))
    # Create rank 2 matrix with checkerboard pattern
    for i in range(0,40):
        for j in range(0,70):
            if divmod(i, 2)[1]==0:
                A[i,j] += -3.0
            if divmod(i, 2)[1]==1:
                A[i,j] += 3.0
            if divmod(j, 2)[1]==0:
                A[i,j] += 1.0
            if divmod(j, 2)[1]==1:
                A[i,j] += -1.0
    plot_decomposition(A)
    B, S = subsample(A, 0.25)
    plot_decomposition(B)
    results = altmin(A, S, 2, 10)
    for (U, V) in results:
        plot_decomposition(np.dot(U, V.T), U, V.T)

    return
```

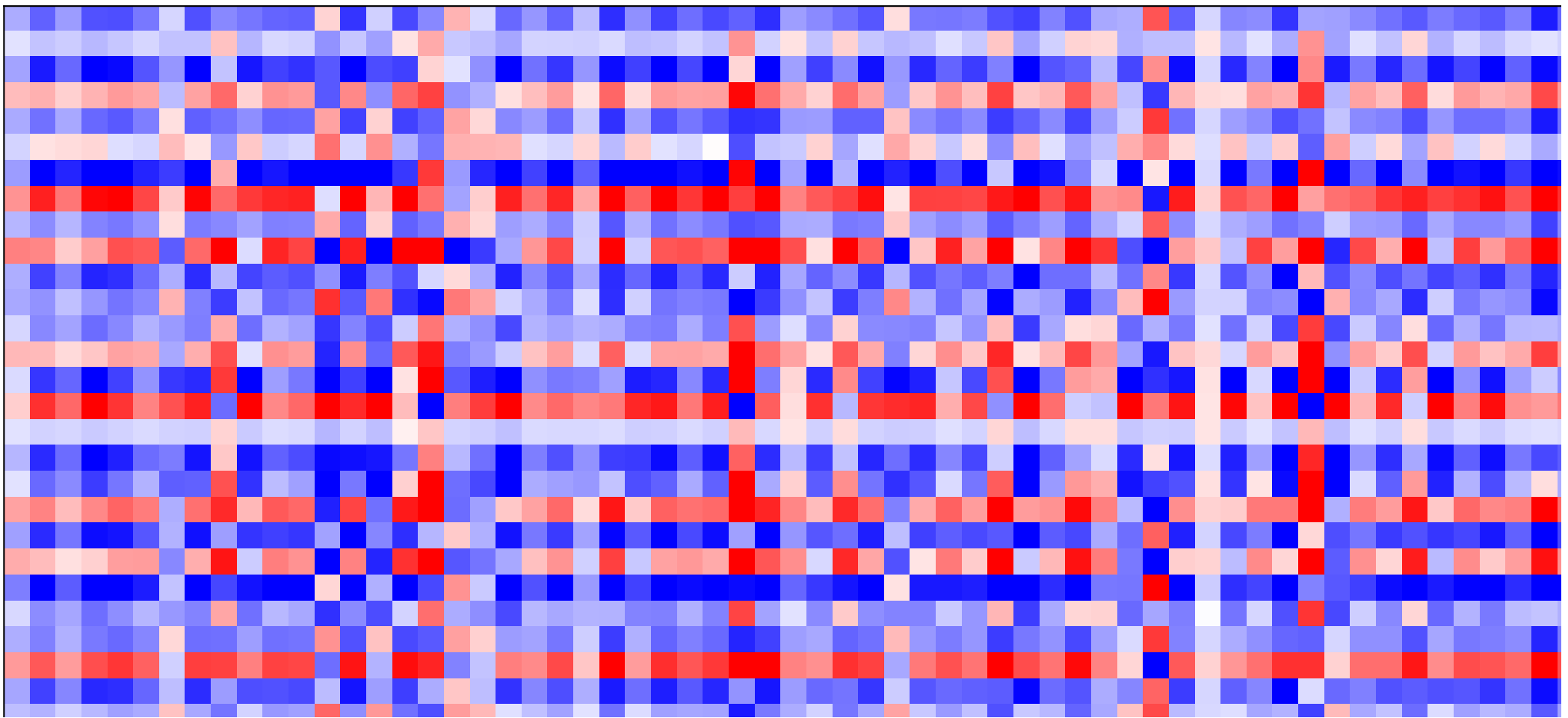
```
example1()
```

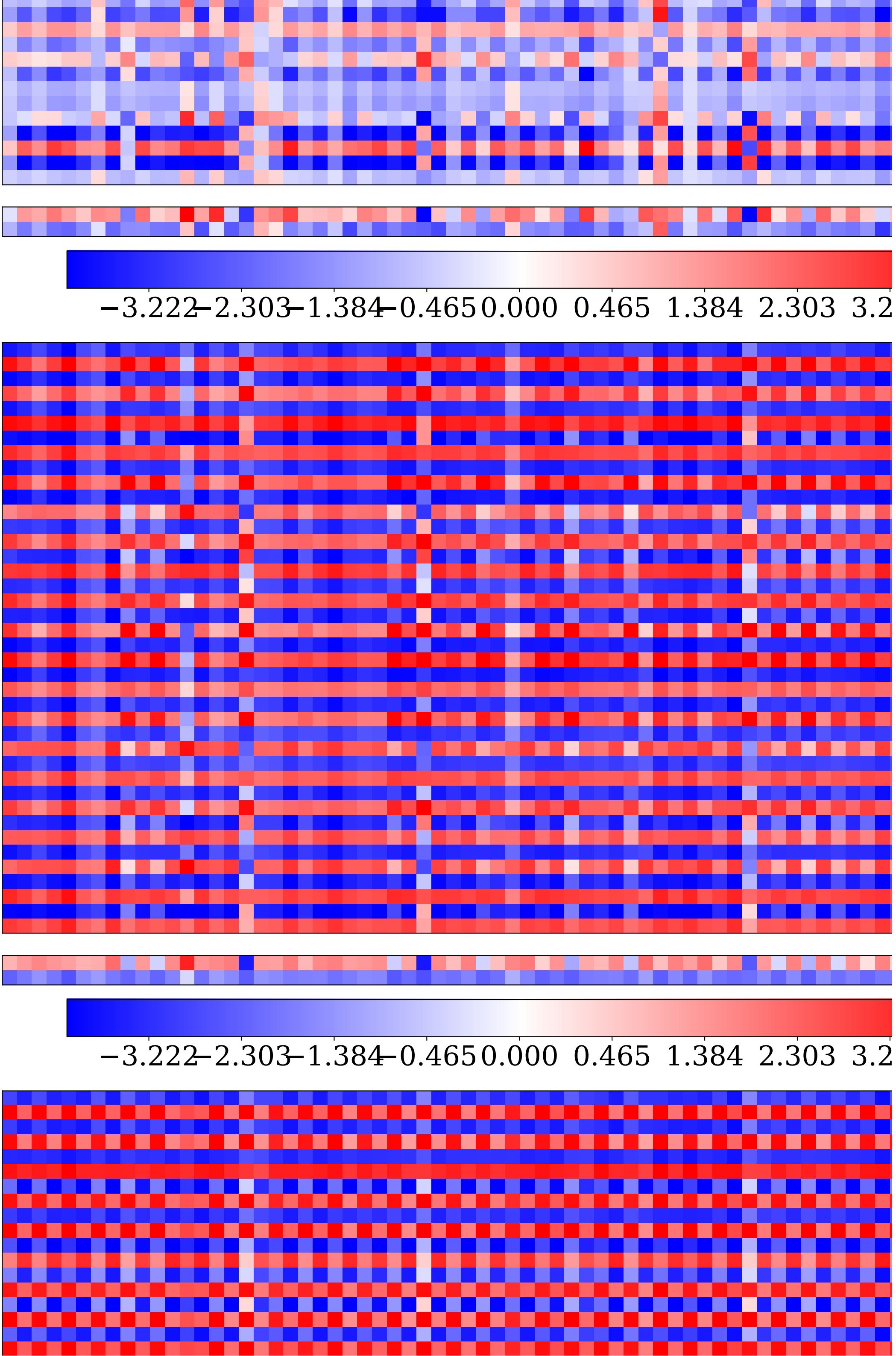

−3.222
−2.303
−1.384
−0.465
0.000
0.465
1.384
2.303
3.2

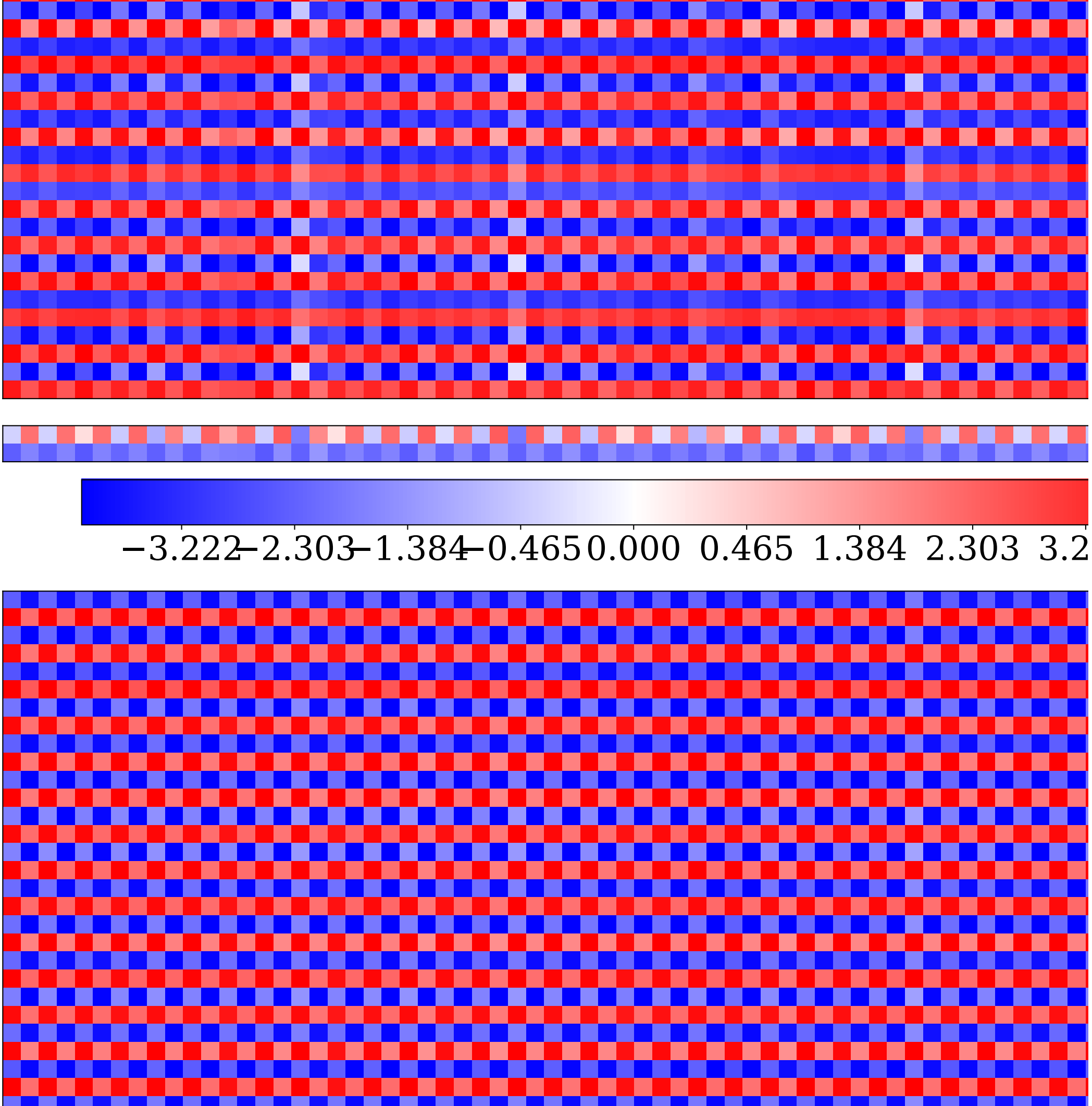



```
/usr/local/lib/python3.6/dist-packages/autograd/tracer.py:48: FutureWarning: `rcond` parameter will change to the default of machine prec
To use the future default and silence this warning we advise to pass `rcond=None`, to keep using the old, explicitly pass `rcond=-1`.
  return f_raw(*args, **kwargs)
```



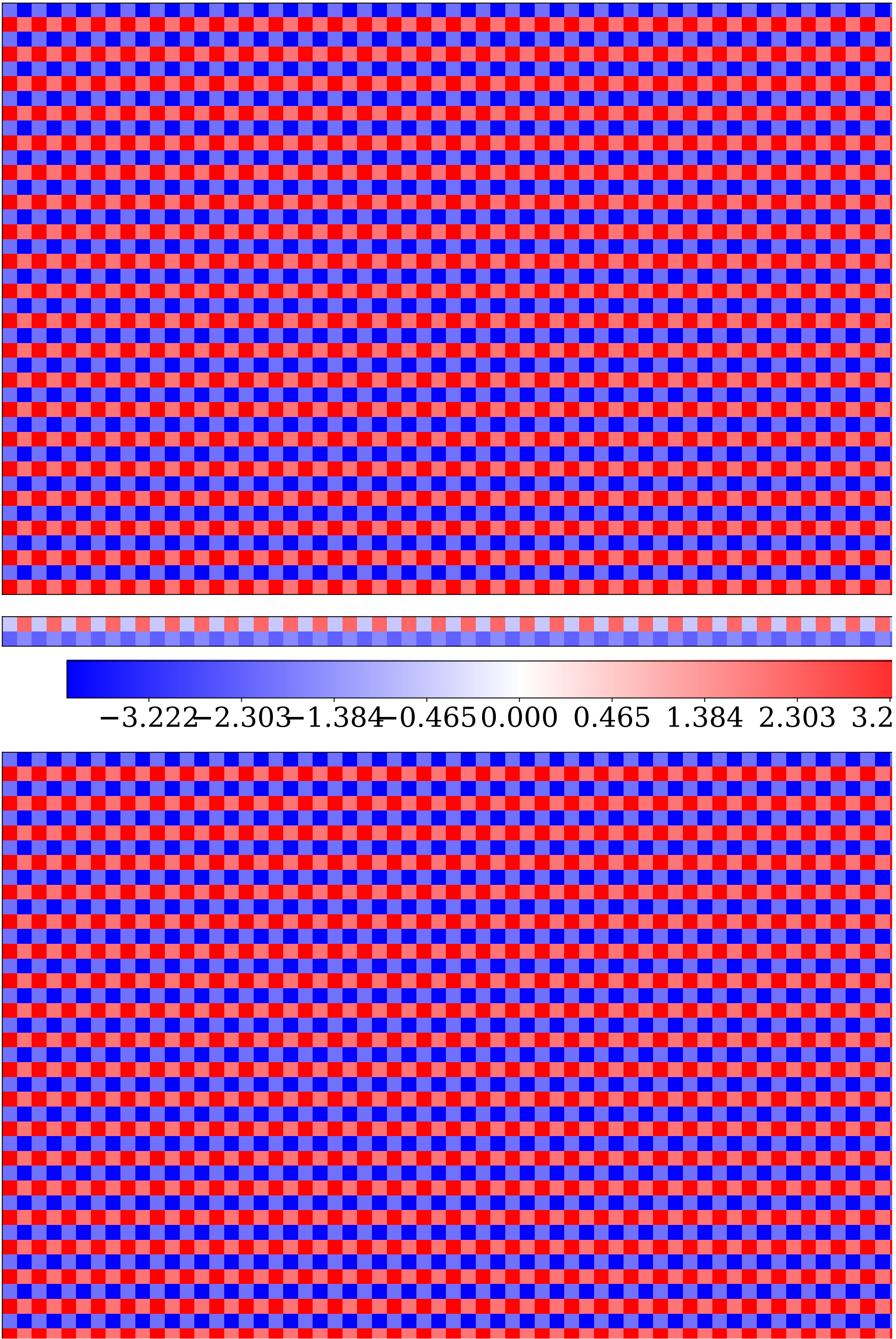


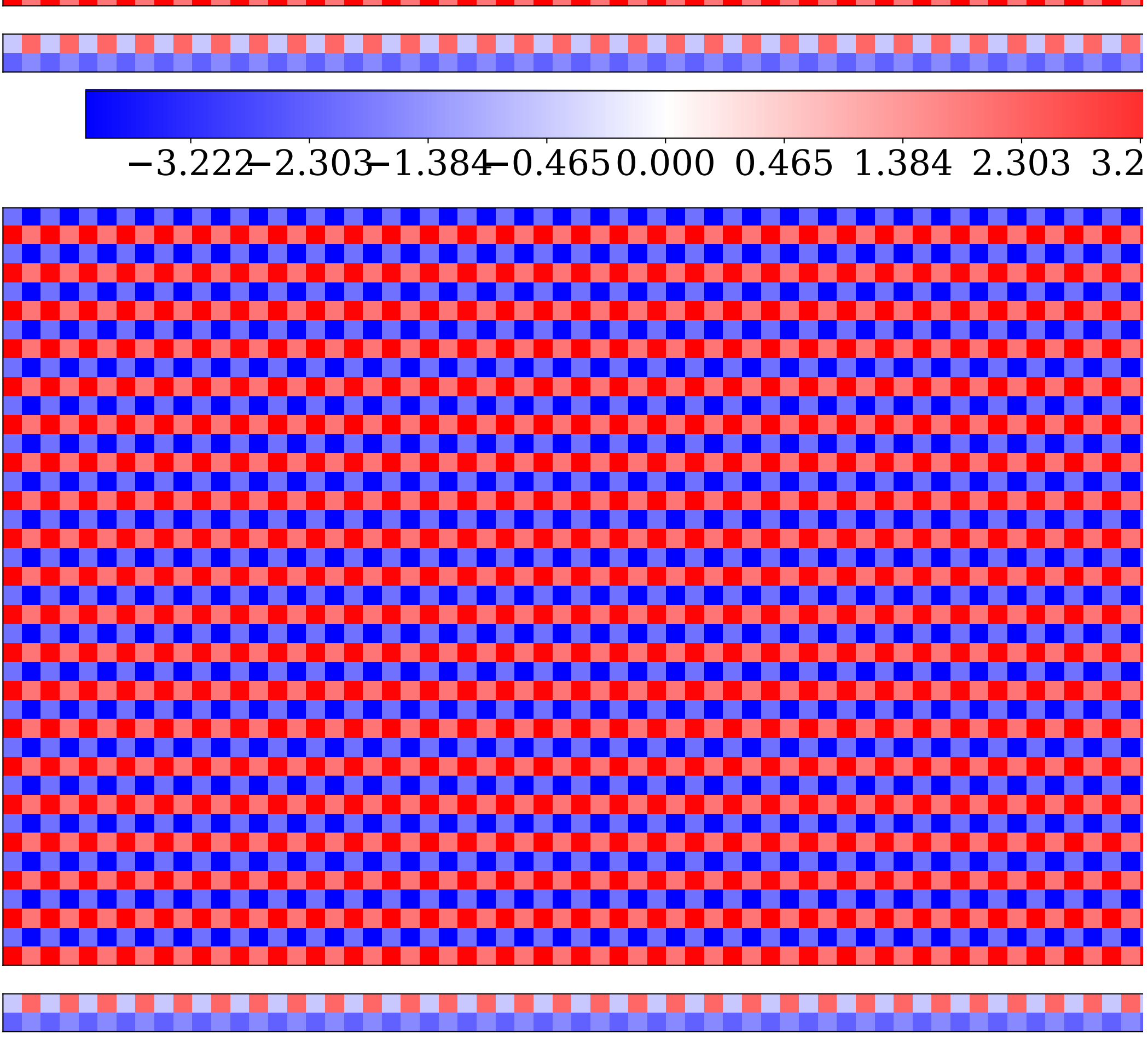

−3.222
−2.303
−1.384
−0.465
0.000
0.465
1.384
2.303
3.2
−3.222
−2.303
−1.384
−0.465
0.000
0.465
1.384
2.303
3.2

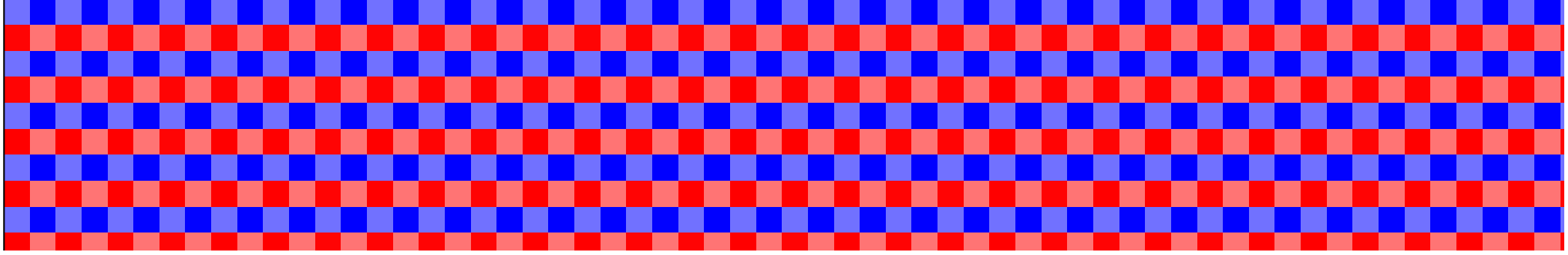

−3.222
−2.303
−1.384
−0.465
0.000
0.465
1.384
2.303
3.2

−3.222
−2.303
−1.384
−0.465
0.000
0.465
1.384
2.303
3.2

−3.222
−2.303
−1.384
−0.465
0.000
0.465
1.384
2.303
3.2


−3.222
−2.303
−1.384
−0.465
0.000
0.465
1.384
2.303
3.2


−3.222
−2.303
−1.384
−0.465
0.000
0.465
1.384
2.303
3.2



We see that after a few iterations, alternating minimization has found a good approximation. Restart this algorithm a few times to see that the convergence is affected strongly by the random initialization.

Exercises

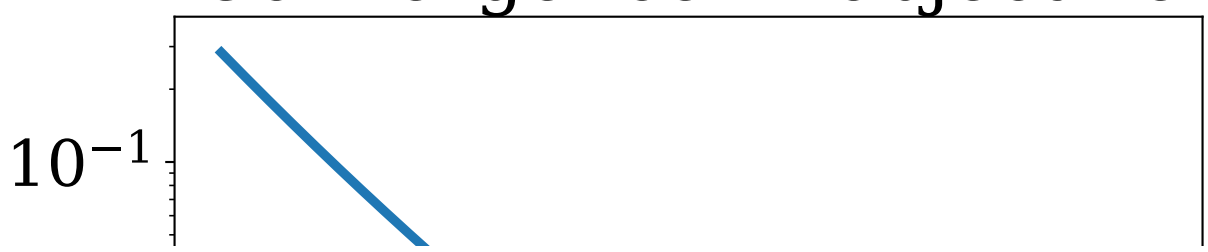
- Code up a more serious implementation that does not ever compute a full $m \times n$ matrix, but rather works with a given set of observed entries. Use stochastic gradient descent as a sub-routine for the updates.
- Do a running time analysis of the algorithm.

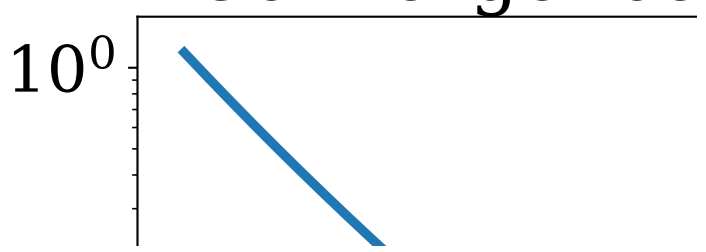
Comparison with gradient descent and nuclear norm projection

Below is our example from Lecture 5.

```
n, k = 1000, 10
# random rank-10 matrix normalized to have nuclear norm 1
U = np.random.normal(0, 1, (n, k))
U = np.linalg.qr(U)[0]
L = np.diag(np.random.uniform(0, 1, k))
L /= np.sum(L)
A = U.dot(L.dot(U.T))
# pick which entries we observe uniformly at random
S = np.random.randint(0, 2, (n, n))
# multiply A by S coordinate-wise
# B = P_\Omega(A)
B = np.multiply(A, S)

def mc_objective(B, S, X):
    """Matrix completion objective."""
    # 0.5*\|P_\Omega(A-X)\|_F^2
    return 0.5 * np.linalg.norm(B-np.multiply(X, S), 'fro')**2

def mc_gradient(B, S, X):
    """Gradient of matrix completion objective."""
    return np.multiply(X, S) - B

```

```
def example2():
    # start from random matrix of nuclear norm 1
    X0 = np.random.normal(0,1, (n,n))
    X0 = nuclear_projection(X0.dot(X0.T))
    objective = lambda X: mc_objective(B, S, X)
    gradient = lambda X: mc_gradient(B, S, X)
    Xs = gradient_descent(X0, [0.2]*40, gradient, nuclear_projection)

    convergence_plot([objective(X) for X in Xs],
                     [np.linalg.norm(A-X, 'fro')**2 for X in Xs])

example2()
```





The algorithm was pretty slow even on this tiny example.

Below we compare it with a variant of alternating minimization that makes a single gradient step in each update. There are numerous natural variants depending on which optimizer we choose. A popular method involves stochastic updates that use only a single entry.

```
def mc_objective_factored(B, S, X, Y):
    """Matrix completion objective."""
    m, n = B.shape
    return 0.5 * np.linalg.norm(B-np.multiply(np.dot(X, Y.T), S))**2

def altmin_gd(rank, num_updates):
    """Toy implementation of alternating minimization."""
    m, n = A.shape
    X = np.linalg.qr(np.random.normal(0, 1, (n, k)))[0]
    Y = np.linalg.qr(np.random.normal(0, 1, (n, k)))[0]
    iterates = [(X, Y)]
    for i in range(num_updates):
        X = X - grad(lambda X: mc_objective_factored(B, S, X, Y))(X)
        Y = Y - grad(lambda Y: mc_objective_factored(B, S, X, Y))(Y)
        iterates.append((X, Y))
    return iterates
```

```
results = altmin_gd(10, 1000)
obj_values = [mc_objective_factored(B, S, X, Y) for (X, Y) in results]
dom_values = [np.linalg.norm(A-X.dot(Y.T), 'fro')**2 for (X, Y) in results]
convergence_plot(obj_values, dom_values)
```

The convergence behavior is pretty peculiar. It rapidly converges to the quality of the all zeros solution and then slows down substantially.

```
mc_objective(B, S, 0)
```

```
0.028210546968991758
```

Tensor completion factorization

Problem

Let us consider a problem of fitting tensor T with the following rank r CP decomposition:

$$\|T - (A, B, C) \cdot I\|_F^2 \to \min_{A \in \mathbb{R}^{I \times r}, B \in \mathbb{R}^{J \times r}, C \in \mathbb{R}^{K \times r}}$$

In this manner we would like to calculate any entry of the tensor T as a following sum of r summands:

$$T_{ijk} = \sum_{p=1}^{r} A_{ip} B_{jp} C_{kp}$$

Schematic illustration of CP-decomposition: