sparse-plex

Release 0.1

Shailesh

Apr 14, 2018

Contents

Contents:

CHAPTER 1

Introduction

Sparse-plex is a MATLAB library for solving sparse representation problems.

Currently the library is broken as it is going through massive changes. Will update later.

The library website is : http://indigits.github.io/sparse-plex/.

Online documentation is hosted at: http://sparse-plex.readthedocs.org/en/latest/.

The project is hosted on GITHUB at: https://github.com/indigits/sparse-plex.

It contains implementations of many state of the art algorithms. Some implementations are simple and straight-forward while some have taken extra efforts to optimize the speed.

In addition to these, the library provides implementations of many other algorithms which are building blocks for the sparse recovery algorithms.

The library aims to solve:

- Single vector sparse recovery or sparse approximation problems
- Multiple vector joint sparse recovery or sparse approximation problems

The library provides

- Various simple dictionaries and sensing matrices
- Implementations of pursuit algorithms
  - Matching pursuit
  - Orthogonal matching pursuit
  - Compressive sampling matching pursuit
  - Basis pursuit
- Some joint recovery algorithms
  - Cluster orthogonal matching pursuit
- Some clustering algorithms

  - Spectral clustering
  - Sparse subspace clustering using l_1 minimization
  - Sparse subspace clustering using orthogonal matching pursuit
- Various utilities for working with matrices, signals, norms, distances, signal comparison, vector spaces
- Some visualization utilities
- Some combinatoric systems
- Various constructions for synthetic sparse signals
- Some optimization algorithms
  - steepest descent
  - conjugate gradient descent
- Detection and estimation algorithms
  - Compressive binary detector

The documentation contains several how-to-do tutorials. They are meant to help beginners in the area ramp up quickly. The documentation is not really a user manual. It doesn't describe all parameters and behavior of a function in detail. Rather, it provides various code examples to explain how things work. Users are requested to read through the source code and relevant papers to get a deeper understanding of the methods.

CHAPTER 2

Getting Started

2.1 Requirements

While much of the library can be used on stock MATLAB distribution with standard toolboxes, some parts of the library are dependent on some specific third party libraries. These dependencies are explained below.

MATLAB toolboxes

- Signal processing toolbox
- Image processing toolbox
- Statistics toolbox
- Optimization toolbox

Third party library dependencies

- CVX http://cvxr.com/cvx/
- LRS library https://github.com/andrewssobral/lrslibrary
- Wavelab http://statweb.stanford.edu/~wavelab/
-

We repeat that only some parts of the library and examples depend on the third party libraries. You can install them on need basis. You don't need to install them in advance.

2.2 Installation

- Download `sparse-plex` library from http://indigits.github.io/sparse-plex/.
- Unzip it in a suitable folder.
- Add following commands to your MATLAB startup script:
  - Change directory to the root directory of `sparse-plex`.

  - Run `spx_setup` function.
  - Change back to whatever directory you want to be in.

Note: Make sure that MATLAB has write permissions to the directory in which you install sparse-plex. Some functions in sparse-plex create some MAT files for caching of intermediate results. Moreover, the sparse-plex setup script also creates a local settings file. For creating these files, write access is needed.

2.3 Getting acquainted

The online library documentation includes a number of step-by-step demonstrations. Follow these tutorials to get familiar with the library.

2.4 Running examples

- Change directory to the root directory of `sparse-plex`.
- Go into `examples` directory.
- Browse the examples.
- Run the example you want.

2.5 Checking the source code

- Change directory to the root directory of `sparse-plex`.
- Go into `library` directory.
- Browse the source code.
  - The source code for `spx` library is maintained in `+spx` directory.
  - Unit-tests for the library are maintained in `tests` directory.

2.6 Verifying the installation

You will require MATLAB XUnit test framework to run the unit tests included with the library.

- Change directory to the root directory of `sparse-plex`.
- Move to the directory `library\\tests`.
- Execute the `runalltests.m` script.
- Verify that all unit tests pass.

2.7 Configuring test data directories

Several examples in sparse-plex are developed on top of standard data sets. These include (but not limited to):

- Standard test images
- Yale Extended B Faces database (cropped images)

In order to execute these examples, access to the data is needed. The data is not distributed along with this software. You can download data and store it on your computer wherever you wish. In order to provide access to this data, you need to tell sparse-plex where does the data lie. This can be done by changing `spx_local.ini` file. When you download and unzip the library, this file doesn't exist. When you run `spx_setup`, `spx_defaults.ini` is copied into `spx_local.ini`.

All you need to do is to point to the right directories which hold the test datasets.

Specific settings in `spx_local.ini` are:

- `standard_test_images_dir`
- `yale_faces_db_dir`

For more information, read the file.

2.8 Building documentation

Only if you really want to do it!

You will require Python Sphinx and other related packages like Pygments library etc. to build the documentation from scratch.

- Change directory to the root directory of `sparse-plex`.
- Go into `docs` directory.
- Build the documentation using Sphinx tool chain.

CHAPTER 3

Demos

3.1 Dirac DCT Tutorial

This tutorial is based on `examples\\ex_dirac_dct_two_ortho_basis.m`.

In this tutorial we will:

- Construct a DCT basis
- Construct a Dirac-DCT dictionary.
- Construct a signal which is a mixture of few impulses and a few sinusoids.
- Construct its representation in the DCT basis.
- Recover its representation in Dirac-DCT dictionary using following sparse recovery algorithms
  - Matching Pursuit
  - Orthogonal Matching Pursuit
  - Basis Pursuit
- Measure the recovery error for different sparse recovery algorithms.

Signal space dimension:

```
N = 256;
```

Dirac basis:

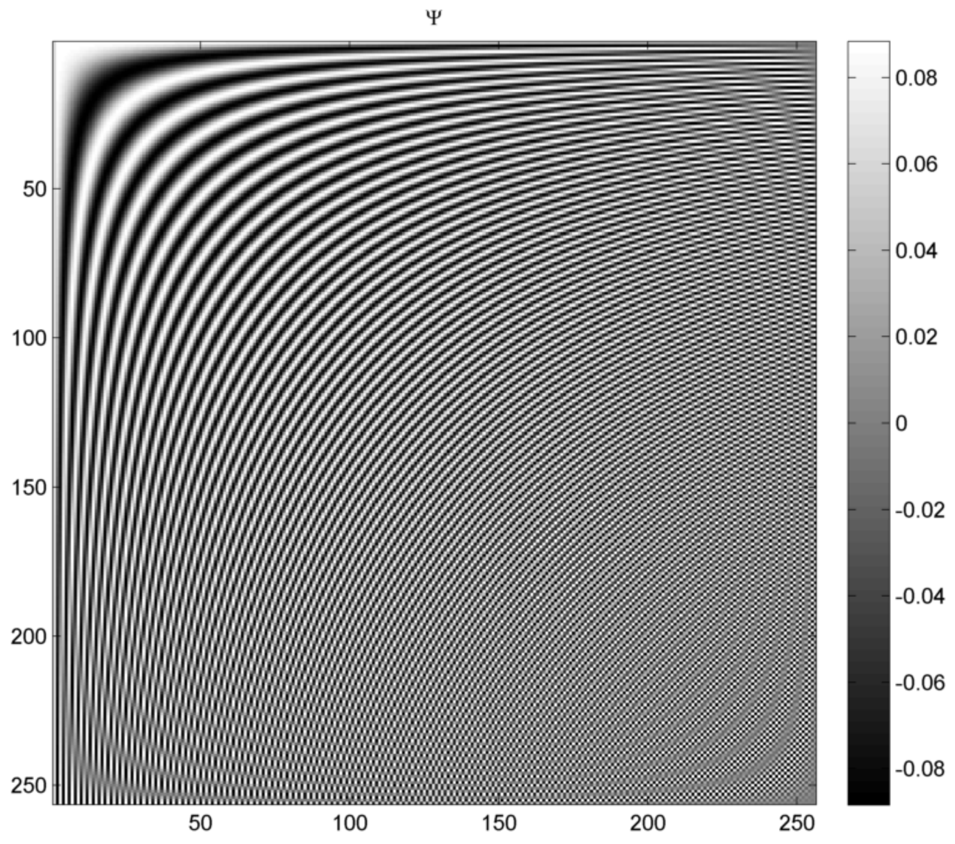
```
I = eye(N);
```

DCT basis:

```
Psi = dctmtx(N)';
```

Visualizing the DCT basis:

```
imagesc(Psi) ;
colormap(gray);
colorbar;
axis image;
title('\Psi');
```

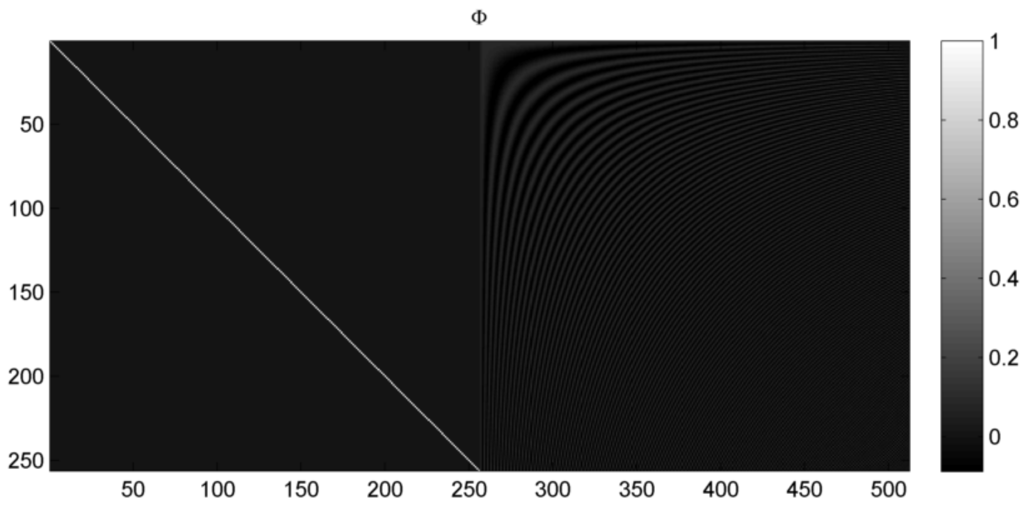


Combining the Dirac and DCT orthonormal bases to form a two-ortho dictionary:

```
Phi = [I  Psi];
```

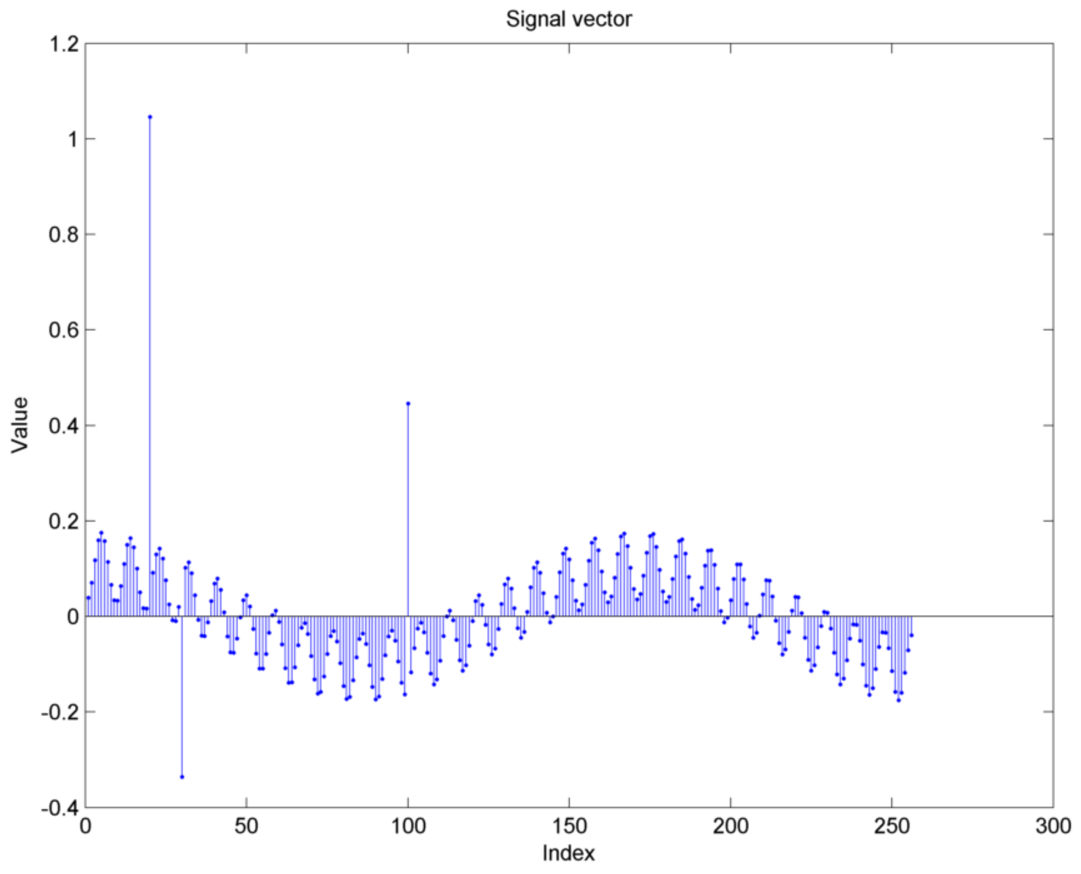
Visualizing the dictionary:

```
imagesc(Phi) ;
colormap(gray);
colorbar;
axis image;
title('\Phi');
```



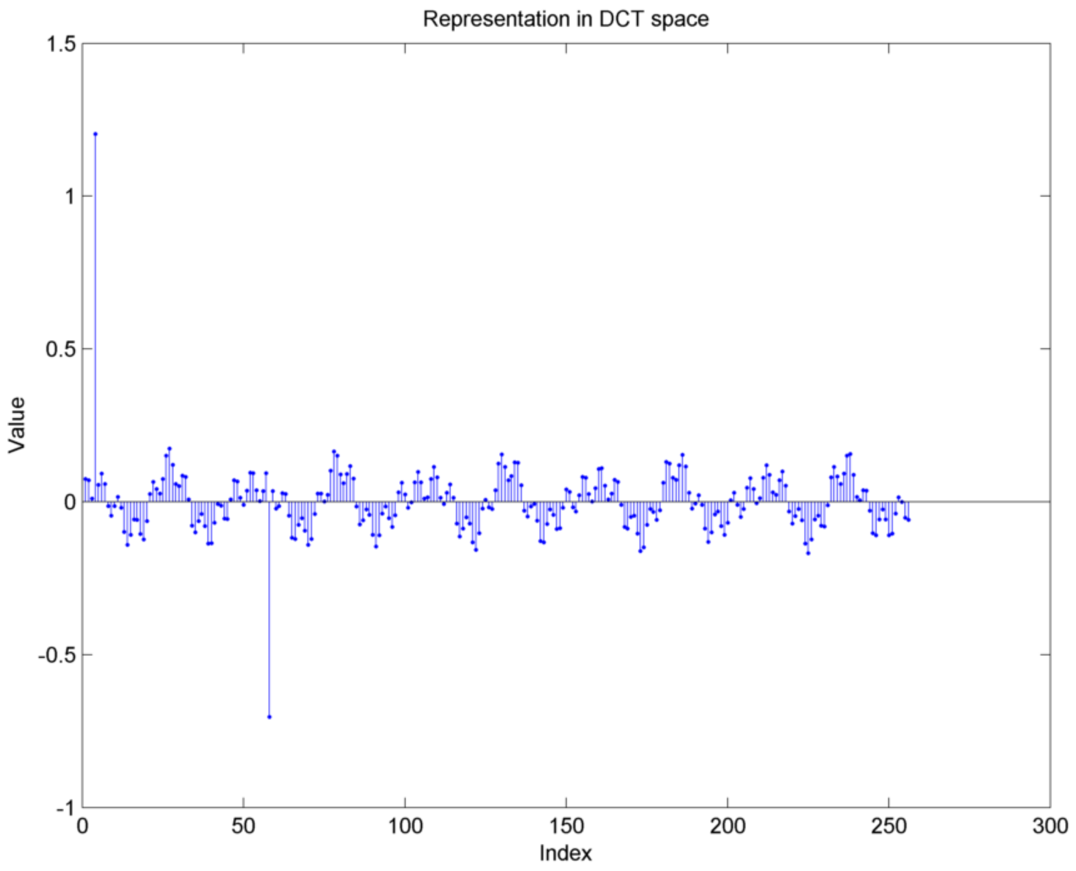
Constructing a signal which is a combination of impulses and cosines:

```
alpha = zeros(2*N, 1);
alpha(20) = 1;
alpha(30) = -.4;
alpha(100) = .6;
alpha(N + 4) = 1.2;
alpha(N + 58) = -.8;
x = Phi * alpha;
K  = 5;
```

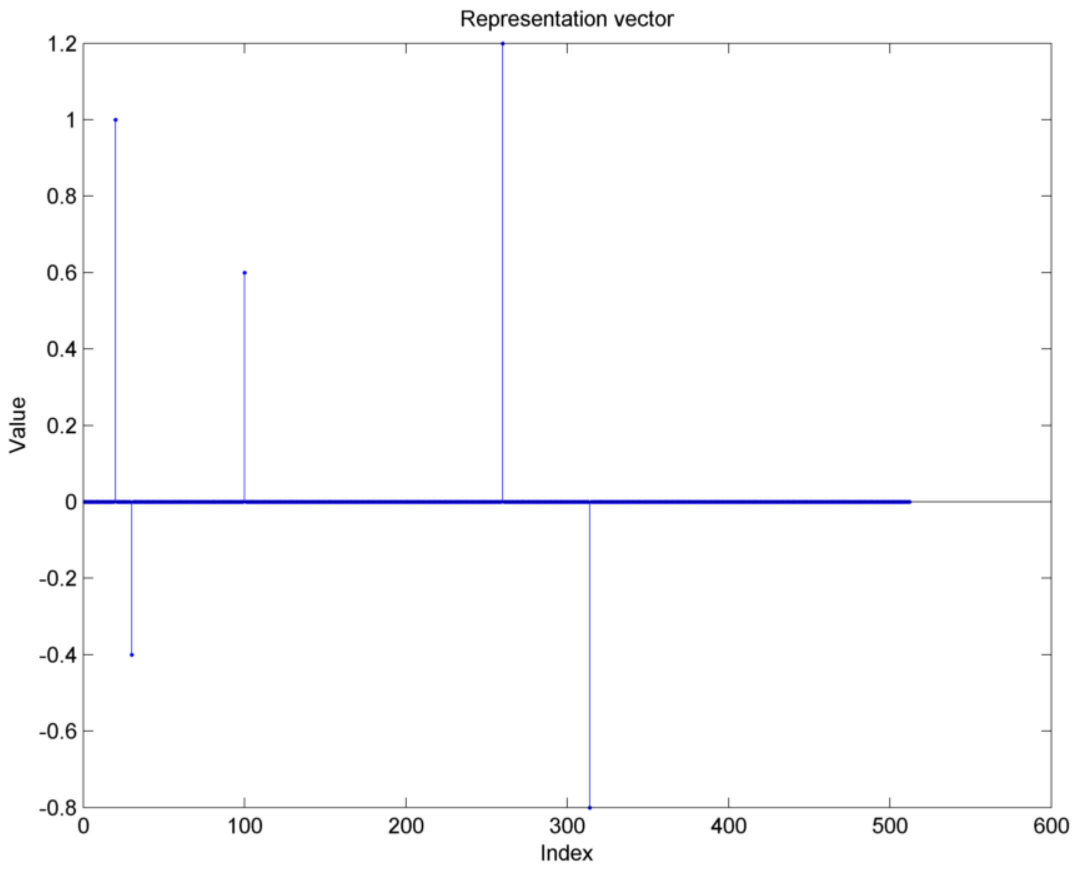


Finding the representation in DCT basis:

```
x_dct = Psi' * x;
```

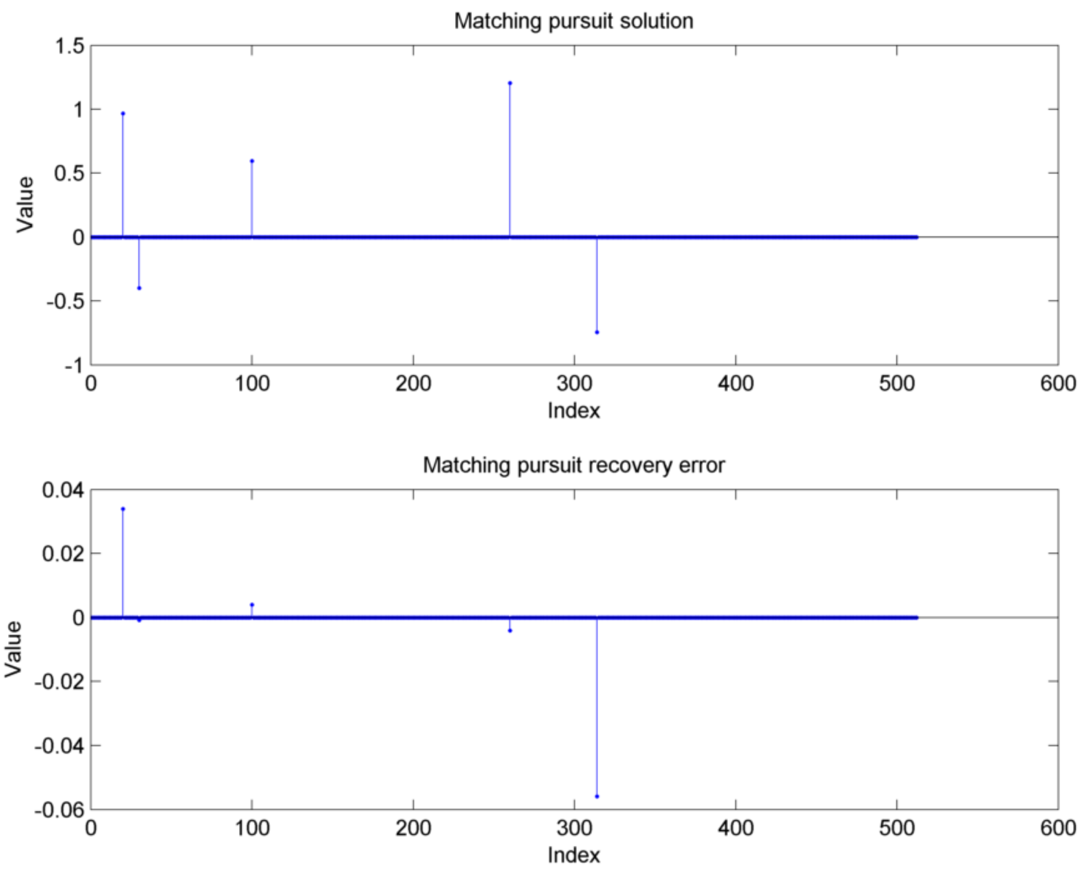


Sparse representation in the Dirac DCT dictionary



Obtaining the sparse representation using matching pursuit algorithm:

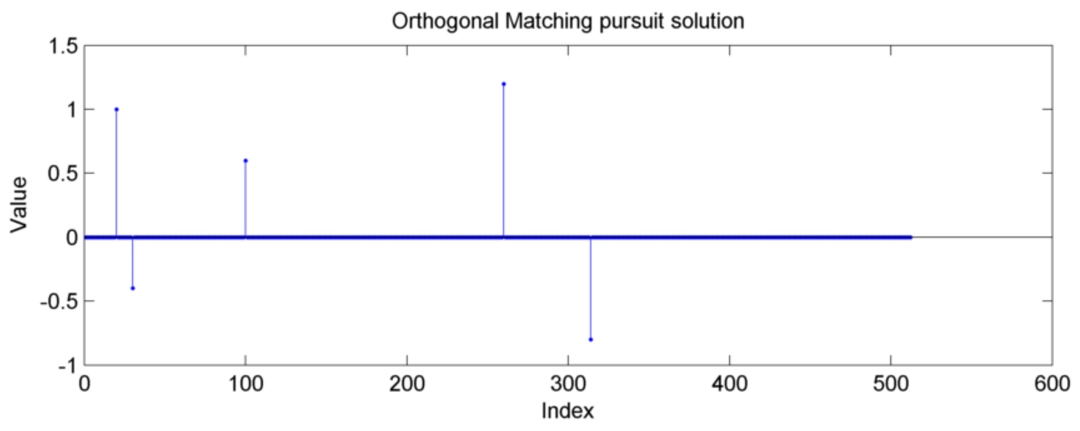
```
solver = spx.pursuit.single.MatchingPursuit(Phi, K);
result = solver.solve(x);
mp_solution = result.z;
mp_diff = alpha - mp_solution;
% Recovery error
mp_recovery_error = norm(mp_diff) / norm(x);
```

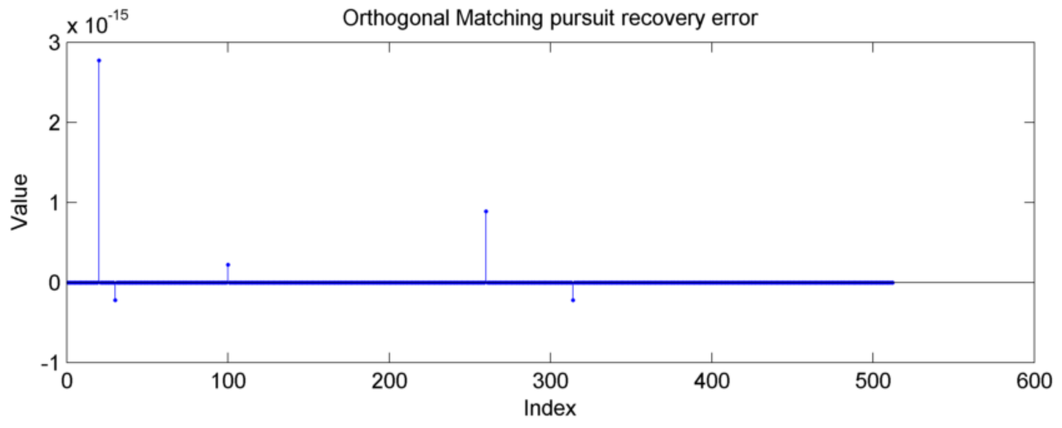


Matching pursuit recovery error: 0.0353.

Obtaining the sparse representation using orthogonal matching pursuit algorithm:

```
solver = spx.pursuit.single.OrthogonalMatchingPursuit(Phi, K);
result = solver.solve(x);
omp_solution = result.z;
omp_diff = alpha - omp_solution;
% Recovery error
omp_recovery_error = norm(omp_diff) / norm(x);
```

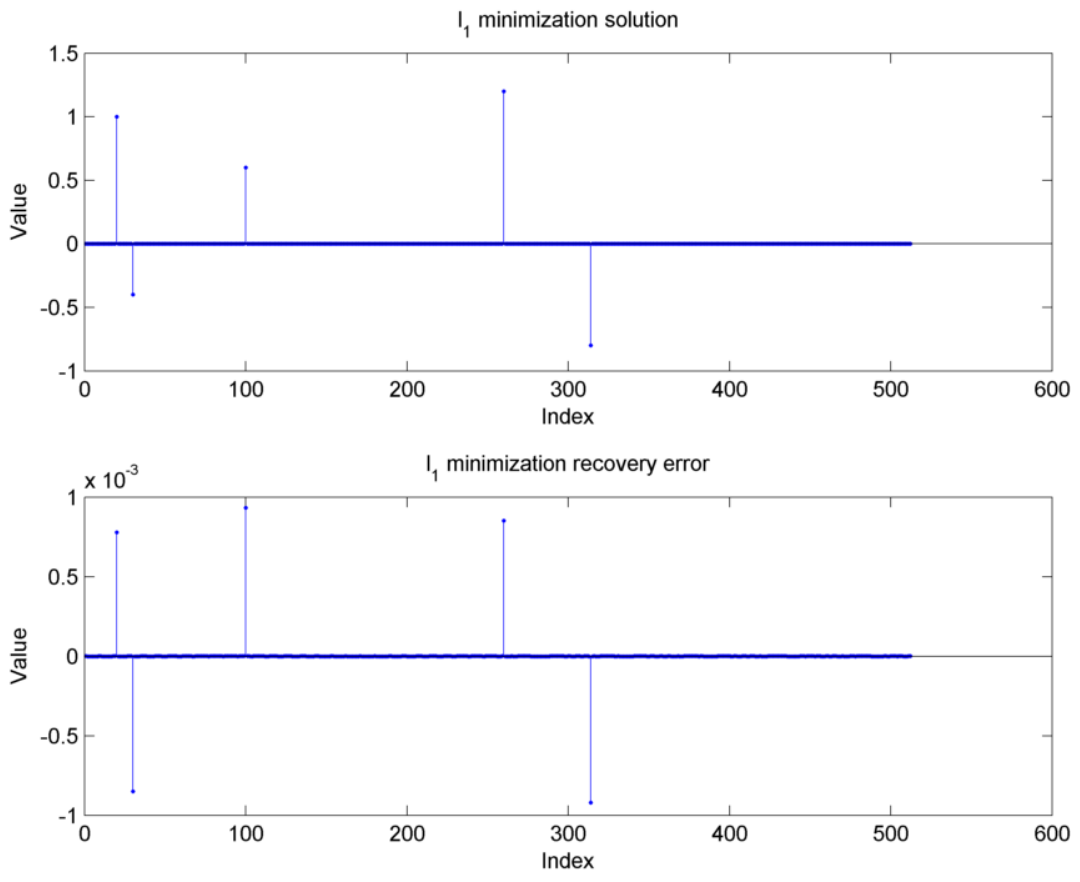




Orthogonal Matching pursuit recovery error: 0.0000.

Obtaining a sparse approximation via basis pursuit:

```
solver = spx.pursuit.single.BasisPursuit(Phi, x);
result = solver.solve_l1_noise();
l1_solution = result;
l1_diff = alpha - l1_solution;
% Recovery error
l1_recovery_error = norm(l1_diff) / norm(x);
```



l_1 recovery error: 0.0010.

3.2 Basic CS Tutorial

This tutorial is based on `examples\\ex_simple_compressed_sensing_demo.m`.

In this tutorial we will:

- Create sparse signals (with Gaussian and bi-uniform distributed non-zero samples).
- Look at how to identify support of a signal.
- Construct a Gaussian sensing matrix.
- Visualize the sensing matrix.
- Compute random measurements on the sparse signal with the sensing matrix.
- Add measurement noise to the measurements.
- Recover the sparse vector using following sparse recovery algorithms
  - Matching Pursuit
  - Orthogonal Matching Pursuit
  - Basis Pursuit

- Measure the recovery error for different sparse recovery algorithms.

Basic setup:

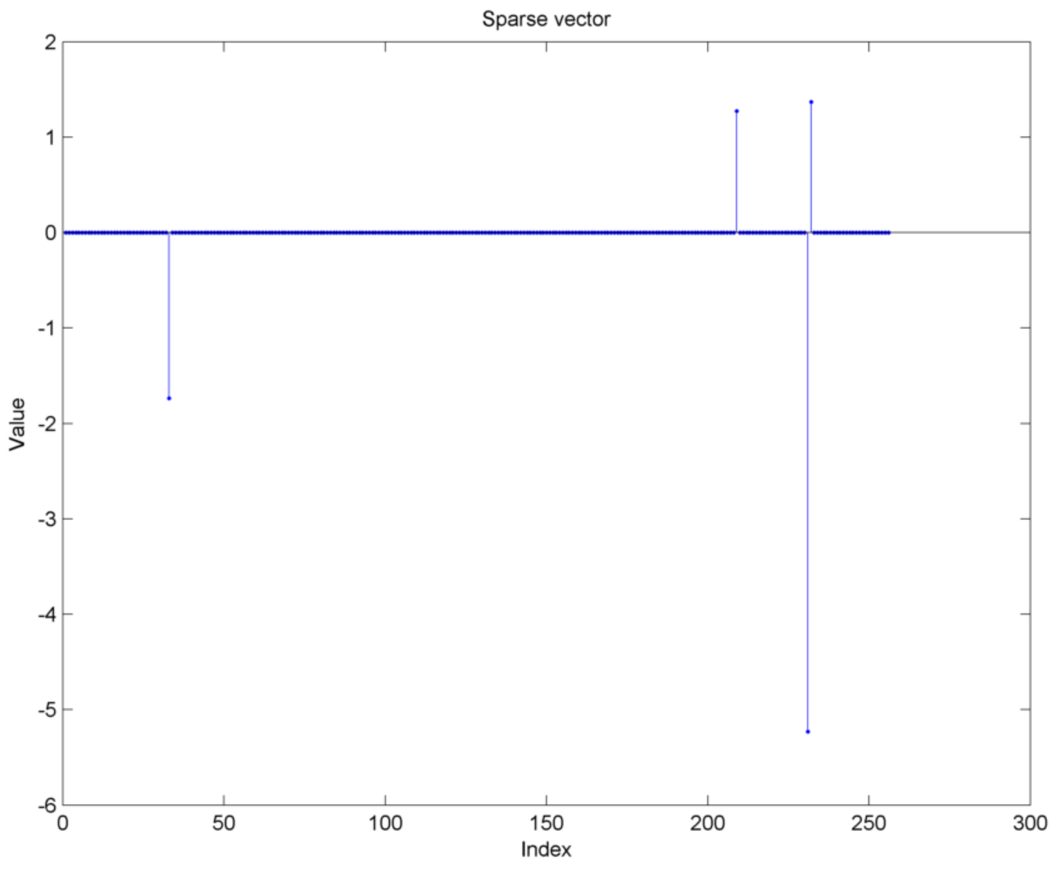
```
% Signal space
N = 1000;
% Number of measurements
M = 200;
% Sparsity level
K = 8;
```

Choosing the support randomly:

```
Omega = randperm(N, K);
```

Constructing a sparse vector with Gaussian entries:

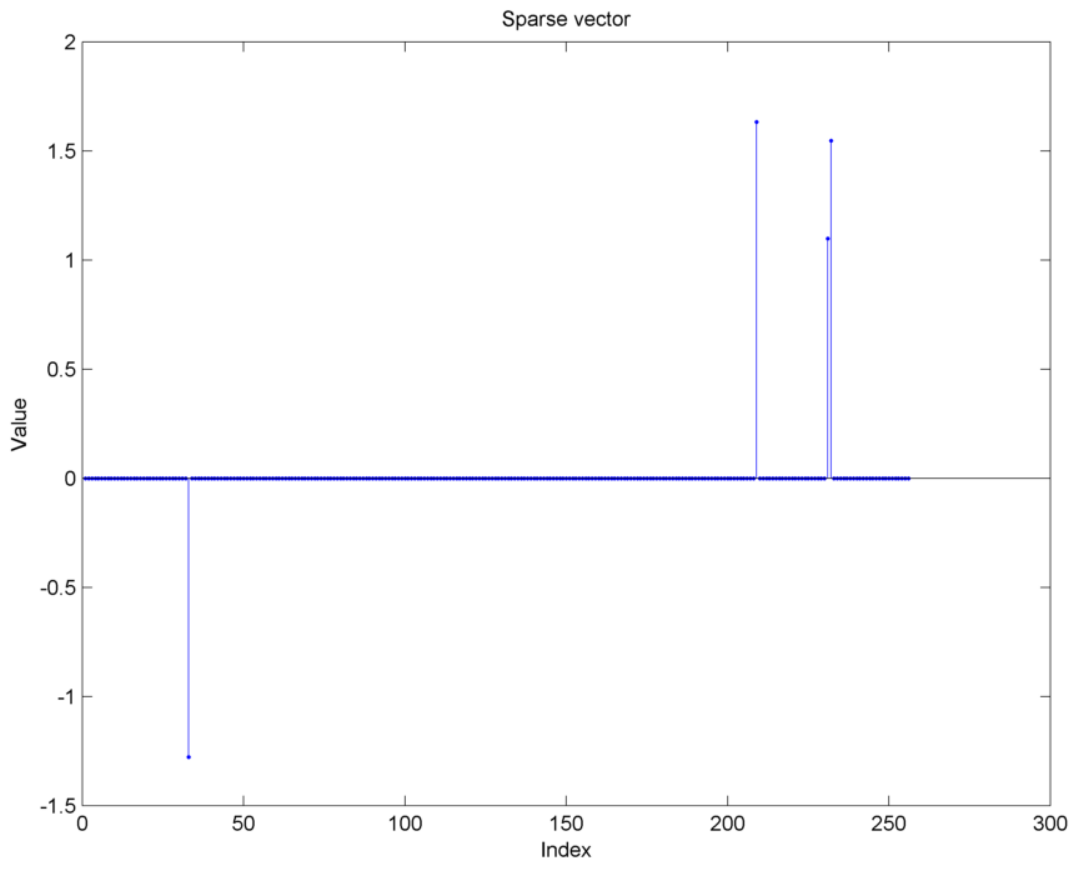
```
% Initializing a zero vector
x = zeros(N, 1);
% Filling it with non-zero Gaussian entries at specified support
x(Omega) = 4 * randn(K, 1);
```



Constructing a bi-uniform sparse vector:

```
a = 1;
b = 2;
```

```
% unsigned magnitudes of non-zero entries
xm = a + (b-a).*rand(K, 1);
% Generate sign for non-zero entries randomly
sgn = sign(randn(K, 1));
% Combine sign and magnitude
x(Omega) = sgn .* xm;
```



Identifying support:

```
find(x ~= 0)'
% 98    127    277    544    630    815    905    911
```

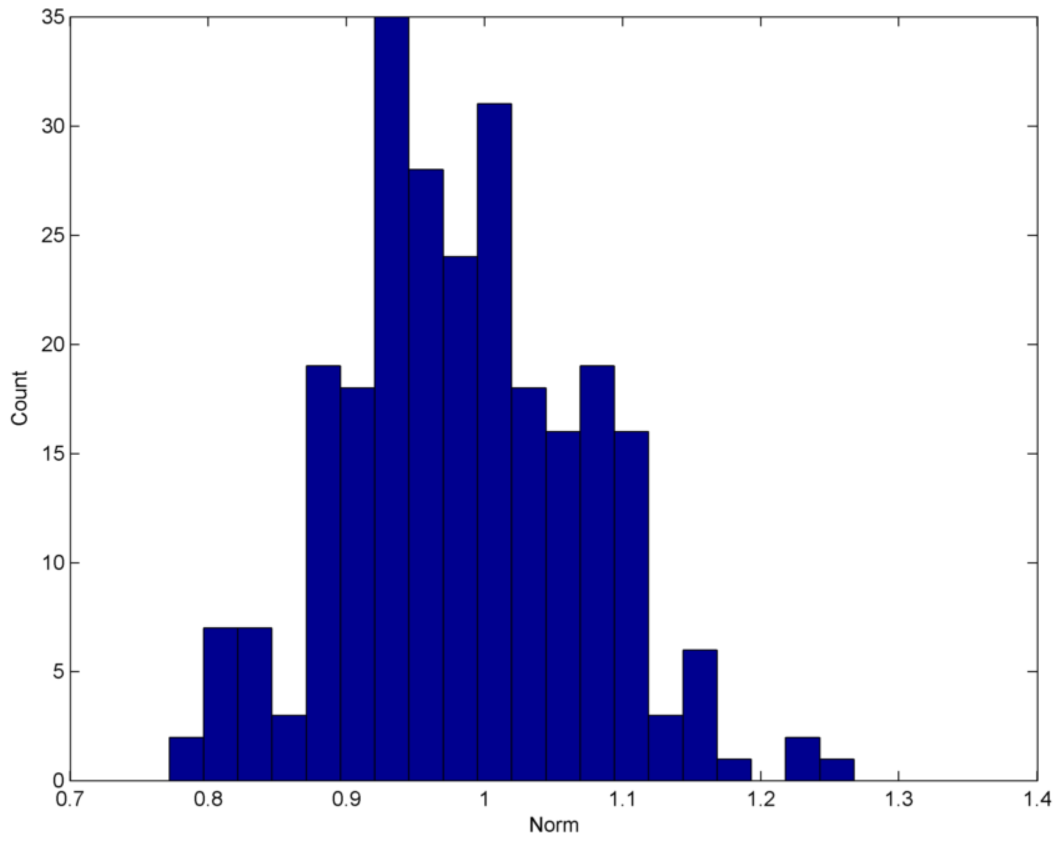
Constructing a Gaussian sensing matrix:

```
Phi = randn(M, N);
% Make sure that variance is 1/sqrt(M)
Phi = Phi ./ sqrt(M);
```

Computing norm of each column:

```
column_norms = sqrt(sum(Phi .* conj(Phi)));
```

Norm histogram

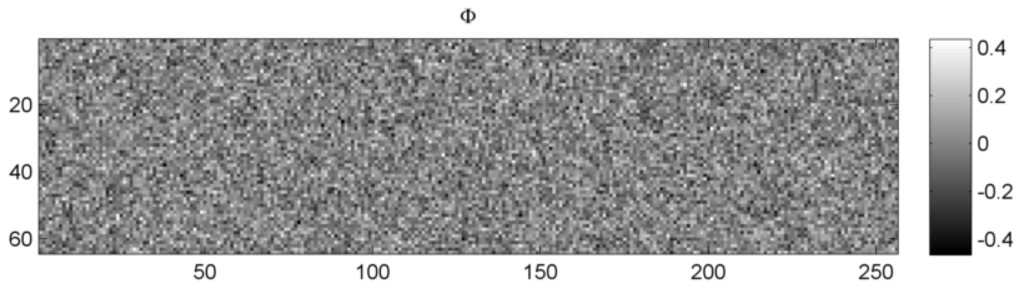


Constructing a Gaussian dictionary with normalized columns:

```
for i=1:N
    v = column_norms(i);
    % Scale it down
    Phi(:, i) = Phi(:, i) / v;
end
```

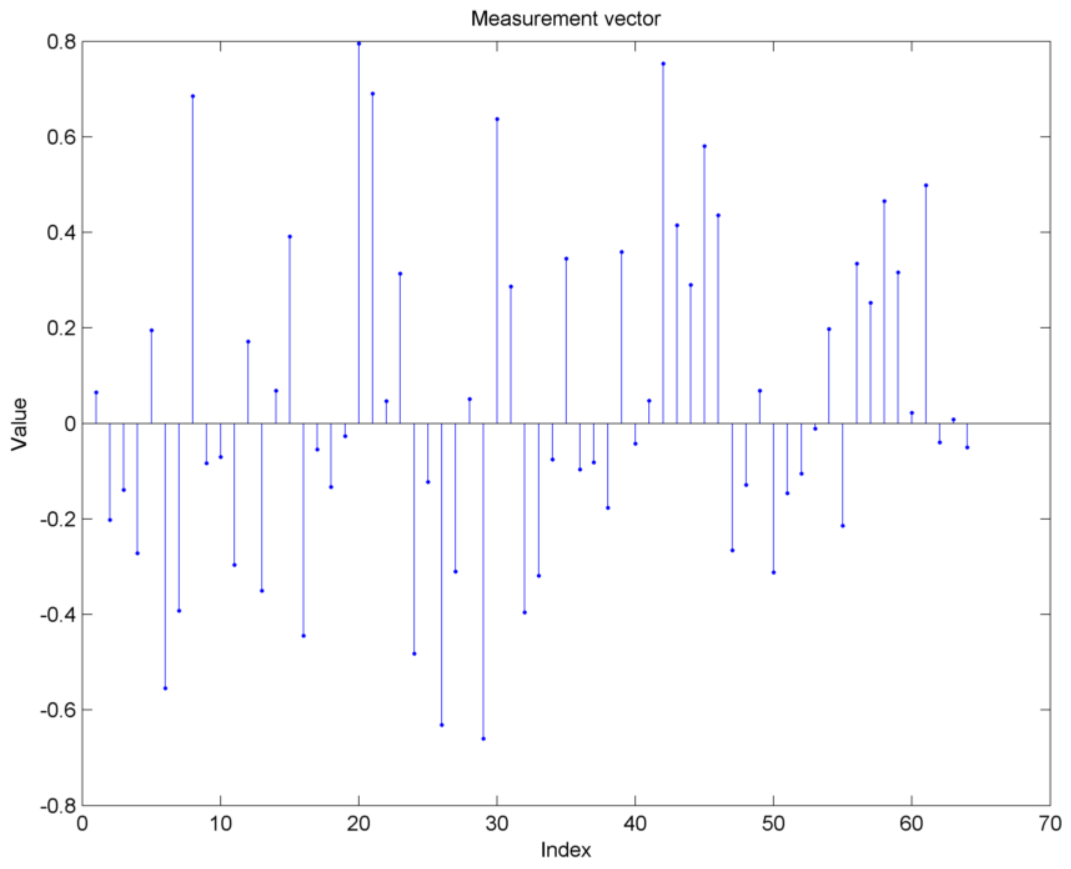
Visualizing the sensing matrix:

```
imagesc(Phi) ;
colormap(gray);
colorbar;
axis image;
```



Making random measurements from sparse high dimensional vector:

```
y0 = Phi * x;
```

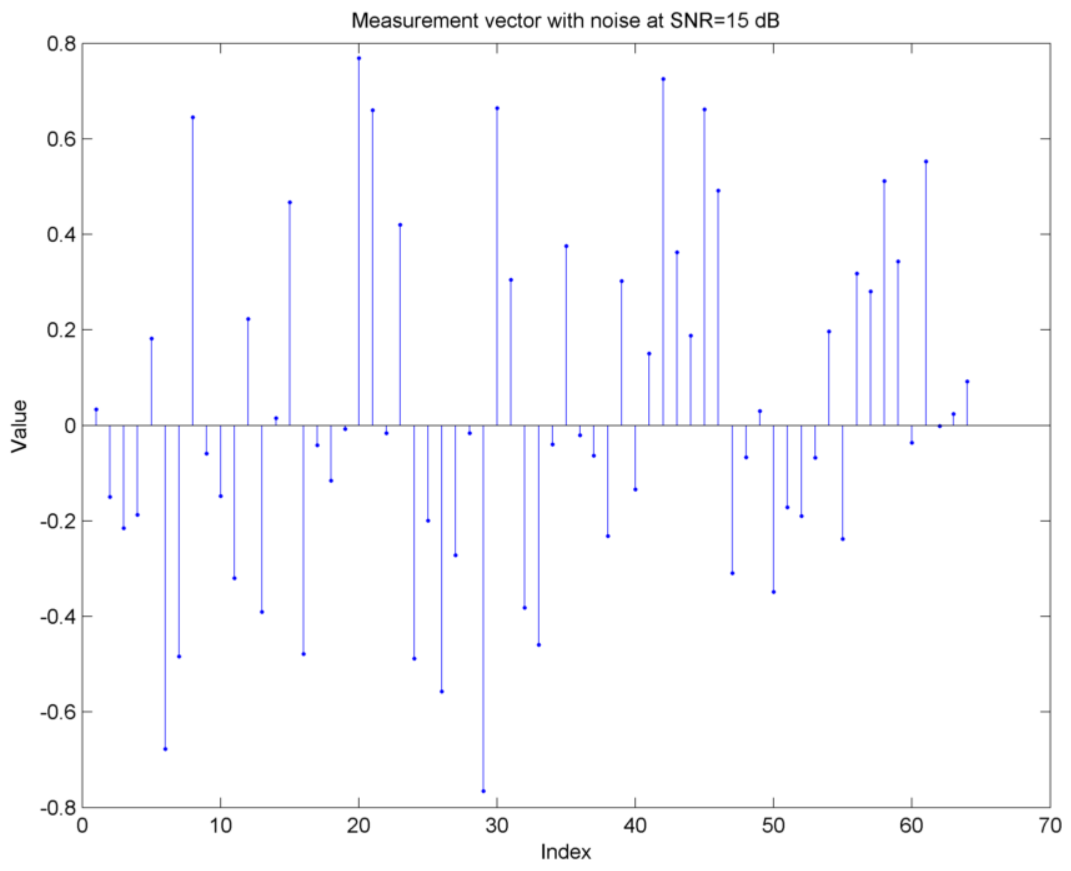


Adding some measurement noise:

```
SNR = 15;
snr = db2pow(SNR);
noise = randn(M, 1);
% we treat each column as a separate data vector
signalNorm = norm(y0);
noiseNorm = norm(noise);
actualNormRatio = signalNorm / noiseNorm;
requiredNormRatio = sqrt(snr);
gain_factor = actualNormRatio  / requiredNormRatio;
noise = gain_factor .* noise;
```

Measurement vector with noise:

```
y = y0 + noise;
```

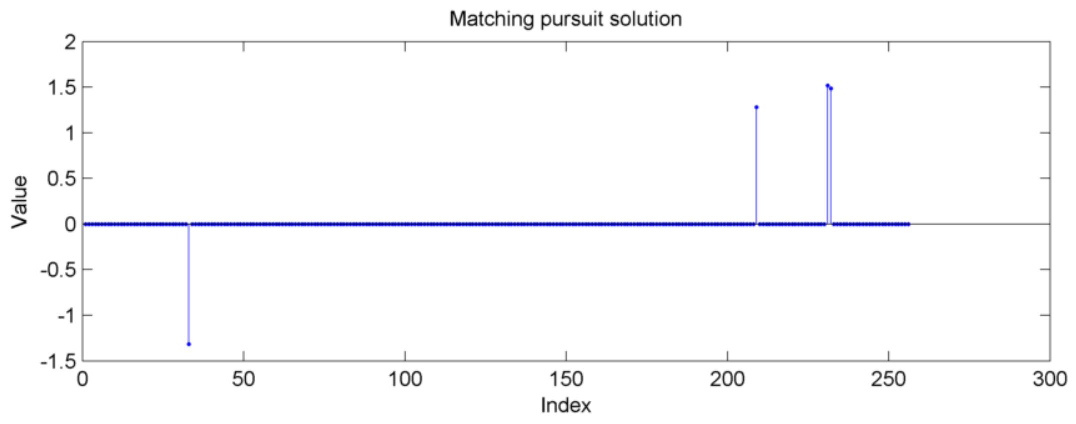


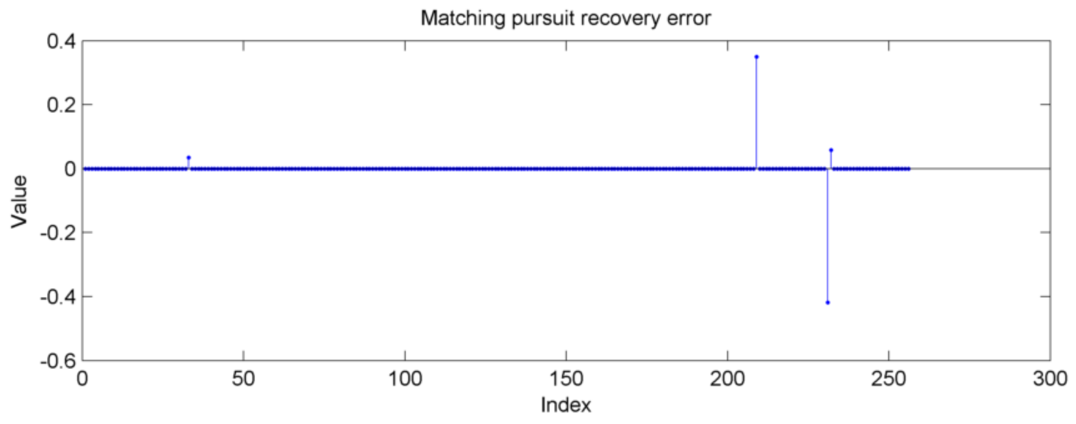
Sparse recovery using matching pursuit:

```
solver = spx.pursuit.single.MatchingPursuit(Phi, K);
result = solver.solve(y);
mp_solution = result.z;
```

Recovery error:

```
mp_diff = x - mp_solution;
mp_recovery_error = norm(mp_diff) / norm(x);
```

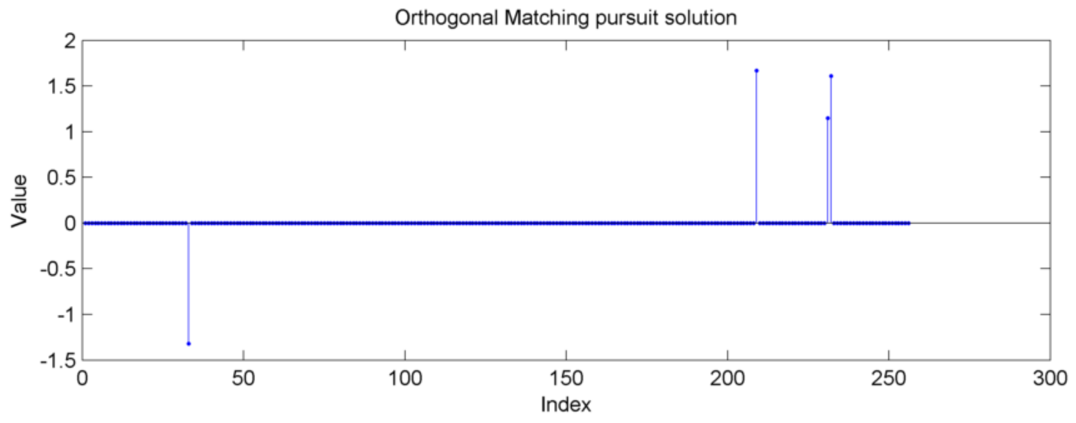


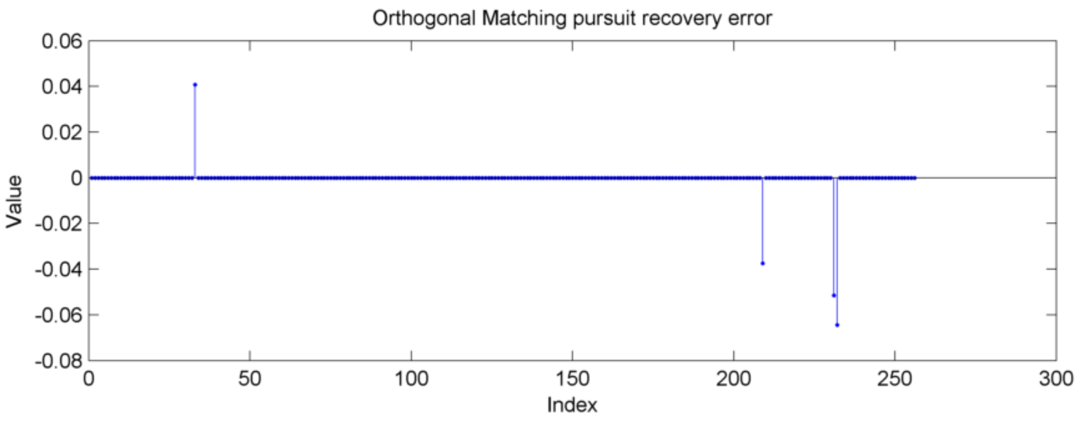


Matching pursuit recovery error: 0.1612.

Sparse recovery using orthogonal matching pursuit:

```
solver = spx.pursuit.single.OrthogonalMatchingPursuit(Phi, K);
result = solver.solve(y);
omp_solution = result.z;
omp_diff = x - omp_solution;
omp_recovery_error = norm(omp_diff) / norm(x);
```

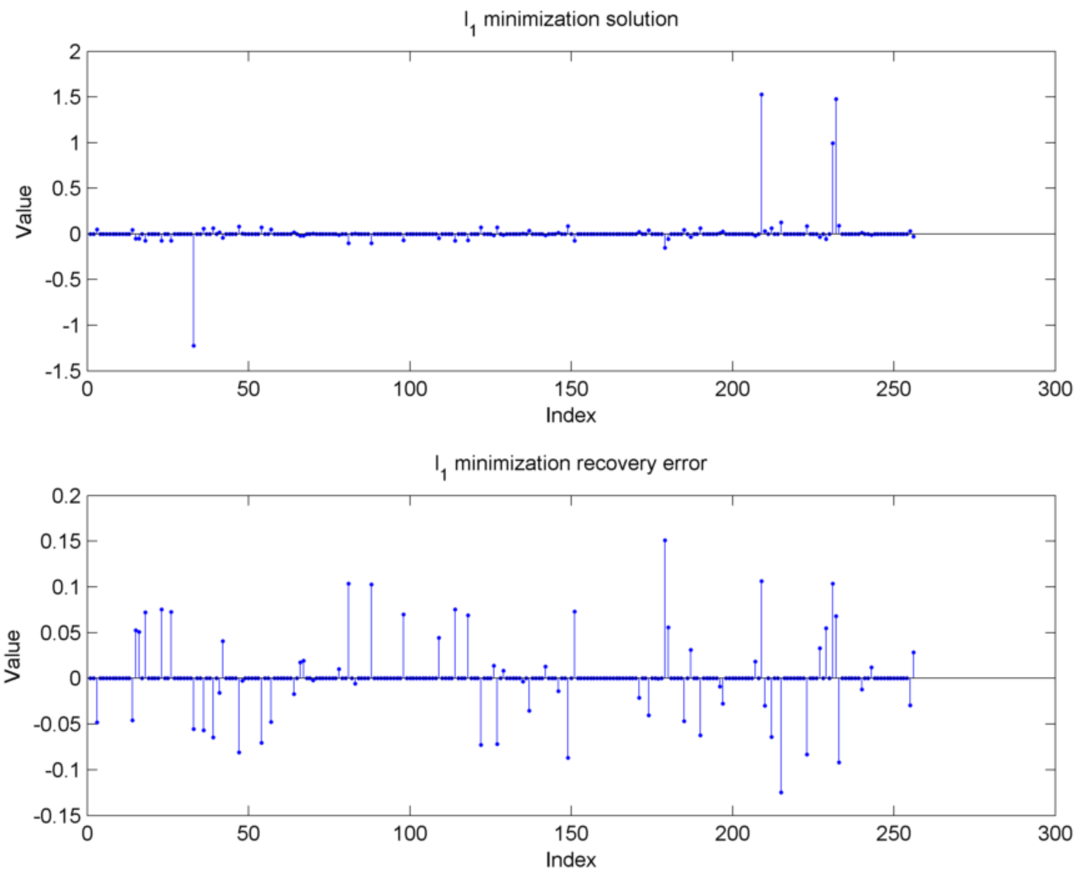




Orthogonal Matching pursuit recovery error: 0.0301.

Sparse recovery using l_1 minimization:

```
solver = spx.pursuit.single.BasisPursuit(Phi, y);
result = solver.solve_l1_noise();
l1_solution = result;
l1_diff = x - l1_solution;
l1_recovery_error = norm(l1_diff) / norm(x);
```



l_1 recovery error: 0.1764.

CHAPTER 4

Library Classes

Contents:

4.1 Sparse recovery pursuit algorithms

Contents:

4.1.1 Matching pursuit

Constructing the solver with dictionary and expected sparsity level:

```
solver = spx.pursuit.single.MatchingPursuit(Dict, K)
```

Using the solver to obtain the sparse representation of one vector:

```
result = solver.solve(y)
```

Using the solver to obtain the sparse representations of all vectors in the signal matrix Y independently:

```
result = solver.solve_all(Y)
```

4.1.2 Orthogonal matching pursuit

Constructing the solver with dictionary and expected sparsity level:

```
solver  = spx.pursuit.single.OrthogonalMatchingPursuit(Dict, K)
```

Using the solver to obtain the sparse representation of one vector:

```
result = solver.solve(y)
```

There are several ways of solving the least squares problem which is an intermediate step in the orthogonal matching pursuit algorithm. Some of these are described below.

Using the solver to obtain the sparse representation of one vector with incremental QR decomposition of the subdictionary for the least squares step:

```
result = solver.solve_qr(y)
```

Using the solver to obtain the sparse representations of all vectors in the signal matrix Y independently:

```
result = solver.solve_all(Y)
```

Using the solver to obtain the sparse representations of all vectors in the signal matrix Y independently using the `linsolve` method for least squares:

```
result = solver.solve_all_linsolve(Y)
```

4.1.3 Basis pursuit and its variations

Basis pursuit is a way of solving the sparse recovery problem via ℓ_1 minimization. We provide multiple implementations for different variations of the problem.

Note: These algorithms are dependent on the CVX toolbox. Please make sure to install them before using the algorithms.

Constructing the solver with dictionary and set of signals to be solved arranged in a signal matrix:

```
solver = spx.pursuit.single.BasisPursuit(Dict, Y)
```

Solving using LASSO method:

```
result = solver.solve_lasso(lambda)
result = solver.solve_lasso()
```

Solving using ℓ_1 minimization assuming that signals are exact sparse:

```
result = solver.solve_l1_exact()
```

Solving using ℓ_1 minimization assuming that signals are noisy:

```
result = solver.solve_l1_noise()
```

4.1.4 Compressive sampling matching pursuit

Constructing the solver with dictionary and expected sparsity level:

```
solver = spx.pursuit.single.CoSaMP(Dict, K)
```

Using the solver to obtain the sparse representation of one vector:

```
result = solver.solve(y)
```

Using the solver to obtain the sparse representations of all vectors in the signal matrix Y independently:

```
result = solver.solve_all(Y)
```

4.1.5 Joint recovery algorithms

Cluster orthogonal matching pursuit

Warning: This is new algorithm under research.

```
solver = spx.pursuit.joint.ClusterOMP(Dict, K)
result = solver.solve(Y)
```

Subspace clustering matching pursuit

Warning: This is new algorithm under research.

```
solver = SPX_SCluMP(Phi, K, options)
solver.recover(Y)
solver.cluster(Y)
```

4.1.6 Introduction

This section focuses on methods which solve the sparse recovery or sparse approximation problems for one vector at a time. A subsection on joint recovery algorithms focuses on solving problems where multiple vectors which have largely common supports can be solved jointly.

For each algorithm, there is a solver. The solver should be constructed first with the dictionary / sensing matrix and some other parameters like sparsity level as needed by the algorithm.

The solver can then be used for solving one problem at a time.

4.2 Common utilities

Contents:

4.2.1 Signals

Our focus is usually on finite dimensional signals. Such signals are usually stored as column vectors in MATLAB. A set of signals with same dimensions can be stored together in the form of a matrix where each column of the matrix is one signal. Such a matrix of signals is called a `signal matrix`.

In this section we describe some helper utility functions which provide extra functionality on top of existing support in MATLAB.

General

Constructing unit (column) vector in a given co-ordinate:

```
>> N = 8; i = 2;
>> spx.commons.vector.unit_vector(N, i)'
0      1      0      0      0      0      0      0
```

Sparsification

Finding the K-largest indices of a given signal:

```
>> x = [0 0 0  1 0 0 -1 0 0 -2 0 0 -3 0 0 7 0 0 4 0 0 -6];
>> K=4;
>> spx.commons.signals.largest_indices(x, K)'
16     22     19     13
```

Constructing the sparse approximation of `x` with `K` largest indices:

```
>> spx.commons.signals.sparseApproximation(x, K)'
0      0      0      0      0      0      0      0      0      0      0      0     -3      0      0␣
↪      7      0      0      4      0      0     -6
```

Searching

`spx.commons.signals.find_first_signal_with_energy_le` finds the first signal in a signal matrix `X` with an energy less than or equal to a given `threshold` energy:

```
[x, i] = spx.commons.signals.find_first_signal_with_energy_le(X, threshold);
```

`x` is the first signal with energy less than the given threshold. `i` is the index of the column in `X` holding this signal.

4.2.2 Working with matrices

Simple checks on matrices

Let us create a simple matrix:

```
A = magic(3);
```

Checking whether the matrix is a square matrix:

```
spx.commons.matrix.is_square(A)
```

Checking if it is symmetric:

```
spx.commons.matrix.is_symmetric(A)
```

Checking if it is a Hermitian matrix:

```
spx.commons.matrix.is_hermitian(A)
```

Checking if it is a positive definite matrix:

```
spx.commons.matrix.is_positive_definite(A)
```

Matrix utilities

`spx.commons.matrix.off_diagonal_elements` returns the off-diagonal elements of a given matrix in a column vector arranged in column major order.

```
A = magic(3);
spx.commons.matrix.off_diagonal_elements(A)'
ans =
    3     4     1     9     6     7
```

`spx.commons.matrix.off_diagonal_matrix` zeros out the diagonal entries of a matrix and returns the modified matrix:

```
spx.commons.matrix.off_diagonal_matrix(A)
ans =

     0     1     6
     3     0     7
     4     9     0
```

`spx.commons.matrix.off_diag_upper_tri_matrix` returns the off diagonal part of the upper triangular part of a given matrix and zeros out the remaining entries:

```
spx.commons.matrix.off_diag_upper_tri_matrix(A)

ans =

     0     1     6
     0     0     7
     0     0     0
```

`spx.commons.matrix.off_diag_upper_tri_elements` returns the elements in the off diagonal part of the upper triangular part of a matrix arranged in column major order:

```
spx.commons.matrix.off_diag_upper_tri_elements(A)'

ans =

     1     6     7
```

`spx.commons.matrix.nonzero_density` returns the ratio of total number of non-zero elements in a matrix with the size of the matrix:

```
spx.commons.matrix.nonzero_density(A)
ans = 1
```

diagonally dominant matrices

Checking whether a matrix is diagonally dominant:

```
spx.commons.matrix.is_diagonally_dominant(A)
```

Making a matrix diagonally dominant:

```
A = spx.commons.matrix.make_diagonally_dominant(A)
```

Both these functions have an extra parameter named `strict`. When set to true, strict diagonal dominance is considered / enforced.

4.2.3 Norms and distances

Distance measurement utilities

Let `X` be a matrix. Treat each column of `X` as a signal.

Euclidean distance between each signal pair can be computed by:

```
spx.commons.distance.pairwise_distances(X)
```

If `X` contains N signals, then the result is an `N x N` matrix whose (i, j)-th entry contains the distance between i-th and j-th signal. Naturally, the diagonal elements are all zero.

An additional second argument can be provided to specify the distance measure to be used. See the documentation of MATLAB `pdist` function for supported distance functions.

For example, for measuring city-block distance between each pair of signals, use:

```
spx.commons.distance.pairwise_distances(X, 'cityblock')
```

Following dedicated functions are faster.

Squared ℓ_2 distances between all pairs of columns of X:

```
spx.commons.distance.sqrd_l2_distances_cw(X)
```

Squared ℓ_2 distances between all pairs of rows of X:

```
spx.commons.distance.sqrd_l2_distances_rw(X)
```

Norm utilities

These functions help in computing norm or normalizing signals in a signal matrix.

Compute ℓ_1 norm of each column vector:

```
spx.norm.norms_l1_cw(X)
```

Compute ℓ_2 norm of each column vector:

```
spx.norm.norms_l2_cw(X)
```

Compute ℓ_∞ norm of each column vector:

```
spx.norm.norms_linf_cw(X)
```

Normalize each column vector w.r.t. ℓ_1 norm:

```
spx.norm.normalize_l1(X)
```

Normalize each column vector w.r.t. ℓ_2 norm:

```
spx.norm.normalize_l2(X)
```

Normalize each row vector w.r.t. ℓ_2 norm:

```
spx.norm.normalize_l2_rw(X)
```

Normalize each column vector w.r.t. ℓ_∞ norm:

```
spx.norm.normalize_linf(X)
```

Scale each column vector by a separate factor:

```
spx.norm.scale_columns(X, factors)
```

Scale each row vector by a separate factor:

```
spx.norm.scale_rows(X, factors)
```

Compute the inner product of each column vector in A with each column vector in B:

```
spx.norm.inner_product_cw(A, B)
```

4.2.4 Sparse signals

Working with signal support

Let's create a sparse vector:

```
>> x = [0 0 0  1 0 0 -1 0 0 -2 0 0 -3 0 0 7 0 0 4 0 0 -6];
```

Sparse support for a vector:

```
>> spx.commons.sparse.support(x)
4     7    10    13    16    19    22
```

ℓ_0 "norm" of a vector:

```
>> spx.commons.sparse.l0norm(x)
7
```

Let us create one more signal:

```
>> y = [3 0 0  0 0 0 0 0 4 0 0 -6 0 0 -5 0 0 -4 0 8 0];
>> spx.commons.sparse.l0norm(y)
6
>> spx.commons.sparse.support(y)
1    10    13    16    19    21
```

Support intersection ratio:

```
>> spx.commons.sparse.support_intersection_ratio(x, y)
0.1364
```

It is the ratio between the size of common indices in the supports of x and y and maximum of the sizes of supports of x and y.

Average support similarity of a reference signal with a set of signals X (each signal as a column vector):

```
spx.commons.sparse.support_similarity(X, reference)
```

Support similarities between two sets of signals (pairwise):

```
spx.commons.sparse.support_similarities(X, Y)
```

Support detection ratios

```
spx.commons.sparse.support_detection_rate(X, trueSupport)
```

K largest indices over a set of vectors:

```
spx.commons.sparse.dominant_support_merged(data, K)
```

Sometimes it's useful to identify and arrange the non-zero entries in a signal in descending order of their magnitude:

```
>> spx.commons.sparse.sorted_non_zero_elements(x)
16    22    19    13    10     4     7
 7    -6     4    -3    -2     1    -1
```

Given a signal `x`, the function `spx.commons.sparse.sorted_non_zero_elements` returns a two row matrix where the first row contains the locations of non-zero elements sorted by their magnitude and second row contains their magnitude. If the magnitude of two non-zero elements is same, then the original order is maintained. The sorting is stable.

4.2.5 Comparing signals

Comparing sparse or approximately sparse signals

`spx.commons.SparseSignalsComparison` class provides a number of methods to compare two sets of sparse signals. It is typically used to compare a set of original sparse signals with corresponding recovered sparse signals.

Let us create two signals of size (N=256) with sparsity level (K=4) with the non-zero entries having magnitude chosen uniformly between [1,2]:

```
N = 256;
K = 4;
% Constructing a sparse vector
% Choosing the support randomly
Omega = randperm(N, K);
% Number of signals
S = 2;
% Original signals
X = zeros(N, S);
% Choosing non-zero values uniformly between (-b, -a) and (a, b)
a = 1;
b = 2;
% unsigned magnitudes of non-zero entries
XM = a + (b-a).*rand(K, S);
% Generate sign for non-zero entries randomly
sgn = sign(randn(K, S));
% Combine sign and magnitude
```

```
XMS = sgn .* XM;
% Place at the right non-zero locations
X(Omega, :) = XMS;
```

Let us create a noisy version of these signals with noise only in the non-zero entries at 15 dB of SNR:

```
% Creating noise using helper function
SNR = 15;
Noise = spx.data.noise.Basic.createNoise(XMS, SNR);
Y = X;
Y(Omega, :) = Y(Omega, :) + Noise;
```

Let us create an instance of sparse signal comparison class:

```
cs = spx.commons.SparseSignalsComparison(X, Y, K);
```

Norms of difference signals [X - Y]:

```
cs.difference_norms()
```

Norms of original signals [X]:

```
cs.reference_norms()
```

Norms of estimated signals [Y]:

```
cs.estimate_norms()
```

Ratios between signal error norms and original signal norms:

```
cs.error_to_signal_norms()
```

SNR for each signal:

```
cs.signal_to_noise_ratios()
```

In case the signals X and Y were not truly sparse, then `spx.commons.SignalsComparison` has the ability to sparsify them by choosing the `K` largest (magnitude) entries for each signal in reference signal set and estimated signal set. `K` is an input parameter taken by the class.

We can access the sparsified reference signals:

```
cs.sparse_references()
```

We can access the sparsified estimated signals:

```
cs.sparse_estimates()
```

We can also examine the support index set for each sparsified reference signal:

```
cs.reference_sparse_supports()
```

Ditto for the supports of sparsified estimated signals:

```
cs.estimate_sparse_supports()
```

We can measure the support similarity ratio for each signal

```
cs.support_similarity_ratios()
```

We can find out which of the signals have a support similarity above a specified threshold:

```
cs.has_matching_supports(1.0)
```

Overall analysis can be easily summarized and printed for each signal:

```
cs.summarize()
```

Here is the output

```
Signal dimension: 256
Number of signals: 2
Combined reference norm: 4.56207362
Combined estimate norm: 4.80070407
Combined difference norm: 0.81126416
Combined SNR: 15.0000 dB
Specified sparsity level: 4

Signal: 1
  Reference norm: 2.81008750
  Estimate norm: 2.91691022
  Error norm: 0.49971207
  SNR: 15.0000 dB
  Support similarity ratio: 1.00

Signal: 2
  Reference norm: 3.59387311
  Estimate norm: 3.81292464
  Error norm: 0.63909106
  SNR: 15.0000 dB
  Support similarity ratio: 1.00
```

Signal space comparison

For comparing signals which are not sparse, we have another helper utility class `spx.commons.SignalsComparison`.

Assuming X is a signal matrix (with each column treated as a signal), and Y is its noisy version, we created the signal comparison instance as:

```
cs = spx.commons.SignalsComparison(X, Y);
```

Most functions are similar to what we had for `spx.commons.SparseSignalsComparison`:

```
cs.difference_norms()
cs.reference_norms()
cs.estimate_norms()
cs.error_to_signal_norms()
cs.signal_to_noise_ratios()
cs.summarize()
```

4.2.6 Working with Numbers

Some algorithms from number theory are useful at times.

Finding integer factors closest to square root:

```
>> [a,b] = spx.discrete.number.integer_factors_close_to_sqr_root(120)
a = 10
b = 12
```

4.2.7 Printing utilities

Sparse signals

Printing a sparse signal as pairs of locations and values:

```
>> x = [0 0 0  1 0 0 -1 0 0 -2 0 0 -3 0 0 7 0 0 4 0 0 -6];
>> spx.io.print.sparse_signal(x)
(4,1) (7,-1) (10,-2) (13,-3) (16,7) (19,4) (22,-6)   N=22, K=7
```

Printing the non-zero entries in a signal in descending order of magnitude with location and value:

```
>> spx.io.print.sorted_sparse_signal(x)
Index:  Value
  16:   7.000000
  22:   -6.000000
  19:   4.000000
  13:   -3.000000
  10:   -2.000000
   4:   1.000000
   7:   -1.000000
```

Latex

Printing a vector in a format suitable for Latex:

```
>> spx.io.latex.printVector([1, 2, 3, 4])
\begin{pmatrix}
1 & 2 & 3 & 4
\end{pmatrix}
```

Printing a matrix in a format suitable for Latex:

```
>> spx.io.latex.printMatrix(randn(3, 4))
\begin{bmatrix}
-0.340285 & 1.13915 & 0.65748 & 0.0187744\\
-0.925848 & 0.427361 & 0.584246 & -0.425961\\
0.00532169 & 0.181032 & -1.61645 & -2.03403
\end{bmatrix}
```

Printing a vector as a set in Latex:

```
>> spx.io.latex.printSet([1, 2, 3, 4])
\{ 1 , 2 , 3 , 4 \}
```

SciRust

SciRust is a related scientific computing library developed by us. Some helper functions have been written to convert MATLAB data into SciRust compatible Rust source code.

Printing a matrix for consumption in SciRust source code:

```
>> spx.io.scirust.printMatrix(magic(3))
matrix_rw_f64(3, 3, [
        8.0, 1.0, 6.0,
        3.0, 5.0, 7.0,
        4.0, 9.0, 2.0
        ]);
```

4.2.8 Sparse recovery

Estimate for the required number of measurements for sparse signals in `N` and sparsity level `K` based on paper by Donoho and Tanner:

```
M = spx.commons.sparse.phase_transition_estimate_m(N, K);
```

Example:

```
>> spx.commons.sparse.phase_transition_estimate_m(1000, 4)
60
```

4.3 Synthetic Signals

4.3.1 Some easy to setup recovery problems

General approach:

```
m = 64;
n = 121;
k = 4;
dict = spx.dict.simple.gaussian_dict(m, n);
gen = spx.data.synthetic.SparseSignalGenerator(n, k);
% create a sparse vector
rep =  gen.biGaussian();
signal = dict*rep;
problem.dictionary = dict;
problem.representation_vector = rep;
problem.sparsity_level = k;
problem.signal_vector = signal;
```

The problems:

```
problem = spx.data.synthetic.recovery_problems.problem_small_1()
problem = spx.data.synthetic.recovery_problems.problem_large_1()
problem = spx.data.synthetic.recovery_problems.problem_barbara_blocks()
```

4.3.2 Sparse signal generation

Create generator:

```
N = 256; K = 4; S = 10;
gen  = spx.data.synthetic.SparseSignalGenerator(N, K, S);
```

Uniform signals:

```
result = gen.uniform();
result = gen.uniform(1, 2);
result = gen.uniform(-1, 1);
```

Bi-uniform signals:

```
result = gen.biUniform();
result = gen.biUniform(1, 2);
```

Gaussian signals:

```
result = gen.gaussian();
```

BiGuassian signals:

```
result = gen.biGaussian();
result = gen.biGaussian(2.0);
result = gen.biGaussian(10.0, 1.0);
```

4.3.3 Compressible signal generation

We can use `randcs` function by *Cevher, V.* for constructing compressible signals:

```
N = 100;
q = 1;
x = randcs(N, q);
plot(x);
plot(randcs(100, .9));
plot(randcs(100, .8));
plot(randcs(100, .7));
plot(randcs(100, .6));
plot(randcs(100, .5));
plot(randcs(100, .4));
lambda = 2;
x = randcs(N, q, lambda);
dist = 'logn';
x = randcs(N, q, lambda, dist);
```

4.3.4 Multi-subspace signal generation

Signals with disjoint supports:

```
% Dimension of representation space
N = 80;
% Number of subspaces
P = 8;
```

```
% Number of signals per subspace
SS = 10;
% Sparsity level of each signal (subspace dimension)
K = 4;
% Create signal generator
sg = spx.data.synthetic.MultiSubspaceSignalGenerator(N, K);
% Create disjoint supports
sg.createDisjointSupports(P);
sg.setNumSignalsPerSubspace(SS);
% Generate  signal representations
sg.biUniform(1, 4);
% Access  signal representations
X = sg.X;
% Corresponding supports
qs = sg.Supports;
```

4.4 Graphics and visualization

In this section we cover:

- Some utility classes which help in specific visualization tasks
- Some external open source libraries / functions which have been integrated in `sparse-plex` to make visualization tasks easier
- Some general techniques for specific visualization tasks

Create a full screen figure:

```
spx.graphics.figure.full_screen;
```

Multiple figures:

```
mf = spx.graphics.Figures();
mf.new_figure('fig 1');
mf.new_figure('fig 2');
mf.new_figure('fig 3');
```

All these figures will be created with same width and height. They will be placed one after another in a stacked manner.

Controlling size of multiple figures:

```
width = 1000;
height = 400;
mf = spx.graphics.Figures(width, height);
```

Display a Gram matrix for a given dictionary `Phi`:

```
spx.graphics.display.display_gram_matrix(Phi);
```

4.4.1 Canvas of a grid of images

Sometimes we wish to show a set of small images in the form of a grid. These images may be patches from a larger image or may be small independent images themselves.

`spx.graphics.canvas` helps in combining the images in the form of a grid on a canvas image.

We provide all the images to be displayed in the form of a matrix where each column consists of one image.

Creating a canvas of image patches:

```
% Let us create some random images of size 50x50
width = 50;
height = 50;
rows = 10;
cols = 10;
images = 255* rand(width*height, rows*cols);
% Let's create a canvas of these images formed into a
% 10 x 10 grid.
canvas = spx.graphics.canvas.create_image_grid(images, rows, cols, ...
    height, width);
% Let's convert the canvas to UINT8 image
canvas = uint8(canvas);
% Let's show the image
imshow(canvas);
% Let's set the proper colormap.
colormap(gray);
% Axis sizing etc.
axis image;
axis off;
```

4.4.2 Displaying a set of signals in the form of a matrix

While working on joint signal recovery problems, we need to visualize a set of signals together. They can be put together in a signal matrix where each column is one (finite dimensional) signal. It is straightforward to create a visualization for these signals:

```
num_signals = 100;
signal_size = 80;
signal_matrix = randn(signal_size, num_signals);
% Let's create a canvas and put all the signals on it.
canvas = spx.graphics.canvas.create_signal_matrix_canvas(signal_matrix);
% Let's show the image
imshow(canvas);
% Let's set the proper colormap.
colormap(gray);
% Axis sizing etc.
axis image;
axis off;
```

4.4.3 Some third party open source libraries

Put a title over all subplots:

```
spx.graphics.suptitle(title);
```

This function is by *Drea Thomas*.

RGB code for given colorname:

```
c = spx.graphics.rgb('DarkRed')
c = spx.graphics.rgb('Green')
plot(x,y,'color',spx.graphics.rgb('orange'))
```

This function is by Kristján Jónasson and is in public domain.

Supported colors:

```
%White colors
'FF','FF','FF', 'White'
'FF','FA','FA', 'Snow'
'F0','FF','F0', 'Honeydew'
'F5','FF','FA', 'MintCream'
'F0','FF','FF', 'Azure'
'F0','F8','FF', 'AliceBlue'
'F8','F8','FF', 'GhostWhite'
'F5','F5','F5', 'WhiteSmoke'
'FF','F5','EE', 'Seashell'
'F5','F5','DC', 'Beige'
'FD','F5','E6', 'OldLace'
'FF','FA','F0', 'FloralWhite'
'FF','FF','F0', 'Ivory'
'FA','EB','D7', 'AntiqueWhite'
'FA','F0','E6', 'Linen'
'FF','F0','F5', 'LavenderBlush'
'FF','E4','E1', 'MistyRose'
%Grey colors'
'80','80','80', 'Gray'
'DC','DC','DC', 'Gainsboro'
'D3','D3','D3', 'LightGray'
'C0','C0','C0', 'Silver'
'A9','A9','A9', 'DarkGray'
'69','69','69', 'DimGray'
'77','88','99', 'LightSlateGray'
'70','80','90', 'SlateGray'
'2F','4F','4F', 'DarkSlateGray'
'00','00','00', 'Black'
%Red colors
'FF','00','00', 'Red'
'FF','A0','7A', 'LightSalmon'
'FA','80','72', 'Salmon'
'E9','96','7A', 'DarkSalmon'
'F0','80','80', 'LightCoral'
'CD','5C','5C', 'IndianRed'
'DC','14','3C', 'Crimson'
'B2','22','22', 'FireBrick'
'8B','00','00', 'DarkRed'
%Pink colors
'FF','C0','CB', 'Pink'
'FF','B6','C1', 'LightPink'
'FF','69','B4', 'HotPink'
'FF','14','93', 'DeepPink'
'DB','70','93', 'PaleVioletRed'
'C7','15','85', 'MediumVioletRed'
%Orange colors
'FF','A5','00', 'Orange'
'FF','8C','00', 'DarkOrange'
'FF','7F','50', 'Coral'
'FF','63','47', 'Tomato'
'FF','45','00', 'OrangeRed'
%Yellow colors
'FF','FF','00', 'Yellow'
'FF','FF','E0', 'LightYellow'
```

```
'FF','FA','CD', 'LemonChiffon'
'FA','FA','D2', 'LightGoldenrodYellow'
'FF','EF','D5', 'PapayaWhip'
'FF','E4','B5', 'Moccasin'
'FF','DA','B9', 'PeachPuff'
'EE','E8','AA', 'PaleGoldenrod'
'F0','E6','8C', 'Khaki'
'BD','B7','6B', 'DarkKhaki'
'FF','D7','00', 'Gold'
%Brown colors
'A5','2A','2A', 'Brown'
'FF','F8','DC', 'Cornsilk'
'FF','EB','CD', 'BlanchedAlmond'
'FF','E4','C4', 'Bisque'
'FF','DE','AD', 'NavajoWhite'
'F5','DE','B3', 'Wheat'
'DE','B8','87', 'BurlyWood'
'D2','B4','8C', 'Tan'
'BC','8F','8F', 'RosyBrown'
'F4','A4','60', 'SandyBrown'
'DA','A5','20', 'Goldenrod'
'B8','86','0B', 'DarkGoldenrod'
'CD','85','3F', 'Peru'
'D2','69','1E', 'Chocolate'
'8B','45','13', 'SaddleBrown'
'A0','52','2D', 'Sienna'
'80','00','00', 'Maroon'
%Green colors
'00','80','00', 'Green'
'98','FB','98', 'PaleGreen'
'90','EE','90', 'LightGreen'
'9A','CD','32', 'YellowGreen'
'AD','FF','2F', 'GreenYellow'
'7F','FF','00', 'Chartreuse'
'7C','FC','00', 'LawnGreen'
'00','FF','00', 'Lime'
'32','CD','32', 'LimeGreen'
'00','FA','9A', 'MediumSpringGreen'
'00','FF','7F', 'SpringGreen'
'66','CD','AA', 'MediumAquamarine'
'7F','FF','D4', 'Aquamarine'
'20','B2','AA', 'LightSeaGreen'
'3C','B3','71', 'MediumSeaGreen'
'2E','8B','57', 'SeaGreen'
'8F','BC','8F', 'DarkSeaGreen'
'22','8B','22', 'ForestGreen'
'00','64','00', 'DarkGreen'
'6B','8E','23', 'OliveDrab'
'80','80','00', 'Olive'
'55','6B','2F', 'DarkOliveGreen'
'00','80','80', 'Teal'
%Blue colors
'00','00','FF', 'Blue'
'AD','D8','E6', 'LightBlue'
'B0','E0','E6', 'PowderBlue'
'AF','EE','EE', 'PaleTurquoise'
'40','E0','D0', 'Turquoise'
'48','D1','CC', 'MediumTurquoise'
```

```
'00','CE','D1', 'DarkTurquoise'
'E0','FF','FF', 'LightCyan'
'00','FF','FF', 'Cyan'
'00','FF','FF', 'Aqua'
'00','8B','8B', 'DarkCyan'
'5F','9E','A0', 'CadetBlue'
'B0','C4','DE', 'LightSteelBlue'
'46','82','B4', 'SteelBlue'
'87','CE','FA', 'LightSkyBlue'
'87','CE','EB', 'SkyBlue'
'00','BF','FF', 'DeepSkyBlue'
'1E','90','FF', 'DodgerBlue'
'64','95','ED', 'CornflowerBlue'
'41','69','E1', 'RoyalBlue'
'00','00','CD', 'MediumBlue'
'00','00','8B', 'DarkBlue'
'00','00','80', 'Navy'
'19','19','70', 'MidnightBlue'
%Purple colors
'80','00','80', 'Purple'
'E6','E6','FA', 'Lavender'
'D8','BF','D8', 'Thistle'
'DD','A0','DD', 'Plum'
'EE','82','EE', 'Violet'
'DA','70','D6', 'Orchid'
'FF','00','FF', 'Fuchsia'
'FF','00','FF', 'Magenta'
'BA','55','D3', 'MediumOrchid'
'93','70','DB', 'MediumPurple'
'99','66','CC', 'Amethyst'
'8A','2B','E2', 'BlueViolet'
'94','00','D3', 'DarkViolet'
'99','32','CC', 'DarkOrchid'
'8B','00','8B', 'DarkMagenta'
'6A','5A','CD', 'SlateBlue'
'48','3D','8B', 'DarkSlateBlue'
'7B','68','EE', 'MediumSlateBlue'
'4B','00','82', 'Indigo'
%Gray repeated with spelling grey
'80','80','80', 'Grey'
'D3','D3','D3', 'LightGrey'
'A9','A9','A9', 'DarkGrey'
'69','69','69', 'DimGrey'
'77','88','99', 'LightSlateGrey'
'70','80','90', 'SlateGrey'
'2F','4F','4F', 'DarkSlateGrey'
```

4.5 Dictionaries

4.5.1 Basic Dictionaries

Some simple dictionaries can be constructed using library functions.

The dictionaries are available in two flavors:

1. As simple matrices

2. As objects which implement the `spx.dict.Operator` abstraction defined below.

The functions returning the dictionary as a simple matrix have a suffix "mtx". The functions returning the dictionary as a `spx.dict.Operator` have the suffix "dict" at the end.

These functions can also be used to construct random **sensing matrices** which are essentially random dictionaries.

Dirac Fourier Dictionary

```
spx.dict.simple.dirac_fourier_dict(N)
```

Dirac DCT Dictionary:

```
spx.dict.simple.dirac_dct_dict(N)
```

Gaussian Dictionary:

```
spx.dict.simple.gaussian_dict(N, D, normalized_columns)
```

Rademacher Dictionary:

```
Phi = spx.dict.simple.rademacher_dict(N, D);
```

Partial Fourier Dictionary:

```
Phi = spx.dict.simple.partial_fourier_dict(N, D);
```

Over complete 1-D DCT dictionary:

```
spx.dict.simple.overcomplete1DDCT(N, D)
```

Over complete 2-D DCT dictionary:

```
spx.dict.simple.overcomplete2DDCT(N, D)
```

Dictionaries from SPIE2011 paper:

```
spx.dict.simple.spie_2011(name) % ahoc, orth, rand, sine
```

4.5.2 Sensing matrices

Gaussian sensing matrix:

```
Phi = spx.dict.simple.gaussian_mtx(M, N);
```

Rademacher sensing matrix:

```
Phi = spx.dict.simple.rademacher_mtx(M, N);
```

Partial Fourier matrix:

```
Phi = spx.dict.simple.partial_fourier_mtx(M, N);
```

4.5.3 Operators

In simple terms, a (finite) dictionary is implemented as a matrix whose columns are atoms of the dictionary. This approach is not powerful enough. A dictionary Φ usually acts on a sparse representation α to obtain a signal $x = \Phi\alpha$. During sparse recovery, the Hermitian transpose of the dictionary acts on the signal [or residual] to compute $\Phi^H x$ or $\Phi^H r$. Thus, the fundamental operations are multiplication by Φ and Φ^H. While, these operations can be directly implemented by using a matrix representation of a dictionary, they are slow and require a large storage for the dictionary. For random dictionaries, this is the only option. But for structured dictionaries and sensing matrices, the whole of dictionary need not be held in memory. The multiplication by Φ and Φ^H can be implemented using fast functions.

Also multiple dictionaries can be combined to construct a composite dictionary, e.g. $\Phi\Psi$.

In order to take care of these scenarios, we define the notion of a generic operator in an abstract class `spx.dict.Operator`. All operators support following methods.

Constructing a matrix representation of the operator:

```
op.double()
```

Computing Φx:

```
op.mtimes(x)
```

The transpose operator:

```
op.transpose()
```

By default it is constructed by computing the matrix representation of the transpose of the operator. But specialized dictionaries can implement it smartly.

The Hermitian transpose operator:

```
op.ctranspose()
```

By default it is constructed by computing the matrix representation of the Hermitian transpose of the operator. But specialized dictionaries can implement it smartly.

Obtaining specific columns from the operator:

```
op.columns(columns)
```

Note that this doesn’t require computing the complete matrix representation of the operator.

op.apply_columns(vectors, columns)

Constructing an operator which uses only the specified columns from this dictionary:

```
op.columns_operator(columns)
```

A specific column of the dictionary:

```
op.column(index)
```

Printing the contents of the dictionary:

```
disp(op)
```

4.5.4 Matrix operators

Matrix operators are constructed by wrapping a given matrix into `spx.dict.MatrixOperator` which is a subclass of `spx.dict.Operator`.

Constructing the matrix operator from a matrix `A`:

```
op = spx.dict.MatrixOperator(A)
```

The matrix operator holds references to the matrix as well as its Hermitian transpose:

```
op.A
op.AH
```

4.5.5 Composite Operators

A composite operator can be created by combining two or more operators:

```
co = spx.dict.CompositeOperator(f, g)
```

4.5.6 Unitary/Orthogonal matrices

```
spx.dict.unitary.uniform_normal_qr(n)
spx.dict.unitary.analyze_rr(O)
spx.dict.unitary.synthesize_rr(rotations, reflections)
spx.dict.unitary.givens_rot(a, b)
```

4.5.7 Dictionary Properties

```
dp = spx.dict.Properties(Dict)

dp.gram_matrix()
dp.abs_gram_matrix()
dp.frame_operator()
dp.singular_values()
dp.gram_eigen_values()
dp.lower_frame_bound()
dp.upper_frame_bound()
dp.coherence()
```

Coherence of a dictionary:

```
mu = spx.dict.coherence(dict)
```

Babel function of a dictionary:

```
mu = spx.dict.babel(dict)
```

Spark of a dictionary (for small sizes):

```
[ K, columns ] = spx.dict.spark( Phi )
```

4.5.8 Equiangular Tight Frames

```
spx.dict.etf.ss_to_etf(M)
spx.dict.etf.is_etf(F)
spx.dict.etf.ss_etf_structure(k, v)
```

4.5.9 Grassmannian Frames

```
spx.dict.grassmannian.minimum_coherence(m, n)
spx.dict.grassmannian.n_upper_bound(m)
spx.dict.grassmannian.min_coherence_max_n(ms)
spx.dict.grassmannian.max_n_for_coherence(m, mu)
spx.dict.grassmannian.alternate_projections(dict, options)
```

4.6 Vector Spaces

Our work is focused on finite dimensional vector spaces $\mathbb{R}^N$ or $\mathbb{C}^N$. We represent a vector space by a basis in the vector space. In this section, we describe several useful functions for working with one or more vector spaces (represented by one basis per vector space).

Basis for intersection of two subspaces:

```
result = spx.la.spaces.insersection_space(A, B)
```

Orthogonal complement of A in B:

```
result = spx.la.spaces.orth_complement(A, B)
```

Principal angles between subspaces spanned by A and B:

```
result = spx.la.spaces.principal_angles_cos(A, B);
result = spx.la.spaces.principal_angles_radian(A, B);
result = spx.la.spaces.principal_angles_degree(A, B);
```

Smallest principal angle between subspaces spanned by A and B:

```
result = spx.la.spaces.smallest_angle_cos(A, B);
result = spx.la.spaces.smallest_angle_rad(A, B);
result = spx.la.spaces.smallest_angle_deg(A, B);
```

Principal angle between two orthogonal bases:

```
result = spx.la.spaces.principal_angles_orth_cos(A, B)
result = spx.la.spaces.smallest_angle_orth_cos(A, B);
```

Smallest angles between subspaces:

```
result = spx.la.spaces.smallest_angles_cos(subspaces, d)
result = spx.la.spaces.smallest_angles_rad(subspaces, d)
result = spx.la.spaces.smallest_angles_deg(subspaces, d)
```

Distance between subspaces based on Grassmannian space:

```
result = spx.la.spaces.subspace_distance(A, B)
```

This is computed as the operator norm of the difference between projection matrices for two subspaces.

Check if v in range of unitary matrix U:

```
result = spx.la.spaces.is_in_range_orth(v, U)
```

Check if v in range of A:

```
result = spx.la.spaces.is_in_range(v, A)
```

A basis for matrix A:

```
result = spx.la.spaces.find_basis(A)
```

Elementary matrices product and row reduced echelon form:

```
[E, R] = spx.la.spaces.elim(A)
```

Basis for null space of A:

```
result = spx.la.spaces.null_basis(A)
```

Bases for four fundamental spaces:

```
[col_space, null_space, row_space, left_null_space]  = spx.la.spaces.four_bases(A)
[col_space, null_space, row_space, left_null_space]  = spx.la.spaces.four_orth_
↪bases(A)
```

4.6.1 Utility for constructing specific examples

Two spaces at a given angle:

```
[A, B]  = spx.data.synthetic.subspaces.two_spaces_at_angle(N, theta)
```

Three spaces at a given angle:

```
[A, B, C] = spx.la.spaces.three_spaces_at_angle(N, theta)
```

Three disjoint spaces at a given angle:

```
[A, B, C] = spx.la.spaces.three_disjoint_spaces_at_angle(N, theta)
```

Map data from k dimensions to n dimensions:

```
result = spx.la.spaces.k_dim_to_n_dim(X, n, indices)
```

Describing relations between three spaces:

```
spx.la.spaces.describe_three_spaces(A, B, C);
```

Usage:

```
d = 4;
theta = 10;
n = 20;
[A, B, C] = spx.la.spaces.three_disjoint_spaces_at_angle(deg2rad(theta), d);
spx.la.spaces.describe_three_spaces(A, B, C);
% Put them together
X = [A B C];
% Put them to bigger dimension
X = spx.la.spaces.k_dim_to_n_dim(X, n);
% Perform a random orthonormal transformation
O = orth(randn(n));
X = O * X;
```

4.7 Combinatorics

4.7.1 Steiner Systems

Steiner system with block size 2:

```
v = 10;
m = spx.discrete.steiner_system.ss_2(v);
```

Steiner system with block size 3 (STS Steiner Triple System):

```
m = spx.discrete.steiner_system.ss_3(v);
```

Bose construction for STS system for v = 6n + 3

```
m = spx.discrete.steiner_system.ss_3_bose(v);
```

Verify if a given incidence matrix is a Steiner system:

```
spx.discrete.steiner_system.is_ss(M, k)
```

Latin square construction:

```
spx.discrete.steiner_system.commutative_idempotent_latin_square(n)
```

Verify if a table is a Latin square:

```
spx.discrete.steiner_system.is_latin_square(table)
```

4.8 Matrix factorization algorithms

Note: Better implementations for these algorithms may be available in stock MATLAB distribution or other third party libraries. These codes were developed for instructional purposes as variations of these algorithms were needed in development of other algorithms in this package.

4.8.1 Various versions of QR Factorization

Gram Schmidt:

```
[Q, R] =  spx.la.qr.gram_schmidt(A)
```

Householder UR:

```
[U, R] = spx.la.qr.householder_ur(A)
```

Householder QR:

```
[Q, R] =  spx.la.qr.householder_qr(A)
```

Householder matrix for a given vector:

```
[H, v] = spx.la.qr.householder_matrix(x)
```

4.9 External Code

almost equal:

```
isalmost(a,b,tol)
```

4.9.1 Timing

```
[t, measurement_overhead, measurement_details] = timeit(f, num_outputs)
```

4.10 Noise

4.10.1 Noise generation

Gaussian noise:

```
ng = spx.data.noise.Basic(N, S);
sigma = 1;
mean = 0;
ng.gaussian(sigma, mean);
```

Creating noise at a specific SNR:

```
% Sparse signal dimension
N = 100;
% Sparsity level
K = 20;
% Number of signals
S = 4;
% Create sparse signals
signals = spx.data.synthetic.SparseSignalGenerator(N, K, S).gaussian();
% Create noise at specific SNR level.
```

```
snrDb = 10;
noises = spx.data.noise.Basic.createNoise(signals, snrDb);
% add signal to noise
signals_with_noise = signals + noises;
% Verify SNR level
20 * log10 (spx.norm.norms_l2_cw(signals) ./ spx.norm.norms_l2_cw(noises))
```

4.10.2 Noise measurement

SNR in dB:

```
result = spx.commons.snr.SNR(signals, noises)
```

SNR in dB from signal and reconstruction:

```
reconstructions = signals_with_noise;
result = spx.commons.snr.recSNRdB(signals, reconstructions)
```

Signal energy in DB

```
result = spx.commons.snr.energyDB(signals)
```

Reconstruction SNR as energy ratio:

```
result = spx.commons.snr.recSNR(signal, reconstruction)
```

Error energy normalized by signal energy:

```
result = spx.commons.snr.normalizedErrorEnergy(signal, reconstruction)
```

Reconstruction SNRs over multiple signals in dB:

```
result = spx.commons.snr.recSNRsdB(signals, reconstructions)
```

Reconstruction SNRs over multiple signals as energy ratios:

```
result = spx.commons.snr.recSNRs(signals, reconstructions)
```

Signal energies:

```
result = spx.commons.snr.energies(signals)
```

Signal energies in dB:

```
result = spx.commons.snr.energiesDB(signals)
```

clustering diclearn bht

CHAPTER 5

Exercises

The best way to learn is by doing exercises yourself. In this section, we present a set of computer exercises which help you learn the fundamentals of sparse representations: algorithms and applications.

Most of these exercises are implemented in some form or other as part of the `sparse-plex` library. Once you have written your own implementations, you may hunt the code in library and compare your implementation with the reference implementation.

The exercises are described in terms of MATLAB programming environment. But they can be easily developed in other programming environments too.

Throughout these exercises, we will develop a set of functions which are reusable for performing various tasks related to sparse representation problems. We suggest you to collect such functions developed by you in one place together so that you can implement the more sophisticated exercises easily later.

5.1 Creating a sparse signal

The first aspect is deciding the support for the sparse signal.

1. Decide on the length of signal N=1024.
2. Decide on the sparsity level K=10.
3. Choose K entries from 1..N randomly as your choice of sparse support. You can use `randperm` function.

Now, we need to consider the values of non-zero entries in the sparse vector. Typically, they are chosen from a random distribution. Few of the common choices are:

- Gaussian
- Uniform
- Bi-uniform

Gaussian

1. Generate K Gaussian random numbers with zero mean and unit standard deviation. You can use `randn` function. You may choose to change the standard deviation, but mean should usually be zero.
2. Create a column vector with N zeros.
3. On the entries indexed by the sparse support set, place the K numbers generated above.

Plotting

1. Use `stem` command to visualize the sparse signal.

Uniform

- Most of the steps are similar to creating a Gaussian sparse vector.
- The `rand` function generates a number uniformly between 0 and 1.
- In order to generate a number uniformly between a and b, we can use the simple trick of `a + (b -a) * rand`

1. Choose a and b (say -4 and 4).
2. Generate K uniformly distributed numbers between a and b.
3. Place them in the N length vector as described above.
4. Plot them.

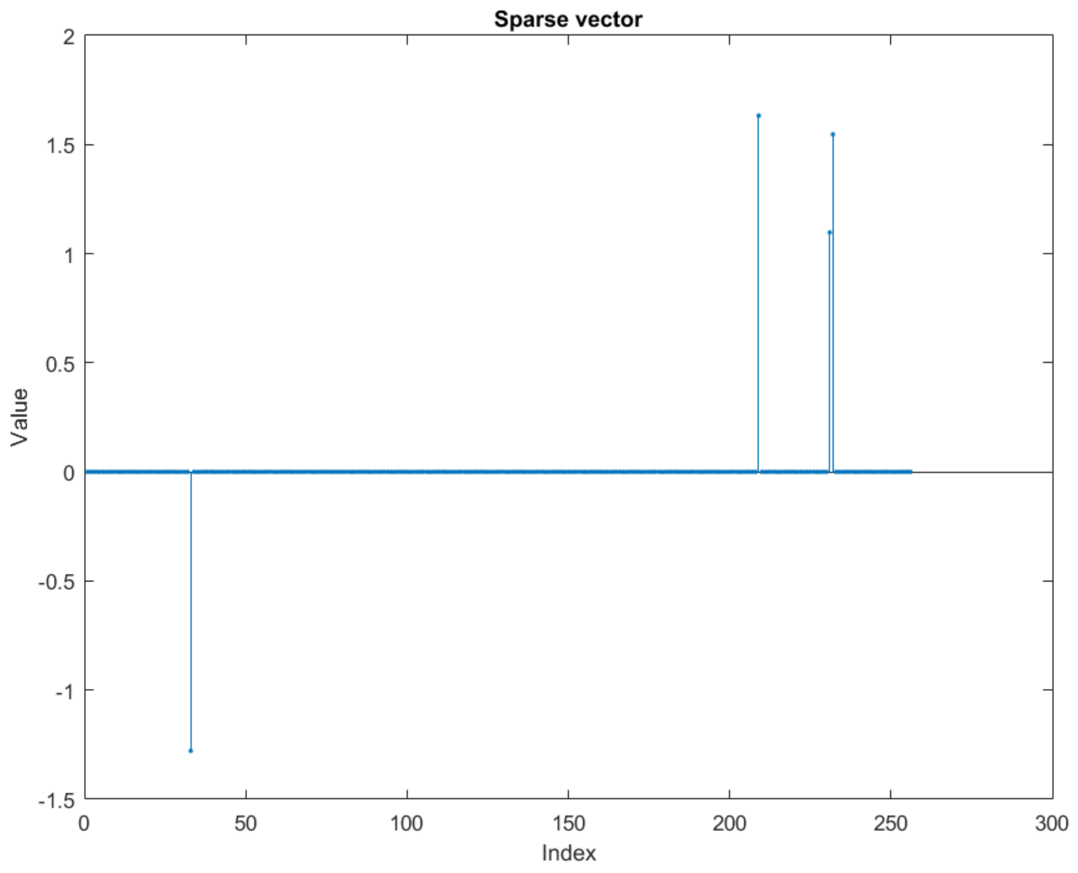
Bi-uniform

A problem with Gaussian and uniform distributions as described above is that they are prone to generate some non-zero entries which are much smaller compared to others.

Bi-uniform approach attempts to avoid this situation. It generates numbers uniformly between [-b, -a] and [a, b] where a and b are both positive numbers with a < b.

1. Choose a and by [say 1 and 2].
2. Generate K uniformly distributed random numbers between a and b (as discussed above). These are the magnitudes of the sparse non-zero entries.
3. Generate K Gaussian numbers and apply `sign` function to them to map them to 1 and -1. Note that with equal probability, the signs would be 1 or -1.
4. Multiply the signs and magnitudes to generate your sparse non-zero entries.
5. Place them in the N length vector as described above.
6. Plot them.

Following image is an example of how a sparse vector looks.



5.2 Creating a two ortho basis

Simplest example of an overcomplete dictionary is Dirac Fourier dictionary.

- You can use `eye(N)` to generate the standard basis of $\mathbb{C}^N$ which is also known as Dirac basis.
- `dftmtx(N)` gives the matrix for forward Fourier transform. Corresponding Fourier basis can be constructed by taking its transpose.
- The columns / rows of `dftmtx(N)` are not normalized. Hence, in order to construct an orthonormal basis, we need to normalize the columns too. This can be easily done by multiplying with $\frac{1}{\sqrt{N}}$.

1. Choose the dimension of the ambient signal space (say N=1024).
2. Construct the Dirac basis for $\mathbb{C}^N$.
3. Construct the orthonormal Fourier basis for $\mathbb{C}^N$.
4. Combine the two to form the two ortho basis (Dirac in left, Fourier in right).

Verification

We assume that the dictionary has been stored in a variable named `Phi`. We will use the mathematical symbol Φ for the same.

- Verify that each column has unit norm.
- Verify that each row has a norm of $\sqrt{2}$.
- Compute the Gram matrix $\Phi' * \Phi$.
- Verify that the diagonal elements are all one.
- Divide the Gram matrix into four quadrants.
- Verify that the first and fourth quadrants are identity matrices.
- Verify that the Gram matrix is symmetric.
- What can you say about the values in 2nd and 3rd quadrant?

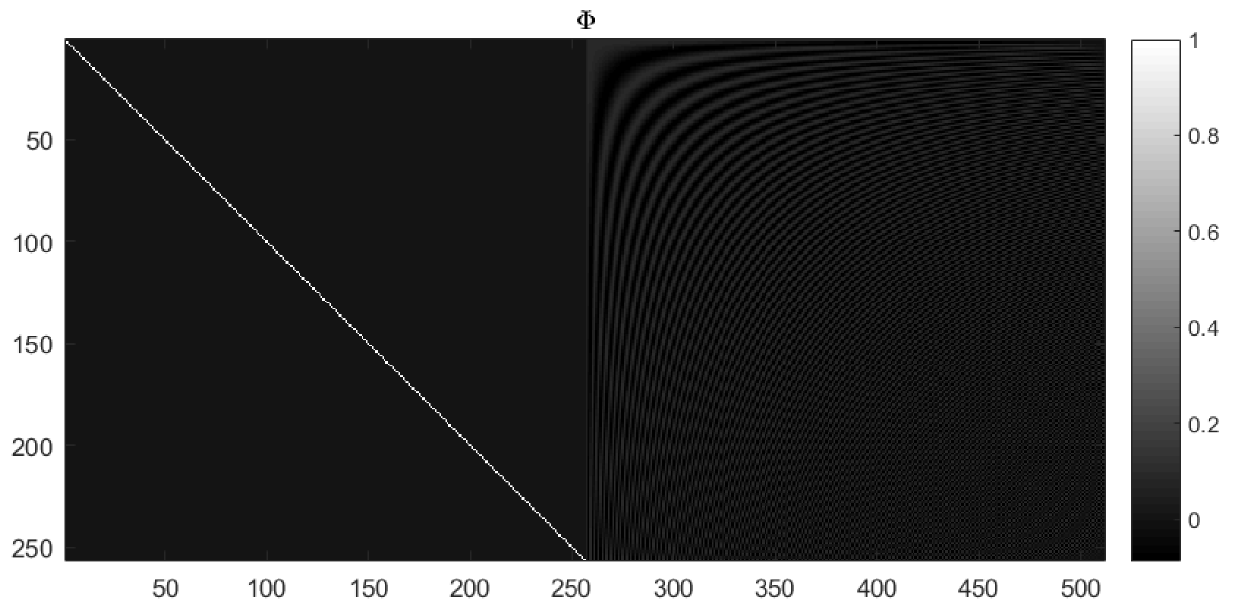
5.3 Creating a Dirac-DCT two-ortho basis

While Dirac-DFT two ortho basis has the lowest possible coherence amongst all pairs of orthogonal bases, it is not restricted to $\mathbb{R}^N$. A good starting point is to consider constructing a Dirac-DCT two ortho basis.

1. Construct the Dirac-DCT two-ortho basis dictionary.

- Replace `dftmtx(N)` by `dctmtx(N)`.
- Follow steps similar to previous exercise to construct a Dirac-DCT dictionary.
- Notice the differences in the Gram matrix of Dirac-DFT dictionary with Dirac-DCT dictionary.
- Construct the Dirac-DCT dictionary for different values of N=(8, 16, 32, 64, 128, 256).
- Look at the changes in the Gram matrix as you vary N for constructing Dirac-DCT dictionary.

An example Dirac-DCT dictionary has been illustrated in the figure below.



Note: While constructing the two-ortho bases is nice for illustration, it should be noted that using them directly for computing Φx is not efficient. This entails full cost of a matrix vector multiplication. An efficient implementation would consider following ideas:

- $\Phi x = [I \Psi] x = I x_1 + \Psi x_2$ where x_1 and x_2 are upper and lower halves of the vector x.
- $I x_1$ is nothing but *x_1*.
- Ψx_2 can be computed by using the efficient implementations of (Inverse) DFT or DCT transforms with appropriate scaling.
- Such implementations would perform the multiplication with dictionary in $O(N \log N)$ time.
- In fact, if the second basis is a wavelet basis, then the multiplication can be carried out in linear time too.
- You are suggested to take advantage of these ideas in following exercises.

Creating a signal which is a mixture of sinusoids and impulses

If we split the sparse vector x into two halves x_1 and x_2 then: * The first half corresponds to impulses from the Dirac basis. * The second half corresponds to sinusoids from DCT or DFT basis.

It is straightforward to construct a signal which is a mixture of impulses and sinusoids and has a sparse representation in Dirac-DFT or Dirac-DCT representation.

1. Pick a suitable value of N (say 64).
2. Construct the corresponding two ortho basis.
3. Choose a sparsity pattern for the vector x (of size 2N) such that some of the non-zero entries fall in first half while some in second half.
4. Choose appropriate non-zero coefficients for x.
5. Compute $y = \Phi x$ to obtain a signal which is a mixture of impulses and sinusoids.

Verification

- It is obvious that the signal is non-sparse in time domain.
- Plot the signal using `stem` function.
- Compute the DCT or DFT representation of the signal (by taking inverse transform).
- Plot the transform basis representation of the signal.
- Verify that the transform basis representation does indeed have some large spikes (corresponding to the non-zero entries in second half of x) but the rest of the representation is also full with (small) non-zero terms (corresponding to the transform representation of impulses).

5.4 Creating a random dictionary

We consider constructing a Gaussian random matrix.

1. Choose the number of measurements M say 128.
2. Choose the signal space dimension N say 1024.
3. Generate a Gaussian random matrix as $\Phi =$ randn(M, N).

Normalization

There are two ways of normalizing the random matrix to a dictionary.

One view considers that all columns or atoms of a dictionary should be of unit norm.

1. Measure the norm of each column. You may be tempted to write a for loop to do the same. While this is alright, but MATLAB is known for its vectorization capabilities. Consider using a combination of `sum` `conj` element wise multiplication and `sqrt` to come up with a function which can measure the column wise norms of a matrix. You may also explore `bsxfun`.
2. Divide each column by its norm to construct a normalized dictionary.
3. Verify that the columns of this dictionary are indeed unit norm.

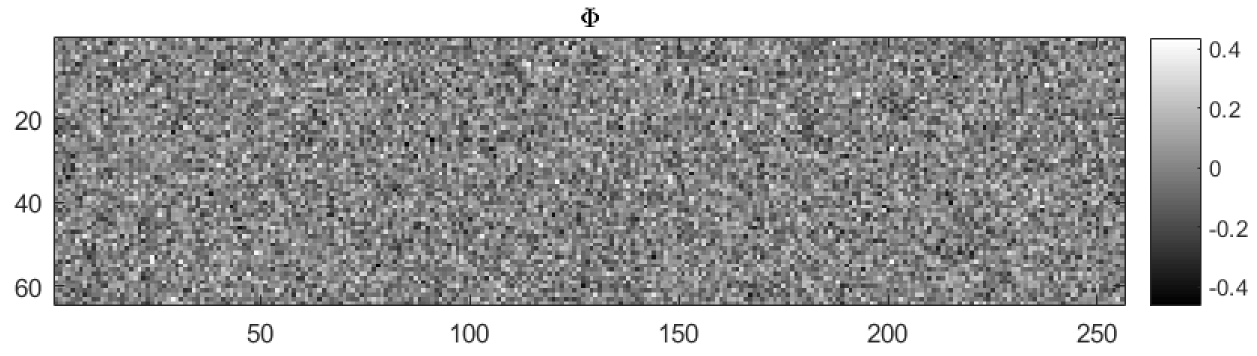
An alternative way considers a probabilistic view.

- We say that each entry in the Gaussian random matrix should be zero mean and variance $\frac{1}{M}$.
- This ensures that on an average the mean of each column is indeed 1 though actual norms of each column may differ.
- As the number of measurements increases, the likelihood of norm being close to one increases further.

We can apply these ideas as follows. Recall that `randn` generates Gaussian random variables with zero mean and unit variance.

1. Divide the whole random matrix by $\frac{1}{\sqrt{M}}$ to achieve the desired sensing matrix.
2. Measure the norm of each column.
3. Verify that the norms are indeed close to 1 (though not exactly).
4. Vary M and N to see how norms vary.
5. Use `imagesc` or `imshow` function to visualize the sensing matrix.

An example Gaussian sensing matrix is illustrated in figure below.

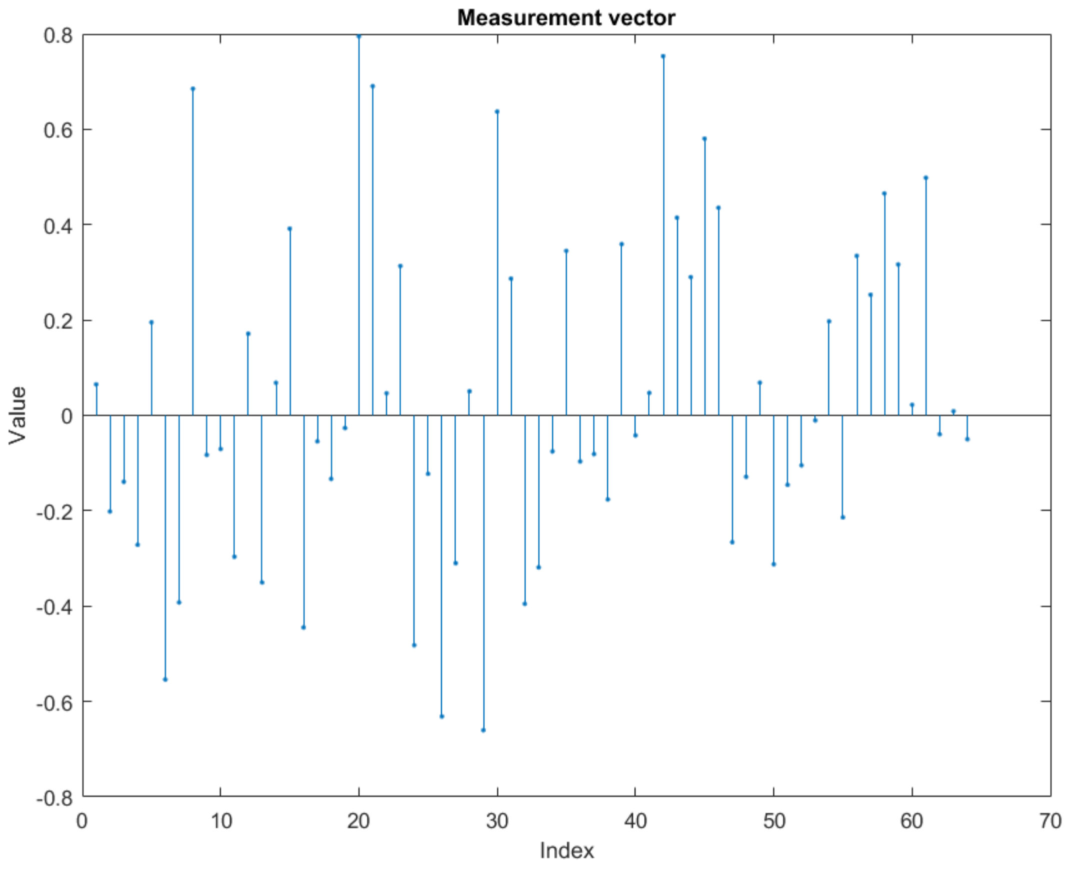


5.5 Taking compressive measurements

1. Choose a sparsity level (say K=10)
2. Choose a sparse support over $1 \dots N$ of size K randomly using `randperm` function.
3. Construct a sparse vector with bi uniform non-zero entries.

4. Apply the Gaussian sensing matrix on to the sparse signal to compute compressive measurement vector $y = \Phi x \in \mathbb{R}^M$.

An example of compressive measurement vector is shown in figure below.



In the sequel we will refer to the computation of noiseless measurement vector by the equation $y = \Phi x$.

When we make measurement noisy, the equation would be $y = \Phi x + e$.

Before we jump into sparse recovery, let us spend some time studying some simple properties of dictionaries.

5.6 Measuring dictionary properties

5.6.1 Gram matrix

You have already done this before. The straight forward calculation is $G = \Phi' * \Phi$ where we are considering the conjugate transpose of the dictionary Φ.

1. Write a function to measure the Gram matrix of any dictionary.
2. Compute the Gram matrix for all the dictionaries discussed above.
3. Verify that Gram matrix is symmetric.

For most of our purposes, the sign or phase of entries in the Gram matrix is not important. We may use the symbol `G` to refer to the Gram matrix in the sequel.

1. Compute absolute value Gram matrix `abs(G)`.

5.6.2 Coherence

Recall that the coherence of a dictionary is largest (absolute value) inner product between any pair of atoms. Actually it's quite easy to read the coherence from the absolute value Gram matrix.

- We reject the diagonal elements since they correspond to the inner product of an atom with itself. For a properly normalized dictionary, they should be 1 anyway.
- Since the matrix is symmetric we need to look at only the upper triangular half or the lower triangular half (excluding the diagonal) to read off the coherence.
- Pick the largest value in the upper triangular half.

1. Write a MATLAB function to compute the coherence.
2. Compute coherence of a Dirac-DFT dictionary for different values of N. Plot the same to see how coherence decreases with N.
3. Do the same for Dirac-DCT.
4. Compute the coherence of Gaussian dictionary (with say N=1024) for different values of M and plot it.
5. In the case of Gaussian dictionary, it is better to take average coherence for same M and N over different instances of Gaussian dictionary of the specified size.

5.6.3 Babel function

Babel function is quite interesting. While the definition looks pretty scary, it turns out that it can be computed very easily from the Gram matrix.

1. Compute the (absolute value) Gram matrix for a dictionary.
2. Sort the rows of the Gram matrix (each row separately) in descending order.
3. Remove the first column (consists of all ones in for a normalized dictionary).
4. Construct a new matrix by accumulating over the columns of the sorted Gram matrix above. In other words, in the new matrix
   - First column is as it is.
   - Second column consists of sum of first and second column of sorted matrix.
   - Third column consists of sum of first to third column of sorted matrix .
   - Continue accumulating like this.
5. Compute the maximum for each column.
6. Your Babel function is in front of you.
7. Write a MATLAB function to carry out the same for any dictionary.
8. Compute the Babel function for Dirac-DFT and Dirac-DCT dictionary with (N=256).
9. Compute the Babel function for Gaussian dictionary with N=256. Actually compute Babel functions for many instances of Gaussian dictionary and then compute the average Babel function.

5.7 Getting started with sparse recovery

Our first objective will be to develop algorithms for sparse recovery in noiseless case.

The defining equation is $y = \Phi x$ where x is the sparse representation vector, Φ is the dictionary or sensing matrix and y is the signal or measurement vector. In any sparse recovery algorithm, following quantities are of core interest:

- x which is unknown to us.
- Φ which is known to us. Sometimes we may know Φ only approximately.
- y which is known to us.
- Given Φ and y, we estimate an approximation of x which we will represent as $\widehat{x}$.
- $\widehat{x}$ is (typically) sparse even if x may be only approximately sparse or compressible.
- Given an estimate $\widehat{x}$, we compute the residual $r = y - \Phi\widehat{x}$. This quantity is computed during the sparse recovery process.
- Measurement or signal error norm $\|r\|_2$. We strive to reduce this as much as possible.
- Sparsity level K. We try to come up with an $\widehat{x}$ which is K-sparse.
- Representation error or recovery error $f = x - \widehat{x}$. This is unknown to us. The recovery process tends to minimize its norm $\|f\|_2$ (if it is working correctly !).

Some notes are in order

- K may or may not be given to us. If K is given to us, we should use it in our recovery process. If it is not given, then we should work with $\|r\|_2$.
- While the recovery algorithm itself doesn't know about x and hence cannot calculate f, a controlled testing environment can carefully choose and x, compute y and pass Φ and y to the recovery algorithm. Thus, the testing environment can easily compute f by using the x known to it and $\widehat{x}$ given by the recovery algorithm.

Usually the sparse recovery algorithms are iterative. In each iteration, we improve our approximation $\widehat{x}$ and reduce $\|r\|_2$.

- We can denote the iteration counter by k starting from 0 onwards.
- We denote k-th approximation by $\widehat{x}^k$ and k-th residual by r^k.
- A typical initial estimate is given by $\widehat{x}^0 = 0$ and thus, $r^0 = y$.

Objectives of recovery algorithm

There are fundamentally two objectives of a sparse recovery algorithm

- Identification of locations at which $\widehat{x}$ has non-zero entries. This corresponds to the sparse support of x.
- Estimation of the values of non-zero entries in $\widehat{x}$.

We will use following notation.

- The identified support will be denoted as Λ. It is the responsibility of the sparse recovery algorithm to guess it.
- If the support is identified gradually in each iteration, we can use the notation Λ^k.
- The actual support of x will be denoted by Ω. Since x is unknown to us hence Ω is also unknown to us within the sparse recovery algorithm. However, the controlled testing environment would know about Ω.

If the support has been identified correctly, then estimation part is quite easy. It's nothing but the application of least squares over the columns of Φ selected by the support set.

Different recovery algorithms vary in how they approach the support identification and coefficient estimations.

- Some algorithms try to identify whole support at once and then estimate the values of non-zero entries.
- Some algorithms identify atoms in the support one at a time and iteratively estimate the non-zero values for the current support.

Simple support identification

- Write a function which sorts a given vector by the decreasing order of magnitudes of its entries.
- Identify the K largest (magnitude) entries in the sorted vector and their locations in the original vector.
- Collect the locations of K largest entries into a set

Note: `[sorted_x, index_vector] = sort(x)` in MATLAB returns both the sorted entries and the index vector such that `sorted_x = x[index_vector]`. Our interest is usually in the `index_vector` as we don't want to really change the order of entries in `x` while identifying the largest K entries.

In MATLAB a set can be represented using an array. You have to be careful to ensure that such a set never have any duplicate elements.

Sparse approximation of a given vector

Given a vector x which may not be sparse, its K sparse approximation which is the best approximation in l_p norm sense can be obtained by choosing the K largest (in magnitude) entries.

1. Write a MATLAB function to compute the K sparse representation of

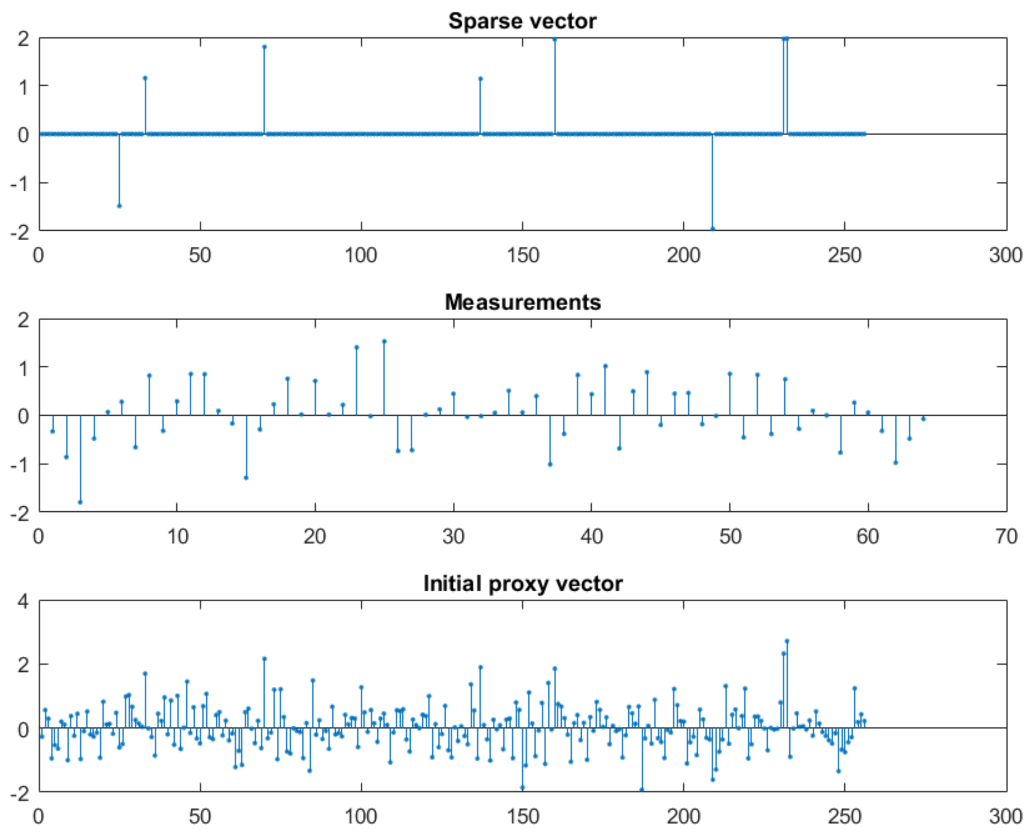
   any vector.

   - Identify the K largest entries and put their locations in the support set Λ.
   - Compute $\Lambda^c = \{1 \dots N\} \setminus \Lambda$.
   - Set the entries corresponding to Λ^c in x to zero.

The proxy vector

A very interesting quantity which appears in many sparse recovery algorithms is the proxy vector $p = \Phi' r$.

The figure below shows a sparse vector, its measurements and corresponding proxy vector $p^0 = \Phi r^0 = \Phi y$.



While the proxy vector may look quite chaotic on first look, it is very interesting to note that it tends to have large entries at exactly the same location as the sparse vector x itself.

if we think about the proxy vector closely, we can notice that each entry in the proxy is the inner product of an atom in Φ with the residual r. Thus, each entry in proxy vector indicates how similar an atom in the dictionary is with the residual.

1. Choose M, N and K and construct a sparse vector x with support Ω and Gaussian dictionary Φ.
2. For the measurement vector $y = \Phi x$, compute $p = \Phi' y$.
3. Identify the K largest entries in p and use their locations to make a guess of support as Λ.
4. Compare the sets Ω and Λ. Measure the support identification ratio as $\frac{|\Lambda \cap \Omega|}{|\Omega|}$ i.e. the ratio of the number of indices common in Λ and Ω with the number of indices in Ω (which is K).
5. Keep M and N fixed and vary K to see how support identification ratio changes. For this, measure average support identification ratio for say 100 trials. You may increase the number of trials if you want.
6. Keep K=4, N=1024 and vary M from 10 to 500 to see how support identification ratio changes. Again use the average value.

Note: The support identification ratio is a critical tool for evaluating the quality of a sparse recovery algorithm. Recall that if the support has been identified correctly, then reconstructing a sparse vector is a simple least squares problem. If the support is identified partially, or some of the indices are incorrect, then it can lead to large recovery errors.

If the support identification ratio is 1, then we have correctly identified the support. Otherwise, we haven't.

For noiseless recovery, if support is identified correctly, then representation will be recovered correctly (unless Φ is ill conditioned). Thus, support identification ratio is a good measure of success or failure of recovery. We don't need to worry about SNR or norm of recovery error.

In the sequel, for noiseless recovery, we will say that recovery succeeds if support identification ratio is 1.

If we run multiple trials of a recovery algorithm (for a specific configuration of K, M, N etc.) with different data, then the **recovery rate** would be the number of trials in which successful recovery happened divided by the total number of trials.

The recovery rate (on reasonably high number of trials) would be our main tool for measuring the quality of a recovery algorithm. Note that the recovery rate depends on

- The representation space dimension N.
- The number of measurements M.
- The sparsity level K.
- The choice of dictionary Φ.

It doesn't really depend much on the choice of distribution for the non-zero entries in x if the entries are i.i.d. Or the dependence as such is not very significant.

5.8 Developing the hard thresholding algorithm

Based on the idea of the proxy vector, we can easily compute a sparse approximation as follows.

1. Identify the K largest entries in the proxy and their locations.
2. Put the locations together in your guess for the support Λ.
3. Identify the columns of Φ corresponding to Λ and construct a submatrix Φ_{Λ}.
4. Compute $x_{\Lambda} = \Phi_{\Lambda}^{\dagger} y$ as the least squares solution of the problem $y = \Phi_{\Lambda} x_{\Lambda}$.
5. Set the remaining entries in x corresponding to Λ^c as zeros.

Put together the algorithm described above in a MATLAB function like `x_hat = hard_thresholding(Phi, y, K)`.

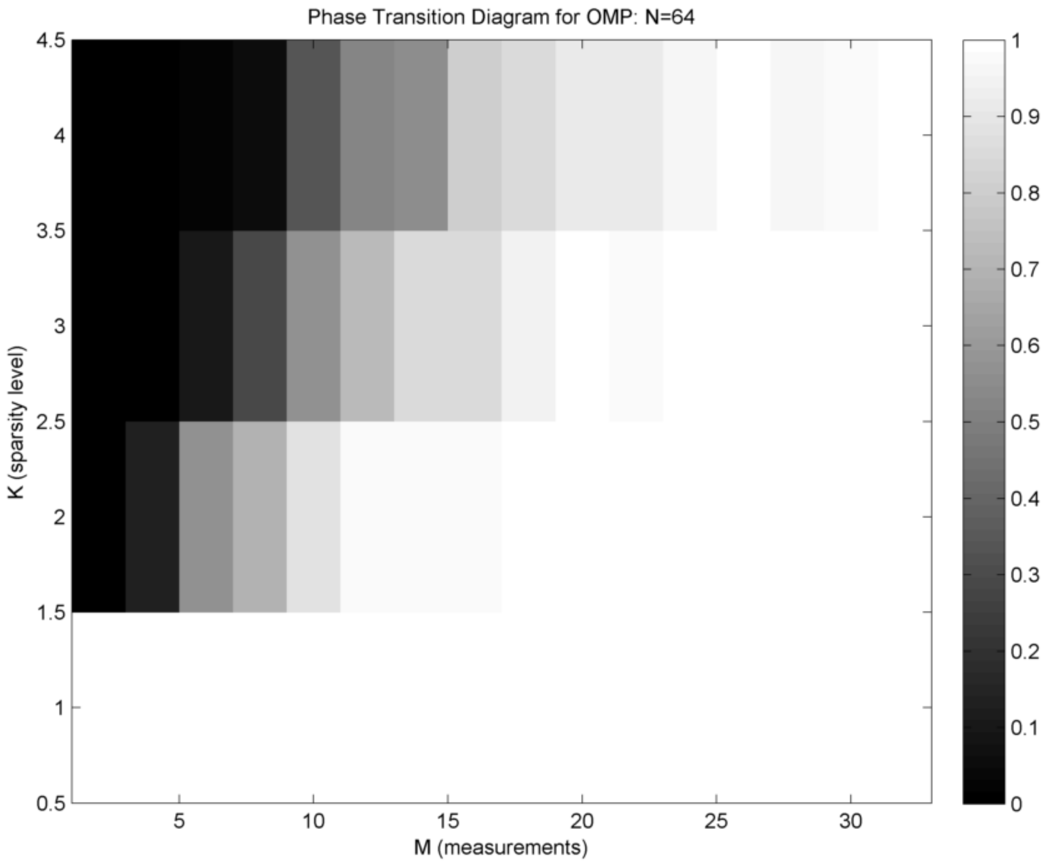
1. Think and explain why hard thresholding will always succeed if $K = 1$.
2. Say $N = 256$ and $K = 2$. What is the required number of measurements at which the recovery rate will be equal to 1.

Phase transition diagram

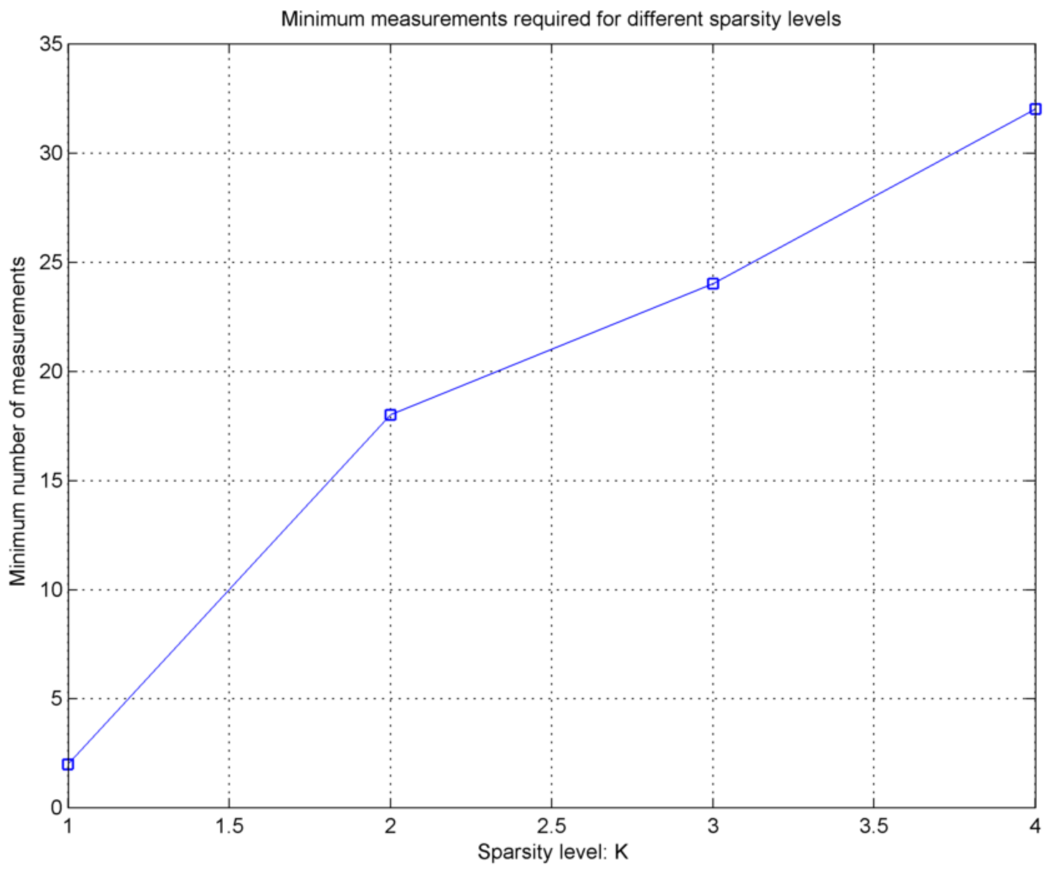
A nice visualization of the performance of a recovery algorithm is via its phase transition diagram. The figure below shows the phase transition diagram for orthogonal matching pursuit algorithm with a Gaussian dictionary and Gaussian sparse vectors.

- N is fixed at 64.
- K is varied from 1 to 4.
- M is varied from 1 and 2 to 32 (N/2) with steps of 2.
- For each configuration of K and M, 1000 trials are conducted and recovery rate is measured.

- In the phase transition diagram, a white cell indicates that for the corresponding K and M, the algorithm is able to recover successfully always.
- A black cell indicates that the algorithm never successfully recovers any signal for the corresponding K and M.
- A gray cell indicates that the algorithm sometimes recovers successfully while sometimes it may fail.
- Safe zone of operation is the white area in the diagram.



In the figure below, we capture the minimum required number of measurements for different values of K for OMP algorithm running on Gaussian sensing matrix.



It is evident that as K increases, the minimum M required for successful recovery also increases.

1. Generate the phase transition diagram for thresholding algorithm with N = 256, K varying from 1 to 16 and M varying from 2 to 128 and a minimum of 100 trials for each configuration.
2. Use the phase transition diagram data for estimating the minimum M for different values of K and plot it.

5.9 Developing the matching pursuit algorithm

You can read the description of matching pursuit algorithms on Wikipedia. This is a simpler algorithm than orthogonal matching pursuit. It doesn't involve any least squares step.

1. Implement the matching pursuit (MP) algorithm in MATLAB.
2. Generate the phase transition diagram for MP algorithm with N = 256, K varying from 1 to 16 and M varying from 2 to 128 and a minimum of 100 trials for each configuration.
3. Use the phase transition diagram data for estimating the minimum M for different values of K and plot it.

5.10 Developing the orthogonal matching pursuit algorithm

The orthogonal matching pursuit algorithm is described in the figure below.

```
$x, r, \Lambda = \text{OMP}(\Phi, y)$;
$x^0 \leftarrow 0$;
$r^0 \leftarrow y$ ;                                    // $r = y - \Phi x$
$\Lambda^0 = \varnothing$ ;                             // Index set of chosen atoms
$k \leftarrow 0$ ;                                      // Iteration counter
repeat
    $h^{k+1} \leftarrow \Phi^T r^k$ ;                   // Match
    $\lambda^{k+1} = \arg\max_{j \notin \Lambda^k} |h_j^{k+1}|$;      // Identify
    $\Lambda^{k+1} \leftarrow \Lambda^k \cup \{\lambda^{k+1}\}$ ;     // Update support
    $x^{k+1} \leftarrow 0$ ;
    $x^{k+1}_{\Lambda^{k+1}} \leftarrow \Phi^{\dagger}_{\Lambda^{k+1}} y$ ;   // Update representation LS
    $y^{k+1} = \Phi x^{k+1}$ ;                          // Update approximation
    $r^{k+1} \leftarrow y - y^{k+1}$ ;                  // Update residual
    $k \leftarrow k + 1$;                               // Update iteration counter
until halting criteria is satisfied;
$x \leftarrow x^k$ ; $\Lambda \leftarrow \Lambda^k$ ; $r \leftarrow r^k$ ;
```

1. Implement the orthogonal matching pursuit (OMP) algorithm in MATLAB.
2. Generate the phase transition diagram for OMP algorithm with N = 256, K varying from 1 to 16 and M varying from 2 to 128 and a minimum of 100 trials for each configuration.
3. Use the phase transition diagram data for estimating the minimum M for different values of K and plot it.

5.11 Sparsifying an image

CHAPTER 6

Scripts

6.1 Preamble

```
close all; clear all; clc;
```

Resetting random numbers:

```
rng('default');
```

Export management flag:

```
export = true;
```

6.2 Figures

Exporting figures:

```
if export
export_fig images\figure_name.png -r120 -nocrop;
export_fig images\figure_name.pdf;
end
```

Typical steps in figures:

```
xlabel('Principal angle (degrees)');
ylabel('Number of subspace pairs');
title('Distribution of principal angles over subspace pairs in signal space');
grid on;
```

CHAPTER 7

Indices and tables

- genindex
- modindex
- search