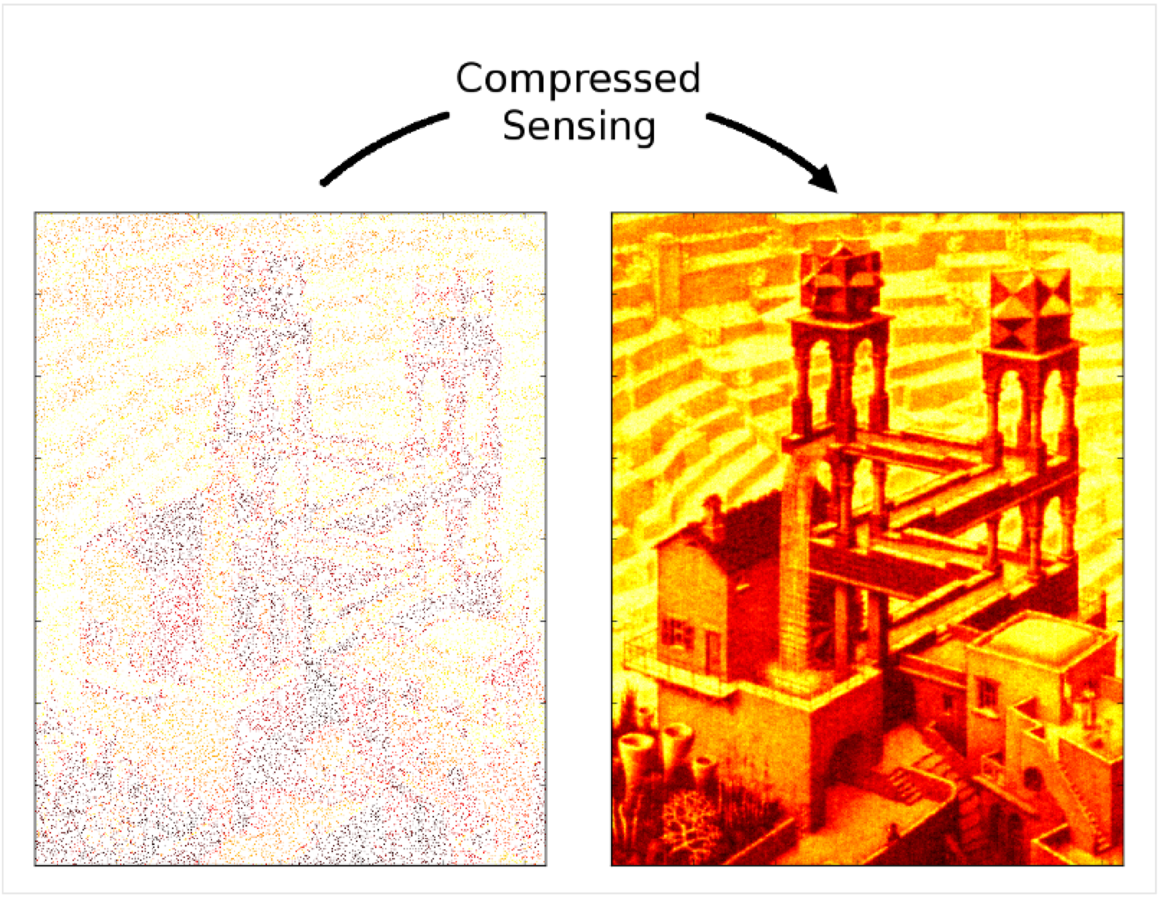
Compressed Sensing in Python

Posted May 26, 2016 in Mathematics, Python. *Updated July 20, 2016.*

Share



Compressed Sensing

In this post I'll be investigating *compressed sensing* (also known as compressive sensing, compressive sampling, and sparse sampling) in Python. Since the idea of compressed sensing can be applied in wide array of subjects, I'll be focusing mainly on how to apply it in one and two dimensions to things like sounds and images. Specifically, I will show how to take a highly incomplete data set of signal samples and reconstruct the underlying sound or image. It is a very powerful technique.

L^1 vs. L^2 Fitting

As you might know, there are many different types of norms. Perhaps the most common and widely recognized one is the L^2 norm:

$$\|\vec{x}\|_2 = \left(\sum_{i=0}^{n} x_i^2\right)^{1/2}$$

The L^2 norm is nice because it is easily calculated, easily differentiated, and it has intuitive appeal (e.g., the norm of a vector is its length). A lot of very important algorithms and methods rely on the L^2, including least squares fitting.

That said, the L^2 norm isn't the goto solution for everything. The other norms also have many interesting and useful properties. Consider the L^1 norm:

$$\|\vec{x}\|_1 = \sum_{i=0}^{n} |x_i|$$

Instead of squaring each element, it simply takes its absolute value. Although the absolute value is annoying in the sense that it often introduces discontinuities in its derivatives, it does have some unique properties when compared to the squaring that takes place in the L^2 norm. Compressed sensing is all about exploiting these properties.

Let's visualize some data with Python to see what I'm talking about.

```
# make sure you've got the following packages installed
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import scipy.optimize as spopt
import scipy.fftpack as spfft
import scipy.ndimage as spimg
import cvxpy as cvx
```

First what we're going to do is create some arbitrary linear data including some noise. Let's use the made-up equation:

$$y = \frac{1}{5}x + 3 + \epsilon$$

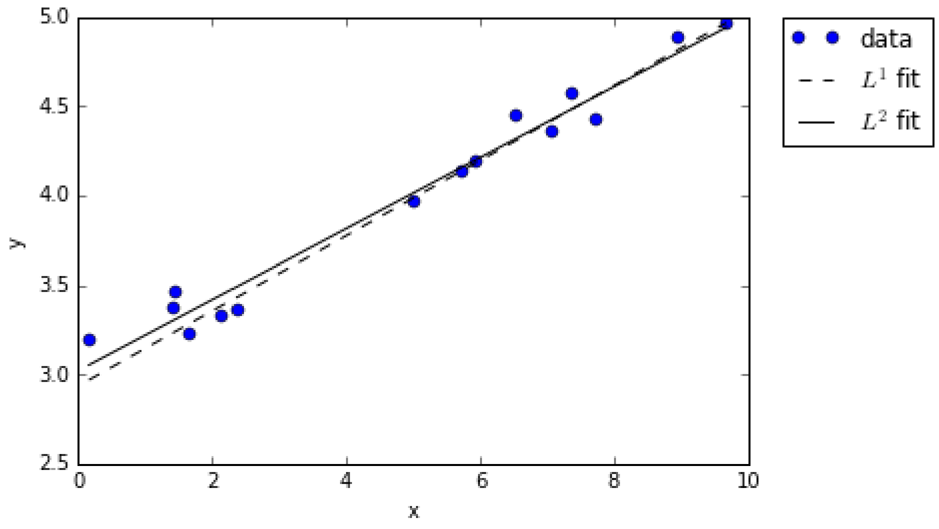
where ϵ is some normally distributed error with standard deviation $\sigma = 0.1$.

```
# generate some data with noise
x = np.sort(np.random.uniform(0, 10, 15))
y = 3 + 0.2 * x + 0.1 * np.random.randn(len(x))
```

Now let's fit two lines to the data samples. For the first line, we'll use the L^1 norm as the criterion for a good fit; for the second line, we'll use the L^2 norm.

```
# find L1 line fit
l1_fit = lambda x0, x, y: np.sum(np.abs(x0[0] * x + x0[1] - y))
```

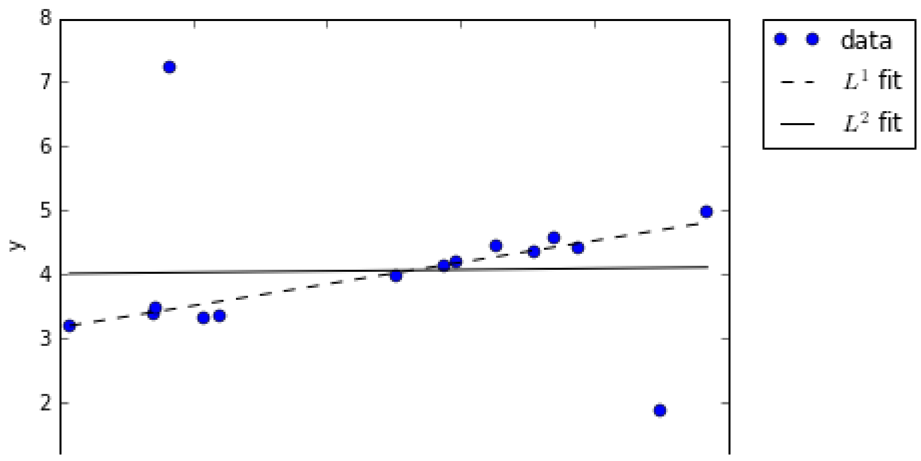
```
xopt1 = spopt.fmin(func=l1_fit, x0=[1, 1], args=(x, y))

# find L2 line fit


l2_fit = lambda x0, x, y: np.sum(np.power(x0[0] * x + x0[1] - y, 2))
xopt2 = spopt.fmin(func=l2_fit, x0=[1, 1], args=(x, y))
```

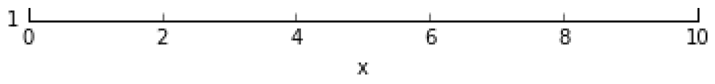


Notice that both of the fits seem to do a pretty good job fitting the data. Sure, they don't line up exactly, but they both are reasonable approximations given the noise.

Now, let's get a tad crazy and add some outliers. In other words, let's perturb a couple of the points, moving them far away from the lines. This isn't actually all that out of the ordinary if you think about it. Outliers frequently occur in real world data, causing all kinds of headaches.

```
# adjust data by adding outlyers
y2 = y.copy()
y2[3] += 4
y2[13] -= 3

# refit the lines
xopt12 = spopt.fmin(func=l1_fit, x0=[1, 1], args=(x, y2))
xopt22 = spopt.fmin(func=l2_fit, x0=[1, 1], args=(x, y2))
```





When we re-plot the L^1 and L^2 fits we see something interesting: the L^1 fit remained true to the overall trend in the data, while the L^2 fit seemed to get “corrupted” by the outliers. Why does this happen? It comes down to the fact that L^2 error gets squared, while L^1 error does not. When you fit a line to data using an L^2 interpretation of error, the displacement of outliers has a disproportional impact because their already-big errors are get getting squared. Just look at the distance of the two outliers in our example and imagine squaring them – of course it’s not surprising that the L^2 line gets skewed!

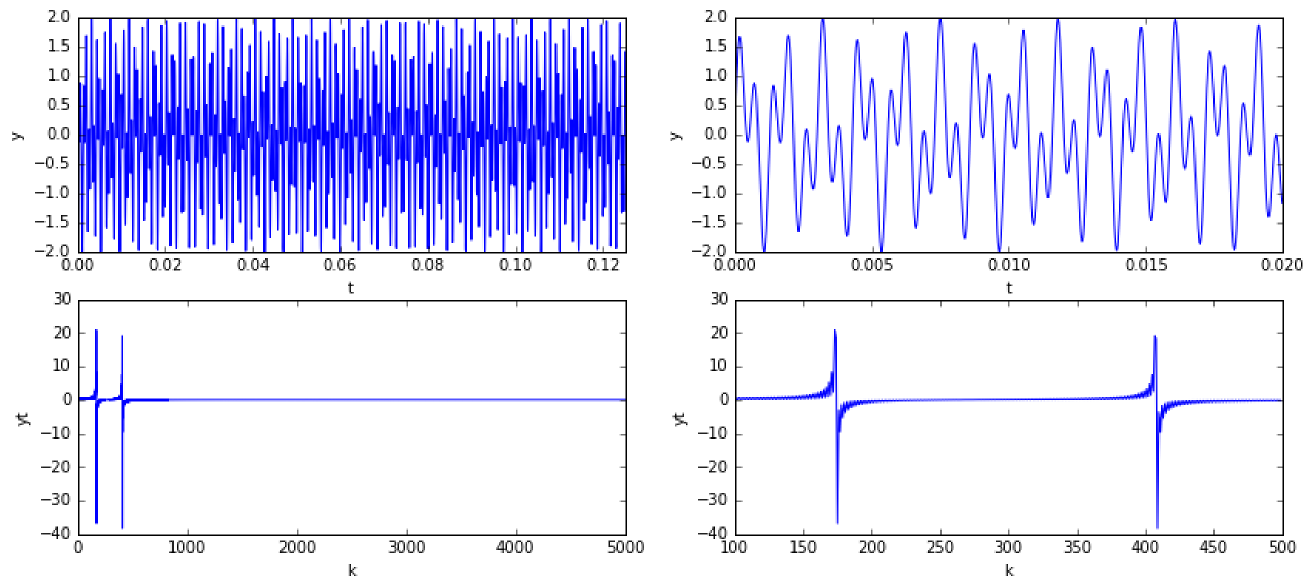
However, when using an L^1 interpretation of error, the outliers contribute no more than their displacement. The result is a cleaner fit that more closely matches our intuition of what a good fit should look like. It’s this interesting property that opens the door to compressed sensing.

Reconstruction of a Simple Signal

In this example (borrowed from Kutz[1]), we will create an artificial sound wave, sample 10% of it, and reconstruct the original signal from the sample of 10%. This is one dimensional compressed sensing.

First, create a signal of two sinusoids.

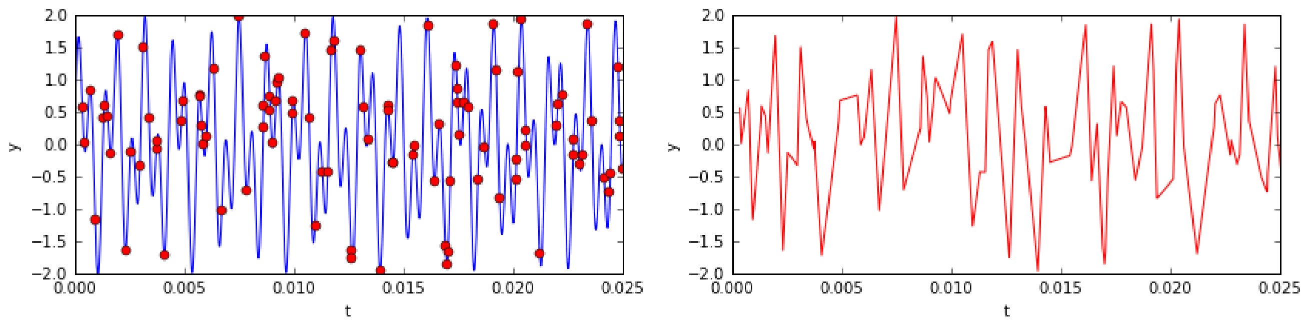
```
# sum of two sinusoids
n = 5000
t = np.linspace(0, 1/8, n)
y = np.sin(1394 * np.pi * t) + np.sin(3266 * np.pi * t)
yt = spfft.dct(y, norm='ortho')
```



In the plots above, we see that the signal has a clear pattern, yet is non-trivial. The plots in the top row are of the signal in the temporal domain at different scales. The plots in the bottom row are of the signal in the spectral domain (i.e., the signal's frequency content). Considering the frequency domain in particular, we note that the spectrum is mostly zero except for the two spikes representing the two sine frequencies.

Now imagine sampling 10% of the temporal signal (see below). You'd have a data set that, to the naked eye, would look like nonsense. The underlying signal is would still be the same, as would be its frequency content (mostly zeros, with the exception of two spikes). One might ask if it is somehow possible to extract those two dominant frequencies from the incomplete data so that we might reconstruct the signal? The answer is yes!

```
# extract small sample of signal
m = 500 # 10% sample
ri = np.random.choice(n, m, replace=False) # random sample of indices
ri.sort() # sorting not strictly necessary, but convenient for plotting
t2 = t[ri]
y2 = y[ri]
```



Compressed sensing in this context is made possible by the fact that the signal's frequency content is highly sparse. This is where the L^1 norm comes into play. What we want to do is, out of all possible signals, locate the *simplest* one that matches up with the known data. In other words, we want to use a minimization routine to find a set of frequencies satisfying two conditions: (a) the underlying signal matches up exactly (or as closely as possible) with that of our data; and (b) the L^1 norm of the frequencies is minimized. Such a routine will yield a sparse solution – exactly what we want.

In Python, there are a couple ways to accomplish this. Perhaps the easiest is to utilize the convex optimization library CVXPY. Use the code below to minimize the norm of the signal's frequencies with the constraint that candidate signals should match up exactly with our incomplete samples.

```
# create idct matrix operator
A = spfft.idct(np.identity(n), norm='ortho', axis=0)
A = A[ri]

# do L1 optimization
vx = cvx.Variable(n)
objective = cvx.Minimize(cvx.norm(vx, 1))
constraints = [A*vx == y2]
```

```
prob = cvx.Problem(objective, constraints)
result = prob.solve(verbose=True)
```

You might be asking: *what the hell is that A matrix?* Well, it's the key to the whole party. Let me explain.

In order to perform the minimization, we must somehow finagle our problem into a linear system of equations:

$$Ax = b$$

Specifically, we want to derive a matrix A that can be multiplied with a solution candidate x to yield b, a vector containing the data samples. In the context of our current problem, the candidate solution x exists in the frequency domain, while the known data b exists in the temporal domain. Clearly, the matrix A performs both a sampling and a transformation from spectral to temporal domains.

Compressed sensing really comes down to being able to correctly derive the A operator. Fortunately, there's a methodology. Start off by letting f be the target signal in vector form (if your signal is 2-dimensional or higher, flatten it) and ϕ be the sampling matrix. Then:

$$b = \phi f$$

Now let ψ be the matrix that transforms a signal from the spectral domain to the temporal domain. Given the solution x in the frequency domain, it follows that:

$$\psi x = f$$

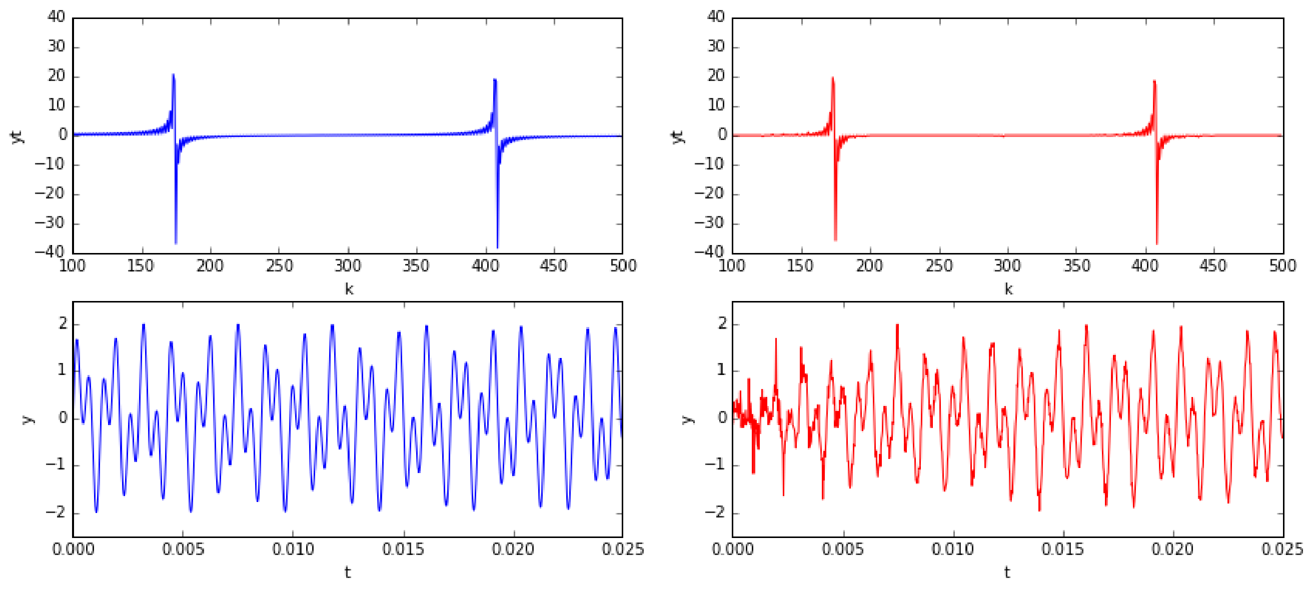
Combining the two equations yields:

$$Ax = b \quad \text{where } A \equiv \phi\psi$$

So, A is simply made up of rows sampled from the domain transform matrix ψ. The ψ matrix is easy to construct – it is the inverse discrete cosine transform acting upon the columns of the identity matrix. The matrix product ψx is the equivalent to doing `idct(x)`.

Now that we've constructed the A matrix and run the minimization, we can reconstruct the signal by transforming the solution out of the frequency domain and back into the temporal. Below, on the left, is the original signal and its frequency content. On the right is our L^1 approximation. I'd say that's pretty good for only using 10% of the data!

```
# reconstruct signal
x = np.array(vx.value)
x = np.squeeze(x)
sig = spfft.idct(x, norm='ortho', axis=0)
```



One problem that stands out is that the quality of the reconstruction degrades noticeably at and around $t = 0$. This is probably due to our sample interval violating the periodic boundary condition requirements of the cosine transform. Of course, given an arbitrary signal sample without any prior knowledge of its nature, it would be hard *not* to violate periodic boundary conditions. The good news is that now we have some very clear indications of the true signal's frequencies. If desired, we could go back and resample the signal within an interval that satisfies periodic boundaries.

Reconstruction of an Image (a 2D Signal)

Now let's use what we learned from the 1-dimensional case to do compressed sensing in 2-dimensions. This is where the real fun begins because we can now try and reconstruct images.

Below, we will use exactly the same methodology as before to randomly sample and reconstruct the image *Waterfall* by M. C. Escher (approx. 1200 by 1600 pixels). Due to memory limitations imposed by the A matrix, we'll start off by considering a downsized version of the image (approx. 50 by 65 pixels). In the section that follows we'll extend the routine to handle large images.

Note that SciPy doesn't provide 2D versions of `dct` or `idct`. However, they can be easily constructed by recognizing that the 2D discrete cosine transform is nothing more than a `dct` acting upon the rows of x followed by a second `dct` action upon its columns (or vice versa):

$$dct\left(dct(x^T)^T\right) \equiv dct\Big(dct(x)^T\Big)^T$$

As a personal preference, I like to tell SciPy's `dct` and `idct` methods to act on the columns of a matrix (as opposed to the default behavior of acting on the rows). First of all, this keeps the

Python code consistent with that of MATLAB. Second, it makes building matrix operators more intuitive (to me at least). For example, if we let Y be an m by n matrix, with I_m and I_n being identity matrices of size m and n respectively, then

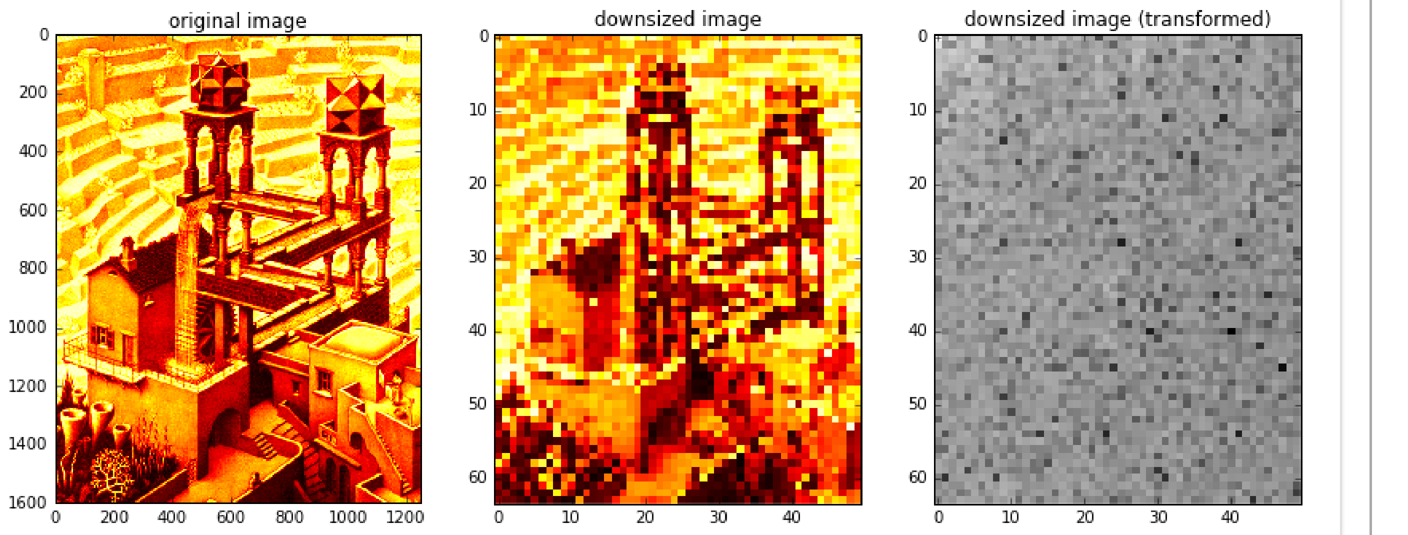
$$dct\,(Y,\ \text{axis=0}) \equiv dct\,(I_m,\ \text{axis=0}) \cdot Y \quad (\text{MATLAB default})$$

whereas

$$dct\,(Y,\ \text{axis=1}) \equiv Y \cdot dct\,(I_n,\ \text{axis=1}) \quad (\text{SciPy default})$$

Either version can be made to work, but I feel like the first one is cleaner because it naturally keeps the matrix operator in front of the operand. Whenever I refer to the `dct` or `idct`, assume that I mean the `axis=0` variety.

```python
def dct2(x):
    return spfft.dct(spfft.dct(x.T, norm='ortho', axis=0).T, norm='ortho', axis=0)

def idct2(x):
    return spfft.idct(spfft.idct(x.T, norm='ortho', axis=0).T, norm='ortho', axis=0)

# read original image and downsize for speed
Xorig = spimg.imread('escher_waterfall.jpeg', flatten=True, mode='L') # read in grayscale
X = spimg.zoom(Xorig, 0.04)
ny,nx = X.shape
```



As in the previous section, we'll take a random sample of image indices, forming our b matrix. Then, we'll generate our A matrix.

Creating the A matrix for 2D image data takes a little more ingenuity than it did in the 1D case. In the derivation that follows, we'll use the Kronecker product $\otimes$ and the fact that the 2D discrete cosine transform is *separable* to produce our operator A.

Let X be an image in the spectral domain and $D_i = idct(I_i)$, where I_i is the identity matrix of size i. Then:

$$\begin{aligned} idct2(X) &= idct\left(idct(X^T)^T\right) \\ &= D_m\left(D_n X^T\right)^T \\ &= D_m X D_n^T \end{aligned}$$

If $vec(X)$ is the vector operator that stacks columns of X on top of each other, then:

$$\begin{aligned} vec\left(D_m X D_n^T\right) &= (D_n \otimes D_m)\, vec(X) \\ &= (D_n \otimes D_m)\, x \quad \text{where and } x \equiv vec(X) \end{aligned}$$

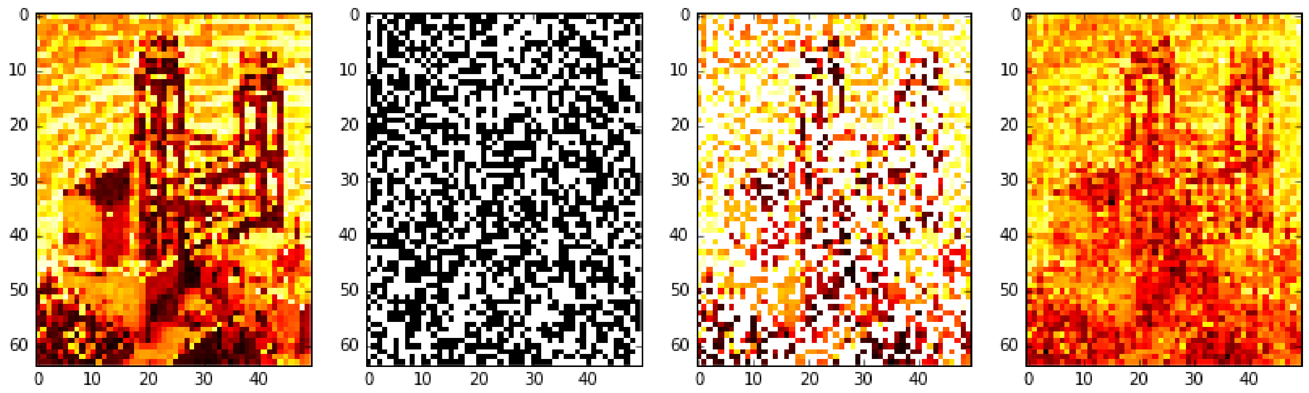
Clearly, the Kronecker product is our desired transformation matrix ψ. Therefore, our matrix A becomes $A = \phi(D_n \otimes D_m)$, where ϕ is the sampling matrix. You can calculate the Kronecker product in Numpy with `numpy.kron`. The main problem with this method is that the Kronecker product can become truly massive very quickly. If your target image is m by n and you're taking k samples, then the A matrix has a size of $(mnk)^2$. That said, for small images it will be fine.

```
# extract small sample of signal
k = round(nx * ny * 0.5) # 50% sample
ri = np.random.choice(nx * ny, k, replace=False) # random sample of indices
b = X.T.flat[ri]
b = np.expand_dims(b, axis=1)

# create dct matrix operator using kron (memory errors for large ny*nx)
A = np.kron(
    spfft.idct(np.identity(nx), norm='ortho', axis=0),
    spfft.idct(np.identity(ny), norm='ortho', axis=0)
    )
A = A[ri,:] # same as phi times kron

# do L1 optimization
vx = cvx.Variable(nx * ny)
objective = cvx.Minimize(cvx.norm(vx, 1))
constraints = [A*vx == b]
prob = cvx.Problem(objective, constraints)
result = prob.solve(verbose=True)
Xat2 = np.array(vx.value).squeeze()
```

Reconstruct the signal and visualize!

```
# reconstruct signal
Xat = Xat2.reshape(nx, ny).T # stack columns
Xa = idct2(Xat)
```

```
# confirm solution
if not np.allclose(X.T.flat[ri], Xa.T.flat[ri]):
    print('Warning: values at sample indices don\'t match original.')


# create images of mask (for visualization)
mask = np.zeros(X.shape)
mask.T.flat[ri] = 255
Xm = 255 * np.ones(X.shape)
Xm.T.flat[ri] = X.T.flat[ri]
```



Okay, the results aren't fabulous. The original image on the far left is barely intelligible as it is. Resolution was low, so we had to take a large-ish sample of 50% (the boolean mask is shown middle left; the masked image is middle right). Regardless, it is clear the procedure worked: the reconstructed image on the far right definitely approximates the original, be it poorly.

Optimization and Scalability

Considering our working proof-of-concept, there are a lot of ways it might be improved. The Kronecker-based method, although easy to implement, proves unusable for large images. What other methods are there?

Convex optimization using CVXPY isn't necessarily the only way to find the L^1 minimum. A little bit of online research led me to the L-BFGS algorithm[6] and its variant, the OWL-QN[3]. The OWL-QN algorithm is of particular interest to us, as it allows one to fit a L^1 regularized model by minimizing a function of the form:

$$f(x) = g(x) + C\|x\|_1$$

where f is a differentiable convex loss function and C is a constant. In our case, we might define f to be the least squares objective function, which is simply the L^2 norm of the residual squared:

$$f(x) = \|Ax - b\|_2^2$$

The gradient of which is:

$$\nabla f(x) = 2(A^T Ax - A^T b)$$

$$\nabla f(x) = 2(A^T Ax - A^T b)$$

Now all that remains is to code it up! After trying several different options, I ended up settling on using libLBFGS (written in C) for its OWL-QN implementation. To make it accessible from

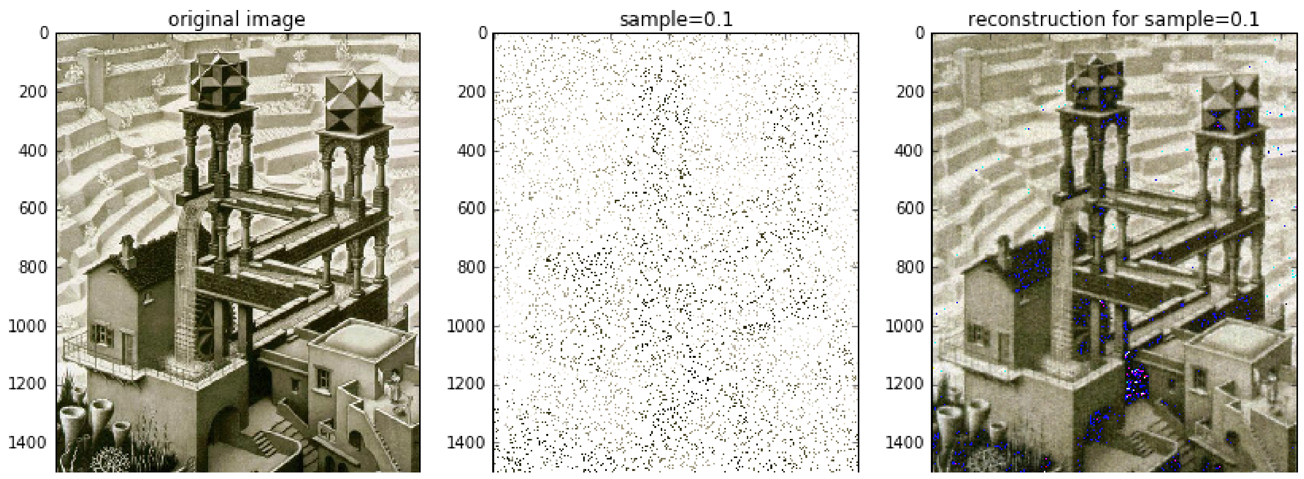
Python, I wrapped it using the C APIs for Python and Numpy. You can find my implementation at PyLBFGS. See the project README for installation instructions and basic use. Let me know if you encounter bugs.

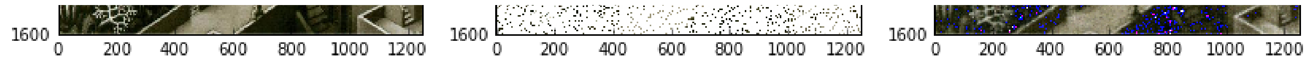
The nice thing about libLBFGS (and by extension PyLBFGS) is that you can define the objective function anyway you like. In other words, we aren't constrained to follow the $Ax = b$ model blindly. All that matters is the we are able to calculate the norm of the residual squared and its gradient. We need not generate A at all!

The following code explains what I mean better than I could with words. Take special note of the `evaluate` callback passed to the OWL-QN algorithm.

```python
from pylbfgs import owlqn

def evaluate(x, g, step):
    """An in-memory evaluation callback."""

    # we want to return two things:
    # (1) the norm squared of the residuals, sum((Ax-b).^2), and
    # (2) the gradient 2*A'(Ax-b)

    # expand x columns-first
    x2 = x.reshape((nx, ny)).T

    # Ax is just the inverse 2D dct of x2
    Ax2 = idct2(x2)

    # stack columns and extract samples
    Ax = Ax2.T.flat[ri].reshape(b.shape)

    # calculate the residual Ax-b and its 2-norm squared
    Axb = Ax - b
    fx = np.sum(np.power(Axb, 2))

    # project residual vector (k x 1) onto blank image (ny x nx)
    Axb2 = np.zeros(x2.shape)
    Axb2.T.flat[ri] = Axb # fill columns-first

    # A'(Ax-b) is just the 2D dct of Axb2
    AtAxb2 = 2 * dct2(Axb2)
    AtAxb = AtAxb2.T.reshape(x.shape) # stack columns

    # copy over the gradient vector
    np.copyto(g, AtAxb)

    return fx
```

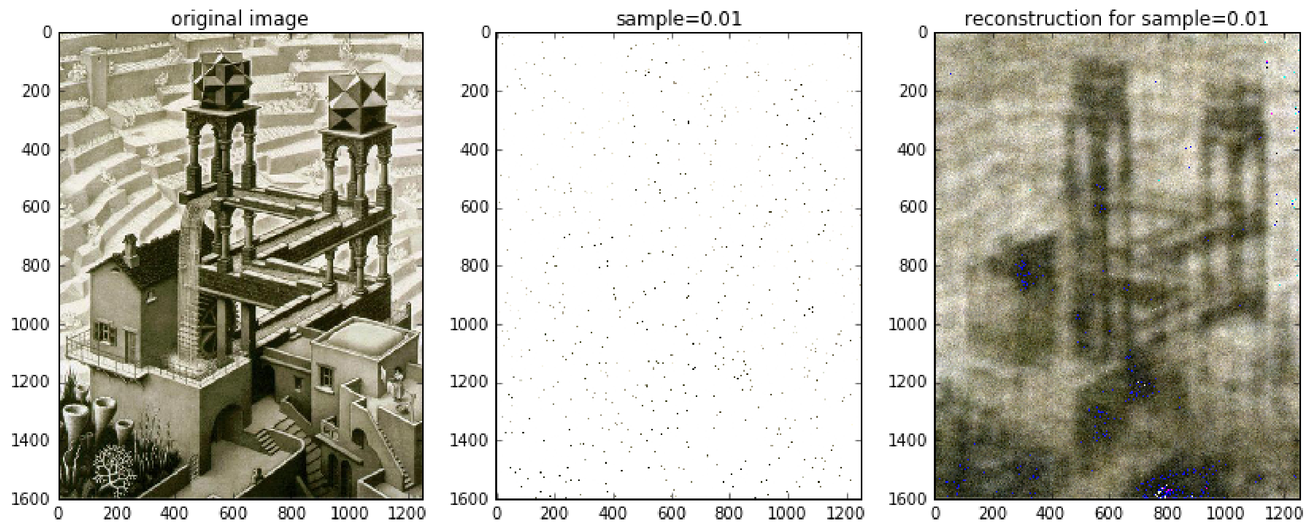
```
# fractions of the scaled image to randomly sample at
sample_sizes = (0.1, 0.01)


# read original image
Xorig = spimg.imread('escher_waterfall.jpeg')
ny,nx,nchan = Xorig.shape

# for each sample size
Z = [np.zeros(Xorig.shape, dtype='uint8') for s in sample_sizes]
masks = [np.zeros(Xorig.shape, dtype='uint8') for s in sample_sizes]
for i,s in enumerate(sample_sizes):

    # create random sampling index vector
    k = round(nx * ny * s)
    ri = np.random.choice(nx * ny, k, replace=False) # random sample of indices

    # for each color channel
    for j in range(nchan):

        # extract channel
        X = Xorig[:,:,j].squeeze()

        # create images of mask (for visualization)
        Xm = 255 * np.ones(X.shape)
        Xm.T.flat[ri] = X.T.flat[ri]
        masks[i][:,:,j] = Xm

        # take random samples of image, store them in a vector b
        b = X.T.flat[ri].astype(float)

        # perform the L1 minimization in memory
        Xat2 = owlqn(nx*ny, evaluate, None, 5)

        # transform the output back into the spatial domain
        Xat = Xat2.reshape(nx, ny).T # stack columns
        Xa = idct2(Xat)
        Z[i][:,:,j] = Xa.astype('uint8')
```





The fast C implementation of OWL-QN allows us to process samples of the entire *Waterfall* image without any scaling required. And instead of doing everything in gray-scale like earlier, we can now afford to process each of the image's three color channels. The solution shown above really demonstrates the power of compressed sensing. The original, full-color image is shown on the left. The middle image is the random 10% sample. The solution image is on the right. Although the solution contains some noticeable blemishes due to bad color channel mixing, the overall accuracy is uncanny. Furthermore, with a little extra care and attention, those blemishes might be removed – either via post-processing (e.g., a Gaussian filter) or via an improved compressed sensing implementation that takes into account color channels. One possibility is to try and determine the probable color palette beforehand and then incorporate it into the compressed sensing routine.

Just for kicks and giggles, I also included an image reconstructed from 1% of the available data. It's definitely blurred, but nonetheless recognizable!



References

1. Kutz, J. N. "Data-driven modeling and scientific computing: Methods for Integrating Dynamics of Complex Sys-tems and Big Data." (2013). ↩
2. Candè, Emmanuel J., and Michael B. Wakin. "An introduction to compressive sampling." Signal Processing Magazine, IEEE 25.2 (2008): 21-30. ↩
3. Andrew, Galen, and Jianfeng Gao. "Scalable training of L1-regularized log-linear models." Proceedings of the 24th international conference on Machine learning. ACM, 2007. ↩
4. Wikipedia contributors. "Compressed sensing." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 26 Mar. 2016. Web. 26 May. 2016. ↩
5. Wikipedia contributors. "Kronecker product." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 19 May. 2016. Web. 26 May. 2016. ↩
6. Wikipedia contributors. "Limited-memory BFGS." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 16 May. 2016. Web. 26 May. 2016. ↩

Tags: c-api compressed compressive dct dct2 escher fitting idct idct2 ipython kron kronecker l1 l2 lbfgs minimization norm numpy optimization owl-qn pylbfgs sampling scipy sensing signal sparse waterfall

Have a question, comment, or concern about this post? Contact me!

Previous post | Next post

Sitemap