Report: Fast Sparse Fourier Transformations for NMR Spectroscopy

By
Badruddin Kamal, U5429737

Supervised by
Thomas Huber, RSC
Alastair Rendall, RSCS

Prepared for COMP8715 @ ANU
Semester 1, 2015.
List of abbreviations

- NMR- Nuclear Magnetic Resonance
- Sparse- Thinly dispersed or scattered
- IST- Iterative Soft Thresholding
- FFT- Fast Fourier transformations
- IFT- Inverse Fourier transformations
- FFTW- FFT and IFT package for Linux
- MPI- Message Passing Interface
- GUI- Graphical User Interface
Table of Contents

Abstract .............................................4
Introduction ....................................4
Background ....................................5
Approach .......................................6
Software .......................................10
Results .........................................14
Discussion .....................................36
Conclusion .....................................38
Citations .......................................39
Abstract

NMR spectroscopy is used to analyze bonds in molecular structures. It can help in understanding synthesized molecular structures. However, this process does take time if uniformly sampled and time wise expensive. To reduce the time requirements sparse sampling in NMR has become quite popular. There are different algorithms based on Maximum Entropy to analyze data, however in our case we will use Fast Fourier Transformation based approach to reconstruct spectra from sparse data. The method we will follow is called Iterative Soft Thresholding (IST). It essentially Fourier transforms and inverse Fourier transforms sparse data, between Frequency domain and Time domain taking the most likely occurring data points and ignoring noise. For this a simple C program that can use Python GUI is made. Also, the project focused on optimizations for one core and possibility of expanding into multiple cores. From analysis of data sparsity performance, we determine the best inputs in constructing the data fast and efficiently. Also we look at the scopes and uses for this project in different fields.

Introduction

Sparse sampling has become main stream in high-dimensional NMR spectroscopy, as it allows to significant speed-up of data acquisition, experiments that normally would take several days can be completed with sparse sampling in several hours, ultimately making these high-dimensional experiments possible. Processing of sparsely sampled NMR requires execution of a Fourier transformation, which is difficult to perform when the data is distributed sparsely and unevenly. This project will explore different implementations of sparse Fast Fourier transformation algorithms for processing NMR data. Performance of these algorithms will be profiled, and code optimized for execution speed. The algorithm followed for the project is known as Iterative Soft Thresholding. That is basically taking the most occurring intensities in sparse data and summing them up to get original data.
Background

Nuclear magnetic resonance (NMR) is the disruptions to magnetic field generated by a molecule when exposed to a uniform magnetic waves. These disruptions generate waves in the time and frequency domains. Though they produce disruptions in other multiple domains, we will only consider the changes in the time and frequency domain.

By analyzing this NMR data we are able to understand different carbon molecular structures. For example, to understand the structure of a T2 viral protein (Hyberts SG, Milbradt AG, Wagner AB, Arthanari H, Wagner G., 2012), we can apply NMR to understand the C-bonds types and Carbon bond structures. Understanding this data is crucial, as it allows us to understand how different Carbon structures effect on other molecules. Which could be essential in fields such as medicine, pharmaceuticals and even in engineering fields to design and produce uniquely structured molecules. This includes applications medicine and vaccines for certain disease’s to stronger materials for structures to new light receptive solar cells based on algae, nano filters etc. (K. D. Moudgil M.D., D. N. Rao Ph.D., B. S. Narang M.D., 1985)

To detect these changes we, use a spectrometer to uniformly sample data in the time domain, however doing this takes time and requires several hours to process and sample. By sampling this data sparsely (thinly dispersed sampled at alternate times, rather than sampling uniformly), that is just taking data at random points we can apply different methods to reconstruct this data. It has already been shown, that by taking random points in 2D, 3D and 4D NMR data can yield really good performance for reconstruction than other methods of sparse sampling. The popular methods for reconstructing this sparse data include Maximum Entropy based methods to FM and CLEAN procedure etc. Most of these methods have a high degree of complexity when it comes to calculating sparse NMR data. Hence, end up requiring 1.5 days or so to calculate on a computer. Oher approaches include iterative weighted approach etc. of them the gradient optimization approaches are the most expensive computationally. (Hyberts SG, Milbradt AG, Wagner AB, Arthanari H, Wagner G., 2012)
However, by applying Fast Fourier based reconstruction techniques such as Iterative Soft Thresholding (IST) etc. the process time can be decreased to several hours. This is due to the fact that, these procedures take into account effect on other dimensions and calculate the highest possible occurrences. By doing so we are able reconstruct the original spectra within 500 iterations making them ideal for fast execution on modern high performance computers with multiple cores etc. (Hyberts SG, Milbradt AG, Wagner AB, Arthanari H, Wagner G., 2012; Xiaobo Qu, Di Guo, Xue Cao, Shuhui Cai, Zhong Chen, 2011) By using optimized performance methods such as loop unrolling, SSE and MPI (later options enabled in FFTW configurations), we can see that the process time can go down. Which can be used to yield reconstructed data faster, by decreasing processing time and utilizing resources optimally.

IST isn’t a new method, it’s been around for a while now. It has been used in MRI, signal processing and image processing to reconstruct sparse data. Though fairly new to this field IST has gained quite a bit of popularity in the field with its fast processing time and performance where it can read up to a 80% sparse file in a short span of time. (Hyberts SG, Milbradt AG, Wagner AB, Arthanari H, Wagner G., 2012)

Approach

To understand how IST works, one has to first understand how Fast Fourier transformations (FFT) and Inverse Fourier transformations (IFT) affect NMR spectral data Fig1. Illustrates a transformation between the time and the frequency domain. Basically by applying IFT on Frequency domain NMR data we convert it to time domain data, and respectively applying FFT to time domain data we get frequency domain data.
Using a spectrometer we gather, the data in the time domain. However, to do so we have to uniformly sample datapoints at time intervals, which can be quite tedious. This can take up to several hours to construct the full time domain data. As such we use sparse NMR sampling to get data at random points and generate the frequency from there. (Hyberts SG, Milbradt AG, Wagner AB, Arthanari H, Wagner G., 2012; Xiaobo Qu, Di Guo, Xue Cao, Shuhui Cai, Zhong Chen, 2011)

After random sampling, we only get parts of the data and set all other points to 0. This is a crucial step as it tells us the relevant and non-relevant data points for our reconstruction. It helps use remove noise from sparsity of the generated NMR data.
Fig. 2 Iterative soft threshold on NMR data.

Now to run IST on this data, we iteratively convert the sparse NMR time data into frequency data and cut out sections of the data above the given threshold and store them to a separate space. After that we apply IFT on the data below threshold and get time domain information. Then we set the sparse points to zero and run this data through iterations again. We keep doing this till we are done with all iterations. Figure 2 illustrates this method. On our first iteration we convert the spectra using FFT to frequency domain. Then we take out sections given the threshold. For example if the threshold is 0.7 we set a threshold margin of 70% of the highest frequency and store all the data above this threshold in a separate space, and keep the remainder of the data convert it to time domain by doing an IFT and set all original sparse points to zero. On the second iteration we do the same run a FFT and cut off sections at the 70% maximum of highest peak. Then we add the top sections with the previous top sections in the separate space and build the frequency. We convert the remainder of the data by IFT to time dimension and set sparse points to zero. We keep doing this till our iteration count is zero. Notice that after a certain number of iterations the added sections to the separate space become very small or almost close to zero. That is where our method converges and our separate construction contains all the original frequency data. (Hyberts SG,
Milbradt AG, Wagner AB, Arthanari H, Wagner G., 2012). In our experiments we consider sigma in frequency domain to be constant for 2D NMR data and we only work on the dimension of intensity.

The IST algorithm is as given

\[ K_\rho(u) := |Tu - g|^2 + \sum_{\lambda \in \Lambda} \rho_\lambda \|u_\lambda\|_q, \]

by means of minimizers of \( J(k) \). \( K_\rho \) is closely related to the soft thresholding operator \( S(\rho) \) in and its minimizer can be approximated by the algorithm with \( \omega \lambda = 0. \) (Massimo Fornasier, Holger Rauhut, 2007)

Implementation algorithm

do:
    read file store in array
    store sparse points index in a separate array

while iteration count not 0 do :
    calculate FFT on data in array
    do:
        find max peak
    do:
        set threshold
        remove sections above threshold
        store them in a separate array
    convert remainder using IFT
do:
  set sparse points to zero
  increment iteration count
  
do:
  store data to file

A few aspects we have to consider while running this algorithm is that the sparsity of the file does affect performance. For example if a file is 100% sparse we cannot reconstruct the original frequency. It’s best to have data less sparse than 90%. However with a very sparse file it takes quite some time ie iterations to reconstruct the data and there is an increase in noise with more sparse data.(Hyberts SG, Milbradt AG, Wagner AB, Arthanari H, Wagner G., 2012)

Also with higher thresholds it’s crucial to keep an eye on performance as it does affect it. Example with a higher threshold it requires more iterations to converge than lower ones. Though the data is more refined, it takes more time to calculate. Essentially meaning that the process time increases. Hence it’s highly desirable to set smaller thresholds. But setting too small of thresholds can lead to presence of noise.(Hyberts SG, Milbradt AG, Wagner AB, Arthanari H, Wagner G., 2012)

**Software**

For the application development plan we follow a spiral development plan where meetings with the supervisor were organized as to project milestones.
The language chosen for development was C and packages used were in FFTW which is available widely. (*FFTW3, n.d.*). FFTW is already optimized for processors and can do calculations efficiently. The program was mostly tested on a Virtual Machine Ubuntu running on a CORE i7 4th generation processor as well as 1 cpu on the NCI computer.

A sample Gaussian, simulating a 2D NMR data was taken in the frequency domain on a 128 X 128 grid to simulate the NMR data. For the purposes of sanitizing performance the sigma value is ignored and kept constant, only the intensity values are considered. To generate a sparse file a program sparse was created which essentially takes this data and generates a sparse file (in csv format named sample_NMR.txt) in the time domain. The file randomly selects points and generates a sparse file according to the percentage requested. The application takes the original NMR file and essentially generates the time domain data. After which it counts against the parameter and removes data points randomly to generate a sparse NMR file, which is named sample_sparse_NMR.txt. In this case if the original file is 128 X 128 it generates sparse output for 256 X 256 essentially to make use of zero filling, we will discuss this further a bit later.

There were different development paths chosen to suit different requirements. However, all of these have a variation of the program ist.c. Which essentially reads a file containing sparse NMR data and generates the original data in istout.txt based on the IST algorithm.

To do our calculations on a single processor we use a 1D array FFTW transformations which are faster (this is because we consider in the 2D NMR data sigma constant and intensity I to be the sparse data) Data is stored in the array in the format of x,y coordinates as x*(total number of y)+y. This essentially gives us an array with search complexity of O(n log n). FFTW has a performance complexity of O(n log n). The IST implementation iterates through this at iterations and does a forward and backward Fourier transformation as well as an array iteration and 1 directional array read essentially meaning the complexity is O(n+3n log n) in the worst case. To use this program it requires FFTW to be installed for compilation and crucially optimized for performance. In this version
of FFTW we configured SSE optimizations already for better code performance on x86 machines as well as MPI extensions. (Alfredo Correa, n.d.)

Another aspect which is implemented is loop unrolling, ideally optimizing code for array read and writes according to cache read strides. This shows a significant reduction of processing time by one 1/3 in all cases. Figure 3 & 4 shows applied loop unrolling in sections of code as well as loop reduction. For Unrolling loops we essentially use increments of 4 to be calculated at once. This allows four data heads to be read at once and improve memory strides on the x86 machine.

```c
for ( i = 0; i < nx; i++ )
{
    for ( j = 0; j < ny; j++ )
    {
        if(i+j<ny)
        {            
            out[i+ny]= out[i*ny]+ ( nx * ny*2 );
            out[i+ny+(j+1)]= out[i*ny+(j+1)] / ( nx * ny*2 );
            out[i+ny+(j+2)]= out[i*ny+(j+2)] / ( nx * ny*2 );
            out[i+ny+(j+3)]= out[i*ny+(j+3)] / ( nx * ny*2 );
            if(out[i+ny]>max){
                max=out[i+ny];
            }
        }
        if(out[i+ny+(j+1)]>max){
            max=out[i+ny+(j+1)];
        }
        if(out[i+ny+(j+2)]>max){
            max=out[i+ny+(j+2)];
        }
        if(out[i+ny+(j+3)]>max){
            max=out[i+ny+(j+3)];
        }
        j=j+1;
    }
}
```

Fig 3: Unrolled Loop
This version is used in the graphical user interface (GUI) version. The version is implemented using swig to pass parameters to ist.c and it uses tkinter on python as well as matplot lib to generate a GUI and plot produced graphs. This version is stored on GitHub as a stable release and can be run on any Linux desktop with Python and FFTW installed. It takes input and output files in CSV format as well as iteration numbers and dimensions and threshold. Figure 5 illustrates the application.
Another property we use is known as zero filling. We essentially use a double sized array to do our calculations on. For example, a 128 X 128 spectral data is taken as 256 X 256 and run IST on. However at the end only the 128 X 128 section of the file is processed and the rest is ignored. This method essentially helps tackle issues of residue in calculations and eliminate noise. The other versions developed were based on FFTW 2D complex transformation. However, due to the presence of noise was abandoned. As there were a lot of errors regarding performance.

An MPI (Message Passing Interface) version was also developed. This version was based on the MPI implementation of FFTW which is experimental. Stanford has a great resource on this. (Alfredo Correa, n.d.) However as it does 2D transformations and has also a presence of noise in the output it was also ignored. Another drawback was that NCI installation was of FFTW is not configured to run MPI version. But a version was developed to run on a virtual machine using 2 cores only. It did show significant performance gains and would be crucial aspect to look into the future to improve performance. This version works only for even number of processors to distribute calculation amongst processors. We do consider sigma in these calculations.

Results

To compare performance we compared the IST output from sparse file with original file. First off we compared the 2D FFTW complex reconstruction and MPI reconstruction with original distribution. However we found too many errors and presence of noise and decided to abandon the approach, as it would be inconclusive. Figure 6 illustrates erroneous values.

Afterwards work was started on a 1D real 2 real FFTWS with zero filling approach. It yielded promising results as a 90% sparse file could be reconstructed with very little time on a single processor. With higher thresholds and many more iterations causing time to be a drawback by several seconds it was still quite reasonable. The contours reconstructed on this one match perfectly with the
original and there was little presence of noise. Figure 7 illustrate the overlap with original data.

Figure 6: 2D and MPI errors with original

Figure 7 a: 1D reconstruction overlapping with original data. (Original Blue & Reconstruction Red)
Figure 7 b: 1D reconstruction contours overlapping with original data.(Original Blue & Reconstruction Red)

Figure 7 c: 1D reconstruction contours overlapping with original data.(Original Blue & Reconstruction Red)
To compare performance on a 1D IST algorithm we calculate the Root Mean Square (RMS) by comparing the generated data points with the original to find errors in the generated data. We use the following method to calculate this.

\[
\text{RMS} = \sqrt{\frac{\text{Original data} - \text{Reconstructed data}}{\text{total data points}}}
\]

This gives us an indication of convergence. We compare this to the number of iterations and file sparsity to understand convergence at different levels of sparsity and thresholds.

In Figure 8 for a full dataset we consider the threshold values of 0.2, 0.4, 0.6 and 0.8. We ignore threshold values of 1.0 as this would not take any sections out, as it would take the entire maximum peak as threshold. Also, we ignore threshold values of 0.0 as that would take the entire peaks including noise and IST would
thus fail. In this experiment we see that with lower threshold (< 0.6) the IST algorithm converges to 0 RMS in less than 20 iterations as it takes big enough sections to process fast. However, with 0.8 threshold it converges at 40 iterations. This is true because the smaller sections require more iterations to converge.

Fig 9 : 20% sparse file 1D IST implementation RMS VS Iterations.

In Figure 9 for a 20% sparse file we consider the threshold values of 0.2, 0.4, 0.6 and 0.8. We ignore threshold values of 1.0 as this would not take any sections out, as it would take the entire maximum peak as threshold. Also we ignore threshold values of 0.0 as that would take the entire peaks including noise and IST would thus fail. As discussed previously. In this experiment we see that with lower threshold (< 0.6) the IST algorithm converges to 0 RMS in less than 25 iterations as it takes big enough sections to process fast. However with 0.8 threshold it converges at 50 iterations. This is true because the smaller sections require more iterations to converge. We observe an overall increase in iterations, this is due to the fact that IST reconstruts frequencies, that were initially set to zero as a means of sparse sampling.
In Figure 10 for a 40% sparse file we consider the threshold values of 0.2, 0.4, 0.6 and 0.8. We ignore threshold values of 1.0 as this would not take any sections out, as it would take the entire maximum peak as threshold. Also we ignore threshold values of 0.0 as that would take the entire peaks including noise and IST would thus fail. As discussed previously. In this experiment we see that with low threshold (< 0.4) the IST algorithm converges to 0 RMS in less than 20 iterations as it takes big enough sections to process fast. At threshold 0.6 it converges at 30 iterations. However, with 0.8 threshold it converges at 65 iterations. This is true because the smaller sections require more iterations to converge. We observe an overall increase in iterations this is due to the fact that IST reconstruts frequencies that were initially set to zero as a means of sparse sampling.
In Figure 11 for a 60% sparse file we consider the threshold values of 0.2, 0.4, 0.6 and 0.8. We ignore threshold values of 1.0 as this would not take any sections out, as it would take the entire maximum peak as threshold. Also we ignore threshold values of 0.0 as that would take the entire peaks including noise and IST would thus fail. As discussed previously. In this experiment we see that with threshold < 0.4 the IST algorithm converges to 0 RMS in less than 25 iterations, as it takes big enough sections to process fast. At threshold 0.6 it converges at 45 iterations. However with 0.8 threshold it converges at 65 iterations. This is true because the smaller sections require more iterations to converge. We observe an overall increase in iterations this is due to the fact that IST reconstructs frequencies that were initially set to zero as a means of sparse sampling.
In Figure 12 for a 80% sparse file we consider the threshold values of 0.2, 0.4, 0.6 and 0.8. We ignore threshold values of 1.0 as this would not take any sections out, as it would take the entire maximum peak as threshold. Also we ignore threshold values of 0.0 as that would take the entire peaks including noise and IST would thus fail. As discussed previously. In this experiment we see that with threshold 0.4 and 0.6 the IST algorithm converges to 0 RMS in less than 60 iterations as it takes big enough sections to process fast. At threshold 0.2 it converges at 50 iterations, but because its taking larger sections it ends up taking the noise in IST reconstruction thus the RMS value is higher. However with 0.8 threshold it converges at 200 iterations. This is true because the smaller sections require more iterations to converge. We observe an overall increase in iterations this is due to the fact that IST reconstruts frequencies that were initially set to zero as a means of sparse sampling.
In Figure 13 for a 90% sparse file we consider the threshold values of 0.2, 0.4, 0.6 and 0.8. We ignore threshold values of 1.0 as this would not take any sections out, as it would take the entire maximum peak as threshold. Also we ignore threshold values of 0.0 as that would take the entire peaks including noise and IST would thus fail. As discussed previously. In this experiment we see that with threshold 0.6 the IST algorithm converges to 0 RMS in less than 180 iterations, as it takes big enough sections to process fast. At threshold 0.2 and 0.4 it converges at 80 and 100 iterations, but because its taking larger sections it does take the noise in IST reconstruction thus the RMS value is higher, this is now also apparent for threshold 0.4. However with 0.8 threshold it converges at 350 iterations. This is true because the smaller sections require more iterations to converge. We observe an overall increase in iterations that is due to the fact that IST reconstructs frequencies that were initially set to zero as a means of sparse sampling.
From the previous analysis we can see that the best threshold value regardless of sparsity would be around 0.6. Also, we observe that with files more than 60% sparse thresholds lower than 0.6 cause errors in values due to the existence of noise in IST sectioning. We also observe that 0.8 or higher thresholds require more iterations thus more calculation time, which makes them undesirable. However in the presence of data noise higher thresholds do help out clean up time but significantly increases process time. (Hyberts SG, Milbradt AG, Wagner AB, Arthanari H, Wagner G., 2012)

Now we consider sparsity vs RMS at set iterations to determine best convergence levels.

![Graph showing RMS vs file sparsity for different thresholds](image)

**Fig 14 :** 10 iterations on 1D IST implementation RMS VS file sparsity

In figure 14 we see that at 10 iterations RMS vs sparsity performance at different thresholds. Like before we ignore threshold values of 1.0 as this would not take any sections out, as it would take the entire maximum peak as threshold. Also we ignore threshold values of 0.0 as that would take the entire peaks including noise and IST would thus fail. We observe a general increase in RMS with the increase
of sparsity for thresholds greater than equal to 0.6. Which is predictable as the number of iterations are too low for any of them to converge. However we do observe that for low sparsity (10-20%) lower thresholds do converge.

Fig 15 : 50 iterations on 1D IST implementation RMS VS file sparsity

In figure 15 we see that at 50 iterations RMS vs sparsity performance at different thresholds. Like before we ignore threshold values of 1.0 as this would not take any sections out, as it would take the entire maximum peak as threshold. Also we ignore threshold values of 0.0 as that would take the entire peaks including noise and IST would thus fail. We observe a general increase in RMS with the increase of sparsity. For thresholds greater than 0.6 their rms gives errors for file more than 30% sparse. Which is predictable as the number of iterations are too low for any of them to converge. However we do observe that for lower sparsity lower thresholds do converge for 70% sparse files after which they give errors as sections are too big.
In figure 16 we see that at 100 iterations RMS vs sparsity performance at different thresholds. Like before we ignore threshold values of 1.0 as this would not take any sections out, as it would take the entire maximum peak as threshold. Also we ignore threshold values of 0.0 as that would take the entire peaks including noise and IST would thus fail. We observe a general increase in RMS with the increase of sparsity. For thresholds greater than 0.6 their rms gives errors for data more than 60% sparse. Which is predictable as the number of iterations are too low for any of them to converge. However we do observe that for lower sparsity lower thresholds do converge for 80% saprse files after which they give errors as sections are too big. Thershold 0.2 gives errors after the file is 80% sparse, which is caused by taking too large sections and presence of more intense noise.
In figure 17 we see that at 150 iterations RMS vs sparsity performance at different thresholds. Like before we ignore threshold values of 1.0 as this would not take any sections out, as it would take the entire maximum peak as threshold. Also we ignore threshold values of 0.0 as that would take the entire peaks including noise and IST would thus fail. We observe a general increase in RMS with the increase of sparsity. For thresholds greater than 0.6 their rms gives errors for data more than 70% sparse. Which is predictable as the number of iterations are too low for any of them to converge. However we do observe that for lower sparsity lower thresholds do converge for 70% sparse files after which they give errors as sections are too big. But for a threshold of 0.6 it does converge at 80% meaning it’s close the right size for optimal rms values.
In figure 18 we see that at 250 iterations RMS vs sparsity performance at different thresholds. Like before we ignore threshold values of 1.0 as this would not take any sections out, as it would take the entire maximum peak as threshold. Also we ignore threshold values of 0.0 as that would take the entire peaks including noise and IST would thus fail. We observe a general increase in RMS with the increase of sparsity. For thresholds greater than equal 0.6 their rms gives errors for data more than 80% sparse. As they have enough iterations to converge close to the right size for optimal rms values. However, we do observe that for lower sparsity lower thresholds do converge for 60% saprse files after which they give errors as the thershold sections are too big.
In figure 19 we see that at 500 iterations RMS vs sparsity performance at different thresholds. Like before we ignore threshold values of 1.0 as this would not take any sections out, as it would take the entire maximum peak as threshold. Also we ignore threshold values of 0.0 as that would take the entire peaks including noise and IST would thus fail. We observe a general increase in RMS with the increase of sparsity. For thresholds greater than equal to 0.6 their rms gives less errors for data more than 90% sparse. Which is due to the number of iterations are enough for them to converge. However we do observe that for lower sparsity lower thresholds do converge for 60% - 70% sparse files after which they give errors as sections are too big.
In figure 20 we see that at 1000 iterations RMS vs sparsity performance at different thresholds. Like before we ignore threshold values of 1.0 as this would not take any sections out, as it would take the entire maximum peak as threshold. Also we ignore threshold values of 0.0 as that would take the entire peaks including noise and IST would thus fail. We observe a general increase in RMS with the increase of sparsity. For thresholds greater than equal to 0.6 their rms gives less errors for data more than 90% sparse. Which is due to the number of iterations are enough for them to converge. However we do observe that for lower sparsity lower thresholds do converge for 60% - 70% saprse files after which they give errors as sections are too big.

From observing this data we can conclude that the best amount of iterations to always converge data would be around 250 or above for therhsold values greater
than equal to 0.6. We can also see that for lower iterations and lower thresholds on a lower sparse dataset can yield good results, but with the increase of sparsity it does underperform.

Another aspect we look into is the process time given number of iterations. We compare the performance changes at different process time and iterations, against threshold values and data sparsity. This would give us a clear image of the time needed for different sparsity.

Figure 21 illustrates the total time required to iterate a full dataset. We observe around 400 - 700 iterations threshold values equal to 0.6 or above require slightly less time than the ones below. However, all the curves are linear and end processing always around the same time.
Figure 2 illustrates the total time required to iterate a 10% sparse dataset. However, all the curves are linear and end processing always around the same time.
Figure 23 illustrates the total time required to iterate a 30% sparse dataset. However, all the curves are linear and end processing always around the same time.
Figure 24 illustrates the total time required to iterate a 50% sparse dataset. However, all the curves are linear and end processing always around the same time.
Fig 25: 70% sparse dataset 1D IST implementation Iterations VS process time.

Figure 25 illustrates the total time required to iterate a 70% sparse dataset. However, all the curves are linear and end processing always around the same time.
Figure 26 illustrates the total time required to iterate a 90% sparse dataset. However, all the curves are linear and end processing always around the same time.

From the above observations we can conclude that with the increase of file sparsity there is no change to process time and the file is processed in linear time. Thus there is no effect of sparsity on process time. The reason for this is that regardless of data sparsity, the algorithm always iterates through all the points hence we get this output.
Discussion

The goal of this project has been to implement an efficient version of IST on sparse NMR data. Though other methods are available IST has a relatively better performance than other algorithms. The reason partly is due to the higher performance of FFT based algorithms. Sparse NMR reconstruction has become really important now, in the field of NMR spectroscopy as it allows us to calculate data in several hours that used to take several days. It is an important step for the field, to help it construct data and understand experiments faster. NMR is important to understand protein synthesis and structures. This allows scope for helping pharmaceuticals to develop and analyze protein based medication and vaccines to be analyzed and developed faster. (Iwona Wawer, U. Holzgrabe, & B. Diehl, n.d.) It can also be used in the development of other molecules for specialized purposes such as nano filter items.

IST is also widely used for MRI image processing and signal processing. It could also be used to store and reconstruct data such as voice and images. Where one would store compressed data and expand it again from a small section. From a small section of the size of 90% data you could essentially reconstruct the original spectra.

The implemented version works for 2D NMR data, sigma is considered constant and the sampling intensity is sparse. So, we essentially run a 1D Fourier transformation on the intensity in Frequency and Time domain. This usually generates relatively clean data with RMS close to zero. After analyzing the performance we were able to reconstruct a wave of 128 X 128 data points that was 90% sparse in about 0.5 seconds. However, this requires a higher threshold around above 0.6 is recommended, though other implementations have shown 0.98 would be best (especially in the case for removing noise and contamination), in this application it converges at 150 iterations on a threshold value of 0.6 for 90% sparse data. (Hyberts SG, Milbradt AG, Wagner AB, Arthanari H, Wagner G., 2012) This means the implemented algorithm requires less iterations for the case of highly sparse files.
From analyzing other aspects, we can see that process time is proportional to iterations and sparsity has no effect on the process time. The time required to process data is dependent on the number of iterations. Also, we observe that for files greater than 60% sparse a threshold value of 0.6 or around is optimal at for 150 iterations or around. With a smaller threshold it takes large data sections that lead to errors in RMS value as it reads noise, though they converge around 50 iterations. For larger threshold values they also converge nicely, but require higher number of iterations. Which means it has a higher process time.

For files less than 60% sparse we see that at low threshold (<0.6) they converge at 20-40 iterations. As they take sufficiently big sections to converge fast and do not have many noise artifacts from sparse data. But with a high threshold such as 0.8 it requires a few hundred iterations almost to converge as it keeps taking small sections. Thus the data is more refined.

We never take threshold values of 1.0 as then we will not be taking any sections in IST, as threshold will be equal to highest peak. We also avoid cases to threshold value of 0.0 as that case would only take all the erroneous data with noise. As the threshold limit would then be set to 0, which is also not desirable.

The original code had a lot of loops and with loop optimization we were able decrease process time down from 2.84 to 2.01 on a sample set, which is almost a 1/3 of the original process time. In general for a 128 X 128 grid taking 2N the space of 256 X 256 data points, for zero scaling it takes about 4.5 seconds to process on a normal desktop. This is crucial as zero scaling helps us remove residue from Fourier transformations easily, thus reducing noise at each transformation.

For future work one thing that is highly recommended to be looked at is the MPI implementation with FFTW. With MPI currently, we see that process time goes down from 2 seconds to 100ms as the work load is distributed amongst processors. However, as the current version doesn’t analyze data properly and there is a presence of noise, it is an aspect that could be looked into to improve process
time. This shows promise for faster processing on many cores essentially promising for processing high dimensional 3D and 4D NMR with many data points on them in a few hours.

Conclusion

NMR spectroscopy is best used for analyzing molecules specially protein structures. It is used to develop different medications for disease’s, as well as analyze and develop new molecules for specialized use. But the time required to do so is expensive by taking sparse data and reconstructing it on the computer it can essentially save up time on reconstructing spectral data, allowing for more time for scientists do to other experiments. Hence forth we developed an application that takes 2D sparse NMR data and using the IST algorithm generates original spectra. The developed application can run on most platforms and the algorithm can easily efficiently calculate the original spectra. This algorithm has already been used in a lot of fields and can be used in other fields such as data compression to save space. The application is optimized for a single core system, but can be extended to multicores to allow fast processing of 3D and 4D data. This project was successful for implementing a 2D sparse NMR spectra reconstruction and has future scopes for more dimensions and processors.
Citations


