modred Documentation

Release 2.1.0

Brandt Belson, Jonathan Tu, Clancy Rowley

May 04, 2021

Contents

1 Introduction **3**

2 Why modred? **5**

3 Installation and requirements **7**

4 Tutorial **9**
- 4.1 Modal decompositions – POD, BPOD, and DMD . . . 9
- 4.2 Model reduction . . . 17
- 4.3 System identification – OKID and ERA . . . 20

5 Interfacing with your data **21**
- 5.1 Vector objects . . . 21
- 5.2 Vector handles . . . 22
- 5.3 Checking requirements automatically . . . 23
- 5.4 How vector objects and handles are used in modred . . . 23
- 5.5 Example . . . 23
- 5.6 Summary and next steps . . . 24

6 More examples **27**
- 6.1 Complex Ginzburg-Landau equation . . . 27
- 6.2 References . . . 27

7 POD **29**

8 BPOD **33**

9 DMD **37**

10 LTIGalerkinProjection **51**

11 ERA **55**

12 OKID **59**

13 Vectors Module **61**

14 VectorSpace **63**

15 Parallel Class **67**

16 Utility Functions **69**

17 Release notes **75**
17.1 modred 2.1.0 . . . 75
17.2 modred 2.0.4 . . . 76
17.3 modred 2.0.3 . . . 76
17.4 modred 2.0.2 . . . 77
17.5 modred 2.0.1 . . . 77
17.6 modred 2.0.0 . . . 77
17.7 modred 1.0.2 . . . 79
17.8 modred 1.0.1 . . . 80
17.9 modred 1.0.0 . . . 80
17.10 modred 0.3.2 . . . 81
17.11 modred 0.3.1 . . . 81
17.12 modred 0.3.0 . . . 82
17.13 modred 0.2.1 . . . 82
17.14 modred 0.2.0 . . . 82

18 Indices and tables **83**

Bibliography **85**

Python Module Index **87**

Index **89**

Contents:

CHAPTER 1

Introduction

Welcome to the modred project!

This is an easy-to-use and parallelized library for computing modal decompositions and reduced-order models.

Parallel implementations of the proper orthogonal decomposition (POD), balanced POD (BPOD), dynamic mode decomposition (DMD), and Petrov-Galerkin projection are provided, as well as serial implementations of the Observer Kalman filter Identification method (OKID) and the Eigensystem Realization Algorithm (ERA). modred is applicable to a wide range of problems and to nearly any type of data.

For smaller and simpler datasets, there is a Matlab-like interface. For larger and more complicated datasets, you can provide modred with classes and functions to interact with your data.

This work was supported by grants from the the National Science Foundation (NSF) and the Air Force Office of Scientific Research (AFOSR).

CHAPTER 2

Why modred?

modred's a good choice for beginners, experts, experimentalists, and computationalists from many fields. The main advantages of modred are summarized below. If you don't know Python, it's a terrific programming language with similarities to Matlab.

Ease of use

For smaller and simpler data, often only a single function call with a Matlab-like interface is needed. For larger and more complicated data, there is a high-level object-oriented interface. The code is written-to-be-read, open source, and well documented. In almost all cases, modred can be run in parallel (MPI) with *no changes* to the code.

Several algorithms included

Parallel implementations of the proper orthogonal decomposition (POD), balanced POD (BPOD), dynamic mode decomposition (DMD), and Petrov-Galerkin projection are provided, as well as serial implementations of the Observer Kalman Filter Identification method (OKID) and the Eigensystem Realization Algorithm (ERA). It is easy to switch between methods.

modred can be easily extended to other methods you might like to use.

Applicable to your data

For the common case of data stacked in arrays, there are simple, Matlab-like, functions to use. For larger and more complicated data, you can provide classes and functions that interface with your data format. These functions should be written with no parallel consideration; modred does the parallelization for you. It is also possible to call existing functions in other languages such as C/C++, Fortran, Java, and Matlab with tools like Cython, SWIG, f2py, and mlabwrap, thus eliminating the need to translate existing code into Python.

Computational speed

The library efficiently orchestrates calls to numpy functions and/or functions that you provide, with little added overhead. If Python's speed limitations become problematic, they can be bypassed by calling compiled code using tools like Cython, SWIG, and f2py.

Further, it is parallelized for a distributed memory architecture using MPI and the mpi4py module. The scaling of speedup/processors is better-than-linear up to several hundred processors, if not more.

Certain methods, such as the ERA and OKID, are typically not computationally demanding and are thus only implemented in serial.

Reliable

Each individual function is unit tested independently and thoroughly, making modred results trustworthy. Furthermore, modred has already been used to analyze and model a variety of complicated, real-world datasets, with great success.

Limitations

The biggest limitation is for datasets so large that it is impossible to have three vector objects in one node's memory at the same time. By design, modred's parallelization scheme doesn't break up individual pieces of data for you, i.e. it doesn't do domain decomposition for you. However, modred could be extended so that you can provide parallelized functions, allowing arbitrarily large data. If you're curious about this, contact Brandt Belson and Jonathan Tu at modred-discuss@googlegroups.com. For now, a work-around is to use "fat" nodes with large amounts of memory.

CHAPTER 3

Installation and requirements

Mandatory:

1. Python 2 (>2.6, tested for 2.7) or Python 3 (>=3.3), http://python.org.
2. Relatively new version of Numpy (>1.6, tested for 1.10), http://numpy.scipy.org.

Optional:

1. For parallel execution, an MPI implementation and mpi4py, http://mpi4py.scipy.org.

To install:

```
[sudo] python setup.py install
```

To be sure it's working, run the unit tests. The parallel tests require mpi4py to be installed:

```
python -c 'import modred.tests; modred.tests.run()'
mpiexec -n 3 python -c 'import modred.tests; modred.tests.run()'
```

Please report problems to modred-discuss@googlegroups.com with the following:

1. Copy of the entire output of the installation and tests
2. Python version (`python -V`)
3. Numpy version (`python -c 'import numpy; print numpy.__version__'`)
4. Your operating system

The documentation is available at http://modred.readthedocs.io.

While unnecessary, you can build the documentation with Sphinx (http://sphinx.pocoo.org). From the modred directory, run `sphinx-build doc doc/build` and then open `doc/build/index.html` in a web browser.

CHAPTER 4

Tutorial

4.1 Modal decompositions – POD, BPOD, and DMD

This tutorial discusses computing modes from data, using the Proper Orthogonal Decomposition (POD), Balanced Proper Orthogonal Decomposition (BPOD), and Dynamic Mode Decomposition (DMD). For details of these algorithms, see [HLBR] for POD and BPOD and [TRLBK] for DMD.

The first step is to collect your data. We call each piece of data a "vector" or "vector object". **By vector, we don't mean a 1D array.** Rather, we mean an element of a vector space. This could be a 1D, 2D, or 3D array, or any other object that satisfies the properties of a vector space. The examples below build on one another, each introducing new aspects.

4.1.1 Example 1 – All data in an array

A simple way to find POD modes is:

```
import numpy as np

import modred as mr


# Create random data
num_vecs = 30
vecs = np.random.random((100, num_vecs))

# Compute POD
num_modes = 5
POD_res = mr.compute_POD_arrays_snaps_method(
    vecs, list(mr.range(num_modes)))
modes = POD_res.modes
eigvals = POD_res.eigvals
```

Let's walk through the important steps. First, we create an array of random data. Each column is a vector represented as a 1D array. Then we call the function `compute_POD_arrays_snaps_method`, which returns the first `num_modes` modes as columns of the array `modes`, and all of the non-zero eigenvalues, sorted from largest to smallest.

This function implements the "method of snapshots", as described in Section 3.4 of [HLBR]. In short, it computes the correlation array X^*X, where X is `vecs`, then finds its eigenvectors and eigenvalues, which are related to the modes.

4.1.2 Example 2 – Inner products with all data in an array

You can use a weighted inner product, specified here by a 1D array of weights, so that the correlation array is X^*WX, where X is `vecs` and W contains the inner product weights. The weights also can, more generally, be an array. The vectors are again represented as columns of an array.

```
import numpy as np

import modred as mr


# Create random data
num_vecs = 100
nx = 100
vecs = np.random.random((nx, num_vecs))

# Define non-uniform grid and corresponding inner product weights
x_grid = 1. - np.cos(np.linspace(0, np.pi, nx))
x_diff = np.diff(x_grid)
weights = 0.5 * np.append(np.append(
    x_diff[0], x_diff[:-1] + x_diff[1:]), x_diff[-1])

# Compute POD
num_modes = 10
POD_res = mr.compute_POD_arrays_direct_method(
    vecs, list(mr.range(num_modes)), inner_product_weights=weights)
modes = POD_res.modes
eigvals = POD_res.eigvals
```

This function computes the singular value decomposition (SVD) of $W^{1/2}X$, and we refer to this as the "direct method" to distinguish it from the method of snapshots in the previous example. The differences between the two are insignificant in most cases. For details, see `pod.compute_POD_arrays_direct_method()`.

4.1.3 Example 3 – Vector handles for loading and saving

This example demonstrates *vector handles*, which are very important because they allow modred to interact with large and complicated datasets without requiring that all of the vectors be stacked into a single array. This is necessary, for example, if the data is too large to all fit in memory simultaneously.

The following example computes direct and adjoint modes using Balanced POD (see Chapter 5 of [HLBR]):

```
import os

import numpy as np

import modred as mr

```

(continues on next page)

(continued from previous page)

```
# Create directory for output files
out_dir = 'tutorial_ex3_out'
if not os.path.isdir(out_dir):
    os.makedirs(out_dir)

# Define handles for snapshots
num_vecs = 30
direct_snapshots = [
    mr.VecHandleArrayText('%s/direct_vec%d.txt' % (out_dir, i))
    for i in mr.range(num_vecs)]
adjoint_snapshots = [
    mr.VecHandleArrayText('%s/adjoint_vec%d.txt' % (out_dir, i))
    for i in mr.range(num_vecs)]

# Save random snapshot data in text files
x = np.linspace(0, np.pi, 200)
for i, snap in enumerate(direct_snapshots):
    snap.put(np.sin(x * i))
for i, snap in enumerate(adjoint_snapshots):
    snap.put(np.cos(0.5 * x * i))

# Calculate and save BPOD modes
my_BPOD = mr.BPODHandles(inner_product=np.vdot, max_vecs_per_node=10)
sing_vals, L_sing_vecs, R_sing_vecs = my_BPOD.compute_decomp(
    direct_snapshots, adjoint_snapshots)
num_modes = 10
mode_nums = list(mr.range(num_modes))
direct_modes = [
    mr.VecHandleArrayText('%s/direct_mode%d' % (out_dir, i))
    for i in mode_nums]
adjoint_modes = [
    mr.VecHandleArrayText('%s/adjoint_mode%d' % (out_dir, i))
    for i in mode_nums]
my_BPOD.compute_direct_modes(mode_nums, direct_modes)
my_BPOD.compute_adjoint_modes(mode_nums, adjoint_modes)
```

First, we create lists `direct_snapshots` and `adjoint_snapshots`. ("Snapshots" is just a common word to describe vectors that come from time-sampling a system as it evolves.) Each element in these lists is a vector handle (in particular, an instance of `VecHandleArrayText`). All vector handles have member functions to load, `vec = vec_handle.get()`, and save, `vec_handle.put(vec)`, but themselves use very little memory because they do *not* internally contain a vector. modred uses these vector handles to load and save individual vectors only as it needs them.

All the snapshots are vectors and each is represented as an array, but they are *not* stacked into a single 2D array. Ordinarily the snapshots would already exist, for instance as files from a simulation or experiment. Here, we artificially generate snapshots and write them to file using the `put()` method.

Next, we compute the BPOD modes. The `BPODHandles` constructor takes an optional argument `max_vecs_per_node=10`, specifying that only 10 vectors (snapshots + modes) can be in one node's memory at one time. The function `compute_decomp` takes lists of *vector handles* as arguments. In this example, note that there are 30 direct and 30 adjoint snapshots, so handles are necessary to avoid violating `max_vecs_per_node=10`.

Similarly, `compute_direct_modes` and `compute_adjoint_modes` take lists of handles. The modes are saved via `handle.put()` rather than returned, which might require too much memory.

Replacing `VecHandleArrayText` with `VecHandlePickle` would load/save all vectors (snapshots and/or modes) to Python's binary pickle files. Note that pickling works with *any* type of vector, including user-defined

ones, whereas saving to text is only written for 1D and 2D arrays.

To run this example in parallel is easy. The only complication is the data must be saved by only one processor, and moving these lines inside an `if` block solves this:

```
parallel = mr.parallel_default_instance
if parallel.is_rank_zero():
    # Loops that call handles.put
    pass
parallel.barrier()
```

After this change, the code will still work in serial, even if mpi4py is not installed. To run this, where the above script is saved as `main_bpod.py`, execute:

```
mpiexec -n 8 python main_bpod.py
```

It is rare to need to handle parallelization yourself, but if you do, you should use the provided `mr.parallel_default_instance` instance as above. Also provided are member functions `parallel.get_rank()` and `parallel.get_num_procs()` (see `parallel` for details).

If you're curious, the text files are saved in a format defined in `util.load_array_text()` and `util.save_array_text()`.

4.1.4 Example 4 – Inner product function

You are free to use any inner product function, as long as it has the interface `value = inner_product(vec1, vec2)` and satisfies the mathematical definition of an inner product (see *Interfacing with your data*). This example uses the trapezoidal rule for inner products on an arbitrary n-dimensional cartesian grid (see `vectors.InnerProductTrapz`). The object `weighted_IP` is callable (it has a special method `__call__`) so it acts as the inner product the usual way: `value = weighted_IP(vec1, vec2)`.

```
import os

import numpy as np

from modred import parallel
import modred as mr


# Create directory for output files
out_dir = 'tutorial_ex4_out'
if not os.path.isdir(out_dir):
    os.makedirs(out_dir)

# Define non-uniform grid and corresponding inner product weights
nx = 80
ny = 100
x_grid = 1. - np.cos(np.linspace(0, np.pi, nx))
y_grid = np.linspace(0, 1., ny) ** 2
Y, X = np.meshgrid(y_grid, x_grid)

# Create random snapshot data
num_vecs = 100
snapshots = [
    mr.VecHandlePickle('%s/vec%d.pkl' % (out_dir, i))
    for i in mr.range(num_vecs)]
if parallel.is_rank_zero():
```

(continues on next page)

(continued from previous page)

```
    for i, snap in enumerate(snapshots):
        snap.put(np.sin(X * i) + np.cos(Y * i))
parallel.barrier()

# Calculate DMD modes and save them to pickle files
weighted_IP = mr.InnerProductTrapz(x_grid, y_grid)
my_DMD = mr.DMDHandles(inner_product=weighted_IP)
my_DMD.compute_decomp(snapshots)
my_DMD.put_decomp(
    '%s/eigvals.txt' % out_dir, '%s/R_low_order_eigvecs.txt' % out_dir,
    '%s/L_low_order_eigvecs.txt' % out_dir,
    '%s/correlation_array_eigvals.txt' % out_dir,
    '%s/correlation_array_eigvecs.txt' % out_dir)
mode_indices = [1, 4, 5, 0, 10]
modes = [
    mr.VecHandlePickle('%s/mode%d.pkl' % (out_dir, i)) for i in mode_indices]
my_DMD.compute_exact_modes(mode_indices, modes)
```

Also shown in this example is the useful `put_decomp`, which, by default saves the arrays associated with the decomposition to text files. (See *Array input and output*.)

Again, this code can be executed in parallel without any modifications.

4.1.5 Example 5 – Shifting and scaling vectors using vector handles

Often vectors contain an offset (also called a shift or translation) such as a mean or equilibrium state, and one might want to do model reduction with this known offset removed. We call this offset the base vector, and it can be subtracted off by the vector handle class as shown in this example.

Note that `handle.put` does *not* use the base vector; the base vector is only subtracted by `handle.get`.

You might also want to scale all of your vectors by factors as you retrieve them for use in modred, and this can also be done by the `handle.get` function. When using both base vector shifting and scaling, the default order is first shifting then scaling: `(vec - base_vec)*scale`.

This examples uses quadrature weights, where each vector is weighted. It also shows how to load vectors in one format (pickle, via `mr.VecHandlePickle`) and save modes in another (text, via `mr.VecHandleArrayText`).

```
import os

import numpy as np

from modred import parallel
import modred as mr


# Create directory for output files
out_dir = 'tutorial_ex5_out'
if not os.path.isdir(out_dir):
    os.makedirs(out_dir)

# Create artificial sample times used as quadrature weights in POD
num_vecs = 100
quad_weights = np.logspace(1., 3., num=num_vecs)
base_vec_handle = mr.VecHandlePickle('%s/base_vec.pkl' % out_dir)
snapshots = [
```

(continues on next page)

(continued from previous page)

```
    mr.VecHandlePickle(
        '%s/vec%d.pkl' % (out_dir, i),base_vec_handle=base_vec_handle,
        scale=quad_weights[i])
    for i in mr.range(num_vecs)]

# Save arbitrary snapshot data
num_elements = 2000
if parallel.is_rank_zero():
    for snap in snapshots + [base_vec_handle]:
        snap.put(np.random.random(num_elements))
parallel.barrier()

# Compute and save POD modes
my_POD = mr.PODHandles(inner_product=np.vdot)
my_POD.compute_decomp(snapshots)
my_POD.put_decomp('%s/sing_vals.txt' % out_dir, '%s/sing_vecs.txt' % out_dir)
my_POD.put_correlation_array('%s/correlation_array.txt' % out_dir)
mode_indices = [1, 4, 5, 0, 10]
modes = [
    mr.VecHandleArrayText('%s/mode%d.txt' % (out_dir, i)) for i in mode_indices]
my_POD.compute_modes(mode_indices, modes)

# Check that modes are orthonormal
vec_space = mr.VectorSpaceHandles(inner_product=np.vdot)
IP_array = vec_space.compute_symm_inner_product_array(modes)
if not np.allclose(IP_array, np.eye(len(mode_indices))):
    print('Warning: modes are not orthonormal')
    print(IP_array)
```

At the end of this example, we use an instance of the low-level class `vectorspace.VectorSpaceHandles` to check that the POD modes are orthonormal. It's generally not necessary to use this class (or do this check), but if the need arises, it should be used (instead of writing new code) since it is tested and parallelized.

4.1.6 Example 6 – User-defined vectors and handles

So far, all of the vectors have been arrays, but you may want to apply modred to data saved in your own custom format with more complicated inner products and other operations. This is no problem at all; modred works with data in any format! That's worth saying again: **modred works with data in any format!** Of course, you'll have to tell modred how to interact with your data, but that's pretty easy. You just need to define and use your own vector handle and vector objects.

There are two important new features of this example: a custom vector class `CustomVector` and a custom vector handle class `CustomVecHandle`. These definitions may be collected together in a file, for instance called `customvector.py`:

```
import pickle
from copy import deepcopy

import numpy as np

import modred as mr


class CustomVector(mr.Vector):
    def __init__(self, grids, data_array):
```

(continues on next page)

(continued from previous page)

```
        self.grids = grids
        self.data_array = data_array
        self.weighted_ip = mr.InnerProductTrapz(*self.grids)


    def __add__(self, other):
        """Return a new object that is the sum of self and other"""
        sum_vec = deepcopy(self)
        sum_vec.data_array = self.data_array + other.data_array
        return sum_vec


    def __mul__(self, scalar):
        """Return a new object that is ``self * scalar`` """
        mult_vec = deepcopy(self)
        mult_vec.data_array = mult_vec.data_array * scalar
        return mult_vec


    def inner_product(self, other):
        return self.weighted_ip(self.data_array, other.data_array)


class CustomVecHandle(mr.VecHandle):
    def __init__(self, vec_path, base_handle=None, scale=None):
        mr.VecHandle.__init__(self, base_handle, scale)
        self.vec_path = vec_path


    def _get(self):
        file_id = open(self.vec_path, 'rb')
        grids = pickle.load(file_id)
        data_array = pickle.load(file_id)
        file_id.close()
        return CustomVector(grids, data_array)


    def _put(self, vec):
        file_id = open(self.vec_path, 'wb')
        pickle.dump(vec.grids, file_id)
        pickle.dump(vec.data_array, file_id)
        file_id.close()


def inner_product(v1, v2):
    return v1.inner_product(v2)
```

Instances of `CustomVector` meet the requirements for a vector object: vector addition `__add__` and scalar multiplication `__mul__` are defined, and the objects are compatible with an inner product function such as `inner_product(v1, v2)`. Note that `CustomVector` inherits from a base class `mr.Vector`. This is not required, but is recommended, as the base class provides some useful additional methods. The member function `inner_product` is useful, but not required. This example also uses the trapezoidal rule for inner products to account for a 3D arbitrary cartesian grid (`vectors.InnerProductTrapz`).

The class `CustomVecHandle` inherits from a base class `mr.VecHandle`, and defines methods `_get` and `_put`, which load and save vectors from/to Pickle files. Note the leading underscore: the functions `get` and `put` (without leading underscore) are defined in the `VecHandle` base class, and take care of scaling or subtracting a base vector.

The methods _get and _put defined here are in turn called by the base class (the "template method" design pattern). While you don't need to understand the guts of these base classes, more is covered in *Interfacing with your data*.

Here's an example using these classes:

```
import os

import numpy as np

from modred import parallel
import modred as mr
from customvector import CustomVector, CustomVecHandle, inner_product


# Create directory for output files
out_dir = 'tutorial_ex6_out'
if not os.path.isdir(out_dir):
    os.makedirs(out_dir)

# Define snapshot handles
direct_snapshots = [
    CustomVecHandle('%s/direct_snap%d.pkl' % (out_dir, i), scale=np.pi)
    for i in mr.range(10)]
adjoint_snapshots = [
    CustomVecHandle('%s/adjoint_snap%d.pkl' % (out_dir, i), scale=np.pi)
    for i in mr.range(10)]

# Create random snapshot data
nx = 50
ny = 30
nz = 20
x = np.linspace(0, 1, nx)
y = np.logspace(1, 2, ny)
z = np.linspace(0, 1, nz) ** 2
if parallel.is_rank_zero():
    for snap in direct_snapshots + adjoint_snapshots:
        snap.put(CustomVector([x, y, z], np.random.random((nx, ny, nz))))
parallel.barrier()

# Compute and save balanced POD modes
my_BPOD = mr.BPODHandles(inner_product=inner_product)
my_BPOD.sanity_check(direct_snapshots[0])
sing_vals, L_sing_vecs, R_sing_vecs = my_BPOD.compute_decomp(
    direct_snapshots, adjoint_snapshots)

# Choose modes so that BPOD projection has less than 10% error
sing_vals_norm = sing_vals / np.sum(sing_vals)
num_modes = np.nonzero(np.cumsum(sing_vals_norm) > 0.9)[0][0] + 1
mode_nums = list(mr.range(num_modes))
direct_modes = [
    CustomVecHandle('%s/direct_mode%d.pkl' % (out_dir, i)) for i in mode_nums]
adjoint_modes = [
    CustomVecHandle('%s/adjoint_mode%d.pkl' % (out_dir, i)) for i in mode_nums]
my_BPOD.compute_direct_modes(mode_nums, direct_modes)
my_BPOD.compute_adjoint_modes(mode_nums, adjoint_modes)
```

This example is similar to previous ones, but some aspects are worth pointing out. The call to `my_BPOD.sanity_check()` runs some basic tests to verify that the vector handles and vectors behave as expected, so this can be useful for debugging. After execution, the modes are saved to `direct_mode0.pkl`, `direct_mode1.pkl` ...

and `adjoint_mode0.pkl`, `adjoint_mode1.pkl`. Also, note that the only time the `CustomVector` class is used is in generating the "fake" random data: most scripts will only need to deal with the vector handle classes, not the vectors themselves.

When you're ready to start using modred, take a look at what types of vectors, file formats, and inner products are supplied in `vectors`. If you don't find what you need, you can define your own vectors and vector handles following examples like this. Built-in classes like `VecHandleArrayText` and `VecHandlePickle` are common cases and serve as good examples since your own custom vector handle classes will probably resemble them.

As usual, this example can be executed in parallel without any modifications.

4.1.7 Array input and output

By default, `put_array` and `get_array` save and load to text files. If you prefer a different format, you can pass your own functions as keyword arguments `put_array` and `get_array` to the constructors. The functions should have the following interfaces: `put_array(array, array_dest)` and `array = get_array(array_source)`. The `array` argument could be a 1D or 2D array.

4.1.8 References

4.2 Model reduction

4.2.1 Linear reduced-order models

We provide a class to find reduced-order models (continuous and discrete time) by projecting linear dynamics onto modes (Petrov-Galerkin projection). The governing equation of the full system is assumed to be either continuous time:

$$\partial x(t)/\partial t = Ax(t) + Bu(t)$$
$$y(t) = Cx(t)$$

or discrete time:

$$x(k+1) = Ax(k) + Bu(k)$$
$$y(k) = Cx(k)$$

where k is the time step. Here x is the state vector. A, B, and C are, in general, linear operators (often arrays). In cases where there are no inputs and outputs, B and C are zero. A acts on x and returns a vector that lives in the same vector space as x. B acts on elements of the input space, $\mathbb{R}^p$, where p is the number of inputs and returns elements of the vector space in which x lives. C acts on x and returns elements of the output space, $\mathbb{R}^q$, where q is the number of outputs.

These dynamical equations can be projected onto a set of modes. First, approximate the state vector as a linear combination of r modes, stacked as columns of array Φ, and time-varying coefficients $q(k)$:

$$x(k) \approx \Phi q(k).$$

Then substitute into the governing equations and take the inner product with a set of adjoint modes, columns of array Ψ. The result is a reduced system for q, which has as many elements as Φ has columns, r. The adjoint, $(\ )^+$, is defined

with respect to inner product weight W.

$$
\begin{aligned}
q(k+1) &= A_r q(k) + B_r u(k) \\
y(k) &= C_r q(k) \\
&\text{where} \\
A_r &= (\Psi^+ \Phi)^{-1} \Psi^+ A \Phi \\
B_r &= (\Psi^+ \Phi)^{-1} \Psi^+ B \\
C_r &= C \Phi
\end{aligned}
$$

An analagous result exists for continuous time.

If the modes are not stacked into arrays, then the following equations are used, where $[\ ]_{i,j}$ denotes row i and column j.

$$
\begin{aligned}
[\Psi^+ \Phi]_{i,j} &= \langle \psi_i, \phi_j \rangle_W \\
[\Psi^+ A \Phi]_{i,j} &= \langle \psi_i, A\phi_j \rangle_W \\
[\Psi^+ B] &= \langle \psi_i, Be_j \rangle_W \\
[C\Phi]_{:,j} &= C\phi_j
\end{aligned}
$$

e_j is the jth standard basis (intuitively Be_j is $[B]_{:,j}$ if B is an array.).

The A, B, and C operators may or may not be available within Python. For example, you may do simulations using code written in another language. For this reason, modred requires only the *action* of the operators on the vectors, i.e., the products $A\phi_j$, Be_j, and $C\phi_j$, and *not* the operators A, B, and C themselves.

Example 1: Smaller data and arrays

Here's an example that uses arrays.

```
import os

import numpy as np

import modred as mr


# Create directory for output files
out_dir = 'rom_ex1_out'
if not os.path.isdir(out_dir):
    os.makedirs(out_dir)

# Create random LTI system
nx = 100
num_inputs = 2
num_outputs = 3
num_basis_vecs = 10
A = np.random.random((nx, nx))
B = np.random.random((nx, num_inputs))
C = np.random.random((num_outputs, nx))

# Create random modes
basis_vecs = np.random.random((nx, num_basis_vecs))

# Perform Galerkin projection and save data
LTI_proj = mr.LTIGalerkinProjectionArrays(basis_vecs)
```

(continues on next page)

(continued from previous page)

```
A_reduced, B_reduced, C_reduced = LTI_proj.compute_model(
    A.dot(basis_vecs), B, C.dot(basis_vecs))
LTI_proj.put_model(
    '%s/A_reduced.txt' % out_dir, '%s/B_reduced.txt' % out_dir,
    '%s/C_reduced.txt' % out_dir)
```

The array `basis_vecs` contains the vectors that define the basis onto which the dynamics are projected.

A few variations of this are illustrative. First, if no inputs or outputs exist, then there is only A_r and no B_r or C_r. The last two lines would then be replaced with:

```
A_reduced = LTI_proj.reduce_A(A.dot(basis_vec_array))
LTI_proj.put_A_reduced('A_reduced.txt')
```

Another variant is if the basis vecs are known to be orthonormal, as is always the case with POD modes. Then, the $\Psi^* W \Phi$ array and its inverse are identity, and computing it is wasteful. Specifying the constructor keyword argument `is_basis_orthonormal=True` tells modred this array is identity and to not compute it.

Example 2: Larger data and vector handles

Here's an example similar to what might arise when doing large simulations in another language or program.

```
import os

import numpy as np

from modred import parallel
import modred as mr


# Create directory for output files
out_dir = 'rom_ex2_out'
if not os.path.isdir(out_dir):
    os.makedirs(out_dir)

# Create handles for modes
num_modes = 10
basis_vecs = [
    mr.VecHandlePickle('%s/dir_mode_%02d.pkl' % (out_dir, i))
    for i in mr.range(num_modes)]
adjoint_basis_vecs = [
    mr.VecHandlePickle('%s/adj_mode_%02d.pkl' % (out_dir, i))
    for i in mr.range(num_modes)]

# Define system dimensions and create handles for action on modes
num_inputs = 2
num_outputs = 4
A_on_basis_vecs = [
    mr.VecHandlePickle('%s/A_on_dir_mode_%02d.pkl' % (out_dir, i))
    for i in mr.range(num_modes)]
B_on_bases = [
    mr.VecHandlePickle('%s/B_on_basis_%02d.pkl' % (out_dir, i))
    for i in mr.range(num_inputs)]
C_on_basis_vecs = [
    np.sin(np.linspace(0, 0.1 * i, num_outputs)) for i in mr.range(num_modes)]
```

(continues on next page)

(continued from previous page)

```
# Create random modes and action on modes. Typically this data already exists
# and this section is unnecesary.
nx = 100
ny = 200
if parallel.is_rank_zero():
    for handle in (
        basis_vecs + adjoint_basis_vecs + A_on_basis_vecs + B_on_bases):
        handle.put(np.random.random((nx, ny)))
parallel.barrier()

# Perform Galerkin projection and save data to disk
inner_product = np.vdot
LTI_proj = mr.LTIGalerkinProjectionHandles(
    inner_product, basis_vecs, adjoint_basis_vec_handles=adjoint_basis_vecs)
A_reduced, B_reduced, C_reduced = LTI_proj.compute_model(
    A_on_basis_vecs, B_on_bases, C_on_basis_vecs)
LTI_proj.put_model(
    '%s/A_reduced.txt' % out_dir, '%s/B_reduced.txt' % out_dir,
    '%s/C_reduced.txt' % out_dir)
```

This example works in parallel with no modifications.

If you do not have the time-derivatives of the direct modes but want a continuous time model, see `ltigalerkinproj.compute_derivs_arrays()` and `ltigalerkinproj.compute_derivs_handles()`.

4.2.2 Eigensystem Realization Algorithm (ERA)

See the documentation and examples provided in `era.compute_ERA_model()` and `era.ERA`.

4.3 System identification – OKID and ERA

See `okid` and `era`.

CHAPTER 5

Interfacing with your data

The simplest way to use modred is with the Matlab-like functions and 2D arrays. However, sometimes your data is too large for this. In these cases, there is a high-level object-oriented interface that works with data in any format and never needs the data stacked into a 2D array. Of course, you'll need to tell modred how to interact with your data. This section explains how to do this and provides some mathematical background.

5.1 Vector objects

The building block of the modal decompositions is the vector object. Sets of these vector objects are decomposed into modes by POD, BPOD, and DMD. Others call these vector objects snapshots, planes of spatial data, fields, time histories, and many other names. Within modred, "vector" refers to the object you, the user, use to represent your data. **By "vector", we do not mean a 1D array**. We do mean an element of a vector space (technically an inner product space). You are free to choose *any* object, from numpy arrays to your own class, so long as it satisfies a few simple requirements.

The vector object must:

1. Support scalar multiplication, i.e. `vector2 = 2.0*vector1`.
2. Support addition with other vectors, i.e. `vector3 = vector1 + vector2`.
3. Be compatible with a user-supplied `inner_product(vector1, vector2)` function.

Numpy arrays already meet requirements 1 and 2. For your own classes, define the special methods `__mul__` and `__add__` for 1 and 2.

You also need an inner product function that takes two vectors and returns a single number. This number can be real or complex, but may not switch from real to complex depending on the input, i.e., it must be real for all inputs or complex for all inputs. Your inner product must satisfy the mathematical definition for an inner product:

- Conjugate symmetry: `inner_product(vec1, vec2) == numpy.conj(inner_product(vec2, vec1))`.
- Linearity: `inner_product(vec1, scalar*vec2) == scalar*inner_product(vec1, vec2)`.

- Implied norm: `inner_product(vec, vec) >= 0` with equality if and only if `vec` is the zero vector.

The two examples we show are numpy's `vdot` and the trapezoidal rule in `vectors.InnerProductTrapz`. It's often a good idea to define an inner product function as a member function of the vector class, and write a simple wrapper. There is an example of this in the tutorial.

The resulting modes are also vectors. We mean this in both the programming sense that modes are vector objects and the mathematical sense that modes live in the same vector space as vectors.

5.1.1 Base class

We provide a useful base class for all user-defined vectors to inherit from, `mr.Vector`. It isn't required to inherit from it, but encouraged because it defines a few useful special functions and has some error checking. If you're curious, take a look at it in the `vectors` module (click on the [source] link on the right side).

5.2 Vector handles

When the vectors are large, it can be inefficient or impossible to have all of them in memory simultaneously. Thus, modred only needs a subset of vectors in memory, loading and saving them as necessary. Therefore, you can provide it with a list of *vector handles*. These are *lightweight* objects that in some sense point to a vector's location, like the filename where it's saved. In general, vector handles get a vector from a location and return it, and also put a vector in a location. That is, they implement this interface:

- Constructor with interface `VectorHandle(location)`.
- A get function with interface `vec = vec_handle.get()`.
- A put function with interface `vec_handle.put(vec)`.

An example would be a constructor that takes a file name as an argument, a `get` that loads and returns the vector, and a `put` that saves the vector to the file name.

One can think of `get` as loading, but it is more general because `get` can retrieve the vector from anywhere (though most often from file). Similarly, one can think of `put` as saving, but it is more general because `put` can send the vector anywhere (though most often to file).

It's natural to think of a vector handle's `get` and `put` as inverses, but they don't have to be. For example, it's acceptable to load an input vector from one file format and save modes to another file format. However, it does mean that if one wanted to load the modes, one couldn't with this vector handle because `get` assumes a different file format.

Another way to handle the case of different input vector and mode (or any output vector) file formats is to define a different vector handle class for each. In this case, technically one wouldn't need a `put` for the input vector handle since one never saves to this format. Similarly, one only needs to write a `get` for the mode vector handle if one wants to load the modes (for example to plot them).

It's very important that the vector handles actually be lightweight (use little memory). modred is most efficient when it uses all of the memory available to have as many vectors in memory as possible. So if vector handles contain vectors or other large data, then modred could run slowly or stop with "out of memory" errors.

5.2.1 Base class

We provide a useful base class for all user-defined vector handles to inherit from. An example of a user-defined vector handle that inherits from `mr.VecHandle` is provided in the tutorial. This isn't required, but strongly encouraged because it contains extra functionality. The `mr.VecHandle` constructor accepts two additional arguments, a base vector handle `base_handle` and a scaling factor `scale`. This allows the `get` function to retrieve a vector, subtract from it a base vector (for example an equilibrium or mean state), scale it (for example by a quadrature weight), and

return the modified vector. The base class achieves this via a `get` that calls the derived class's member function `_get` and performs the additional operations for base vectors and/or scaling. The base class's `put` simply calls `_put` of the derived class. Examples are shown in the tutorial.

One might be concerned that the base class is reloading the base vector at each call of `get`, but this is avoidable. As long as the `base_handle` you give each vector handle instance is equal (with respect to ==), then the base vector is loaded on the first call of `get` and stored as `mr.VecHandle.cached_base_vec`, which is used by all instances of classes derived from `mr.VecHandle`.

If you're curious, feel free to take a look at it in the `vectors` module (click on the [source] link on the right side).

5.3 Checking requirements automatically

First, we encourage you to write your own tests (see module `unittest`) to be sure your vector object and vector handle work as you expect. Many classes provide a function `sanity_check` that checks a few common mistakes in your vector object addition, scalar multiplication, and inner products. *We encourage you to run* `sanity_check` *every time you use modred.* We used to call this the `idiot_check` as motivation to use it; keep that in mind!

5.4 How vector objects and handles are used in modred

The classes `POD`, `BPOD`, and `DMD` have similar interfaces which interact with vectors and vector handles. First, each has `compute_decomp` functions that take lists of vector handles, `vec_handles`, as arguments. Within the `compute_decomp` functions, `vec = vec_handle.get()` is called repeatedly to retrieve vectors as needed. In fact, `compute_decomp` does not "know" or "care" what's inside the vector handles and vectors; only that they satisfy the requirements.

More information about these methods is provided in the documentation for each class.

5.5 Example

An example of a custom class for vectors and vector handles is shown below:

```
import pickle
from copy import deepcopy

import numpy as np

import modred as mr


class CustomVector(mr.Vector):
    def __init__(self, grids, data_array):
        self.grids = grids
        self.data_array = data_array
        self.weighted_ip = mr.InnerProductTrapz(*self.grids)


    def __add__(self, other):
        """Return a new object that is the sum of self and other"""
        sum_vec = deepcopy(self)
        sum_vec.data_array = self.data_array + other.data_array
```

(continues on next page)

(continued from previous page)

```
        return sum_vec


    def __mul__(self, scalar):
        """Return a new object that is ``self * scalar`` """
        mult_vec = deepcopy(self)
        mult_vec.data_array = mult_vec.data_array * scalar
        return mult_vec


    def inner_product(self, other):
        return self.weighted_ip(self.data_array, other.data_array)


class CustomVecHandle(mr.VecHandle):
    def __init__(self, vec_path, base_handle=None, scale=None):
        mr.VecHandle.__init__(self, base_handle, scale)
        self.vec_path = vec_path


    def _get(self):
        file_id = open(self.vec_path, 'rb')
        grids = pickle.load(file_id)
        data_array = pickle.load(file_id)
        file_id.close()
        return CustomVector(grids, data_array)


    def _put(self, vec):
        file_id = open(self.vec_path, 'wb')
        pickle.dump(vec.grids, file_id)
        pickle.dump(vec.data_array, file_id)
        file_id.close()


def inner_product(v1, v2):
    return v1.inner_product(v2)
```

For an example using this class, see the tutorial in *Modal decompositions – POD, BPOD, and DMD*.

5.6 Summary and next steps

Summarizing, to use modred on arbitrary data, define

1. A vector object that has:
    1. Vector addition ("+", `__add__`),
    2. Scalar multiplication ("*", `__mul__`),
    3. Optional: inherits from `vectors.Vector`.
2. A function `inner_product(vec1, vec2)`.
3. A vector handle class that has:
    1. Member function `get()` which returns a vector handle.

2. Member function `put(vec)` where `vec` is a vector handle.
3. Optionally inherits from `vectors.VecHandle`. If so, member function names in 1 and 2 change to `_get` and `_put`.

Then you can get started using any of the modal decomposition classes! Before writing your own classes, check out `vectors`, which has several common vector and vector handles classes.

For large data, Python's speed limitations can be bypassed by implementing functions in compiled languages such as C/C++ and Fortran and accessing them within python with Cython, SWIG, f2py, etc.

CHAPTER 6

More examples

6.1 Complex Ginzburg-Landau equation

The example `examples/main_CGL.py` reproduces the computation of the BPOD modes of the linearized Complex Ginzburg-Landau equation, as in [Ilak2010]. The system is relatively small, and so all of the vectors are stacked into matrices.

6.2 References

CHAPTER 7

POD

class modred.pod.**PODHandles**(*inner_product=None*, *get_array=<function load_array_text>*, *put_array=<function save_array_text>*, *max_vecs_per_node=None*, *verbosity=1*)

Proper Orthogonal Decomposition implemented for large datasets.

Kwargs: `inner_product`: Function that computes inner product of two vector objects.

`put_array`: Function to put an array out of modred, e.g., write it to file.

`get_array`: Function to get an array into modred, e.g., load it from file.

`max_vecs_per_node`: Maximum number of vectors that can be stored in memory, per node.

`verbosity`: 1 prints progress and warnings, 0 prints almost nothing.

Computes POD modes from vector objects (or handles). Uses `vectorspace.VectorSpaceHandles` for low level functions.

Usage:

```
myPOD = POD(inner_product=my_inner_product)
myPOD.compute_decomp(vec_handles)
myPOD.compute_modes(range(10), modes)
```

See also *compute_POD_arrays_snaps_method()*, *compute_POD_arrays_direct_method()*, and `vectors`.

compute_decomp(*vec_handles*, *atol=1e-13*, *rtol=None*)

Computes correlation array X^*WX and its eigendecomposition.

Args: `vec_handles`: List of handles for vector objects.

Kwargs: `atol`: Level below which eigenvalues of correlation array are truncated.

`rtol`: Maximum relative difference between largest and smallest eigenvalues of correlation array. Smaller ones are truncated.

Returns: `eigvals`: 1D array of eigenvalues of correlation array.

eigvecs: Array whose columns are eigenvectors of correlation array.

compute_eigendecomp(*atol=1e-13*, *rtol=None*)
Computes eigendecomposition of correlation array.

Kwargs: atol: Level below which eigenvalues of correlation array are truncated.

rtol: Maximum relative difference between largest and smallest eigenvalues of correlation array. Smaller ones are truncated.

Useful if you already have the correlation array and to want to avoid recomputing it.

Usage:

```
POD.correlation_array = pre_existing_correlation_array
POD.compute_eigendecomp()
POD.compute_modes(range(10), mode_handles, vec_handles=vec_handles)
```

compute_modes(*mode_indices*, *mode_handles*, *vec_handles=None*)
Computes POD modes and calls put on them using mode handles.

Args: mode_indices: List of indices describing which modes to compute, e.g. range(10) or [3, 0, 5].

mode_handles: List of handles for modes to compute.

Kwargs: vec_handles: List of handles for vector objects. Optional if when calling *compute_decomp()*.

compute_proj_coeffs()
Computes orthogonal projection of vector objects onto POD modes.

Returns: proj_coeffs: Array of projection coefficients for vector objects, expressed as a linear combination of POD modes. Columns correspond to vector objects, rows correspond to POD modes.

get_correlation_array(*src*)
Gets the correlation array from source (memory or file).

Args: src: Source from which to retrieve correlation array.

get_decomp(*eigvals_src*, *eigvecs_src*)
Gets the decomposition arrays from sources (memory or file).

Args: eigvals_src: Source from which to retrieve eigenvalues of correlation array.

eigvecs_src: Source from which to retrieve eigenvectors of correlation array.

get_proj_coeffs(*src*)
Gets projection coefficients from source (memory or file).

Args: src: Source from which to retrieve projection coefficients.

put_correlation_array(*dest*)
Puts correlation array to dest.

put_decomp(*eigvals_dest*, *eigvecs_dest*)
Puts the decomposition arrays in destinations (memory or file).

Args: eigvals_dest: Destination in which to put eigenvalues of correlation array.

eigvecs_dest: Destination in which to put the eigenvectors of correlation array.

put_eigvals(*dest*)
Puts eigenvalues of correlation array to dest.

put_eigvecs(*dest*)
Puts eigenvectors of correlation array to `dest`.

put_proj_coeffs(*dest*)
Puts projection coefficients to `dest`

sanity_check(*test_vec_handle*)
Checks that user-supplied vector handle and vector satisfy requirements.

Args: `test_vec_handle`: A vector handle to test.

See `vectorspace.VectorSpaceHandles.sanity_check()`.

modred.pod.**compute_POD_arrays_direct_method**(*vecs*, *mode_indices=None*, *inner_product_weights=None*, *atol=1e-13*, *rtol=None*)
Computes POD modes using data stored in an array, using direct method.

Args: `vecs`: Array whose columns are data vectors (X).

Kwargs: `mode_indices`: List of indices describing which modes to compute. Examples are `range(10)` or `[3, 0, 6, 8]`. If no mode indices are specified, then all modes will be computed.

`inner_product_weights`: 1D or 2D array of inner product weights. Corresponds to W in inner product $v_1^* W v_2$.

`atol`: Level below which eigenvalues of correlation array are truncated.

`rtol`: Maximum relative difference between largest and smallest eigenvalues of correlation array. Smaller ones are truncated.

`return_all`: Return more objects; see below. Default is false.

Returns: `res`: Results of POD computation, stored in a namedtuple with the following attributes:

- `eigvals`: 1D array of eigenvalues of correlation array (E).
- `modes`: Array whose columns are POD modes.
- `proj_coeffs`: Array of projection coefficients for vector objects, expressed as a linear combination of POD modes. Columns correspond to vector objects, rows correspond to POD modes.
- `eigvecs`: Array wholse columns are eigenvectors of correlation array (U).

Attributes can be accessed using calls like `res.modes`. To see all available attributes, use `print(res)`.

The algorithm is

1. SVD $USV^* = W^{1/2}X$
2. Modes are $W^{-1/2}U$

where X, W, S, V, correspond to `vecs`, `inner_product_weights`, `eigvals**0.5`, and `eigvecs`, respectively.

Since this method does not square the vectors and singular values, it is more accurate than taking the eigendecomposition of the correlation array X^*WX, as in the method of snapshots (*compute_POD_arrays_snaps_method()*). However, this method is slower when X has more rows than columns, i.e. there are fewer vectors than elements in each vector.

modred.pod.**compute_POD_arrays_snaps_method**(*vecs*, *mode_indices=None*, *inner_product_weights=None*, *atol=1e-13*, *rtol=None*)
Computes POD modes using data stored in an array, using the method of snapshots.

Args: `vecs`: Array whose columns are data vectors (X).

Kwargs: `mode_indices`: List of indices describing which modes to compute. Examples are `range(10)` or `[3, 0, 6, 8]`. If no mode indices are specified, then all modes will be computed.

`inner_product_weights`: 1D or 2D array of inner product weights. Corresponds to W in inner product $v_1^* W v_2$.

`atol`: Level below which eigenvalues of correlation array are truncated.

`rtol`: Maximum relative difference between largest and smallest eigenvalues of correlation array. Smaller ones are truncated.

Returns: `res`: Results of POD computation, stored in a namedtuple with the following attributes:

- `eigvals`: 1D array of eigenvalues of correlation array (E).
- `modes`: Array whose columns are POD modes.
- `proj_coeffs`: Array of projection coefficients for vector objects, expressed as a linear combination of POD modes. Columns correspond to vector objects, rows correspond to POD modes.
- `eigvecs`: Array wholse columns are eigenvectors of correlation array (U).
- `correlation_array`: Correlation array ($X^* W X$).

Attributes can be accessed using calls like `res.modes`. To see all available attributes, use `print(res)`.

The algorithm is

1. Solve eigenvalue problem $X^* W X U = U E$
2. Coefficient array $T = U E^{-1/2}$
3. Modes are XT

where X, W, $X^* W X$, and T correspond to `vecs`, `inner_product_weights`, `correlation_array`, and `build_coeffs`, respectively.

Since this method "squares" the vector array and thus its singular values, it is slightly less accurate than taking the SVD of X directly, as in *`compute_POD_arrays_direct_method()`*. However, this method is faster when X has more rows than columns, i.e. there are more elements in each vector than there are vectors.

CHAPTER 8

BPOD

class modred.bpod.**BPODHandles**(*inner_product=None*, *put_array=<function save_array_text>*, *get_array=<function load_array_text>*, *max_vecs_per_node=None*, *verbosity=1*)

Balanced Proper Orthogonal Decomposition implemented for large datasets.

Kwargs: `inner_product`: Function that computes inner product of two vector objects.

`put_array`: Function to put an array out of modred, e.g., write it to file.

`get_array`: Function to get an array into modred, e.g., load it from file.

`max_vecs_per_node`: Maximum number of vectors that can be stored in memory, per node.

`verbosity`: 1 prints progress and warnings, 0 prints almost nothing.

Computes direct and adjoint BPOD modes from direct and adjoint vector objects (or handles). Uses `vectorspace.VectorSpaceHandles` for low level functions.

Usage:

```
myBPOD = BPODHandles(inner_product=my_inner_product)
myBPOD.compute_decomp(direct_vec_handles, adjoint_vec_handles)
myBPOD.compute_direct_modes(range(50), direct_modes)
myBPOD.compute_adjoint_modes(range(50), adjoint_modes)
```

See also *compute_BPOD_arrays()* and `vectors`.

compute_SVD(*atol=1e-13*, *rtol=None*)

Computes singular value decomposition of the Hankel array.

Kwargs:

`atol`: Level below which Hankel singular values are truncated.

`rtol`: Maximum relative difference between largest and smallest Hankel singular values. Smaller ones are truncated.

Useful if you already have the Hankel array and want to avoid recomputing it.

Usage:

```
my_BPOD.Hankel_array = pre_existing_Hankel_array
my_BPOD.compute_SVD()
my_BPOD.compute_direct_modes(
    range(10), mode_handles, direct_vec_handles=direct_vec_handles)
```

compute_adjoint_modes(*mode_indices*, *mode_handles*, *adjoint_vec_handles=None*)
Computes adjoint BPOD modes and calls `put` on them using mode handles.

Args: `mode_indices`: List of indices describing which adjoint modes to compute, e.g. `range(10)` or `[3, 0, 5]`.

`mode_handles`: List of handles for adjoint modes to compute.

Kwargs: `adjoint_vec_handles`: List of handles for adjoint vector objects. Optional if given when calling *compute_decomp()*.

compute_adjoint_proj_coeffs()
Computes biorthogonal projection of adjoint vector objects onto adjoint BPOD modes, using direct BPOD modes.

Returns: `adjoint_proj_coeffs`: Array of projection coefficients for adjoint vector objects, expressed as a linear combination of adjoint BPOD modes. Columns correspond to adjoint vector objects, rows correspond to adjoint BPOD modes.

compute_decomp(*direct_vec_handles*, *adjoint_vec_handles*, *num_inputs=1*, *num_outputs=1*, *atol=1e-13*, *rtol=None*)
Computes Hankel array $H = Y^* X$ and its singular value decomposition $UEV^* = H$.

Args: `direct_vec_handles`: List of handles for direct vector objects (X). They should be stacked so that if there are p inputs, the first p handles should all correspond to the same timestep. For instance, these are often all initial conditions of p different impulse responses.

`adjoint_vec_handles`: List of handles for adjoint vector objects (Y). They should be stacked so that if there are a outputs, the first q handles should all correspond to the same timestep. For instance, these are often all initial conditions of p different adjointimpulse responses.

Kwargs: `num_inputs`: Number of inputs to the system.

`num_outputs`: Number of outputs of the system.

`atol`: Level below which Hankel singular values are truncated.

`rtol`: Maximum relative difference between largest and smallest Hankel singular values. Smaller ones are truncated.

Returns: `sing_vals`: 1D array of Hankel singular values (E).

`L_sing_vecs`: Array of left singular vectors of Hankel array (U).

`R_sing_vecs`: Array of right singular vectors of Hankel array (V).

compute_direct_modes(*mode_indices*, *mode_handles*, *direct_vec_handles=None*)
Computes direct BPOD modes and calls `put` on them using mode handles.

Args: `mode_indices`: List of indices describing which direct modes to compute, e.g. `range(10)` or `[3, 0, 5]`.

`mode_handles`: List of handles for direct modes to compute.

Kwargs: `direct_vec_handles`: List of handles for direct vector objects. Optional if given when calling *compute_decomp()*.

compute_direct_proj_coeffs()
Computes biorthogonal projection of direct vector objects onto direct BPOD modes, using adjoint BPOD modes.

Returns: direct_proj_coeffs: Array of projection coefficients for direct vector objects, expressed as a linear combination of direct BPOD modes. Columns correspond to direct vector objects, rows correspond to direct BPOD modes.

get_Hankel_array(*src*)
Gets the Hankel array from source (memory or file).

Args: src: Source from which to retrieve Hankel singular values.

get_adjoint_proj_coeffs(*src*)
Gets the adjoint projection coefficients from source (memory or file).

Args: src: Source from which to retrieve adjoint projection coefficients.

get_decomp(*sing_vals_src*, *L_sing_vecs_src*, *R_sing_vecs_src*)
Gets the decomposition arrays from sources (memory or file).

Args: sing_vals_src: Source from which to retrieve Hankel singular values.

L_sing_vecs_src: Source from which to retrieve left singular vectors of Hankel array.

R_sing_vecs_src: Source from which to retrieve right singular vectors of Hankel array.

get_direct_proj_coeffs(*src*)
Gets the direct projection coefficients from source (memory or file).

Args: src: Source from which to retrieve direct projection coefficients.

put_Hankel_array(*dest*)
Puts Hankel array to dest.

put_L_sing_vecs(*dest*)
Puts left singular vectors of Hankel array to dest.

put_R_sing_vecs(*dest*)
Puts right singular vectors of Hankel array to dest.

put_adjoint_proj_coeffs(*dest*)
Puts adjoint projection coefficients to dest

put_decomp(*sing_vals_dest*, *L_sing_vecs_dest*, *R_sing_vecs_dest*)
Puts the decomposition arrays in destinations (memory or file).

Args: sing_vals_dest: Destination in which to put Hankel singular values.

L_sing_vecs_dest: Destination in which to put left singular vectors of Hankel array.

R_sing_vecs_dest: Destination in which to put right singular vectors of Hankel array.

put_direct_proj_coeffs(*dest*)
Puts direct projection coefficients to dest

put_sing_vals(*dest*)
Puts Hankel singular values to dest.

sanity_check(*test_vec_handle*)
Checks that user-supplied vector handle and vector satisfy requirements.

Args: test_vec_handle: A vector handle to test.

See vectorspace.VectorSpaceHandles.sanity_check().

modred.bpod.**compute_BPOD_arrays**(*direct_vecs*, *adjoint_vecs*, *num_inputs=1*, *num_outputs=1*, *direct_mode_indices=None*, *adjoint_mode_indices=None*, *inner_product_weights=None*, *atol=1e-13*, *rtol=None*)

Computes BPOD modes using data stored in arrays, using method of snapshots.

Args: `direct_vecs`: Array whose columns are direct data vectors (X). They should be stacked so that if there are p inputs, the first p columns should all correspond to the same timestep. For instance, these are often all initial conditions of p different impulse responses.

`adjoint_vecs`: Array whose columns are adjoint data vectors (Y). They should be stacked so that if there are q outputs, the first q columns should all correspond to the same timestep. For instance, these are often all initial conditions of q different adjoint impulse responses.

Kwargs: `num_inputs`: Number of inputs to the system.

`num_outputs`: Number of outputs of the system.

`direct_mode_indices`: List of indices describing which direct modes to compute. Examples are `range(10)` or `[3, 0, 6, 8]`. If no mode indices are specified, then all modes will be computed.

`adjoint_mode_indices`: List of indices describing which adjoint modes to compute. Examples are `range(10)` or `[3, 0, 6, 8]`. If no mode indices are specified, then all modes will be computed.

`inner_product_weights`: 1D or 2D array of inner product weights. Corresponds to W in inner product $v_1^* W v_2$.

`atol`: Level below which Hankel singular values are truncated.

`rtol`: Maximum relative difference between largest and smallest Hankel singular values. Smaller ones are truncated.

Returns: `res`: Results of BPOD computation, stored in a namedtuple with the following attributes:

- `sing_vals`: 1D array of Hankel singular values (E).
- `direct_modes`: Array whose columns are direct modes.
- `adjoint_modes`: Array whose columns are adjoint modes.
- `direct_proj_coeffs`: Array of projection coefficients for direct vector objects, expressed as a linear combination of direct BPOD modes. Columns correspond to direct vector objects, rows correspond to direct BPOD modes.
- **`adjoint_proj_coeffs`: Array of projection coefficients for adjoint** vector objects, expressed as a linear combination of adjoint BPOD modes. Columns correspond to adjoint vector objects, rows correspond to adjoint BPOD modes.
- `L_sing_vecs`: Array whose columns are left singular vectors of Hankel array (U).
- `R_sing_vecs`: Array whose columns are right singular vectors of Hankel array (V).
- `Hankel_array`: Hankel array ($Y^* W X$).

Attributes can be accessed using calls like `res.direct_modes`. To see all available attributes, use `print(res)`.

See also *BPODHandles*.

CHAPTER 9

DMD

class modred.dmd.**DMDHandles**(*inner_product=None*, *get_array=<function load_array_text>*, *put_array=<function save_array_text>*, *max_vecs_per_node=None*, *verbosity=1*)

Dynamic Mode Decomposition implemented for large datasets.

Kwargs: `inner_product`: Function that computes inner product of two vector objects.

`put_array`: Function to put an array out of modred, e.g., write it to file.

`get_array`: Function to get an array into modred, e.g., load it from file.

`max_vecs_per_node`: Maximum number of vectors that can be stored in memory, per node.

`verbosity`: 1 prints progress and warnings, 0 prints almost nothing.

Computes DMD modes from vector objects (or handles). It uses `vectorspace.VectorSpaceHandles` for low level functions.

Usage:

```
myDMD = DMDHandles(inner_product=my_inner_product)
myDMD.compute_decomp(vec_handles)
myDMD.compute_exact_modes(range(50), mode_handles)
```

See also *compute_DMD_arrays_snaps_method()*, *compute_DMD_arrays_direct_method()*, and `vectors`.

compute_adjoint_modes(*mode_indices*, *mode_handles*, *vec_handles=None*)

Computes adjoint DMD modes and calls `put` on them using mode handles.

Args: `mode_indices`: List of indices describing which projected modes to compute, e.g. `range(10)` or `[3, 0, 5]`.

`mode_handles`: List of handles for projected modes to compute.

Kwargs: `vec_handles`: List of handles for vector objects. Optional if when calling *compute_decomp()*.

compute_decomp(*vec_handles*, *adv_vec_handles=None*, *atol=1e-13*, *rtol=None*, *max_num_eigvals=None*)

Computes eigendecomposition of low-order linear map approximating relationship between vector objects, returning various arrays necessary for computing and characterizing DMD modes.

Args: `vec_handles`: List of handles for vector objects.

Kwargs: `adv_vec_handles`: List of handles for vector objects advanced in time. If not provided, it is assumed that the vector objects describe a sequential time-series. Thus `vec_handles` becomes `vec_handles[:-1]` and `adv_vec_handles` becomes `vec_handles[1:]`.

`atol`: Level below which DMD eigenvalues are truncated.

`rtol`: Maximum relative difference between largest and smallest DMD eigenvalues. Smaller ones are truncated.

`max_num_eigvals`: Maximum number of DMD eigenvalues that will be computed. This is enforced by truncating the basis onto which the approximating linear map is projected. Computationally, this corresponds to truncating the eigendecomposition of the correlation array. If set to None, no truncation will be performed, and the maximum possible number of DMD eigenvalues will be computed.

Returns: `eigvals`: 1D array of eigenvalues of low-order linear map, i.e., the DMD eigenvalues.

`R_low_order_eigvecs`: Array whose columns are right eigenvectors of approximating low-order linear map.

`L_low_order_eigvecs`: Array whose columns are left eigenvectors of approximating low-order linear map.

`correlation_array_eigvals`: 1D array of eigenvalues of correlation array.

`correlation_array_eigvecs`: Array whose columns are eigenvectors of correlation array.

compute_eigendecomp(*atol=1e-13*, *rtol=None*, *max_num_eigvals=None*)

Computes eigendecompositions of correlation array and approximating low-order linear map.

Kwargs: `atol`: Level below which eigenvalues of correlation array are truncated.

`rtol`: Maximum relative difference between largest and smallest eigenvalues of correlation array. Smaller ones are truncated.

`max_num_eigvals`: Maximum number of DMD eigenvalues that will be computed. This is enforced by truncating the basis onto which the approximating linear map is projected. Computationally, this corresponds to truncating the eigendecomposition of the correlation array. If set to None, no truncation will be performed, and the maximum possible number of DMD eigenvalues will be computed.

Useful if you already have the correlation array and cross-correlation array and want to avoid recomputing them.

Usage:

```
DMD.correlation_array = pre_existing_correlation_array
DMD.cross_correlation_array = pre_existing_cross_correlation_array
DMD.compute_eigendecomp()
DMD.compute_exact_modes(
    mode_idx_list, mode_handles, adv_vec_handles=adv_vec_handles)
```

Another way to use this is to compute a DMD using a truncated basis for the projection of the approximating linear map. Start by either computing a full decomposition or by loading pre-computed correlation and cross-correlation arrays.

Usage:

```
# Start with a full decomposition
DMD_eigvals, correlation_array_eigvals = DMD.compute_decomp(
    vec_handles)[0, 3]

# Do some processing to determine the truncation level, maybe based
# on the DMD eigenvalues and correlation array eigenvalues
desired_num_eigvals = my_post_processing_func(
    DMD_eigvals, correlation_array_eigvals)

# Do a truncated decomposition
DMD_eigvals_trunc = DMD.compute_eigendecomp(
  max_num_eigvals=desired_num_eigvals)

# Compute modes for truncated decomposition
DMD.compute_exact_modes(
    mode_idx_list, mode_handles, adv_vec_handles=adv_vec_handles)
```

Since it doesn't overwrite the correlation and cross-correlation arrays, `compute_eigendecomp` can be called many times in a row to do computations for different truncation levels. However, the results of the decomposition (e.g., `self.eigvals`) do get overwritten, so you may want to call a `put` method to save those results somehow.

compute_exact_modes(*mode_indices*, *mode_handles*, *adv_vec_handles=None*)

Computes exact DMD modes and calls `put` on them using mode handles.

Args: `mode_indices`: List of indices describing which exact modes to compute, e.g. `range(10)` or `[3, 0, 5]`.

`mode_handles`: List of handles for exact modes to compute.

Kwargs: `vec_handles`: List of handles for vector objects. Optional if when calling *compute_decomp()*.

compute_proj_coeffs()

Computes projection of vector objects onto DMD modes. Note that a biorthogonal projection onto exact DMD modes is analytically equivalent to a least-squares projection onto projected DMD modes.

Returns: `proj_coeffs`: Array of projection coefficients for vector objects, expressed as a linear combination of DMD modes. Columns correspond to vector objects, rows correspond to DMD modes.

`adv_proj_coeffs`: Array of projection coefficients for vector objects advanced in time, expressed as a linear combination of DMD modes. Columns correspond to vector objects, rows correspond to DMD modes.

compute_proj_modes(*mode_indices*, *mode_handles*, *vec_handles=None*)

Computes projected DMD modes and calls `put` on them using mode handles.

Args: `mode_indices`: List of indices describing which projected modes to compute, e.g. `range(10)` or `[3, 0, 5]`.

`mode_handles`: List of handles for projected modes to compute.

Kwargs: `vec_handles`: List of handles for vector objects. Optional if when calling *compute_decomp()*.

compute_spectrum()

Computes DMD spectral coefficients. These coefficients come from a biorthogonal projection of the first vector object onto the exact DMD modes, which is analytically equivalent to doing a least-squares projection onto the projected DMD modes.

Returns: `spectral_coeffs`: 1D array of DMD spectral coefficients, calculated as the magnitudes of the projection coefficients of first data vector. The projection is onto the span of the DMD modes using the (biorthogonal) adjoint DMD modes. Note that this is the same as a least-squares projection onto the span of the DMD modes.

get_correlation_array(*src*)
Gets the correlation array from source (memory or file).

Args: `src`: Source from which to retrieve correlation array.

get_cross_correlation_array(*src*)
Gets the cross-correlation array from source (memory or file).

Args: `src`: Source from which to retrieve cross-correlation array.

get_decomp(*eigvals_src*, *R_low_order_eigvecs_src*, *L_low_order_eigvecs_src*, *correlation_array_eigvals_src*, *correlation_array_eigvecs_src*)
Gets the decomposition arrays from sources (memory or file).

Args: `eigvals_src`: Source from which to retrieve eigenvalues of approximating low-order linear map (DMD eigenvalues).

`R_low_order_eigvecs_src`: Source from which to retrieve right eigenvectors of approximating low-order linear DMD map.

`L_low_order_eigvecs_src`: Source from which to retrieve left eigenvectors of approximating low-order linear DMD map.

`correlation_array_eigvals_src`: Source from which to retrieve eigenvalues of correlation array.

`correlation_array_eigvecs_src`: Source from which to retrieve eigenvectors of correlation array.

get_proj_coeffs(*src*, *adv_src*)
Gets the projection coefficients and advanced projection coefficients from sources (memory or file).

Args: `src`: Source from which to retrieve projection coefficients.

`adv_src`: Source from which to retrieve advanced projection coefficients.

get_spectral_coeffs(*src*)
Gets the spectral coefficients from source (memory or file).

Args: `src`: Source from which to retrieve spectral coefficients.

put_L_low_order_eigvecs(*dest*)
Puts left eigenvectors of approximating low-order linear map to `dest`.

put_R_low_order_eigvecs(*dest*)
Puts right eigenvectors of approximating low-order linear map to `dest`.

put_correlation_array(*dest*)
Puts correlation array to `dest`.

put_correlation_array_eigvals(*dest*)
Puts eigenvalues of correlation array to `dest`.

put_correlation_array_eigvecs(*dest*)
Puts eigenvectors of correlation array to `dest`.

put_cross_correlation_array(*dest*)
Puts cross-correlation array to `dest`.

put_decomp(*eigvals_dest*, *R_low_order_eigvecs_dest*, *L_low_order_eigvecs_dest*, *correlation_array_eigvals_dest*, *correlation_array_eigvecs_dest*)
Puts the decomposition arrays in destinations (memory or file).

Args: `eigvals_dest`: Destination in which to put eigenvalues of approximating low-order linear map (DMD eigenvalues).

`R_low_order_eigvecs_dest`: Destination in which to put right eigenvectors of approximating low-order linear map.

`L_low_order_eigvecs_dest`: Destination in which to put left eigenvectors of approximating low-order linear map.

`correlation_array_eigvals_dest`: Destination in which to put eigenvalues of correlation array.

`correlation_array_eigvecs_dest`: Destination in which to put eigenvectors of correlation array.

put_eigvals(*dest*)
Puts eigenvalues of approximating low-order-linear map (DMD eigenvalues) to `dest`.

put_proj_coeffs(*dest*, *adv_dest*)
Puts projection coefficients to `dest`, advanced projection coefficients to `adv_dest`.

put_spectral_coeffs(*dest*)
Puts DMD spectral coefficients to `dest`.

sanity_check(*test_vec_handle*)
Checks that user-supplied vector handle and vector satisfy requirements.

Args: `test_vec_handle`: A vector handle to test.

See `vectorspace.VectorSpaceHandles.sanity_check()`.

class modred.dmd.**TLSqrDMDHandles**(*inner_product=None*, *get_array=<function load_array_text>*, *put_array=<function save_array_text>*, *max_vecs_per_node=None*, *verbosity=1*)
Total Least Squares Dynamic Mode Decomposition implemented for large datasets.

Kwargs: `inner_product`: Function that computes inner product of two vector objects.

`put_array`: Function to put an array out of modred, e.g., write it to file.

`get_array`: Function to get an array into modred, e.g., load it from file.

`max_vecs_per_node`: Maximum number of vectors that can be stored in memory, per node.

`verbosity`: 1 prints progress and warnings, 0 prints almost nothing.

Computes Total-Least-Squares DMD modes from vector objects (or handles). It uses `vectorspace.VectorSpaceHandles` for low level functions.

Usage:

```
myDMD = TLSqrDMDHandles(inner_product=my_inner_product)
myDMD.compute_decomp(vec_handles, max_num_eigvals=100)
myDMD.compute_exact_modes(range(50), mode_handles)
```

The total-least-squares variant of DMD accounts for an asymmetric treatment of the non-time-advanced and time-advanced data in standard DMD, in which the former is assumed to be perfect information, whereas all noise is limited to the latter. In many cases, since the non-time-advanced and advanced data come from the same source, noise could be present in either.

Note that max_num_eigvals must be set to a value smaller than the rank of the dataset. In other words, if the projection basis for total-least-squares DMD is not truncated, then the algorithm reduces to standard DMD. For over-constrained datasets (number of vector objects is larger than the size of each vector object), this occurs naturally. For under-constrained datasets, (number of vector objects is smaller than size of vector objects), this must be done explicitly by the user. At this time, there is no standard method for choosing a truncation level. One approach is to look at the roll-off of the correlation array eigenvalues, which contains information about the "energy" content of each projection basis vectors.

Also, note that *TLSqrDMDHandles* inherits from *DMDHandles*, so certain methods are available, even though they are not implemented/documented here (namely several `put` functions).

See also *compute_TLSqrDMD_arrays_snaps_method()*, *compute_TLSqrDMD_arrays_direct_method()*, and `vectors`.

compute_decomp(*vec_handles*, *adv_vec_handles=None*, *atol=1e-13*, *rtol=None*, *max_num_eigvals=None*)

Computes eigendecomposition of low-order linear map approximating relationship between vector objects, returning various arrays necessary for computing and characterizing DMD modes.

Args: `vec_handles`: List of handles for vector objects.

Kwargs: `adv_vec_handles`: List of handles for vector objects advanced in time. If not provided, it is assumed that the vector objects describe a sequential time-series. Thus `vec_handles` becomes `vec_handles[:-1]` and `adv_vec_handles` becomes `vec_handles[1:]`.

`atol`: Level below which DMD eigenvalues are truncated.

`rtol`: Maximum relative difference between largest and smallest DMD eigenvalues. Smaller ones are truncated.

`max_num_eigvals`: Maximum number of DMD eigenvalues that will be computed. This is enforced by truncating the basis onto which the approximating linear map is projected. Computationally, this corresponds to truncating the eigendecomposition of the correlation array. If set to None, no truncation will be performed, and the maximum possible number of DMD eigenvalues will be computed.

Returns: `eigvals`: 1D array of eigenvalues of low-order linear map, i.e., the DMD eigenvalues.

`R_low_order_eigvecs`: Array whose columns are right eigenvectors of approximating low-order linear map.

`L_low_order_eigvecs`: Array whose columns are left eigenvectors of approximating low-order linear map.

`sum_correlation_array_eigvals`: 1D array of eigenvalues of sum correlation array.

`sum_correlation_array_eigvecs`: Array whose columns are eigenvectors of sum correlation array.

`proj_correlation_array_eigvals`: 1D array of eigenvalues of projected correlation array.

`proj_correlation_array_eigvecs`: Array whose columns are eigenvectors of projected correlation array.

Note that the truncation level (corresponding to `max_num_eigvals`) must be set to a value smaller than the rank of the dataset, otherwise total-least-squares DMD reduces to standard DMD. This occurs naturally for over-constrained datasets, but must be enforced by the user for under-constrined datasets.

compute_eigendecomp(*atol=1e-13*, *rtol=None*, *max_num_eigvals=None*)

Computes eigendecompositions of correlation array and approximating low-order linear map.

Kwargs: `atol`: Level below which eigenvalues of correlation array are truncated.

`rtol`: Maximum relative difference between largest and smallest eigenvalues of correlation array. Smaller ones are truncated.

max_num_eigvals: Maximum number of DMD eigenvalues that will be computed. This is enforced by truncating the basis onto which the approximating linear map is projected. Computationally, this corresponds to truncating the eigendecomposition of the correlation array. If set to None, no truncation will be performed, and the maximum possible number of DMD eigenvalues will be computed.

Useful if you already have the correlation array and cross-correlation array and want to avoid recomputing them.

Usage:

```
TLSqrDMD.correlation_array = pre_existing_correlation_array
TLSqrDMD.adv_correlation_array = pre_existing_adv_correlation_array
TLSqrDMD.cross_correlation_array =
    pre_existing_cross_correlation_array
TLSqrDMD.compute_eigendecomp()
TLSqrDMD.compute_exact_modes(
    mode_idx_list, mode_handles, adv_vec_handles=adv_vec_handles)
```

Another way to use this is to compute TLSqrDMD using different basis truncation levels for the projection of the approximating linear map. Start by either computing a full decomposition or by loading pre-computed correlation and cross-correlation arrays.

Usage:

```
# Start with a full decomposition
DMD_eigvals, correlation_array_eigvals = TLSqrDMD.compute_decomp(
    vec_handles)[0, 3]

# Do some processing to determine the truncation level, maybe based
# on the DMD eigenvalues and correlation array eigenvalues
desired_num_eigvals = my_post_processing_func(
    DMD_eigvals, correlation_array_eigvals)

# Do a truncated decomposition
DMD_eigvals_trunc = TLSqrDMD.compute_eigendecomp(
  max_num_eigvals=desired_num_eigvals)

# Compute modes for truncated decomposition
TLSqrDMD.compute_exact_modes(
    mode_idx_list, mode_handles, adv_vec_handles=adv_vec_handles)
```

Since it doesn't overwrite the correlation and cross-correlation arrays, compute_eigendecomp can be called many times in a row to do computations for different truncation levels. However, the results of the decomposition (e.g., self.eigvals) do get overwritten, so you may want to call a put method to save those results somehow.

Note that the truncation level (corresponding to max_num_eigvals) must be set to a value smaller than the rank of the dataset, otherwise total-least-squares DMD reduces to standard DMD. This occurs naturally for over-constrained datasets, but must be enforced by the user for under-constrined datasets.

compute_proj_coeffs()

Computes projection of (de-noised) vector objects onto DMD modes. Note that a biorthogonal projection onto exact DMD modes is analytically equivalent to a least-squares projection onto projected DMD modes.

Returns: proj_coeffs: Array of projection coefficients for (de-noised)vector objects, expressed as a linear combination of DMD modes. Columns correspond to vector objects, rows correspond to DMD modes.

adv_proj_coeffs: Array of projection coefficients for (de-noised) vector objects advanced in time, expressed as a linear combination of DMD modes. Columns correspond to vector objects, rows

correspond to DMD modes.

compute_spectrum()
Computes DMD spectral coefficients. These coefficients come from a biorthogonal projection of the first (de-noised) vector object onto the exact DMD modes, which is analytically equivalent to doing a least-squares projection onto the projected DMD modes.

Returns: `spectral_coeffs`: 1D array of DMD spectral coefficients.

get_adv_correlation_array(*src*)
Gets the advanced correlation array from source (memory or file).

Args: `src`: Source from which to retrieve advanced correlation array.

get_decomp(*eigvals_src*, *R_low_order_eigvecs_src*, *L_low_order_eigvecs_src*, *sum_correlation_array_eigvals_src*, *sum_correlation_array_eigvecs_src*, *proj_correlation_array_eigvals_src*, *proj_correlation_array_eigvecs_src*)
Gets the decomposition arrays from sources (memory or file).

Args: `eigvals_src`: Source from which to retrieve eigenvalues of approximating low-order linear map (DMD eigenvalues).

`R_low_order_eigvecs_src`: Source from which to retrieve right eigenvectors of approximating low-order linear DMD map.

`L_low_order_eigvecs_src`: Source from which to retrieve left eigenvectors of approximating low-order linear DMD map.

`sum_correlation_array_eigvals_src`: Source from which to retrieve eigenvalues of sum correlation array.

`sum_correlation_array_eigvecs_src`: Source from which to retrieve eigenvectors of sum correlation array.

`proj_correlation_array_eigvals_src`: Source from which to retrieve eigenvalues of projected correlation array.

`proj_correlation_array_eigvecs_src`: Source from which to retrieve eigenvectors of projected correlation array.

get_proj_correlation_array(*src*)
Gets the projected correlation array from source (memory or file).

Args: `src`: Source from which to retrieve projected correlation array.

get_sum_correlation_array(*src*)
Gets the sum correlation array from source (memory or file).

Args: `src`: Source from which to retrieve sum correlation array.

put_adv_correlation_array(*dest*)
Puts advanced correlation array to `dest`.

put_correlation_array_eigvals(*dest*)
This method is not available for total least squares DMD

put_correlation_array_eigvecs(*dest*)
This method is not available for total least squares DMD

put_decomp(*eigvals_dest*, *R_low_order_eigvecs_dest*, *L_low_order_eigvecs_dest*, *sum_correlation_array_eigvals_dest*, *sum_correlation_array_eigvecs_dest*, *proj_correlation_array_eigvals_dest*, *proj_correlation_array_eigvecs_dest*)
Puts the decomposition arrays in destinations (file or memory).

Args: `eigvals_dest`: Destination in which to put eigenvalues of approximating low-order linear map (DMD eigenvalues).

`R_low_order_eigvecs_dest`: Destination in which to put right eigenvectors of approximating low-order linear map.

`L_low_order_eigvecs_dest`: Destination in which to put left eigenvectors of approximating low-order linear map.

`sum_correlation_array_eigvals_dest`: Destination in which to put eigenvalues of sum correlation array.

`sum_correlation_array_eigvecs_dest`: Destination in which to put eigenvectors of sum correlation array.

`proj_correlation_array_eigvals_dest`: Destination in which to put eigenvalues of projected correlation array.

`proj_correlation_array_eigvecs_dest`: Destination in which to put eigenvectors of projected correlation array.

put_proj_correlation_array(*dest*)
Puts projected correlation array to `dest`.

put_proj_correlation_array_eigvals(*dest*)
Puts eigenvalues of projected correlation array to `dest`.

put_proj_correlation_array_eigvecs(*dest*)
Puts eigenvectors of projected correlation array to `dest`.

put_sum_correlation_array(*dest*)
Puts sum correlation array to `dest`.

put_sum_correlation_array_eigvals(*dest*)
Puts eigenvalues of sum correlation array to `dest`.

put_sum_correlation_array_eigvecs(*dest*)
Puts eigenvectors of sum correlation array to `dest`.

modred.dmd.**compute_DMD_arrays_direct_method**(*vecs*, *adv_vecs=None*, *mode_indices=None*, *inner_product_weights=None*, *atol=1e-13*, *rtol=None*, *max_num_eigvals=None*)

Computes DMD modes using data stored in arrays, using direct method.

Args: `vecs`: Array whose columns are data vectors.

Kwargs: `adv_vecs`: Array whose columns are data vectors advanced in time. If not provided, then it is assumed that the vectors describe a sequential time-series. Thus `vecs` becomes `vecs[:, :-1]` and `adv_vecs` becomes `vecs[:, 1:]`.

`mode_indices`: List of indices describing which modes to compute. Examples are `range(10)` or `[3, 0, 6, 8]`. If no mode indices are specified, then all modes will be computed.

`inner_product_weights`: 1D or 2D array of inner product weights. Corresponds to W in inner product $v_1^* W v_2$.

`atol`: Level below which eigenvalues of correlation array are truncated.

`rtol`: Maximum relative difference between largest and smallest eigenvalues of correlation array. Smaller ones are truncated.

`max_num_eigvals`: Maximum number of DMD eigenvalues that will be computed. This is enforced by truncating the basis onto which the approximating linear map is projected. Computationally, this cor-

responds to truncating the eigendecomposition of the correlation array. If set to None, no truncation will be performed, and the maximum possible number of DMD eigenvalues will be computed.

Returns: `res`: Results of DMD computation, stored in a namedtuple with the following attributes:

- `eigvals`: 1D array of eigenvalues of approximating low-order linear map (DMD eigenvalues).
- `spectral_coeffs`: 1D array of DMD spectral coefficients, calculated as the magnitudes of the projection coefficients of first data vector. The projection is onto the span of the DMD modes using the (biorthogonal) adjoint DMD modes. Note that this is the same as a least-squares projection onto the span of the DMD modes.
- `exact_modes`: Array whose columns are exact DMD modes.
- `proj_modes`: Array whose columns are projected DMD modes.
- `adjoint_modes`: Array whose columns are adjoint DMD modes.
- `proj_coeffs`: Array of projection coefficients for data vectors expressed as a linear combination of DMD modes. Columns correspond to vector objects, rows correspond to DMD modes. The projection is onto the span of the DMD modes using the (biorthogonal) adjoint DMD modes. Note that this is the same as a least-squares projection onto the span of the DMD modes.
- `adv_proj_coeffs`: Array of projection coefficients for data vectors advanced in time, expressed as a linear combination of DMD modes. Columns correspond to vector objects, rows correspond to DMD modes. The projection is onto the span of the DMD modes using the (biorthogonal) adjoint DMD modes. Note that this is the same as a least-squares projection onto the span of the DMD modes.
- `R_low_order_eigvecs`: Array of right eigenvectors of approximating low-order linear map.
- `L_low_order_eigvecs`: Array of left eigenvectors of approximating low-order linear map.
- `correlation_array_eigvals`: 1D array of eigenvalues of correlation array.
- `correlation_array_eigvecs`: Array of eigenvectors of correlation array.

Attributes can be accessed using calls like `res.exact_modes`. To see all available attributes, use `print(res)`.

This method does not square the array of vectors as in the method of snapshots (*compute_DMD_arrays_snaps_method()*). It's slightly more accurate, but slower when the number of elements in a vector is more than the number of vectors, i.e., when `vecs` has more rows than columns.

`modred.dmd.`**`compute_DMD_arrays_snaps_method`**(*vecs*, *adv_vecs=None*, *mode_indices=None*, *inner_product_weights=None*, *atol=1e-13*, *rtol=None*, *max_num_eigvals=None*)

Computes DMD modes using data stored in arrays, using method of snapshots.

Args: `vecs`: Array whose columns are data vectors.

Kwargs: `adv_vecs`: Array whose columns are data vectors advanced in time. If not provided, then it is assumed that the vectors describe a sequential time-series. Thus `vecs` becomes `vecs[:, :-1]` and `adv_vecs` becomes `vecs[:, 1:]`.

`mode_indices`: List of indices describing which modes to compute. Examples are `range(10)` or `[3, 0, 6, 8]`. If no mode indices are specified, then all modes will be computed.

`inner_product_weights`: 1D or 2D array of inner product weights. Corresponds to W in inner product $v_1^* W v_2$.

`atol`: Level below which eigenvalues of correlation array are truncated.

`rtol`: Maximum relative difference between largest and smallest eigenvalues of correlation array. Smaller ones are truncated.

max_num_eigvals: Maximum number of DMD eigenvalues that will be computed. This is enforced by truncating the basis onto which the approximating linear map is projected. Computationally, this corresponds to truncating the eigendecomposition of the correlation array. If set to None, no truncation will be performed, and the maximum possible number of DMD eigenvalues will be computed.

Returns: res: Results of DMD computation, stored in a namedtuple with the following attributes:

- eigvals: 1D array of eigenvalues of approximating low-order linear map (DMD eigenvalues).
- spectral_coeffs: 1D array of DMD spectral coefficients, calculated as the magnitudes of the projection coefficients of first data vector. The projection is onto the span of the DMD modes using the (biorthogonal) adjoint DMD modes. Note that this is the same as a least-squares projection onto the span of the DMD modes.
- exact_modes: Array whose columns are exact DMD modes.
- proj_modes: Array whose columns are projected DMD modes.
- adjoint_modes: Array whose columns are adjoint DMD modes.
- proj_coeffs: Array of projection coefficients for data vectors expressed as a linear combination of DMD modes. Columns correspond to vector objects, rows correspond to DMD modes. The projection is onto the span of the DMD modes using the (biorthogonal) adjoint DMD modes. Note that this is the same as a least-squares projection onto the span of the DMD modes.
- adv_proj_coeffs: Array of projection coefficients for data vectors advanced in time, expressed as a linear combination of DMD modes. Columns correspond to vector objects, rows correspond to DMD modes. The projection is onto the span of the DMD modes using the (biorthogonal) adjoint DMD modes. Note that this is the same as a least-squares projection onto the span of the DMD modes.
- R_low_order_eigvecs: Array of right eigenvectors of approximating low-order linear map.
- L_low_order_eigvecs: Array of left eigenvectors of approximating low-order linear map.
- correlation_array_eigvals: 1D array of eigenvalues of correlation array.
- correlation_array_eigvecs: Array of eigenvectors of correlation array.
- correlation_array: Correlation array; elements are inner products of data vectors with each other.
- cross_correlation_array: Cross-correlation array; elements are inner products of data vectors with data vectors advanced in time. Going down rows, the data vector changes; going across columns the advanced data vector changes.

Attributes can be accessed using calls like res.exact_modes. To see all available attributes, use print(res).

This uses the method of snapshots, which is faster than the direct method (see *compute_DMD_arrays_direct_method()*) when vecs has more rows than columns, i.e., when there are more elements in a vector than there are vectors. However, it "squares" this array and its singular values, making it slightly less accurate than the direct method.

modred.dmd.**compute_TLSqrDMD_arrays_direct_method**(*vecs*, *adv_vecs=None*, *mode_indices=None*, *inner_product_weights=None*, *atol=1e-13*, *rtol=None*, *max_num_eigvals=None*)

Computes Total Least Squares DMD modes using data stored in arrays, using direct method.

Args: vecs: Array whose columns are data vectors.

Kwargs: `adv_vecs`: Array whose columns are data vectors advanced in time. If not provided, then it is assumed that the vectors describe a sequential time-series. Thus `vecs` becomes `vecs[:, :-1]` and `adv_vecs` becomes `vecs[:, 1:]`.

`mode_indices`: List of indices describing which modes to compute. Examples are `range(10)` or `[3, 0, 6, 8]`. If no mode indices are specified, then all modes will be computed.

`inner_product_weights`: 1D or 2D array of inner product weights. Corresponds to W in inner product $v_1^* W v_2$.

`atol`: Level below which eigenvalues of correlation array are truncated.

`rtol`: Maximum relative difference between largest and smallest eigenvalues of correlation array. Smaller ones are truncated.

`max_num_eigvals`: Maximum number of DMD eigenvalues that will be computed. This is enforced by truncating the basis onto which the approximating linear map is projected. Computationally, this corresponds to truncating the eigendecomposition of the correlation array. If set to None, no truncation will be performed, and the maximum possible number of DMD eigenvalues will be computed.

Returns: `res`: Results of DMD computation, stored in a namedtuple with the following attributes:

- `eigvals`: 1D array of eigenvalues of approximating low-order linear map (DMD eigenvalues).
- `spectral_coeffs`: 1D array of DMD spectral coefficients, calculated as the magnitudes of the projection coefficients of first (de-noised) data vector. The projection is onto the span of the DMD modes using the (biorthogonal) adjoint DMD modes. Note that this is the same as a least-squares projection onto the span of the DMD modes.
- `exact_modes`: Array whose columns are exact DMD modes.
- `proj_modes`: Array whose columns are projected DMD modes.
- `adjoint_modes`: Array whose columns are adjoint DMD modes.
- `proj_coeffs`: Array of projection coefficients for (de-noised) data vectors expressed as a linear combination of DMD modes. Columns correspond to vector objects, rows correspond to DMD modes. The projection is onto the span of the DMD modes using the (biorthogonal) adjoint DMD modes. Note that this is the same as a least-squares projection onto the span of the DMD modes.
- `adv_proj_coeffs`: Array of projection coefficients for (de-noised) data vectors advanced in time, expressed as a linear combination of DMD modes. Columns correspond to vector objects, rows correspond to DMD modes. The projection is onto the span of the DMD modes using the (biorthogonal) adjoint DMD modes. Note that this is the same as a least-squares projection onto the span of the DMD modes.
- `R_low_order_eigvecs`: Array of right eigenvectors of approximating low-order linear map.
- `L_low_order_eigvecs`: Array of left eigenvectors of approximating low-order linear map.
- `sum_correlation_array_eigvals`: 1D array of eigenvalues of sum correlation array.
- `sum_correlation_array_eigvecs`: Array whose columns are eigenvectors of sum correlation array.
- `proj_correlation_array_eigvals`: 1D array of eigenvalues of projected correlation array.
- `proj_correlation_array_eigvecs`: Array whose columns are eigenvectors of projected correlation array.

Attributes can be accessed using calls like `res.exact_modes`. To see all available attributes, use `print(res)`.

This method does not square the array of vectors as in the method of snapshots (*compute_DMD_arrays_snaps_method()*). It's slightly more accurate, but slower when the number of elements in a vector is more than the number of vectors, i.e., when `vecs` has more rows than columns.

Note that max_num_eigvals must be set to a value smaller than the rank of the dataset. In other words, if the projection basis for total-least-squares DMD is not truncated, then the algorithm reduces to standard DMD. For over-constrained datasets (number of columns in data array is larger than the number of rows), this occurs naturally. For under-constrained datasets, (number of vector objects is smaller than size of vector objects), this must be done explicitly by the user. At this time, there is no standard method for choosing a truncation level. One approach is to look at the roll-off of the correlation array eigenvalues, which contains information about the "energy" content of each projection basis vectors.

modred.dmd.**compute_TLSqrDMD_arrays_snaps_method**(*vecs*, *adv_vecs=None*, *mode_indices=None*, *inner_product_weights=None*, *atol=1e-13*, *rtol=None*, *max_num_eigvals=None*)

Computes Total Least Squares DMD modes using data stored in arrays, using method of snapshots.

Args: `vecs`: Array whose columns are data vectors.

Kwargs: `adv_vecs`: Array whose columns are data vectors advanced in time. If not provided, then it is assumed that the vectors describe a sequential time-series. Thus `vecs` becomes `vecs[:, :-1]` and `adv_vecs` becomes `vecs[:, 1:]`.

`mode_indices`: List of indices describing which modes to compute. Examples are `range(10)` or `[3, 0, 6, 8]`. If no mode indices are specified, then all modes will be computed.

`inner_product_weights`: 1D or 2D array of inner product weights. Corresponds to W in inner product $v_1^* W v_2$.

`atol`: Level below which eigenvalues of correlation array are truncated.

`rtol`: Maximum relative difference between largest and smallest eigenvalues of correlation array. Smaller ones are truncated.

`max_num_eigvals`: Maximum number of DMD eigenvalues that will be computed. This is enforced by truncating the basis onto which the approximating linear map is projected. Computationally, this corresponds to truncating the eigendecomposition of the correlation array. If set to None, no truncation will be performed, and the maximum possible number of DMD eigenvalues will be computed.

Returns: `res`: Results of DMD computation, stored in a namedtuple with the following attributes:

- `eigvals`: 1D array of eigenvalues of approximating low-order linear map (DMD eigenvalues).
- `spectral_coeffs`: 1D array of DMD spectral coefficients, calculated as the magnitudes of the projection coefficients of first (de-noised) data vector. The projection is onto the span of the DMD modes using the (biorthogonal) adjoint DMD modes. Note that this is the same as a least-squares projection onto the span of the DMD modes.
- `exact_modes`: Array whose columns are exact DMD modes.
- `proj_modes`: Array whose columns are projected DMD modes.
- `adjoint_modes`: Array whose columns are adjoint DMD modes.
- `proj_coeffs`: Array of projection coefficients for (de-noised) data vectors expressed as a linear combination of DMD modes. Columns correspond to vector objects, rows correspond to DMD modes. The projection is onto the span of the DMD modes using the (biorthogonal) adjoint DMD modes. Note that this is the same as a least-squares projection onto the span of the DMD modes.

- `adv_proj_coeffs`: Array of projection coefficients for (de-noised) data vectors advanced in time, expressed as a linear combination of DMD modes. Columns correspond to vector objects, rows correspond to DMD modes. The projection is onto the span of the DMD modes using the (biorthogonal) adjoint DMD modes. Note that this is the same as a least-squares projection onto the span of the DMD modes.
- `R_low_order_eigvecs`: Array of right eigenvectors of approximating low-order linear map.
- `L_low_order_eigvecs`: Array of left eigenvectors of approximating low-order linear map.
- `sum_correlation_array_eigvals`: 1D array of eigenvalues of sum correlation array.
- `sum_correlation_array_eigvecs`: Array whose columns are eigenvectors of sum correlation array.
- `proj_correlation_array_eigvals`: 1D array of eigenvalues of projected correlation array.
- `proj_correlation_array_eigvecs`: Array whose columns are eigenvectors of projected correlation array.
- `correlation_array`: Correlation array; elements are inner products of data vectors with each other.
- `adv_correlation_array`: Advanced correlation array; elements are inner products of advanced data vectors with each other.
- `cross_correlation_array`: Cross-correlation array; elements are inner products of data vectors with data vectors advanced in time. Going down rows, the data vector changes; going across columns the advanced data vector changes.

Attributes can be accessed using calls like `res.exact_modes`. To see all available attributes, use `print(res)`.

This uses the method of snapshots, which is faster than the direct method (see *compute_TLSqrDMD_arrays_direct_method()*) when `vecs` has more rows than columns, i.e., when there are more elements in a vector than there are vectors. However, it "squares" this array and its singular values, making it slightly less accurate than the direct method.

Note that max_num_eigvals must be set to a value smaller than the rank of the dataset. In other words, if the projection basis for total-least-squares DMD is not truncated, then the algorithm reduces to standard DMD. For over-constrained datasets (number of columns in data array is larger than the number of rows), this occurs naturally. For under-constrained datasets, (number of vector objects is smaller than size of vector objects), this must be done explicitly by the user. At this time, there is no standard method for choosing a truncation level. One approach is to look at the roll-off of the correlation array eigenvalues, which contains information about the "energy" content of each projection basis vectors.

CHAPTER 10

LTIGalerkinProjection

Module for Galerkin projection of LTI systems.

class modred.ltigalerkinproj.**LTIGalerkinProjectionArrays**(*basis_vecs*, *adjoint_basis_vecs=None*, *is_basis_orthonormal=False*, *inner_product_weights=None*, *put_array=<function save_array_text>*)

Computes A, B, and C arrays of reduced-order model (ROM) of a linear time-invariant (LTI) system, from data stored in arrays.

Args: `basis_vecs`: Array whose columns are basis vectors.

Kwargs: `adjoint_basis_vec_handles`: Array whose columns are adjoint basis vectors. If not given, then assumed to be the same as `basis_vecs`.

`is_basis_orthonormal`: Boolean for biorthonormality of the basis vecs. `True` if the basis and adjoint basis vectors are biorthonormal. Default is `False`.

`inner_product_weights`: 1D or 2D array of inner product weights. Corresponds to W in inner product $v_1^* W v_2$.

`put_array`: Function to put an array out of modred, e.g., write it to file.

This class projects discrete- or continuous-time dynamics onto a set of basis vectors, producing reduced-order models. For example, the basis vectors could be POD, BPOD, or DMD modes. It uses `vectorspace.VectorSpaceArrays` for low level functions.

Usage:

```
LTI_proj = LTIGalerkinProjectionArray(
    basis_vecs, adjoint_basis_vecs=adjoint_basis_vecs,
    is_basis_orthonormal=True)
A, B, C = LTI_proj.compute_model(
    A_on_basis_vecs, B_on_standard_basis_array, C_on_basis_vecs)
```

compute_model(*A_on_basis_vecs*, *B_on_standard_basis_array*, *C_on_basis_vecs*)
Computes the continous- or discrete-time reduced A, B, and C arrays.

Args: A_on_basis_vecs: Array whose columns contain $A * basis_v ecs$, i.e., the results of A acting on individual basis vectors. A can represent either discrete- or continuous-time dynamics. For continuous time systems, see also *compute_derivs_arrays()*.

B_on_standard_basis_array: Array whose jth column contains $B * e_j$, i.e., the action of the B array on the jth standard basis vector (a vector with a 1 at the jth index and zeros elsewhere).

C_on_basis_vecs: Array whose columns contain $C * basis_v ecs$, i.e., the results of C acting on individual basis vectors.

Returns: A_reduced: Reduced-order A array.

B_reduced: Reduced-order B array.

C_reduced: Reduced-order C array.

reduce_A(*A_on_basis_vecs*)
Computes the continous- or discrete-time reduced A array.

Args: A_on_basis_vecs: Array whose columns contain $A * basis_v ecs$, i.e., the results of A acting on individual basis vectors. A can represent either discrete- or continuous-time dynamics. For continuous time systems, see also *compute_derivs_arrays()*.

Returns: A_reduced: Reduced-order A array.

reduce_B(*B_on_standard_basis_array*)
Computes the continous- or discrete-time reduced B array.

Args: B_on_standard_basis_array: Array whose jth column contains $B * e_j$, i.e., the action of the B array on the jth standard basis vector (a vector with a 1 at the jth index and zeros elsewhere).

Returns: B_reduced: Reduced-order B array.

Note that if you found the modes via sampling initial condition responses to B (and C), then your snapshots may be missing a factor of 1 / dt (where dt is the first simulation time step). This can be remedied by multiplying B_reduced by dt.

reduce_C(*C_on_basis_vecs*)
Computes the continous- or discrete-time reduced C array.

Args: C_on_basis_vecs: Array whose columns contain $C * basis_v ecs$, i.e., the results of C acting on individual basis vectors.

Returns: C_reduced: Reduced-order C array.

class modred.ltigalerkinproj.**LTIGalerkinProjectionHandles**(*inner_product*, *basis_vec_handles*, *adjoint_basis_vec_handles=None*, *is_basis_orthonormal=False*, *put_array=<function save_array_text>*, *verbosity=1*, *max_vecs_per_node=10000*)
Computes A, B, and C arrays of reduced-order model (ROM) of a linear time-invariant (LTI) system, using vector handles.

Args: inner_product: Function that computes inner product of two vector objects.

basis_vec_handles: List of handles for basis vector objects.

Kwargs: `adjoint_basis_vec_handles`: List of handles for adjoint basis vector objects. If not given, then assumed to be the same as `basis_vec_handles`.

`is_basis_orthonormal`: Boolean for biorthonormality of the basis vecs. `True` if the basis and adjoint basis vectors are biorthonormal. Default is `False`.

`put_array`: Function to put an array out of modred, e.g., write it to file.

`max_vecs_per_node`: Maximum number of vectors that can be stored in memory, per node.

`verbosity`: 1 prints progress and warnings, 0 prints almost nothing.

This class projects discrete- or continuous-time dynamics onto a set of basis vectors, producing reduced-order models. For example, the basis vectors could be POD, BPOD, or DMD modes.

Usage:

```
LTI_proj = LTIGalerkinProjectionHandles(
    inner_product, basis_vec_handles,
    adjoint_basis_vec_handles=adjoint_basis_vec_handles,
    is_basis_orthonormal=True)
A, B, C = LTI_proj.compute_model(
    A_on_basis_vec_handles, B_on_standard_basis_handles, C_on_basis_vecs)
```

compute_model(*A_on_basis_vec_handles*, *B_on_standard_basis_handles*, *C_on_basis_vecs*)
Computes the continous- or discrete-time reduced A, B, and C arrays.

Args: `A_on_basis_vec_handles`: List of handles for the results of A acting on individual basis vectors. A can represent either discrete- or continuous-time dynamics. For continuous time systems, see also *compute_derivs_handles()*.

`B_on_standard_basis_handles`: List of handles for the results of B acting on standard basis vectors (vectors with a 1 at the jth index and zeros elsewhere).

`C_on_basis_vecs`: List of 1D arrays, each of which is the result of C acting on an individual basis vector.

Returns: `A_reduced`: Reduced-order A array.

`B_reduced`: Reduced-order B array.

`C_reduced`: Reduced-order C array.

reduce_A(*A_on_basis_vec_handles*)
Computes the continous- or discrete-time reduced A array.

Args: `A_on_basis_vec_handles`: List of handles for the results of A acting on individual basis vectors. A can represent either discrete- or continuous-time dynamics. For continuous time systems, see also *compute_derivs_handles()*.

Returns: `A_reduced`: Reduced-order A array.

reduce_B(*B_on_standard_basis_handles*)
Computes the continous- or discrete-time reduced B array.

Args: `B_on_standard_basis_handles`: List of handles for the results of B acting on standard basis vectors (vectors with a 1 at the jth index and zeros elsewhere).

Returns: `B_reduced`: Reduced-order B array.

Note that if you found the modes via sampling initial condition responses to B (and C), then your snapshots may be missing a factor of 1/dt (where dt is the first simulation time step). This can be remedied by multiplying `B_reduced` by dt.

reduce_C(*C_on_basis_vecs*)
Computes the continous- or discrete-time reduced C array.

Args: C_on_basis_vecs: List of 1D arrays, each of which is the result of C acting on an individual basis vector.

Returns: C_reduced: Reduced-order C array.

modred.ltigalerkinproj.**compute_derivs_arrays**(*vecs*, *adv_vecs*, *dt*)
Computes 1st-order time derivatives using data stored in arrays.

Args: vecs: Array whose columns are data vectors.

adv_vecs: Array whose columns are data vectors advanced in time.

dt: Time step, corresponding to time advanced between vec_handles and adv_vec_handles

Returns: deriv_vecs: Array whose columns are time derivatives of data vectors.

Computes d(vec)/dt = (vec(t=dt) - vec(t=0)) / dt.

modred.ltigalerkinproj.**compute_derivs_handles**(*vec_handles*, *adv_vec_handles*, *deriv_vec_handles*, *dt*)
Computes 1st-order time derivatives of vector objects, using handles.

Args: vec_handles: List of handles for vector objects.

adv_vec_handles: List of handles for vector objects advanced dt in time.

deriv_vec_handles: List of handles for time derivatives of vector objects.

dt: Time step, corresponding to time advanced between vec_handles and adv_vec_handles

Computes d(vec)/dt = (vec(t=dt) - vec(t=0)) / dt.

modred.ltigalerkinproj.**standard_basis**(*num_dims*)
Returns list of standard basis vectors of space R^n.

Args: num_dims: Number of dimensions.

Returns: basis: The standard basis as a list of 1D arrays [array([1,0,...]), array([0,1,...]), ...].

CHAPTER 11

ERA

Functions and classes fo rERA models. See paper by Ma et al. 2011, TCFD.

class modred.era.**ERA**(*put_array=<function save_array_text>*, *mc=None*, *mo=None*, *verbosity=1*)

Eigensystem realization algorithm (ERA), implemented for discrete-time systems.

Kwargs: `put_array`: Function to put an array out of modred, e.g., write it to file.

`mc`: Number of Markov parameters for controllable dimension of Hankel array.

`mo`: Number of Markov parameters for observable dimension of Hankel array.

`verbosity`: 1 prints progress and warnings, 0 prints almost nothing.

Computes reduced-order model (ROM) of a discrete-time system, using output data from an impulse response.

Simple usage:

```
# Obtain array "Markovs" with dims [time, output, input]
myERA = ERA()
A, B, C = myERA.compute_model(Markovs, 50)
sing_vals = myERA.sing_vals
```

Another example:

```
# Obtain Markov parameters
myERA = ERA()
myERA.compute_model(Markovs, 50)
myERA.put_model('A.txt', 'B.txt', 'C.txt')
```

Notes:

- Default values of `mc` and `mo` are equal and maximal for a balanced model.
- The Markov parameters are to be given in the time-sampled format: $dt * [0, 1, P, P+1, 2P, 2P+1, ...]$
- The special case where $P = 2$ results in $dt * [0, 1, 2, 3, ...]$, see *make_sampled_format()*.
- The functions `util.load_signals()` and `util.load_multiple_signals()` are often useful.

- See convenience function *compute_ERA_model()*.

compute_model(*Markovs*, *num_states*, *mc=None*, *mo=None*)

Computes the A, B, and C arrays of the linear time-invariant (LTI) reduced-order model (ROM).

Args: `Markovs`: Array of Markov parameters with indices [time, output, input]. `Markovs[i]` is the Markov parameter CA^iB.

`num_states`: Number of states in reduced-order model.

Kwargs: `mc`: Number of Markov parameters for controllable dimension of Hankel array.

`mo`: Number of Markov parameters for observable dimension of Hankel array.

Assembles the Hankel arrays from self.Markovs and computes a singular value decomposition. Uses the results to form the A, B, and C arrays.

Note that the default values of `mc` and `mo` are equal and maximal for a balanced model.

Tip: For discrete time systems the impulse is applied over a time interval dt and so has a time-integral $1 * dt$ rather than 1. This means the reduced B array is "off" by a factor of dt. You can account for this by multiplying B by dt.

put_decomp(*sing_vals_dest*, *L_sing_vecs_dest*, *R_sing_vecs_dest*, *Hankel_array_dest*, *Hankel_array2_dest*)

Puts the decomposition arrays and Hankel arrays in destinations (file or memory).

Args: `sing_vals_dest`: Destination in which to put Hankel singular values.

`L_sing_vecs_dest`: Destination in which to put left singular vectors of Hankel array.

`R_sing_vecs_dest`: Destination in which to put right singular vectors of Hankel array.

`Hankel_array_dest`: Destination in which to put Hankel array.

`Hankel_array2_dest`: Destination in which to put second Hankel array.

put_model(*A_dest*, *B_dest*, *C_dest*)

Puts the A, B, and C arrays of the linear time-invariant (LTI) reduced-order model (ROM) in destinations (file or memory).

Args: `A_dest`: Destination in which to put A array of reduced-order model.

`B_dest`: Destination in which to put B array of reduced-order model.

`C_dest`: Destination in which to put C array of reduced-order model.

put_sing_vals(*sing_vals_dest*)

Puts the singular values to `sing_vals_dest`.

`modred.era.`**compute_ERA_model**(*Markovs*, *num_states*)

Convenience function to compute linear time-invariant (LTI) reduced-order model (ROM) arrays A, B, and C using the eigensystem realization algorithm (ERA) with default settings.

Args: `Markovs`: Array of Markov parameters with indices [time, output, input]. `Markovs[i]` is the Markov parameter CA^iB.

`num_states`: Number of states in reduced-order model.

Returns: `A`: A array of reduced-order model.

`B`: B array of reduced-order model.

`C`: C array of reduced-order model.

Usage:

```
# Obtain ``Markovs`` array w/indicies [time, output, input]
num_states = 20
A, B, C = compute_ERA_model(Markovs, num_states)
```

Notes:

- Markov parameters are defined as $[CB, CAB, CA^PB, CA^(P+1)B, ...]$
- The functions `util.load_signals()` and `util.load_multiple_signals()` are often useful.

`modred.era.`**`make_sampled_format`**(*times*, *Markovs*, *dt_tol=1e-06*)

Converts samples at [0 1 2 . . .] into samples at [0 1 1 2 2 3 . . .].

Args: `times`: Array of time values or time step indices.

`Markovs`: Array of Markov parameters with indices [time, output, input]. `Markovs[i]` is the Markov parameter CA^iB.

Kwargs: `dt_tol`: Allowable deviation from uniform time steps.

Returns: `time_steps`: Array of time step indices, [0 1 1 2 2 3 . . .].

`Markovs`: Output array at the time step indices.

`dt`: Time interval between each time step.

Takes a series of data at times $dt * [0123...]$ and duplicates entries so that the result is sampled at $dt * [011223...]$. When the second format is used in the eigensystem realization algorithm (ERA), the resulting model has a time step of dt rather than $2 * dt$.

CHAPTER 12

OKID

OKID function. (Book: Applied System Identification, Jer-Nan Juang, 1994)

modred.okid.**OKID**(*inputs*, *outputs*, *num_Markovs*)

Approximates Markov paramters using arbitrary input and output data.

Args: `inputs`: Array of input signals. Each row corresponds to a different input, each each column to a different time.

`outputs`: Array of output signals. Each row corresponds to a different output, each each column to a different time.

`num_Markovs`: Number of Markov parameters to estimate.

Returns: `Markovs_est`: Array of Markov paramemters. Array dimensions correspond to times, outputs, and inputs, respectively. Thus `Markovs_est[ti]` is the Markov parameter at time index `ti`.

OKID can be sensitive to the choice of parameters. A few tips:

- Use a tail (input=0) for your input/output data, otherwise the Markov parameters might grow rather than decay at large times.
- If necessary, artificially append your data with zero input and exponentially decaying output.
- Set `num_Markovs` less than or equal to half of the number of samples ($num_Markovs <= num_samples/2$). Estimating too many Markov parameters can produce spurious oscillations.
- Data with more than one input tends to be harder to work with.

CHAPTER 13

Vectors Module

Vectors, handles, and inner products.

We recommend using these functions and classes when possible. Otherwise, you can write your own vector class and/or vector handle, see documentation *Interfacing with your data*.

class modred.vectors.**InnerProductTrapz**(**grids*)

Callable that computes inner product of n-dimensional arrays defined on a spatial grid, using the trapezoidal rule.

Args: `*grids`: 1D arrays of grid points, in the order of the spatial dimensions.

Usage:

```
nx = 10
ny = 11
x_grid = 1 - np.cos(np.linspace(0, np.pi, nx))
y_grid = np.linspace(0, 1.0, ny)**2
my_trapz = InnerProductTrapz(x_grid, y_grid)

v1 = np.random.random((nx,ny))
v2 = np.random.random((nx,ny))
IP_v1_v2 = my_trapz(v1, v2)
```

inner_product(*vec1*, *vec2*)

Computes inner product.

class modred.vectors.**VecHandle**(*base_vec_handle=None*, *scale=None*)

Recommended base class for vector handles (not required).

get()

Get a vector, using the private (user-overwritten) `_get` function. If available, the base vector will be subtracted from the specified vector. Then, if a scale factor is specified, the base-subtracted vector will be scaled. The scaled, base-subtracted vector is then returned.

put(*vec*)

Put a vector to file or memory using the private (user-overwritten) `_put` function.

class modred.vectors.**VecHandleArrayText**(*vec_path*, *base_vec_handle=None*, *scale=None*, *is_complex=False*)
Gets and puts array vector objects from/in text files.

class modred.vectors.**VecHandleInMemory**(*vec=None*, *base_vec_handle=None*, *scale=None*)
Gets and puts vectors from/in memory.

class modred.vectors.**VecHandlePickle**(*vec_path*, *base_vec_handle=None*, *scale=None*)
Gets and puts any vector object from/in pickle files.

class modred.vectors.**Vector**
Recommended base class for vector objects (not required).

modred.vectors.**inner_product_array_uniform**(*vec1*, *vec2*)
Takes inner product of numpy arrays without weighting. The first element is conjugated, i.e., IP = np.dot(vec1.conj().T, v2)

CHAPTER 14

VectorSpace

class modred.vectorspace.**VectorSpaceArrays**(*weights=None*)

Implements inner products and linear combinations using data stored in arrays.

Kwargs: `inner_product_weights`: 1D array of inner product weights. Corresponds to W in inner product $v_1^* W v_2$.

class modred.vectorspace.**VectorSpaceHandles**(*inner_product=None*, *max_vecs_per_node=None*, *verbosity=1*, *print_interval=10*)

Provides efficient, parallel implementations of vector space operations, using handles.

Kwargs: `inner_product`: Function that computes inner product of two vector objects.

`max_vecs_per_node`: Maximum number of vectors that can be stored in memory, per node.

`verbosity`: 1 prints progress and warnings, 0 prints almost nothing.

`print_interval`: Minimum time (in seconds) between printed progress messages.

This class implements low-level functions for computing large numbers of vector sums and inner products. These functions are used by high-level classes in `pod`, `bpod`, `dmd` and `ltigalerkinproj`.

Note: Computations are often sped up by using all available processors, even if this lowers `max_vecs_per_node` proportionally. However, this depends on the computer and the nature of the functions supplied, and sometimes loading from file is slower with more processors.

compute_inner_product_array(*row_vec_handles*, *col_vec_handles*)

Computes array whose elements are inner products of the vector objects in `row_vec_handles` and `col_vec_handles`.

Args: `row_vec_handles`: List of handles for vector objects corresponding to rows of the inner product array. For example, in BPOD this is the adjoint snapshot array Y.

`col_vec_handles`: List of handles for vector objects corresponding to columns of the inner product array. For example, in BPOD this is the direct snapshot array X.

Returns: `IP_array`: 2D array of inner products.

The vectors are retrieved in memory-efficient chunks and are not all in memory at once. The row vectors and column vectors are assumed to be different. When they are the same, use `compute_symm_inner_product()` for a 2x speedup.

Each MPI worker (processor) is responsible for retrieving a subset of the rows and columns. The processors then send/receive columns via MPI so they can be used to compute all inner products for the rows on each MPI worker. This is repeated until all MPI workers are done with all of their row chunks. If there are 2 processors:

```
      | x o |
rank0 | x o |
      | x o |
  -
      | o x |
rank1 | o x |
      | o x |
```

In the next step, rank 0 sends column 0 to rank 1 and rank 1 sends column 1 to rank 0. The remaining inner products are filled in:

```
      | x x |
rank0 | x x |
      | x x |
  -
      | x x |
rank1 | x x |
      | x x |
```

When the number of columns and rows is not divisible by the number of processors, the processors are assigned unequal numbers of tasks. However, all processors are always part of the passing cycle.

The scaling is:

- num gets / processor ~ $(n_r * n_c/((max - 2) * n_p * n_p)) + n_r/n_p$
- num MPI sends / processor ~ $(n_p - 1) * (n_r/((max - 2) * n_p)) * n_c/n_p$
- num inner products / processor ~ $n_r * n_c/n_p$

where n_r is number of rows, n_c number of columns, max is `max_vecs_per_proc = max_vecs_per_node/num_procs_per_node`, and n_p is the number of MPI workers (processors).

If there are more rows than columns, then an internal transpose and un-transpose is performed to improve efficiency (since n_c only appears in the scaling in the quadratic term).

compute_symm_inner_product_array(*vec_handles*)

Computes symmetric array whose elements are inner products of the vector objects in `vec_handles` with each other.

Args: `vec_handles`: List of handles for vector objects corresponding to both rows and columns. For example, in POD this is the snapshot array X.

Returns: `IP_array`: 2D array of inner products.

See the documentation for *compute_inner_product_array()* for an idea how this works. Efficiency is achieved by only computing the upper-triangular elements, since the array is symmetric. Within the upper-triangular portion, there are rectangular chunks and triangular chunks. The rectangular chunks are divided up among MPI workers (processors) as weighted tasks. Once those have been computed, the triangular chunks are dealt with.

lin_combine(*sum_vec_handles*, *basis_vec_handles*, *coeff_array*, *coeff_array_col_indices=None*)

Computes linear combination(s) of basis vector objects and calls `put` on result(s), using handles.

Args: `sum_vec_handles`: List of handles for the sum vector objects.

`basis_vec_handles`: List of handles for the basis vector objects.

`coeff_array`: Array whose rows correspond to basis vectors and whose columns correspond to sum vectors. The rows and columns correspond, by index, to the lists `basis_vec_handles` and `sum_vec_handles`. In matrix notation, we can write `sums = basis * coeff_array`

Kwargs: `coeff_array_col_indices`: List of column indices. Only the `sum_vecs` corresponding to these columns of the coefficient array are computed. If no column indices are specified, then all columns will be used.

Each MPI worker (processor) retrieves a subset of the basis vectors to compute as many outputs as an MPI worker (processor) can have in memory at once. Each MPI worker (processor) computes the "layers" from the basis it is responsible for, and for as many modes as it can fit in memory. The layers from all MPI workers (processors) are summed together to form the `sum_vecs` and `put` is called on each.

Scaling is:

num gets/worker = $n_s/(n_p * (max - 2)) * n_b/n_p$

passes/worker = $(n_p - 1) * n_s/(n_p * (max - 2)) * (n_b/n_p)$

scalar multiplies/worker = $n_s * n_b/n_p$

where n_s is number of sum vecs, n_b is number of basis vecs, n_p is number of processors, max = `max_vecs_per_node`.

print_msg(*msg*, *output_channel='stdout'*)
Print a message from rank zero MPI worker/processor.

sanity_check(*test_vec_handle*)
Checks that user-supplied vector handle and vector satisfy requirements.

Args: `test_vec_handle`: A vector handle to test.

The add and multiply functions are tested for the vector object. This is not a complete testing, but catches some common mistakes. An error is raised if a check fails.

CHAPTER 15

Parallel Class

Parallel class and functions for distributed memory

`modred.parallel.`**`barrier`**`()`
Wrapper for Barrier(); forces all processors/MPI workers to synchronize.

`modred.parallel.`**`bcast`**`(`*vals*`)`
Broadcasts values from rank zero processor/MPI worker to all others.

Args: `vals`: Values to broadcast from rank zero processor/MPI worker.

Returns: `outputs`: Broadcasted values

`modred.parallel.`**`call_and_bcast`** (*func*, **args*, ***kwargs*)
Calls function on rank zero processor/MPI worker and broadcasts outputs to all others.

Args: `func`: A callable that takes `*args` and `**kwargs`

`*args`: Required arguments for `func`.

`**kwargs`: Keyword args for `func`.

Usage:

```
# Adds one to the rank, but only evaluated on rank 0, so
# ``outputs==1`` on all processors/MPI workers.
outputs = parallel.call_and_bcast(lambda x: x+1, parallel.get_rank())
```

`modred.parallel.`**`call_from_rank_zero`** (*func*, **args*, ***kwargs*)
Calls function from rank zero processor/MPI worker, does not call `barrier()`.

Args: `func`: Function to call.

`*args`: Required arguments for `func`.

`**kwargs`: Keyword args for `func`.

Usage:

```
parallel.call_from_rank_zero(lambda x: x+1, 1)
```

modred.parallel.**check_for_empty_tasks**(*task_assignments*)
Convenience function that checks for empty processor/MPI worker assignments.

Args: `task_assignments`: List of task assignments.

Returns: `empty_tasks`: `True` if any processor/MPI worker has no tasks, otherwise `False`.

modred.parallel.**find_assignments**(*tasks*, *task_weights=None*)
Evenly distributes tasks among all processors/MPI workers using task weights.

Args: `tasks`: List of tasks. A "task" can be any object that corresponds to a set of operations that needs to be completed. For example `tasks` could be a list of indices, telling each processor/MPI worker which indices of an array to operate on.

Kwargs: `task_weights`: List of weights for each task. These are used to equally distribute the workload among processors/MPI workers, in case some tasks are more expensive than others.

Returns: `task_assignments`: 2D list of tasks, with indices corresponding to [rank][task_index]. Each processor/MPI worker is responsible for `task_assignments[rank]`

modred.parallel.**get_hostname**()
Returns hostname for this node.

modred.parallel.**get_node_ID**()
Returns unique ID number for this node.

modred.parallel.**get_num_MPI_workers**()
Returns number of MPI workers (currently same as number of processors).

modred.parallel.**get_num_nodes**()
Returns number of nodes.

modred.parallel.**get_num_procs**()
Returns number of processors (currently same as number of MPI workers).

modred.parallel.**get_rank**()
Returns rank of this processor/MPI worker.

modred.parallel.**is_distributed**()
Returns True if there is more than one processor/MPI worker and mpi4py was imported properly.

modred.parallel.**is_rank_zero**()
Returns True if rank is zero, False if not.

modred.parallel.**print_from_rank_zero**(*msg*, *output_channel='stdout'*)
Prints `msg` from rank zero processor/MPI worker only.

CHAPTER 16

Utility Functions

A group of useful functions

modred.util.**Hankel**(*first_col*, *last_row=None*)
Construct a Hankel array, whose skew diagonals are constant.

Args: `first_col`: 1D array corresponding to first column of Hankel array.

Kwargs: `last_row`: 1D array corresponding to the last row of Hankel array. First element will be ignored. Default is an array of zeros of the same size as `first_col`.

Returns: Hankel: 2D array with dimensions `[len(first_col), len(last_row)]`.

modred.util.**Hankel_chunks**(*first_col_chunks*, *last_row_chunks=None*)
Construct a Hankel array using chunks, whose elements have Hankel structure at the chunk level (constant along skew diagonals), rather than at the element level.

Args: `first_col_chunks`: List of 2D arrays corresponding to the first column of Hankel array chunks.

Kwargs: `last_row_chunks`: List of 2D arrays corresponding to the last row of Hankel array chunks. Default is a list of arrays of zeros.

Returns: Hankel: 2D array with dimension `[len(first_col) * first_col[0].shape[0], len(last_row) * last_row[0].shape[1]]`.

class modred.util.**InnerProductBlock**(*inner_product*)
Only used in tests. Takes inner product of all vectors.

exception modred.util.**UndefinedError**

modred.util.**atleast_2d_col**(*array*)
Converts 1d arrays to 2d arrays, but always as column vectors

modred.util.**atleast_2d_row**(*array*)
Converts 1d arrays to 2d arrays, but always as row vectors

modred.util.**balanced_truncation**(*A*, *B*, *C*, *order=None*, *return_sing_vals=False*)
Balance and truncate discrete-time linear time-invariant (LTI) system defined by A, B, C arrays. It's not very accurate due to numerical issues.

Args: A, B, C: LTI discrete-time arrays.

Kwargs: order: Order (number of states) of truncated system. Default is to use the maximal possible value (can truncate system afterwards).

Returns: A_balanced, B_balanced, C_balanced: LTI discrete-time arrays of balanced system.

If return_sing_vals is True, also returns:

sing_vals: Hankel singular values.

Notes:

- D is unchanged by balanced truncation.
- This function is not computationally efficient or accurate relative to Matlab's balancmr.

modred.util.**drss**(*num_states*, *num_inputs*, *num_outputs*)
Generates a discrete-time random state-space system.

Args: num_states: Number of states.

num_inputs: Number of inputs.

num_outputs: Number of outputs.

Returns: A, B, C: State-space arrays of discrete-time system.

By construction, all eigenvalues are real and stable.

modred.util.**eig_biorthog**(*array*, *scale_choice='left'*)
Wrapper for numpy.linalg.eig that returns both left and right eigenvectors. Eigenvalues and eigenvectors are sorted and scaled so that the left and right eigenvector arrays are orthonormal.

Args: array: Array to take eigendecomposition of.

Kwargs: scale_choice: Determines whether 'left' (default) or 'right' eigenvectors will be scaled to yield a biorthonormal set. The other eigenvectors will be left unscaled, leaving them with unit norms.

Returns: evals: 1D array of eigenvalues.

R_evecs: Array whose columns are right eigenvectors.

L_evecs: Array whose columns are left eigenvectors.

modred.util.**eigh**(*array*, *atol=1e-13*, *rtol=None*, *is_positive_definite=False*)
Wrapper for numpy.linalg.eigh. Computes eigendecomposition of a Hermitian array.

Args: array: Array to take eigendecomposition of.

atol: Value below which eigenvalues (and corresponding eigenvectors) are truncated.

rtol: Maximum relative difference between largest and smallest eigenvalues. Smaller ones are truncated.

is_positive_definite: If true, array being decomposed will be assumed to be positive definite. Tolerance will be automatically adjusted (if necessary) so that only positive eigenvalues are returned.

Returns: eigvals: 1D array of eigenvalues, sorted in descending order (of magnitude).

eigvecs: Array whose columns are eigenvectors.

modred.util.**flatten_list**(*my_list*)
Flatten a list of lists into a single list.

modred.util.**get_data_members**(*obj*)
Returns a dictionary containing data members of obj.

modred.util.**get_file_list**(*directory*, *file_extension=None*)
Returns list of files in `directory` with `file_extension`.

modred.util.**impulse**(*A*, *B*, *C*, *num_time_steps=None*)
Generates impulse response outputs for a discrete-time system.

Args: `A`, `B`, `C`: State-space system arrays.

Kwargs: `num_time_steps`: Number of time steps to simulate.

Returns: `outputs`: Impulse response outputs, with indices corresponding to [time step, output, input].

No D array is included, but one can simply be prepended to the output if it is non-zero.

modred.util.**load_array_text**(*file_name*, *delimiter=None*, *is_complex=False*)
Reads data saved in a text file, returns an array.

Args: `file_name`: Name of file from which to load data.

Kwargs: `delimiter`: Delimiter in file. Default is same as `numpy.loadtxt`.

`is_complex`: Boolean describing whether the data to be loaded is complex valued.

Returns: `array`: 2D array containing loaded data.

See *save_array_text()* for the format used by this function.

modred.util.**load_multiple_signals**(*signal_paths*, *delimiter=None*)
Loads multiple signal files from text files with columns [t channel1 channel2 ...].

Args: `signal_paths`: List of filepaths to files containing signals.

Returns: `time_values`: 1D array of time values.

`all_signals`: Array of signals with indices [path, time, signal].

See *load_signals()*.

modred.util.**load_signals**(*signal_path*, *delimiter=None*)
Loads signals from text files with columns [t signal1 signal2 ...].

Args: `signal_paths`: List of filepaths to files containing signals.

Returns: `time_values`: 1D array of time values.

`signals`: Array of signals with dimensions [time, signal].

Convenience function. Example file has format:

```
0 0.1 0.2
1 0.2 0.46
2 0.2 1.6
3 0.6 0.1
```

modred.util.**lsim**(*A*, *B*, *C*, *inputs*, *initial_condition=None*)
Simulates a discrete-time system with arbitrary inputs.

$x(n+1) = Ax(n) + Bu(n)$

$y(n) = Cx(n)$

Args: `A`, `B`, and `C`: State-space system arrays.

`inputs`: Array of inputs u, with dimensions `[num_time_steps, num_inputs]`.

Kwargs: `initial_condition`: Initial condition $x(0)$.

Returns: `outputs`: Array of outputs y, with dimensions `[num_time_steps, num_outputs]`.

D array is assumed to be zero.

modred.util.**make_iterable**(*arg*)
Checks if arg is iterable. If not, makes it a one-element list. Otherwise returns arg.

modred.util.**rss**(*num_states*, *num_inputs*, *num_outputs*)
Generates a continuous-time random state-space system.

Args: num_states: Number of states.

num_inputs: Number of inputs.

num_outputs: Number of outputs.

Returns: A, B, C: State-space arrays of continuous-time system.

By construction, all eigenvalues are real and stable.

modred.util.**save_array_text**(*array*, *file_name*, *delimiter=None*)
Saves a 1D or 2D array to a text file.

Args: array: 1D or 2D or array to save to file.

file_name: Filepath to location where data is to be saved.

Kwargs: delimiter: Delimiter in file. Default is same as numpy.savetxt.

Format of saved files is:

```
2.3 3.1 2.1 ...
5.1 2.2 9.8 ...
7.6 3.1 5.5 ...
0.1 1.9 9.1 ...
...
```

Complex data is saved in the following format (as floats):

```
real[0,0] imag[0,0] real[0,1] imag[0,1] ...
real[1,0] imag[1,0] real[1,1] imag[1,1] ...
...
```

Files can be read in Matlab with the provided functions or often with Matlab's load.

modred.util.**smart_eq**(*arg1*, *arg2*)
Checks for equality, accounting for the fact that numpy's == doesn't return a bool. In that case, returns True only if all elements are equal.

modred.util.**sum_arrays**(*arr1*, *arr2*)
Used for allreduce command.

modred.util.**sum_lists**(*list1*, *list2*)
Sums the elements of each list, returns a new list.

This function is used in MPI reduce commands, but could be used elsewhere too.

modred.util.**svd**(*array*, *atol=1e-13*, *rtol=None*)
Wrapper for numpy.linalg.svd, computes the singular value decomposition of an array.

Args: array: Array to take singular value decomposition of.

Kwargs: atol: Level below which singular values are truncated.

rtol: Maximum relative difference between largest and smallest singular values. Smaller ones are truncated.

Returns: `U`: Array whose columns are left singular vectors.

`S`: 1D array of singular values.

`V`: Array whose columns are right singular vectors.

Truncates `U`, `S`, and `V` such that the singular values obey both `atol` and `rtol`.

CHAPTER 17

Release notes

17.1 modred 2.1.0

- Now work only with numpy arrays (not matrices)! Many method names have been changed.
- Implemented the total least-squares variant of DMD, which deals with noise better than standard DMD does. Also did a major rewrite of the unit tests, which now check mathematical properties, rather than comparing the results of `modred` routines to alternate (e.g., brute force) implementations. The new unit tests are more robust and fail much less often.
- Note that a previously unrecorded bug fix has been added to the release notes for version 2.0.2 (regarding correct handling of complex-valued data in computing inner product arrays).

New features and improvements

- Can now compute adjoint DMD modes.

Bug fixes

- `hostname` is now determined using a Windows-friendly method.
- OKID test files now included when installing using `pip`.
- Fixed some minor bugs in example files, which now run.
- Correctly handle `mode_indices=None` option in `VecSpaceHandles.lin_combine()`.
- Complex-valued vectors are now correctly handled in `InnerProductTrapz()`.

Interface changes

- `inner_product` is now a keyword argument in `PODHandles`, `BPODHandles`, and `DMDHandles`, since sometimes instances of these classes are created to compute modes, which does not require computing any inner products.
- Added "direct" to methods for projection coefficients in BPOD.
- Changed symmetric to symm, i.e., `VecSpaceHandles.compute_symmetric_inner_product_array()` is now `VecSpaceHandles.compute_symm_inner_product_array()`.

- Changed return values in array methods for all decomps. Now a single namedtuple is returned, with fields for various internal data, such as `results.eigvals`.
- `parallel` is now a module, not a class. This way users don't have to worry about using the default instance or accidentally creating new instances.
- The attribute `summed_correlation_mats` has been renamed `sum_correlation_mat` to make it clear that the attribute contains a single matrix. All corresponding methods have been renamed accordingly.
- Added compute_adjoint_modes for DMD, TLSqrDMD.
- Added get method for POD projection coefficients.
- Added get method for BPOD projection coefficients.
- Added get methods for DMD handles.
- Added get methods for TLSqrDMD handles.
- Added `broadcast` method to `parallel` module.
- `mode_indices` is now optional in the various compute modes methods

Internal changes

- Exact DMD modes now scaled by DMD eigenvalues.
- Left and right eigenvectors are now computed using *scipy.linalg.eig*, rather than using *numpy.eig* twice, once on the original array and once on its conjugate transpose. This avoids potential numerical issues that cause disagreement between the eigenvalues of the original array and its conjugate transpose (which should be the same, theoretically).
- Removed trailing whitespace from files, as is often done automatically when using Emacs.
- Rewrote total-least squares DMD tests.
- Rewrote DMD tests.
- Rewrote BPOD tests.
- Rewrote POD tests.
- Rewrote ERA tests.
- Improved error checking in `InnerProductTrapz()`.

17.2 modred 2.0.4

Fixed bug for Windows environments. Updated documentation to use Read the Docs.

Bug fixes

- `parallel` modules now uses `socket` module instead of `os` module to find hostname, which is portable to Windows environments.

17.3 modred 2.0.3

Minor bug fix to OKID tests.

Bug fixes

- OKID files needed for tests are included in the source distribution. All doc files are also included.

17.4 modred 2.0.2

Minor bug fixes.

Bug fixes

- Complex-valued vectors are now correctly handled in `VecSpaceHandles.compute_inner_product_mat()` and `VecSpaceHandles.compute_symmetric_inner_product_mat()`.

Interface changes

- Order of returned values for `dmd.compute_DMD_matrices_snaps_method()` and `dmd.compute_DMD_matrices_direct_method()` is now consistent for both values of `return_all`.

17.5 modred 2.0.1

Minor bug fix.

Bug fixes

- `dmd.DMDHandles.compute_spectrum()` now returns real numbers, as it should have before, instead of complex values.

17.6 modred 2.0.0

Main changes are an updated interface for DMD that matches the latest theory and support for Python 3. Python 3 support was primarily implemented by Pierre Augier (pa371 [-at-] damtp [-dot-] cam [-dot-] ac [-dot-] uk). Thanks, Pierre!

New features and improvements

- Python 3 is now supported!
- Documentation has been updated for clarity and consistency, and example code works with the latest interface.
- DMD implementation now matches newest theory, laid out in a 2014 paper by Tu et al. in the Journal of Computational Dynamics. Features were only added, i.e., none were removed. Any DMD computations previously done using modred can be reproduced, though the names of some function calls have changed. Namely, `dmd.DMDHandles.compute_proj_modes()` replaces `dmd.DMDHandles.compute_modes()`, and `dmd.DMDHandles.put_eigvals()` replaces `dmd.DMDHandles.put_ritz_vals()`. Generally, the term "projected modes" has replaced "modes," and similarly "eigenvalues" has replaced "Ritz values." "Exact modes" are now availble in addition to the projected modes.

  A full list of the new functions consists of: `dmd.DMDHandles.compute_exact_modes()`, `dmd.DMDHandles.compute_proj_modes()`, `dmd.DMDHandles.compute_spectrum()`, `dmd.DMDHandles.compute_proj_coeffs()`, `dmd.DMDHandles.compute_eigendecomp()`, `dmd.DMDHandles.put_spectral_coeffs()`, and `dmd.DMDHandles.put_eigvals()`.
- The `compute_decomp` step in DMD has been refactored, resulting in the new method `dmd.DMDHandles.compute_eigendecomp()`. This method can be used to restart DMD computations from saved correlation and cross-correlation matrices, or to compute a DMD using a truncated basis.
- Absolute and relative tolerances can now be passed in using the keyword arguments `atol` and `rtol`, respectively, when calling `compute_decomp` in either POD, BPOD, or DMD. These are then passed on into internal

computations of singular value decompositions or eigendecompositions of positive definite matrices. They allow the user to filter out singular values or eigenvalues that should be considered numerical artifacts. They can also be used to truncate the computations and limit the number of modes making up the decompositions.

- In DMD, truncation can also be achieved by setting the keyword argument `max_num_eigvals` in either `dmd.DMDHandles.compute_decomp()` or `dmd.DMDHandles.compute_eigendecomp()`.
- Added new methods that compute the projection of the original data vectors onto the modes, for POD, BPOD, and DMD, respectively: `pod.PODHandles.compute_proj_coeffs()`, `bpod.BPODHandles.compute_proj_coeffs()`, `bpod.BPODHandles.compute_adj_proj_coeffs()`, `dmd.DMDHandles.compute_proj_coeffs()`.

Bug fixes

- Fixed minor bug in the function `util.impulse`.
- Fixed minor bug in `testvectorspace.py`
- Fixed minor bugs in loading/saving test files, some related to delimiters.
- Fixed bug in `testutil` where `eig_biorthog` was assuming the wrong number of return values.
- Fixed minor bugs in DMD tests related to casting of matrices/arrays.

Interface changes

- Changed the returned values in `dmd.compute_DMD_matrices_snaps_method()`, `dmd.compute_DMD_matrices_direct_method()`, `dmd.DMDHandles.compute_decomp()`.
- Changed the order of the returned values in `pod.PODHandles.compute_decomp()`. `bpod.BPODHandles.compute_decomp()`.
- Changed the order of the arguments in `pod.PODHandles.get_decomp()`, `pod.PODHandles.put_decomp()`, `bpod.BPODHandles.get_decomp()`, `bpod.BPODHandles.put_decomp()`, and `era.ERA.put_decomp()`.
- Changed the arguments to `dmd.DMDHandles.get_decomp()` and `dmd.DMDHandles.put_decomp()`.
- Added the following new methods that compute projections onto modes: `pod.PODHandles.compute_proj_coeffs()`, `bpod.BPODHandles.compute_proj_coeffs()`, `bpod.BPODHandles.compute_adj_proj_coeffs()`, and `dmd.DMDHandles.compute_proj_coeffs()`.
- Added the following new methods that save projection coefficients: `pod.PODHandles.put_proj_coeffs()`, `bpod.BPODHandles.put_direct_proj_coeffs()`, `bpod.BPODHandles.put_adjoint_proj_coeffs()`, and `dmd.DMDHandles.put_proj_coeffs()`.
- Added the following new methods in the updated `DMDHandles` class: `dmd.DMDHandles.compute_exact_modes()`, `dmd.DMDHandles.compute_spectrum()`, `dmd.DMDHandles.compute_eigendecomp()`, `dmd.DMDHandles.put_R_low_order_eigvecs()`, `dmd.DMDHandles.put_L_low_order_eigvecs()`, `dmd.DMDHandles.put_correlation_mat_eigvals()`, `dmd.DMDHandles.put_correlation_mat_eigvecs()`, `dmd.DMDHandles.put_cross_correlation_mat()`, and `dmd.DMDHandles.put_spectral_coeffs()`.
- `dmd.DMDHandles.compute_proj_modes()` replaces `dmd.DMDHandles.compute_modes()`.
- `dmd.DMDHandles.put_eigvals()` replaces `dmd.DMDHandles.put_ritz_vals()`.
- `dmd.DMDHandles.put_build_coeffs()` and `dmd.DMDHandles.put_mode_norms()` are now deprecated.

- Optional `atol` and `rtol` arguments were added to `pod.PODHandles.compute_decomp()`, `bpod.BPODHandles.compute_decomp()`, `dmd.DMDHandles.compute_decomp()`.
- Optional `max_num_eigvals` argument added to `dmd.DMDHandles.compute_decomp()`.
- `util.svd`, `util.eigh`, and `util.eig_biorthog` now consistently return numpy matrices. Previously, the SVD method returned matrices but the eigendecompositions returned arrays.

Internal changes

- In DMD, the build coefficients are no longer considered part of the decomposition and are no longer saved as internal attributes. Instead, its constituent parts define the decomposition (and are saved as internal attributes). Thus computation of the build coefficients in DMD has been moved from the `compute_decomp` method to the `compute_exact_modes` and `compute_proj_modes` methods, respectively, which makes more sense mathematically.
- Added `util.eig_biorthog()` method to compute both left and right eigenvectors of a matrix, scaled to yield a biorthogonal set.
- Added optional `atol` and `rtol` arguments to `util.svd()` and `util.eigh()`.
- Updated tests for `util.svd` and `util.eigh`. Properties of the decompositions are now checked, rather than simply duplicating the computations using built-in numpy methods. This allows for better testing of truncated decompositions. Truncation levels are determined during testing, to ensure that truncation actually occurs and is tested.
- Updated tests for `util.biorthog` to reduce number of failures. Some failures are to be expected due to the fact that we test on random data, but these are much less frequent now.
- Changed how positive definite matrices are generated for use as inner product weight matrices. Previous implementation led to failed tests.
- Changed default delimiter when loading test arrays to `None`.
- Improved type checking to allow for any iterable container, not just lists.
- Removed dependencies on `util.make_list` where possible.
- Removed some duplicate code in `util` module, where `eig_biorthog` had been implemented twice.
- The packaging has been improved.
- Ported to python >= 3.3 using python-future.
- Replaced instances of `xrange` with `range` for compatability with Python 3. (In Python 3, `xrange` has been renamed as `range`.) This is not as efficient in Python 2, but only occurs in a few places and with small enough loops that the impact should be negligible.
- Added a few more checks for `None` values, as Python 3 doesn't allow comparisons of floats to `None`.

17.7 modred 1.0.2

We increased the speed of the BPOD implementations.

New features and improvements

- None

Bug fixes

- None

Interface changes

- None

Internal changes

- BPOD classes now compute fewer inner products. The number of inner products is now the sum of the number of direct vectors and the number of adjoint vectors, whereas previously it was the product. This is achieved by taking advantage of a property of the adjoint.

17.8 modred 1.0.1

Small changes mostly related to examples.

New features and improvements

- None

Bug fixes

- Changed a tutorial example.

Interface changes

- None

Internal changes

- None

17.9 modred 1.0.0

Many interface changes including new classes and functions for different sized data.

New features and improvements

- New functions and classes for data that fits entirely on one node's memory. These are `pod.compute_POD_matrices_snaps_method()`, `pod.compute_POD_matrices_direct_method()`, `bpod.compute_BPOD_matrices()`, `dmd.compute_DMD_matrices_snaps_method()`, `dmd.compute_DMD_matrices_direct_method()`, `ltigalerkinproj.LTIGalerkinProjectionMatrices`, and `vectorspace.VectorSpaceMatrices`. These replace the `in_memory` member functions and improve computational efficiency for small data.
- Added balanced truncation `util.balanced_truncation()`.

Bug fixes

- None

Interface changes

- The old classes `POD`, `BPOD`, `DMD`, are now only for large data and have their names appended with "`Handles`".
- Old classes `LTIGalerkinProjection`, and `VectorSpace` have been split into two, and names appended with "`Matrices`" and "`Handles`".
- All `in_memory` member functions have been removed, replaced by the functions and classes above.
- Removed the `index_from` optional argument in `compute_modes` functions. Mode numbers are now always indexed from zero and are renamed mode indices.

- The `VectorSpace` member function `compute_modes` has been removed and its functionality moved to `lin_combine`.
- `LTIGalerkinProjection` member function `compute_model` uses the result of an operator on a vector, rather than the operator itself. See `ltigalerkinproj.LTIGalerkinProjectionHandles.compute_model()`. The operator classes have been removed.

Internal changes

- OKID now uses least squares instead of a pseudo-inverse for improved numerical stability.
- Added `util.InnerProductBlock` for testing.

17.10 modred 0.3.2

The main change is a bug fix in `util.lsim()`.

New features and improvements

None

Bug fixes

- Function `util.lsim()`, which is only provided for the user's convenience, is simplified and corrected.

Interface changes

- `util.lsim()`.

Internal changes

None

17.11 modred 0.3.1

The main change is a bug fix in the `numpy.eigh` wrapper, `util.eigh()`.

New features and improvements

None

Bug fixes

- The POD and DMD classes now use `util.eigh()` with the `is_positive_definite` flag set to `True`. This eliminates the possibility of small negative eigenvalues that sometimes appear due to numerical precision which led to errors.

Interface changes

None

Internal changes

- Function `util.eigh()` now has a flag for positive definite matrices. When `True`, the function will automatically adjust the tolerance such that only positive eigenvalues are returned.

17.12 modred 0.3.0

New features and improvements

- New class `ltigalerkinproj.LTIGalerkinProjection` for LTI Galerkin projections. Replaces and generalizes old class `BPODLTIROM`.
- Improved print messages to print every 10 seconds and be more informative.

Bug fixes

- Corrected small error in symmetric inner product matrix calculation (used by POD and DMD) where some very small matrix entries were double the true value.
- Fixed race condition in `vectorspace.VectorSpace.lin_combine()` by adding a barrier.

Interface changes

- Removed class `BPODLTIROM`.
- Changed order of indices in Markov parameters returned by `okid.OKID()`.
- Changed all uses of `hankel` to `Hankel` to be consistent with naming convention.

Internal changes

- Added `parallel.Parallel.call_and_bcast()` method to `Parallel` class.
- Changed interface of `helper.add_to_path()`.
- `dmd.DMD` no longer uses an instance of `pod.POD`.
- The equals operator of vector handles now better deals with vectors which are numpy array objects.

17.13 modred 0.2.1

No noteworthy changes from v0.2.0, figuring out pypi website.

17.14 modred 0.2.0

First publicly available version.

CHAPTER 18

Indices and tables

- genindex
- modindex
- search

Bibliography

[HLBR] P. Holmes, J. L. Lumley, G. Berkooz, and C. W. Rowley. *Turbulence, Coherent Structures, Dynamical Systems and Symmetry*, 2nd edition, Cambridge University Press, 2012.

[TRLBK] J. H. Tu, C. W. Rowle, D. M. Luchtenburg, S. L. Brunton, J. N. Kutz. On Dynamic Mode Decmposition: Theory and Applications. *Journal of Computational Dynamics*, 1:391-421, Dec. 2014.

[Ilak2010] M. Ilak, S. Bagheri, L. Brandt, C. W. Rowley, D. S. Henningson. Model Reduction of the Nonlinear Complex Ginzburg–Landau Equation. 2nd edition, Cambridge University Press, 2012. *SIAM Journal of Applied Dynamical Systems*, 9:4:1284-1302, 2010.

Python Module Index

m

modred.bpod, 33
modred.dmd, 37
modred.era, 55
modred.ltigalerkinproj, 51
modred.okid, 59
modred.parallel, 67
modred.pod, 29
modred.util, 69
modred.vectors, 61
modred.vectorspace, 63

Index

A

atleast_2d_col() (*in module modred.util*), 69
atleast_2d_row() (*in module modred.util*), 69

B

balanced_truncation() (*in module modred.util*), 69
barrier() (*in module modred.parallel*), 67
bcast() (*in module modred.parallel*), 67
BPODHandles (*class in modred.bpod*), 33

C

call_and_bcast() (*in module modred.parallel*), 67
call_from_rank_zero() (*in module modred.parallel*), 67
check_for_empty_tasks() (*in module modred.parallel*), 68
compute_adjoint_modes() (*modred.bpod.BPODHandles method*), 34
compute_adjoint_modes() (*modred.dmd.DMDHandles method*), 37
compute_adjoint_proj_coeffs() (*modred.bpod.BPODHandles method*), 34
compute_BPOD_arrays() (*in module modred.bpod*), 35
compute_decomp() (*modred.bpod.BPODHandles method*), 34
compute_decomp() (*modred.dmd.DMDHandles method*), 37
compute_decomp() (*modred.dmd.TLSqrDMDHandles method*), 42
compute_decomp() (*modred.pod.PODHandles method*), 29
compute_derivs_arrays() (*in module modred.ltigalerkinproj*), 54
compute_derivs_handles() (*in module modred.ltigalerkinproj*), 54
compute_direct_modes() (*modred.bpod.BPODHandles method*), 34
compute_direct_proj_coeffs() (*modred.bpod.BPODHandles method*), 34
compute_DMD_arrays_direct_method() (*in module modred.dmd*), 45
compute_DMD_arrays_snaps_method() (*in module modred.dmd*), 46
compute_eigendecomp() (*modred.dmd.DMDHandles method*), 38
compute_eigendecomp() (*modred.dmd.TLSqrDMDHandles method*), 42
compute_eigendecomp() (*modred.pod.PODHandles method*), 30
compute_ERA_model() (*in module modred.era*), 56
compute_exact_modes() (*modred.dmd.DMDHandles method*), 39
compute_inner_product_array() (*modred.vectorspace.VectorSpaceHandles method*), 63
compute_model() (*modred.era.ERA method*), 56
compute_model() (*modred.ltigalerkinproj.LTIGalerkinProjectionArray method*), 51
compute_model() (*modred.ltigalerkinproj.LTIGalerkinProjectionHand method*), 53
compute_modes() (*modred.pod.PODHandles method*), 30
compute_POD_arrays_direct_method() (*in module modred.pod*), 31
compute_POD_arrays_snaps_method() (*in module modred.pod*), 31
compute_proj_coeffs() (*modred.dmd.DMDHandles method*), 39
compute_proj_coeffs() (*modred.dmd.TLSqrDMDHandles method*), 43
compute_proj_coeffs() (*modred.pod.PODHandles method*), 30
compute_proj_modes() (*modred.dmd.DMDHandles method*), 39
compute_spectrum() (*modred.dmd.DMDHandles method*), 39
compute_spectrum()

(*modred.dmd.TLSqrDMDHandles method*), 44
compute_SVD() (*modred.bpod.BPODHandles method*), 33
compute_symm_inner_product_array() (*modred.vectorspace.VectorSpaceHandles method*), 64
compute_TLSqrDMD_arrays_direct_method() (*in module modred.dmd*), 47
compute_TLSqrDMD_arrays_snaps_method() (*in module modred.dmd*), 49

D

DMDHandles (*class in modred.dmd*), 37
drss() (*in module modred.util*), 70

E

eig_biorthog() (*in module modred.util*), 70
eigh() (*in module modred.util*), 70
ERA (*class in modred.era*), 55

F

find_assignments() (*in module modred.parallel*), 68
flatten_list() (*in module modred.util*), 70

G

get() (*modred.vectors.VecHandle method*), 61
get_adjoint_proj_coeffs() (*modred.bpod.BPODHandles method*), 35
get_adv_correlation_array() (*modred.dmd.TLSqrDMDHandles method*), 44
get_correlation_array() (*modred.dmd.DMDHandles method*), 40
get_correlation_array() (*modred.pod.PODHandles method*), 30
get_cross_correlation_array() (*modred.dmd.DMDHandles method*), 40
get_data_members() (*in module modred.util*), 70
get_decomp() (*modred.bpod.BPODHandles method*), 35
get_decomp() (*modred.dmd.DMDHandles method*), 40
get_decomp() (*modred.dmd.TLSqrDMDHandles method*), 44
get_decomp() (*modred.pod.PODHandles method*), 30
get_direct_proj_coeffs() (*modred.bpod.BPODHandles method*), 35
get_file_list() (*in module modred.util*), 70
get_Hankel_array() (*modred.bpod.BPODHandles method*), 35
get_hostname() (*in module modred.parallel*), 68
get_node_ID() (*in module modred.parallel*), 68
get_num_MPI_workers() (*in module modred.parallel*), 68
get_num_nodes() (*in module modred.parallel*), 68
get_num_procs() (*in module modred.parallel*), 68
get_proj_coeffs() (*modred.dmd.DMDHandles method*), 40
get_proj_coeffs() (*modred.pod.PODHandles method*), 30
get_proj_correlation_array() (*modred.dmd.TLSqrDMDHandles method*), 44
get_rank() (*in module modred.parallel*), 68
get_spectral_coeffs() (*modred.dmd.DMDHandles method*), 40
get_sum_correlation_array() (*modred.dmd.TLSqrDMDHandles method*), 44

H

Hankel() (*in module modred.util*), 69
Hankel_chunks() (*in module modred.util*), 69

I

impulse() (*in module modred.util*), 71
inner_product() (*modred.vectors.InnerProductTrapz method*), 61
inner_product_array_uniform() (*in module modred.vectors*), 62
InnerProductBlock (*class in modred.util*), 69
InnerProductTrapz (*class in modred.vectors*), 61
is_distributed() (*in module modred.parallel*), 68
is_rank_zero() (*in module modred.parallel*), 68

L

lin_combine() (*modred.vectorspace.VectorSpaceHandles method*), 64
load_array_text() (*in module modred.util*), 71
load_multiple_signals() (*in module modred.util*), 71
load_signals() (*in module modred.util*), 71
lsim() (*in module modred.util*), 71
LTIGalerkinProjectionArrays (*class in modred.ltigalerkinproj*), 51
LTIGalerkinProjectionHandles (*class in modred.ltigalerkinproj*), 52

M

make_iterable() (*in module modred.util*), 72
make_sampled_format() (*in module modred.era*), 57
modred.bpod (*module*), 33
modred.dmd (*module*), 37
modred.era (*module*), 55
modred.ltigalerkinproj (*module*), 51
modred.okid (*module*), 59

modred.parallel (*module*), 67
modred.pod (*module*), 29
modred.util (*module*), 69
modred.vectors (*module*), 61
modred.vectorspace (*module*), 63

O

OKID() (*in module modred.okid*), 59

P

PODHandles (*class in modred.pod*), 29
print_from_rank_zero() (*in module modred.parallel*), 68
print_msg() (*modred.vectorspace.VectorSpaceHandles method*), 65
put() (*modred.vectors.VecHandle method*), 61
put_adjoint_proj_coeffs() (*modred.bpod.BPODHandles method*), 35
put_adv_correlation_array() (*modred.dmd.TLSqrDMDHandles method*), 44
put_correlation_array() (*modred.dmd.DMDHandles method*), 40
put_correlation_array() (*modred.pod.PODHandles method*), 30
put_correlation_array_eigvals() (*modred.dmd.DMDHandles method*), 40
put_correlation_array_eigvals() (*modred.dmd.TLSqrDMDHandles method*), 44
put_correlation_array_eigvecs() (*modred.dmd.DMDHandles method*), 40
put_correlation_array_eigvecs() (*modred.dmd.TLSqrDMDHandles method*), 44
put_cross_correlation_array() (*modred.dmd.DMDHandles method*), 40
put_decomp() (*modred.bpod.BPODHandles method*), 35
put_decomp() (*modred.dmd.DMDHandles method*), 40
put_decomp() (*modred.dmd.TLSqrDMDHandles method*), 44
put_decomp() (*modred.era.ERA method*), 56
put_decomp() (*modred.pod.PODHandles method*), 30
put_direct_proj_coeffs() (*modred.bpod.BPODHandles method*), 35
put_eigvals() (*modred.dmd.DMDHandles method*), 41
put_eigvals() (*modred.pod.PODHandles method*), 30
put_eigvecs() (*modred.pod.PODHandles method*), 30
put_Hankel_array() (*modred.bpod.BPODHandles method*), 35
put_L_low_order_eigvecs() (*modred.dmd.DMDHandles method*), 40
put_L_sing_vecs() (*modred.bpod.BPODHandles method*), 35
put_model() (*modred.era.ERA method*), 56
put_proj_coeffs() (*modred.dmd.DMDHandles method*), 41
put_proj_coeffs() (*modred.pod.PODHandles method*), 31
put_proj_correlation_array() (*modred.dmd.TLSqrDMDHandles method*), 45
put_proj_correlation_array_eigvals() (*modred.dmd.TLSqrDMDHandles method*), 45
put_proj_correlation_array_eigvecs() (*modred.dmd.TLSqrDMDHandles method*), 45
put_R_low_order_eigvecs() (*modred.dmd.DMDHandles method*), 40
put_R_sing_vecs() (*modred.bpod.BPODHandles method*), 35
put_sing_vals() (*modred.bpod.BPODHandles method*), 35
put_sing_vals() (*modred.era.ERA method*), 56
put_spectral_coeffs() (*modred.dmd.DMDHandles method*), 41
put_sum_correlation_array() (*modred.dmd.TLSqrDMDHandles method*), 45
put_sum_correlation_array_eigvals() (*modred.dmd.TLSqrDMDHandles method*), 45
put_sum_correlation_array_eigvecs() (*modred.dmd.TLSqrDMDHandles method*), 45

R

reduce_A() (*modred.ltigalerkinproj.LTIGalerkinProjectionArrays method*), 52
reduce_A() (*modred.ltigalerkinproj.LTIGalerkinProjectionHandles method*), 53
reduce_B() (*modred.ltigalerkinproj.LTIGalerkinProjectionArrays method*), 52
reduce_B() (*modred.ltigalerkinproj.LTIGalerkinProjectionHandles method*), 53
reduce_C() (*modred.ltigalerkinproj.LTIGalerkinProjectionArrays method*), 52
reduce_C() (*modred.ltigalerkinproj.LTIGalerkinProjectionHandles method*), 53
rss() (*in module modred.util*), 72

S

sanity_check() (*modred.bpod.BPODHandles method*), 35
sanity_check() (*modred.dmd.DMDHandles method*), 41
sanity_check() (*modred.pod.PODHandles method*), 31

sanity_check() (*modred.vectorspace.VectorSpaceHandles method*), 65
save_array_text() (*in module modred.util*), 72
smart_eq() (*in module modred.util*), 72
standard_basis() (*in module modred.ltigalerkinproj*), 54
sum_arrays() (*in module modred.util*), 72
sum_lists() (*in module modred.util*), 72
svd() (*in module modred.util*), 72

T

TLSqrDMDHandles (*class in modred.dmd*), 41

U

UndefinedError, 69

V

VecHandle (*class in modred.vectors*), 61
VecHandleArrayText (*class in modred.vectors*), 61
VecHandleInMemory (*class in modred.vectors*), 62
VecHandlePickle (*class in modred.vectors*), 62
Vector (*class in modred.vectors*), 62
VectorSpaceArrays (*class in modred.vectorspace*), 63
VectorSpaceHandles (*class in modred.vectorspace*), 63