## CONTENTS:

1 Introduction 
   1.1 Functional Programming ................................................................. 3
   1.2 Linear Operators .............................................................................. 4
   1.3 Greedy Sparse Recovery/Approximation Algorithms ........................... 4
   1.4 Convex Optimization based Recovery Algorithms ............................... 4
   1.5 Evaluation Framework ....................................................................... 4
   1.6 Installation ......................................................................................... 5
   1.7 Further Reading ............................................................................... 5

2 Tutorials 
   2.1 Dirac DCT Dictionaries ..................................................................... 7
   2.2 Alternating direction algorithms for 11 problems in compressive sensing . 16

3 Getting Started 
   3.1 Installing as a package ..................................................................... 31
   3.2 Working with the source code in development mode ............................ 31
   3.3 Examples .......................................................................................... 31

4 CR.Sparse API Documentation 
   4.1 Sparsifying Dictionaries and Sensing Matrices .................................. 33
   4.2 Linear Operators .............................................................................. 37
   4.3 Greedy Sparse Recovery/Approximation Algorithms ........................ 44
   4.4 Convex Optimization based Sparse Recovery/Approximation Algorithms 53
   4.5 Sample Data Generation Utilities ....................................................... 55
   4.6 Utilities in cr.sparse module .............................................................. 55
   4.7 Evaluation Framework ...................................................................... 62
   4.8 Linear Algebra Subroutines ................................................................. 62

5 Benchmarks 
   5.1 Computation Time Comparison of Sparse Recovery Methods ............... 65
   5.2 Orthogonal Matching Pursuit .............................................................. 69

6 Indices and tables 

Python Module Index 

Index
GITHUB Repo: cr-sparse
This library aims to provide XLA/JAX based Python implementations for various algorithms related to:

- Sparse approximation
- Compressive sensing
- Linear operators

Bulk of this library is built using functional programming techniques which is critical for the generation of efficient numerical codes for CPU and GPU architectures.

1.1 Functional Programming

Functional Programming is a programming paradigm where computer programs are constructed by applying and composing functions. Functions define a tree of expressions which map values to other values (akin to mathematical functions) rather than a sequence of iterative statements. Some famous languages based on functional programming are Haskell and Common Lisp. A key idea in functional programming is a pure function. A pure function has following properties:

- The return values are identical for identical arguments.
- The function has no side-effects (no mutation of local static variables, non-local variables, etc.).

XLA is a domain-specific compiler for linear algebra. XLA uses JIT (just-in-time) compilation techniques to analyze the structure of a numerical algorithm written using it. It then specializes the algorithm for actual runtime dimensions and types of parameters involved, fuses multiple operations together and emits efficient native machine code for devices like CPUs, GPUs and custom accelerators (like Google TPUs).

JAX is a front-end for XLA and Autograd with a NumPy inspired API. Unlike NumPy, JAX arrays are always immutable. While \( x[0] = 10 \) is perfectly fine in NumPy as arrays are mutable, the equivalent functional code in JAX is
\[
\texttt{x = x.at[0].set(10)}.
\]
1.2 Linear Operators

Efficient linear operator implementations provide much faster computations compared to direct matrix vector multiplication. A framework for building and composing linear operators has been provided in cr.sparse.lop. Functionality includes:

- Basic operators: identity, matrix, diagonal, zero, flipud, sum, pad_zeros, symmetrize, restriction, etc.
- Signal processing: fourier_basis_1d, dirac_fourier_basis_1d, etc.
- Random dictionaries: gaussian_dict, rademacher_dict, random_onb_dict, random_orthonormal_rows_dict, etc.
- Operator algebra: neg, scale, add, subtract, compose, transpose, hermitian, hcat, etc.
- Additional utilities

1.3 Greedy Sparse Recovery/Approximation Algorithms

JAX based implementations for the following algorithms are included.

- Orthogonal Matching Pursuit
- Compressive Sampling Matching Pursuit
- Subspace Pursuit
- Iterative Hard Thresholding
- Hard Thresholding Pursuit

1.4 Convex Optimization based Recovery Algorithms

A port of yall1 (Your algorithms for $\ell_1$) has been provided. It provides alternating directions method of multipliers based solutions for basis pursuit, basis pursuit denoising, basis pursuit with inequality constraints, their non-negative counterparts and other variants.

1.5 Evaluation Framework

The library also provides
- Various simple dictionaries and sensing matrices
- Sample data generation utilities
- Framework for evaluation of sparse recovery algorithms
1.6 Installation

Basic installation from PYPI:

```
python -m pip install cr-sparse
```

Installation from GitHub:

```
python -m pip install git+https://github.com/carnotresearch/cr-sparse.git
```

1.7 Further Reading

- Functional programming
- How to Think in JAX
- JAX - The Sharp Bits
2.1 Dirac DCT Dictionaries

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
  - Orthogonal Matching Pursuit
  - Measure the recovery error for different sparse recovery algorithms.

```python
# Make sure that cr-sparse package is available on colab
!python -m pip install --quiet git+https://github.com/carnotresearch/cr-sparse.git
```

We use JAX for numerical computations. Importing JAX dependencies:

```python
import jax
import jax.numpy as jnp
```

Importing relevant modules from the CR-Sparse library:

```python
import cr.sparse as crs
import cr.sparse.dict as dict
import cr.sparse.pursuit as pursuit
```

We will use matplotlib for all visualizations in this notebook.

```python
import matplotlib as mpl
import matplotlib.pyplot as plt
```

```python
%matplotlib inline
```

We will be working with signals in the space $\mathbb{R}^M$. The dimension of signal space:
The standard basis or identity basis or Dirac basis for the signal space $I \in \mathbb{R}^{M \times M}$ can be easily constructed using JAX:

```python
I = jnp.eye(M)
fig, plt = plt.figure(figsize=(8,6), dpi= 100, facecolor='w', edgecolor='k')
plt.imshow(I, extent=[0, 1, 0, 1])
plt.gray()
plt.colorbar()
plt.title(r'Dirac basis');
```
2.1.1 DCT basis construction

```python
Psi = dict.dct_basis(M)
Psi.shape
```

WARNING: No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and...rerun for more info.)

```
(256, 256)
```

Visualizing the DCT basis

```python
fig = plt.figure(figsize=(8, 6), dpi=100, facecolor='w', edgecolor='k')
plt.imshow(Psi, extent=[0, 1, 0, 1])
plt.gray()
plt.colorbar()
plt.title(r'$\Psi$');
```
2.1.2 Dirac-DCT dictionary construction

A Dirac-DCT dictionary is a concatenation of the two orthonormal bases (the Dirac basis and the DCT basis). A simple way to do the same using Jax is:

```
[38]: jnp.hstack([I, Psi]);
```

CR-Sparse provides a ready-made method for constructing a Dirac-DCT dictionary for \( \mathbb{R}^M \):

```
[11]: Phi = dict.dirac_dct_basis(M)
```

As expected, the number of atoms in this dictionary is twice of \( M \):

```
[12]: Phi.shape
[12]: (256, 512)
[39]: N = Phi.shape[1]
[39]: 512
```

We can say that the dictionary \( \Phi \) maps representations in \( \mathbb{R}^N \) to signals in \( \mathbb{R}^M \). \( \mathbb{R}^N \) is called the representation space and \( \mathbb{R}^M \) is called the signal space.

Visualizing the Dirac-DCT dictionary:

```
[14]: fig=plt.figure(figsize=(8,6), dpi= 100, facecolor='w', edgecolor='k')
plt.imshow(Phi, extent=[0, 2, 0, 1])
plt.gray()
plt.colorbar()
plt.title(r'$\Phi$');
```
2.1.3 A mixture signal of impulses and sinusoids

We can combine some of the atoms from the Dirac basis and some of the atoms from the DCT basis, alternatively, some atoms from Dirac-DCT dictionary to form a signal which is a mixture of impulses and sinusoids. The atoms from Dirac basis contribute the impulses and the atoms from the DCT basis contribute the sinusoids.

```
[40]: x = crs.build_signal_from_indices_and_values(N, [20, 30, 100, M+3, M+57], [1, -0.4, 0.6, 1.2, -0.8])
y = Phi @ x
```

Visualizing the mixture signal

```
[41]: fig=plt.figure(figsize=(8,6), dpi= 100, facecolor='w', edgecolor='k')
plt.stem(y, markerfmt='.', use_line_collection=True);
```
Notice the three impulses. The periodic component is coming from the two sinusoids.

\( x \) in the equation \( y = \Phi x \) above is the sparse representation of \( y \) in the dictionary \( \Phi \). It is called a sparse representation as it consists of very few non-zero values corresponding to the atoms which contribute in the construction of \( y \). CR-Sparse provides some convenient functions to extract the non-zero indices and non-zero values of a sparse representation.

The support \( I \) of the representation \( x \):

\[
\begin{align*}
\text{crs.nonzero_indices}(x) & \\
\text{DeviceArray([ 20, 30, 100, 259, 313], dtype=int32)}
\end{align*}
\]

The non-zero entries in \( x \) over \( I \) a.k.a. \( x_I \):

\[
\begin{align*}
\text{crs.nonzero_values}(x) & \\
\text{DeviceArray([ 1. , -0.4, 0.6, 1.2, -0.8], dtype=float32)}
\end{align*}
\]

We remind that we can write \( y = \Phi x \) as \( y = \Phi_I x_I \) too as atoms with zero contribution in \( y \) can be removed from the computation.
2.1.4 Mixture signal representation in DCT basis

The representation of a signal $y$ in an orthogonal basis $\Psi$ is constructed by computing $\Psi^T y$. Let’s see how does our $y$ fare in the DCT basis:

$$y_{dct} = \Psi^T y$$

We can see that there are two impulses in the DCT basis representation of $y$ which correspond to the sinusoids present in $y$. However, it also consists of a mix of sinusoids which are the representation of the impulses in $y$ in the DCT basis. Thus, the representation of $y$ in the DCT basis $\Psi$ is not sparse. In fact, $y$ doesn’t have a sparse representation in any orthonormal basis for $\mathbb{R}^M$. 

```
[44]: y_dct = Psi.T @ y
[45]: fig = plt.figure(figsize=(8,6), dpi=100, facecolor='w', edgecolor='k')
       plt.stem(y_dct, markerfmt='.', use_line_collection=True);
```
2.1.5 Building a sparse representation in the Dirac DCT basis

Constructing the representation of \( y \) in a dictionary is a non-trivial problem as it is an overcomplete set of atoms. There are infinite possible representations of \( y \) in \( \Phi \). However, under suitable conditions, the sparse representation of \( y \) in \( \Phi \) is unique. Also, under suitable conditions, there exist algorithms which can recover the sparse representation of \( y \) in \( \Phi \).

One class of algorithms is called greedy pursuit algorithms. A representative in this class is Orthogonal Matching Pursuit (OMP). In order to run this algorithm, it is imperative that we are aware of the number of non-zero entries in the sparse representation (a.k.a. the sparsity of the representation of \( y \) in \( \Phi \)). Fortunately, in this example, we know this trivially:

\[
K = \text{crs.nonzero_indices(x).size}
\]

CR-Sparse provides a ready-made implementation of OMP. Let’s import the module which provides this functionality.

```python
import cr.sparse.pursuit.omp
```

Running the OMP sparse recovery algorithm:

\[
solution = \text{pursuit.omp.solve(Phi, y, K)}
\]

In order to save space, the algorithm returns the nonzero values and their indices in the sparse representation:

The values at non-zero entries in the representation \( x_I \):

```python
solution.x_I
```

DeviceArray(
[  1.1999999 ,  1. , -0.7999999 ,  0.59999996, -0.39999998],
dtype=float32)

The indices of non-zero entries \( I \):

```python
solution.I
```

DeviceArray([259, 20, 313, 100, 30], dtype=int32)

Note that the indices as identified by OMP may not be in the same order as earlier. The order of indices doesn’t matter.

CR-Sparse provides a convenient function to combine the non-zero indices and values to form the overall sparse representation in \( \mathbb{R}^N \).

```python
z = crs.build_signal_from_indices_and_values(N, solution.I, solution.x_I)
```

Visualizing the representation:

```python
fig=plt.figure(figsize=(8,6), dpi= 100, facecolor='w', edgecolor='k')
plt.stem(z, markerfmt='.', use_line_collection=True);
```
2.1.6 Sparse representation error

In general, OMP may be fed a signal $y = \Phi x + e$ where $e$ is the sparse representation error. When OMP estimates $x$ is actually computes an approximation $\hat{x}$. It also estimates a residual $r = y - \Phi \hat{x}$.

The energy of the residual is given by:

[27]: solution.r_norm_sqr

[27]: DeviceArray(3.802608e-14, dtype=float32)

In this case, the energy of residual is very small as there was no representation error in the original computation $y = \Phi x$. This residual energy corresponds to floating point computation errors.

Visualizing the residual:

[32]: fig=plt.figure(figsize=(8,6), dpi= 100, facecolor='w', edgecolor='k')
plt.stem(solution.r, markerfmt='.', use_line_collection=True);
2.2 Alternating direction algorithms for l1 problems in compressive sensing

We provide a port of YALL1 basic package. This is built on top of JAX and can be used to solve the following $\ell_1$ minimization problems.

The basis pursuit problem

$$\min_x \|Wx\|_{w,1} \text{ s.t. } Ax = b$$

The L1/L2 minimization or basis pursuit denoising problem

$$\min_x \|Wx\|_{w,1} + \frac{1}{2\rho} \|Ax - b\|_2^2$$

The L1 minimization problem with L2 constraints

$$\min_x \|Wx\|_{w,1} \text{ s.t. } \|Ax - b\|_2 \leq \delta$$

We also support corresponding non-negative counter-parts.

The nonnegative basis pursuit problem

$$\min_x \|Wx\|_{w,1} \text{ s.t. } Ax = b \text{ and } x \succeq 0$$
The nonnegative L1/L2 minimization or basis pursuit denoising problem

$$\min_x \|Wx\|_{w,1} + \frac{1}{2\rho} \|Ax - b\|_2^2 \text{ s.t. } x \succeq 0$$

The nonnegative L1 minimization problem with L2 constraints

$$\min_x \|Wx\|_{w,1} \text{ s.t. } \|Ax - b\|_2 \leq \delta \text{ and } x \succeq 0$$

In the above, $W$ is a sparsifying basis s.t. $Wx = \alpha$ is a sparse representation of $x$ in $W$ given by $\alpha = W^Tx$. For simple examples, we can assume $W = I$ is the identity basis.

The $\| \cdot \|_{w,1}$ is the weighted L1 (semi-) norm defined as

$$\|x\|_{w,1} = \sum_{i=1}^{n} w_i |x_i|$$

for a given non-negative weight vector $w$. In the simplest case, we assume $w = 1$ reducing it to the famous $\ell_1$ norm.

Import relevant libraries

[1]:
```
from jax.config import config
config.update("jax_enable_x64", True)
import jax
import jax.numpy as jnp
import numpy as np
from jax import random
from jax import jit, grad, vmap
norm = jnp.linalg.norm
```

[2]:
```
import cr.sparse as crs
import cr.sparse.dict as crdict
import cr.sparse.data as crdata
import cr.sparse.lop as lop
from cr.sparse.cvx.adm import yall1
```

[3]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt
%matplotlib inline
```

Setup a problem with a random sensing matrix with orthonormal rows

[4]:
```
N = 1000
M = 300
K = 50
```

[5]:
```
key = random.PRNGKey(0)
key1, key2, key3, key4 = random.split(key, 4)
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and...
...rerun for more info.)
```

[6]:
```
A = crdict.random_orthonormal_rows(key1, M, N)
```

2.2. Alternating direction algorithms for l1 problems in compressive sensing
CR.Sparse

[7]: crs.has_orthogonal_rows(A)

[7]: DeviceArray(True, dtype=bool)

[8]: fig=plt.figure(figsize=(8,6), dpi=100, facecolor='w', edgecolor='k')
    plt.imshow(A, extent=[0, 2, 0, 1])
    plt.gray()
    plt.colorbar()
    plt.title(r'$A$');

[9]: x, omega = crdata.sparse_normal_representations(key2, N, K, 1)
    x = jnp.squeeze(x)

[10]: fig=plt.figure(figsize=(8,6), dpi=100, facecolor='w', edgecolor='k')
    plt.stem(x, markerfmt='.')
2.2.1 Standard sparse recovery problems for compressive sensing

Basis pursuit

The simple form of basis pursuit problem is:

\[ \min_x \|x\|_1 \text{ s.t. } Ax = b \]

[12]: # Compute the measurements
b\theta = A.times(x)

[13]: fig=plt.figure(figsize=(8,6), dpi= 100, facecolor='w', edgecolor='k')
plt.stem(b\theta, markerfmt='.');
sol = yall1.solve(A, b0)

int(sol.iterations), int(sol.n_times), int(sol.n_trans)

(30, 61, 32)

norm(sol.x-x)/norm(x)

DeviceArray(0.01208655, dtype=float64)

fig=plt.figure(figsize=(8,6), dpi= 100, facecolor='w', edgecolor='k')
plt.subplot(211)
plt.title('original')
plt.stem(x, markerfmt='.', linefmt='gray');
plt.subplot(212)
plt.stem(sol.x, markerfmt='.');
plt.title('reconstruction');
Basis pursuit denoising

The simple form of L1-L2 unconstrained minimization or basis pursuit denoising is:

$$\min_x \|x\|_1 + \frac{1}{2\rho}\|Ax - b\|_2^2$$

```
[18]: %timeit yall1.solve(A, b0).x.block_until_ready()
4.77 ms ± 47.5 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
[19]: sigma = 0.01
noise = sigma * random.normal(key3, (M,))
```

```
[20]: crs.snr(b0, noise)
[20]: DeviceArray(27.34386008, dtype=float64)
```

```
[21]: b = b0 + noise
```
```python
[22]: sol = yall1.solve(A, b, rho=0.01)

[23]: int(sol.iterations), int(sol.n_times), int(sol.n_trans)
[23]: (28, 57, 30)

[24]: norm(sol.x-x)/norm(x)
[24]: DeviceArray(0.05690205, dtype=float64)

[25]: fig=plt.figure(figsize=(8,6), dpi=100, facecolor='w', edgecolor='k')
plt.subplot(211)
plt.title('original')
plt.stem(x, markerfmt='.', linefmt='gray');
plt.subplot(212)
plt.stem(sol.x, markerfmt='.');
plt.title('reconstruction');

original

reconstruction

[26]: %timeit yall1.solve(A, b, rho=0.01).x.block_until_ready()
4.47 ms ± 49 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```
Basis pursuit with inequality constraints

The simple form of L1 minimization with L2 constraints or basis pursuit with inequality constraints is:

$$\min_x \|x\|_1 \text{ s.t. } \|Ax - b\|_2 \leq \delta$$

[27]: delta = float(norm(noise))
    delta

[27]: 0.16467458902598495

[28]: sol = yall1.solve(A, b, delta=delta)

[29]: int(sol.iterations), int(sol.n_times), int(sol.n_trans)

[29]: (26, 53, 28)

[30]: norm(sol.x-x)/norm(x)

[30]: DeviceArray(0.05584753, dtype=float64)

[31]: fig=plt.figure(figsize=(8,6), dpi= 100, facecolor='w', edgecolor='k')
    plt.subplot(211)
    plt.title('original')
    plt.stem(x, markerfmt='.', linefmt='gray');
    plt.subplot(212)
    plt.stem(sol.x, markerfmt='.');
    plt.title('reconstruction');
2.2.2 Non-negative counterparts

In this case, the signal $x$ with the sparse representation $\alpha = Wx$ has only non-negative entries. i.e. if an entry in $x$ is non-zero, it is positive. This is typical for images.

Let us construct a sparse representation with non-negative entries.

```python
xp = jnp.abs(x)
fig=plt.figure(figsize=(8,6), dpi= 100, facecolor='w', edgecolor='k')
plt.stem(xp, markerfmt='.');
```
Non-negative basis pursuit

The simple form of basis pursuit for non-negative $x$ is:

$$\min_x \|x\|_1 \text{ s.t. } Ax = b \text{ and } x \succeq 0$$

[35]: b0p = A.times(xp)

[36]: fig=plt.figure(figsize=(8,6), dpi= 100, facecolor='w', edgecolor='k')
    plt.stem(b0p, markerfmt='.');
```python
sol = yall1.solve(A, b0p, nonneg=True)

int(sol.iterations), int(sol.n_times), int(sol.n_trans)

(36, 73, 38)

norm(sol.x-xp)/norm(xp)

DeviceArray(0.01095682, dtype=float64)

fig=plt.figure(figsize=(8,6), dpi= 100, facecolor='w', edgecolor='k')
plt.subplot(211)
plt.title('original')
plt.stem(xp, markerfmt='.', linefmt='gray');
plt.subplot(212)
plt.stem(sol.x, markerfmt='.');
plt.title('reconstruction');
```
Non-negative basis pursuit denoising

The simple form of L1-L2 unconstrained minimization with non-negative $x$ is:

$$\min_x \|x\|_1 + \frac{1}{2\rho} \|Ax - b\|_2^2 \text{ s.t. } x \succeq 0$$

[41]: \%
\timeit yall1.solve(A, b0p, nonneg=True).x.block_until_ready()
5.27 ms ± 72.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

[42]: crs.snr(b0p, noise)
DeviceArray(27.43652935, dtype=float64)

[43]: bp = b0p + noise

[44]: sol = yall1.solve(A, bp, nonneg=\textbf{True}, rho=0.01)
```python
int(sol.iterations), int(sol.n_times), int(sol.n_trans)
```

```
(28, 57, 30)
```

```
norm(sol.x-xp)/norm(xp)
```

```
DeviceArray(0.04551954, dtype=float64)
```

```python
fig=plt.figure(figsize=(8,6), dpi= 100, facecolor='w', edgecolor='k')
plt.subplot(211)
plt.title('original')
plt.stem(xp, markerfmt='.', linefmt='gray');
plt.subplot(212)
plt.stem(sol.x, markerfmt='.');
plt.title('reconstruction');
```

```python
%timeit yall1.solve(A, bp, nonneg=True, rho=0.01).x.block_until_ready()
```

```
4.58 ms ± 85.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```
Non-negative basis pursuit with inequality constraints

\[ \min_x \|x\|_1 \ \text{s.t.} \ \|Ax - b\|_2 \leq \delta \ \text{and} \ x \geq 0 \]

[49]: sol = yall1.solve(A, bp, delta=delta)

[50]: int(sol.iterations), int(sol.n_times), int(sol.n_trans)
[50]: (24, 49, 26)

[51]: norm(sol.x-xp)/norm(xp)
[51]: DeviceArray(0.05724303, dtype=float64)

[52]: fig = plt.figure(figsize=(8,6), dpi=100, facecolor='w', edgecolor='k')
plt.subplot(211)
plt.title('original')
plt.stem(xp, markerfmt='.', linefmt='gray');
plt.subplot(212)
plt.stem(sol.x, markerfmt='.');
plt.title('reconstruction');
```
[53]: %timeit yall1.solve(A, bp, delta=delta).x.block_until_ready()
    4.35 ms ± 17 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```
3.1 Installing as a package

Directly from our GITHUB repository:

```
python -m pip install git+https://github.com/carnotresearch/cr-sparse.git
```

3.2 Working with the source code in development mode

Clone the repository:

```
git clone https://github.com/carnotresearch/cr-sparse.git
```

Change into the code:

```
cd cr-sparse
```

Install the package in development mode:

```
python -m pip install -e .
```

3.3 Examples

Explore the examples directory:

```
cd examples
```
4.1 Sparsifying Dictionaries and Sensing Matrices

4.1.1 Functions for constructing sparsifying dictionaries and sensing matrices

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gaussian_mtx(key, N, D[, normalize_atoms])</code></td>
<td>A dictionary/sensing matrix where entries are drawn independently from normal distribution.</td>
</tr>
<tr>
<td><code>rademacher_mtx(key, M, N)</code></td>
<td>A dictionary/sensing matrix where entries are drawn independently from Rademacher distribution.</td>
</tr>
<tr>
<td><code>random_onb(key, N)</code></td>
<td>Generates a random orthonormal basis</td>
</tr>
<tr>
<td><code>hadamard(n[, dtype])</code></td>
<td>Hadamard matrices of size ( n \times n )</td>
</tr>
<tr>
<td><code>hadamard_basis(n)</code></td>
<td>A Hadamard basis</td>
</tr>
<tr>
<td><code>dirac_hadamard_basis(n)</code></td>
<td>A dictionary consisting of identity basis and hadamard bases</td>
</tr>
<tr>
<td><code>dct_basis(N)</code></td>
<td>DCT Basis</td>
</tr>
<tr>
<td><code>dirac_dct_basis(n)</code></td>
<td>A dictionary consisting of identity and DCT bases</td>
</tr>
<tr>
<td><code>dirac_hadamard_dct_basis(n)</code></td>
<td>A dictionary consisting of identity, Hadamard and DCT bases</td>
</tr>
<tr>
<td><code>fourier_basis(n)</code></td>
<td>Fourier basis</td>
</tr>
</tbody>
</table>

**cr.sparse.dict.gaussian_mtx**

`cr.sparse.dict.gaussian_mtx(key, N, D, normalize_atoms=True)`

A dictionary/sensing matrix where entries are drawn independently from normal distribution.
cr.sparse.dict.rademacher_mtx
\[ \text{cr.sparse.dict.rademacher_mtx}(key, M, N) \]
A dictionary/sensing matrix where entries are drawn independently from Rademacher distribution.

cr.sparse.dict.random_onb
\[ \text{cr.sparse.dict.random_onb}(key, N) \]
Generates a random orthonormal basis

cr.sparse.dict.hadamard
\[ \text{cr.sparse.dict.hadamard}(n, dtype=\langle \text{class 'int'} \rangle) \]
Hadamard matrices of size \( n \times n \)

cr.sparse.dict.hadamard_basis
\[ \text{cr.sparse.dict.hadamard_basis}(n) \]
A Hadamard basis

cr.sparse.dict.dirac_hadamard_basis
\[ \text{cr.sparse.dict.dirac_hadamard_basis}(n) \]
A dictionary consisting of identity basis and hadamard bases

cr.sparse.dict.dct_basis
\[ \text{cr.sparse.dict.dct_basis}(N) \]
DCT Basis

cr.sparse.dict.dirac_dct_basis
\[ \text{cr.sparse.dict.dirac_dct_basis}(n) \]
A dictionary consisting of identity and DCT bases

cr.sparse.dict.dirac_hadamard_dct_basis
\[ \text{cr.sparse.dict.dirac_hadamard_dct_basis}(n) \]
A dictionary consisting of identity, Hadamard and DCT bases
4.1.2 Dictionary properties

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gram(A)</code></td>
<td>Computes the Gram matrix $G = A^T A$</td>
</tr>
<tr>
<td><code>frame(A)</code></td>
<td>Computes the frame matrix $G = AA^T$</td>
</tr>
<tr>
<td><code>coherence_with_index(A)</code></td>
<td>Returns the coherence of a dictionary $A$ along with indices of most correlated atoms</td>
</tr>
<tr>
<td><code>coherence(A)</code></td>
<td>Computes the coherence of a dictionary $A$</td>
</tr>
<tr>
<td><code>frame_bounds(A)</code></td>
<td>Computes the frame bounds (largest and smallest singular value)</td>
</tr>
<tr>
<td><code>upper_frame_bound(A)</code></td>
<td>Computes the upper frame bound for a dictionary</td>
</tr>
<tr>
<td><code>lower_frame_bound(A)</code></td>
<td>Computes the lower frame bound for a dictionary</td>
</tr>
<tr>
<td><code>babel(A)</code></td>
<td>Computes the babel function for a dictionary (generalized coherence)</td>
</tr>
</tbody>
</table>

4.1. Sparsifying Dictionaries and Sensing Matrices
### `cr.sparse.dict.frame_bounds`

**`cr.sparse.dict.frame_bounds(A)`**

Computes the frame bounds (largest and smallest singular value)

### `cr.sparse.dict.upper_frame_bound`

**`cr.sparse.dict.upper_frame_bound(A)`**

Computes the upper frame bound for a dictionary

### `cr.sparse.dict.lower_frame_bound`

**`cr.sparse.dict.lower_frame_bound(A)`**

Computes the lower frame bound for a dictionary

### `cr.sparse.dict.babel`

**`cr.sparse.dict.babel(A)`**

Computes the babel function for a dictionary (generalized coherence)

### 4.1.3 Dictionary comparison

These functions are useful for comparing dictionaries during the dictionary learning process.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mutual_coherence_with_index(A,B)</code></td>
<td>Mutual coherence between two dictionaries A and B along with indices of most correlated atoms</td>
</tr>
<tr>
<td><code>mutual_coherence(A,B)</code></td>
<td>Mutual coherence between two dictionaries A and B</td>
</tr>
<tr>
<td><code>matching_atoms_ratio(A,B[, distance_threshold])</code></td>
<td>Identifies how many atoms are very close between dictionaries A and B</td>
</tr>
</tbody>
</table>

### `cr.sparse.dict.mutual_coherence_with_index`

**`cr.sparse.dict.mutual_coherence_with_index(A,B)`**

Mutual coherence between two dictionaries A and B along with indices of most correlated atoms
**cr.sparse.dict.mutual_coherence**

**cr.sparse.dict.mutual_coherence(A, B)**

“Mutual coherence between two dictionaries A and B

**cr.sparse.dict.matching_atoms_ratio**

**cr.sparse.dict.matching_atoms_ratio(A, B, distance_threshold=0.01)**

Identifies how many atoms are very close between dictionaries A and B

### 4.2 Linear Operators

We provide a collection of linear operators with efficient JAX based implementations that are relevant in standard signal/image processing problems. We also provide a bunch of utilities to combine and convert linear operators.

This module is inspired by pylops although the implementation approach is different.

A linear operator $T : X \rightarrow Y$ connects a model space $X$ to a data space $Y$.

A linear operator satisfies following laws:

$$T(x + y) = T(x) + T(y)$$

$$T(\alpha x) = \alpha T(x)$$

Thus, for a general linear combination:

$$T(\alpha x + \beta y) = \alpha T(x) + \beta T(y)$$

We are concerned with linear operators $T : \mathbb{F}^n \rightarrow \mathbb{F}^m$ where $\mathbb{F}$ is either the field of real numbers or complex numbers. $X = \mathbb{F}^n$ is the model space and $Y = \mathbb{F}^m$ is the data space.

Such a linear operator can be represented by a two dimensional matrix $A$.

The forward operation is given by:

$$y = Ax.$$ 

The corresponding adjoint operation is given by:

$$\hat{x} = A^H y$$

We represent a linear operator by a pair of functions `times` and `trans`. The `times` function implements the forward operation while the `trans` function implements the adjoint operation.

An inverse problem consists of computing $x$ given $y$ and $A$.

### 4.2.1 Data types

<table>
<thead>
<tr>
<th>Operator</th>
<th>Represents a finite linear operator $T : A \rightarrow B$ where $A$ and $B$ are finite vector spaces.</th>
</tr>
</thead>
</table>

4.2. Linear Operators
Represents a finite linear operator $T: A \rightarrow B$ where $A$ and $B$ are finite vector spaces.

Parameters

- **times** – A function implementing $T(x)$
- **trans** – A function implementing $T^H(x)$
- **m** – The dimension of the destination vector space $B$
- **n** – The dimension of the source vector space $A$
- **linear** – Indicates if the operator is linear or not
- **jit_safe** – Indicates if the operator can be safely JIT compiled
- **matrix_safe** – Indicates if the operator can accept a matrix of vectors

Note: While most of the operators in the library are linear operators, some are not. Prominent examples include operators like real part operator, imaginary part operator. These operators are provided for convenience.

Attributes

- **jit_safe** Indicates if the times and trans functions can be safely jit compiled
- **linear** Indicates if the operator is linear or not
- **matrix_safe** Indicates if the operator can accept a matrix of vectors
- **shape** Dimension of the linear operator $(m, n)$
- **times** A linear function mapping from $A$ to $B$
- **trans** Corresponding adjoint linear function mapping from $B$ to $A$

4.2.2 Basic operators

- **identity**(m[n]) Returns an identity linear operator from $A$ to $B$
- **matrix**(A) Converts a two-dimensional matrix to a linear operator
- **diagonal**(d) Returns a linear operator which can be represented by a diagonal matrix
- **zero**(m[n]) Returns a linear operator which maps everything to 0 vector in data space
- **flipud**(n) Returns an operator which flips the order of entries in input upside down

continues on next page
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sum(n)</code></td>
<td>Returns an operator which computes the sum of a vector</td>
</tr>
<tr>
<td><code>pad_zeros(n, before, after)</code></td>
<td>Adds zeros before and after a vector.</td>
</tr>
<tr>
<td><code>symmetrize(n)</code></td>
<td>An operator which constructs a symmetric vector by prepending the input in reversed order</td>
</tr>
<tr>
<td><code>restriction(n, indices)</code></td>
<td>An operator which computes ( y = x[I] ) over an index set ( I )</td>
</tr>
</tbody>
</table>

**cr.sparse.lop.identity**

```python
cr.sparse.lop.identity(m, n=None)
Returns an identity linear operator from A to B
```

**cr.sparse.lop.matrix**

```python
cr.sparse.lop.matrix(A)
Converts a two-dimensional matrix to a linear operator
```

**cr.sparse.lop.diagonal**

```python
cr.sparse.lop.diagonal(d)
Returns a linear operator which can be represented by a diagonal matrix
```

**cr.sparse.lop.zero**

```python
cr.sparse.lop.zero(m, n=None)
Returns a linear operator which maps everything to 0 vector in data space
```

**cr.sparse.lop.flipud**

```python
cr.sparse.lop.flipud(n)
Returns an operator which flips the order of entries in input upside down
```

**cr.sparse.lop.sum**

```python
cr.sparse.lop.sum(n)
Returns an operator which computes the sum of a vector
```

**cr.sparse.lop.pad_zeros**

```python
cr.sparse.lop.pad_zeros(n, before, after)
Adds zeros before and after a vector.
```

**Note:** This operator is not JIT compliant
cr.sparse.lop.symmetrize

**cr.sparse.lop.symmetrize**(*)
n
An operator which constructs a symmetric vector by pre-pending the input in reversed order

**cr.sparse.lop.restriction**

**cr.sparse.lop.restriction**(*)

An operator which computes \( y = x[I] \) over an index set \( I \)

### 4.2.3 Signal processing operators

**fourier_basis_1d**(*)

Returns an operator which represents the DFT orthonormal basis

**dirac_fourier_basis_1d**(*)

Returns an operator for a two-orthobasis dictionary consisting of Dirac basis and Fourier basis

### 4.2.4 Random compressive sensing operators

**gaussian_dict**(*)

An operator which represents a Gaussian sensix matrix (with normalized columns)

**rademacher_dict**(*)

An operator which represents a Rademacher sensing matrix

**random_onb_dict**(*)

An operator representing a random orthonormal basis

**random_orthonormal_rows_dict**(*)

An operator whose rows are orthonormal (sampled from a random orthonormal basis)
4.2.5 Convenience operators

These operators are technically not linear on $\mathbb{F}^n \to \mathbb{F}^m$

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>real(n)</code></td>
<td>Returns the real parts of a vector of complex numbers</td>
</tr>
</tbody>
</table>

Note: This is a self-adjoint operator. This is not a linear operator.

4.2.6 Operator algebra

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>neg(A)</code></td>
<td>Returns the negative of a linear operator $T = -A$</td>
</tr>
<tr>
<td><code>scale(A, alpha)</code></td>
<td>Returns the linear operator $T = \alpha A$ for the operator $A$</td>
</tr>
<tr>
<td><code>add(A, B)</code></td>
<td>Returns the sum of two linear operators $T = A + B$</td>
</tr>
<tr>
<td><code>subtract(A, B)</code></td>
<td>Returns a linear operator $T = A - B$</td>
</tr>
<tr>
<td><code>compose(A, B)</code></td>
<td>Returns the composite linear operator $T = AB$ such that $T(x) = A(B(x))$</td>
</tr>
<tr>
<td><code>transpose(A)</code></td>
<td>Returns the transpose of a given operator $T = A^T$</td>
</tr>
<tr>
<td><code>hermitian(A)</code></td>
<td>Returns the Hermitian transpose of a given operator $T = A^H$</td>
</tr>
<tr>
<td><code>hcat(A, B)</code></td>
<td>Returns the linear operator $T = [A \ B]$</td>
</tr>
</tbody>
</table>

continues on next page
Table 10 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>power(A, p)</code></td>
<td>Returns the linear operator $T = A^p$</td>
</tr>
</tbody>
</table>

**cr.sparse.lop.neg**

`cr.sparse.lop.neg(A)`

Returns the negative of a linear operator $T = -A$

**cr.sparse.lop.scale**

`cr.sparse.lop.scale(A, alpha)`

Returns the linear operator $T = \alpha A$ for the operator $A$

**cr.sparse.lop.add**

`cr.sparse.lop.add(A, B)`

Returns the sum of two linear operators $T = A + B$

**cr.sparse.lop.subtract**

`cr.sparse.lop.subtract(A, B)`

Returns a linear operator $T = A - B$

**cr.sparse.lop.compose**

`cr.sparse.lop.compose(A, B)`

Returns the composite linear operator $T = AB$ such that $T(x) = A(B(x))$

**cr.sparse.lop.transpose**

`cr.sparse.lop.transpose(A)`

Returns the transpose of a given operator $T = A^T$

**cr.sparse.lop.hermitian**

`cr.sparse.lop.hermitian(A)`

Returns the Hermitian transpose of a given operator $T = A^H$

**cr.sparse.lop.hcat**

`cr.sparse.lop.hcat(A, B)`

Returns the linear operator $T = [A \ B]$
cr.sparse.lop.power

cr.sparse.lop.power(A, p)
 Returns the linear operator \( T = A^p \)

4.2.7 Operator parts

\[
\begin{align*}
\text{column}(T, i) & \quad \text{Returns the i-th column of the operator } T \\
\text{columns}(T, \text{indices}) & \quad \text{Returns the i-th column of the operator } T
\end{align*}
\]

cr.sparse.lop.column

cr.sparse.lop.column(T, i)
 Returns the i-th column of the operator T

cr.sparse.lop.columns

cr.sparse.lop.columns(T, indices)
 Returns the i-th column of the operator T

4.2.8 Properties of a linear operator

These are still experimental and not efficient.

\[
\begin{align*}
\text{upper_frame_bound}(T) & \quad \text{Computes the upper frame bound for a linear operator}
\end{align*}
\]

cr.sparse.lop.upper_frame_bound

cr.sparse.lop.upper_frame_bound(T)
 Computes the upper frame bound for a linear operator

4.2.9 Utilities

\[
\begin{align*}
\text{jit}(\text{operator}) & \quad \text{Returns the same linear operator with compiled times and trans functions} \\
\text{to_matrix}(A) & \quad \text{Converts a linear operator to a matrix} \\
\text{to_adjoint_matrix}(A) & \quad \text{Converts the adjoint of a linear operator to a matrix} \\
\text{to_complex_matrix}(A) & \quad \text{Converts a linear operator to a matrix in complex numbers}
\end{align*}
\]
CR.Sparse

**cr.sparse.lop.jit**

**cr.sparse.lop.jit(operator)**

Returns the same linear operator with compiled times and trans functions

**cr.sparse.lop.to_matrix**

**cr.sparse.lop.to_matrix(A)**

Converts a linear operator to a matrix

**cr.sparse.lop.to_adjoint_matrix**

**cr.sparse.lop.to_adjoint_matrix(A)**

Converts the adjoint of a linear operator to a matrix

**cr.sparse.lop.to_complex_matrix**

**cr.sparse.lop.to_complex_matrix(A)**

Converts a linear operator to a matrix in complex numbers

### 4.3 Greedy Sparse Recovery/Approximation Algorithms

- *Basic Matching Pursuit Based Algorithms*
- *Hard Thresholding Based Algorithms*
- *Data Types*
- *Utilities*
- *Using the greedy algorithms*

#### Algorithm versions

Several algorithms are available in multiple versions.

The library allows a dictionary or a sensing process to be represented as either:

- A matrix of real/complex values
- A linear operator with fast implementation (see cr.sparse.lop module)

E.g. a partial sensing sensing matrix can be implemented more efficiently using a linear operator consisting of applying the fourier transform followed by selecting a subset of fourier measurements.

- A prefix **matrix** _indicates_ an implementation which accepts matrices as dictionaries or compressive sensors.
- A prefix **operator** _indicates_ an implementation which accepts linear operators described in cr.sparse.lop module as dictionaries or compressive sensors.
- A suffix **_jit** means_ that it is the JIT (Just In Time) compiled version of the original implementation.
- A suffix **_multi** means_ that it is the version of implementation which can process multiple signals/measurement vectors simultaneously. The recovery problem \( y = \Phi x + e \) is extended to \( Y = \Phi X + E \) such that:
– Each column of $Y$ represents one signal/measurement vector
– Each column of $X$ represents one representation vector to be recovered
– Each column of $E$ representation corresponding measurement error/noise.

Conditions on dictionaries/sensing matrices

Different algorithms have different requirements on the dictionaries or sensing matrices:

- Some algorithms accept overcomplete dictionaries/sensing matrices with unit norm columns
- Some algorithms accept overcomplete dictionaries/sensing matrices with orthogonal rows
- Some algorithms accept any overcomplete dictionary

4.3.1 Basic Matching Pursuit Based Algorithms

Orthogonal Matching Pursuit

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>omp.solve(Phi, y, max_iters[, max_res_norm])</code></td>
<td>Solves the recovery/approximation problem $y = \Phi x + e$ using Orthogonal Matching Pursuit</td>
</tr>
<tr>
<td><code>omp.matrix_solve(Phi, y, max_iters[, ...])</code></td>
<td>Solves the recovery/approximation problem $y = \Phi x + e$ using Orthogonal Matching Pursuit</td>
</tr>
<tr>
<td><code>omp.matrix_solve_jit(Phi, y, max_iters[, ...])</code></td>
<td>Solves the recovery/approximation problem $y = \Phi x + e$ using Orthogonal Matching Pursuit</td>
</tr>
<tr>
<td><code>omp.matrix_solve_multi(Phi, y, max_iters[, ...])</code></td>
<td>Solves the MMV recovery/approximation problem $Y = \Phi X + E$ using Orthogonal Matching Pursuit</td>
</tr>
</tbody>
</table>

`cr.sparse.pursuit.omp.solve`

Solves the recovery/approximation problem $y = \Phi x + e$ using Orthogonal Matching Pursuit

`cr.sparse.pursuit.omp.solve(Phi, y, max_iters, max_res_norm=1e-06)`

Solves the recovery/approximation problem $y = \Phi x + e$ using Orthogonal Matching Pursuit

`cr.sparse.pursuit.omp.matrix_solve`

Solves the recovery/approximation problem $y = \Phi x + e$ using Orthogonal Matching Pursuit

`cr.sparse.pursuit.omp.matrix_solve(Phi, y, max_iters, max_res_norm=1e-06)`

Solves the recovery/approximation problem $y = \Phi x + e$ using Orthogonal Matching Pursuit

`cr.sparse.pursuit.omp.matrix_solve_jit`

Solves the recovery/approximation problem $y = \Phi x + e$ using Orthogonal Matching Pursuit

`cr.sparse.pursuit.omp.matrix_solve_jit(Phi, y, max_iters, max_res_norm=1e-06)`

Solves the recovery/approximation problem $y = \Phi x + e$ using Orthogonal Matching Pursuit

4.3. Greedy Sparse Recovery/Approximation Algorithms 45
cr.sparse.pursuit.omp.matrix_solve_multi

`cr.sparse.pursuit.omp.matrix_solve_multi(Phi, y, max_iters, max_res_norm=1e-06)` Solves the MMV recovery/approximation problem $Y = \Phi X + E$ using Orthogonal Matching Pursuit

Extends `cr.sparse.pursuit.omp.solve()` using `jax.vmap()`.

Compressive Sensing Matching Pursuit (CSMP) Algorithms

**Compressive Sampling Matching Pursuit**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cosamp.solve(Phi, y, K[, max_iters, ...])</code></td>
<td>Solves the sparse recovery problem $y = \Phi x + e$ using Compressive Sampling Matching Pursuit for linear operators</td>
</tr>
<tr>
<td><code>cosamp.matrix_solve(Phi, y, K[, max_iters, ...])</code></td>
<td>Solves the sparse recovery problem $y = \Phi x + e$ using Compressive Sampling Matching Pursuit for matrices</td>
</tr>
<tr>
<td><code>cosamp.matrix_solve_jit(Phi, y, K[, ...])</code></td>
<td>Solves the sparse recovery problem $y = \Phi x + e$ using Compressive Sampling Matching Pursuit for matrices</td>
</tr>
<tr>
<td><code>cosamp.operator_solve(Phi, y, K[, ...])</code></td>
<td>Solves the sparse recovery problem $y = \Phi x + e$ using Compressive Sampling Matching Pursuit for linear operators</td>
</tr>
<tr>
<td><code>cosamp.operator_solve_jit(Phi, y, K[, ...])</code></td>
<td>Solves the sparse recovery problem $y = \Phi x + e$ using Compressive Sampling Matching Pursuit for linear operators</td>
</tr>
</tbody>
</table>

**CR.Sparse**

**cr.sparse.pursuit.cosamp.solve**

`cr.sparse.pursuit.cosamp.solve(Phi, y, K, max_iters=None, res_norm_rtol=0.0001)` Solves the sparse recovery problem $y = \Phi x + e$ using Compressive Sampling Matching Pursuit for linear operators

**cr.sparse.pursuit.cosamp.matrix_solve**

`cr.sparse.pursuit.cosamp.matrix_solve(Phi, y, K, max_iters=None, res_norm_rtol=0.0001)` Solves the sparse recovery problem $y = \Phi x + e$ using Compressive Sampling Matching Pursuit for matrices
cr.sparse.pursuit.cosamp.matrix_solve_jit

`cr.sparse.pursuit.cosamp.matrix_solve_jit(Phi, y, K[, max_iters=None, res_norm_rtol=0.0001])`
Solves the sparse recovery problem $y = \Phi x + e$ using Compressive Sampling Matching Pursuit for matrices

---

cr.sparse.pursuit.cosamp.operator_solve

`cr.sparse.pursuit.cosamp.operator_solve(Phi, y, K[, max_iters=None, res_norm_rtol=0.0001])`
Solves the sparse recovery problem $y = \Phi x + e$ using Compressive Sampling Matching Pursuit for linear operators

---

cr.sparse.pursuit.cosamp.operator_solve_jit

`cr.sparse.pursuit.cosamp.operator_solve_jit(Phi, y, K[, max_iters=None, res_norm_rtol=0.0001])`
Solves the sparse recovery problem $y = \Phi x + e$ using Compressive Sampling Matching Pursuit for linear operators

---

**Subspace Pursuit**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sp.solve(Phi, y, K[, ...])</code></td>
<td>Solves the sparse recovery problem $y = \Phi x + e$ using Subspace Pursuit for linear operators</td>
</tr>
<tr>
<td><code>sp.matrix_solve(Phi, y, K[, ...])</code></td>
<td>Solves the sparse recovery problem $y = \Phi x + e$ using Subspace Pursuit for matrices</td>
</tr>
<tr>
<td><code>sp.matrix_solve_jit(Phi, y, K[, ...])</code></td>
<td>Solves the sparse recovery problem $y = \Phi x + e$ using Subspace Pursuit for matrices</td>
</tr>
<tr>
<td><code>sp.operator_solve(Phi, y, K[, ...])</code></td>
<td>Solves the sparse recovery problem $y = \Phi x + e$ using Subspace Pursuit for linear operators</td>
</tr>
<tr>
<td><code>sp.operator_solve_jit(Phi, y, K[, ...])</code></td>
<td>Solves the sparse recovery problem $y = \Phi x + e$ using Subspace Pursuit for linear operators</td>
</tr>
</tbody>
</table>

---

**Greedy Sparse Recovery/Approximation Algorithms**

4.3. Greedy Sparse Recovery/Approximation Algorithms
**CR.Sparse**

**cr.sparse.pursuit.sp.matrix_solve**

```python
cr.sparse.pursuit.sp.matrix_solve(Phi, y, K, max_iters=None, res_norm_rtol=0.0001)
```

Solves the sparse recovery problem $y = \Phi x + e$ using Subspace Pursuit for matrices

**cr.sparse.pursuit.sp.matrix_solve_jit**

```python
cr.sparse.pursuit.sp.matrix_solve_jit(Phi, y, K, max_iters=None, res_norm_rtol=0.0001)
```

Solves the sparse recovery problem $y = \Phi x + e$ using Subspace Pursuit for matrices

**cr.sparse.pursuit.sp.operator_solve**

```python
cr.sparse.pursuit.sp.operator_solve(Phi, y, K, max_iters=None, res_norm_rtol=0.0001)
```

Solves the sparse recovery problem $y = \Phi x + e$ using Subspace Pursuit for linear operators

**cr.sparse.pursuit.sp.operator_solve_jit**

```python
cr.sparse.pursuit.sp.operator_solve_jit(Phi, y, K, max_iters=None, res_norm_rtol=0.0001)
```

Solves the sparse recovery problem $y = \Phi x + e$ using Subspace Pursuit for linear operators

### 4.3.2 Hard Thresholding Based Algorithms

**Iterative Hard Thresholding**

```python
iht.solve(Phi, y, K[, normalized, ...])
```

Solves the sparse recovery problem $y = \Phi x + e$ using Iterative Hard Thresholding for linear operators

```python
iht.matrix_solve(Phi, y, K[, normalized, ...])
```

Solves the sparse recovery problem $y = \Phi x + e$ using Iterative Hard Thresholding for matrices

```python
iht.matrix_solve_jit(Phi, y, K[, ...])
```

Solves the sparse recovery problem $y = \Phi x + e$ using Iterative Hard Thresholding for matrices

```python
iht.operator_solve(Phi, y, K[, normalized, ...])
```

Solves the sparse recovery problem $y = \Phi x + e$ using Iterative Hard Thresholding for linear operators

```python
iht.operator_solve_jit(Phi, y, K[, ...])
```

Solves the sparse recovery problem $y = \Phi x + e$ using Iterative Hard Thresholding for linear operators

**cr.sparse.pursuit.iht.solve**

```python
cr.sparse.pursuit.iht.solve(Phi, y, K, normalized=False, step_size=None, max_iters=None, res_norm_rtol=0.0001)
```

Solves the sparse recovery problem $y = \Phi x + e$ using Iterative Hard Thresholding for linear operators
The `cr.sparse.pursuit.iht.matrix_solve` function solves the sparse recovery problem $y = \Phi x + e$ using Iterative Hard Thresholding for matrices.

```python
cr.sparse.pursuit.iht.matrix_solve(Phi, y, K, normalized=False, step_size=None, max_iters=None, res_norm_rtol=0.0001)
```

Solves the sparse recovery problem $y = \Phi x + e$ using Iterative Hard Thresholding for matrices.

The `cr.sparse.pursuit.iht.matrix_solve_jit` function is a JIT-compiled version of the above function.

```python
cr.sparse.pursuit.iht.matrix_solve_jit(Phi, y, K, normalized=False, step_size=None, max_iters=None, res_norm_rtol=0.0001)
```

Solves the sparse recovery problem $y = \Phi x + e$ using Iterative Hard Thresholding for matrices.

The `cr.sparse.pursuit.iht.operator_solve` function solves the sparse recovery problem $y = \Phi x + e$ using Iterative Hard Thresholding for linear operators.

```python
cr.sparse.pursuit.iht.operator_solve(Phi, y, K, normalized=False, step_size=None, max_iters=None, res_norm_rtol=0.0001)
```

Solves the sparse recovery problem $y = \Phi x + e$ using Iterative Hard Thresholding for linear operators.

The `cr.sparse.pursuit.iht.operator_solve_jit` function is a JIT-compiled version of the above function.

```python
cr.sparse.pursuit.iht.operator_solve_jit(Phi, y, K, normalized=False, step_size=None, max_iters=None, res_norm_rtol=0.0001)
```

Solves the sparse recovery problem $y = \Phi x + e$ using Iterative Hard Thresholding for linear operators.

**Hard Thresholding Pursuit**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>htp.solve</code></td>
<td>Solves the sparse recovery problem $y = \Phi x + e$ using Hard Thresholding</td>
</tr>
<tr>
<td></td>
<td>Pursuit for linear operators</td>
</tr>
<tr>
<td><code>htp.matrix_solve</code></td>
<td>Solves the sparse recovery problem $y = \Phi x + e$ using Hard Thresholding</td>
</tr>
<tr>
<td></td>
<td>Pursuit for matrices</td>
</tr>
<tr>
<td><code>htp.matrix_solve_jit</code></td>
<td>Solves the sparse recovery problem $y = \Phi x + e$ using Hard Thresholding</td>
</tr>
<tr>
<td></td>
<td>Pursuit for matrices</td>
</tr>
<tr>
<td><code>htp.operator_solve</code></td>
<td>Solves the sparse recovery problem $y = \Phi x + e$ using Hard Thresholding</td>
</tr>
<tr>
<td></td>
<td>Pursuit for linear operators</td>
</tr>
<tr>
<td><code>htp.operator_solve_jit</code></td>
<td>Solves the sparse recovery problem $y = \Phi x + e$ using Hard Thresholding</td>
</tr>
<tr>
<td></td>
<td>Pursuit for linear operators</td>
</tr>
</tbody>
</table>

The `cr.sparse.pursuit.htp.solve` function solves the sparse recovery problem $y = \Phi x + e$ using Hard Thresholding Pursuit for linear operators.

```python
cr.sparse.pursuit.htp.solve(Phi, y, K, normalized=False, step_size=None, max_iters=None, res_norm_rtol=0.0001)
```

Solves the sparse recovery problem $y = \Phi x + e$ using Hard Thresholding Pursuit for linear operators.
CR.Sparse

`cr.sparse.pursuit.htp.matrix_solve`

```python
cr.sparse.pursuit.htp.matrix_solve(Phi, y, K, normalized=False, step_size=None, max_iters=None, res_norm_rtol=0.0001)
```

Solves the sparse recovery problem $y = \Phi x + e$ using Hard Thresholding Pursuit for matrices

`cr.sparse.pursuit.htp.matrix_solve_jit`

```python
cr.sparse.pursuit.htp.matrix_solve_jit(Phi, y, K, normalized=False, step_size=None, max_iters=None, res_norm_rtol=0.0001)
```

Solves the sparse recovery problem $y = \Phi x + e$ using Hard Thresholding Pursuit for matrices

`cr.sparse.pursuit.htp.operator_solve`

```python
cr.sparse.pursuit.htp.operator_solve(Phi, y, K, normalized=False, step_size=None, max_iters=None, res_norm_rtol=0.0001)
```

Solves the sparse recovery problem $y = \Phi x + e$ using Hard Thresholding Pursuit for linear operators

`cr.sparse.pursuit.htp.operator_solve_jit`

```python
cr.sparse.pursuit.htp.operator_solve_jit(Phi, y, K, normalized=False, step_size=None, max_iters=None, res_norm_rtol=0.0001)
```

Solves the sparse recovery problem $y = \Phi x + e$ using Hard Thresholding Pursuit for linear operators

### 4.3.3 Data Types

**RecoverySolution**

Represents the solution of a sparse recovery problem

```python
```

Represents the solution of a sparse recovery problem

Consider a sparse recovery problem $y = \Phi x + e$. Assume that $x$ is supported on an index set $I$ i.e. the non-zero values of $x$ are in the sub-vector $x_I$, then the equation can be rewritten as $y = \Phi_I x_I + e$.

Solving the sparse recovery problem given $\Phi$ and $x$ involves identifying $I$ and estimating $x_I$. Then, the residual is $r = y - \Phi_I x_I$. An important quantity during the sparse recovery is the (squared) norm of the residual $\|r\|_2^2$ which is an estimate of the energy of error $e$.

This type combines all of this information together.

**Parameters**

- $x_I$ – estimate(s) of $x_I$
- $I$ – identified index set(s) $I$
- $r$ – residual(s) $r = y - \Phi_I x_I$
• **r_norm_sqr** – squared norm of residual \( \| r \|_2^2 \)

• **iterations** – Number of iterations required for the algorithm to converge

**Note:** The tuple can be used to solve multiple measurement vector problems also. In this case, each column (of individual parameters) represents the solution of corresponding single vector problems.

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I )</td>
<td>The support for non-zero values</td>
</tr>
<tr>
<td>iterations</td>
<td>The number of iterations it took to complete</td>
</tr>
<tr>
<td>( r )</td>
<td>The residuals</td>
</tr>
<tr>
<td>( r_{norm_sqr} )</td>
<td>The residual norm squared</td>
</tr>
<tr>
<td>( x_I )</td>
<td>Non-zero values</td>
</tr>
</tbody>
</table>

#### 4.3.4 Utilities

- **abs_max_idx(h)**
  Returns the index of entry with highest magnitude

- **gram_chol_update(L, v)**
  Incrementally updates the Cholesky factorization \( G = LL^T \) where \( G = \Phi^T \Phi \)

**cr.sparse.pursuit.abs_max_idx**

**cr.sparse.pursuit.abs_max_idx(h)**
Returns the index of entry with highest magnitude

**cr.sparse.pursuit.gram_chol_update**

**cr.sparse.pursuit.gram_chol_update(L, v)**
Incrementally updates the Cholesky factorization \( G = LL^T \) where \( G = \Phi^T \Phi \)

#### 4.3.5 Using the greedy algorithms

These algorithms solve the inverse problem \( y = \Phi x + e \) where \( \Phi \) and \( y \) are known and \( x \) is desired with \( \Phi \) being an overcomplete dictionary or a sensing matrix.

For sparse approximation problems, we require the following to invoke any of these algorithms:

- A sparsifying dictionary \( \Phi \).
- A signal \( y \) which is expected to have a sparse or compressible representation \( x \) in \( \Phi \).

For sparse recovery problems, we require the following to invoke any of these algorithms:

- A sensing matrix \( \Phi \) with suitable RIP or other properties.
- A measurement vector \( y \) generated by applying \( \Phi \) to a sparse signal \( x \)
A synthetic example

Build a Gaussian dictionary/sensing matrix:

```python
from jax import random
import cr.sparse.dict as crdict
M = 128
N = 256
key = random.PRNGKey(0)
Phi = crdict.gaussian_mtx(key, M, N)
```

Build a K-sparse signal with Gaussian non-zero entries:

```python
import cr.sparse.data as crdata
import jax.numpy as jnp
K = 16
key, subkey = random.split(key)
x, omega = crdata.sparse_normal_representations(key, N, K, 1)
x = jnp.squeeze(x)
```

Build the measurement vector:

```python
y = Phi @ x
```

We have built the necessary inputs for a sparse recovery problem. It is time to run the solver.

Import a sparse recovery solver:

```python
from cr.sparse.pursuit import cosamp
```

Solve the recovery problem:

```python
solution = cosamp.matrix_solve(Phi, y, K)
```

You can choose any other solver.

The support for the non-zero entries in the solution is given by `solution.I` and the values for non-zero entries are given by `solution.x_I`. You can build the sparse representation as follows:

```python
from cr.sparse import build_signal_from_indices_and_values
x_hat = build_signal_from_indices_and_values(N, solution.I, solution.x_I)
```

Finally, you can use the utility to evaluate the quality of reconstruction:

```python
from cr.sparse.ef import RecoveryPerformance
rp = RecoveryPerformance(Phi, y, x, x_hat)
rp.print()
```

This would output something like:

```
M: 128, N: 256, K: 16
x_norm: 3.817, y_norm: 3.922
x_hat_norm: 3.817, h_norm: 1.55e-06, r_norm: 1.72e-06
recovery_snr: 127.83 dB, measurement_snr: 127.16 dB
T0: [ 27 63 79 85 88 111 112 124 131 137 160 200 230 234 235 250]
R0: [ 27 63 79 85 88 111 112 124 131 137 160 200 230 234 235 250]
```
Overlap: [27 63 79 85 88 111 112 124 131 137 160 200 230 234 235 250], Correct: 16

success: True

4.4 Convex Optimization based Sparse Recovery/Approximation Algorithms

• Alternating Directions Methods

4.4.1 Alternating Directions Methods

A tutorial has been provided to explore these methods in action. The `yall1.solve` method is an overall wrapper method for solving different types of $\ell_1$ minimization problems. It in turn calls the lower level methods for solving specific types of problems.

- `yall1.solve(A, b[, x0, z0, W, weights, ...])` Wrapper method to solve a variety of $\ell_1$ minimization problems using ADMM
- `yall1.solve_bp(A, b, x0, z0, w, nonneg, ...)` Solves the problem $\min_x \|x\|_1 \text{s.t.} x = b$ using ADMM
- `yall1.solve_l1_l2(A, b, x0, z0, w, nonneg, ...)` Solves the problem $\min_x \|x\|_1 + \frac{1}{2\rho} \|Ax - b\|_2^2$ using ADMM
- `yall1.solve_l1_l2con(A, b, x0, z0, w, ...)` Solves the problem $\min_x \|x\|_1 \text{s.t.} \|Ax - b\|_2 \leq \delta$ using ADMM

```
cr.sparse.cvx.adm.yall1.solve

`cr.sparse.cvx.adm.yall1.solve(A, b, x0=None, z0=None, W=None, weights=None, nonneg=False, rho=0.0, delta=0.0, gamma=1.0, tolerance=0.005, max_iters=9999, jit=True)`

Wrapper method to solve a variety of $\ell_1$ minimization problems using ADMM
```

Parameters

- `A (jax.numpy.ndarray)` – Sensing matrix/dictionary
- `b (jax.numpy.ndarray)` – Signal being approximated
- `x0 (jax.numpy.ndarray)` – Initial value of solution (primary variable) $x$
- `z0 (jax.numpy.ndarray)` – Initial value of dual variable $z$
- `nonneg (bool)` – Flag to indicate if values in the solution are all non-negative
- `W (jax.numpy.ndarray)` – The sparsifying orthonormal basis such that $Wx$ is sparse
- `weights (jax.numpy.ndarray)` – The weights for individual entries in $x$
- `rho (float)` – weight for the quadratic penalty term

4.4. Convex Optimization based Sparse Recovery/Approximation Algorithms 53
**delta** float – constraint on the residual norm

**gamma** float – ADMM update parameter for \( x \)

**max_iters** int – maximum number of ADMM iterations

**Returns** Solution vector \( x \) and residual \( r \)

**Return type** RecoveryFullSolution

This function implements eq 2.25 of the paper.

**cr.sparse.cvx.adm.yall1.solve_bp**

**cr.sparse.cvx.adm.yall1.solve_bp**(A, b, x0, z0, w, nonneg, gamma, tolerance, max_iters)

Solves the problem \( \min \| x \|_{1} \text{s.t.} x = b \) using ADMM

This function implements eq 2.29 of the paper.

**cr.sparse.cvx.adm.yall1.solve_bp_jit**

**cr.sparse.cvx.adm.yall1.solve_bp_jit**(A, b, x0, z0, w, nonneg, gamma, tolerance, max_iters)

Solves the problem \( \min \| x \|_{1} \text{s.t.} x = b \) using ADMM

This function implements eq 2.29 of the paper.

**cr.sparse.cvx.adm.yall1.solve_l1_l2**

**cr.sparse.cvx.adm.yall1.solve_l1_l2**(A, b, x0, z0, w, nonneg, rho, gamma, tolerance, max_iters)

Solves the problem \( \min \| x \|_{1} + \frac{1}{2\rho} \| Ax - b \|_{2}^{2} \) using ADMM

This function implements eq 2.25 of the paper.

**cr.sparse.cvx.adm.yall1.solve_l1_l2_jit**

**cr.sparse.cvx.adm.yall1.solve_l1_l2_jit**(A, b, x0, z0, w, nonneg, rho, gamma, tolerance, max_iters)

Solves the problem \( \min \| x \|_{1} + \frac{1}{2\rho} \| Ax - b \|_{2}^{2} \) using ADMM

This function implements eq 2.25 of the paper.

**cr.sparse.cvx.adm.yall1.solve_l1_l2con**

**cr.sparse.cvx.adm.yall1.solve_l1_l2con**(A, b, x0, z0, w, nonneg, delta, gamma, tolerance, max_iters)

Solves the problem \( \min \| x \|_{1} \text{s.t.} \| Ax - b \|_{2} \leq \delta \) using ADMM

This function implements eq 2.27 of the paper.
Solves the problem \( \min \| x \|_1 \) s.t. \( \| A x - b \|_2 \leq \delta \) using ADMM

This function implements eq 2.27 of the paper.

4.5 Sample Data Generation Utilities

Utilities for generating test data

4.6 Utilities in cr.sparse module

4.6.1 Some checks and utilities for matrices (2D arrays)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transpose(A)</td>
<td>Returns the transpose of an array</td>
</tr>
<tr>
<td>hermitian(a)</td>
<td>Returns the conjugate transpose of an array</td>
</tr>
<tr>
<td>is_matrix(A)</td>
<td>Checks if an array is a matrix</td>
</tr>
<tr>
<td>is_square(A)</td>
<td>Checks if an array is a square matrix</td>
</tr>
<tr>
<td>is_symmetric(A)</td>
<td>Checks if an array is a symmetric matrix</td>
</tr>
<tr>
<td>is_hermitian(A)</td>
<td>Checks if an array is a Hermitian matrix</td>
</tr>
<tr>
<td>is_positive_definite(A)</td>
<td>Checks if an array is a symmetric positive definite matrix</td>
</tr>
<tr>
<td>has_orthogonal_columns(A)</td>
<td>Checks if a matrix has orthogonal columns</td>
</tr>
<tr>
<td>has_orthogonal_rows(A)</td>
<td>Checks if a matrix has orthogonal rows</td>
</tr>
<tr>
<td>has_unitary_columns(A)</td>
<td>Checks if a matrix has unitary columns</td>
</tr>
<tr>
<td>has_unitary_rows(A)</td>
<td>Checks if a matrix has unitary rows</td>
</tr>
</tbody>
</table>

4.6. Utilities in cr.sparse module

4.6.1 Some checks and utilities for matrices (2D arrays)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transpose(A)</td>
<td>Returns the transpose of an array</td>
</tr>
<tr>
<td>hermitian(a)</td>
<td>Returns the conjugate transpose of an array</td>
</tr>
<tr>
<td>is_matrix(A)</td>
<td>Checks if an array is a matrix</td>
</tr>
<tr>
<td>is_square(A)</td>
<td>Checks if an array is a square matrix</td>
</tr>
<tr>
<td>is_symmetric(A)</td>
<td>Checks if an array is a symmetric matrix</td>
</tr>
<tr>
<td>is_hermitian(A)</td>
<td>Checks if an array is a Hermitian matrix</td>
</tr>
<tr>
<td>is_positive_definite(A)</td>
<td>Checks if an array is a symmetric positive definite matrix</td>
</tr>
<tr>
<td>has_orthogonal_columns(A)</td>
<td>Checks if a matrix has orthogonal columns</td>
</tr>
<tr>
<td>has_orthogonal_rows(A)</td>
<td>Checks if a matrix has orthogonal rows</td>
</tr>
<tr>
<td>has_unitary_columns(A)</td>
<td>Checks if a matrix has unitary columns</td>
</tr>
<tr>
<td>has_unitary_rows(A)</td>
<td>Checks if a matrix has unitary rows</td>
</tr>
</tbody>
</table>

**cr.sparse.transpose**

```
cr.sparse.transpose(A)
```

Returns the transpose of an array

**Parameters**

- \( A \) (*jax.numpy.ndarray*) – A JAX array

**Returns**

Transpose of the array

**Return type**

*jax.numpy.ndarray*

**cr.sparse.hermitian**

```
cr.sparse.hermitian(a)
```

Returns the conjugate transpose of an array

**Parameters**

- \( A \) (*jax.numpy.ndarray*) – A JAX array

**Returns**

Conjugate transpose of the array

**Return type**

*jax.numpy.ndarray*
CR.Sparse

**cr.sparse.is_matrix**

```python
def is_matrix(A):
    # Checks if an array is a matrix
    Parameters
    A (jax.numpy.ndarray) – A JAX array
    Returns
    True if the array is a matrix, False otherwise.
    Return type
    bool
```

**cr.sparse.is_square**

```python
def is_square(A):
    # Checks if an array is a square matrix
    Parameters
    A (jax.numpy.ndarray) – A JAX array
    Returns
    True if the array is a square matrix, False otherwise.
    Return type
    bool
```

**cr.sparse.is_symmetric**

```python
def is_symmetric(A):
    # Checks if an array is a symmetric matrix
    Parameters
    A (jax.numpy.ndarray) – A JAX array
    Returns
    True if the array is a symmetric matrix, False otherwise.
    Return type
    bool
```

**cr.sparse.is_hermitian**

```python
def is_hermitian(A):
    # Checks if an array is a Hermitian matrix
    Parameters
    A (jax.numpy.ndarray) – A JAX array
    Returns
    True if the array is a Hermitian matrix, False otherwise.
    Return type
    bool
```

**cr.sparse.is_positive_definite**

```python
def is_positive_definite(A):
    # Checks if an array is a symmetric positive definite matrix
    Parameters
    A (jax.numpy.ndarray) – A JAX array
    Returns
    True if the array is a symmetric positive definite matrix, False otherwise.
    Return type
    bool
```

Symmetric positive definite matrices have real and positive eigenvalues. This function checks if all the eigenvalues are positive.
4.6. Utilities in cr.sparse module

4.6.2 Row wise and column wise norms for signal/representation matrices

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>norms_l1_cw(X)</td>
<td>Computes the $l_1$ norm of each column of a matrix</td>
</tr>
<tr>
<td>norms_l1_rw(X)</td>
<td>Computes the $l_1$ norm of each row of a matrix</td>
</tr>
<tr>
<td>norms_l2_cw(X)</td>
<td>Computes the $l_2$ norm of each column of a matrix</td>
</tr>
<tr>
<td>norms_l2_rw(X)</td>
<td>Computes the $l_2$ norm of each row of a matrix</td>
</tr>
<tr>
<td>norms_linf_cw(X)</td>
<td>Computes the $l_{\infty}$ norm of each column of a matrix</td>
</tr>
<tr>
<td>norms_linf_rw(X)</td>
<td>Computes the $l_{\infty}$ norm of each row of a matrix</td>
</tr>
<tr>
<td>sqr_norms_l2_cw(X)</td>
<td>Computes the squared $l_2$ norm of each column of a matrix</td>
</tr>
<tr>
<td>sqr_norms_l2_rw(X)</td>
<td>Computes the $l_2$ norm of each row of a matrix</td>
</tr>
</tbody>
</table>

continues on next page
## Table 24 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>normalize_11_cw(X)</code></td>
<td>Normalize each column of X per l_1-norm</td>
</tr>
<tr>
<td><code>normalize_11_rw(X)</code></td>
<td>Normalize each row of X per l_1-norm</td>
</tr>
<tr>
<td><code>normalize_12_cw(X)</code></td>
<td>Normalize each column of X per l_2-norm</td>
</tr>
<tr>
<td><code>normalize_12_rw(X)</code></td>
<td>Normalize each row of X per l_2-norm</td>
</tr>
</tbody>
</table>

**CR.Sparse.norms_l1_cw**

CR.Sparse.norms_l1_cw(X)

Computes the l_1 norm of each column of a matrix

**CR.Sparse.norms_l1_rw**

CR.Sparse.norms_l1_rw(X)

Computes the l_1 norm of each row of a matrix

**CR.Sparse.norms_l2_cw**

CR.Sparse.norms_l2_cw(X)

Computes the l_2 norm of each column of a matrix

**CR.Sparse.norms_l2_rw**

CR.Sparse.norms_l2_rw(X)

Computes the l_2 norm of each row of a matrix

**CR.Sparse.norms_linf_cw**

CR.Sparse.norms_linf_cw(X)

Computes the l_inf norm of each column of a matrix

**CR.Sparse.norms_linf_rw**

CR.Sparse.norms_linf_rw(X)

Computes the l_inf norm of each row of a matrix

**CR.Sparse.sqr_norms_l2_cw**

CR.Sparse.sqr_norms_l2_cw(X)

Computes the squared l_2 norm of each column of a matrix
cr.sparse.sqr_norms_l2_rw

```python
cr.sparse.sqr_norms_l2_rw(X)
```
Computes the $l_2$ norm of each row of a matrix

cr.sparse.normalize_l1_cw

```python
cr.sparse.normalize_l1_cw(X)
```
Normalize each column of $X$ per $l_1$-norm

4.6.3 Sparse representations

Following functions analyze or construct representation vectors which are known to be sparse.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nonzero_values(x)</code></td>
<td>Returns the values of non-zero entries in $x`</td>
</tr>
<tr>
<td><code>nonzero_indices(x)</code></td>
<td>Returns the indices of non-zero entries in $x`</td>
</tr>
<tr>
<td><code>randomize_rows(key, X)</code></td>
<td>Randomizes the rows in $X$</td>
</tr>
<tr>
<td><code>randomize_cols(key, X)</code></td>
<td>Randomizes the columns in $X$</td>
</tr>
<tr>
<td><code>largest_indices(x, K)</code></td>
<td>Returns the indices of $K$ largest entries in $x$ by magnitude</td>
</tr>
<tr>
<td><code>hard_threshold(x, K)</code></td>
<td>Returns the indices and corresponding values of largest $K$ non-zero entries in a vector $x`</td>
</tr>
<tr>
<td><code>hard_threshold_sorted(x, K)</code></td>
<td>Returns the sorted indices and corresponding values of largest $K$ non-zero entries in a vector $x`</td>
</tr>
<tr>
<td><code>sparse_approximation(x, K)</code></td>
<td>Keeps only largest $K$ non-zero entries by magnitude in a vector $x`</td>
</tr>
<tr>
<td><code>build_signal_from_indices_and_values(length, </code>...`)</td>
<td>Builds a sparse signal from its non-zero entries (specified by their indices and values)</td>
</tr>
<tr>
<td><code>dynamic_range(x)</code></td>
<td>Returns the ratio of largest and smallest values (by magnitude) in $x$ (dB)</td>
</tr>
<tr>
<td><code>nonzero_dynamic_range(x)</code></td>
<td>Returns the ratio of largest and smallest non-zero values (by magnitude) in $x$ (dB)</td>
</tr>
</tbody>
</table>

4.6. Utilities in cr.sparse module
CR.Sparse

**cr.sparse.nonzero_values**

```python
cr.sparse.nonzero_values(x)
```

Returns the values of non-zero entries in x

**cr.sparse.nonzero_indices**

```python
cr.sparse.nonzero_indices(x)
```

Returns the indices of non-zero entries in x

**cr.sparse.randomize_rows**

```python
cr.sparse.randomize_rows(key, X)
```

Randomizes the rows in X

**cr.sparse.randomize_cols**

```python
cr.sparse.randomize_cols(key, X)
```

Randomizes the columns in X

**cr.sparse.largest_indices**

```python
cr.sparse.largest_indices(x, K)
```

Returns the indices of K largest entries in x by magnitude

**cr.sparse.hard_threshold**

```python
cr.sparse.hard_threshold(x, K)
```

Returns the indices and corresponding values of largest K non-zero entries in a vector x

**cr.sparse.hard_threshold_sorted**

```python
cr.sparse.hard_threshold_sorted(x, K)
```

Returns the sorted indices and corresponding values of largest K non-zero entries in a vector x

**cr.sparse.sparse_approximation**

```python
cr.sparse.sparse_approximation(x, K)
```

Keeps only largest K non-zero entries by magnitude in a vector x
cr.sparse.build_signal_from_indices_and_values

**cr.sparse.build_signal_from_indices_and_values**(length, indices, values)
Builds a sparse signal from its non-zero entries (specified by their indices and values)

**cr.sparse.dynamic_range**

**cr.sparse.dynamic_range**(x)
Returns the ratio of largest and smallest values (by magnitude) in x (dB)

**cr.sparse.nonzero_dynamic_range**

**cr.sparse.nonzero_dynamic_range**(x)
Returns the ratio of largest and smallest non-zero values (by magnitude) in x (dB)

### Sparse representation matrices (row-wise)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>largest_indices_rw</strong>(X, K)</td>
<td>Returns the indices of K largest entries by magnitude in each row of X</td>
</tr>
<tr>
<td><strong>take_along_rows</strong>(X, indices)</td>
<td>Picks K entries from each row of X specified by indices matrix</td>
</tr>
<tr>
<td><strong>sparse_approximation_rw</strong>(X, K)</td>
<td>Keeps only largest K non-zero entries by magnitude in each row of X</td>
</tr>
</tbody>
</table>

**cr.sparse.largest_indices_rw**

**cr.sparse.largest_indices_rw**(X, K)
Returns the indices of K largest entries by magnitude in each row of X

**cr.sparse.take_along_rows**

**cr.sparse.take_along_rows**(X, indices)
Picks K entries from each row of X specified by indices matrix
CR.Sparse

**cr.sparse.sparse_approximation_rw**

**cr.sparse.sparse_approximation_rw**(\(X, K\))

Keeps only largest \(K\) non-zero entries by magnitude in each row of \(X\)

**Sparse representation matrices (column-wise)**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>largest_indices_cw(X, K)</code></td>
<td>Returns the indices of (K) largest entries by magnitude in each column of (X)</td>
</tr>
<tr>
<td><code>take_along_cols(X, indices)</code></td>
<td>Picks (K) entries from each column of (X) specified by indices matrix</td>
</tr>
<tr>
<td><code>sparse_approximation_cw(X, K)</code></td>
<td>Keeps only largest (K) non-zero entries by magnitude in each column of (X)</td>
</tr>
</tbody>
</table>

**cr.sparse.largest_indices_cw**

**cr.sparse.largest_indices_cw**(\(X, K\))

Returns the indices of \(K\) largest entries by magnitude in each column of \(X\)

**cr.sparse.take_along_cols**

**cr.sparse.take_along_cols**(\(X, indices\))

Picks \(K\) entries from each column of \(X\) specified by indices matrix

**cr.sparse.sparse_approximation_cw**

**cr.sparse.sparse_approximation_cw**(\(X, K\))

Keeps only largest \(K\) non-zero entries by magnitude in each column of \(X\)

### 4.7 Evaluation Framework

It is a set of tools to evaluate the performance of sparse recovery algorithms:

- Reconstruction quality of individual sparse recovery problems
- Success rates across multiple sparsity levels

### 4.8 Linear Algebra Subroutines

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>solve_Lx_b(L, b)</code></td>
<td>Solves the system (L x = b) using back substitution</td>
</tr>
<tr>
<td><code>solve_LTx_b(L, b)</code></td>
<td>Solves the system (L^T x = b) using back substitution</td>
</tr>
<tr>
<td><code>solve_Ux_b(U, b)</code></td>
<td>Solves the system (U x = b) using back substitution</td>
</tr>
<tr>
<td><code>solve_UTx_b(U, b)</code></td>
<td>Solves the system (U^T x = b) using back substitution</td>
</tr>
</tbody>
</table>

continues on next page
Table 29 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>solve_spd_chol(L, b)</td>
<td>Solves a symmetric positive definite system $A x = b$ where $A = L L'$</td>
</tr>
</tbody>
</table>

4.8.1 cr.sparse.la.solve_Lx_b

```python
cr.sparse.la.solve_Lx_b(L, b)
```
Solves the system $L \cdot x = b$ using back substitution

4.8.2 cr.sparse.la.solve_LTx_b

```python
cr.sparse.la.solve_LTx_b(L, b)
```
Solves the system $L^T \cdot x = b$ using back substitution

4.8.3 cr.sparse.la.solve_Ux_b

```python
cr.sparse.la.solve_Ux_b(U, b)
```
Solves the system $U \cdot x = b$ using back substitution

4.8.4 cr.sparse.la.solve_UTx_b

```python
cr.sparse.la.solve_UTx_b(U, b)
```
Solves the system $U^T \cdot x = b$ using back substitution

4.8.5 cr.sparse.la.solve_spd_chol

```python
cr.sparse.la.solve_spd_chol(L, b)
```
Solves a symmetric positive definite system $A \cdot x = b$ where $A = L \cdot L'$
5.1 Computation Time Comparison of Sparse Recovery Methods

5.1.1 Performance on CPU

System configuration

- MacBook Pro 2019 Model
- Processor: 1.4 GHz Quad Core Intel Core i5
- Memory: 8 GB 2133 MHz LPDDR3

Problem Specification

- Gaussian sensing matrices (normalized to unit norm columns)
- Sparse vectors with non-zero entries drawn from Gaussian distributions
- M, N, K have been chosen so that all algorithms under comparison are known to converge to successful recovery.

Remarks

- All algorithms have been benchmarked for both 32-bit and 64-bit floating point calculations. Benchmarks are separately presented for them.
- It was separately verified that sparse recovery results were identical for both with or without JIT acceleration.
- Python %timeit magic was used for benchmarking.
- Every algorithm has been run several times on the given problem and the average time has been computed.
- Average times have been reported in jit_off and jit_on columns with milliseconds units.
Algorithm structures

The table below highlights the differences in the structure of different algorithms under consideration. These differences are key reason for the computational complexity.

<table>
<thead>
<tr>
<th>Method</th>
<th>Correlation with residual</th>
<th>Least squares</th>
<th>Hard thresholding</th>
<th>Step size</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP</td>
<td>Yes</td>
<td>Cholesky update</td>
<td>1 atom</td>
<td>No</td>
</tr>
<tr>
<td>SP</td>
<td>Yes</td>
<td>2 (2K atoms and K atoms)</td>
<td>K atoms and K atoms</td>
<td>No</td>
</tr>
<tr>
<td>CoSaMP</td>
<td>Yes</td>
<td>1 (3K atoms)</td>
<td>2K atoms and K atoms</td>
<td>No</td>
</tr>
<tr>
<td>IHT</td>
<td>Yes</td>
<td>0</td>
<td>K atoms</td>
<td>Dynamic</td>
</tr>
<tr>
<td>NIHT</td>
<td>Yes</td>
<td>0</td>
<td>K atoms</td>
<td>Fixed</td>
</tr>
<tr>
<td>HTP</td>
<td>Yes</td>
<td>1 (K atoms)</td>
<td>K atoms</td>
<td>Fixed</td>
</tr>
<tr>
<td>NHTP</td>
<td>Yes</td>
<td>1 (K atoms)</td>
<td>K atoms</td>
<td>Dynamic</td>
</tr>
</tbody>
</table>

Benchmarks for 32-bit

<table>
<thead>
<tr>
<th>Method</th>
<th>M</th>
<th>N</th>
<th>K</th>
<th>Iterations</th>
<th>jit_off</th>
<th>jit_on</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP</td>
<td>200</td>
<td>1000</td>
<td>20</td>
<td>20</td>
<td>105.78</td>
<td>2.14</td>
<td>49.48</td>
</tr>
<tr>
<td>SP</td>
<td>200</td>
<td>1000</td>
<td>20</td>
<td>3</td>
<td>1645.32</td>
<td>2.73</td>
<td>602.34</td>
</tr>
<tr>
<td>CoSaMP</td>
<td>200</td>
<td>1000</td>
<td>20</td>
<td>4</td>
<td>309.01</td>
<td>6.20</td>
<td>49.84</td>
</tr>
<tr>
<td>IHT</td>
<td>200</td>
<td>1000</td>
<td>20</td>
<td>65</td>
<td>232.99</td>
<td>36.27</td>
<td>6.42</td>
</tr>
<tr>
<td>NIHT</td>
<td>200</td>
<td>1000</td>
<td>20</td>
<td>16</td>
<td>240.96</td>
<td>5.64</td>
<td>42.72</td>
</tr>
<tr>
<td>HTP</td>
<td>200</td>
<td>1000</td>
<td>20</td>
<td>5</td>
<td>1491.00</td>
<td>13.71</td>
<td>108.76</td>
</tr>
<tr>
<td>NHTP</td>
<td>200</td>
<td>1000</td>
<td>20</td>
<td>4</td>
<td>1467.35</td>
<td>1.98</td>
<td>741.88</td>
</tr>
</tbody>
</table>

Benchmarks for 64-bit

<table>
<thead>
<tr>
<th>Method</th>
<th>M</th>
<th>N</th>
<th>K</th>
<th>Iterations</th>
<th>jit_off</th>
<th>jit_on</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP</td>
<td>200</td>
<td>1000</td>
<td>20</td>
<td>20</td>
<td>112.69</td>
<td>2.43</td>
<td>46.42</td>
</tr>
<tr>
<td>SP</td>
<td>200</td>
<td>1000</td>
<td>20</td>
<td>4</td>
<td>1324.79</td>
<td>4.49</td>
<td>295.02</td>
</tr>
<tr>
<td>CoSaMP</td>
<td>200</td>
<td>1000</td>
<td>20</td>
<td>5</td>
<td>293.50</td>
<td>9.82</td>
<td>29.90</td>
</tr>
<tr>
<td>IHT</td>
<td>200</td>
<td>1000</td>
<td>20</td>
<td>77</td>
<td>209.22</td>
<td>48.81</td>
<td>4.29</td>
</tr>
<tr>
<td>NIHT</td>
<td>200</td>
<td>1000</td>
<td>20</td>
<td>19</td>
<td>196.66</td>
<td>7.23</td>
<td>27.21</td>
</tr>
<tr>
<td>HTP</td>
<td>200</td>
<td>1000</td>
<td>20</td>
<td>6</td>
<td>1218.62</td>
<td>18.96</td>
<td>64.28</td>
</tr>
<tr>
<td>NHTP</td>
<td>200</td>
<td>1000</td>
<td>20</td>
<td>5</td>
<td>1238.37</td>
<td>2.79</td>
<td>443.68</td>
</tr>
</tbody>
</table>
5.1.2 Subjective Analysis

64-bit vs 32-bit

- There are differences in number of iterations for convergence
- Every algorithm except OMP takes more iterations to converge with 64-bit compared to 32-bit floating point computations.
- In case of OMP, number of iterations is decided by sparsity. Hence, it is same for both 32-bit and 64-bit.
- It was separately established that success rates of these algorithms suffers somewhat for 32-bit floating point calculations.
- In other words, 32-bit computations are more aggressive and may be inaccurate.
- On CPUs, the floating point units are 64-bit. Hence, using 32-bit floating point computations doesn’t give us much speedup. 32-bit computation would be more relevant for GPUs.
- The general trend of computation times (with JIT on) for both 32-bit and 64-bit are similar. i.e. algorithms which are slower for 32-bit are slower for 64-bit too.

Rest of the discussion is focused on the results for 64-bit sparse recovery. .. rubric:: All algorithms without JIT vs with JIT

- It is clear that all algorithms exhibit significant speedups with the introduction of JIT acceleration.
- The speedup is as low as 4x for IHT and as high as 443x in NHTP.
- Before JIT, OMP is the fastest algorithm and SP is the slowest.
- After JIT acceleration, OMP is the fastest algorithm while IHT is the slowest. NHTP comes as a close second. Incidentally, NHTP is faster than OMP for 32-bit.
- NHTP and SP show significant speedups with JIT. HTP, OMP, CoSaMP and NIHT show modest gains. IHT doesn’t seem to provide much optimization opportunities.
- It appears that steps like dynamic step size computation (in NIHT, NHTP) and least squares (in SP, CoSaMP, HTP, NHTP) tend to get aggressively optimized and lead to massive speed gains.

OMP

- With JIT on, OMP is actually one of the fastest algorithms in the mix (for both 32-bit and 64-bit).
- In the current implementations, OMP is the only one in which the least squares step has been optimized using Cholesky updates.
- This is possible as OMP structure allows for adding atoms one at a time to the mix.
- Other algorithms change several atoms [add / remove] in each iteration. Hence, such optimizations are not possible.
- The least squares steps in other algorithms can be accelerated using small number of conjugate gradients iterations. However, this hasn’t been implemented yet.
SP vs CoSaMP

- CoSaMP has one least squares step (on 3K indices) in each iteration.
- SP (Subspace Pursuit) has two least squares steps in each iteration.
- Without JIT, CoSaMP is 4x faster.
- With JIT, SP becomes 2x faster than CoSaMP.
- Thus, SP seems to provide more aggressive optimization opportunities.

IHT vs NIHT

- IHT and NIHT are both simple algorithms. They don’t involve a least squares step in their iterations.
- The main difference is that the step-size fixed for IHT and it is computed on every iteration in NIHT.
- The dynamic step size leads to reduction in the number of iterations for NIHT. From 77 to 19, 4x reduction.
- Without JIT, there is no significant difference between IHT and NIHT. Thus, step-size computation seems to contribute a lot to computation time without acceleration.
- With JIT, step-size computation seems to be aggressively optimized. NIHT after JIT is 6x faster than IHT even though the number of iterations reduces by only 4 times and there is extra overhead of computing the step size. This appears to be counter-intuitive.

IHT vs HTP

- The major difference in the two algorithms is that HTP performs a least squares estimate on the current guess of signal support.
- The number of iterations reduces 13 times due to the least squares step but it has its own extra overhead.
- Without JIT, HTP becomes much slower than IHT (6x slower). Thus, overhead of a least squares step is quite high.
- HTP is about 3x faster than IHT with JIT. This makes sense. The number of iterations reduced by 13 times and the overhead of least squares was added.

HTP vs NHTP

- Just like NIHT, NHTP also introduces computing the step size dynamically in every iteration.
- It helps in reducing the number of iterations from 6 to 5.
- In this case, the benefit of dynamic step size is not visible much in terms of iterations.
- Without JIT, NHTP is somewhat slower than HTP.
- However, with JIT, NHTP is 6x faster than HTP. This speedup is unusual as there is just 20% reduction in number of iterations and there is the overhead of step size computation.
5.2 Orthogonal Matching Pursuit

Speed benchmarks for JAX implementation

Each row of the following table describes:

- problem type and configuration (M x N is dictionary size, K is sparsity level)
- Average time taken in CPU/GPU configurations
- Speed improvement ratios

System used

- All benchmarks have been generated on Google Colab
- CPU and GPU configurations Google Colab have been used

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>K</th>
<th>CPU</th>
<th>CPU + JIT</th>
<th>CPU / CPU + JIT</th>
<th>GPU</th>
<th>GPU + JIT</th>
<th>GPU / GPU + JIT</th>
<th>CPU + JIT / GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>1024</td>
<td>16</td>
<td>148 ms</td>
<td>8.27 ms</td>
<td>17.9x</td>
<td>139 ms</td>
<td>1.28 ms</td>
<td>108x</td>
<td>6.46x</td>
</tr>
</tbody>
</table>

Observations

- JIT (Just In Time) compilation seems to give significant performance improvements in both CPU and GPU architectures
- Current implementation seems to be slower on GPU vs CPU with JIT.
- GPU speed gain over CPU (with JIT on) is relatively meager. On TensorFlow, people regularly report 30x improvements between CPU to GPU for neural networks implemented using Keras.

Possible deficiencies

- There is opportunity to improve parallelization in the OMP implementation.
- Cholesky update based implements depends heavily on solving triangular systems.
- GPUs may not be great at solving triangular systems.
CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search
C

`cr.sparse.data`, 55
CR.Sparse
<table>
<thead>
<tr>
<th>Index</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>abs_max_idx() (in module cr.sparse.pursuit), 51</td>
</tr>
<tr>
<td></td>
<td>add() (in module cr.sparse.lop), 42</td>
</tr>
<tr>
<td>B</td>
<td>babel() (in module cr.sparse.dict), 36</td>
</tr>
<tr>
<td></td>
<td>build_signal_from_indices_and_values() (in module cr.sparse), 61</td>
</tr>
<tr>
<td>C</td>
<td>coherence() (in module cr.sparse.dict), 35</td>
</tr>
<tr>
<td></td>
<td>coherence_with_index() (in module cr.sparse.dict), 35</td>
</tr>
<tr>
<td></td>
<td>column() (in module cr.sparse.lop), 43</td>
</tr>
<tr>
<td></td>
<td>columns() (in module cr.sparse.lop), 43</td>
</tr>
<tr>
<td></td>
<td>compose() (in module cr.sparse.lop), 42</td>
</tr>
<tr>
<td></td>
<td>cr.sparse.data module, 55</td>
</tr>
<tr>
<td>D</td>
<td>dct_basis() (in module cr.sparse.dict), 34</td>
</tr>
<tr>
<td></td>
<td>diagonal() (in module cr.sparse.lop), 39</td>
</tr>
<tr>
<td></td>
<td>dirac_dct_basis() (in module cr.sparse.dict), 34</td>
</tr>
<tr>
<td></td>
<td>dirac_fourier_basis_ld() (in module cr.sparse.lop), 40</td>
</tr>
<tr>
<td></td>
<td>dirac_hadamard_basis() (in module cr.sparse.dict), 34</td>
</tr>
<tr>
<td></td>
<td>dirac_hadamard_dct_basis() (in module cr.sparse.dict), 34</td>
</tr>
<tr>
<td></td>
<td>dynamic_range() (in module cr.sparse), 61</td>
</tr>
<tr>
<td>F</td>
<td>flipud() (in module cr.sparse.lop), 39</td>
</tr>
<tr>
<td></td>
<td>fourier_basis() (in module cr.sparse.dict), 35</td>
</tr>
<tr>
<td></td>
<td>fourier_basis_ld() (in module cr.sparse.lop), 40</td>
</tr>
<tr>
<td></td>
<td>frame() (in module cr.sparse.dict), 35</td>
</tr>
<tr>
<td></td>
<td>frame_bounds() (in module cr.sparse.dict), 36</td>
</tr>
<tr>
<td>G</td>
<td>gaussian_dict() (in module cr.sparse.lop), 41</td>
</tr>
<tr>
<td></td>
<td>gaussian_mtx() (in module cr.sparse.dict), 33</td>
</tr>
<tr>
<td></td>
<td>gram() (in module cr.sparse.dict), 35</td>
</tr>
<tr>
<td></td>
<td>gram_chol_update() (in module cr.sparse.pursuit), 51</td>
</tr>
<tr>
<td>H</td>
<td>hadamard() (in module cr.sparse.dict), 34</td>
</tr>
<tr>
<td></td>
<td>hadamard_basis() (in module cr.sparse.dict), 34</td>
</tr>
<tr>
<td></td>
<td>hard_threshold() (in module cr.sparse), 60</td>
</tr>
<tr>
<td></td>
<td>hard_threshold_sorted() (in module cr.sparse), 60</td>
</tr>
<tr>
<td></td>
<td>has_orthogonal_columns() (in module cr.sparse), 57</td>
</tr>
<tr>
<td></td>
<td>has_orthogonal_rows() (in module cr.sparse), 57</td>
</tr>
<tr>
<td></td>
<td>has_unitary_columns() (in module cr.sparse), 57</td>
</tr>
<tr>
<td></td>
<td>has_unitary_rows() (in module cr.sparse), 57</td>
</tr>
<tr>
<td></td>
<td>hermitian() (in module cr.sparse), 55</td>
</tr>
<tr>
<td></td>
<td>hermitian() (in module cr.sparse.lop), 42</td>
</tr>
<tr>
<td>I</td>
<td>identity() (in module cr.sparse.lop), 39</td>
</tr>
<tr>
<td></td>
<td>is_hermitian() (in module cr.sparse), 56</td>
</tr>
<tr>
<td></td>
<td>is_matrix() (in module cr.sparse), 56</td>
</tr>
<tr>
<td></td>
<td>is_positive_definite() (in module cr.sparse), 56</td>
</tr>
<tr>
<td></td>
<td>is_square() (in module cr.sparse), 56</td>
</tr>
<tr>
<td></td>
<td>is_symmetric() (in module cr.sparse), 56</td>
</tr>
<tr>
<td>J</td>
<td>jit() (in module cr.sparse.lop), 44</td>
</tr>
<tr>
<td>L</td>
<td>largest_indices() (in module cr.sparse), 60</td>
</tr>
<tr>
<td></td>
<td>largest_indices_cw() (in module cr.sparse), 62</td>
</tr>
<tr>
<td></td>
<td>largest_indices_rw() (in module cr.sparse), 61</td>
</tr>
<tr>
<td></td>
<td>lower_frame_bound() (in module cr.sparse.dict), 36</td>
</tr>
<tr>
<td>M</td>
<td>matching_atoms_ratio() (in module cr.sparse.dict), 37</td>
</tr>
<tr>
<td></td>
<td>matrix() (in module cr.sparse.lop), 39</td>
</tr>
<tr>
<td></td>
<td>matrix_solve() (in module cr.sparse.pursuit.cosamp), 46</td>
</tr>
<tr>
<td></td>
<td>matrix_solve() (in module cr.sparse.pursuit.htp), 50</td>
</tr>
<tr>
<td></td>
<td>matrix_solve() (in module cr.sparse.pursuit.iht), 49</td>
</tr>
</tbody>
</table>
CR.Sparse

matrix_solve() (in module cr.sparse.pursuit.omp), 45
matrix_solve() (in module cr.sparse.pursuit.sp), 48
matrix_solve_jit() (in module cr.sparse.pursuit.cosamp), 47
matrix_solve() (in module cr.sparse.pursuit.omp), 50
matrix_solve_jit() (in module cr.sparse.pursuit.cosamp), 47
matrix_solve() (in module cr.sparse.pursuit.iht), 49
matrix_solve_jit() (in module cr.sparse.pursuit.omp), 45
matrix_solve() (in module cr.sparse.pursuit.sp), 48
matrix_solve_multi() (in module cr.sparse.pursuit.omp), 46

module
  cr.sparse.data, 55
  mutual_coherence() (in module cr.sparse.dict), 37
  mutual_coherence_with_index() (in module cr.sparse.dict), 36

N
  neg() (in module cr.sparse.omp), 42
  nonzero_dynamic_range() (in module cr.sparse), 61
  nonzero_indices() (in module cr.sparse), 60
  nonzero_values() (in module cr.sparse), 60
  normalize_l1_cw() (in module cr.sparse), 59
  normalize_l1_rw() (in module cr.sparse), 59
  normalize_l2_cw() (in module cr.sparse), 59
  normalize_l2_rw() (in module cr.sparse), 59
  norms_l1_cw() (in module cr.sparse), 58
  norms_l1_rw() (in module cr.sparse), 58
  norms_l2_cw() (in module cr.sparse), 58
  norms_l2_rw() (in module cr.sparse), 58
  norms_linf_cw() (in module cr.sparse), 58
  norms_linf_rw() (in module cr.sparse), 58

O
  Operator (class in cr.sparse.omp), 38
  operator_solve() (in module cr.sparse.pursuit.cosamp), 47
  operator_solve() (in module cr.sparse.pursuit.omp), 50
  operator_solve() (in module cr.sparse.pursuit.iht), 49
  operator_solve() (in module cr.sparse.pursuit.sp), 48
  operator_solve_jit() (in module cr.sparse.pursuit.cosamp), 47
  operator_solve_jit() (in module cr.sparse.pursuit.omp), 50
  operator_solve_jit() (in module cr.sparse.pursuit.iht), 49
  operator_solve_jit() (in module cr.sparse.pursuit.sp), 48

P
  pad_zeros() (in module cr.sparse.omp), 39

power() (in module cr.sparse.omp), 43

R
  rademacher_dict() (in module cr.sparse.omp), 41
  rademacher_mtx() (in module cr.sparse.dict), 34
  random_onb() (in module cr.sparse.dict), 34
  random_onb_dict() (in module cr.sparse.omp), 41
  random_orthonormal_rows_dict() (in module cr.sparse.omp), 41
  randomize_cols() (in module cr.sparse), 60
  randomize_rows() (in module cr.sparse), 60
  real() (in module cr.sparse.omp), 41
  RecoverySolution (class in cr.sparse.pursuit), 50
  restriction() (in module cr.sparse.omp), 40

S
  scale() (in module cr.sparse.omp), 42
  solve() (in module cr.sparse.cvx.adm.yall1), 53
  solve() (in module cr.sparse.pursuit.cosamp), 46
  solve() (in module cr.sparse.pursuit.omp), 49
  solve() (in module cr.sparse.pursuit.iht), 48
  solve() (in module cr.sparse.pursuit.sp), 47
  solve_bp() (in module cr.sparse.cvx.adm.yall1), 54
  solve_bp_jit() (in module cr.sparse.cvx.adm.yall1), 54
  solve_l1_l2() (in module cr.sparse.cvx.adm.yall1), 54
  solve_l1_l2_jit() (in module cr.sparse.cvx.adm.yall1), 54
  solve_l1_l2con() (in module cr.sparse.cvx.adm.yall1), 54
  solve_l1_l2con_jit() (in module cr.sparse.cvx.adm.yall1), 54
  solve_LTx_b() (in module cr.sparse.la), 63
  solve_Lx_b() (in module cr.sparse.la), 63
  solve_spd_chol() (in module cr.sparse.la), 63
  solve_UTx_b() (in module cr.sparse.la), 63
  solve_Ux_b() (in module cr.sparse.la), 63
  sparse_approximation() (in module cr.sparse), 60
  sparse_approximation_cw() (in module cr.sparse), 62
  sparse_approximation_rw() (in module cr.sparse), 62
  sgr_norms_l2_cw() (in module cr.sparse), 58
  sgr_norms_l2_rw() (in module cr.sparse), 59
  subtract() (in module cr.sparse.omp), 42
  sum() (in module cr.sparse.omp), 39
  symmetrize() (in module cr.sparse.omp), 40

T
  take_along_cols() (in module cr.sparse), 62
  take_along_rows() (in module cr.sparse), 61
  to_adjoint_matrix() (in module cr.sparse.omp), 44
to_complex_matrix() (in module cr.sparse.lop), 44

transposed() (in module cr.sparse.lop), 42

to_matrix() (in module cr.sparse.lop), 44

transpose() (in module cr.sparse), 55

transpose() (in module cr.sparse.lop), 42

U

upper_frame_bound() (in module cr.sparse.dict), 36
upper_frame_bound() (in module cr.sparse.lop), 43

Z

zero() (in module cr.sparse.lop), 39