CHAPTER 6

IMPLEMENTATION

The implementation phase of software development is concerned with translating design specification into source code. The primary goal of software implementation is production of source code that is easy to read and understand. Source code clarity is enhanced by structural coding techniques, by good coding style, by appropriate supporting documents, by good internal comments etc.

Program codes are written following the structured coding technique, which linearizes the control flow, so that the execution sequence follows the sequence in which codes are written. This enhances the readability of code, which eases understanding, debugging, testing, documentation, and modification of the programs.

6.1 Signal Generation

The first step in this project is the generation of an analog signal. The user is asked to enter a key between 0 and 9.Each input is associated with two frequencies that are obtained from Table 6.1.The signal generated comprises of two sinusoids. The signal 'f' is generated by using formula (1).

INPUT	Frequency 1(f1) Hz	Frequency 2(f2) Hz
0	941	1336
1	697	1209
2	697	1336
3	697	1477
4	770	1209
5	770	1336
6	770	1477
7	852	1209
8	852	1336
9	852	1477

Table 6.1 Component Frequencies

$$f= (sin (2*pi*f1*t) +sin (2*pi*f2*t))/2 \text{.....(1)}$$

where f1 and f2 are the two component frequencies.

6.2 Sampling

The next step is to sample the given analog signal. The sampling frequency (Fs) is fixed at 40000Hz and the signal is sampled every $1/8^{th}$ of the time period, thus resulting in 5000 samples. The compression ratio is fixed at 90%. Thus, 500 samples are taken from the 5000. These samples are obtained by applying a matrix known as the sampling matrix (ϕ) to the signal of interest(c). This signal of interest contains all the major coefficients of the given signal, i.e., the non-sparse components. This is a $n \times 1$ vector where n=5000. Vector 'c' is obtained by applying the basis matrix (ψ) to the vector 'f'. Here, the basis function has been chosen to be the Discrete Cosine Transform (DCT). DCT was chosen as the basis function because it requires lesser terms to represent a typical signal in comparison with other basis functions such as Fast Fourier Transform, Fourier Transform etc. The main function of the Discrete Cosine Transform is to transform the given signal which is in the time domain to the frequency domain.

The choice of the sampling matrix is key to the success of the compressed sensing paradigm. It is a $m\ X\ n$ measurement matrix where $\ll n$, provided the signal is sparse or nearly-sparse in the original or some transform domain. Lower values for 'm' are allowed for sensing matrices that are more incoherent within the domain (original or transform) in which signal is sparse. This explains why compressed sensing is more concerned with sensing matrices based on random functions as opposed to Dirac delta functions under conventional sensing. Although, Dirac impulses are maximally incoherent with sinusoids in all dimensions, however data of interest might not be sparse in sinusoids and a sparse basis (original or transform) incoherent with Dirac impulses might not exist.

On the other hand, random measurements can be used for signals s-sparse in any basis as long as ϕ obeys the following condition:

$$\boldsymbol{m = s.log\ \left(\frac{n}{s}\right)} \text{.....(2)}$$

As per available literature, ϕ can be a Gaussian, Bernoulli, Fourier or incoherent measurement matrix . Equations (3) and (4) quantify 'm' with respect to incoherence between sensing matrix and sparse basis. Another important consideration for robust compressive sampling is that measurement matrix well preserves the important information pieces in signal of interest. This is typically ensured by checking Restricted Isometry Property (RIP) of reconstruction matrix A (product of measurement matrix with representation basis) . RIP is defined on isometry constant δ_S of a matrix, which is the smallest number such that:

$$(\mathbf{1}-\boldsymbol{\delta_S})||\boldsymbol{x}||_{l_2}^2 \leq ||\boldsymbol{Ax}||_{l_2}^2 \leq (\mathbf{1}+\boldsymbol{\delta_S})||\boldsymbol{x}||_{l_2}^2 \text{ .....(3)}$$

holds for all s-sparse vectors 'x'. We will loosely say that a matrix obeys the RIP of order s if δ_S is not too close to one. RIP insures that all subsets of s columns taken from matrix are nearly orthogonal and sparse signal is not in null space of matrix being used to sense it (as otherwise it cannot be reconstructed). Similarly, if δ_{2S} is sufficiently less than one, then all pair-wise distances between s-sparse signals must be well preserved in the measurement space, as shown by:

$$(\mathbf{1}-\boldsymbol{\delta_{2S}})||\boldsymbol{x_1}-\boldsymbol{x_2}||_{l_2}^2 \leq ||\mathbf{A}(\boldsymbol{x_1}-\boldsymbol{x_2})||_{l_2}^2 \leq (\mathbf{1}+\boldsymbol{\delta_{2S}})||\boldsymbol{x_1}-\boldsymbol{x_2}||_{l_2}^2 \text{ .....(4)}$$

for s-Sparse vectors *x*1 and *x*2.

Given a matrix, it is not possible to verify whether it obeys RIP or not, however, it has been found that when a random matrix is chosen it is highly probable that the matrix obeys RIP. In this project, A matrix of zeroes with random elements made one is employed as the sampling matrix. When the sampling matrix is applied to vector 'c', we obtain the required samples.

The entire sampling process described so far can also be understood with the help of the following code snippet:

```
Fs = 40000; %sampling frequency
t = (1:Fs/8)'/Fs; %sampling rate
f = (sin(2*pi*a*t) + sin(2*pi*b*t))/2;
n = length(f); %n=5000
m = ceil(n/10); %m=500
```

```
k = randperm(n)';
k = sort(k(1:m));
b = f(k); %random samples
```

Figure 6.1 gives a diagrammatic representation of the compressed sensing method:

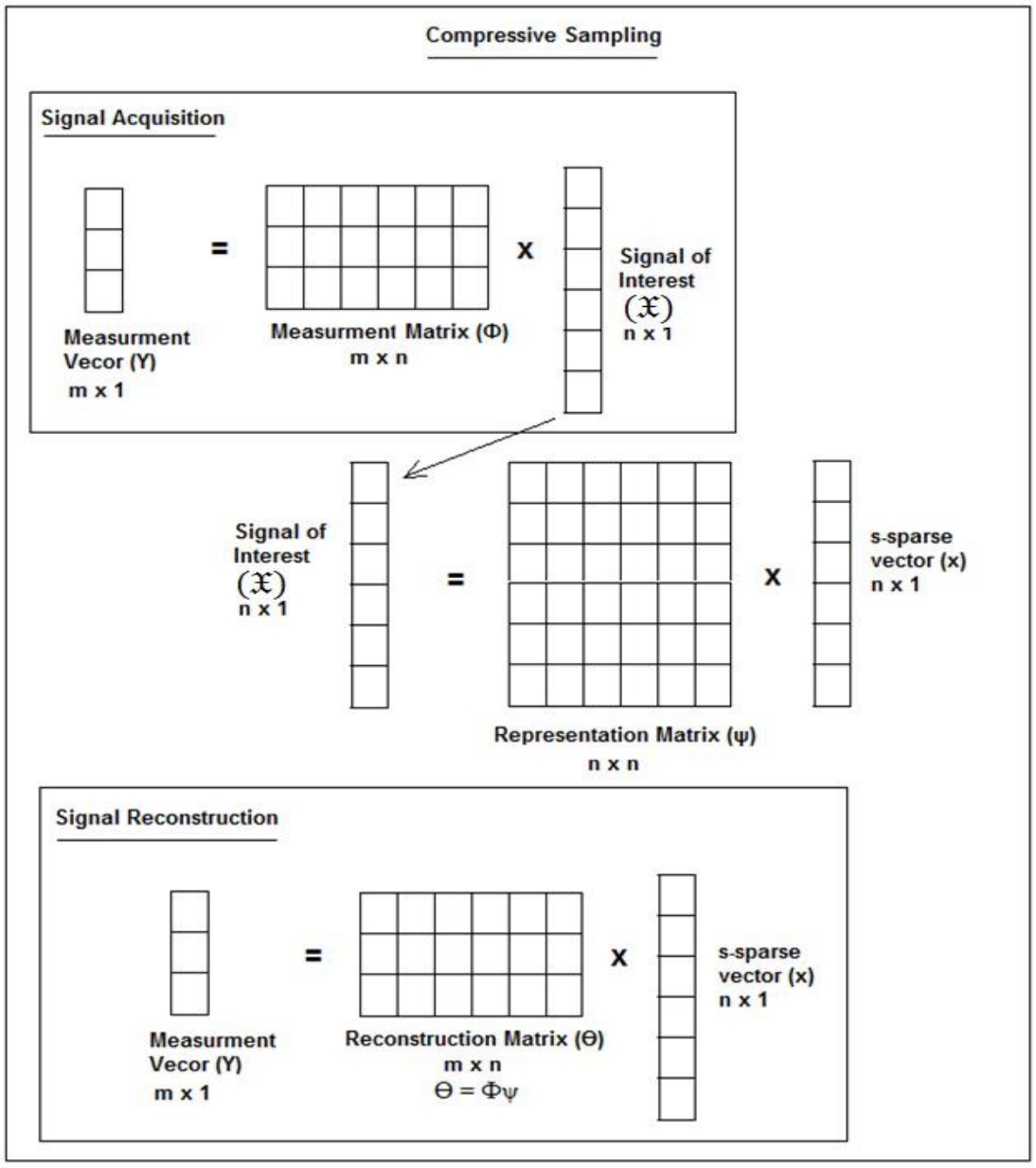

Figure 6.1 Compressed Sensing method

6.3 Transferring the samples

Once the samples are generated, they are to be used for reconstruction by the reconstruction module. This process takes place in the following steps:

- Write samples and other parameters to a file
- Establish a connection with the reconstruction module
- Send the file to the reconstruction module
- Retrieve the samples and parameters from the received file

6.3.1 Write samples and other parameters to a file

The parameters that are sent to the reconstruction module depend on the algorithm being implemented. The samples and parameters are written to the file using the save command, they are stored in the ascii format as double values with tab as delimiter between the values. The save command used depends on the algorithm being implemented. The parameters vary with the type of reconstruction algorithm being used. The code snippet for saving the necessary parameters for the three algorithms is given below:

Orthogonal Matching Pursuit:

```
save('C:\Users\abhishek_sharma92\Desktop\project\code\allOMP.m
at','lenb','lenk','lent','b','k','n','m','Fs','t','f','-
ascii','-double','-tabs')
```

Basis Pursuit:

```
save('C:\Users\abhishek_sharma92\Desktop\project\code\allBP.ma
t','lenb','lenk','lent','b','k','n','m','Fs','t','f','-
ascii','-double','-tabs')
```

Approximate Message Passing:

```
save('C:\Users\abhishek_sharma92\Desktop\project\code\allAMP.m
at','lenb','lenk','lent','b','k','n','m','Fs','T','tol','t','f
','-ascii','-double','-tabs')
```

Here,

b- Random samples

k- Vector containing indices of samples

n- Total number of samples (5000)

m- Number of samples taken (500)

Fs- Sampling frequency (40000 Hz)

t- Time at which each sample was taken

f- Original signal (sent only to calculate error)

T- Number of Iterations (1000)

tol - Tolerance (0.001)

6.3.2 Establish a connection with the reconstruction module

A connection is established between the sampling module (client) and the reconstruction module (server) using socket programming in Java. A socket program written in Java language can communicate to a program written in non-Java (say C or C++) socket program. A unique port number is assigned for the communication. A server (program) runs on a specific computer and has a socket that is bound to a specific port. The server listens to the socket for a client to make a connection. If everything goes well, the server accepts the connection. The sampling module and the reconstruction module must be running on devices that are connected to the same network. The sampling module can only establish a connection with the reconstruction module if it knows the destination IP address. Once the IP address is verified, a communication channel is created.

The following code snippet demonstrates the sample file being sent:

```
client('192.168.2.8',3000,'C:\Users\abhishek_sharma92\Desktop\
project\code\allAMP.mat')
```

The following snippet shows how the client establishes a connection with the server :

```
% connect to server
clientSocket = Socket(serverIp, serverPort);
fprintf(1, '\n\nConnected to server\n');
```

6.3.3 Send file to the reconstruction module

In this step, the file is sent to the intended destination. A buffer is created for the data to be written into and then the '.mat' file which contains the data is opened. A channel is then created and the contents of the '.mat' file are written completely to the buffer. After this, the connection is closed.

The code snippet for the sequence of steps described above is given below:

```
% create buffer to write data into
array  = zeros(1,BUFFER_SIZE,'int8');
buffer = java.nio.ByteBuffer.wrap(array);

% open file to send
fin = FileInputStream(fileToSend);
bin = java.nio.channels.Channels.newChannel(fin);

% create channel to write to
cis = clientSocket.getOutputStream();
myChannel = java.nio.channels.Channels.newChannel(cis);

fprintf(1, 'Starting to Send File\n');

while bin.read(buffer) > 0 % till there is nothing to read
   % output read data to socket
    buffer.flip();
    myChannel.write(buffer);
    buffer.clear();
end

% close everything
cis.close()
clientSocket.close();
bin.close();
fin.close();
fprintf(1, 'File Sent')
```

6.3.4 Retrieve the samples and parameters from the received file

Before the client can send the file, the server must be running. The following code snippet runs the server:

```
% create server socket
serverSocket = ServerSocket(serverPort);
fprintf(1, ['Server up on port %d\n'], serverPort);
```

When the client tries to establish a connection, the server accepts the connection and reads the contents of the buffer and writes it to a file on the device on which the server is running. The code snippet for the process is given below:

```
% accept incoming connection
clientSocket = serverSocket.accept();
clientIp=clientSocket.getInetAddress().toString();
fprintf(1, 'Client Connected, IP = %s\n', char(clientIp));
% create buffer to put read data into
array  = zeros(1,BUFFER_SIZE,'int8');
buffer = java.nio.ByteBuffer.wrap(array);

% create channel to read from
cis = clientSocket.getInputStream();
myChannel = java.nio.channels.Channels.newChannel(cis);

% create output file to put received file in
fout = FileOutputStream(fileName);
bof = java.nio.channels.Channels.newChannel(fout);

fprintf(1, 'Starting to Write File\n');

while myChannel.read(buffer) > 0 % till there is nothing
to read
    % output read data to file
    buffer.flip();
    bof.write(buffer);
    buffer.clear();
end
```

```
% close everything
cis.close();
fout.flush();
bof.close();
fout.close();
clientSocket.close;
serverSocket.close;
fprintf(1, 'File Written\n');
```

The next step is to retrieve the values of the samples and various parameters correctly from the file. To do this, an empty cell array of large size is created. The parameters are retrieved based on prior knowledge of the length of each of the parameter vectors. These are then converted to double data types from string type. The code snippet for the above process is shown below:

```
s=cell(100000,1);
sizS = 100000;
lineCt = 1;
fid = fopen('C:\Users\abhishek_sharma92\Desktop\t1.txt');
tline = fgetl(fid);
while ischar(tline)
      s{lineCt} = tline;
lineCt = lineCt + 1;
tline = fgetl(fid);
end

%# remove empty entries in s
s(lineCt:end) = [];
lenb=str2double(s(1));
lenk=str2double(s(2));
b=str2double(s(3:(lenb+2)));
k=str2double(s(lenb+3:(lenb+lenk+2)));
n=str2double(s(lenb+lenk+3));
m=str2double(s(lenb+lenk+4));
Fs=str2double(s(lenb+lenk+5));
t=str2double(s((lenb+lenk+6):end));
```

6.4 Reconstruction

The final stage is the reconstruction of the original signal from the samples using the various parameters retrieved. The first step in doing so is the creation of the reconstruction matrix 'A'. The creation of this matrix is demonstrated through the following code snippet:

```
% A = rows of DCT matrix with indices of random sample.
A = zeros(m,n);
for i = 1:m
      ek = zeros(1,n);
      ek(k(i)) = 1;
      A(i,:) = idct(ek);
End
```

There are various types of reconstruction algorithms, The types and classification of the same is given in Figure 6.2.

The problem to be solved can now be expressed as a minimization problem of the following form:

$$\boldsymbol{min||x||_0 \; subject\ to\ Ax = b}$$

In this project, algorithms belonging to three different categories were employed for solving the above problem.

The algorithms implemented are:

- Orthogonal Matching Pursuit ,Type: Greedy Iterative Algorithms
- Basis Pursuit, Type: Convex Relaxation
- Approximate Message Passing, Type: Iterative Thresholding Algorithms

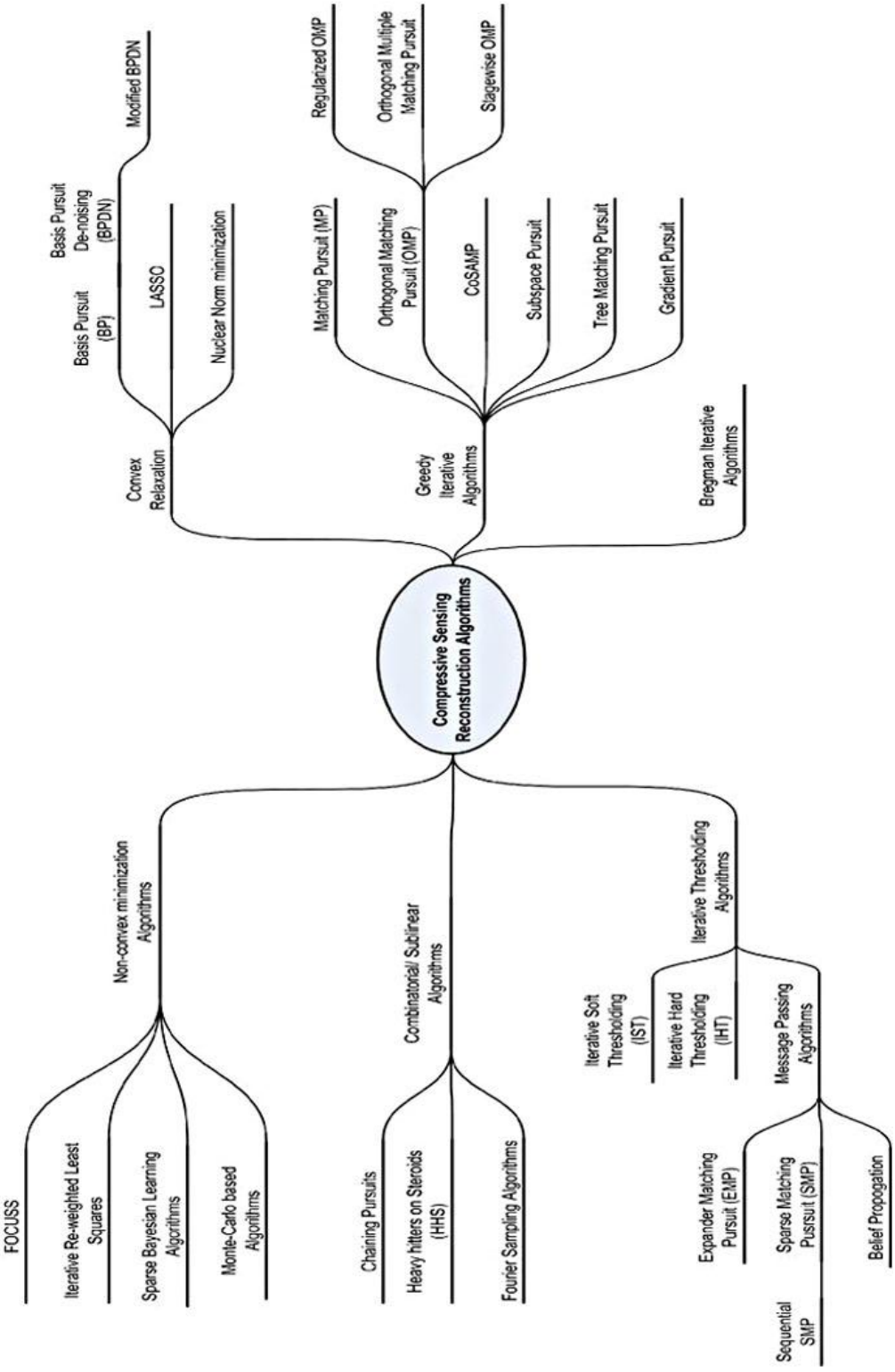

Figure 6.2 Compressed sensing reconstruction algorithms and their classification

6.4.1 Orthogonal Matching Pursuit

Orthogonal matching pursuit (OMP) constructs an approximation by going through an iteration process. At each iteration the locally optimum solution is calculated. This is done by finding the column vector in A which most closely resembles a residual vector r. The residual vector starts out being equal to the vector that is required to be approximated i.e. r = b and is adjusted at each iteration to take into account the vector previously chosen. It is the hope that this sequence of locally optimum solutions will lead to the global optimum solution.

Input:

- signal b and matrix A.
- stopping criterion e.g. until a level of accuracy is reached

Output:

- Approximation vector c.

Algorithm:

1. Start by setting the residual $r_0 = b$, the time t=0 and index set $V_{0=}\emptyset$

2. Let $v_t = i, where\ a_i$ gives the solution of max$< r_t, a_k >$ where a_k are the row vectors of A.
3. Update the set $V_t\ with\ v_t: V_t = V_{t-1} \cup \{V_t\}$

4. Solve the least-squares problem:

$$\boldsymbol{min_{c\in cV_t}} ||\boldsymbol{b} - \sum_{\boldsymbol{j=1}}^{\boldsymbol{t}} \boldsymbol{c(v_j)a_{v_j}}||_{\boldsymbol{2}}$$

5. Calculate the new residual using c

$$\boldsymbol{r_{t=}r_{t-1}} - \sum_{\boldsymbol{j=1}}^{\boldsymbol{t}} \boldsymbol{c(v_j)a_{v_j}}$$

6. Set $\boldsymbol{t \leftarrow t + 1}$

7. Check stopping criterion (residual=0). If the criterion has not been satisfied then return to step 2.

6.4.2 Basis Pursuit

The idea of Basis Pursuit is to replace the difficult sparse problem with an easier optimization problem. Below the formal definition of the sparse problem is given:

Sparse Problem:

$$\boldsymbol{min||x||_0 \; subject \; to \; Ax = b}$$

The difficultly with the above problem is the L_0 norm. Finding the L_0 norm is an NP-Hard problem. Basis Pursuit replaces the L_0 norm with the L_1 to make the problem easier to work with:

Basis Pursuit:

$$\boldsymbol{min||x||_1 \; subject \; to \; Ax = b}$$

The solution to the above problem can be found with relative ease. There are methods that will find the solution to the BP problem but does it lead to a sparse solution? The answer in general is no but under the right conditions it can be guaranteed that it will find a sparse solution or even the sparsest solution. This is because L1 norm is only concerned with the value of entries not the quantity. A vector with a small L1 could have very small valued non zero entries in every position which would give it a large L0 norm.

In this project, the Primal-Dual Interior Point Algorithm is used to solve the Basis Pursuit Problem.

6.4.2.1 Primal Dual Interior Point Algorithm

Primal problem:

(P) minimize $c^T x$

Subject to : $Ax = b, x \geq 0$

(m equalities ,n variables)

Dual problem:

(D) maximize $b^T y$

Subject to : $A^T y + s = c, s \geq 0$

Derivation summary:

Step 1: Remove the inequalities from (P) using a barrier term

(PB) minimize $c^T x - \rho \sum_j ln\,(x_j)$

Subject to: $Ax = b, x \geq 0$

Where, ρ is a positive barrier parameter.

Step 2: State the Lagrange function

The Lagrange function is:

$$L(x, y) \coloneqq c^T x - \rho \sum_j ln\,(x_j) - y^T(Ax - b)$$

Where, y contains the Lagrange multipliers.

Step 3: State the Lagrange optimality conditions

The optimality conditions are:

$$\nabla_x L(x, y) = c - \rho X^{-1} e - A^T y = 0,$$

$$\nabla_y L(x, y) = Ax - b = 0$$

Where $X \coloneqq diag(x) \coloneqq \begin{bmatrix} x1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & xn \end{bmatrix}, e = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$

Let, $s = \rho X^{-1} e$, hence $Xs = \rho e$

Equivalent optimality conditions:

(O) $Ax = b, x > 0$

$A^T y + s = c, s \geq 0$

$Xs = \rho e$

Step 4: Solving the optimality conditions

The nonlinear optimality conditions are solved using Newton's method:

$$\nabla f(x^k d_x) = f(x^k),$$

$$x^{k+1} = x^k + \alpha d_x$$

Where $\alpha \in (0,1]$ is the step size, solves f(x)=0.

Define:

$$F_\gamma(x, y, s) := \begin{bmatrix} Ax - b \\ A^T y + s - c \\ Xs - \gamma\mu e \end{bmatrix}, \rho := \gamma\mu = \gamma x^T s/n.$$

Where $\gamma \geq 0$

Given $(x^0, s^0) > 0$,

Then one step of Newton's method applied to

$$F_\gamma(x, y, s) = 0, x, s \geq 0$$

is given by :

$$\nabla F_\gamma(x^0, y^0, s^0) \begin{bmatrix} d_x \\ d_y \\ d_s \end{bmatrix} = -F_\gamma(x^0, y^0, s^0)$$

$$and \begin{bmatrix} x^1 \\ y^1 \\ s^1 \end{bmatrix} := \begin{bmatrix} x^0 \\ y^0 \\ s^0 \end{bmatrix} + \alpha \begin{bmatrix} d_x \\ d_y \\ d_s \end{bmatrix}, \text{ where } \alpha\text{=0.01}$$

Algorithm:

1. Choose (x^0, y^0, s^0) such that $x^0, s^0 > 0$.

2. Choose γ, θ ∈ (0, 1), ε >0

3. k := 0

4. while max $(\|Ax^k - b\|, \|A^T y^k + s^k - c\|, (x^k)^T s^k) \geq \varepsilon$

5. μ^k:= $((x^k)^T s^k)$/n

6. Solve:

$$Ad_x = -(Ax^k - b),$$

$$A^T d_y + d_s = -(A^T y^k + s^k - c),$$

$$s^k d_x + X^k d_s = -X^k s^k + \gamma \mu^k e.$$

7. Compute:

$$\alpha^k := \theta \max \{\bar{\alpha}: x^k + \bar{\alpha} d_x \geq 0, s^k + \bar{\alpha} d_s \geq 0, \theta\bar{\alpha} \leq 1\}$$

8. $(x^{k+1}; y^{k+1}; s^{k+1}) := (x^k; y^k; s^k) + \alpha^k (d_x; d_y; d_s)$

9. k := k+ 1

10. end while

6.4.3 Approximate Message Passing

Another class of algorithms with low computational complexity is the *Iterative Thresholding schemes*. ***Approximate Message Passing (AMP)*** is a variant of this scheme. AMP reconstructs the signal as effectively as L1 while running much faster. The idea behind these algorithms is that when a signal is represented in terms of a suitable basis, smaller coefficients are set to zeroes while the larger coefficients above a given threshold are possibly shrunk.

In each iteration, a residual is calculated along with a new threshold. With every step, these values change and the algorithm breaks when the stopping condition is reached. As

in the case of OMP, the stopping rule is dependent on the noise structure. For the noiseless case, the stopping rule is that the residual reaches zero.

Algorithm:

1. $x^{t+1} = \eta(A^* z^t + x^t; \tau^t)$.

2. $b - Ax^t + \frac{z^{t-1}}{\delta}\langle \eta'(A^* z^{t-1} + x^{t-1}; \tau^{t-1})\rangle$, here the last term is called the Onsager Reaction Term.

3. $\tau^t = \frac{\tau^{t-1}}{\delta}\langle \eta'(A^* z^{t-1} + x^t; \tau^{t-1})\rangle$.

Here,

x^t– Current estimate

A – Measurement matrix

A *- Transpose of A

z^t- Current residual

b- Vector of interest (m x 1)

t- Iteration counter

$\eta -$ Scalar threshold function

τ- Threshold

δ- Under sampling rate

Here,

δ= m/n

τ= 0.001, initially

$$\eta(x;\tau) = \begin{cases} x + \tau, x < -\tau \\ 0, -\tau \le x \le \tau, \\ x - \tau, x > \tau \end{cases}$$

For a vector u=(u(1),u(2)…u(n)),

$\langle u \rangle \equiv \sum_{i=1}^{n} u(i)/n$,

$\eta'(x) = \frac{\partial(\eta(x)}{\partial x}$