Compressive Sensing

Compressive sensing is broadly where we perform compression as part of sensing, but it has become synonomous with an idea that we might call sparse signal recovery from randomly projected data.

Compressive (or compressed) sensing leads on from the talk last week about the Johnson-Linderstrauss lemma and dimension reduction.

Key things to remember from last time

1. Wide data is data that lives in a very high dimensional space, and it is common.
2. The Johnson-Lindenstrauss lemma provides two insights:
   - we can do dimension reduction that doesn't depend on the input dimension
   - random matrix projects provide a means to do dimension reduction
3. But, distances in simplisitc high-D space are not intuitive.

But actually, most wide data actual lives in a much smaller space that the naive embedding. For instance, a 20 megapixel image lives in a 3 x 20 million dimensional space, but almost all points in that space are not valie images – they are just noise. Real images have qualities like blocks of smoothly changing colours, and distinct edges. However, the space that the images lie on is not a simple linear subspace.

So today I will put together a few ideas:

- sparsity and compression
- representation and transformation
- reconstruction, inverse problems, and regularisation

to explain what compressed sensing is about.

As before, you may not end up using it specifically but the ideas expressed here are crucial to a deep understanding of how and why many machine learning techniques work.

Sparsity and compression

A signal is sparse if most of its values are zero. In formal math terms: a signal $\mathbf{x} \in \mathbb{R}^N$ is sparse if

$$||\mathbf{x}||_0 \ll N, \tag{1}$$

where $||\mathbf{x}||_p$ denotes the L^p norm:

$$||\mathbf{x}||_p = \left(\sum_{i=1}^{N} |x_i|^p \right)^{1/p}, \tag{2}$$

with limiting cases cases $||\mathbf{x}||_0 = \sum_{i=1}^{N} I(|x_i| > 0)$ and $||\mathbf{x}||_\infty = \max\{x_i\}$. So the L^0 "norm" counts the number of non-zero values.

We call a signal k**-sparse** if $||\mathbf{x}||_0 \leq k$.

https://en.wikipedia.org/wiki/Lp_space

A sparse signal lives on the axes of a space, e.g., in a 3D space, a sparse signal might have one non-zero value, so it lies on one of the axes.

Most signals aren't perfectly sparse, but many are compressible, which means that only a few of the values matter. That could be because most are near zero, or some are "masked" by others.

Signals are compressible because

1. Physics constrains them (e.g., the data are controlled by some dynamic system) and they live in a smaller part of the potential "wide" space.
2. We can't perceive some aspect of the signal, so we may as well drop it and look at the part that matters.

formally, we might defined compressibility by the error from approximating the signal by only k values.

I'll show you an example in a moment.

There are many forms of compressibility, leading to compression algorithms, but we are interested here in lossy compression, where some information is lost in the process, but because of the facts above, the loss is inconsesquential.

Representation and Transformation

It is common that a signal is not obviously compressible. That is, most of the values of the raw vector $\mathbf{x}$ are non-zero, and seem important. A simple example is (once again) and image, where most values seem important.

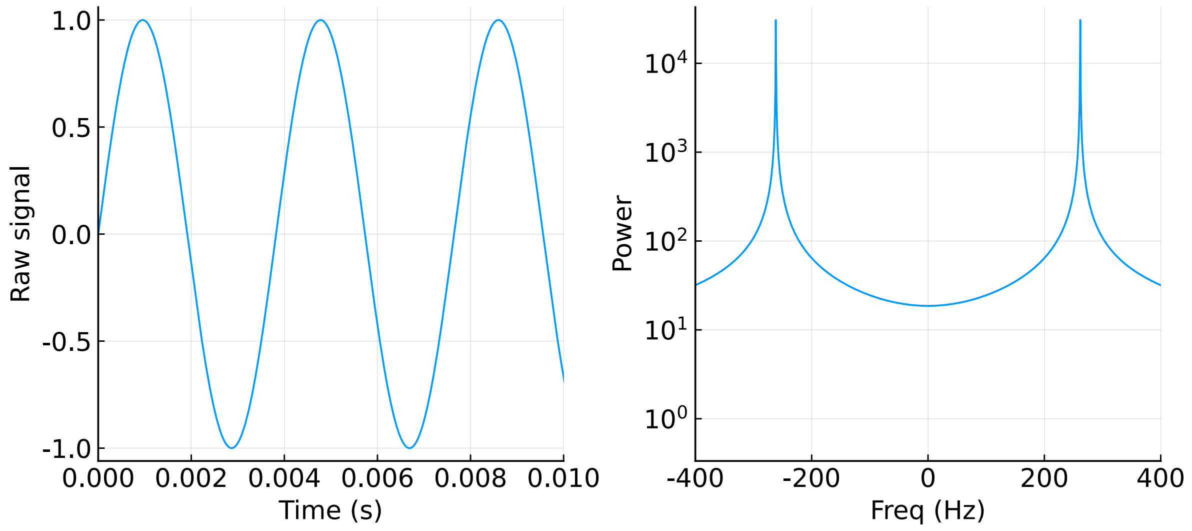
But images are *very* compressible in the frequency domain. Real images have large "flatish" areas of nearly constant colour, i.e., regions where only low-frequency changes are observed. Also, human vision has different sensitivities to different aspects of images:, e.g., we don't see all the details of textures (high-frequency). Thus if we can transform the image into a space where these aspects dominate, we can see if the image is compressible in the sense above.

One of the classic transforms – the Fourier transform – coupled with one of the most important algorithms of the 20th century – the Fast Fourier Transform (FFT) – provide us with a way of doing so.

Here's a quick example of Middle-C = 263.1 Hz transformed in Julia.

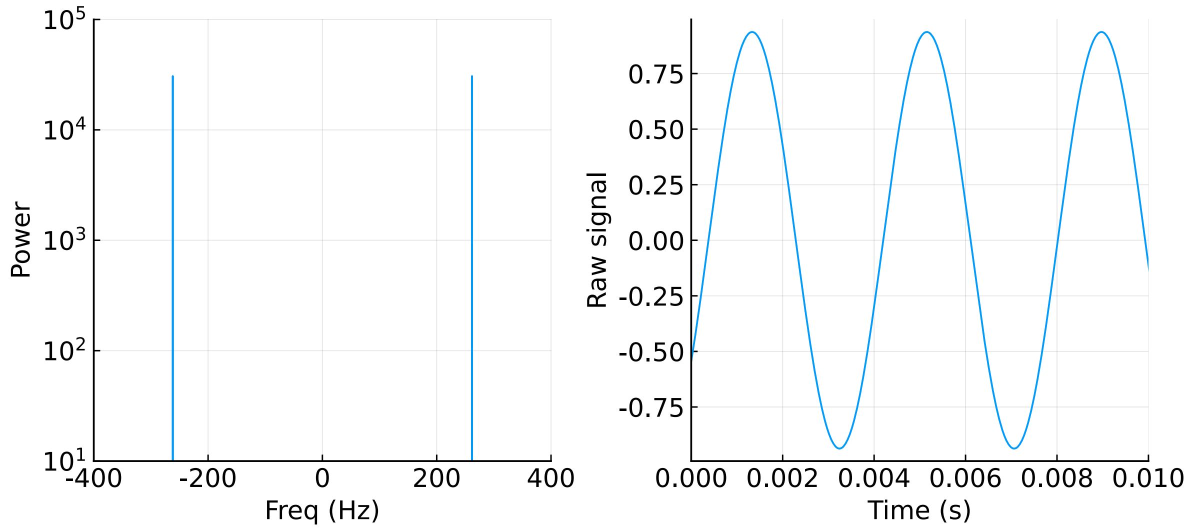
```julia
using Plots
font = Plots.font("Helvetica", 14)
pyplot(guidefont=font, xtickfont=font, ytickfont=font, legendfont=font)
using FFTW # https://juliamath.github.io/FFTW.jl/

N = 2^16
sampling_freq = 44100 # e.g., 44.1 kHz as in CD digital
x = (0:N-1)/sampling_freq

freq = 261.63 # Middle-C in Hz

y = sin.(2π*freq*x)

p1 = plot(x, y; label="")
plot!(p1; xlim=(0, 0.01), xlabel="Time (s)", ylabel="Raw signal" )

zx = fftshift(fftfreq(N, sampling_freq))
zy = fftshift(fft(y))

p2 = plot( zx, abs.(zy); label="" )
plot!(p2; xlim=(-400, 400), xlabel="Freq (Hz)", ylabel="Power", yaxis = (:log10, (10^1, 10^5)) )
```

```julia

plot(p1,p2,layout=(1,2),size=(900,400))
```



You can see that the simple sine function in the original domain transforms into an almost sparse signal in the frequency domain. Note the log-y axis most of the values are orders of magnitude smaller than the two main spikes.

Reconstruction: noting that many values in the DFT here are non-zero, we might ask how important are they? The second example here shows the reverse – take the frequency domain signal above, truncate the values (set anything except the two main peaks to zero) and then perform the inverse FFT.



JPEG compsression does exactly this type of process to images but using the Discrete Cosine Transform (DCT) a real version of the DFT. Roughly, JPG does the followign

1. Colour transform and downsampling
2. Divides the image into small blocks (8x8 is the standard)
3. Performs a DCT on each block
4. Quantize the values of the DCT
5. Encode the quantized bits (lossless compression)

Quantization is the interesting bit. The quantization matrix is an 8x8 block of integers Q, and we take the DCT block C and peform

$$\lfloor C./Q \rfloor. \tag{3}$$

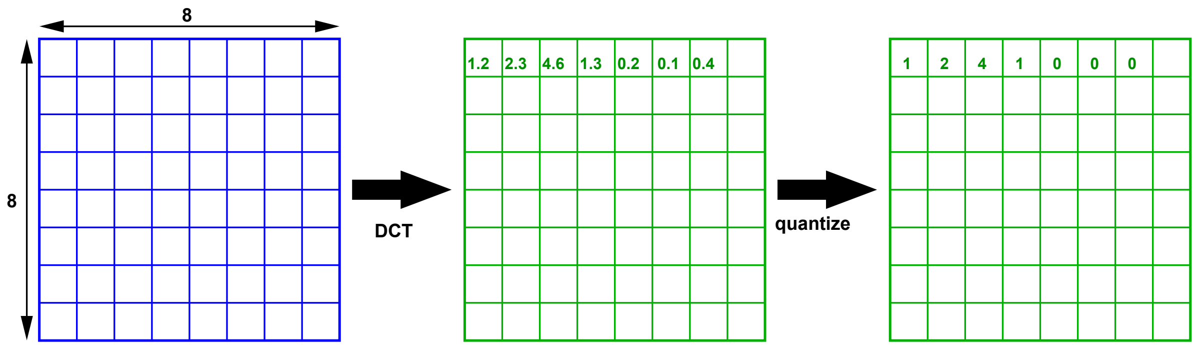
That is, we divide (element wise) the matrix of values for a block by the quantization values. The matrix Q is not uniform. Examples include:

- https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossy/jpeg/coeff.htm
- http://www.robertstocker.co.uk/jpeg/jpeg_new_10.htm

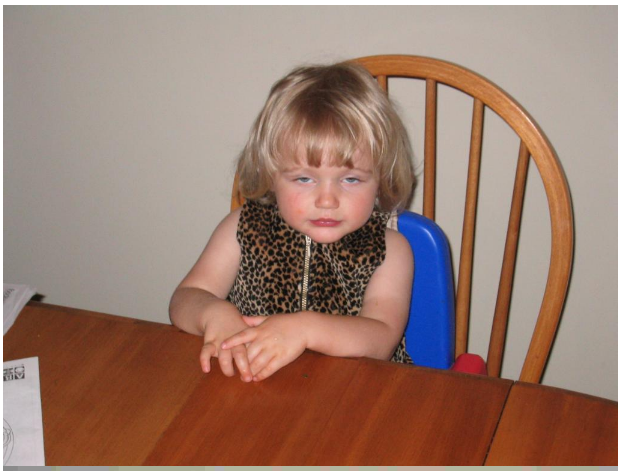
E.g.,

3	5	7	9	11	13	15	17
5	7	9	11	13	15	17	19
7	9	11	13	15	17	19	21
9	11	13	15	17	19	21	23
11	13	15	17	19	21	23	25
13	15	17	19	21	23	25	27
15	17	19	21	23	25	27	29
17	19	21	23	25	27	29	31

Higher frequencies are towards the bottom and right. So these are quantized more. The net affect is that many of the values are truncated to zero. Effectively they are discarded.



JPEG examples: with different levels of quantisation/quality.



Commonly we transform into frequency domain, e.g., using the FFT. This is a clever $O(NlogN)$ algorithm, but we can represent it as a matrix transformation:

$$\mathbf{z} = W\mathbf{x}, \tag{4}$$

where $\mathbf{x}$ is a vector version of our signal (say the image) and $\mathbf{z}$ is the transform. In order that this transform be invertible, we should not lose information, so it should actually be a *change of basis*. There are many ways of performing changes of basis in linear algebra (see Maths 1A) and there are many corresponding transformations, e.g.,

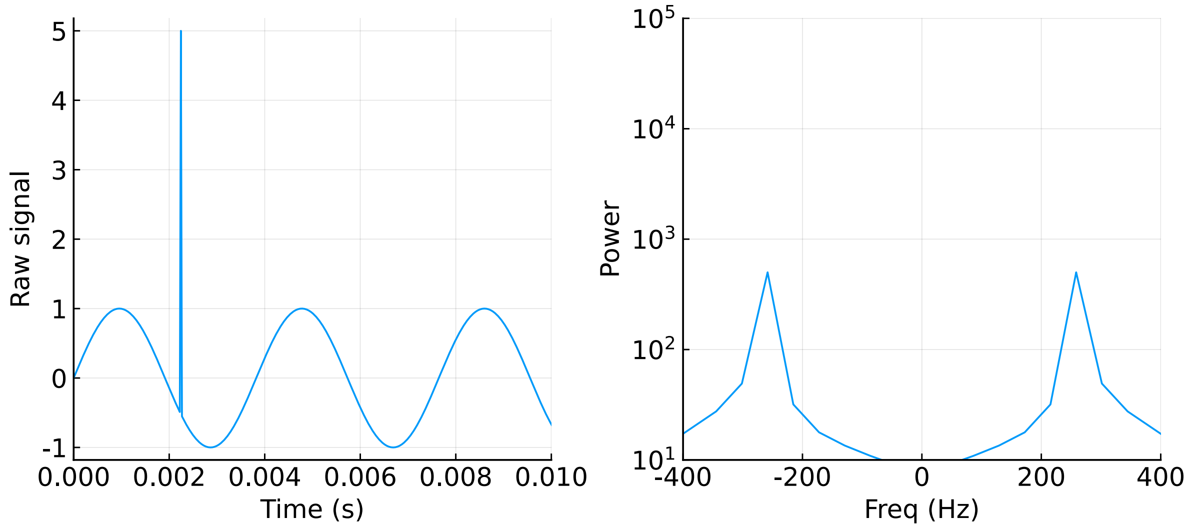
- Wavelets transform into a time-scale space
- PCA transformss into a space where the axes are ordered by an notion of importances in explaining variation.

We can think of a signal as an abstract “thing” and when we write it down as a vector $\mathbf{x}$ we are **representing** it in terms of a set of basis vectors. The standard representation, in the time domain, represents a signal in terms of a set of deltas (spikes), one at each time sample points. This is the basis made of of vectors $\mathbf{e}_i = (0, 0, 0, \ldots, 0, 1, 0, \ldots, 0)$. The Fourier basis vectors are all sinusoids. In general, we might write our basis vectors or functions as $\psi_i(t)$ and our signal is represented as

$$x(t) = \sum_i a_i \psi_i(t), \tag{5}$$

where the coeeficients a_i are put into our vector as $\mathbf{a}$. The Fourier basis function ψ_i are complex sinusoids and the time-domain basis has basis functions that are Dirac deltas.

An interesting question is what you do with a signal like the following:



The spike in the signal is spread out in the Fourier/frequency domain, so the signal is not truly compressible in either. We can't represent this signal sparsely with either spikes (int he time domain) or sinusoids (in the frequency domain) but we can represent it very sparsely as $x(t) = \sin(ft) + \delta(t - t_0)$, where f is the frequency of the sinusoid and t_0 the time of the spike.

We need here to think about more general representations than standard orthogona bases. A key idea is that of an "overcomplete" dictionary and basis pursuit, i.e., the idea of creating a dictionary for representing signals that might include Fourier (sine and cosine) functions as well as deltas. The pursuit would look for a sparse combined representation within this space, but because it is overcomplete we can no longer do this through a simple matrix product.

BTW, there are cameras that do JPEG compression in hardware, as they capture an image so there is never a RAW image that gets compressed. Remember that for later.

Reconstruction, inverse problems, and regularisation

Transformation is just

$$\mathbf{z} = W\mathbf{x}, \tag{6}$$

but if this is an invertible transform then W is a square, non-singular matrix. But if we are working, for instance, with an overcomplete dictionary, the transform matrix W might be non-square. Also in many problems we are forced into measuring (by cost or other practicalities) such that we measure through some transformation.

Hence, a very common problem is reconstruction of a signal $\mathbf{x}$ from a transformed measurement $\mathbf{z}$. Classic problems in this form are sometimes called *tomography* problems by analogy to CAT (Computer Axial Tomomography) scans. These reconstruction problems are inverse problems (I shall stick to linear problems for now). They have a set of general characteristics:

- The equation above is more realistically $\mathbf{z} = W\mathbf{x} + \epsilon$, where ϵ is noise. Hence, there may not actually be an exact solution to the set of equations.
- The matrix W is usually not square and in many cases the length of $\mathbf{z}$ is much smaller than $\mathbf{x}$ so the equations are (massively) underconstrained.

So we can't solve this using the usual linear algebra you have been taught.

The easiest way to create a framework for all of these problems is to consider them as optimisation problems, i.e., we write them as

$$\underset{\mathbf{x}}{\operatorname{argmin}}\, f(\mathbf{x}) \text{ such that } \mathbf{z} = W\mathbf{x}. \tag{7}$$

That is, maximise some objective that determines what a "good" solution should look like such that the measurements are respected. The function $f(\cdot)$ might be, for instance, chosen to push towards a Bayesian prior.

There are two problems with this formulation:

1. If there is noise, the "such that" won't hold, so the approach is not robust even to floating point errors.
2. It leads us to a constrained optimisation problem, which might in some cases be hard.

The usual alternative is to **regularize**. This is a very important concept that you can come at from various directions that I don't have time to explore now, e.g.,

- https://towardsdatascience.com/regularization-an-important-concept-in-machine-learning-5891628907ea
- https://en.wikipedia.org/wiki/Regularization_(mathematics)

but for our purposes we solve a new optimisation problem

$$\underset{\mathbf{x}}{\operatorname{argmin}} f(\mathbf{x}) + \lambda||\mathbf{z} - W\mathbf{x}||. \tag{8}$$

That is we solve a problem that tries to find a good solution that closely satisfies the measurment constaints (the extra term in the objective is a type of Lagrange multiplier).

Note the extra parameter λ. This let's you perform a tradeoff between your "model" implied by $f(\cdot)$ and your measurements. So

- If you think the measurements are very noisy, you migth chose a smaller λ.
- If you think the measurements are very accurate, you might chose a larger λ.

The secret of all of these, however, lies in the details of $f(\cdot)$ and the norm we use $||\cdot||$. There are very many alternatives.

Sensing and compression as one action = compressive sensings

Now we get to put all of these ideas together.

1. Remember that transformation and sensing can happen as one step in hardware (e.g., JPEG) so
   - I will assume that my signal $\mathbf{x}$ is compressible (after some tranformation of the raw signal), and
   - I will also use a random projection matrix A to perform a dimension reduction on the signal to get

   $$\mathbf{y} = A\mathbf{x} \tag{9}$$

   where $\mathbf{x} \in R^n$ and $\mathbf{y} \in R^k$ where $k \ll n$ (i.e., A is a $k \times n$ matrix).
2. Now I want to reconstruct an approximation $\hat{\mathbf{x}}$ from $\mathbf{y}$ and I shall suggest the following:

   $$\underset{\mathbf{x}}{\operatorname{argmin}} \lambda||\mathbf{x}||_0 + ||\mathbf{z} - W\mathbf{x}||_2^2. \tag{10}$$

There are two key ideas that came out of the compressive sensing literature:

1. If the signal is compressible, then the optimisation above will find the best approximation for it. The choice of $f(\mathbf{x}) = ||\mathbf{x}||_0$ is determined by the fact that I know $\mathbf{x}$ is compressible, so I seek to find an approximation of this with as few non-zero values as possible, but it isn't completely obvious that this will work. The results in the literature guarantee it will with high probability.
2. The optimisation problem posed above is intractable (it is NP-hard) but there is a very tractable alternative that will work with high probabilitsy:

$$\underset{\mathbf{x}}{\operatorname{argmin}}\, \lambda||\mathbf{x}||_1 + ||\mathbf{z} - W\mathbf{x}||_2^2. \tag{11}$$

The second problem is a quadratic program and quite tractable even for large problems though there are lots of variations on best ways to morph this and solve it.

The matrix A has to satisfy the RIP (Restricted Isometry Property), but that basically is a condition similar to that in JL and random matrices are good with high probability, e.g., Theorem 4.4 from https://www.jstage.jst.go.jp/article/transcom/E96.B/3/E96.B_685/_pdf/-char/en restates:

Theorem Let $A \in \mathbb{R}^{m\times n}$ be a Gaussian ran-dom matrix having i.i.d. elements of mean 0 and variance $1/m$ or a Bernoulli random matrix having i.i.d. elements equal to $\pm 1/\sqrt{m}$ with probability 1/2.

Then let $\varepsilon, \delta \in (0,1)$ and if

$$m \geq C\delta^{-2}\left(k \ln \frac{n}{k} + \ln \varepsilon^{-1}\right), \tag{12}$$

for a constant $C > 0$, then the constant of the isometry constant of A satisfies $\delta_k \leq \delta$ with probability at least $1 - \varepsilon$.

Here, δ_k is the error (sim to JL) in the sense it is the smallest number such that

$$(1-\delta_k)||\mathbf{x}||_2^2 \leq ||A\mathbf{x}||_2^2 \leq (1+\delta_k)||\mathbf{x}||_2^2, \tag{13}$$

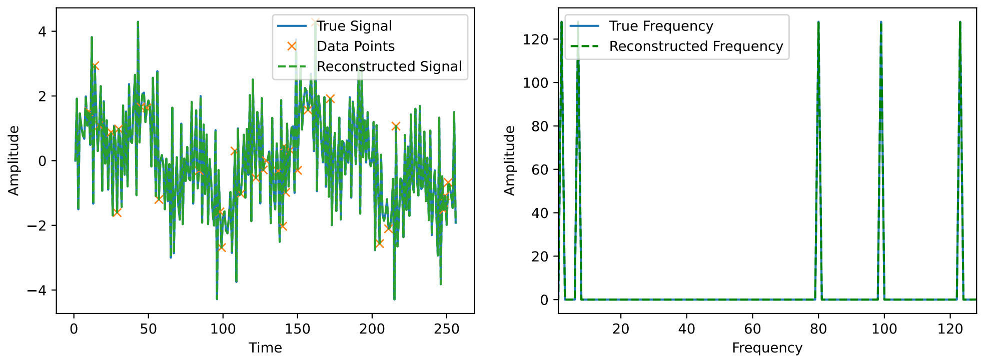
for all k-sparse signals $\mathbf{x}$. So, given a certain maximum amount of sparsity k, we can choose a (lower) dimension m, and just choose a random projection matrix from $\mathbb{R}^n \to \mathbb{R}^m$. The dimension m looks similar to that in the JL lemma but note that n (the original dimension) is present in this, if only through a log function.

Note that here we never actually collect the wide data $\mathbf{x}$. The projection is pre-calculated, and implemented in hardware so we only collect $\mathbf{y}$. We don't need to ever write down the full version of $\hat{\mathbf{x}}$ either as most of it will be zeros, so we can use a sparse array format to only record the non-zero values.

Julia Example 1

https://discourse.julialang.org/t/compressed-sensing-using-structuredoptimization-jl/35098

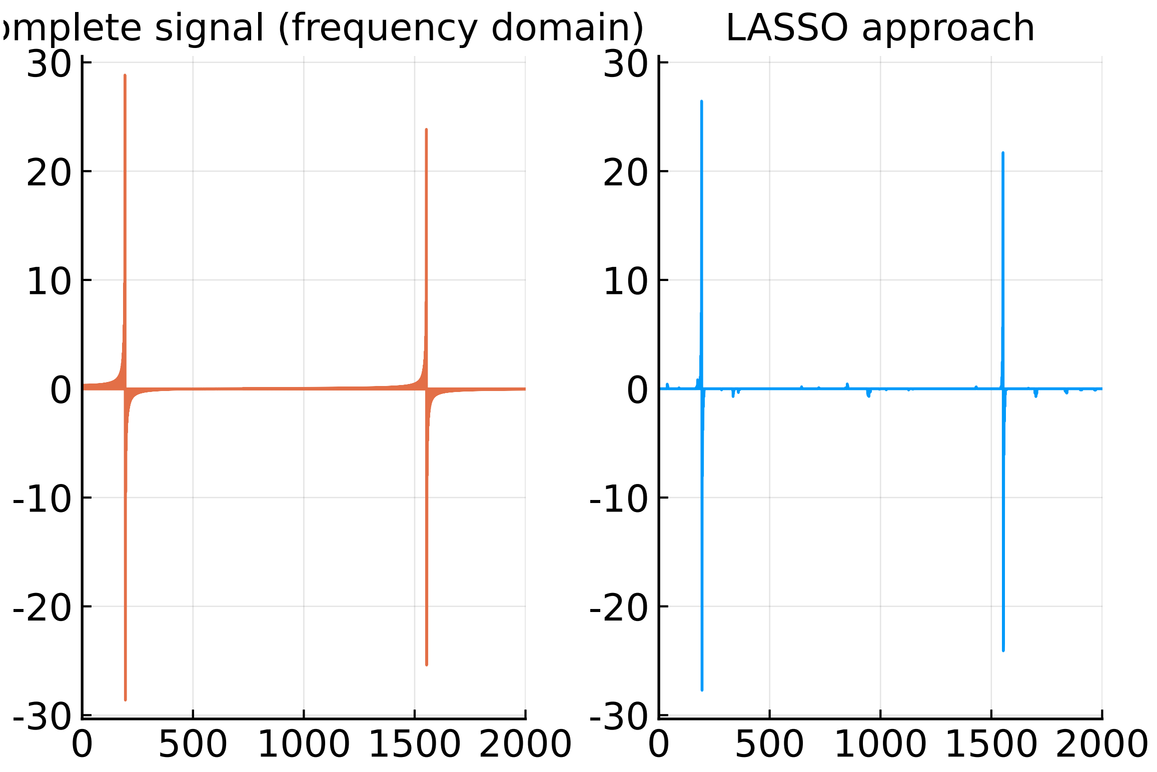
This example isn't quite doing what we are talking about – perhaps it is better called compressive sampling.



Julia Example 2

https://nextjournal.com/yusri/compressed-sensing-in-julia

Likewise this example.



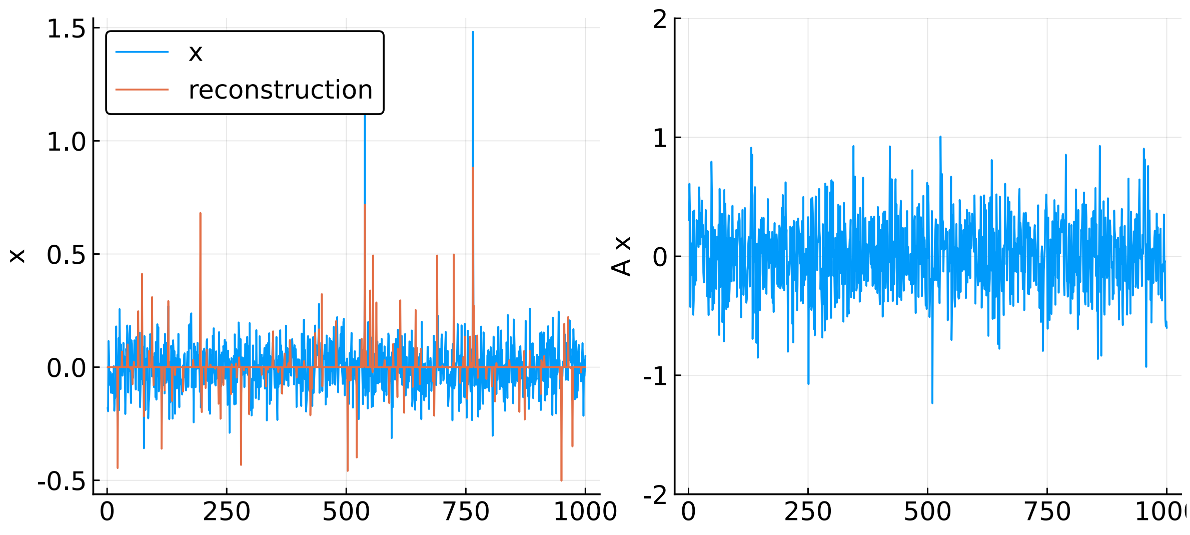
Julia Example 3

```julia
using LinearAlgebra
using Random
using StableRNGs
using StructuredOptimization
using Distributions

# parameters
n = 10_000                # signal length
k = ceil(Int, n/1000)     # sparsity

δ = 0.5      # RIP "accuracy" par
ε = 0.001    # 1-ε is prob that matrix is RIP
C = 5.0      # not sure what this should be
m = ceil( Int, C * δ^(-2) * ( k*log(n/k) + log( ε^(-1) ) ) )

# generate a random noise + sparse signal
seed = 1
rng = StableRNG(seed)
μ = 0.0
σ = 0.1
d = Normal( μ, σ )
x = rand( d, n )

ks = rand(rng, 1:n, k )
x[ks] = 1 .+ rand(rng, Uniform(), k )

# generate random projection matrix
normal = Normal(0.0, 1 / sqrt(m))
A = rand(rng, normal, m, n)
```

```

# projection
y = A*x

# reconstruct
xx = Variable(Float64, n)
lambda = 1e-3*norm(A'*y,Inf)
@time @minimize ls(A*xx - y) + lambda*norm(xx,1)
```

A signal with $n = 10,000$ and $k = 10$, and parameters chosen so that $m = 1,520$, took 17.4 seconds.

The next one took 1538.6 seconds to process a vector with $n = 100,000$ and $m = 13,954$. The first 1000 points are shown below. Seems around $O(n^2)$ at a guess.



Not that impressive. Dimension reduction was only 1 order of magnitude, and reconstruction has lots of extra terms in it.

There are more compelling examples, but you need to get better at the optimisation and the process as well. There are many, many papers on how to do this stuff better.

Never-the-less it is leading to a compelling set of thoughts about how to work with wide data.

Other comments about Julia

Other comments about Julia: There is a “CompressedSensing” package, but it doesn’t seem to work (it’s 7 years old).

The DCT and it inverse seems to exist in Julia in FFTW, but isn’t documented (externall – the internal help exists). I guess there is a fair bit more that isn’t or is. Maybe using the MDCT module: https://juliapackages.com/p/mdct.

What else?

Once you understand all this stuff (at least when you understand it better than me) you start to see all signals as representations of some underying thing.

- ANNs are just doing some type of non-linear representation, often over-complete, internally. Compressive sensing gives you insights into what should work there.
- Data assimilliation should be tracking state in complex embedded spaces, so why not compressed spaces?

- Signals (mathematically) are continuous so a lot of this theory can be performed in a more general setting, with discrete signals occurring as sampled versions.
- Many other topics link up, e.g.., basis pursuit, LASSO, Shrinkage, …
- How do I know how sparse my data will be?

Links

Introductions, tutorials and reviews

- https://cnx.org/contents/9wtroLnw@5.12:lh465hW1@7/Introduction-to-compressive-sensing
- https://www.jstage.jst.go.jp/article/transcom/E96.B/3/E96.B_685/_pdf/-char/en
- https://www.raeng.org.uk/publications/other/candes-presentation-frontiers-of-engineering
- https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8260873
- https://authors.library.caltech.edu/10092/1/CANieeespm08.pdf

Resources links

- https://dsp.rice.edu/cs/

Key papers and people

- David Donoho, Compressed sensing. (IEEE Trans. on Information Theory, 52(4), pp. 1289 - 1306, April 2006)
- Emmanuel Candès and Terence Tao, Near optimal signal recovery from random projections: Universal encoding strategies? (IEEE Trans. on Information Theory, 52(12), pp. 5406 - 5425, December 2006)
- Emmanuel Candès and Justin Romberg, Practical signal recovery from random projections. (Preprint, Jan. 2005)
- Emmanuel Candès, Justin Romberg, and Terence Tao, Stable signal recovery from incomplete and inaccurate measurements. (Communications on Pure and Applied Mathematics, 59(8), pp. 1207-1223, August 2006)
- Jarvis Haupt and Rob Nowak, Signal reconstruction from noisy random projections. (IEEE Trans. on Information Theory, 52(9), pp. 4036-4048, September 2006)
- Richard Baraniuk, Mark Davenport, Ronald DeVore, and Michael Wakin, A simple proof of the restricted isometry property for random matrices. (Constructive Approximation, 28(3), pp. 253-263, December 2008) [Formerly titled "The Johnson-Lindenstrauss lemma meets compressed sensing"]

In Julia:

- https://nextjournal.com/yusri/compressed-sensing-in-julia
- https://github.com/dahlend/CompressedSensing.jl
- https://docs.juliahub.com/MRIReco/h2KbQ/0.2.4/compressedSensing.html
- https://discourse.julialang.org/t/compressed-sensing-using-structuredoptimization-jl/35098