# Lab Notes for Compressed Sensing Coding

Anthony Salerno

02.17.15 - May 4, 2015

# Contents

**1 Proof of Principle (02.17.15 - 02.23.15) 2**
1.1 Introduction . . . 2
1.2 Preamble... . . . 2
1.2.1 Hypothesis . . . 2
1.2.2 Notes - Unpolished . . . 2
1.3 Simulations . . . 4
1.3.1 Preamble . . . 4
1.4 Hypothesis . . . 4

**2 Analysis of CS Code – Sanity Check (03.10.15) 5**
2.1 Preamble... . . . 5
2.2 Down-sampling . . . 5

**3 Simulations - (03.11.15 - 04.01.15) 7**
3.1 Preamble . . . 7
3.2 Hypothesis . . . 7
3.3 Results . . . 7
3.3.1 Numerical Simulations . . . 7
3.3.2 Brain Data . . . 9
3.4 Discussion . . . 10
3.5 Other Notes . . . 10
3.6 Post-Meeting Notes . . . 11

**4 Directional Continuum Sampling 12**
4.1 Preamble . . . 12
4.1.1 Basis . . . 12
4.1.2 Meeting Notes . . . 12
4.1.3 Meeting with Leigh . . . 12

**5 Return to CS with sparseMRI_v0.2 15**
5.1 What happened? . . . 15
5.2 Reanalysis and Understanding! . . . 15
5.2.1 High Level Code . . . 15
5.2.2 Lower Level – `fnlCg.m` . . . 16
5.2.2.1 What's Happening – Superficially . . . 16
5.2.2.1.1 Compute the gradient of the sparse data . . . 16
5.2.2.1.2 Compute the preobjective and objective . . . 19

# 1 Proof of Principle (02.17.15 - 02.23.15)

## 1.1 Introduction

Beginning work on the "Proof of Principle" as per the 12.19.14 Meeting Notes - Plan 2

The notes state that we are to:

- Pick one slice - extract slice from dataset with FT on RO (i.e. in dim space) and no FT on PE1 and PE2 (aka PE and SL)
- Isotropic undersampling (I think this is a bad idea)
- Recon slices independently with Lustig code (using $\lambda_2 psi[m] + \lambda_1 TV[m]$)
- Add a directional similarity term to the recon (so we can recon similar slices simultaneously) $(\lambda_3 ||m_j - m_k||_2 f(\vec{d_j} \cdot \vec{d_k}))$
- However, the form of $f((\vec{d_j} \cdot \vec{d_k}))$ is still unknown

Note that the git repository for this work can be found at `https://github.com/aasalerno/Lustig`. For pre-emptive notes on the code that Lustig wrote, please see `demo_SheppLoganTV_Notes.txt`.

## 1.2 Preamble...

### 1.2.1 Hypothesis

To begin, today we should be able to get a decent code running that can do Lustig's code on any specified slice that we choose. I expect that the code will work as expected and provide a good rendition of the undersampled data with minimal alterations required. The hope is that we can use the basis of the code as an engine (that will need some optimization) in order to work with the data as we so hope.

### 1.2.2 Notes - Unpolished

First spent a ridiculous amount of time getting LaTeXup and running on this computer... But it works now!

The code is written both on my computer as well as on the lab computers – via a central git on git-hub – using the same datasets as required. The datasets in use are Jacob's data, reconstructed purely using the standard recon algorithm (that is, no MATLAB involved).

For all pushes and pulls, see the git-hub repository. There are many (and it can be mapped by day!)

The code is to be written such that it is built and any values that are above 1-sampFac will be included. This means that when we are actually tacking on the $r$ correction, we need to do it as $1 - \frac{r}{\max(r)}$

Seem to have a problem with the values that I'm getting. The outer portions of k-space seem to have almost no chance of being chosen. With a penalty threshold of only 0.25 (given a sampFac of 0.5), only 40% of values are chosen. Most of the lost points are on the corners of $k$-space, so this may be ok... However, I need to look into **possibly making the CS type sequence specific, for things like cylindrical acquisition**.

Date: 02/20/15
As of about 11:00 (commit 6a9cfb111c508df0af9a7222ba277ef118ba147), the code `testMap.m` is up and running. It will be used in order to actually build the map of what we are undersampling and then this will be pumped into an adapted version of Lustig's code in order to get it up and running.

Encountered a bit of an issue with how the data is obtained. Since this information is only the magnitude of the data, I'm going to make a version that can handle taking the real and imaginary parts of the data, then feeding that through to the next set of functions.

As of 12:08 PM, functionality has been added for the function to handle two files containing the real and imaginary parts of the $k$-space information!

Now, we want to add in the CS part of the code. Here is where things become a little painful, but we can look at the datasets and actually do what is done in CS.

For some unapparent reason, the code doesn't seem to want to work. This is really annoying and irritating as the error seems to be stemming from the raw files, which may render a tonne of previous work that I've done utterly useless. Ok, scratch that – the issue was my method of plotting. I wasn't plotting the abs of the data.

As of about 5:00 it is working. I'm going to git push it and go home.

Date: 02/23/15

Upon a first look at the code that I wrote last week (adapted from `demo_SheppLoganTV.m`, the code doesn't seem to have too great of an effect. The differences between the "im_dc" (density corrected original) and the CS reconstructed data is on the order of $10^{-9}$, when the data is on the order of 1.

- One of the first things that I should try is to change the TV and L1 penalties as they are currently set to 0.01 each. This may be too low to have an effect on real data – the code that I am adapting from is using a numerical phantom.
- The next step would be to try different slices or different data. Perhaps the noisiness of this data is causing a problem, but can be fixed with some more phase corrections.

I changed the values of the two penalties ($\lambda_1$ and $\lambda_2$), increasing them by a factor of 10, each becoming 0.1. The overall residual became a little bit larger, but is still on the order of $10^{-9}$, which is negligible.

In order to make the data make a bit more sense (i.e. make sure I'm not making a stupid mistake on how I'm building the filter, etc.) I'm going to use Lustig's method of building the undersampling pattern.

I changed the code `testMap.m` pretty substantially, so see git-hub for a previous version. The alterations were made to try and use the other undersampling pattern. One thing that I may do later is alter Lustig's PDF that is produced by tacking on the directionality afterwards.

After changing to Lustig's method of undersampling, we see some bigger differences, but, the undersampling kind of makes it look pretty bad. It may be how the FT (using this '`p2DFT`') instead of just doing an FFT as it is done in MATLAB . This seems like something weird that they don't need to do, as it just makes the data look bad... Though there must be a reason for this

As it stands – 4:00PM 02/23/15 – this is as far as I can go without further understanding how well everything is going to work for the perpendicular and parallel to the gradient direction for the undersampling technique.

## 1.3 Simulations

### 1.3.1 Preamble

Here what we want to ensure is that by applying specific filters, we are obtaining what we expect. The expectation is that using a mask that is parallel to the gradient direction (applied in $k$-space) is going to look significantly worse than a mask applied perpendicular to the gradient direction.

## 1.4 Hypothesis

Using a simulation, we will be able to tell distinct differences between the use of the two different maps. The map using a perpendicular mask on the undersampled $k$-space will look better than the data undersampled using the parallel mask, as the smearing will not be as bad in the cross direction of the diffusion.

The code is written in `buildLambda.m`. It is currently built just to handle a 2D case and then *eventually* handled to build a 3D. In order to make it work with the code, an idea is to make it have a $z$-stack (in our case, a $y$ stack, or a 'read-out" stack).

# 2 Analysis of CS Code – Sanity Check (03.10.15)

Date: 03/10/15

## 2.1 Preamble...

As of this point, I've finally given my seminar (which apparently went well!) and now, I'm trying to see if there's something inherently wrong with how I'm running my CS codes, as they aren't producing the effects that I would expect. The noise looks *really* weird, and isn't how I'd expect it to be.

So, for today, we are going to change things up a bit, and move on to try and understand exactly what's happening in this code – I will try to include figures where I believe it makes sense due to the simplicity of doing so.

In order to simplify things, I made a file using the raw data from Jacob's Diffusion Weighted Imaging data taken on July 29, 2014. The raw data files are located in `/projects/muisjes/asalerno/CS/data`. Specifically we are using brain 10, direction 32. This brain was chosen to be the simplest of cases, as this is a $b_0$ scan, thus there is no expected major differences because of the lack of diffusion weighting.

## 2.2 Down-sampling

One *major* thing to note is the effect that down-sampling has. We can see it here:

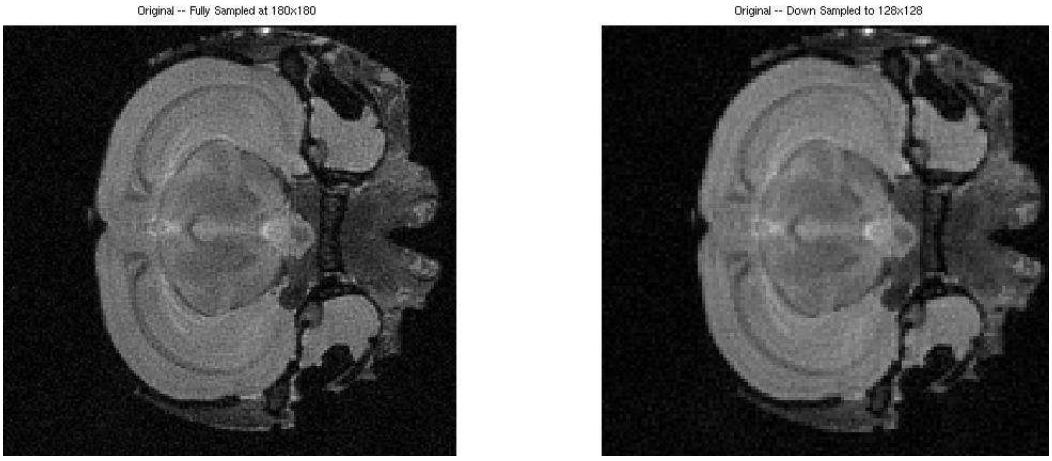


Figure 1: We can see the significant blurring effect that downsampling the figure to be 128x128 has. The blurring cannot be fixed, as in order to do this, we must sacrifice some higher spatial frequency information. Note that this is using MATLAB blurring. This is explained further in the next section and figure 2.

The downsampling is absolutely required because of how the wavelet transform is done. The 2D Wavelet transform has a $2^n$ size requirement in order to work (Lustig 2007, Yang 2015). We don't particularly care about the number of slices being $2^n$ as this will be fully sampled anyway.

It seems like MATLAB does something a little weird with how it downsamples the FFT... It produces what looks like just a blurred copy of the image (as seen above) but it doesn't seem to get the data from the centre, but instbuildLambdaead chooses the data in the top right section (before shifting) preferentially. This can be seen in Figure 2.

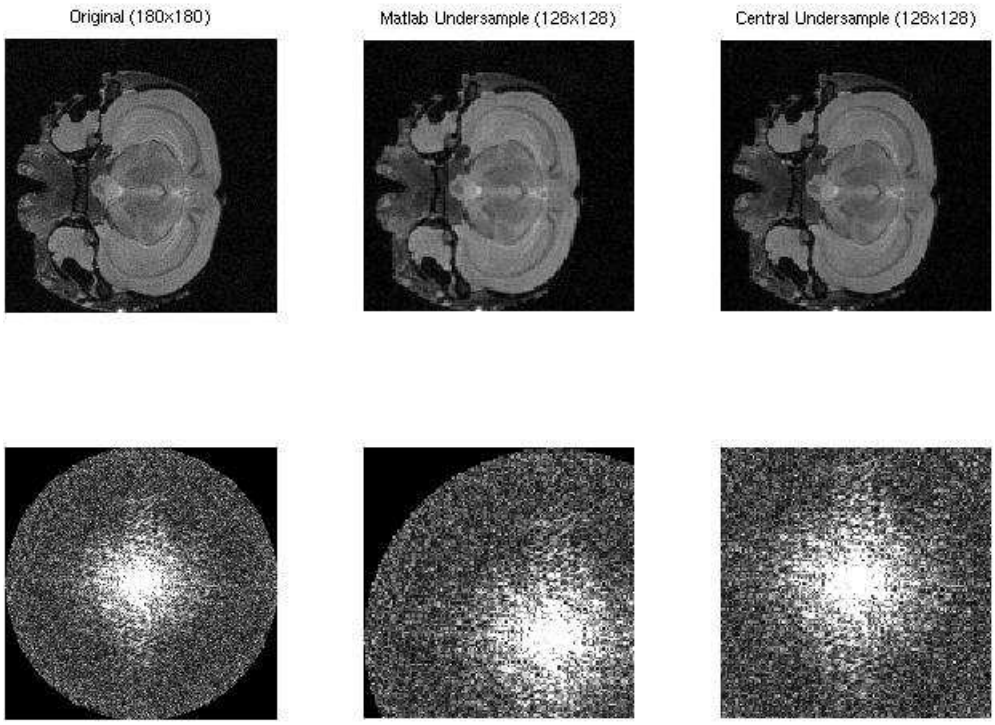


Figure 2: In this figure, we see some small differences between the method of undersampling done in MATLAB vs a logical undersampling method that takes the data from the centre of $k$-space. All data here taken from slice 150 *after* doing a 1D FFT in the readout direction.

We can see in the above figure that MATLAB does something really weird and so, we will be using my method in order to do reconstructions that alter the number of voxels per dimension.

In order to make sure that we're using the right one, the file we must use is `kspaceDS.10.32.mat`, as this is the dataset that has been undersampled properly. For this work, what we will do right now, is look at how we can try to do a reconstruction properly using compressed sensing. We may be able to do this logically using some of the existing work here. In the angiography case, they use the previous slice as the base image as "not much is expected to change". We can do better than this, as we can try to do a combined reconstruction, where the $b_0$ average of the same slice (to start) would be the same.

# 3 Simulations - (03.11.15 - 04.01.15)

03.23.15 (work completed on and before 03.19.15)

## 3.1 Preamble

The point of running this simulation is to see what the best undersampling method to use would be. Our plan of attack is to do this experiment two fold, firstly with purely numerical data – i.e. a "phantom" that I design, and then afterwards using true brain data and seeing if we can get an understanding of what the best method would be.

Some of this information is noted at an earlier point in time,

## 3.2 Hypothesis

The expectation is that the best style would be to use the "parallel to the gradient direction" as this will give us the sharpest edge information in order to tell a cross section of the fibres that we are expecting to see. The worst should be the "perpendicular" case. It is noted that our nomenclature from before was misleading. In order to ensure that we have it correctly, the following figure explains what is meant by each.

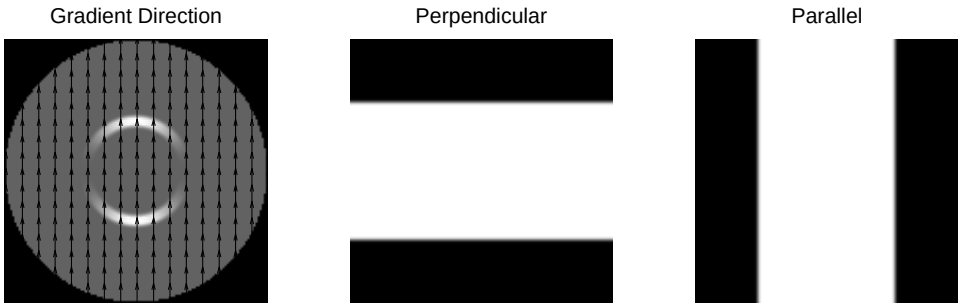


Figure 3: We can see the direction of the gradient in the left-most panel, and the definitions of the others. The parallel will give us the "cross-fibre" information mostly, and the perpendicular will give us the "along the fibre" information.

## 3.3 Results

### 3.3.1 Numerical Simulations

For the numerical simulations, we started with a phantom that was comprised of three parts. The phantom can be seen in the following image.

When we worked with this data, we used multiple different types of undersampling filters, and compared them to the fully sampled case. The ones that we used are:

- Circle
- Square
- Parallel Strip
- Perpendicular Strip
- Parallel Ellipse

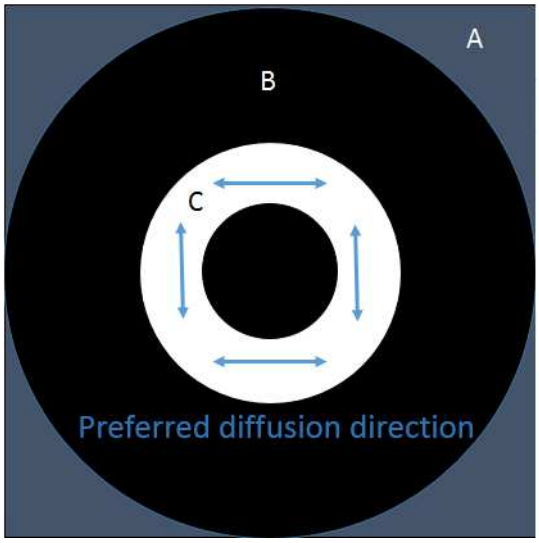


Figure 4: This figure is the numerical phantom that we used. Region (A) represents an area of all zero data, used to serve as a reference point when plotting the figure (it forces the dynamic range to begin at zero). Region (B) represents an area of isotropic diffusion, where $\lambda_1 = \lambda_2 = \lambda_3 = 10^{-3}$. This area should have an FA value of zero. Region (C) represents an area with restricted diffusion, where $\lambda_1 = 10^{-3}$, but $\lambda_2 = \lambda_3 = 5 \times 10^{-4}$. This gives a theoretical FA value of approx 0.41. It is noted that the $b$ value used in this experiment was 1917 $\frac{\text{s}}{\text{mm}^2}$

- Perpendicular Ellipse

For the two ellipses, it is noted that the long axis to short axis ratio was set to be 2 (i.e. a/b = 2 if $\left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right)^2 = 1$). It is also noted that $x$ and $y$ here are those of the axis rotated relative the gradient vector direction.

Upon comparing all of our datasets, we found that the square and the circle gave the smallest RMS error. This can be seen via a bar chart – this chart combined all the ROIs from all directions (doing the fully sampled - undersampled), and and ran an RMS on all of the data.

For these plots, we used a ROI that just covered the circle – quite tightly – so that we didn't have much of a confounding factor. We still saw, though, that the "circle in the centre" (as seen in the next plot) was the best method to reduce the error that we get in our data.

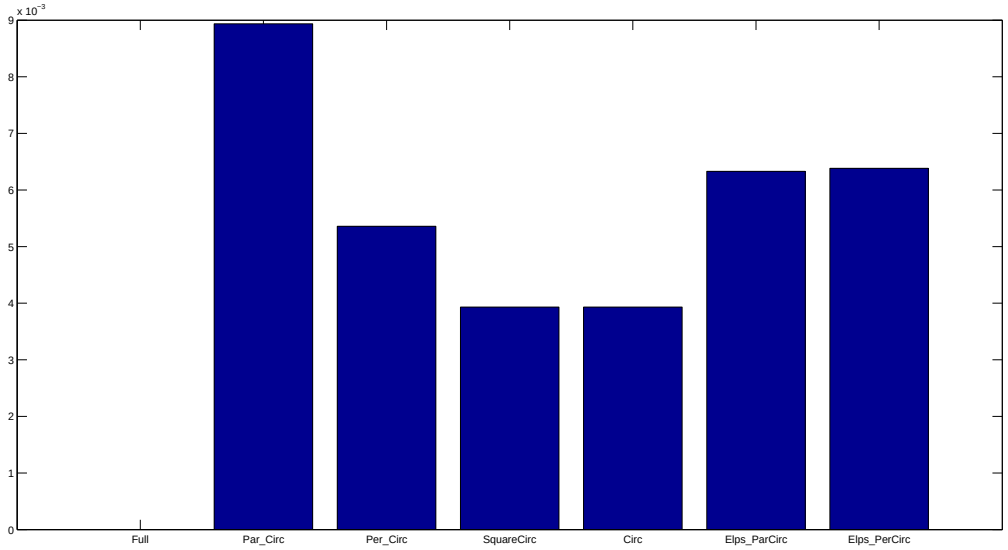


Figure 5: This figure shows us how the RMS differs for the different undersampling types. We can see the undersampling types in the next figure.

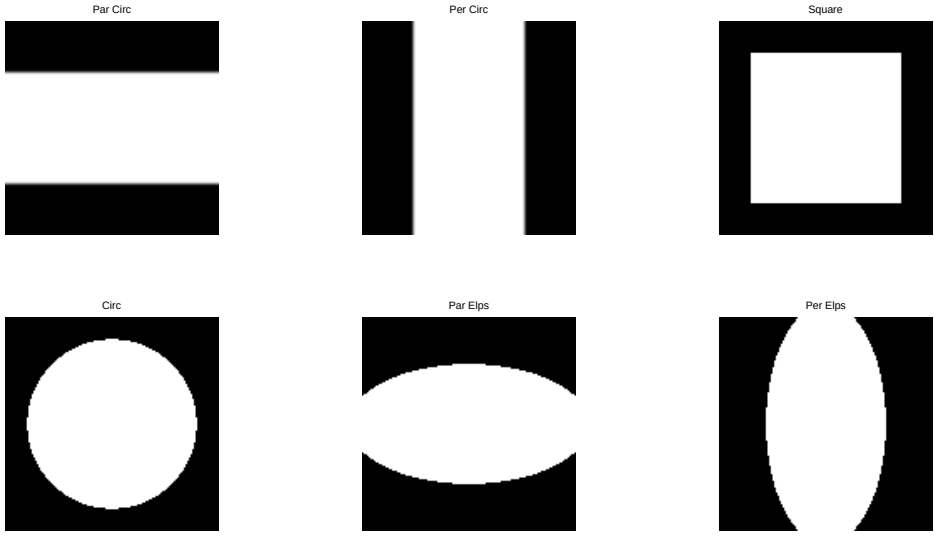


Figure 6: The different filters that we used for the plots. It is noted that the codes used almost no roll off. For this example, we are assuming the gradient direction to be totally vertical.

### 3.3.2 Brain Data

When proceeding to brain data, I expect that we would be the same results. Using data that can be found in `/projects/muisjes/asalerno/CS/data`, and undersampling methods that can be found in `/micehome/asalerno/Documents/CompressedSensing`, we ran the same undersampling techniques, but got some interesting results. The ellipse method is now *worse* than the strip method! Reasons for this are currently unknown. The dataset being shown is brain 2, all 30

directions, however, this can (and will) be redone for more brains in order to get a more realistic understanding of what's happening.

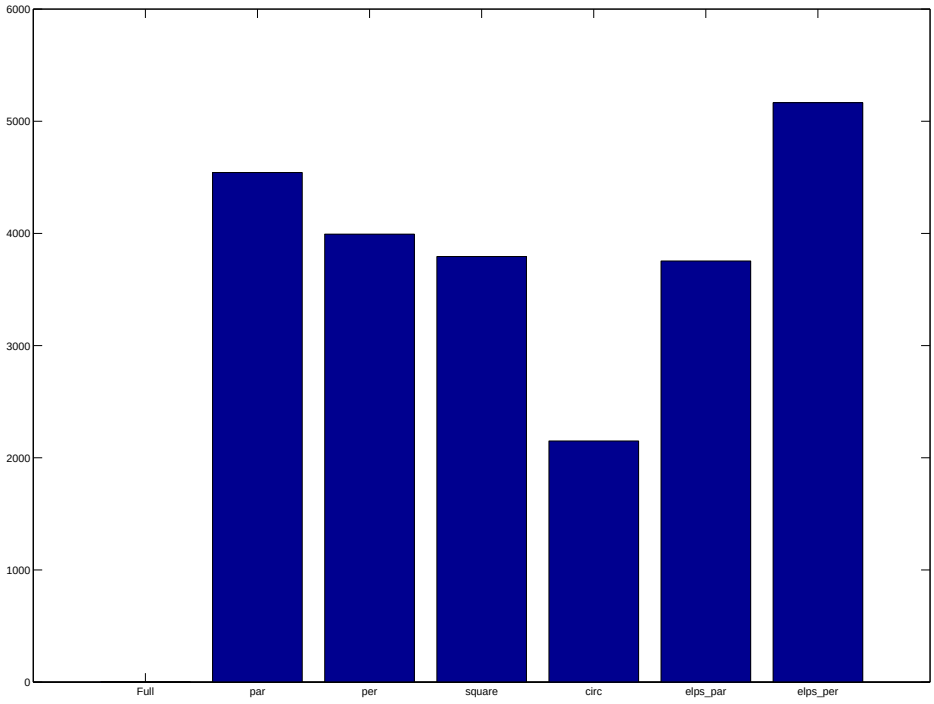


Figure 7: RMS error from fully sampled data in comparison to the fully sampled case.

### 3.4 Discussion

03.25.15 - 04.01.15 Surprisingly, we didn't particularly get what we expected. When running the simulation for the numerical phantom, the data didn't seem very different at all – at least when comparing FA values. The data actually gave us results that were confounding... When being compared to the "standard" methods of undersampling (i.e. Circular or low resolution undersampling), the parallel and perpendicular cases fell through and were not effective. However, even when looking for a "happy medium" – an elliptical technique, which we expect would have the lower RMS error of the circular technique, but the specificity of the directionally specific techniques – we don't find that (with $n = 4$ for brains tested).

### 3.5 Other Notes

Due to space constraints, I've had to delete a lot of the (extraneous) data from the folders. I removed all of Jacob's recons (as they're unimportant, and stored elsewhere). I also deleted (inadvertently) the FA map that I produced for it, but all in all, it wasn't that important anyway. Since it is so easy to reproduce from the original information (i.e. the fully sampled case) I'm also getting rid of the undersampled real and imaginary data.

### 3.6 Post-Meeting Notes

*Meeting Date: 04.01.15*
Brian stated that I should do some work on looking at the line profiles of everything that has been done because RMS is akin to looking primarily at DC information. This should give us a better rendition of what we want because we're interested in the edges more than the DC information.

# 4 Directional Continuum Sampling

04.14.15 – Never completed

## 4.1 Preamble

### 4.1.1 Basis

The idea behind what this method of acquisition is that we won't necessarily need CS for a reconstruction. The notion is to use a continuum of directions, as opposed to a specific number of discrete directions – while only taking a small number of points for each "direction" we would be able to do a full reconstruction using a solid angle of points (and possibly, be able to shrink the required number of points by utilizing CS).

### 4.1.2 Meeting Notes

The notes are about building the simulation, as well as some work in relation to how to build the simulation. As of 04.14.15, I don't have a firm grasp of what I want to do and/or how I plan to do it. However, I believe that looking at PE tables (and including that in my solutions) will be a good method.

- If CS is to be used, incorporate the directional information before we look at 3D recon. A possible form could be:

$$e^{-\frac{\alpha_{jk}^2}{2\sigma^2}}$$

  Where $\alpha_{jk} = \angle \vec{d_j} \vec{d_k}$

- Do a weighted average to obtain the proper eigenvalues for the simulation

### 4.1.3 Meeting with Leigh

After meeting with Leigh SN to discuss methods of how to best do something that would make sense with respect to the FSE acquisition method. She pointed me to an email from Brian and the locations of some tables.

**Email from Brian**

```
Hello Jacob et al,

You can try the following command for generation of cylindrical FSE tables.


/home/bjnieman/source/vnmr/cyltable_gen.py --nf 64 --cylindrical --etl 8 --etlorder 2,1,0,3,4,5


Obviously, you can replace ‘‘nf 64", ‘‘etl 8", ‘‘etlorder 2,1,0,3,4,5,6,7" and ‘‘256
256" (as well as the output filename) with numbers of your choice.  nf should be divisible
by etl.
```

To view the tables and inspect for problems, you can use:

/home/bjnieman/source/vnmr/petable_display.py test_nf64_etl8_256_256_nf64_ni64

Options -nacq and -etl <number> allow you to display the number of acquisitions (repeated in the table) or the echo number of each acquisition.

Come to me if you have questions.

Brian

**Email from Leigh**

Here is the location of several different petables at 75 um (272 x 272 in the phase encode, $xz$ plane):

/home/leigh/cyltables/75um/angorder/LSN_E3_angorder_272_etl8_nf64_ni80

/home/leigh/cyltables/75um/cylaxisdist/LSN_E3_cylaxisdist_272_etl8_nf64_ni80

/home/leigh/cyltables/75um/pseudocyl/LSN_E3_pseudocyl_etl8_272_nf64_ni80

If you need to look at all of the smaller tables (12 each) have a look in the parent folder, but know that there are a LOT of other tables in those folders, so just be careful to read the names carefully (they are different in the E3 E4 parts, where E3 means the third echo in the centre and E4 means the fourth).

To make these tables I used the following commands:

/home/bjnieman/source/vnmr/cyltable_gen.py --nf 64 --etl 8 --cylindrical --angorder --individ_

/home/bjnieman/source/vnmr/cyltable_gen.py --nf 64 --etl 8 --cylindrical --cylaxisd

NOTE, PSEUDOCYL IS A LITTLE DIFFERENT

/home/bjnieman/source/vnmr/cyltable_gen.py --nf 64 --etl 8 --cylindrical --pseudocyl_pe2 --indi

Also, note that nf and the table axis (272) must be divisible by etl. These tables

```
all have a resolution of 75 um.

Let me know if you need anything else.

Leigh
```

These locations as listed above will be used in order to find the information about the PE tables so I make sure that what I do is compatible with the scanner – thus I'm not testing something that can't be done.

# 5 Return to CS with sparseMRI_v0.2

## 5.1 What happened?

When I was working with the code, at some point in time, I did something that fatally affected the code, causing problems and iterative work that would lead to results that were terrible – equivalent to doing nothing at all. The CS reconstruction looked just like the zero-filled reconstruction. To figure this out, I re downloaded the codes and reran them, as I was having problems with the phantom reconstruction as well (which should work absolutely perfectly!). The code was re downloaded and placed in `/micehome/asalerno/Documents/Lustig`.

## 5.2 Reanalysis and Understanding!

Here, I will discuss each code and exactly what's happening. The basis of the code will be from the `demo_SheppLoganTV.m`, just like the previous time that I did this (which I will be looking at as a reference as well!), though this time, I will *not* edit the code as much so that I don't break it again!

### 5.2.1 High Level Code

Note that the folder `Orthogonal` needed to be added from Lustig's other code – specifically the code that goes through Wavelets and their uses etc.

**p2DFT**:

I don't quite get what the point of this is, though it seems like it's some sort of 2D Discrete Fourier Transform, in matrix form. This plays off of Brian's idea of using a matrix of matrices on the diagonal to apply this type of transform to the data.

**XFM**:
This is the all important sparsifying transform. As a default it is set to be 1 for the phantom. For testing purposes, I will change this to be a wavelet. When this is run, using a Daubechies wavelet, at $6^{\text{th}}$ order. As defined by Lustig, this means:

```
The Daubechies filters are minimal phase filters that generate wavelets which
have a minimal support for a given number of vanishing moments.  They are
indexed by their length, Par, which may be one of 4, 6, 8, 10, 12, 14, 16,
18 or 20.  The number of vanishing moments is par/2.
```

Using the Wavelet transform, from a standard Shepp-Logan phantom, we get decent results!

**Eight Iteration `for` loop**

Weirdly enough, this looks better as the iterations go on... It isn't running the *exact* same thing each time, but using the previous data (seemingly) for each iteration. So, in theory, having a max of 64 iterations would be the same as having 8 iterations in the `for` loop with each loop having a max iteration limit of 8. This will be tested at some point – go to it here.

**Questions**

1. What exactly does the p2DFT do? Why use this instead of an FFT?
2. What effects would different XFMs have (even using the same type, but with different numbers)?
3. How do the two parameters that we have (soon to be made 3) affect the image?

#### 5.2.2 Lower Level – `fnlCg.m`

The prime importance of this part of the code is to understand `fnlCG.m` and the objective functions, because this is the basis of how everything is done in this code. By understanding this, I'll be able to build on it (or possibly even port it!).

Originally, this code had all of the codes in one script, however, I moved all extra functions into their own scripts, located in the folder `./grads`.

For this analysis, I will be switching between mathmode and verbatim. To make it easier, here are the switches:

| Represents | Code | Math Mode |
|---|---|---|
| **Raw Data** ($k$-space) | `params.data` | $y$ |
| **Iterative Data (sparse)** | `x` | $x$ or $m$ |
| **Fourier Transform** | `params.FT` | $\mathcal{F}_u[\cdot]$ |
| **Sparsifying Transform** | `params.XFM` | $\phi[\cdot]$ |
| **Inverse Fourier Transform** | `params.FT'` | $\mathcal{F}_u^{-1}[\cdot]$ |
| **Inverse Sparsifying Transform** | `params.XFM'` | $\phi^{-1}[\cdot]$ |
| **Gradient of the data** | `dx` | $\nabla m$ |

Table 1: Table of conversions between math mode and code.

##### 5.2.2.1 What's Happening – Superficially

The first thing that is done is looking at the line search parameters. These are parameters chosen for the minimization technique that's employed here. Nothing of major importance here since it is all set previously in the `params` variable.

###### 5.2.2.1.1 Compute the gradient of the sparse data

The equation that we're editing is: $g_0 = \nabla\phi(x)$ This calculates the gradient of the *sparse* dataset. It uses `wGradient.m`, which relies on calculation of the objective functions (which still needs to be discussed).

**`gOBJ.m` - Gradient of the Objective Function**

This code is just one line:

```
gradObj = 2*(params.XFM*(params.FT'*(params.FT*(params.XFM'*x) - params.data)));.
```

It is important to note that when the transform matrices are multiplied, it applies the transform, but multiplying by the transpose applies the inverse transform.

The important players here are:

- `XFM` - The sparsifying transform
- `FT` - The Fourier Transform
- `x` - Our data that is undergoing iterations
- `data` - The original set of data

This line does the following:

1. Returns the data to image space
2. Puts the data in $k$-space
3. Subtracts the original dataset from it in $k$-space
4. Returns the data to image space
5. Puts image into sparse space
6. Multiplies it by 2

There's some intrinsic penalty right off the bat because of how the calculations are done. Firstly, with sharp edges, the FT's are not the exact same. In addition, the sparsifying transform likely has an effect.

When looking at the phantom data, it seems that the major differences between `x` and `params.data` (not surprisingly) come from the edges, but the differences are decently sized ($\sim$0.5, compared to an image max of $\sim$1.)

This code seems to do this part of the Compressed Sensing equations:

$$||\mathcal{F}_u m - y||_2 < \epsilon$$

Since we are taking the `abs` of this when plotting (and looking at it's value in comparison to $\epsilon$) this is effectively taking the $\ell_2$ norm of the data.

### `gXFM.m` - Gradient of the $\ell_1$ Transform Operator

This function is also a one liner:

```
grad = p*x.*(x.*conj(x)+params.l1Smooth).^(p/2-1);
```

Here, `p` represents the pNorm value for the $\ell_1$ term. It is a *scalar* value!
In mathematical notation (to be simpler to read) we have:

$$\nabla f = px(|x|^2 + \alpha_1)^{\frac{p}{2}-1}$$

In this case, again, $x$ represents the sparsified data (i.e. it is in a sparse space) that continuously undergoes iterations. It's noted that $|x|^2 = xx^*$. $\alpha_1$ here represents an $\ell_1$ smoothing term. In the example here, it is set to be $10^{-15}$, and $p$ is set to be 1.

This equation doesn't seem to come up in the paper. If we analytically look at what this gives us, we can easily see what effect we should get. We will use the parameters as stated in the previous paragraph.

$$\begin{aligned}\nabla f &= px(|x|^2+\alpha_1)^{\frac{p}{2}-1} \qquad\qquad |x|^2+\alpha_1 \approx |x|^2\\ &= x(|x|^2)^{\frac{1}{2}-1}\\ &= x(|x|^2)^{-\frac{1}{2}}\\ &= x(|x|^{-1})\\ &= 1\end{aligned}$$

When we run the code, the difference between the range of values are $1 \pm 10^{-9}$. So we can see that the analysis makes sense. I will note, though, that I *don't* get what this part does exactly. Brian and I *NEED* to talk about this!

### `gTV.m` - Gradient of the TV operator

As the title says, this handles the TV operator. This, however, is more than one line, as the TV is applied to the data in *image space*, but $x$ is iterated in the *sparse space*.

Something of interest is that when the TV operator is applied, the size of the matrix doubles, and increases a dimension! $x$ went from being [256 x 256] to [256 x 256 x 2]. There seems to be some orthogonality to it, as seen in Figure 8.

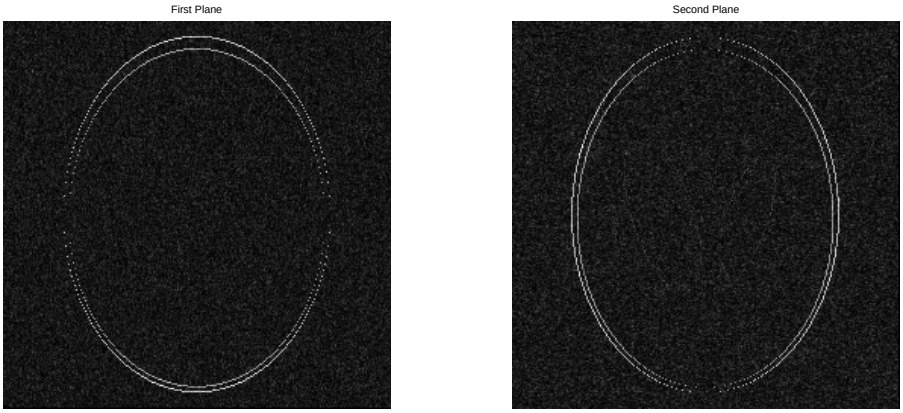


Figure 8: Comparison of the two [256 x 256] planes given in after the TV operator is applied

Overall, however, this equation has a similar form to the one before it, but requires a few transforms in order to be put back into the sparse space.
Let $D_x = TV[\phi^{-1}[x]]$:

$$\nabla f = \phi[TV^{-1}[pD_x(|D_x|^2+\alpha_1)^{\frac{p}{2}-1}]]$$

The terms inside of $\phi[TV^{-1}[\cdots]]$ are analogous to the terms in the above section. Again, we utilized $|D_x|^2 = D_x D_x^*$. In this case, however, it is noted that the data doesn't follow the same "1" value as the `gXFM` did, even though the terms inside the transforms are all 1. My guess is that the noise of this distribution (which will happen in normal DAQ) will cause this value to increase

dramatically (which is logical because variation will occur due to noise from voxel to voxel – if it didn't, the assumption of Gaussian or Rician noise would be incorrect).

#### 5.2.2.1.2 Compute the preobjective and objective

The preobjective (and objective) functions are calculated early in order to "make it cheap to compute many times for efficient line-search". Some of these values are calculated many times and often, so this would be a good place to attack when we are looking at optimization.

**`preobjective.m` - Precalculates Transforms to Make the Line Search Cheap**

This code is quite simple, as it just applies the transforms to both $x$ and $\mathrm{d}x$ and places them in variables. The good news is that it makes them readily accessible, but the bad news is that in large datasets, this can become quite memory consuming.

The nomenclature is long, annoying, tedious, but all in all, understandable. The key is to look for each set of transforms. From this code, the outputs that we get are:

- `FTXFMtx` - $k$-space of $x$ ($\mathcal{F}_u[\phi^{-1}[x]]$)
- `FTXFMtdx` - $k$-space of $\mathrm{d}x$ ($\mathcal{F}_u[\phi^{-1}[\mathrm{d}x]]$)
- `DXFMtx` - TV of the image space of $x$ ($TV[\phi^{-1}[x]]$)
- `DXFMtdx` - TV of the image space of $\mathrm{d}x$ ($TV\phi^{-1}[\mathrm{d}x]]$)

**`objective.m` - Calculates the Objective Function**

The first thing that caught me in this code is the parameter $t$. It's just set to 1 in this code, but this adds *yet another parameter* for us to play with.
In this code, we calculate the objective as:

```
obj = FTXFMtx + t*FTXFMtdx - params.data;
```

In mathematical terms, we have:

$$Obj = \sum_{i=1}^{N} \left(\mathcal{F}_u[\phi^{-1}[x]]_i + t\mathcal{F}_u[\phi^{-1}[\mathrm{d}x]]_i - y_i\right)^2$$

Where $y$ is the original data that we started with (off the scanner). We let $i$ represent the $i^{\text{th}}$ voxel of the data, since the objective value that we obtain is to be a measure of our loss. To put it into perspective, we are figuring out what the $\ell_2^2$ in $k$-space between the data, plus it's gradient, from the original data off the scanner. For all conversions, remember to see Table 1.

This then happens again for the XFM and TV differences, but in a slightly different manner, which will be explained below.

For those two, the code is as follows:

```
w = x(:) + t*dx(:);
XFM = (w.*conj(w)+params.l1Smooth).^(p/2);
```

The only difference that exists between the two is what the `w` is calculated from. In `TV` it uses `DXFMt(d)x`, and in `XFM` it uses `(d)x`. This makes sense because the total variation uses the TV of image space (which is what `DXFMtx` is), and the `XFM` is calculated via the sparsified data, contained in `x`.

The code was written in such a way that the TV and XFM weights don't need to be constant. The overall value given from the objective function is calculated as:

```
res = obj + (TV) + (XFM) ;
```

Where the `TV` and `XFM` were multiplied by their respective weights. The RMS value is also calculated here, but it isn't used in `fnlCg.m`.

This objective function is calculated once with $t = 0$, as a reference, and then again with $t$ as a parameter – the reference is used to break a loop. With $t = 0$, we lose all effects of the gradient, and so this is the differences that we would have with just the transforms (i.e. the problems I was talking about earlier because of the sharp edges and cutoffs, etc.).

### The importance of $\alpha$ and $\beta$

```
f1 > f0 - alpha*t*abs(g0(:)'*dx(:))
t = t*beta
```

`f1` is the value from the objective function when $t \neq 0$ `f0` is the value from the objective function when $t = 0$.

$\alpha$ and $\beta$ are very important in this analysis, as they give different weights when attempting to get out of the line search.

$\alpha$ tells us how strongly we will weight the effects of the gradient to take off of the original dataset if we wish to get comparable objective functions between the original dataset and the iterated dataset.

$\beta$ tells us how much are we going to weight the gradient when doing our calculations for the objective function. As we weight it less and less, we *should* be getting closer and closer to the case where $t = 0$, giving us smaller values.

Once we get through the while loop via $\alpha$ and $\beta$ we update our data ($x$) and our gradients ($g_0$) and go through the calculations again with new starting criteria unless the $\ell_2$ of the gradient is less than $\epsilon$, or we've reached our iteration limit.