ADMINISTRIVIA

- Recitation will be this Friday (9/21) in Jolley 309.
  - Will go over topics relevant to Pset.
  - Problem Set due next Tuesday

GENERAL

Notes on Parallelization

Say your simple code looked like this

```python
for i in indices:
p = func1(x[i])
z = func2(x[ifunc(i)])
p = func1(x[i])
out[i] = func4(p)
```

Replace as

```python
p = func1(x[indices])
z = func2(p[indices])
p = func1(x[indices])
out[indices] = func4(p)
```

- Most numpy functions that act on single numbers can be used elementwise on arrays.
- Think about all the steps you would do for each number in a loop: Can these steps be carried out independently for different loop indices?
- If so, replace them with array operations.

IMAGE RESTORATION

\[
\hat{X} = \arg \min_X \|X - Y\|^2 + (X - \mu)^T \Sigma^{-1} (X - \mu)
\]

where, \( \mu = W^T \mu_c, \Sigma = W^T D_c W \).

The solution is:

\[
\hat{X} = (I + W^T D_c^2)^{-1} W (Y + W^T D_c^{-1} \mu_c)
\]

How would you code this up?
Remember, (circular) convolution is diagonalized in the Fourier domain!

\[
\hat{X} = W^{T}(I + D^{-1}D^{-1})^{-1}W \text{ } (Y + W^{T}D^{-1} \mu_c)
\]
\[
= W^{T}(I + D^{-1}D^{-1})^{-1}(WY + D^{-1} \mu_c)
\]

Note that here \( \mu_c \) and \( \text{sigma}2c \) are arrays the same size as \( xc \).

More general optimization setting:

\[
\hat{X} = \arg \min_x \|X[n] - Y[n]\|^2 + \lambda \sum_n \left((G_x \ast X[n])^2 + ((G_y \ast X[n])^2\right)
\]

Let \( A_x \) and \( A_y \) be matrices corresponding to convolution with \( G_x \) and \( G_y \).

\[
\hat{X} = \arg \min_x \|X - Y\|^2 + \lambda \left(\|A_x X\|^2 + \|A_y X\|^2\right)
\]
\[
= \arg \min(X - Y)^T(X - Y) + \lambda X^T(A_x^T A_x + A_y^T A_y)X
\]

Let's change this to our standard quadratic form:

\[
= \arg \min_x X^T \left(I + \lambda (A_x^T A_x + A_y^T A_y)\right) X - 2X^T Y + Y^T Y
\]

And so, \( \hat{X} = \left(I + \lambda (A_x^T A_x + A_y^T A_y)\right)^{-1} Y \)

How do you do this matrix inverse?

\[
A_x^T A_x = S D_x S^* = S \|D_x\|^2 S^*
\]
\[
A_x^T A_y = S \|D_y\|^2 S^*
\]
\[
I + \lambda (A_x^T A_x + A_y^T A_y) = I + \lambda S \|D_x\|^2 + \|D_y\|^2 S^*
\]
\[
= S \left(I + \lambda \left(\|D_x\|^2 + \|D_y\|^2\right)\right) S^*
\]

What is this doing?

It's down-weighting frequency components by a (real) factor where \( \|D_x\|^2 \) and \( \|D_y\|^2 \) are high.

Those are high for higher frequencies, because \( G_x \) and \( G_y \) are derivative filters.

So this operation down-weights higher frequency components ⇒ Smooths the image.
De-blurring / De-convolution

Say we know that our image has been blurred by a known blur kernel $k$

$$Y[n] = (X * k)[n] + e[n]$$

$$\hat{X} = \arg \min_X \|X[n] - Y[n]\|^2 + \lambda \sum_n ((G_x * X)[n])^2 + ((G_y * X)[n])^2$$

where $A_k$ represents the action of convolution by blur kernel $k$.

$$\hat{X} = (A_k^T A_k + \lambda (A_k^T A_y + A_y^T A_y))^{-1} A_y^T Y$$

Note that there is now $A_y^T Y$ instead of just $Y$.

If $\lambda = 0$, this reduces to:

$$\hat{X} = (A_k^T A_k)^{-1} A_y^T Y$$

But $K_y[u, v]$ may be zero or close to zero, in which case dividing will amplify noise.

So the Fourier transform of the kernel $k$ is telling us which frequency components are severely attenuated by the kernel.

The "regularization" with $\lambda > 0$ helps stabilize the inversion, because if $K_y[u, v]$ is low for some $(u, v)$, then the factor will downscale the input coefficient $Y[u, v]$.

This is called Wiener filtering or Wiener Deconvolution.
We discussed cases when you know of a basis (wavelet / Fourier) where you can diagonalize your quadratic system matrix, and have a closed form expression for its inverse.

Not always the case. What if you wanted to exactly model valid convolution (not approximate it as circular)?

What if you observed values at a subset of pixels?

Generally, what if you wanted to compute $X = Q^{-1}Y$ for some arbitrary symmetric positive-definite $Q$.

Let's consider a case where you can form $Q$.

- Never compute $Q^{-1}$, and then multiply by $Y$.
  - Numerically unstable, more expensive.

Call `scipy.linalg.solve`:

- Cholesky / LDL Decomposition: $Q = L D L^T$
  - Always exists for a positive definite matrix. $L$ is lower triangular.

Solve $X = Q^{-1}Y$

More generally, when $Q = A_1^T A_1 + A_2^T A_2 + \ldots$, where $A_1, A_2, \ldots$ are sparse operations, that involve convolutions and element-wise operations.

- If $A_1$ is convolution with $k$, then you can get the effect of multiplying $A_1^T A_1$ with an image $Y$ by
  - Convolving with $k$ first
  - Then, convolving the result with the flipped version of $k$

- If $A_2$ is valid convolution, $A_2^T$ will correspond to "full" convolution with flipped version of $k$.

- If $A_2$ is convolution with $k$ followed by element-wise multiplication with a mask image, then $A_2^T A_2$ is
  - Convolution with $k$
  - Multiply by mask
  - Multiply by mask again
  - Convolution with flipped version of $k$

- So even when we can't form $Q$, we can carry out the actions $QY$, as well as $Z^T QY$

- Compute $QY$
- Take element-wise product of the result with $Z$ and sum.

Solve by the Conjugate Gradient method.

- Generic algorithm for solving $Qx = b$ for symmetric positive definite $Q$.
- Useful when you can multiply by $Q$ but not 'form' it.

Basic Idea

- For a given set of vectors $\{p_1, p_2, \ldots, p_N\}$
  - that are same size as $x$
  - linearly independent
  - $N = \text{dimensionality of } x$

- We can write any $x = \sum_i \alpha_i p_i$

- If we also choose the vectors to be 'conjugate' such that $p_i^T Q p_j = 0$ for $i \neq j$.

  $Qx = b \rightarrow p_k^T Qx = p_k^T b \rightarrow \alpha_i p_i^T Q p_k = p_k^T b \rightarrow \alpha_i = \frac{p_k^T b}{p_k^T Q p_k}$
CONJUGATE GRADIENT

Iterative Algorithm

- Begin with some guess $x_0$ for $x$ (say all zeros)
- $k = 0, r_0 \leftarrow b - Qx_0, \quad p_0 \leftarrow r_0$
- Repeat
  - $\alpha_k \leftarrow \frac{r^T_k r_{k+1}}{p^T_k Q p_k}$
  - $x_{k+1} = x_k + \alpha_k p_k$
  - $r_{k+1} = r_k - \alpha_k Q p_k$
  - $\beta_k = \frac{r^T_{k+1} r_{k+1}}{r^T_k r_k}$
  - $p_{k+1} = r_{k+1} + \beta_k p_k$
  - $k = k + 1$


Think about what you would do when: $Q = (A_k^T A_k + \lambda (A_k^T A_k + A_k^T A_k))^T, \quad b = A_k^T Y$
For simplicity, have discrete wavelengths. Approximate integration as summation.

\[
\begin{align*}
    r[n] &= \int L(\lambda) I_R(\lambda) d\lambda \\
    g[n] &= \int L(\lambda) I_G(\lambda) d\lambda \\
    b[n] &= \int L(\lambda) I_B(\lambda) d\lambda
\end{align*}
\]

\[L(\lambda, n) \rightarrow L[\lambda, n] \text{ or } L[n] \in \mathbb{R}^B\]

Think of the incident light being a \(B \gg 3\) channel image \(L[n]\).

\[L(\lambda, n) \rightarrow L[\lambda, n] \text{ or } L[n] \in \mathbb{R}^B\]

There are cameras that actually capture such “hyperspectral” images.

\[X[n] = \Pi^T L[n], \quad \Pi = [I_R \quad I_G \quad I_B] \quad (B \times 3 \text{ Matrix})\]

Think of the incident light being a \(B \gg 3\) channel image \(L[n]\).

- 3 dimensional projection from higher dimensional space.
- Invariant to changes in the “null space” of \(\Pi\).