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

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

where $f \colon \mathbb{R}^{m \times n} \to \mathbb{R}$ 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)
```

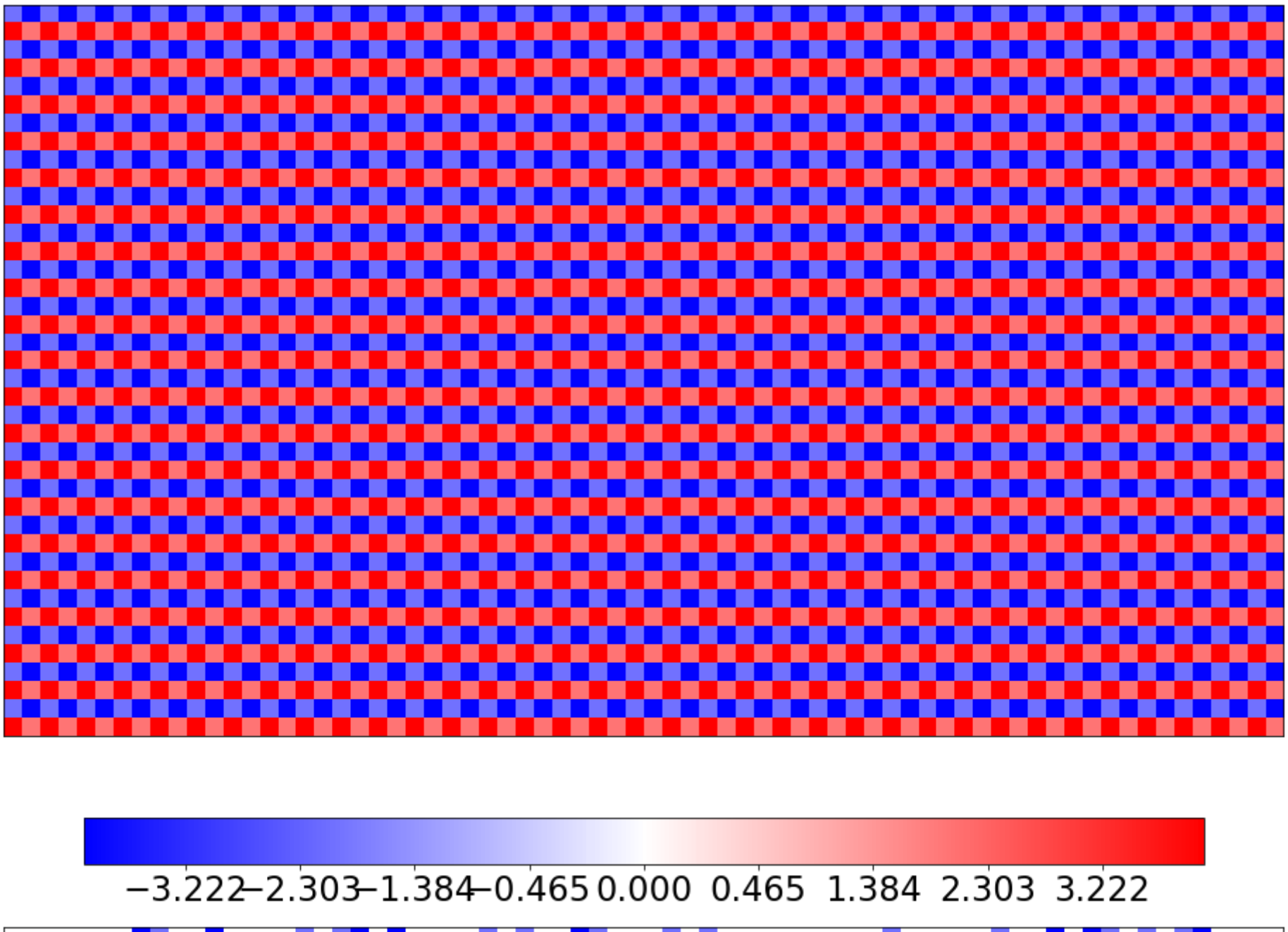
Below is code to plot a matrix decomposition in a neat manner. Ignore this for now, unless you're really into matplotlib.

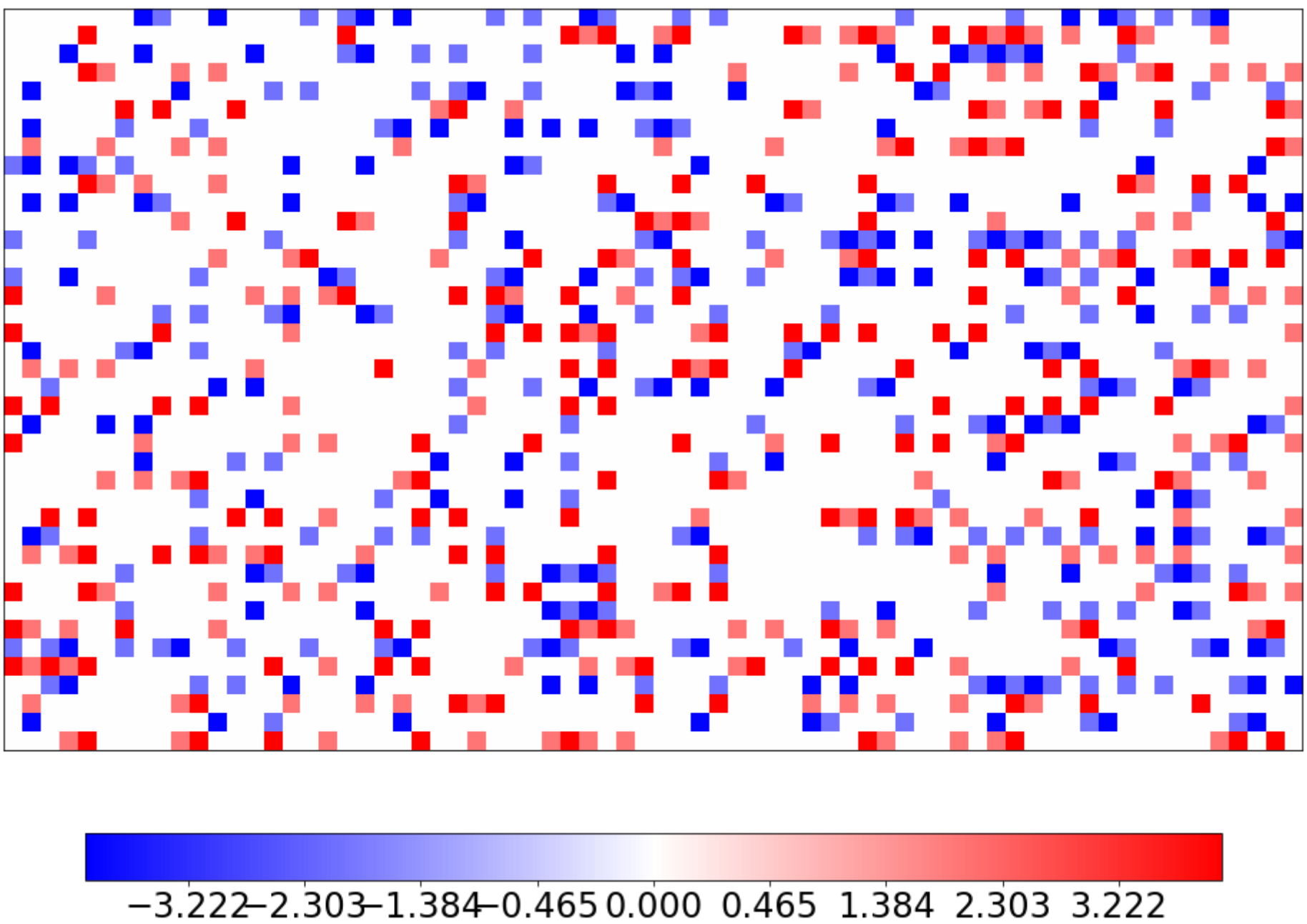
```
def plot_decomposition(A, U=None, V=None):
    """Plot matrix decomposition."""
    m, n = A.shape
    fig_height = 9
    fig_width = float(n)*fig_height/m

    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.

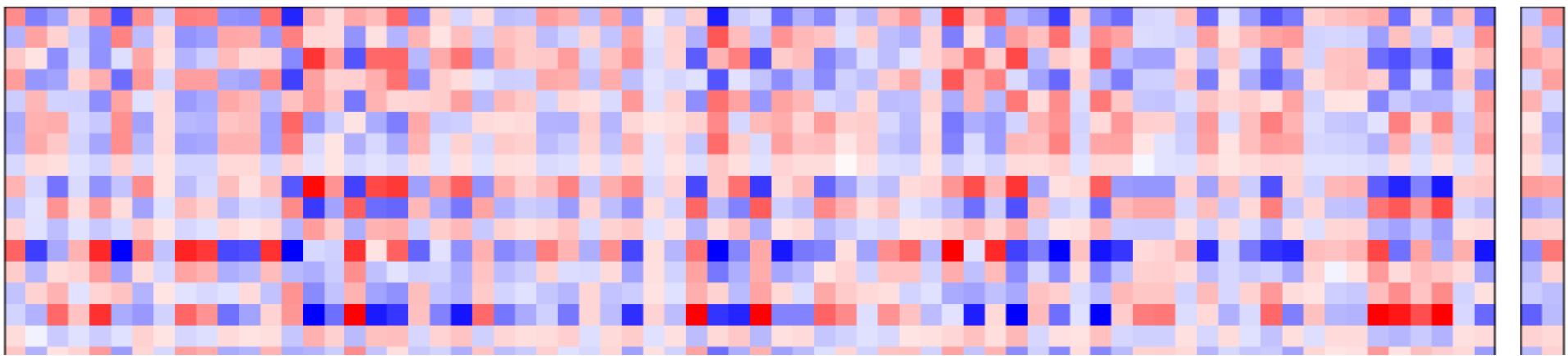
```
def subsample(A, density):
    """Randomly zero out entries of the input matrix."""
    C = np.matrix.copy(A)
    B = np.random.uniform(0, 1, C.shape)
    C[B > density] = 0
    return C, B <= density

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
```

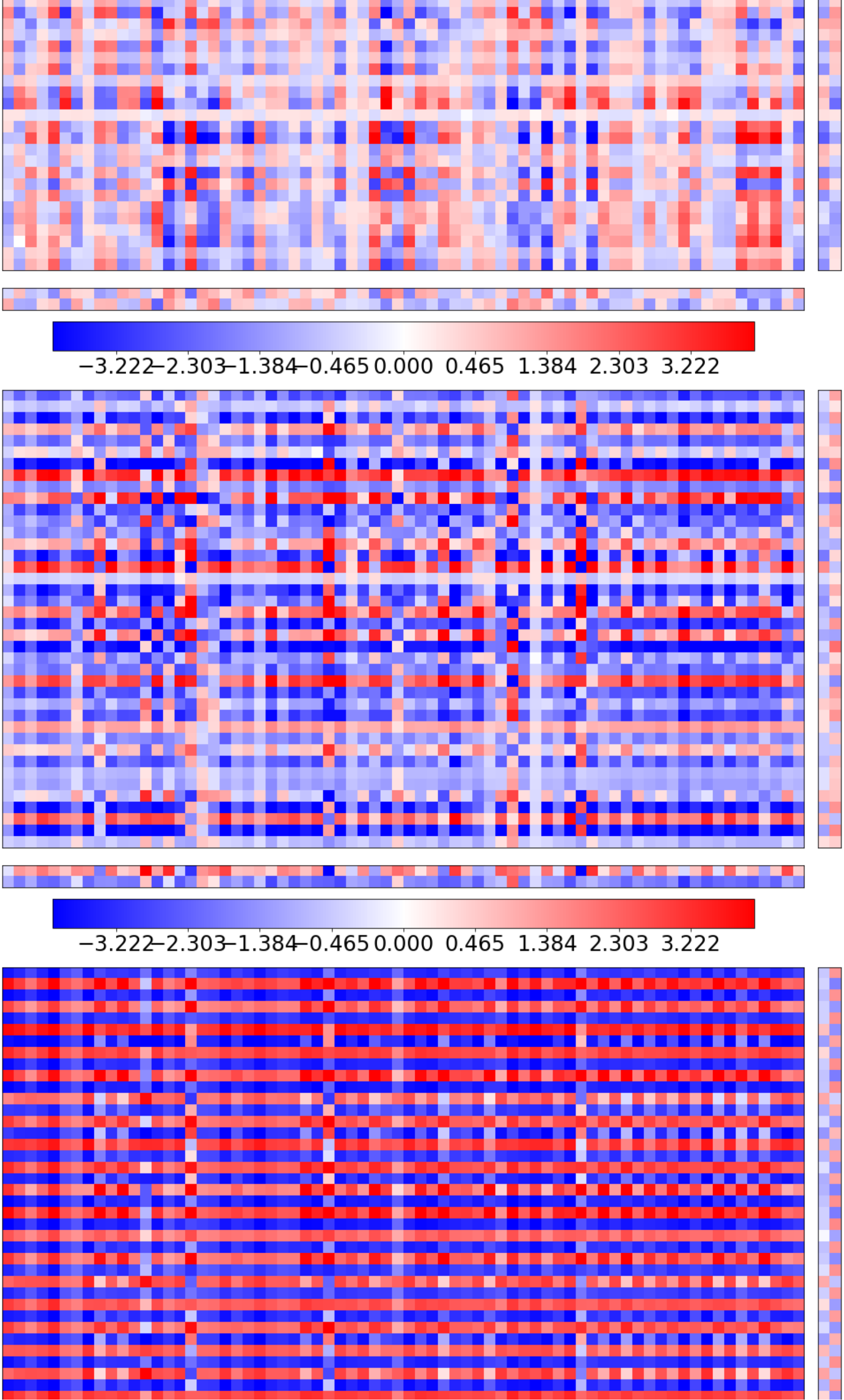
```
example1()
```

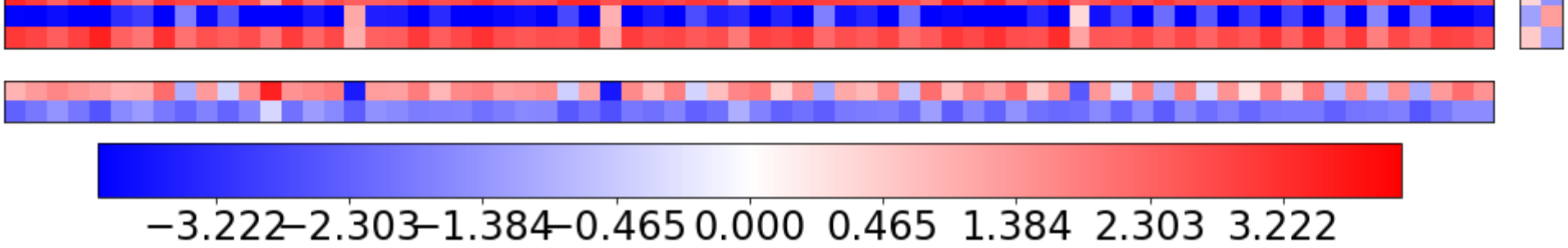


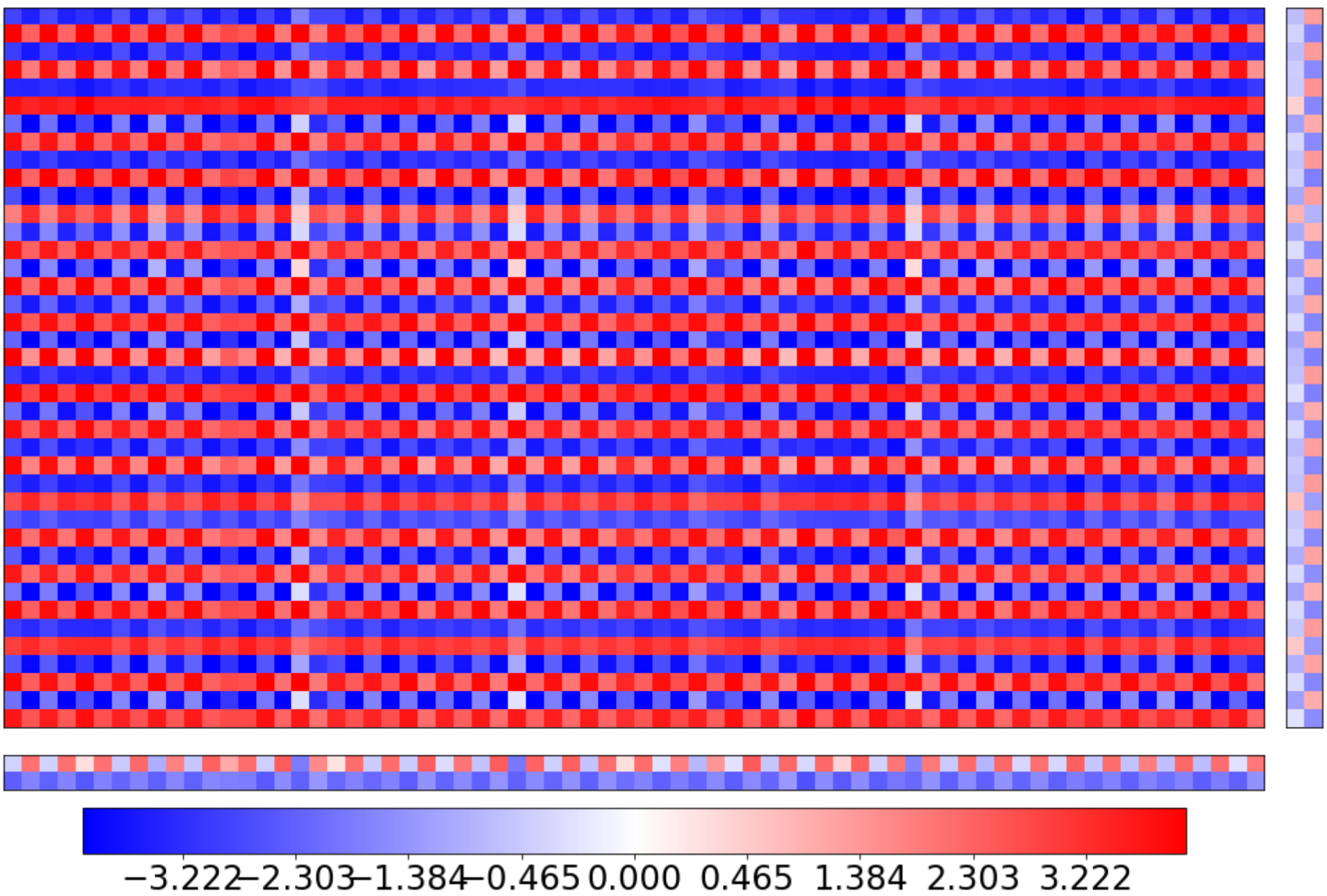


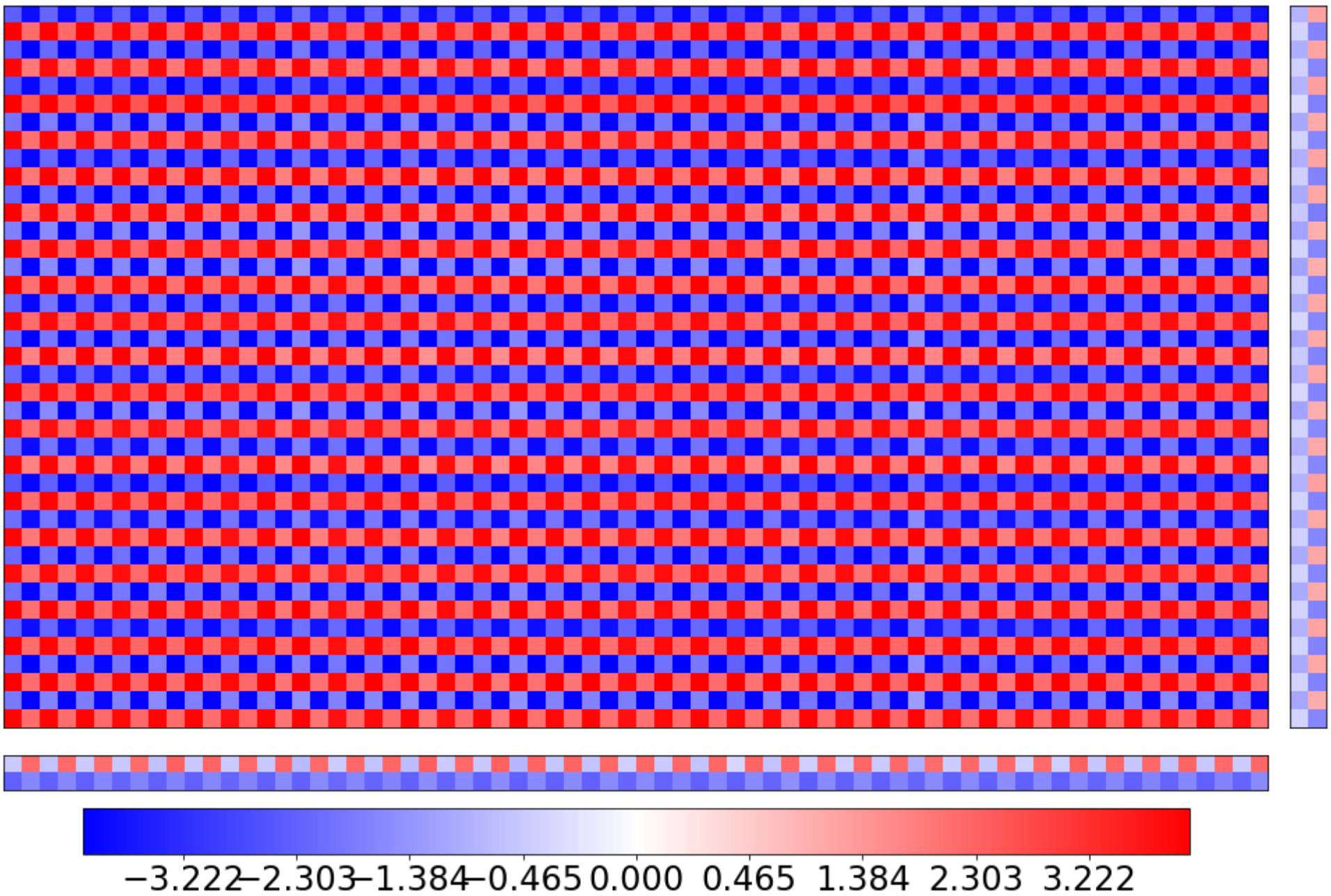

```
/usr/local/lib/python3.6/dist-packages/autograd/tracer.py:48: FutureWarning: `rcond` parameter will change to the default of machine prec
To use the future default and silence this warning we advise to pass `rcond=None`, to keep using the old, explicitly pass `rcond=-1`.
  return f_raw(*args, **kwargs)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

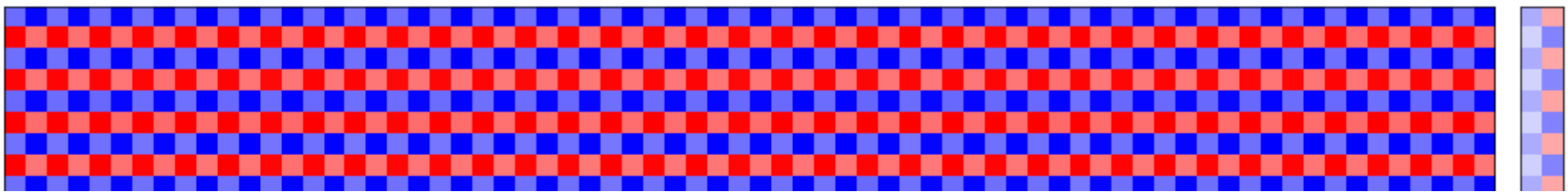


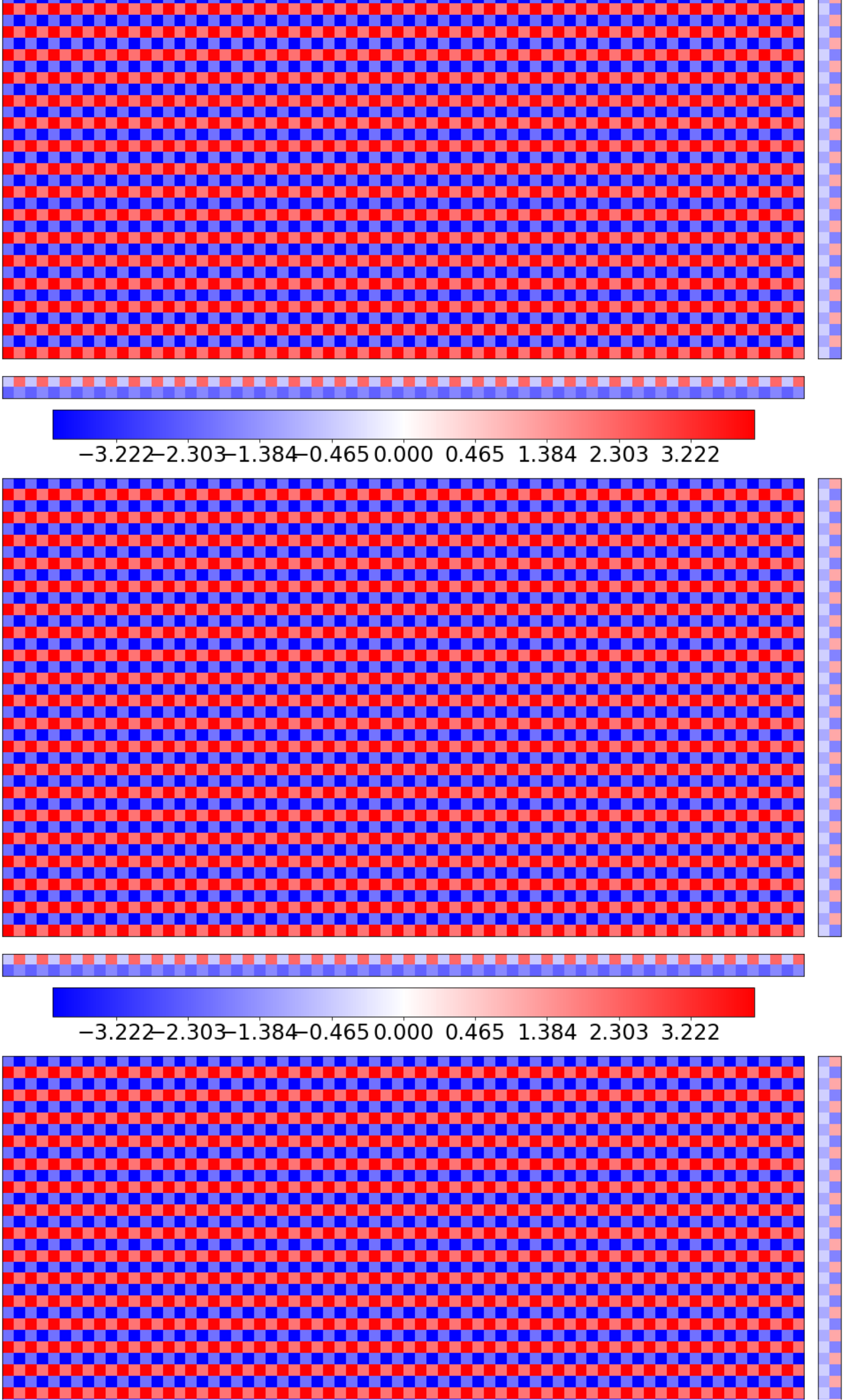

−3.222 −2.303 −1.384 −0.465 0.000 0.465 1.384 2.303 3.222
−3.222 −2.303 −1.384 −0.465 0.000 0.465 1.384 2.303 3.222

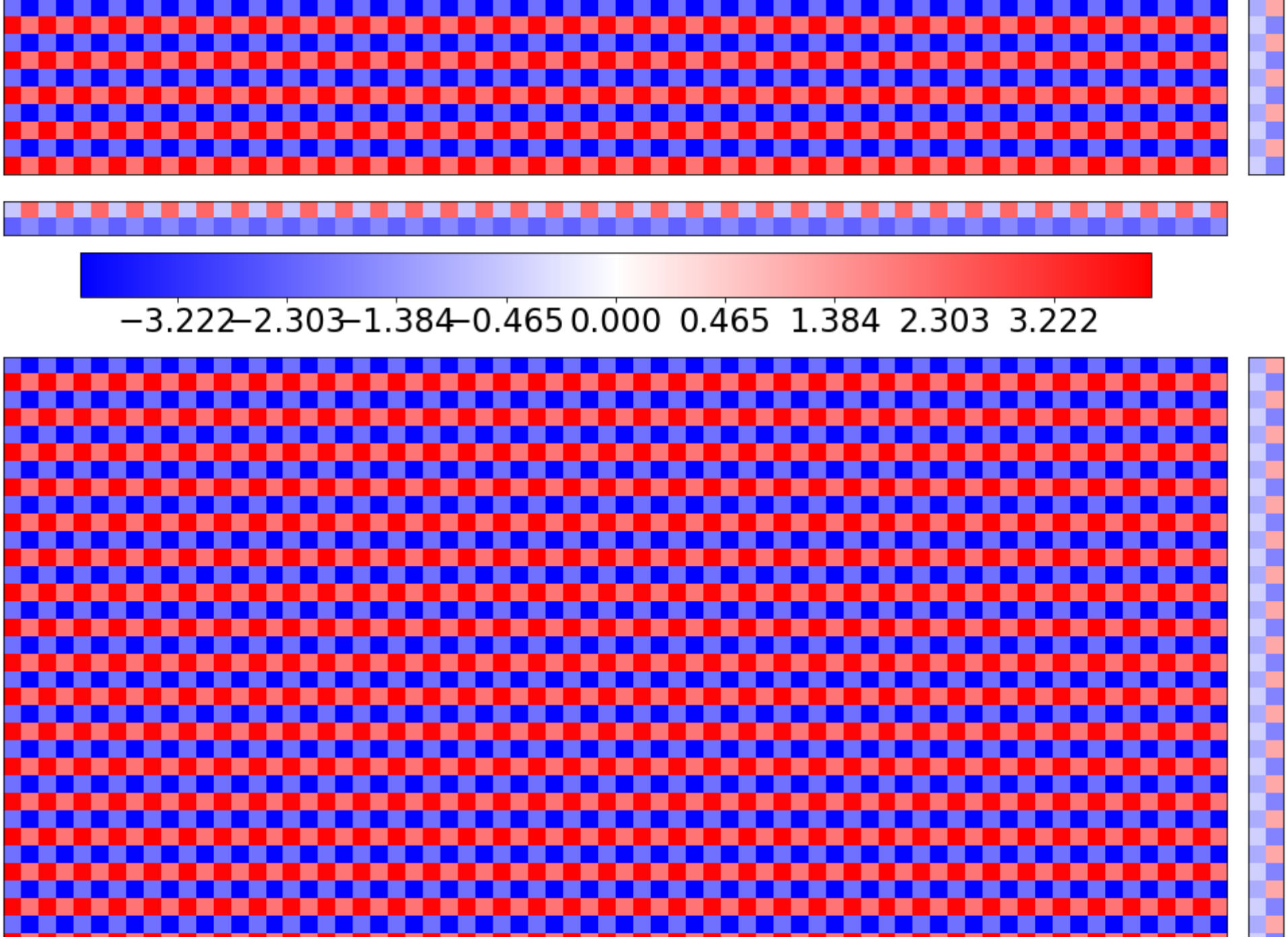

−3.222 −2.303 −1.384 −0.465 0.000 0.465 1.384 2.303 3.222


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


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




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



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.

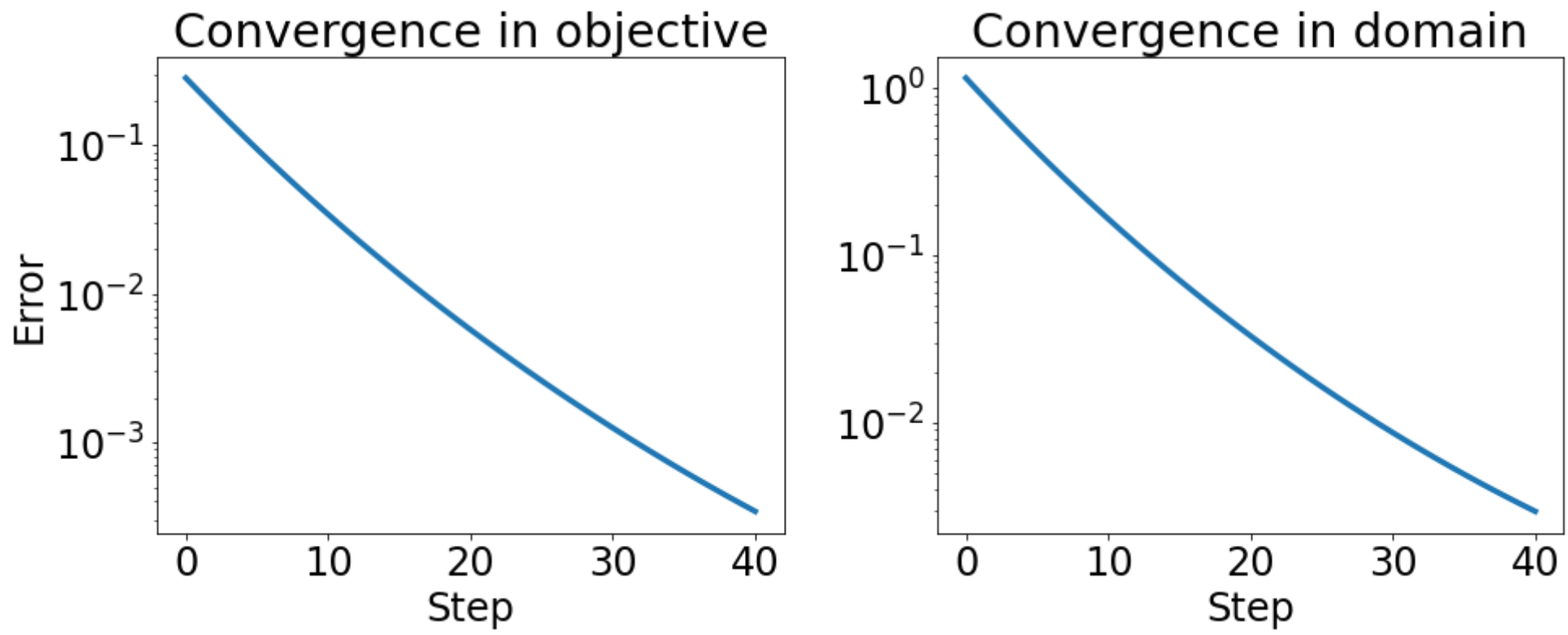
Comparison with gradient descent and nuclear norm projection

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

```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-009f4fd18d9a> in <module>()
      1 n, k = 1000, 10
      2 # random rank-10 matrix normalized to have nuclear norm 1
----> 3 U = np.random.normal(0, 1, (n, k))
      4 U = np.linalg.qr(U)[0]
```

```
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

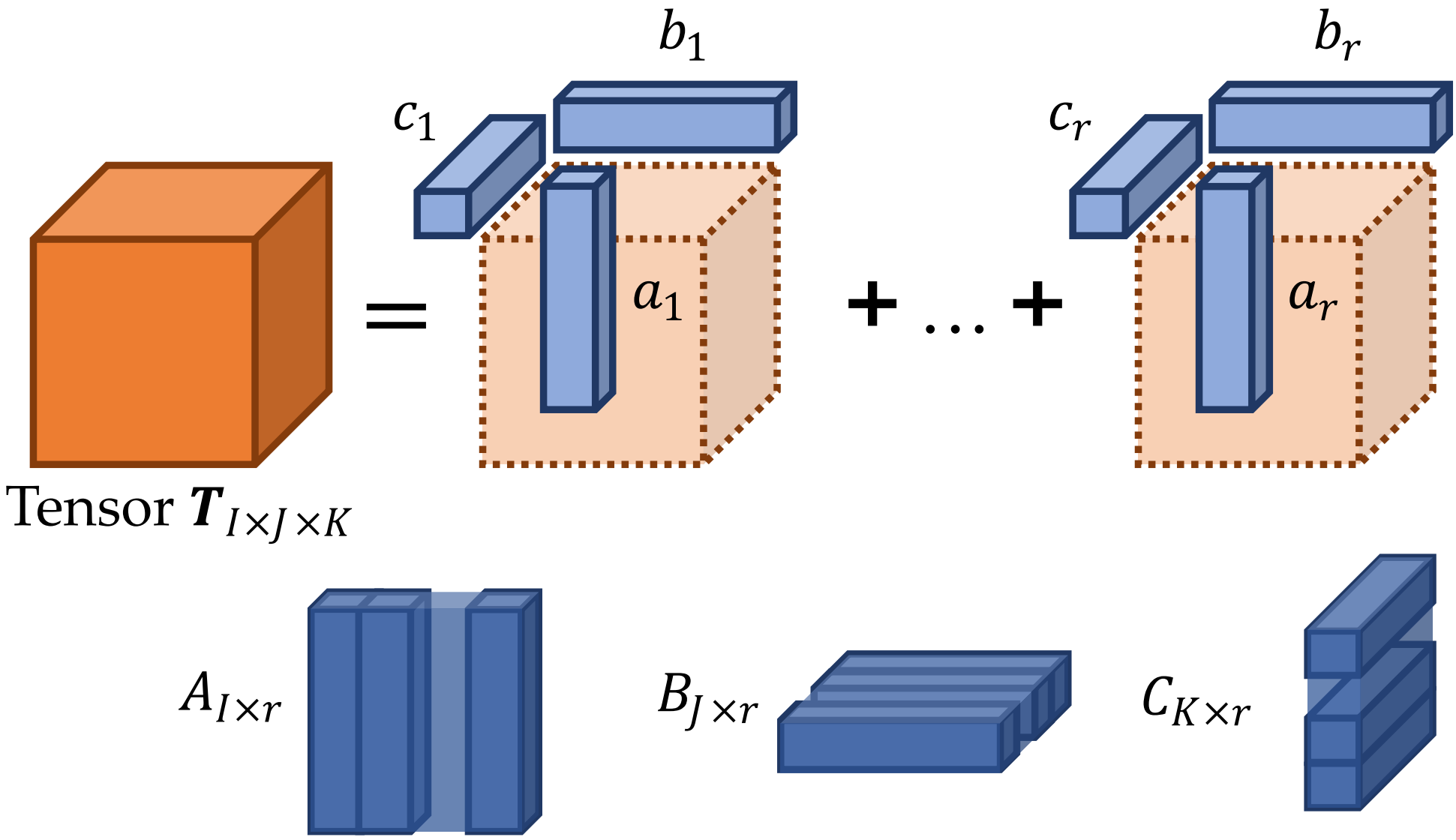
Let us consider a problem of fitting tensor T with the following rank r CP decomposition:

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

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:



This problem can be also formulated in a matrix form

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

Data

Random rank R CP tensor

We will construct tensor T by randomly sampling entries of the factor matrices $A \in \mathbb{R}^{I \times r}, B \in \mathbb{R}^{J \times r}, C \in \mathbb{R}^{K \times r}$ from the standard normal distribution $\mathcal{N}(0, 1)$. Each column of the factor matrices is normalized to unit length, and the constructed tensor is denoted as

$$T = (A, B, C) \cdot I + \mu \frac{\|(A, B, C) \cdot I\|_F}{\|N\|_F} N,$$

where N stands for the 3d Gaussian noise tensor and μ is a small signal-to-noise parameter.

Alternating Least Squares (ALS) algorithm

The basic idea of ALS is to fix all the variables, except one and write down optimal solution, then repeat the same for all the variables.

Input Tensor T, stopping criteria ε, rank r

- Initialize A^0, B^0, C^0 as gaussian random matrices with $\mathcal{N}(0, 1)$ distribution, $\hat{T}^0 = (A^0, B^0, C^0) \cdot I, k = 0$
- **while** $\frac{\|\hat{T}^k - T\|_F}{\|T\|_F} \geq \varepsilon$:
  - $(B^\top)^{k+1} = (C^k \odot A^k)^\dagger \hat{T}^k_{KI \times J}$
  - $(C^\top)^{k+1} = (A^k \odot B^{k+1})^\dagger \hat{T}^k_{IJ \times K}$
  - $(A^\top)^{k+1} = (B^{k+1} \odot C^{k+1})^\dagger \hat{T}^k_{JK \times I}$
  - $\hat{T}^{k+1} = (A^{k+1}, B^{k+1}, C^{k+1}) \cdot I$
  - $k = k + 1$

Return Tensor $\hat{T}^k$ with CP rank r such that $\frac{\|\hat{T}^k - T\|_F}{\|T\|_F} < \varepsilon$

```
!pip install -U tensorly
%pip install wandb -q
```

```
Collecting tensorly
  Downloading https://files.pythonhosted.org/packages/e0/da/2cf86192ab6ed57b3b1c836753df958d8ccd9495ed2de9828f9ff4867629/tensorly-0.5.1.t
     |████████████████████████████████| 112kB 8.6MB/s
Requirement already satisfied, skipping upgrade: numpy in /usr/local/lib/python3.6/dist-packages (from tensorly) (1.19.5)
Requirement already satisfied, skipping upgrade: scipy in /usr/local/lib/python3.6/dist-packages (from tensorly) (1.4.1)
Collecting nose
  Downloading https://files.pythonhosted.org/packages/15/d8/dd071918c040f50fa1cf80da16423af51ff8ce4a0f2399b7bf8de45ac3d9/nose-1.3.7-py3-r
     |████████████████████████████████| 163kB 10.8MB/s
Building wheels for collected packages: tensorly
  Building wheel for tensorly (setup.py) ... done
  Created wheel for tensorly: filename=tensorly-0.5.1-cp36-none-any.whl size=149171 sha256=408840f82c64146ed8d0cd80315d7f6b9c9dcbfc87ff93
  Stored in directory: /root/.cache/pip/wheels/88/1e/e7/b9677b2046cc87e17931b4b5781941786d3ee647825ca40ea6
Successfully built tensorly
Installing collected packages: nose, tensorly
Successfully installed nose-1.3.7 tensorly-0.5.1
     |████████████████████████████████| 2.0MB 8.7MB/s
     |████████████████████████████████| 102kB 9.3MB/s
     |████████████████████████████████| 163kB 37.1MB/s
     |████████████████████████████████| 133kB 36.0MB/s
     |████████████████████████████████| 71kB 8.3MB/s
  Building wheel for pathtools (setup.py) ... done
  Building wheel for subprocess32 (setup.py) ... done
```

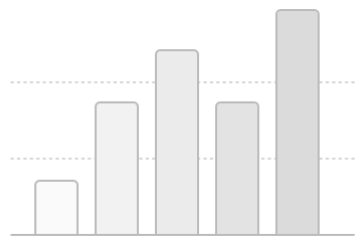
```
# Ignore excessive warnings
import logging
logging.propagate = False
logging.getLogger().setLevel(logging.ERROR)

# WandB – Import the wandb library
import wandb
WANDB_NAME = 'my_als_project'
wandb.init(project=WANDB_NAME)
```

```
wandb: Appending key for api.wandb.ai to your netrc file: /root/.netrc
```

Tracking run with wandb version 0.10.17
Syncing run **stellar-totem-1** to Weights & Biases (Documentation).
Project page: https://wandb.ai/skoltech_optimization/my_als_project
Run page: https://wandb.ai/skoltech_optimization/my_als_project/runs/1nlg3o9m
Run data is saved locally in `/content/wandb/run-20210202_123349-1nlg3o9m`

Run(1nlg3o9m)

Search panels



No metrics logged yet.

Charts will appear here as data is logged.

```
run_parameters = {}
run_parameters['WANDB_NAME']            = 'my_als_project'
run_parameters['WANDB_GROUP']           = 'Debugging runs'
run_parameters['N_EXPERIMENTS']         =  3
run_parameters['SEEDS']                 = range(999, 999-run_parameters['N_EXPERIMENTS'], -1)
run_parameters['DIM']                   =  30
run_parameters['RANK']                  =  20
run_parameters['MODE']                  =  'random'
run_parameters['REGULARIZATION_COEF']   =  0
run_parameters['NOISE']                 = 1e-5
run_parameters['N_ITER']                =  500
run_parameters['LIST_OF_METHODS']       =  ['ALS']
```