凸优化大作业

> 计73 陈海天 2016010106
>
> 计75 李阳崑 2017011235

第1题

(a)

(a) Solving (1) by calling mosek and gurobi directly in Matlab or python. The package "CVX" is not allowed to use here. Compare the performance between the simplex methods and interior point methods.

(a.1) 随机生成数据

运行 `generate_random_data.py` 生成随机的 C, μ, ν，再运行 `gurobi_computeEMD.py` 得到结果。

gurobi中三种方法的运行结果如下（C, μ, ν 在 `random_data` 文件夹下，与老师所给的 `test_data` 格式保持一致）：

	CPU time (s)	iters	result
primal simplex	0.00	31	2.406049817e+00
dual simplex	0.00	13	2.406049817e+00
barrier	0.01	6	2.406049817e+00

(a.2) shift_of_Ricker数据

和随机数据的计算过程一样，也是运行 `gurobi_computeEMD.py` 得到结果。两种单纯形法的耗时差不多，内点法的耗时稍长；迭代轮数是内点法最少，对偶单纯形法次之，原始单纯形法最多。已知标准答案为 4， 4 和算法结果之差就是最后的结果误差，三种算法误差在同一个数量级（ 10^{-7} ）；具体来说，对偶单纯形法的误差大于原始单纯形法和内点法。

	CPU time (s)	iters	result	err
primal simplex	0.15	2310	3.999999523e+00	4.77e-7
dual simplex	0.15	1192	3.999999460e+00	5.40e-7
barrier	0.20	178	3.999999526e+00	4.74e-7

(a.3) DOTmark数据集

需要将计算两张图片A和B之间的Wasserstein距离（Earth Mover's Distance, EMD）转化为题中所给的问题。做法如下：

对于图A中的像素点 (x_1, y_1) 和图B中的像素点 (x_2, y_2)，取二者的欧式距离为cost matrix的元素。因此，对于大小为 $n \times n$ 的两张图片，$C \in \mathbb{R}^{n^2 \times n^2}$ 。所搬运的像素量不能超过该点的像素值，因此图A中每个像素点的像素值就是 $\mu \in \mathbb{R}^{n^2}$ ，图B中每个像素点的像素值就是 $\nu \in \mathbb{R}^{n^2}$ 。

实现代码在 `gurobi_DOTmark.py` 中，我们首先用 `init_cost_mat` 和 `init_cost_dict` 两个函数初始化cost matrix。因为cost matrix只和图片大小有关，因此我们用 `init_cost_mat` 和 `init_cost_dict` 将其算好后存下来（ `cost_dict_size_32.pkl` 和 `cost_mat_size_32.npy` ），之后计算EMD时可直接调用。

计算给定两张图片的EMD由 `compute1EMD` 函数完成，该函数接收三个参数：

- mtd: 使用什么优化方法，0是primal simplex, 1是dual simplex, 2是barrier
- num1: 图片A的编号（我们用0-9代表数据集文件名中1001-1010的编号）
- num2: 图片B的编号

按照gurobi的规范实现即可。需要注意的是像素点的值是正整数，而 μ 和 ν 是概率分布，因此将像素值转换成 μ 和 ν 时需要归一化。

我们在本地测试了 `GRFrough` 数据集、图片大小为 32×32 的结果，两两之间计算EMD，共有45组结果，放在 `results/gurobi_GRFrough_32.xlsx` 中，截图显示如下：

primal simplex					dual simplex					barrier				
file1	file2	CPU time	iterations	result	file1	file2	CPU time	iterations	result	file1	file2	CPU time	iterations	result
0	1	2.18	47811	8.97E-01	0	1	1.56	13169	8.97E-01	0	1	7.69	834	8.97E-01
0	2	1.87	52569	7.23E-01	0	2	1.41	8775	7.23E-01	0	2	6.58	835	7.23E-01
0	3	2.10	55757	9.96E-01	0	3	1.61	14013	9.96E-01	0	3	7.23	662	9.96E-01
0	4	2.11	50632	6.81E-01	0	4	1.42	8354	6.81E-01	0	4	7.58	899	6.81E-01
0	5	1.96	48586	8.63E-01	0	5	1.38	9919	8.63E-01	0	5	7.62	705	8.63E-01
0	6	1.93	58019	6.47E-01	0	6	1.43	7668	6.47E-01	0	6	6.80	830	6.47E-01
0	7	1.86	54848	6.79E-01	0	7	1.82	7872	6.79E-01	0	7	7.28	769	6.79E-01
0	8	1.85	44884	5.61E-01	0	8	1.41	8113	5.61E-01	0	8	7.17	677	5.61E-01
0	9	1.68	41986	6.93E-01	0	9	1.40	9204	6.93E-01	0	9	6.92	562	6.93E-01
1	2	2.27	66340	1.04E+00	1	2	1.63	13437	1.04E+00	1	2	7.81	733	1.04E+00
1	3	2.12	66690	9.96E-01	1	3	1.54	12398	9.96E-01	1	3	7.82	635	9.96E-01
1	4	2.07	63904	5.95E-01	1	4	1.42	7611	5.95E-01	1	4	7.17	917	5.95E-01
1	5	2.13	64533	9.90E-01	1	5	1.71	12749	9.90E-01	1	5	7.58	919	9.90E-01
1	6	2.15	66871	1.01E+00	1	6	1.53	12685	1.01E+00	1	6	7.50	713	1.01E+00
1	7	2.04	66170	8.53E-01	1	7	1.43	10348	8.53E-01	1	7	6.48	599	8.53E-01
1	8	2.19	64095	6.88E-01	1	8	1.41	9601	6.88E-01	1	8	7.58	649	6.88E-01
1	9	1.95	57478	6.64E-01	1	9	1.42	8285	6.64E-01	1	9	7.18	940	6.64E-01
2	3	2.02	55984	9.52E-01	2	3	1.47	12439	9.52E-01	2	3	7.96	674	9.52E-01
2	4	1.85	49601	9.13E-01	2	4	1.40	11922	9.13E-01	2	4	6.85	748	9.13E-01
2	5	2.04	61689	6.83E-01	2	5	1.43	9291	6.83E-01	2	5	6.56	586	6.83E-01
2	6	1.94	57694	6.70E-01	2	6	1.40	8305	6.70E-01	2	6	7.60	653	6.70E-01
2	7	1.88	46658	8.10E-01	2	7	1.37	9180	8.10E-01	2	7	7.28	629	8.10E-01
2	8	1.88	48564	8.73E-01	2	8	1.44	10843	8.73E-01	2	8	7.72	967	8.73E-01
2	9	1.95	50505	7.85E-01	2	9	1.53	11123	7.85E-01	2	9	7.32	628	7.85E-01
3	4	1.84	52256	1.14E+00	3	4	1.78	17127	1.14E+00	3	4	8.06	834	1.14E+00
3	5	1.98	61364	8.74E-01	3	5	1.39	9531	8.74E-01	3	5	7.69	1163	8.74E-01
3	6	2.05	64430	1.12E+00	3	6	1.60	14602	1.12E+00	3	6	7.97	674	1.12E+00
3	7	1.86	54136	1.17E+00	3	7	1.45	13764	1.17E+00	3	7	7.35	724	1.17E+00
3	8	1.89	55445	1.10E+00	3	8	1.60	14321	1.10E+00	3	8	7.50	1066	1.10E+00
3	9	1.90	58748	1.01E+00	3	9	1.93	17219	1.01E+00	3	9	7.85	883	1.01E+00
4	5	2.30	74238	9.36E-01	4	5	1.54	12411	9.36E-01	4	5	7.10	847	9.36E-01
4	6	2.23	72545	7.82E-01	4	6	1.42	10180	7.82E-01	4	6	7.16	701	7.82E-01
4	7	2.09	66787	6.36E-01	4	7	1.40	8715	6.36E-01	4	7	7.79	691	6.36E-01
4	8	1.94	58825	5.28E-01	4	8	1.48	6628	5.28E-01	4	8	6.20	658	5.28E-01
4	9	2.01	52488	5.02E-01	4	9	1.34	6872	5.02E-01	4	9	6.37	532	5.02E-01
5	6	1.89	51651	9.07E-01	5	6	1.69	13415	9.07E-01	5	6	6.80	926	9.07E-01
5	7	1.98	62686	9.86E-01	5	7	1.76	14171	9.86E-01	5	7	6.99	578	9.86E-01
5	8	1.83	53761	8.95E-01	5	8	1.67	14195	8.95E-01	5	8	7.50	939	8.95E-01
5	9	1.86	50265	7.61E-01	5	9	1.46	10639	7.61E-01	5	9	6.89	651	7.61E-01
6	7	1.71	46866	5.83E-01	6	7	1.55	7443	5.83E-01	6	7	6.77	667	5.83E-01
6	8	1.91	54230	6.55E-01	6	8	1.50	8861	6.55E-01	6	8	7.97	824	6.55E-01
6	9	1.91	49874	8.74E-01	6	9	1.36	10959	8.74E-01	6	9	7.64	969	8.74E-01
7	8	1.88	50207	6.27E-01	7	8	1.37	7608	6.27E-01	7	8	7.12	621	6.27E-01
7	9	1.94	53142	7.94E-01	7	9	1.59	9842	7.94E-01	7	9	7.04	902	7.94E-01

从result一列可以看到，原始单纯形法、对偶单纯形法和障碍法都能让该问题收敛。两张图像的EMD大致都在 0.5 到 1.2 的范围内。

从迭代轮数上看，原始单纯形法需要的迭代轮数最多，需要约 4 万 - 7.5 万轮。对偶单纯形法次之，需要 1 万轮左右。障碍法所需轮数最少，大部分情况下 1000 轮之内就能收敛。

从CPU时间上来看，对偶单纯形法速度最快，约 1.5 秒计算出结果；原始单纯形法次之，需要 2 秒左右；障碍法耗时最长，需要 7 到 8 秒。

从以上结果可知，障碍法每轮的计算量大、耗时长，故即使迭代轮数少，总耗时量也大。原始单纯形法每轮计算量最小。从总耗时来讲，对偶单纯形法解本问题最快。

需要指出的是，以上的CPU耗时是gurobi自动输出的结果，只包括了 `optimize()` 函数的运行时间。在执行 `optimize()` 函数之前，还有初始化模型的过程，包括初始化待优化变量 `x` 、初始化约束条件和初始化目标函数：

```
model = gp.Model(model_name)
model.Params.Method = mtd  # 0: primal simplex, 1: dual simplex, 2: barrier, 3: concurrent
x = model.addVars(coord, vtype=GRB.CONTINUOUS, name="x")
row_sum = model.addConstrs( (x.sum(i, '*') == mu[i] for i in range(COST_SIZE)), name='row_sum')
col_sum = model.addConstrs( (x.sum('*', j) == nu[j] for j in range(COST_SIZE)), name='col_sum')
model.setObjective(x.prod(val), GRB.MINIMIZE)
```

这个过程在本数据集上大概会消耗 45 秒时间。

(b)

(b.1) ADMM算法实现

理论推导

我们考虑的最优传输问题的定义如下：

$$\min_X \sum_{i,j} C_{i,j}X_{i,j},\ s.t.\ X\mathbf{1}_n = \mu,\ X^T\mathbf{1}_n = \nu,\ X \geqslant 0$$

其中 $X, C \in \mathbb{R}^{n\times n}$，$\mu, \nu \in \mathbb{R}^n$，我们将 X, C 视为维数 n^2 的向量，可将上问题转化为标准的线性规划问题：

$$\min_x c^T x,\ s.t.\ Ax = b,\ x \geqslant 0$$

其中 x, c 为 X, C 的向量表示形式，$b^T = (\mu^T, \nu^T)$，$A \in \mathbb{R}^{2n\times n^2}$，$A$ 的具体形式为：

$$\begin{pmatrix} \mathbf{1}_n & 0 & \cdots & 0 \\ 0 & \mathbf{1}_n & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{1}_n \\ I_n & I_n & \cdots & I_n \end{pmatrix}$$

该线性规划问题的对偶问题为：

$$\max_y b^T y,\ s.t.\ A^T y + s = c,\ s \geqslant 0$$

其中 $y \in \mathbb{R}^{2n}$，$s \in \mathbb{R}^{n^2}$，写出对偶问题的增广拉格朗日函数

$$L_t(x, y, s) = -b^T y + x^T(A^T y + s - c) + \frac{t}{2}||A^T y + s - c||_2^2$$

基于ADMM的迭代规则，我们可以显式地写出每一个迭代步骤：

$$y^{k+1} = arg\min_y L_t(y; x^k, s^k) = -(AA^T)^-((Ax^k - b)/t + A(s^k - c))$$
$$s^{k+1} = arg\min_s L_t(s; x^k, y^{k+1}) = \max\{0, c - A^T y^{k+1} - x^k/t\}$$
$$x^{k+1} = x^k + t(A^T y^{k+1} + s^{k+1} - c)$$

因为 A 不是满秩，所以在 y 的更新中使用的是 AA^T 的伪逆 $(AA^T)^-$。

具体实现

我们使用python3实现了ADMM算法，矩阵相关运算使用的是numpy库。

理论推导的形式将问题转换为了标准的线性规划问题，但是 $A \in \mathbb{R}^{2n\times n^2}$ 维数太大且只有 $2n^2$ 个非零值，具体实现的过程中不应该直接使用 A 的原始值进行运算。为此我们将与 A 相关的运算转换为更加高效的操作。具体的转换和对应代码如下：

- A 乘向量化的矩阵，将其转换为对原始矩阵的操作：

$$Ax = \begin{pmatrix} X\mathbf{1}_n \\ X^T\mathbf{1}_n \end{pmatrix}$$

```
def A_mul(X: np.ndarray):
    return X.sum(axis=1), X.sum(axis=0)
```

- A 转置乘 y，将 y 视为两个 n 维向量 p, q 的拼接：

$$A^Ty = \begin{pmatrix} p_1+q_1 & p_1+q_2 & \cdots & p_1+q_n \\ p_2+q_1 & p_2+q_2 & \cdots & p_2+q_n \\ \vdots & \vdots & \ddots & \vdots \\ p_n+q_1 & p_n+q_2 & \cdots & p_n+q_n \end{pmatrix} = (p,p,\cdots,p) + (q,q,\cdots,q)^T$$

```
def AT_mul(y1: np.ndarray, y2: np.ndarray, n: int):
    return y1.reshape(-1,1)+y2.reshape(1,-1)
```

- AA^T 的伪逆，如果运算求解会带来非常明显的精度问题。我们观察了 AA^T 及其伪逆的数值结果，可以验证 AA^T 及其伪逆的具体形式可以直接如下表示：

$$AA^T = \begin{pmatrix} nI_n & \mathbf{1}_{n\times n} \\ \mathbf{1}_{n\times n} & nI_n \end{pmatrix}$$
$$(AA^T)^- = \frac{1}{4n^2}\begin{pmatrix} 4nI_n - 3_{n\times n} & 1_{n\times n} \\ 1_{n\times n} & 4nI_n - 3_{n\times n} \end{pmatrix}$$

那么 $(AA^T)^-$ 与 y 相乘的运算可简化为

$$(AA^T)^-y = \frac{1}{4n^2}\begin{pmatrix} 4np + (\sum q_i - 3\sum p_i)\mathbf{1}_n \\ 4nq + (\sum p_i - 3\sum q_i)\mathbf{1}_n \end{pmatrix}$$

```
# 迭代更新y的操作
y1  = (x1  - mu)/t + s1 - c1
y2  = (x2  - nu)/t + s2 - c2
y1s = y1.sum()
y2s = y2.sum()
y11 = 4*n*y1 - 3*y1s + y2s
y22 = 4*n*y1 - 3*y2s + y1s
```

GPU加速

由于直接使用CPU计算速度较慢，我们将代码改成了pytorch的版本，将所有的numpy操作转换成了pytorch中对应的操作。

(b.2) 实验设置

本实验 numpy 和 torch 的随机种子皆设为 2333 。

ADMM算法有 1 个超参数 t ，我们将在 **b.4** 节中介绍我们搜索超参的情况， **b.3** 节中我们任选 $t = 0.1$ 。

另外，从 **b.4** 节的实验过程中我们观察到ADMM每步迭代距离（ x^k 和 x^{k+1} 的距离）会有较大波动，在其收敛之前可能出现 $\Delta x < 10^{-12}$ 的情况，因此我们在本实验中不使用每步迭代距离作为判停方法，而是固定迭代轮数。对 **b.3** 节小数据集，我们迭代 1 万轮；对 **b.4** 和 **b.5** 节中的实验，我们迭代 100 万轮。

(b.3) 随机生成数据

用 **1.a** 中相同的随机数据（ `random_data` 文件夹下）进行实验，目的是为了初步测试算法的正确性。

运行 `admm_computeEMD_torch.py` 可得结果。

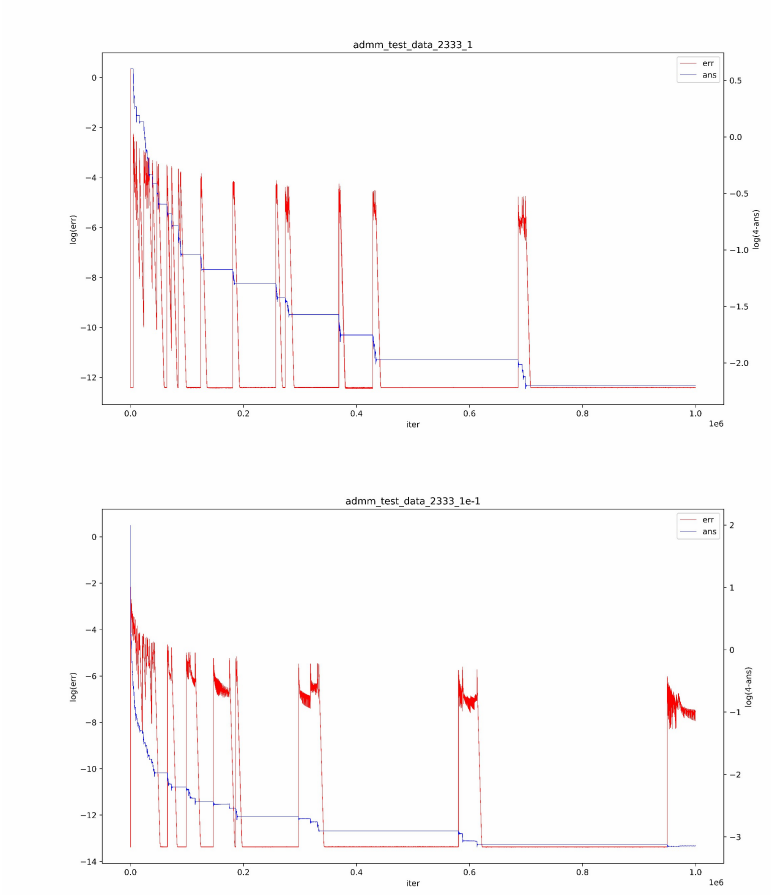
观察到迭代步长在 6140 轮时降至 0 （即低于float32精度），将此时的结果作为收敛结果：

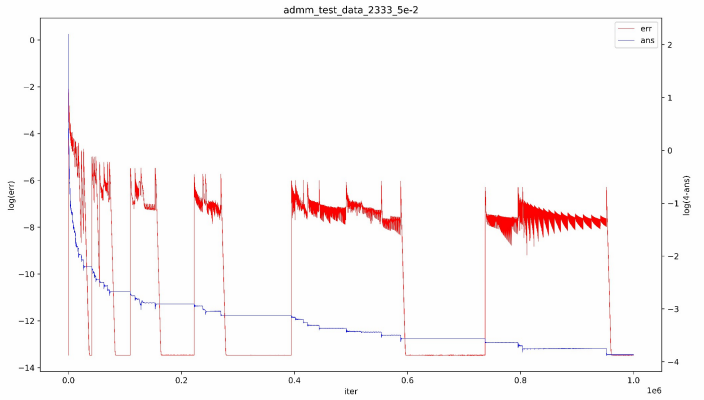
	CPU time (s)	iters	result
ADMM	6.14	6140	2.4060498166914988

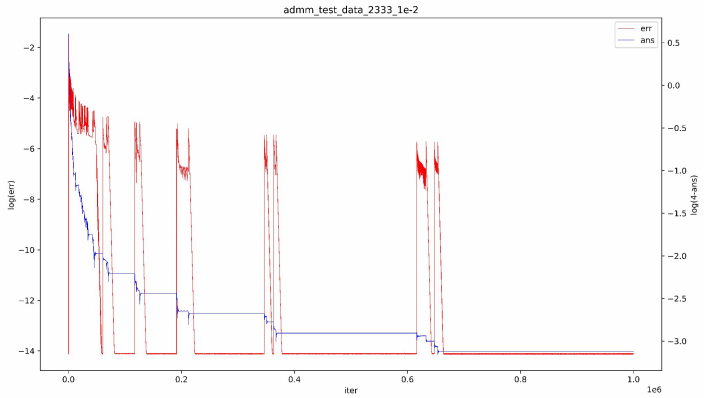
参照 **a.1** 节中gurobi算出的结果，可知算法收敛，且精度更高。

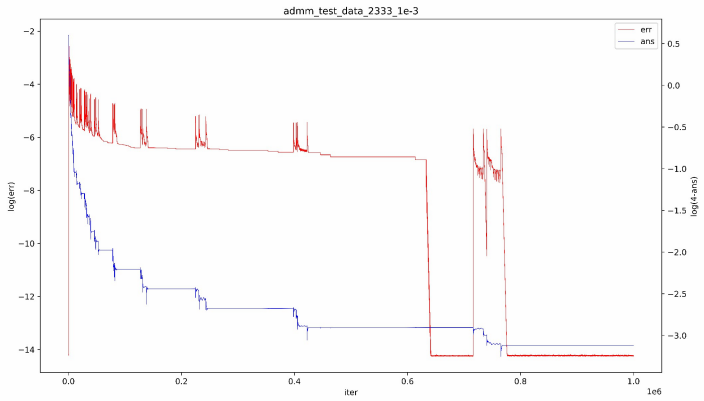
(b.4) shift_of_Ricker数据

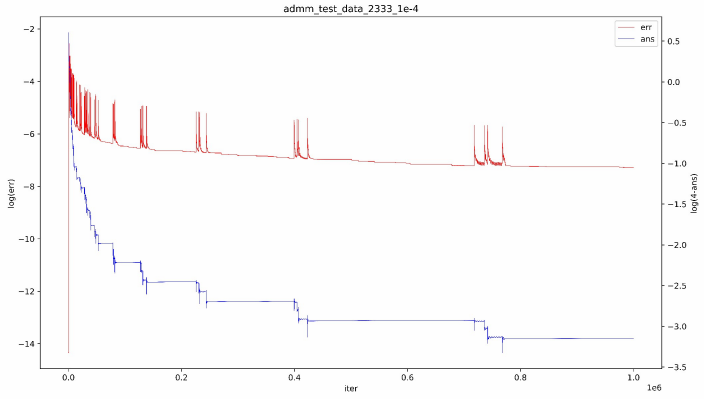
运行 `admm_computeEMD_torch.py` 可得结果。由于已知问题答案为 4 ，我们迭代 100 万轮，每 10 轮打印出迭代步长和当下结果与 4 的差值（都取对数），取不同的 t 绘制图线：

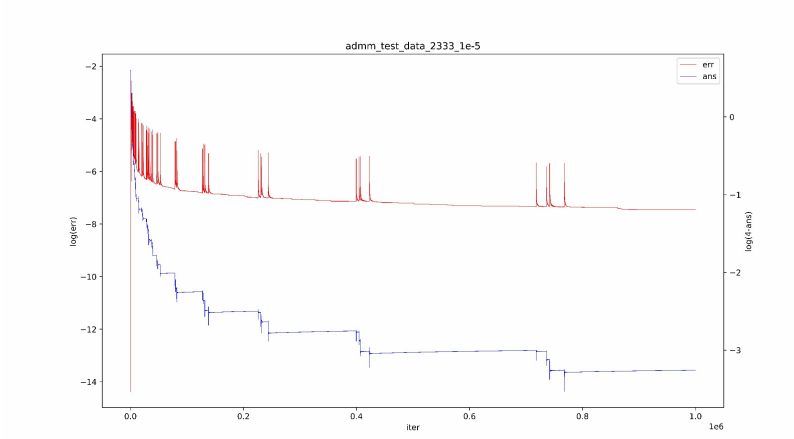


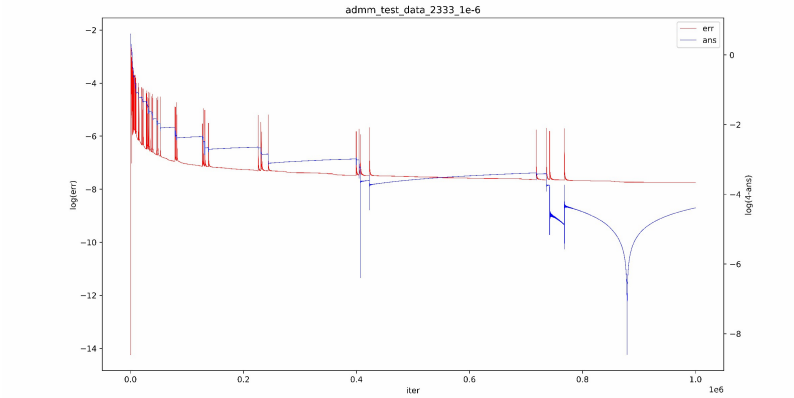

admm_test_data_2333_5e-2
err
ans
log(err)
log(4-ans)
iter
1e6

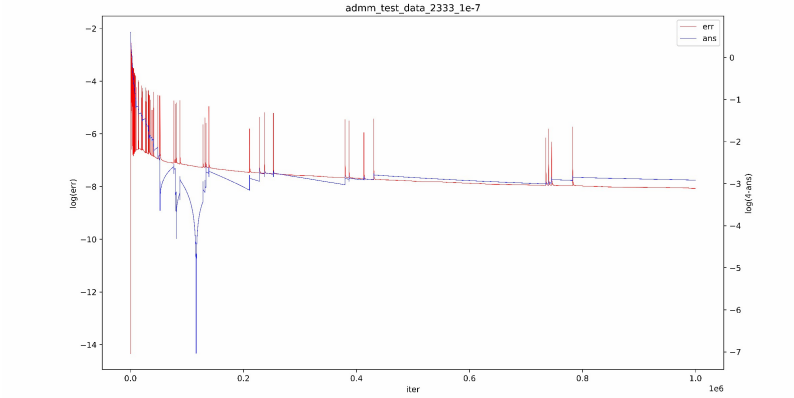

admm_test_data_2333_1e-2
err
ans
log(err)
log(4-ans)
iter
1e6

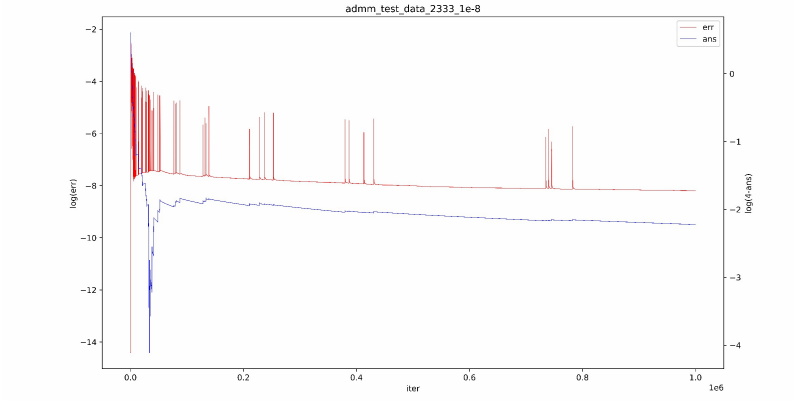

admm_test_data_2333_1e-3
err
ans
log(err)
log(4-ans)
iter
1e6

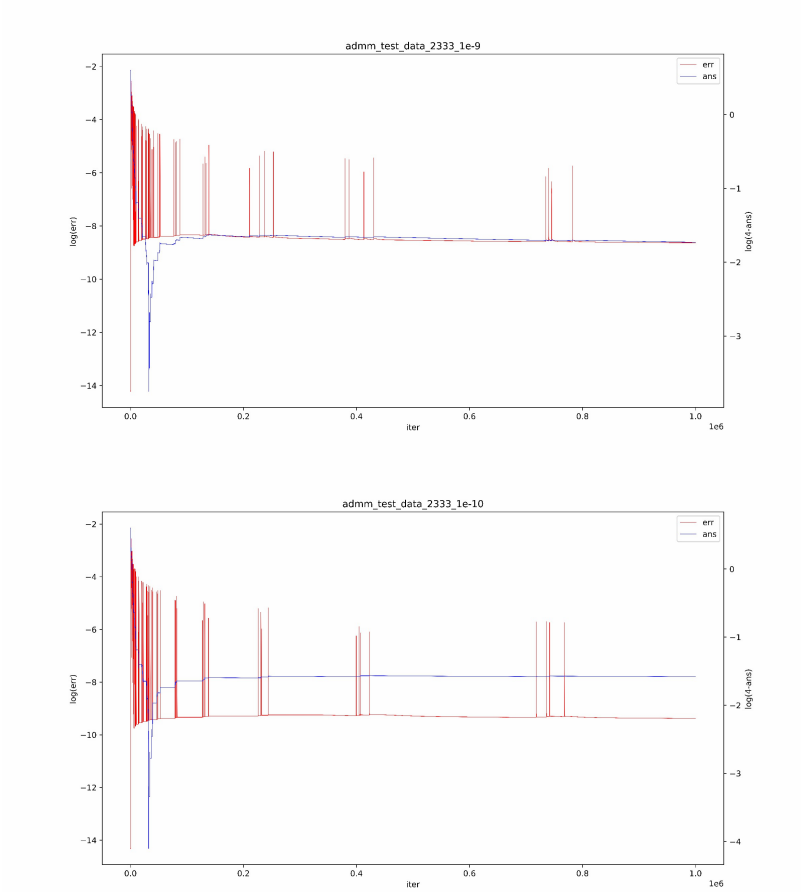

admm_test_data_2333_1e-4
err
ans
log(err)
log(4-ans)
iter
1e6


admm_test_data_2333_1e-5
err
ans
log(err)
log(4-ans)
iter
1e6


admm_test_data_2333_1e-6
err
ans
log(err)
log(4-ans)
iter
1e6


admm_test_data_2333_1e-7
err
ans
log(err)
log(4-ans)
iter
1e6


admm_test_data_2333_1e-8
err
ans
log(err)
log(4-ans)
iter
1e6



从左到右、从上到下 t 依次取 $\{1, 0.1, 0.05, 0.01, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}, 10^{-7}, 10^{-8}, 10^{-9}, 10^{-10}\}$ 共 12 个值。

从图中我们看到，红线（迭代步长）会有一个“降至 10^{-12} 以下 $\longrightarrow$ 保持低位 $\longrightarrow$ 跃升至 10^{-6} 以上 $\longrightarrow$ 高位波动 $\longrightarrow$ ……”的循环。而每次红线跃升时，就是蓝线（当前目标函数值与标准值之差）下降之时。并且随着 t 减小，循环周期变长，即相当于在横向上被“拉长”。

蓝线在 t 较小（小于等于 10^{-6}）时可看到一个深谷，且对随着t减小，深谷左移，即相当于在横向上被“压缩”。

根据这两个性质，我们希望蓝线的深谷能落在我们的迭代过程内，这也是我们 **b.5** 问取 $t = 5 \times 10^{-6}$ 的原因。过小的 t 更新过程过快，反而使算法表现变差。

(b.5) DOTmark数据集

根据 **b.4** 节对 t 对ADMM表现影响的分析，我们希望蓝线的第 1 个深谷落在第 100 万轮内，因此我们取 10^{-6} 和 10^{-7} 的中间值：$t = 5 \times 10^{-6}$ 来进行DOTmark数据集上的实验。

从实验结果来看，45组数据中的每一组都很好地收敛。

第2题

(a) 问题描述

我们考虑参考文献1中第4章所提到的熵正则化的Kantorovich最优传输问题。

Kantorovich最优传输问题是一个特殊的线性规划问题，所以应用先进的线性规划问题都可以进行求解。但是面对大规模问题时，常用的线性规划算法（例如内点法）都有较大的局限性。为此我们引入熵正则化的方法来近似求解。熵的具体定义如下所示：

$$H(X) = -\sum_{i,j} X_{i,j}(\log(X_{i,j}) - 1)$$

此时的最优传输问题为：

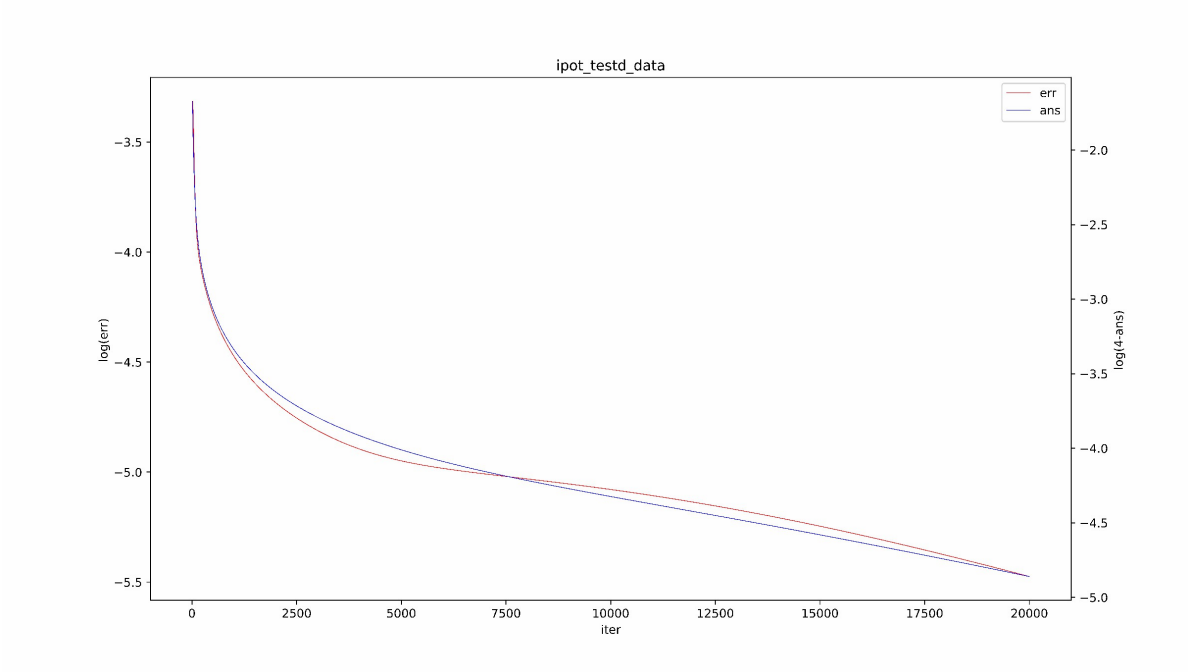
$$\min_X \sum_{i,j} C_{i,j}X_{i,j} - \epsilon H(X),\ s.t.\ X\mathbf{1}_n = \mu,\ X^T\mathbf{1}_n = \nu,\ X \geqslant 0$$

(b) 实现算法

我们实现了参考文献中的IPOT算法，代码的主要部分如下：

```
while itr < 20000:
    itr += 1
    Q = G*X
    for _ in range(L):
        a = mu / np.matmul(Q,b)
        b = nu / np.matmul(Q.T,a)
    X = (Q*b).T*a
```

在test_data数据集上进行测试，迭代 20000 轮，我们得到了如下结果：



可看到与我们实现的ADMM相比，IPOT算法收敛得快得多也稳定得多，20000 轮时就将与结果得误差缩小至 10^{-5} 数量级。