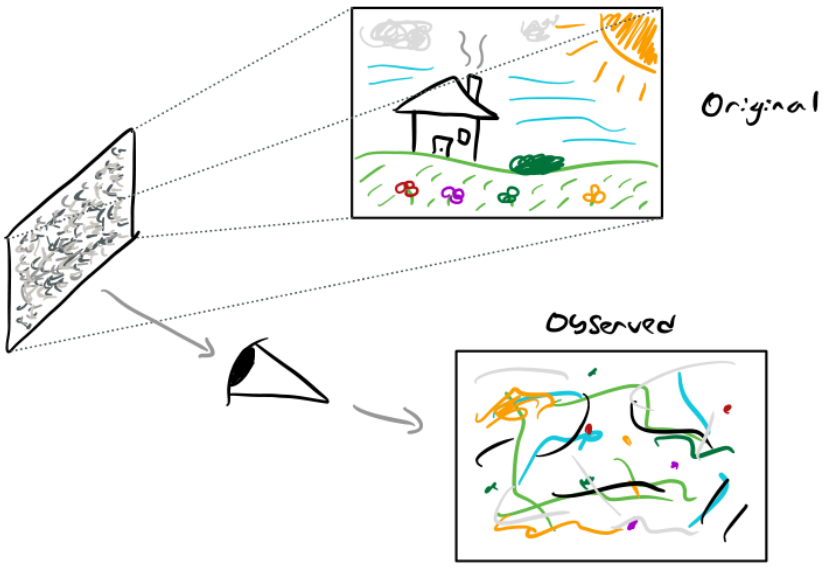
ROBUST "SPARKLE VISION" VIA SINKHORN DIVERGENCE

NICOLAS BOLLE

1. The problem

1.1. **Sparkle Vision.** I'll consider a modification of the problem posed in [33] (solved by the "Sparkle Vision" algorithm) of reproducing an image that has been reflected off of a complicated reflective surface (such as a wall of glitter):



For simpler reflective surfaces, like warped mirrors, it's possible to undistort the image by hand: hold up a checkerboard pattern and manually grow/shrink areas in some image software until the image looks normal. But when the scrambling is more extreme it's infeasible to do it manually, so we need to use a large set of calibration images and some algorithm to infer the distortion. If we assume the incoming image is $d_1 \times d_2$ pixels given by a vector $x \in \mathbb{R}^D$ ($D := d_1 d_2$) and the distorted image is $d_1' \times d_2'$ given by a vector $y \in \mathbb{R}^{D'}$ ($D' = d_1' d_2'$), then the distortition is given by a linear map $y = Bx$, $B \in \mathbb{R}^{D' \times D}$. Using the standard basis images e_i gives a way of determining B, but there are two issues: 1. we'd like a way of leveraging an overcomplete basis, for robustness to noise, and 2. we would like to have the option of using more complicated calibration images, such as frames of a moving scene.

This already gives an idea of how to estimate B, but let's introduce the general framework: we'll frame it as a Procrustes problem, where the distorted images are given by $Y \in \mathbb{R}^{D' \times N}$, the undistorted images are given by $X \in \mathbb{R}^{D \times N}$, and we are looking for a matrix $A \in \mathbb{R}^{D \times D'}$ so that $AY = X$. For some basic robustness to noise we'll solve

$$A := \operatorname{argmin}_A \|AY - X\|_F^2$$

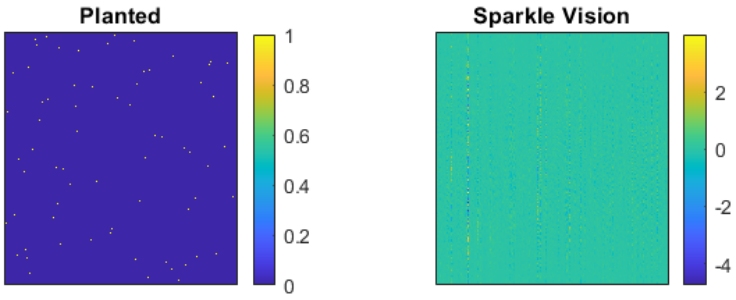
(I'll avoid special scripts on the optimal A for ease of notation). Ideally we'd hope for $A \geq 0$ and some conditions to ensure A and the implied map B don't create extra light, but we'll follow [33] and ignore these issues since this setup already does very well in practice. Note that $B = A^{-1}$ when B is invertible and there is no noise on the images.

The solution to this minimization is well know [17], since it's just several simulatenous least squares problems. We have

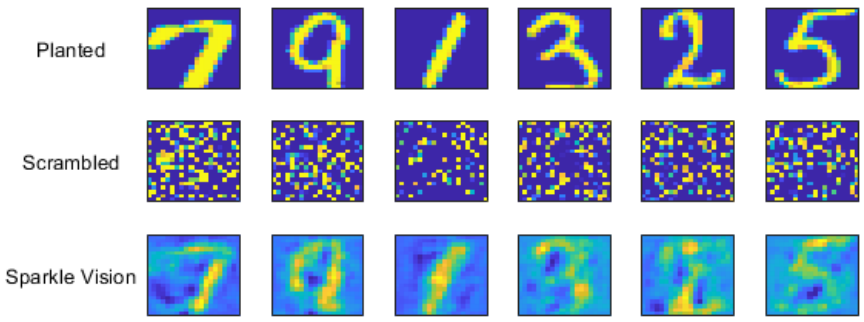
$$A = XY^{-1}$$

where $Y^{-1} \in \mathbb{R}^{N \times D'}$ is the Moore-Penrose pseudoinverse. This is easily calculated with standard libraries and runs very quickly. If we take the coulmns of X to be a reasonable (linearly independent outside the kernel of B) set of $N \geq \min(D, D')$ calibration images and assume no noise, this recovers A perfectly. We can also use additional calibration images to make this robust to pixel noise, and if we'd like to give different weights to the calibration images we can just appropriately scale the columns of Y. From here we can make some small improvements, by say thresholding the inputs Y to reduce effects of noise; this gives most of the computational insights from [33]. The takeaway is that the computational part of unscrambling images is surprisingly trivial, and fast.

1.2. **Image shift concerns.** As mentioned in [33], image shifts can cause issues. If the system is already properly calibrated (i.e. we've computed A and it nearly inverts B), decoding an image $y \in \mathbb{R}^{D'}$ just requires computing Ay but this will fail terribly if y is off by as much as a single pixel, which can easily happen from someone bumping the camera, vibrations, or a gust of wind. A simple solution is to compute Ay_v for various image shifts $v \in \mathbb{R}^2$ and select the most image-like result. This solves that issue, but what if the pixel shifts happened during calibration? That is, what if when we measure $Y = BX$ we actually see $Y = BX'$ where columns of X' are columns of X randomly shifted by a fraction of a pixel? Or equivalently, generate $Y = BX$ and then replace each column of X by a shifted copy. In this case, the Sparkle Vision algorithm fails dramatically:



This is part of the results of a simulation I ran in which the calibration images were each randomly shifted by $v \sim N(0, I) \in \mathbb{R}^2$. On the left is the matrix A we hope to recover (a permutation matrix), and on the right is what Sparkle Vision recovers. And we can also see how well it recovers digits from test data:



Here the second row is the effect of scrambling the image and then using Sparkle Vision to unscramble it, with the matrix A found above. Clearly it's doing something right, but the result is worse than blurry because of all the texture added to the background. It is possible to do some processing on the guess for A to improve it: remove outlier entries and rescale. I explored this and found we do get an okay guess for A, but with the same issues as above.

The fact that such a simple algorithm succeeds on the original problem is impressive, but it relies on near-perfect alignment throughout calibration. So I give an algorithm that can deal with this issue, being robust to translational noise (and really any local distortions) during calibration.[1]

2. Robust Sparkle Vision

2.1. **Overview.** We hope to undo *global distortions* (that scramble an image nearly incoherently) with a matrix A, with robustness to *local distortions* given by small shifts and warps of the image. A natural way to quantify local distortions is with the Wasserstein metric and mass transport: we can consider the cost of moving the pixels of one image to the pixels of another, and for local distortions that cost will be small. Viewing the original Sparkle Vision objective as

$$\sum_{n=1}^{N} \|Ay_n - x_n\|_2^2$$

we can see that the ℓ^2 norm isn't "continuous" with respect to local distortions: a single-pixel image is "far away" from itself whether its translated by 1 pixel or 100 pixels. So we replace it with the Wasserstein metric:

$$\sum_{n=1}^{N} W(Ay_n, x_n)$$

To optimize this we can't use spectral methods like before, gradient-free methods are typically slow, and the Wasserstein metric is already slow to compute. So we replace it with the Sinkhorn divergence [12], which we know is fast to compute and differentiable [13]. Now our objective is

$$\sum_{n=1}^{N} S(Ay_n, x_n, \lambda, C)$$

with parameter λ, and it's worth mentioning that there's choice in the cost matrix $C \in \mathbb{R}^{D\times D}$. Indexing C with pairs $\big((i,j),(\ell,k)\big)$, we'll choose

$$C_{(i,j),(\ell,k)} = \min\big(\sqrt{(i-k)^2+(j-\ell)^2},\, r\big)$$

$r = 10$, and $\lambda = 10$ since they work well in practice. The r is so that the entries of C aren't needlessly large, since the Sinkhorn algorithm uses $e^{-\lambda C}$ and does divisions so $r = \infty$ could lead to issues when things are numerically zero or near zero; using r is fine since we only care about robustness to local distortions, and it seems to perform better than $r = \infty$ in practice. Finally, we'll add an ℓ^1 penalty since it might be reasonable to expect A is sparse:

$$O := \frac{1}{N}\sum_{n=1}^{N} S(Ay_n, x_n, \lambda, C) + \rho \sum_{i=1}^{D}\sum_{j=1}^{D'} |A_{ij}|$$

[1] A related problem is that of dealing with non-translational local distortions when decoding some y, since now we can't just consider shifts y_v. It should be possible to maximize "image-ness" over a set of local distortions, using Sinkhorn to find the optimal local distortion via gradient ascent. Or, we can consider the further problem of these distortions during calibration: both the reference images X and scrambled images Y are locally distorted. Here it should be possible to add another Sinkhorn step to undistort the Y, so that we can recover the proper A; and then the undistortions of Y give a reasonable prior on how a new image y should be undistorted before applying A. But I'll leave these for future work.

Now we can optimize the objective with gradient descent on A. Two important issues:

1) To compute Sinhorn divergence between vectors a, b, we need $a, b \geq 0$ and $\|a\|_1 = \|b\|_1$, so after each gradient descent step we have to "round" A by enforcing $A \geq 0$ and its column sums to be 1. At this point you might think we can set $\rho = 0$, since rounding A will keep the ℓ^1 penalty constant. But in practice having the penalty there helps the gradient descent converge to the true A, so we'll keep it for computing gradients but will ignore it when reporting the value of the objective function.
2) To compute the Sinkhorn divergence and its gradient we'll need the input vectors to be strictly positive, so before starting the optimization we'll add a small amount noise to X and Y to avoid pixel values of zero.

2.2. **Gradients.** Focusing on a single $s_n := S(Ay_n, x_n, \lambda, C)$ term of the objective we can compute $\frac{\partial s_n}{\partial a_n} \in \mathbb{R}^D$ by following [13] (see Appendix A for details), where $a_n := Ay_n$. So by the chain rule,

$$\frac{\partial s_n}{\partial A_{ij}} = \sum_{k=1}^{D} \left(\frac{\partial s_n}{\partial a_n}\right)_k \frac{\partial (a_n)_k}{\partial A_{ij}}$$

We can compute

$$\frac{\partial (a_n)_k}{\partial A_{ij}} = \frac{\partial (Ax_n)_k}{\partial A_{ij}} = \begin{cases} (x_n)_j & k = i \\ 0 & \text{otherwise} \end{cases}$$

so

$$\frac{\partial s_n}{\partial A_{ij}} = \left(\frac{\partial s_n}{\partial a_n}\right)_i (x_n)_j$$

that is

$$\frac{\partial s_n}{\partial A_{ij}} = \left(\frac{\partial s_n}{\partial a_n}\right) x_n^T$$

where the layout of the gradient is the same as that of A. If $G \in \mathbb{R}^{D \times N}$ is a matrix with columns the $\frac{\partial s_n}{\partial a_n}$, then it follows that

$$\frac{\partial}{\partial A} \sum_{n=1}^{N} S(Ax_n, y_n, \lambda, C) = GX'$$

Then for the ℓ^1 term the gradient is just

$$\frac{\partial}{\partial A} \sum_{k=1}^{D} \sum_{\ell=1}^{D'} |A_{k\ell}| = \operatorname{sign}(A)$$

so all together

$$\frac{\partial O}{\partial A} = \frac{1}{N} GX' + \rho \operatorname{sign}(A)$$

2.3. **Implementation.** I implemented everything in MATLAB (2022a) [5]. There is publicly available code for the Sinkhorn iteration [11], but I did it myself for a no-frills version that's easy to parse. There is a Python package for mass transport problems including basic gradients [1], but it does not include sharp gradients and being in Python was a dealbreaker speed-wise.

Since the Sparkle Vision algorithm works perfectly when there is perfect alignment throughout calibration, after applying $B = A^{-1}$ I'll shift the images in X by a random vector $v \sim N(0, \varepsilon^2 I)$. For example, a shift of 0.8 pixels to the right means that we average the original image with a weight of 0.2 and the image shifted 1 pixel to the right with a weight of 0.8.

Some small notes/guidance for the implementation:

- Setting up a problem:
  - Process the data into some convenient form, like a matrix X or R/G/B channels. Normalize each image to have a pixel sum of one so they all have the same influence in the objective, unless you have a good reason to weight certain images as more important in the calibration. Save this data (in my case to a .mat file) for easy access.
  - After picking A, find $Y = A^{-1}X$ and then apply the random shifts to X.
  - Create copies of X, Y with added noise to avoid pixel values of zero, for use with Sinkhorn. Too much noise and the recovery will be slow, too little and you'll have singular matrix issues when computing Sinkhorn gradients. I added a noise of $\frac{0.1}{D}$ to each pixel of X and $\frac{0.1}{D'}$ to each pixel of Y.
- Running robust Sparkle Vision
  - Run the Sinkhorn iteration to a reasonable precision. Or rather than think about precision, just run it for a decent number of iterations.
  - Use a step size η to scale gradients, and make sure to set a reasonable ρ.
  - Use stochastic gradient descent, at least in the first few iterations to speed things up. I was lazy about it, so for each iteration I'd pick k of the N calibration images at random and not bother with training epochs.
  - Thresholding the estimate for A to keep only the largest entries tends to reduce blur.
  - A fully hands-off optimization would be great, but since the goal is to get image-like things it's not a bad idea to run a few iterations, look at the results, tweak parameters, and run some more iterations. Overfitting is possible, but I'd like to think the stochastic gradients and implicit regularization due to the number of entries in A will ensure the result is reasonable.

3. Numerics

I tested Sparkle Vision and robust Sparkle Vision on three datasets: MNIST[2], CIFAR-10[3], and some pictures of vegetables[4][2]. All computations were run on an Intel i7-10750H CPU at 2.6 GHz, and MATLAB required at most 4 GB memory in all the following tests.

For each data set the training data consists of some number $N > D$ digits, the planted A is a random permutation matrix (so $D' = D$), and I reserve some test data for visualizing the performance at the end. In each case I used $\varepsilon = 1$ for the random translational noise; $\frac{1}{D}$ noise added to each pixel for Sinkhorn; $\lambda = 10$, $r = 10$ and 100 iterations for Sinkhorn; $\eta = 0.1, \rho = 0.001$, and 100 iterations of $k = 100$ stochastic gradient descent.

The original Sparkle Vision paper [33] simulates the "glitter wall" setup, but this uses ray tracing which I'm not familiar with so I avoided this approach and opted for the permutation matrix. I tried other approaches, like a simulated "glitter wall" idea or just random noise for A, but the robust Sparkle Vision algorithm would run into numerical issues. A good avenue for future work would be using more complicated and realistic planted A matrices, or just physically creating a "glitter wall" experiment. The numerics below serve as a proof of concept.

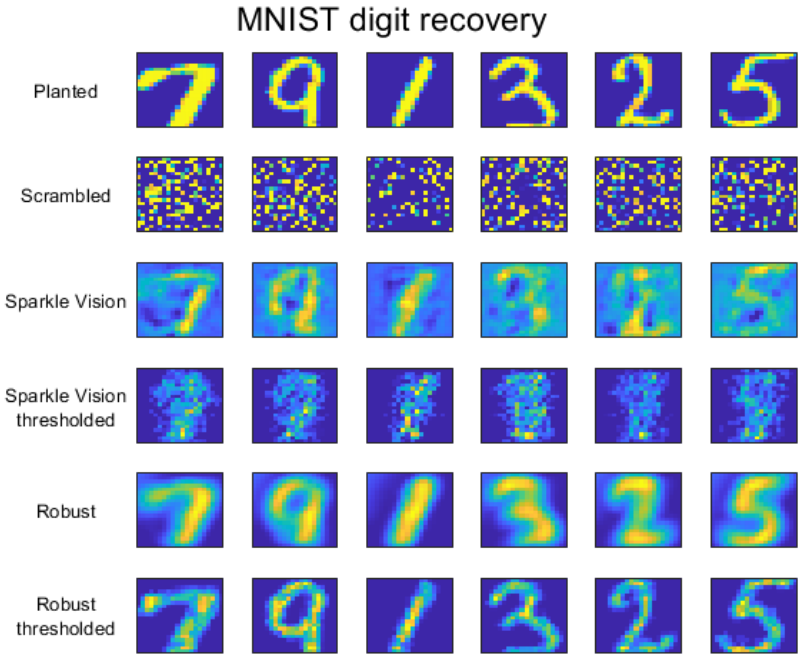
Lastly, in an application we would take care to use a curated set of standard high-contrast calibration images. Here I use digits, pictures of cars/dogs/etc., and vegetables for calibration to have somewhat realistic images while maintaining simplicity.

[2] http://yann.lecun.com/exdb/mnist/

[3] https://www.cs.toronto.edu/~kriz/cifar.html

[4] https://www.kaggle.com/datasets/misrakahmed/vegetable-image-dataset

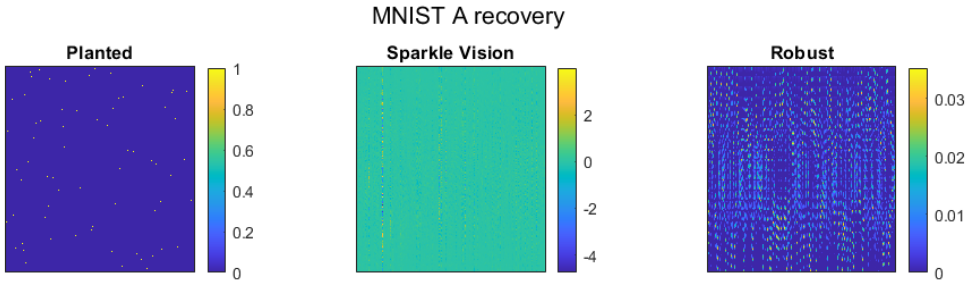
3.1. **MNIST.** Using $N = 1000$ training samples of the 20×20 digits, we see the following:



Sparkle Vision ran instantly, whereas robust Sparkle Vision took 6 seconds.

Here the top row is examples of input digits (not included in the training data), row 2 is the scrambled digits, and rows 3 and 5 are the results of unscrambling using the A predicted by Sparkle Vision and robust Sparkle Vision, respectively. Rows 4 and 6 are the results of thresholding the matrix A predicted by Sparkle Vision or robust Sparkle Vision to keep only the largest 5 entries in each column (and then normalizing the columns).

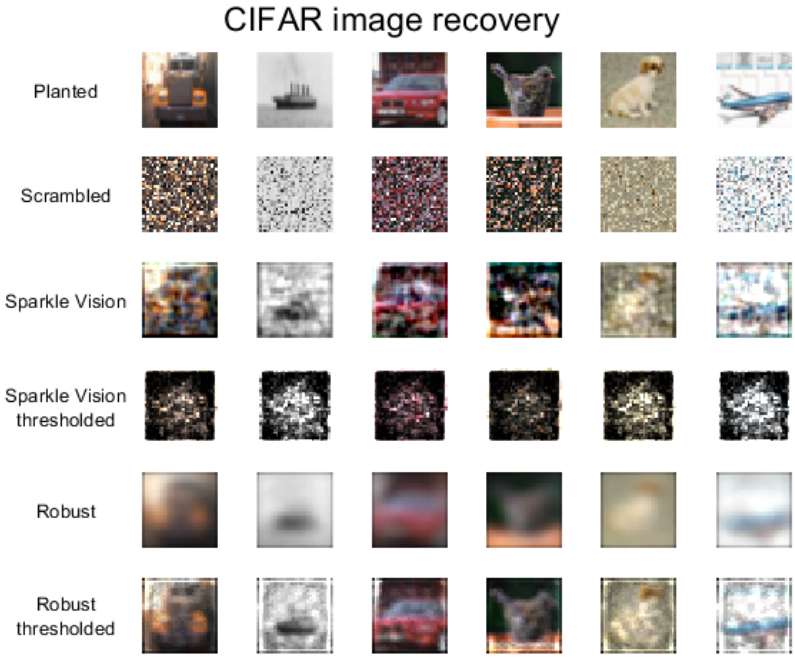
Robust Sparkle Vision clearly outperforms the more basic algorithm, and it gives near-perfect recovery with thresholding. Here are the matrices $A \in \mathbb{R}^{D \times D}$ visualized:



From both figures, it seems robust Sparkle Vision just has some blurriness issues: instead of mapping one pixel to the single target pixel, it tends to map it to the surrounding pixels too. For example, it realizes the edges of the image should be mostly dark; Sparkle Vision does not. Perhaps some processing more clever than thresholding could recover the planted A perfectly.

Running more iterations does not seem to help.

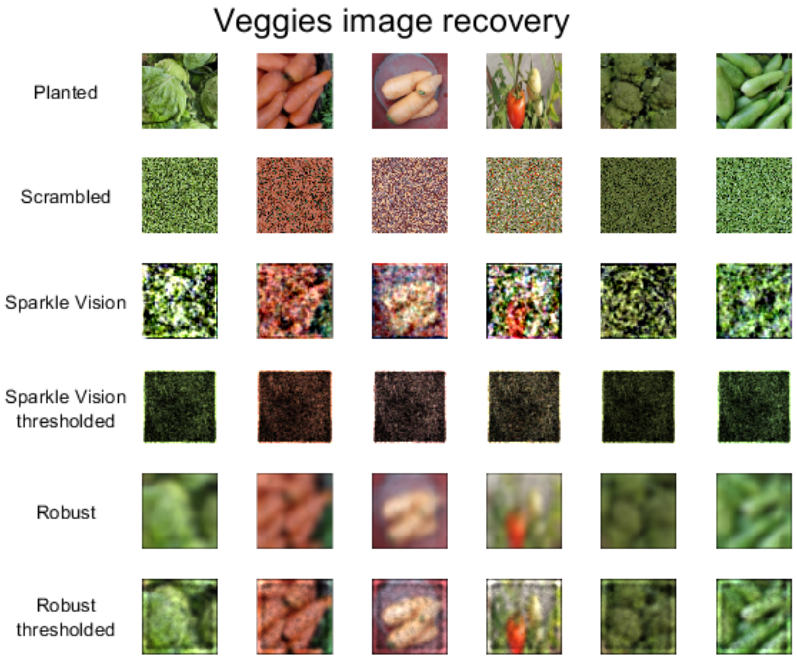
3.2. **CIFAR-10.** This is a dataset of 32×32 color images so I split each image into its RGB components, giving 3x the number of training images. Now with $N = 3000$ (i.e. 1000 images, split into their 3 channels), we get the following:



Sparkle Vision ran in under a second, whereas robust Sparkle Vision took 40 seconds. Here, the thresholded version kept the 10 largest entries of each column of A.

Robust Sparkle Vision somewhat outperforms Sparkle Vision. Its recovered images (5th row) are very blurry, but the thresholded robust Sparkle Vision result appear somewhat better than the Sparkle Vision results (particularly the truck and car). I think the ways in which each algorithm's results are bad tells us that robust Sparkle Vision is the better one: the Sparkle Vision solution is very noisy in jagged ways, and the fact that its thresholded version performs poorly suggests that its reconstruction of A is not good; meanwhile Sparkle Vision gives a correct blurry result, and the thresholded version is a reaonably good de-blurring. Some experimenting showed that the Sparkle Vision result is a terrible initialization for robust Sparkle Vision, leading to convergence to bad local minima; this seems telling.

3.3. **Vegetables.** I picked this data set since it featured many similar images with high detail, which should be a difficult case for robust Sparkle Vision: less variety should make the calibration harder, and the detail should be hard to recover. The images were originally 224×224 pixels, so I scaled them down to be 56×56. Now with $N = 6000$, results were:



Sparkle Vision ran in 12 seconds, whereas robust Sparkle Vision took 6 minutes. Here, the thresholded version kept the 20 largest entries of each column of A. Again, the robust Sparkle Vision recovery gives a good blurry image. And the thresholded version is a decent de-blurring: the images look dirty, but correct.

To push things to their limit, I tried using 112×112 images. In this case, Sparkle Vision took 6 minutes and robust Sparkle Vision took nearly 1 minute per iteration, but ran into fatal numerical issues on the second iteration. Working on this larger scale is definitley possible, but requires more numerically stable implementations of the Sinkhorn iteration.

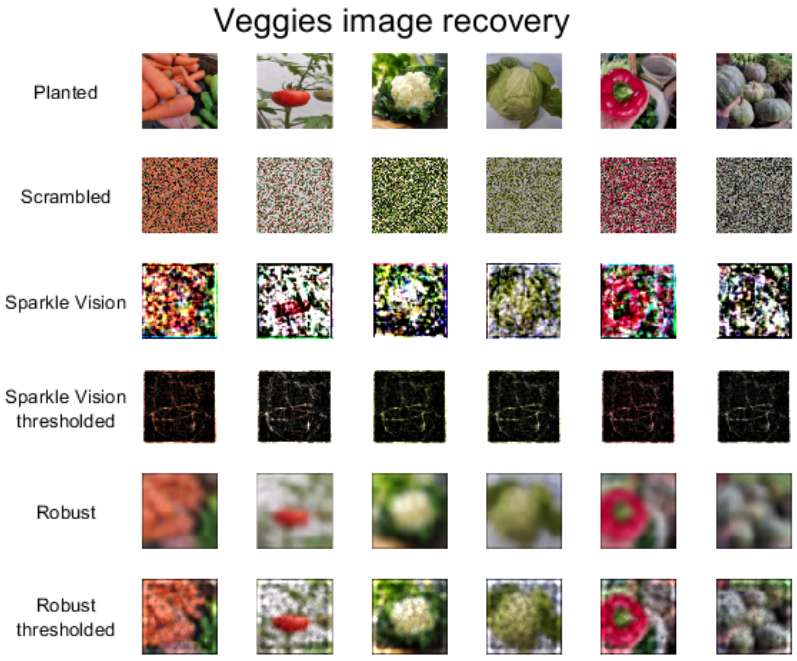
4. Discussion

The robust Sparkle Vision program gives a reasonably principled and natural generalization of the basic Sparkle Vision program, applying Sinkhorn gradient methods that have seen some popularity in the last few years. The numerics above suggest that it is well-suited to the task, but requires more care in making sure the Sinkhorn algorithm is stable for larger applications. It also has an additional advantage not showcased above: it should be robust to any local distortions, not just translations.

One possible improvement would be using sharp Sinkhorn gradients (see Appendix A) but in testing these were far less stable, reaching numerical issues before converging and giving worse results. While there is literature on stabilizing the Sinkhorn iteration, to my knowledge there is nothing on stabilizing the sharp Sinkhorn gradients algorithm. In the appendix I give one known trick to improve stability [24], but there is no theoretical justification for it and surely there are other improvements to be made. Progress on this front could help improve on the above robust Sparkle Vision results: after the regularized gradients converge, switch to sharp gradients ot reduce the blur. Hower, it's worth mentioning the basic implementation of sharp gradients makes the program much slower.

Another option is to improve the post-processing on the recovered A: the above work gives a correct blurry A, so maybe deconvolution and similar methods could be used to enhance the image.

Lastly, for rank reasons Sparkle Vision requires $N > D$ to recover A but this may not be the case for robust Sparkle Vision. Repeating the vegetable test with $N \sim D$ gives reasonable results for robust Sparkle Vision but causes color mis-alignments and white/black pixels with Sparkle Vision. For an extreme case, taking $N = 999 < D/3$ gives the following:



5. Project autobiography

This is just an account of how I ended up on this project, mainly to my future self.

Working with Dustin Mixon, my original interest was in using semidefinite programming to find theory-friendly approaches to combinatorial problems. One early project I worked on (with the help of Dan Packer) was a linear version of SqueezeFit [25], where we looked for stronger recovery theorems. After working on that for a while and getting some results but nothing ground-breaking, I switched to a completely different project involving mass transport which would eventually lead to the topic of this write-up.

I worked with Dustin and Soledad Villar (JHU) on the "sliced Wasserstein" metric, which is a computationally fast (at least in 2 dimensions) approximation to the Wasserstein metric. It's been analyzed before in various ways [7], with some notable recent examples [23, 29]. The context for all this is that earth mover's distances (i.e. mass transport problems) are useful [16], and sliced Wasserstein had found some use in calculating barycenters [6, 19]. So I worked on some basic analysis of sliced Wasserstein as a metric: comparing it to Wasserstein by improving on the metric-equivalence results in [7]; and trying to prove a closed form expression for sliced Wasserstein on Gaussian distributions, since the closed form for Wasserstein on Gaussians is known [26].

I did get some rough results, but couldn't complete the proofs I'd have liked. But at some point along the way Soleded recommended I read the seminal paper on Sinkhorn divergences by Marco Cuturi [12], and after seeing just how fast and accurate Sinkhorn is and that it can be applied to efficient barycenter computations [14], I decided I really liked mass transport and I'd look for chances to play around with it in general.

At Dustin's weekly meetings I presented on Sinkhorn divergences (along with Natalia Kravtsova, who was fun to talk to and helped me understand the Bregman projection framework [3]) a few times, specifically a paper on taking the gradient of the transport [24]. While reading this, I saw Cuturi's paper on using gradients of Sinkhorn divergence to compute Wasserstein barycenters [13], and from there I was hooked. It was around this time that I decided to not pursue the PhD and instead opt for a master's, so with that in mind I spent winter break taking a break from research altogether, but playing around with implenting the barycenters myself [4]. I did manage to find good average-case digits, but clustering digits with k-means and Sinkhorn didn't work out.

Once spring semester started, Dustin found me a mass-transport-related project with his collaborator Bianca Dumitrascu. The goal is to select important genes from a DNA sequence for predictive purposes, a popular problem since we'd rather not have to sequence someone's entire DNA to figure out how we think they'll react to cancer drugs. Our approach is to view it as a projection recovery problem: if there's n genes and $m \ll n$ responses to predict, we'll assume there's a choice of m of the n genes so that those gene expressions and desired responses are reasonably well-aligned in $\mathbb{R}^m$. In that case, we know the optimal transport of how points in dataset 1 (genes) in $\mathbb{R}^m$ correspond to points in dataset 2 (responses); so it's an inverse transport problem, of using the known transport to infer a projection. Along the way I rediscovered a simpler version of the approach in [22]; the full method is computationally hard overkill for our purposes.

Working on this project got me comfortable using Sinkhorn gradients in more complicated objective functions. But then a related paper [18] led me to learn about Procrustes analysis [17], and with that I discarded the need for Sinkhorn. So to resurrect it, I melded the projection recovery project with Wasserstein barycenters ideas to create a novel problem where Sinkhorn is necessary. After a literature search I saw that a simpler version had already been considered in [33], solved with Procrustes analysis. I didn't find any follow-up papers addressing image shift issues, so my work would fit in perfectly.

Along the way I did a bunch of other things: the Algorithms for Threat Detection competition (with Dan Packer) where we implemented powerful SVD-inspired techniques [21] for recommendation algorithms; more reading on convex programs, theory and algorithms [9]; and various papers from group meetings (such as [8]). These were all helpful in giving me a sense of what is and isn't possible, and what techniques I liked using.

For the group meetings, I spent some time tracing the origins of the Sinkhorn divergence story. The standard references are [15] showing that the form of the Sinkhorn transport is well-known; and then the namesake of the algorithm given by [20, 30], and most importantly [31] which are the standard proofs of the relevant fact for diagonally scaled matrices. The origin of the entropy formulation in Sinkhorn divergence goes back to [32], which shows there is an entropy interpretation for the gravity model in transportation theory. And the gravity model has origins in the 1920s [27]; for more history, see [15] (which can be hard to get a hold of, I had to use OhioLink to call it in from Toledo). This was all fun to uncover since I've been thinking about (human) transportation a lot recently, and the gravity model happened to come up in a video I saw a few weeks ago [10].

The whole novelty of robust Sparkle Vision is bringing the Sinkhorn gradients idea to the image recovery problem, nothing more than cross-pollination. And it hasn't required that much work: I think it's only been 2 weeks since I came up with the robust Sparkle Vision idea. But it does nicely build off of and tie together the mass transport ideas I've thought about over the past year. It's been a nice project with fun surprises (like using sharp gradients for 2 weeks and suddently realizing regularized gradients are much better in practice!), and makes a nice end to my time at OSU.

References

[1] Pot: Python optimal transport. `https://pythonot.github.io`. Accessed: 2022-04-05.

[2] M. I. Ahmed, S. Mahmud Mamun, and A. U. Zaman Asif. Dcnn-based vegetable image classification using transfer learning: A comparative study. In *2021 5th International Conference on Computer, Communication and Signal Processing (ICCCSP)*, pages 235–243, 2021.

[3] J.-D. Benamou, G. Carlier, M. Cuturi, L. Nenna, and G. Peyré. Iterative bregman projections for regularized transportation problems. 2014.

[4] N. Bolle. barycenters. `https://github.com/nicolas-bolle/barycenters`.

[5] N. Bolle. robust_sparkle_vision. `https://github.com/nicolas-bolle/robust_sparkle_vision`.

[6] N. Bonneel, J. Rabin, G. Peyré, and H. Pfister. Sliced and radon wasserstein barycenters of measures. *J Math Imaging Vis*, 2015.

[7] N. Bonnotte. Unidimensional and evolution methods for optimal transportation. 2013.

[8] N. Boumal, V. Voroninski, and A. S. Bandeira. The non-convex burer-monteiro approach works on smooth semidefinite programs. 2016.

[9] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[10] CityNerd. Induced demand & roadway widening: Everything you always wanted to know (and weren't afraid to ask). `https://youtu.be/za56H2BGamQ?t=320`.

[11] M. Cuturi. Marco cuturi - sinkhorn scaling for optimal transport. `https://marcocuturi.net/SI.html`. Accessed: 2022-04-05.

[12] M. Cuturi. Sinkhorn distances: Lightspeed computation of optimal transportation distances. 2013.

[13] M. Cuturi and A. Doucet. Fast computation of wasserstein barycenters. 2013.

[14] M. Cuturi and A. Doucet. Fast computation of wasserstein barycenters. 2013.

[15] S. Erlander and N. Stewart. *The Gravity Model in Transportation Analysis: Theory and Extensions*. Topics in Transportation. Taylor & Francis, 1990.

[16] C. Gottschlich and D. Schuhmacher. The shortlist method for fast computation of the earth mover's distance and finding optimal solutions to transportation problems. *CoRR*, abs/1405.7903, 2014.

[17] J. C. Gower and G. B. Dijksterhuis. *Procrustes Analysis*. Oxford University Press, Oxford, 2004.

[18] E. Grave, A. Joulin, and Q. Berthet. Unsupervised alignment of embeddings with wasserstein procrustes. 2018.

[19] R. Julien, G. Peyré, J. Delon, and B. Marc. Wasserstein barycenter and its application to texture mixing. 2011.

[20] P. Knopp and R. Sinkhorn. Concerning nonnegative matrices and doubly stochastic matrices. *Pacific Journal of Mathematics*, 21(2):343 – 348, 1967.

[21] Y. Koren. Factorization meets the neighborhood: A multifaceted collaborative filtering model. KDD '08, page 426–434, New York, NY, USA, 2008. Association for Computing Machinery.
[22] R. Li, X. Ye, H. Zhou, and H. Zha. Learning to match via inverse optimal transport. 2018.
[23] T. Lin, Z. Zheng, E. Y. Chen, M. Cuturi, and M. I. Jordan. On projection robust optimal transport: Sample complexity and model misspecification. 2020.
[24] G. Luise, A. Rudi, M. Pontil, and C. Ciliberto. Differential properties of sinkhorn approximation for learning with wasserstein distance. 2018.
[25] C. McWhirter, D. G. Mixon, and S. Villar. Squeezefit: Label-aware dimensionality reduction by semidefinite programming. 2018.
[26] I. Olkin and F. Pukelsheim. The distance between two random vectors with given dispersion matrices. *Elsevier*, 1982.
[27] W. J. Reilly et al. Methods for the study of retail relationships. 1929.
[28] B. Schmitzer. Stabilized sparse scaling algorithms for entropy regularized transport problems, 2016.
[29] M. Shifat-E.-Rabbi, X. Yin, A. H. M. Rubaiyat, S. Li, S. Kolouri, A. Aldroubi, J. M. Nichols, and G. K. Rohde. Radon cumulative distribution transform subspace modeling for image classification. *CoRR*, abs/2004.03669, 2020.
[30] R. Sinkhorn. A Relationship Between Arbitrary Positive Matrices and Doubly Stochastic Matrices. *The Annals of Mathematical Statistics*, 35(2):876 – 879, 1964.
[31] R. Sinkhorn. Diagonal equivalence to matrices with prescribed row and column sums. *The American Mathematical Monthly*, 74(4):402–405, 1967.
[32] A. Wilson. A statistical theory of spatial distribution models. *Transportation Research*, 1(3):253–269, 1967.
[33] Z. Zhang, P. Isola, and E. H. Adelson. Sparkle vision: Seeing the world through random specular microfacets. 2014.

Appendix A. Sinkhorn gradient details

Standard theory tells us dual variables give subgradients [9]. For our purposes, that makes computing gradients of Sinkhorn divergence easy: given two histograms $a, b \in \mathbb{R}^D_{>0}$ with $\|a\|_1 = \|b\|_1$ and letting s be the regularized Sinkhorn divergence, run the Sinkhorn iteration to obtain vectors u, v (see Appendix B) and we obtain the subgradient

$$\frac{\partial s}{\partial a} = \frac{\log(u)}{\lambda} + t1$$

with a single degree of freedom. In the implementation, I chose t so that the gradient vector is centered (has sum zero) as in [13].

There is another gradient story for "sharp" gradients, derived in [24], which I include for completeness, to fix some small errors in that paper, and to fill in some details. Sharp gradients would theoretically perform better, but require much better numerical stability and take much longer. They are a good avenue for future work, in making the recovery less blurry.

A.1. **Differentiability.** Fixing $\lambda > 0$ and $C \in \mathbb{R}^{D\times D}_{\geq 0}$ a cost matrix, given two histograms $a, b \in \mathbb{R}^D_{>0}$ with $\|a\|_1 = \|b\|_1$ the (sharp) Sinkhorn divergence between them is $s := \langle T^*, C\rangle$ where

$$\begin{aligned} T^* := \operatorname{argmin}_{T\in\mathbb{R}^{D\times D}} & \sum_{i,j=1}^D T_{ij}C_{ij} + \frac{1}{\lambda}\sum_{i,j=1}^D T_{ij}\log T_{ij} \\ \text{subject to} \quad & \sum_{j=1}^D T_{ij} = a_i \quad \forall i, \quad \sum_{i=1}^D T_{ij} = b_j \quad \forall j \end{aligned}$$

where the $T > 0$ constraint is implied by the domain of the objective. This is the *primal program*. We want to find the gradient $\frac{\partial s}{\partial a}$, which will require stepping through the gradient of T^* [24].

Since the objective in the primal program the sum of an affine term and strictly convex term (since entropy is strictly concave) and we have the feasible point $T = \frac{1}{\|a\|_1}ab^T$, by Slater's condition we have strong duality and can recover the unique optimizer as the saddle point of the Langrangian [9]. Doing this will let us write T more explicitly as a function of a.

To simplify things later on, let's modify the objective by taking

$$\sum_{i,j=1}^D T_{ij}\log T_{ij} \quad \mapsto \quad \sum_{i,j=1}^D T_{ij}\big(\log T_{ij} - 1\big)$$

This doesn't affect things, since it offsets the objective by a factor of $\sum_{i,j=1}^D T_{i,j} = D\|b_1\|$ which we're assuming is constant.

So, we write the Langrangian

$$\begin{aligned} \mathcal{L}(a; T; \alpha, \beta) = & \sum_{i,j=1}^D T_{ij}C_{ij} + \frac{1}{\lambda}\sum_{i,j=1}^D T_{ij}\big(\log T_{ij} - 1\big) \\ & + \sum_{i=1}^D \alpha_i\left(a_i - \sum_{j=1}^D T_{ij}\right) + \sum_{j=1}^D \beta_j\left(b_j - \sum_{i=1}^D T_{ij}\right) \\ = & \alpha^T a + \beta^T b + \sum_{i,j=1}^D T_{ij}(C_{ij} - \alpha_i - \beta_j) + \frac{1}{\lambda}\sum_{i,j=1}^D T_{ij}\big(\log T_{i,j} - 1\big) \end{aligned}$$

keeping the dependence on a explicit since we'll need it later. The saddle point of the Langrangian occurs when its gradient in T is zero, so we compute

$$\begin{aligned} 0 &= \frac{\partial}{\partial T_{ij}} \mathcal{L}(a; T; \alpha, \beta) \\ &= (C_{ij} - \alpha_i - \beta_j) + \frac{1}{\lambda}\left(1 \cdot \left(\log T_{ij} - 1\right) + T_{ij} \cdot \frac{1}{T_{ij}}\right) \\ &= C_{ij} - \alpha_i - \beta_j + \frac{1}{\lambda} T_{ij} \end{aligned}$$

so that

$$T^*_{ij} = e^{\lambda(\alpha^*_i + \beta^*_j - C_{ij})}$$

that is

$$T^* = \operatorname{diag}(e^{\lambda \alpha^*}) e^{-\lambda C} \operatorname{diag}(e^{\lambda \beta^*})$$

where $\alpha^*, \beta^* \in \mathbb{R}^n$ are optimal dual variables. The Sinkhorn iteration gives us a way to compute α^*, β^*, and thus T^* efficiently (see Appendix B).

To show s is smooth in a, since T^* is smooth in α^*, β^* it suffices to show that α^*, β^* are smooth in a. There's the blatant issue that α^*, β^* are not uniquely defined, but we'll deal with this in due time. The goal will be to leverage the implicit function theorem, constructing a function $\psi(a; \alpha, \beta)$ so that setting $\psi = 0$ for a given a implicitly defines α^*, β^*. Towards this, referring back to the Lagrangian and substituting the new expression for T we have the *dual program*

$$g(a; \alpha, \beta) = \alpha^T a + \beta^T b - \frac{1}{\lambda} \sum_{i,j=1}^{D} e^{\lambda(\alpha_i + \beta_j - C_{ij})}$$

Optimal α^*, β^* can be found by setting $\nabla_{(\alpha,\beta)} g(a; \alpha, \beta) = 0$, so we'll let $\psi(a; \alpha, \beta) = \nabla_{(\alpha,\beta)} g(a; \alpha, \beta)$. We can compute this:

$$\psi(a; \alpha, \beta) = \begin{bmatrix} a_1 - \sum_j e^{\lambda(\alpha_1 + \beta_j - C_{1j})} \\ \vdots \\ a_D - \sum_j e^{\lambda(\alpha_D + \beta_j - C_{Dj})} \\ b_1 - \sum_i e^{\lambda(\alpha_i + \beta_1 - C_{i1})} \\ \vdots \\ b_D - \sum_i e^{\lambda(\alpha_i + \beta_D - C_{iD})} \end{bmatrix} \in \mathbb{R}^{2D}$$

To use the implicit function theorem we need to show $\nabla_{(\alpha,\beta)} \psi(a; \alpha, \beta)$ is nonsingular, so we compute it as the Hessian of the dual problem:

$$H := \nabla_{(\alpha,\beta)} \psi(a; \alpha, \beta) = -\lambda \begin{bmatrix} \operatorname{diag}(a) & T \\ T^T & \operatorname{diag}(b) \end{bmatrix} \in \mathbb{R}^{2D \times 2D}$$

after using our expression for the T_{ij}. I'm writing T in place of T^* for ease of notation.

If H were nonsingular then the smoothness of g in α, β implies (by the implicit function theorem) that there is a smooth function $a \mapsto (\alpha^*, \beta^*)$, but $\det(H) = 0$ and furthermore the α^*, β^* aren't uniquely defined. But since the only degree of freedom is passing a scalar between them, setting $\beta^*_D = 0$ *does* make them uniquely defined in a. So we let $\overline{\beta} \in \mathbb{R}^{D-1}$ be β without its last component and instead consider

$$\overline{g}(a; \alpha, \overline{\beta}) = \alpha^T a + \overline{\beta}^T \overline{b} - \frac{1}{\lambda} \sum_{i=1}^{D} \left(\sum_{j=1}^{D-1} e^{\lambda(\alpha_i + \overline{\beta}_j - C_{ij})} + e^{\lambda(\alpha_i - C_{ij})} \right)$$

where $\overline{b}$ is b with its last component removed. This is just the dual program with $\beta_D = 0$ implicitly enforced. So we can calculate a $\overline{\psi}$ and corresponding Hessian $\overline{H}$ (it suffices to use previous results, setting $\beta_D = 0$ and removing the last component), and we get

$$\overline{H} := \nabla_{(\alpha,\overline{\beta})}\, \overline{\psi}(a; \alpha, \overline{\beta}) = -\lambda \begin{bmatrix} \operatorname{diag}(a) & \overline{T} \\ \overline{T}^T & \operatorname{diag}(\overline{b}) \end{bmatrix} \in \mathbb{R}^{(2D-1)\times(2D-1)}$$

where $\overline{T}$ is T with its last column removed. The Gershgorin circle theorem shows this is negative definite, so the implicit function theorem argument goes through and $\alpha^*, \overline{\beta}^*$ are smooth in a, which shows s is smooth in a.

A.2. **Algorithm for the gradient.** For notation, I'll drop the $\cdot^*$ on optimal solutions. To explicitly compute the gradient, let $\gamma(a) := (\alpha, \overline{\beta}) \in \mathbb{R}^{2D-1}$ be the function we want to differentiate. Then $\overline{\psi}(a; \gamma(a)) = 0$ so by the chain rule

$$0 = \nabla_a \overline{\psi}(a; \gamma(a)) = \nabla_1 \overline{\psi}(a, \gamma(a)) + \nabla_a \gamma(a) \nabla_2 \overline{\psi}(a, \gamma(a)) \in \mathbb{R}^{D\times(2D-1)}$$

where we pick the gradient layout so that $\nabla_a s$ will end up as a column vector in the end. Referring back to a formula for ψ, we see

$$M := \nabla_1 \overline{\psi}(a, \gamma(a)) = [I_D\ 0_{D\times(D-1)}]$$

and $\nabla_2 \overline{\psi}(a, \gamma(a)) = \overline{H}$, so that

$$[\nabla_a \alpha,\ \nabla_a \overline{\beta}] = \nabla_a \gamma(a) = -M \overline{H}^{-1}$$

with the columns of $\nabla_a \alpha$ giving the gradients $\nabla_a(\alpha_i)$. Now to get $\nabla_a s$, recalling $s = \sum_{i,j=1}^{D} C_{ij} e^{\lambda(\alpha_i + \beta_j - C_{ij})}$ (where we're assuming $\beta_D = 0$), we see

$$\nabla_a s = \lambda \sum_{i,j=1}^{D} C_{ij} T_{ij} \nabla_a \alpha_i + \lambda \sum_{i=1}^{D} \sum_{j=1}^{D-1} C_{ij} T_{ij} \nabla_a \overline{\beta}_j$$

For a more algorithm-friendly form, let $L = C \odot T$ (element-wise product) and $\overline{L}$ be L with its last column removed. Then we write

$$\begin{aligned} \nabla_a s &= \lambda \sum_{i}^{D} \nabla_a \alpha_i \sum_{j=1}^{D} L_{ij} + \lambda \sum_{j=1}^{D-1} \nabla_a \overline{\beta}_j \sum_{i=1}^{D} \overline{L}_{ij} \\ &= \lambda (\nabla_a \alpha\, L\, 1_D + \nabla_a \overline{\beta}\, \overline{L}^T\, 1_D) \end{aligned}$$

To clean up the ∇_a terms, by taking the inverse of $\overline{H}$ as a block matrix we have

$$\overline{H}^{-1} = \frac{-1}{\lambda} \begin{bmatrix} K_1 & K_2 \\ * & * \end{bmatrix}$$

and $\nabla_a \alpha = \frac{1}{\lambda} K_1$, $\nabla_a \overline{\beta} = \frac{1}{\lambda} K_2$ where

$$K_1 = K^{-1} \qquad K_2 = -K^{-1}\, \overline{T}\, \operatorname{diag}(\overline{b})^{-1} \qquad K := \operatorname{diag}(a) - \overline{T} \operatorname{diag}(\overline{b})^{-1} \overline{T}^T$$

So

$$\nabla_a s = K^{-1} \big(L\, 1_D - \overline{T}\, \operatorname{diag}(\overline{b})^{-1}\, \overline{L}^T\, 1_D \big)$$

or

$$\nabla_a s = \operatorname{solve}(K,\ f) \qquad f := L\, 1_D - \overline{T}\, \operatorname{diag}(\overline{b})^{-1}\, \overline{L}^T\, 1_D$$

Some notes for implementation:

1) Even though we performed our analysis with vectors $\alpha, \overline{\beta}$, in the end we only needed
$$T = \operatorname{diag}(e^{\lambda\alpha})e^{-\lambda C}\operatorname{diag}(e^{\lambda\beta})$$
So we can use the Sinkhorn iteration (see Appendix B) to find T. A typical algorithm for finding the gradient is to do the Sinkhorn iteration for T, form $\overline{T}$, L, $\overline{L}$, find K and f, and solve the linear system.
2) Unless the Sinkhorn iteration converged perfectly, the row/column sums of T will not equal a, b. This would be fine since slightly inaccurate gradients are not an issue, except it's enough of an error to cause K to be singular. So we should instead use
$$a = T1_D \qquad \overline{b} = \overline{T}^T 1_{D-1}$$
when finding K.
3) There are small optimizations, like
   - $\overline{T}\operatorname{diag}(b)^{-1}$ is best computed as an entrywise matrix-vector multiplication with implicit expansion, rather than forming a full diagonal matrix.
   - $\operatorname{diag}(\overline{b})^{-1}(\overline{L}^T 1_D)$ should be computed before multipliying by $\overline{T}$, so that we just do an entrywise product of vectors.

Appendix B. The Sinkhorn iteration

Here I'll explain the details of the basic Sinkhorn iteration algorithm from [12]. There are variations on this algorithm that alleviate some numerical issues [1, 28], but I'll be content with just the basic approach. Let $a, b \in \mathbb{R}^D_{>0}$ be the histograms to compare, $C \in \mathbb{R}^{D\times D}_{\geq 0}$ a cost matrix, and λ our regularization parameter.

If we let $K = e^{-\lambda C}$ (calculated entrywise), up to a change of variables the above section derives
$$T = \operatorname{diag}(u)K\operatorname{diag}(v)$$
where $T > 0$, by the domain of the original program. So by Sinkhorn theory on diagonal scalings [20], the equation has a single solution for u, v up to passing a multiplicative constant between them, and valid u, v can be computed with a simple alternating matrix scaling algorithm. In its most compact form, the algorithm is the update
$$u \leftarrow a \odot \frac{1}{K\left(b \odot \frac{1}{K^T u}\right)}$$
and $v = b \odot \frac{1}{K^T u}$, where the $\odot$ and divisions are element-wise. To understand this, start with $u_0 = v_0 = 1_D$ and $A_0 = K$, and at step k of the iteration we'll scale the columns of A_{k-1} so the matrix matches the b-marginals and then scale the rows of the matrix so the resulting A_k matches the a-marginals. Iteratively updating the matrix would lead to accumulation of error, so instead we'll write $A_k = \operatorname{diag}(u_k)K\operatorname{diag}(v_k)$ and recalculate the scale factors u_k, v_k at each step. So

1) v_k is picked such that
$$\operatorname{diag}(u_{k-1})K\operatorname{diag}(v_k)$$
has column sums b.
2) u_k is picked such that
$$\operatorname{diag}(u_k)K\operatorname{diag}(v_k)$$
has row sums a.

To find the formula $u_{k-1} \mapsto u_k$, start by noting the row sums of $K \operatorname{diag}(v_k)$ are

$$\big(K \operatorname{diag}(v_k)\big) 1_D = K v_k$$

so

$$u_k = a \odot \frac{1}{K v_k}$$

achieves the desired scaling. Similarly, the column sums of $\operatorname{diag}(u_{k-1})K$ are

$$\big(1_D^T \operatorname{diag}(u_{k-1} K\big)^T = K^T u_{k-1}$$

so

$$v_k = b \odot \frac{1}{K^T u_{k-1}}$$

achieves the desired scaling. The algorithm follows.

Finally, Marco Cuturi's Sinkhorn code [11] has another matrix U, which is just $U := K \odot C$ so that the cost is simply

$$\langle T, C \rangle = \langle \operatorname{diag}(u) K \operatorname{diag}(v), C \rangle = u^T U v$$

Precomputing K, U for a given λ, C avoids a lot of redundant computations.