

---

# paragami Documentation

*Release 0.33*

**Contributors**

**Feb 08, 2019**



---

## Contents

---

<b>1</b>	<b>Installation and Testing</b>	<b>3</b>
<b>2</b>	<b>Examples</b>	<b>5</b>
<b>3</b>	<b>API</b>	<b>19</b>



Parameter folding and flattening, parameter origami<sup>1</sup>: `paragami`!

This is a library (very much still in development) intended to make sensitivity analysis easier for optimization problems. The core functionality consists of tools for “folding” and “flattening” collections of parameters – i.e., for converting data structures of constrained parameters to and from vectors of unconstrained parameters.

The purpose is to automate much of the boilerplate required to perform optimization and sensitivity analysis for statistical problems that employ optimization or estimating equations.

The functionality of `paragami` can be divided into three mutually supportive pieces:

- Tools for converting structured parameters to and from “flattened” representations,
- Tools for wrapping functions to accept flattened parameters as arguments, and
- Tools for using functions that accept flattened parameters to perform sensitivity analysis.

A good place to get started is the *Examples*.

For additional background and motivation, see the following papers:

Covariances, Robustness, and Variational Bayes

Ryan Giordano, Tamara Broderick, Michael I. Jordan

<https://arxiv.org/abs/1709.02536>

A Swiss Army Infinitesimal Jackknife

Ryan Giordano, Will Stephenson, Runjing Liu, Michael I. Jordan, Tamara Broderick

<https://arxiv.org/abs/1806.00550>

Evaluating Sensitivity to the Stick Breaking Prior in Bayesian Nonparametrics

Runjing Liu, Ryan Giordano, Michael I. Jordan, Tamara Broderick

<https://arxiv.org/abs/1810.06587>

---

<sup>1</sup> Thanks to Stéfan van der Walt for the suggesting the package name.



# CHAPTER 1

---

## Installation and Testing

---

To get the latest released version, just use pip:

```
$ pip install paragami
```

However, paragami is under active development, and you may want to install the latest version from github:

```
$ pip install git+git://github.com/rgiordan/paragami
```

To run the tests, in the root of the repository, run:

```
$ python3 -m pytest
```

To see code coverage, in the root of the repository, run:

```
$ coverage run --include='paragami/[A-Za-z]*.py' -m pytest
$ coverage html
```

Then view the `htmlcov/index.html` in your web browser.





### 2.1 Flattening and Folding With Covariance Matrices.

```
[35]: import numpy as np
import paragami
```

In this example, we will consider flattening and folding a simple symmetric positive semi-definite matrix:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}.$$

Of course, symmetry and positive semi-definiteness impose constraints on the entries  $a_{ij}$  of  $A$ .

#### 2.1.1 Flattening and Folding.

##### In the Original Space.

Let us first consider how to represent  $A$  as a vector, which we call simply *flattening*, and then as an unconstrained vector, which we call *free flattening*.

When a parameter is flattened, it is simply re-shaped as a vector. Every number that was in the original parameter will occur exactly once in the flattened shape. (In the present case of a matrix, this is exactly the same as `np.flatten`.)

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \xrightarrow{\text{flatten}} A_{\text{flat}} = \begin{bmatrix} a_{\text{flat},1} \\ a_{\text{flat},2} \\ a_{\text{flat},3} \\ a_{\text{flat},4} \\ a_{\text{flat},5} \\ a_{\text{flat},6} \\ a_{\text{flat},7} \\ a_{\text{flat},8} \\ a_{\text{flat},9} \end{bmatrix} = \begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \\ a_{31} \\ a_{32} \\ a_{33} \end{bmatrix}$$

Converting to and from  $A$  and  $A_{flat}$  can be done with the `flatten` method of a `paragami.PSDSymmetricMatrixPattern` pattern.

For the moment, because we are flattening, not free flattening, we use the option `free=False`. We will discuss the `free=True` option shortly.

```
[2]: # A sample positive semi-definite matrix.
a = np.eye(3) + np.random.random((3, 3))
a = 0.5 * (a + a.T)

# Define a pattern and fold.
a_pattern = paragami.PSDSymmetricMatrixPattern(size=3)
a_flat = a_pattern.flatten(a, free=False)

print('Now, a_flat contains the elements of a exactly as shown in the formula above.\n
→')
print('a:\n{}\n'.format(a))
print('a_flat:\n{}\n'.format(a_flat))
```

Now,  $a_{flat}$  contains the elements of  $a$  exactly as shown in the formula above.

```
a:
[[1.46982005 0.44700975 0.635101 ]
 [0.44700975 1.54334054 0.60507272]
 [0.635101   0.60507272 1.34595469]]

a_flat:
[1.46982005 0.44700975 0.635101   0.44700975 1.54334054 0.60507272
 0.635101   0.60507272 1.34595469]
```

We can also convert from  $A_{flat}$  back to  $A$  by ‘folding’.

```
[3]: print('Folding the flattened value recovers the original matrix.\n')
a_fold = a_pattern.fold(a_flat, free=False)
print('a:\n{}\n'.format(a))
print('a_fold:\n{}\n'.format(a_fold))
```

Folding the flattened value recovers the original matrix.

```
a:
[[1.46982005 0.44700975 0.635101 ]
 [0.44700975 1.54334054 0.60507272]
 [0.635101   0.60507272 1.34595469]]

a_fold:
[[1.46982005 0.44700975 0.635101 ]
 [0.44700975 1.54334054 0.60507272]
 [0.635101   0.60507272 1.34595469]]
```

By default, flattening and folding perform checks to make sure the result is a valid instance of the parameter type – in this case, a symmetric positive definite matrix.

The diagonal of a positive semi-definite matrix must not be less than 0, and folding checks this when `validate=True`, which it is by default.

```
[19]: a_flat_bad = np.array([-1, 0, 0, 0, 0, 0, 0, 0, 0])
print('A bad folded value: {}'.format(a_flat_bad))
```

(continues on next page)

(continued from previous page)

```
try:
    a_fold_bad = a_pattern.fold(a_flat_bad, free=False)
except ValueError as err:
    print('Folding with a_pattern raised the following ValueError:\n{}'.format(err))
```

A bad folded value: [-1 0 0 0 0 0 0 0 0]

Folding with a\_pattern raised the following ValueError:  
Diagonal is less than the lower bound 0.0.

If validate\_value is False, folding will produce an invalid matrix without an error.

```
[22]: a_fold_bad = a_pattern.fold(a_flat_bad, free=False, validate_value=False)
print('Folding a non-pd matrix with validate=False:\n{}'.format(a_fold_bad))
```

Folding a non-pd matrix with validate=False:

```
[[ -1  0  0]
 [  0  0  0]
 [  0  0  0]]
```

However, it will not produce a matrix of the wrong shape even when validate is False.

```
[24]: a_flat_very_bad = np.array([1, 0, 0])
print('A very bad folded value: {}'.format(a_flat_very_bad))
try:
    a_fold_very_bad = a_pattern.fold(a_flat_very_bad, free=False, validate_
    ↪value=False)
except ValueError as err:
    print('Folding with a_pattern raised the following ValueError:\n{}'.format(err))
```

A very bad folded value: [1 0 0].

Folding with a\_pattern raised the following ValueError:  
Wrong length for PSDSymmetricMatrix flat value.

You can always check validity of a folded value with the validate\_folded method of a pattern, which returns a boolean and an error message.

```
[30]: valid, msg = a_pattern.validate_folded(a_fold)
print('Valid: {}.\tMessage: {}'.format(valid, msg))

valid, msg = a_pattern.validate_folded(a_fold - 10 * np.eye(3))
print('Valid: {}.\tMessage: {}'.format(valid, msg))

Valid: True.      Message:
Valid: False.    Message: Diagonal is less than the lower bound 0.0.
```

## In an Unconstrained Space: “Free” Flattening and Folding.

Ordinary flattening converts a 3x3 symmetric PSD matrix into a 9-d vector. However, as seen above, not every 9-d vector is a valid 3x3 symmetric positive definite matrix. It is useful to have an “free” flattened representation of a parameter, where every finite value of the free flattened vector corresponds to a guaranteed valid.

To accomplish this for a symmetric positive definite matrix, we consider the Cholesky decomposition  $A_{chol}$ . This is an lower-triangular matrix with positive diagonal entries such that  $A = A_{chol}A_{chol}^T$ . By taking the log of the diagonal of  $A_{chol}$  and stacking the non-zero entries, we can construct a 6-d vector, every value of which corresponds to a

symmetric PSD matrix.

$$A \rightarrow A_{chol} = \begin{bmatrix} \alpha_{11} & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} \end{bmatrix} \rightarrow A_{freeflat} = \begin{bmatrix} \log(\alpha_{11}) \\ \alpha_{21} \\ \alpha_{31} \\ \log(\alpha_{22}) \\ \alpha_{32} \\ \log(\alpha_{33}) \end{bmatrix}.$$

The details of the freeing transform aren't important to the end user, as `paragami` takes care of the transformation behind the scenes with the option `free=True`. We denote the flattened  $A$  in the free parameterization as  $A_{freeflat}$ .

The free flat value `a_freeflat` is not immediately recognizable as a.

```
[33]: a_freeflat = a_pattern.flatten(a, free=True)
print('a:\n{}\n'.format(a))
print('a_freeflat:\n{}\n'.format(a_freeflat))

a:
[[1.46982005 0.44700975 0.635101 ]
 [0.44700975 1.54334054 0.60507272]
 [0.635101   0.60507272 1.34595469]]

a_freeflat:
[ 0.19256999  0.36870999  0.1708697   0.52385454  0.34722226 -0.02513753]
```

However, it transforms correctly back to `a` when folded.

```
[32]: a_freefold = a_pattern.fold(a_freeflat, free=True)
print('a:\n{}\n'.format(a))
print('a_fold:\n{}\n'.format(a_freefold))

a:
[[1.46982005 0.44700975 0.635101 ]
 [0.44700975 1.54334054 0.60507272]
 [0.635101   0.60507272 1.34595469]]

a_fold:
[[1.46982005 0.44700975 0.635101 ]
 [0.44700975 1.54334054 0.60507272]
 [0.635101   0.60507272 1.34595469]]
```

Any length-six vector will free fold back to a valid PSD matrix up to floating point error. Let's draw 100 random vectors, fold them, and check that this is true.

```
[34]: # Draw random free vectors and confirm that they are positive semi definite.
def assert_is_pd(mat):
    eigvals = np.linalg.eigvals(mat)
    assert np.min(eigvals) >= -1e-8

for draw in range(100):
    a_rand_freeflat = np.random.normal(scale=2, size=(6, ))
    a_rand_fold = a_pattern.fold(a_rand_freeflat, free=True)
    assert_is_pd(a_rand_fold)
```

### Default values for free.

You can set a default value for whether or not a parameter is free.

```
[49]: a_free_pattern = paragami.PSDSymmetricMatrixPattern(size=3, free_default=True)

a_freeflat = a_free_pattern.flatten(a)
print('a_freeflat:\n{}\n'.format(a_freeflat))
print('a:\n{}\n'.format(a_free_pattern.fold(a_freeflat)))
```

```
a_freeflat:
[ 0.19256999  0.36870999  0.1708697   0.52385454  0.34722226 -0.02513753]

a:
[[1.46982005 0.44700975 0.635101   ]
 [0.44700975 1.54334054 0.60507272]
 [0.635101   0.60507272 1.34595469]]

a_flat:
[1.46982005 0.44700975 0.635101   0.44700975 1.54334054 0.60507272
 0.635101   0.60507272 1.34595469]

a_flat:
[1.46982005 0.44700975 0.635101   0.44700975 1.54334054 0.60507272
 0.635101   0.60507272 1.34595469]
```

The default is be overridden by setting the argument free.

```
[51]: a_flat = a_free_pattern.flatten(a, free=False)
print('a_flat:\n{}\n'.format(a_flat))
print('a:\n{}\n'.format(a_free_pattern.fold(a_flat, free=False)))

a_flat:
[1.46982005 0.44700975 0.635101   0.44700975 1.54334054 0.60507272
 0.635101   0.60507272 1.34595469]

a:
[[1.46982005 0.44700975 0.635101   ]
 [0.44700975 1.54334054 0.60507272]
 [0.635101   0.60507272 1.34595469]]
```

You can change the default by setting the attribute free\_default.

```
[53]: # Now this pattern is misnamed!
a_free_pattern.free_default = False
print('a_flat:\n{}\n'.format(a_free_pattern.flatten(a)))

a_flat:
[1.46982005 0.44700975 0.635101   0.44700975 1.54334054 0.60507272
 0.635101   0.60507272 1.34595469]
```

An error is raised if free\_default is None and free is not specified.

```
[55]: a_free_pattern.free_default = None
try:
    a_free_pattern.flatten(a)
except ValueError as err:
    print('Folding with a_free_pattern raised the following ValueError:\n{}'.
        ↪format(err))
```

Folding with a\_free\_pattern raised the following ValueError:  
If ``free\_default`` is ``None``, ``free`` must be specified.

## 2.2 Flattening and Folding for Optimization and Frequentist Covariances.

```
[1]: import autograd
from autograd import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline
import paragami

# Use the original scipy ("osp") for functions we don't need to differentiate.
# When using scipy functions in functions that are passed to autograd,
# use autograd.scipy instead.
import scipy as osp
```

### 2.2.1 Using Flattening and Folding for Optimization.

#### An Example Model.

Suppose we are interested in optimizing some function of  $A$ , say, a normal model in which the data  $x_n \sim \mathcal{N}(0, A)$ . Specifically, Let the data be  $X = (x_1, \dots, x_N)$ , where  $x_n \in \mathbb{R}^3$ , and write a loss function as

$$\ell(X, A) = - \sum_{n=1}^N \log P(x_n|A) = \frac{1}{2} \sum_{n=1}^N (x_n^T A^{-1} x_n - \log |A|)$$

Let's simulate some data under this model.

```
[13]: np.random.seed(42)

num_obs = 100

# True value of A
true_a = np.eye(3) * np.diag(np.array([1, 2, 3])) + np.random.random((3, 3)) * 0.1
true_a = 0.5 * (true_a + true_a.T)

# Data
def draw_data(num_obs, true_a):
    return np.random.multivariate_normal(
        mean=np.zeros(3), cov=true_a, size=(num_obs, ))

x = draw_data(num_obs, true_a)
print('X shape: {}'.format(x.shape))
```

```
X shape: (100, 3)
```

We can estimate the covariance matrix using the negative log likelihood as a loss function.

```
[14]: def get_loss(x, a):
    num_obs = x.shape[0]
    a_inv = np.linalg.inv(a)
    a_det_sign, a_log_det = np.linalg.slogdet(a)
    assert a_det_sign > 0
    return 0.5 * (np.einsum('ni,ij,nj', x, a_inv, x) + num_obs * a_log_det)

print('Loss at true parameter: {}'.format(get_loss(x, true_a)))

Loss at true parameter: 242.28536625488033
```

### Using autograd and scipy.optimize with paragami.

We would like to minimize the function loss using tools like `scipy.optimize.minimize`. Standard optimization functions take vectors, not matrices, as input, and often require the vector to take valid values in the entire domain.

As-written, our loss function takes a positive definite matrix as an input. We can wrap the loss as a function of the free flattened value using the `paragami.FlattenFunctionInput` class. That is, we want to define a function  $\ell_{freeflat}$  so that

$$\ell_{freeflat}(X, A_{freeflat}) = \ell(X, A).$$

```
[17]: a_pattern = paragami.PSDSymmetricMatrixPattern(size=3)

# The arguments mean we're flattening the function get_loss, using
# the pattern a_pattern, with free parameterization, and the parameter
# is the second one (argnums uses 0-indexing like autograd).
get_freeflat_loss = paragami.FlattenFunctionInput(
    original_fun=get_loss, patterns=a_pattern, free=True, argnums=1)

print('The two losses are the same when evaluated on the folded and flat values:\n')
print('Original loss: \t\t{}'.format(get_loss(x, true_a)))
true_a_freeflat = a_pattern.flatten(true_a, free=True)
print('Free-flattened loss: \t{}'.format(
    get_freeflat_loss(x, true_a_freeflat)))

The two losses are the same when evaluated on the folded and flat values:

Original loss:          242.28536625488033
Free-flattened loss:    242.28536625488036
```

The resulting function can be passed directly to `autograd` and `scipy.optimize`, and we can estimate

$$\hat{A}_{freeflat} := \operatorname{argmin}_{A_{freeflat}} \ell_{freeflat}(X, A_{freeflat}).$$

Note that (as of writing) A bad approximation caused failure to predict improvement errors are common with second order methods even when optimization was successful. This is because `osp.optimize.minimize` only uses the norm of the gradient before multiplication by the inverse Hessian as a convergence criterion.

```
[26]: get_freeflat_loss_grad = autograd.grad(get_freeflat_loss, argnum=1)
      get_freeflat_loss_hessian = autograd.hessian(get_freeflat_loss, argnum=1)

      def get_optimum(x, init_val):
          loss_opt = osp.optimize.minimize(
              method='trust-ncg',
              x0=init_val,
              fun=lambda par: get_freeflat_loss(x, par),
              jac=lambda par: get_freeflat_loss_grad(x, par),
              hess=lambda par: get_freeflat_loss_hessian(x, par),
              options={'gtol': 1e-8, 'disp': False})
          return loss_opt

      init_val = np.zeros(a_pattern.flat_length(free=True))
      loss_opt = get_optimum(x, init_val)

      print('Optimization status: {}\nOptimal value: {}'.format(
          loss_opt.message, loss_opt.fun))

Optimization status: A bad approximation caused failure to predict improvement.
Optimal value: 239.37556057055355
```

The optimization was in the free flattened space, so to get the optimal value of  $A$  we must fold it. We can see that the optimal value is close to the true value of  $A$ , though it differs due to randomness in  $X$ .

```
[27]: optimal_freeflat_a = loss_opt.x
      optimal_a = a_pattern.fold(optimal_freeflat_a, free=True)
      print('True a:\n{}\n\nOptimal a:\n{}'.format(true_a, optimal_a))

True a:
[[1.03745401 0.07746864 0.03950388]
 [0.07746864 2.01560186 0.05110853]
 [0.03950388 0.05110853 3.0601115 ]]

Optimal a:
[[ 1.13076002 -0.16382566  0.18449819]
 [-0.16382566  1.97854146  0.3020592 ]
 [ 0.18449819  0.3020592   2.78831733]]
```

## 2.2.2 Using Flattening and Folding with the Fisher Information for Frequentist Uncertainty.

### Fisher Information and the Delta Method.

Suppose we wanted to use the Hessian of the objective (the observed Fisher information) to estimate a frequentist confidence region for  $A$ . In standard notation, covariance is of a vector, so we can write what we want in terms of  $A_{flat}$  as  $\text{Cov}(A_{flat})$ . The covariance between two elements of  $A_{flat}$  corresponds to that between two elements of  $A$ . For example, using the notation given above,

$$\begin{aligned}\text{Cov}(a_{flat,1}, a_{flat,2}) &= \text{Cov}(a_{11}, a_{12}) = \text{Cov}(a_{11}, a_{21}) \\ \text{Var}(a_{flat,4}) &= \text{Var}(a_{21}) = \text{Var}(a_{12}), \\ &\text{etc.}\end{aligned}$$



Here, we will use the observed Fisher information of  $\ell_{freeflat}$  and the Delta method to estimate  $\text{Cov}(A_{flat})$ .

$$\text{Cov}(A_{freeflat}) \approx - \left( \frac{\partial^2 \ell_{freeflat}}{\partial A_{freeflat} \partial A_{freeflat}^T} \bigg|_{\hat{A}_{freeflat}} \right)^{-1} \quad (\text{Fisher information})$$

$$\text{Cov}(A_{free}) \approx \left( \frac{dA_{free}}{dA_{freeflat}^T} \right) \text{Cov}(A_{freeflat}) \left( \frac{dA_{free}}{dA_{freeflat}^T} \right)^T \quad (\text{Delta method})$$

The Hessian required for the covariance can be calculated directly using autograd. (Note that the loss is the negative of the log likelihood.) The shape is, of course, the size of  $A_{freeflat}$ .

```
[5]: fisher_info = -1 * get_freeflat_loss_hessian(x, loss_opt.x)
print("The shape of the Fisher information amtrix is {}".format(fisher_info.shape))

The shape of the Fisher information amtrix is (6, 6).
```

The Jacobian matrix  $\frac{dA_{free}}{dA_{freeflat}^T}$  of the “unfreeing transform”  $A_{free} = A_{free}(A_{freeflat})$  is provided by paragami as a function of the *folded* parameter. Following standard notation for Jacobian matrices, the rows correspond to  $A_{flat}$ , the output of the unfreeing transform, and the columns correspond to  $A_{freeflat}$ , the input to the unfreeing transform.

By default this Jacobian matrix is sparse (in large problems, most flat parameters are independent of most free flat parameters), but a dense matrix is fine in this small problem, so we use `sparse=False`.

```
[6]: freeing_jac = a_pattern.unfreeing_jacobian(optimal_a, sparse=False)
print("The shape of the Jacobian matrix is {}".format(freeing_jac.shape))

The shape of the Jacobian matrix is (9, 6).
```

We can now plug in to estimate the covariance.

```
[7]: # Estimate the covariance of the flattened value using the Hessian at the optimum.
a_flattened_cov = -1 * freeing_jac @ np.linalg.solve(fisher_info, freeing_jac.T)
```

## A Cautionary Note on Using the Fisher Information With Constrained Variables.

Note that the estimated covariance is rank-deficient. This is expected, since, for example,  $A_{12}$  and  $A_{21}$  cannot vary independently.

```
[8]: print('The shape of the covariance matrix is {}'.format(a_flattened_cov.shape))
print('The rank of the covariance matrix is {}'.format(np.linalg.matrix_rank(a_
    ↪ flattened_cov)))

The shape of the covariance matrix is (9, 9).
The rank of the covariance matrix is 6.
```

Suppose we had erroneously defined the function  $\ell_{flat}(A_{flat})$  and tried to estimate the covariance of  $A$  using the Hessian of  $\ell_{flat}$ . Then the resulting Hessian would have been *full rank*, because the loss function `get_loss` does not enforce the constraint that  $A$  be symmetric.

```
[9]: print('An example of an erroneous use of Fisher information!')
get_flat_loss = paragami.FlattenFunctionInput(
    original_fun=get_loss, patterns=a_pattern, free=False, argnums=1)
get_flat_loss_hessian = autograd.hessian(get_flat_loss, argnum=1)
a_flat_opt = a_pattern.flatten(optimal_a, free=False)
bad_fisher_info = get_flat_loss_hessian(x, a_flat_opt)
```

(continues on next page)

(continued from previous page)

```
bad_a_flattened_cov = -1 * np.linalg.inv(bad_fisher_info)

print('The shape of the erroneous covariance matrix is {}'.format(bad_a_flattened_
    ↪cov.shape))
print('The rank of the erroneous covariance matrix is {}'.format(np.linalg.matrix_
    ↪rank(bad_a_flattened_cov)))
```

An example of an erroneous use of Fisher information!  
 The shape of the erroneous covariance matrix is (9, 9).  
 The rank of the erroneous covariance matrix is 9.

Theoretically, we are not justified using the Hessian of  $\ell_{flat}$  to estimate the covariance of its optimizer because the optimum is not “interior” – that is, the argument  $A_{flat}$  cannot take legal values in a neighborhood of the optimum, since such values may not be valid covariance matrices. Overcoming this difficulty is a key advantage of using unconstrained parameterizations.

## Inspecting and Checking the Result.

This shape of  $\text{Cov}(A_{flat})$  is inconvenient because it’s not obvious visually which entry of the flattened vector corresponds to which element of  $A$ . Again, we can use folding to put the estimated marginal standard deviations in a readable shape.

Because the result is not a valid covariance matrix, and we are just using the pattern for its shape, we set `validate` to `False`.

```
[10]: a_pattern.verify = False
a_sd = a_pattern.fold(np.sqrt(np.diag(a_flattened_cov)), free=False, validate_
    ↪value=False)
print('The marginal standard deviations of the elements of A:\n{}'.format(a_sd))
```

```
The marginal standard deviations of the elements of A:
[[0.15991362 0.15046908 0.17852051]
 [0.15046908 0.27980802 0.23681303]
 [0.17852051 0.23681303 0.39432762]]
```

As a sanity check, we can compare this estimated covariance with the variability incurred by drawing new datasets and re-optimizing.

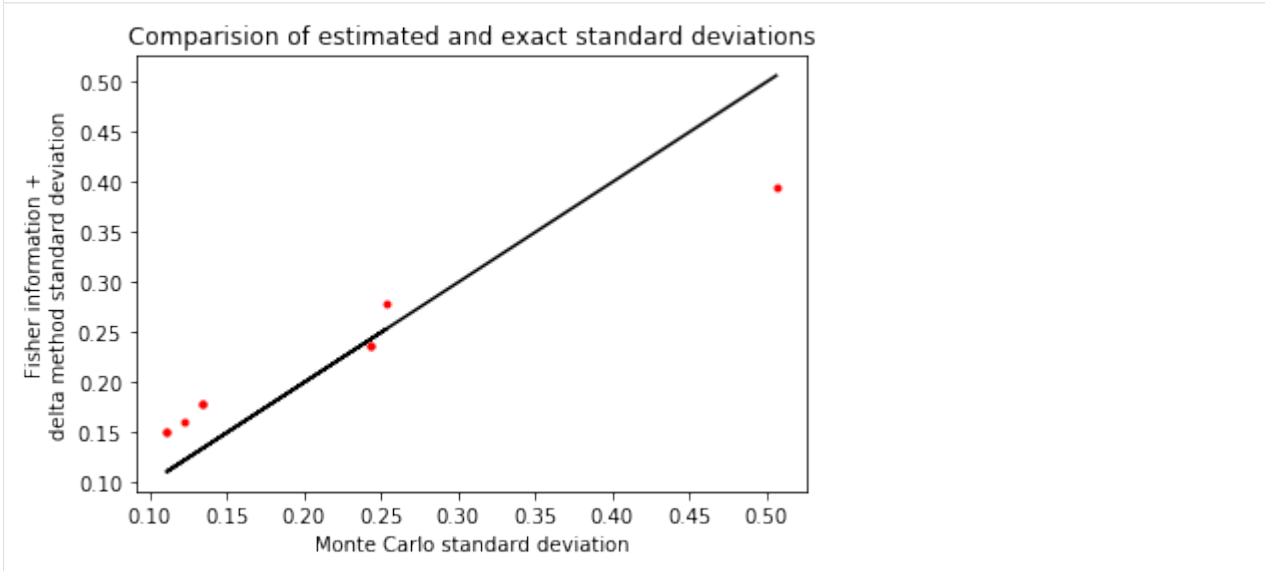
```
[11]: num_sims = 20
optimal_a_draws = np.empty((num_sims, ) + true_a.shape)
for sim in range(num_sims):
    new_x = draw_data(num_obs, true_a)
    new_loss_opt = get_optimum(new_x)
    optimal_a_draws[sim] = a_pattern.fold(new_loss_opt.x, free=True)
```

```
[12]: a_sd_monte_carlo = np.std(optimal_a_draws, axis=0)

plt.plot(a_sd_monte_carlo.flatten(), a_sd.flatten(), 'r.')
plt.plot(a_sd_monte_carlo.flatten(), a_sd_monte_carlo.flatten(), 'k')
plt.xlabel('Monte Carlo standard deviation')
plt.ylabel('Fisher information +\ndelta method standard deviation')
plt.title('Comparision of estimated and exact standard deviations')

print('Actual standard deviation:\n{}'.format(a_sd_monte_carlo))
print('Estimated standard deviation:\n{}'.format(a_sd))
```

```
Actual standard deviation:
[[0.12203849 0.11104834 0.13347622]
 [0.11104834 0.25363392 0.24355346]
 [0.13347622 0.24355346 0.50647563]]
Estimated standard deviation:
[[0.15991362 0.15046908 0.17852051]
 [0.15046908 0.27980802 0.23681303]
 [0.17852051 0.23681303 0.39432762]]
```



## 2.3 Default values for `free`.

```
[1]: import numpy as np
import paragami
```

You can set a default value for whether or not a parameter is free.

```
[4]: a = np.eye(3) + np.random.random((3, 3))
a = 0.5 * (a + a.T)

a_free_pattern = paragami.PSDSymmetricMatrixPattern(size=3, free_default=True)

a_freeflat = a_free_pattern.flatten(a)
print('a_freeflat:\n{}\n'.format(a_freeflat))
print('a:\n{}\n'.format(a_free_pattern.fold(a_freeflat)))

a_freeflat:
[ 0.13919912  0.41973416  0.12037979  0.7396427  0.44432253 -0.06185827]

a:
[[1.32101217 0.48242269 0.85011051]
 [0.48242269 1.4483919  0.81161586]
 [0.85011051 0.81161586 1.6281241  ]]
```

The default is be overridden by setting the argument `free`.

```
[5]: a_flat = a_free_pattern.flatten(a, free=False)
print('a_flat:\n{}\n'.format(a_flat))
print('a:\n{}\n'.format(a_free_pattern.fold(a_flat, free=False)))

a_flat:
[1.32101217 0.48242269 0.85011051 0.48242269 1.4483919 0.81161586
 0.85011051 0.81161586 1.6281241 ]

a:
[[1.32101217 0.48242269 0.85011051]
 [0.48242269 1.4483919 0.81161586]
 [0.85011051 0.81161586 1.6281241 ]]
```

You can change the default by setting the attribute `free_default`.

```
[6]: # Now this pattern is misnamed!
a_free_pattern.free_default = False
print('a_flat:\n{}\n'.format(a_free_pattern.flatten(a)))

a_flat:
[1.32101217 0.48242269 0.85011051 0.48242269 1.4483919 0.81161586
 0.85011051 0.81161586 1.6281241 ]
```

An error is raised if `free_default` is `None` and `free` is not specified.

```
[7]: a_free_pattern.free_default = None
try:
    a_free_pattern.flatten(a)
except ValueError as err:
    print('Folding with a_free_pattern raised the following ValueError:\n{}'.
    ↪format(err))
```

```
Folding with a_free_pattern raised the following ValueError:
If ``free_default`` is ``None``, ``free`` must be specified.
```

Pattern containers override the default values of their contents so you don't accidentally mix free and non-free flattened values.

```
[19]: dict_pattern = paragami.PatternDict(free_default=True)

dict_pattern['a1'] = paragami.PSDSymmetricMatrixPattern(size=3, free_default=False)
dict_pattern['a2'] = paragami.PSDSymmetricMatrixPattern(size=3, free_default=True)

print('\nThis pattern alone is non-free by default:')
print(dict_pattern['a1'].flatten(a))

print('\nThis pattern alone is free by default:')
print(dict_pattern['a2'].flatten(a))

print('\nBut the dictionary pattern overrides the default:')
param_dict = { 'a1': a, 'a2': a }
print(dict_pattern.flatten(param_dict))

print('\nIf no default is specified, an error is raised ' +
```

(continues on next page)

(continued from previous page)

```
'so that you do not accidentally mix free and non-free flat values.')
dict_pattern_noddefault = paragami.PatternDict()
try:
    dict_pattern_noddefault.flatten(param_dict)
except ValueError as err:
    print('Folding a container with no default raised the follding ValueError:\n{}'.
        ↪format(err))
```

This pattern alone is non-free by default:

```
[1.32101217 0.48242269 0.85011051 0.48242269 1.4483919  0.81161586
 0.85011051 0.81161586 1.6281241 ]
```

This pattern alone is free by default:

```
[ 0.13919912  0.41973416  0.12037979  0.7396427   0.44432253 -0.06185827]
```

But the dictionary pattern overrides the default:

```
[ 0.13919912  0.41973416  0.12037979  0.7396427   0.44432253 -0.06185827
 0.13919912  0.41973416  0.12037979  0.7396427   0.44432253 -0.06185827]
```

If no default is specified, an error is raised so that you do not accidentally mix ↪  
↪free and non-free flat values.

Folding a container with no default raised the follding ValueError:

If ``free\_default`` is ``None``, ``free`` must be specified.



### 3.1 Pattern Parent Class

Every pattern described herein inherits from the parent *Pattern* class and implements its methods.

**class** `paragami.base_patterns.Pattern` (*flat\_length*, *free\_flat\_length*, *free\_default=None*)  
 A abstract class for a parameter pattern.

See derived classes for examples.

**\_\_init\_\_** (*flat\_length*, *free\_flat\_length*, *free\_default=None*)

#### Parameters

**flat\_length** [*int*] The length of a non-free flattened vector.

**free\_flat\_length** [*int*] The length of a free flattened vector.

**as\_dict** ()

Return a dictionary of attributes describing the pattern.

The dictionary should completely describe the pattern in the sense that if the contents of two patterns' dictionaries are identical the patterns should be considered identical.

If the keys of the returned dictionary match the arguments to **\_\_init\_\_**, then the default methods for **to\_json** and **from\_json** will work with no additional modification.

**empty** (*valid*)

Return an empty parameter in its folded shape.

#### Parameters

**valid** [*bool*] Whether or folded shape should be filled with valid values.

#### Returns

**folded\_val** [Folded value] A parameter value in its original folded shape.

**empty\_bool** (*value*)

Return folded shape containing booleans.

#### Parameters

**value** [*bool*] The value with which to fill the folded shape.

#### Returns

**folded\_bool** [Folded value] A boolean value in its original folded shape.

**flat\_indices** (*folded\_bool, free=None*)

Get which flattened indices correspond to which folded values.

#### Parameters

**folded\_bool** [Folded booleans] A variable in the folded shape but containing booleans. The elements that are `True` are the ones for which we will return the flat indices.

**free** [*bool*] Whether or not the flattened value is to be in a free parameterization. If not specified, the attribute `free_default` is used.

#### Returns

**indices** [*numpy.ndarray* (N,)] A list of indices into the flattened value corresponding to the `True` members of `folded_bool`.

**flat\_length** (*free=None*)

Return the length of the pattern's flattened value.

#### Parameters

**free** [*bool*, optional] Whether or not the flattened value is to be in a free parameterization. If not specified, `free_default` is used.

#### Returns

**length** [*int*] The length of the pattern's flattened value.

**flatten** (*folded\_val, free=None, validate\_value=None*)

Flatten a folded value into a flat vector.

#### Parameters

**folded\_val** [Folded value] The parameter in its original folded shape.

**free** [*bool*, optional] Whether or not the flattened value is to be in a free parameterization. If not specified, the attribute `free_default` is used.

**validate\_value** [*bool*] Whether to check that the folded value is valid. If `None`, the pattern will employ a default behavior.

#### Returns

**flat\_val** [*numpy.ndarray*, (N,)] The flattened value.

**fold** (*flat\_val, free=None, validate\_value=None*)

Fold a flat value into a parameter.

#### Parameters

**flat\_val** [*numpy.ndarray*, (N,)] The flattened value.

**free** [*bool*, optional.] Whether or not the flattened value is a free parameterization. If not specified, the attribute `free_default` is used.

**validate\_value** [*bool*, optional.] Whether to check that the folded value is valid. If `None`, the pattern will employ a default behavior.

#### Returns



**folded\_val** [Folded value] The parameter value in its original folded shape.

**freeing\_jacobian** (*folded\_val*, *sparse=True*)

The Jacobian of the map from a flat free value to a flat value.

If the folded value of the parameter is *val*, *val\_flat* = *flatten(val, free=False)*, and *val\_freeflat* = *flatten(val, free=True)*, then this calculates the Jacobian matrix *d val\_free / d val\_freeflat*. For entries with no dependence between them, the Jacobian is taken to be zero.

#### Parameters

**folded\_val** [Folded value] The folded value at which the Jacobian is to be evaluated.

**sparse** [*bool*, optional] Whether to return a sparse or a dense matrix.

#### Returns

“**numpy.ndarray**“, (N, M) The Jacobian matrix *d val\_free / d val\_freeflat*. Consistent with standard Jacobian notation, the elements of *val\_free* correspond to the rows of the Jacobian matrix and the elements of *val\_freeflat* correspond to the columns.

#### See also:

*Pattern.unfreeing\_jacobian*

**classmethod from\_json** (*json\_string*)

Return a pattern from *json\_string* created by *to\_json*.

#### See also:

*Pattern.to\_json*

**random** ()

Return an random, valid parameter in its folded shape.

---

**Note:** There is no reason this provides a meaningful distribution over folded values. This function is intended to be used as a convenience for testing.

---

#### Returns

**folded\_val** [Folded value] A random parameter value in its original folded shape.

**to\_json** ()

Return a JSON representation of the pattern.

#### See also:

*Pattern.from\_json*

**unfreeing\_jacobian** (*folded\_val*, *sparse=True*)

The Jacobian of the map from a flat value to a flat free value.

If the folded value of the parameter is *val*, *val\_flat* = *flatten(val, free=False)*, and *val\_freeflat* = *flatten(val, free=True)*, then this calculates the Jacobian matrix *d val\_freeflat / d val\_free*. For entries with no dependence between them, the Jacobian is taken to be zero.

#### Parameters

**folded\_val** [Folded value] The folded value at which the Jacobian is to be evaluated.

**sparse** [*bool*, optional] If `True`, return a sparse matrix. Otherwise, return a dense numpy 2d array.

#### Returns

**“numpy.ndarray“, (N, N)** The Jacobian matrix  $d \text{ val\_freeflat} / d \text{ val\_free}$ . Consistent with standard Jacobian notation, the elements of `val_freeflat` correspond to the rows of the Jacobian matrix and the elements of `val_free` correspond to the columns.

#### See also:

*Pattern.freeing\_jacobian*

**validate\_folded** (*folded\_val*, *validate\_value=None*)

Check whether a folded value is valid.

#### Parameters

**folded\_val** [Folded value] A parameter value in its original folded shape.

**validate\_value** [*bool*] Whether to validate the value in addition to the shape. The shape is always validated.

#### Returns

**is\_valid** [*bool*] Whether `folded_val` is an allowable shape and value.

**err\_msg** [*str*]

## 3.2 Numeric patterns

### 3.2.1 Numeric Arrays

**class** `paragami.numeric_array_patterns.NumericArrayPattern` (*shape*, *lb=-inf*, *ub=inf*, *default\_validate=True*, *free\_default=None*)

A pattern for (optionally bounded) arrays of numbers.

#### Attributes

**default\_validate: ‘bool’, optional** Whether or not the array is checked by default to lie within the specified bounds.

**\_\_init\_\_** (*shape*, *lb=-inf*, *ub=inf*, *default\_validate=True*, *free\_default=None*)

#### Parameters

**shape: ‘tuple’ of ‘int’** The shape of the array.

**lb: ‘float’** The (inclusive) lower bound for the entries of the array.

**ub: ‘float’** The (inclusive) upper bound for the entries of the array.

**default\_validate: ‘bool’, optional** Whether or not the array is checked by default to lie within the specified bounds.

**free\_default: ‘bool’, optional** Whether the pattern is free by default.

**as\_dict** ()

Return a dictionary of attributes describing the pattern.

The dictionary should completely describe the pattern in the sense that if the contents of two patterns' dictionaries are identical the patterns should be considered identical.

If the keys of the returned dictionary match the arguments to `__init__`, then the default methods for `to_json` and `from_json` will work with no additional modification.

**empty** (*valid*)

Return an empty parameter in its folded shape.

#### Parameters

**valid** [*bool*] Whether or folded shape should be filled with valid values.

#### Returns

**folded\_val** [Folded value] A parameter value in its original folded shape.

**flat\_indices** (*folded\_bool*, *free=None*)

Get which flattened indices correspond to which folded values.

#### Parameters

**folded\_bool** [Folded booleans] A variable in the folded shape but containing booleans. The elements that are `True` are the ones for which we will return the flat indices.

**free** [*bool*] Whether or not the flattened value is to be in a free parameterization. If not specified, the attribute `free_default` is used.

#### Returns

**indices** [*numpy.ndarray* (N,)] A list of indices into the flattened value corresponding to the `True` members of `folded_bool`.

**flat\_length** (*free=None*)

Return the length of the pattern's flattened value.

#### Parameters

**free** [*bool*, optional] Whether or not the flattened value is to be in a free parameterization. If not specified, `free_default` is used.

#### Returns

**length** [*int*] The length of the pattern's flattened value.

**flatten** (*folded\_val*, *free=None*, *validate\_value=None*)

Flatten a folded value into a flat vector.

#### Parameters

**folded\_val** [Folded value] The parameter in its original folded shape.

**free** [*bool*, optional] Whether or not the flattened value is to be in a free parameterization. If not specified, the attribute `free_default` is used.

**validate\_value** [*bool*] Whether to check that the folded value is valid. If `None`, the pattern will employ a default behavior.

#### Returns

**flat\_val** [*numpy.ndarray*, (N,)] The flattened value.

**fold** (*flat\_val*, *free=None*, *validate\_value=None*)

Fold a flat value into a parameter.

#### Parameters

**flat\_val** [*numpy.ndarray*, (N, )] The flattened value.

**free** [*bool*, optional.] Whether or not the flattened value is a free parameterization. If not specified, the attribute `free_default` is used.

**validate\_value** [*bool*, optional.] Whether to check that the folded value is valid. If `None`, the pattern will employ a default behavior.

#### Returns

**folded\_val** [Folded value] The parameter value in its original folded shape.

**validate\_folded** (*folded\_val*, *validate\_value=None*)  
Check whether a folded value is valid.

#### Parameters

**folded\_val** [Folded value] A parameter value in its original folded shape.

**validate\_value** [*bool*] Whether to validate the value in addition to the shape. The shape is always validated.

#### Returns

**is\_valid** [*bool*] Whether `folded_val` is an allowable shape and value.

**err\_msg** [*str*]

### 3.2.2 Symmetric Positive Definite Matrices

```
class paragami.psdmatrix_patterns.PSDSymmetricMatrixPattern (size,
                                                                diag_lb=0.0,    de-
                                                                fault_validate=True,
                                                                free_default=None)
```

A pattern for a symmetric, positive-definite matrix parameter.

#### Attributes

**validate:** **Bool** Whether or not the matrix is automatically checked for symmetry positive-definiteness, and the diagonal lower bound.

**\_\_init\_\_** (*size*, *diag\_lb=0.0*, *default\_validate=True*, *free\_default=None*)

#### Parameters

**size:** **'int'** The length of one side of the square matrix.

**diag\_lb:** **'float'** A lower bound for the diagonal entries. Must be  $\geq 0$ .

**default\_validate:** **'bool', optional** Whether or not to check for legal (i.e., symmetric positive-definite) folded values by default.

**free\_default:** **'bool', optional** Default setting for free.

**as\_dict** ()

Return a dictionary of attributes describing the pattern.

The dictionary should completely describe the pattern in the sense that if the contents of two patterns' dictionaries are identical the patterns should be considered identical.

If the keys of the returned dictionary match the arguments to `__init__`, then the default methods for `to_json` and `from_json` will work with no additional modification.

**diag\_lb** ()

Returns the diagonal lower bound.

**empty** (*valid*)

Return an empty parameter in its folded shape.

**Parameters**

**valid** [*bool*] Whether or folded shape should be filled with valid values.

**Returns**

**folded\_val** [Folded value] A parameter value in its original folded shape.

**flat\_indices** (*folded\_bool*, *free=None*)

Get which flattened indices correspond to which folded values.

**Parameters**

**folded\_bool** [Folded booleans] A variable in the folded shape but containing booleans. The elements that are `True` are the ones for which we will return the flat indices.

**free** [*bool*] Whether or not the flattened value is to be in a free parameterization. If not specified, the attribute `free_default` is used.

**Returns**

**indices** [*numpy.ndarray* (N,)] A list of indices into the flattened value corresponding to the `True` members of `folded_bool`.

**flatten** (*folded\_val*, *free=None*, *validate\_value=None*)

Flatten a folded value into a flat vector.

**Parameters**

**folded\_val** [Folded value] The parameter in its original folded shape.

**free** [*bool*, optional] Whether or not the flattened value is to be in a free parameterization. If not specified, the attribute `free_default` is used.

**validate\_value** [*bool*] Whether to check that the folded value is valid. If `None`, the pattern will employ a default behavior.

**Returns**

**flat\_val** [*numpy.ndarray*, (N,)] The flattened value.

**fold** (*flat\_val*, *free=None*, *validate\_value=None*)

Fold a flat value into a parameter.

**Parameters**

**flat\_val** [*numpy.ndarray*, (N,)] The flattened value.

**free** [*bool*, optional.] Whether or not the flattened value is a free parameterization. If not specified, the attribute `free_default` is used.

**validate\_value** [*bool*, optional.] Whether to check that the folded value is valid. If `None`, the pattern will employ a default behavior.

**Returns**

**folded\_val** [Folded value] The parameter value in its original folded shape.

**shape** ()

Returns the matrix shape, i.e., (size, size).

**size** ()

Returns the matrix size.

**validate\_folded** (*folded\_val*, *validate\_value=None*)

Check that the folded value is valid.

If *validate\_value = True*, checks that *folded\_val* is a symmetric, matrix of the correct shape with diagonal entries greater than the specified lower bound. Otherwise, only the shape is checked.

---

**Note:** This method does not currently check for positive-definiteness.

---

#### Parameters

**folded\_val** [Folded value] A candidate value for a positive definite matrix.

**validate\_value: 'bool', optional** Whether to check the matrix for attributes other than shape. If *None*, the value of *self.default\_validate* is used.

#### Returns

**is\_valid** [*bool*] Whether *folded\_val* is a valid positive semi-definite matrix.

**err\_msg** [*str*] A message describing the reason the value is invalid or an empty string if the value is valid.

### 3.2.3 Simplexes

**class** `paragami.simplex_patterns.SimplexArrayPattern` (*simplex\_size*, *array\_shape*,  
*default\_validate=True*,  
*free\_default=None*)

A pattern for an array of simplex parameters.

The last index represents entries of the simplex. For example, if *array\_shape*=(2, 3) and *simplex\_size*=4, then the pattern is for a 2x3 array of 4d simplexes. If such value of the simplex array is given by *val*, then *val.shape* = (2, 3, 4) and *val[i, j, :]* is the *i,j'th* of the six simplicial vectors, i.e., '*np.sum(val[i, j, :])*' equals 1 for each *i* and *j*.

#### Attributes

**default\_validate: Bool** Whether or not the simplex is checked by default to be non-negative and to sum to one.

#### Methods

<b>array_shape: tuple of ints</b>	The shape of the array of simplexes, not including the simplex dimension.
<b>simplex_size: int</b>	The length of each simplex.
<b>shape: tuple of ints</b>	The shape of the entire array including the simplex dimension.

**\_\_init\_\_** (*simplex\_size*, *array\_shape*, *default\_validate=True*, *free\_default=None*)

#### Parameters

**simplex\_size: 'int'** The length of the simplexes.

**array\_shape: 'tuple' of 'int'** The size of the array of simplexes (not including the simplexes themselves).

**default\_validate: 'bool', optional** Whether or not to check for legal (i.e., positive and normalized) folded values by default.

**free\_default: 'bool', optional** The default value for free.

**as\_dict()**

Return a dictionary of attributes describing the pattern.

The dictionary should completely describe the pattern in the sense that if the contents of two patterns' dictionaries are identical the patterns should be considered identical.

If the keys of the returned dictionary match the arguments to `__init__`, then the default methods for `to_json` and `from_json` will work with no additional modification.

**empty** (*valid*)

Return an empty parameter in its folded shape.

#### Parameters

**valid** [*bool*] Whether or folded shape should be filled with valid values.

#### Returns

**folded\_val** [Folded value] A parameter value in its original folded shape.

**flat\_indices** (*folded\_bool*, *free=None*)

Get which flattened indices correspond to which folded values.

#### Parameters

**folded\_bool** [Folded booleans] A variable in the folded shape but containing booleans. The elements that are `True` are the ones for which we will return the flat indices.

**free** [*bool*] Whether or not the flattened value is to be in a free parameterization. If not specified, the attribute `free_default` is used.

#### Returns

**indices** [*numpy.ndarray* (N,)] A list of indices into the flattened value corresponding to the `True` members of `folded_bool`.

**flatten** (*folded\_val*, *free=None*, *validate\_value=None*)

Flatten a folded value into a flat vector.

#### Parameters

**folded\_val** [Folded value] The parameter in its original folded shape.

**free** [*bool*, optional] Whether or not the flattened value is to be in a free parameterization. If not specified, the attribute `free_default` is used.

**validate\_value** [*bool*] Whether to check that the folded value is valid. If `None`, the pattern will employ a default behavior.

#### Returns

**flat\_val** [*numpy.ndarray*, (N,)] The flattened value.

**fold** (*flat\_val*, *free=None*, *validate\_value=None*)

Fold a flat value into a parameter.

#### Parameters

**flat\_val** [*numpy.ndarray*, (N,)] The flattened value.

**free** [*bool*, optional.] Whether or not the flattened value is a free parameterization. If not specified, the attribute `free_default` is used.

**validate\_value** [*bool*, optional.] Whether to check that the folded value is valid. If `None`, the pattern will employ a default behavior.

#### Returns

**folded\_val** [Folded value] The parameter value in its original folded shape.

**classmethod from\_json** (*json\_string*)

Return a pattern instance from *json\_string* created by *to\_json*.

**validate\_folded** (*folded\_val*, *validate\_value=None*)

Check whether a folded value is valid.

#### Parameters

**folded\_val** [Folded value] A parameter value in its original folded shape.

**validate\_value** [*bool*] Whether to validate the value in addition to the shape. The shape is always validated.

#### Returns

**is\_valid** [*bool*] Whether *folded\_val* is an allowable shape and value.

**err\_msg** [*str*]

## 3.3 Containers of Patterns

Containers of patterns are themselves patterns, and so can contain instantiations of themselves.

### 3.3.1 Dictionaries of Patterns

**class** `paragami.pattern_containers.PatternDict` (*free\_default=None*)

A dictionary of patterns (which is itself a pattern).

#### Examples

```
import paragami

# Add some patterns.
dict_pattern = paragami.PatternDict()
dict_pattern['vec'] = paragami.NumericArrayPattern(shape=(2, ))
dict_pattern['mat'] = paragami.PSDSymmetricMatrixPattern(size=3)

# Dictionaries can also contain dictionaries (but they have to
# be populated /before/ being added to the parent).
sub_dict_pattern = paragami.PatternDict()
sub_dict_pattern['vec1'] = paragami.NumericArrayPattern(shape=(2, ))
sub_dict_pattern['vec2'] = paragami.NumericArrayPattern(shape=(2, ))
dict_pattern['sub_dict'] = sub_dict_pattern

# We're done adding patterns, so lock the dictionary.
dict_pattern.lock()

# Get a random initial value for the whole dictionary.
dict_val = dict_pattern.random()
print(dict_val['mat']) # Prints a 3x3 positive definite numpy matrix.

# Get a flattened value of the whole dictionary.
```

(continues on next page)



(continued from previous page)

```
dict_val_flat = dict_pattern.flatten(dict_val, free=True)

# Get a new random folded value of the dictionary.
new_dict_val_flat = np.random.random(len(dict_val_flat))
new_dict_val = dict_pattern.fold(new_dict_val_flat, free=True)
```

## Methods

<b>lock:</b>	Prevent additional patterns from being added or removed.
--------------	--

**\_\_init\_\_** (*free\_default=None*)

### Parameters

**flat\_length** [*int*] The length of a non-free flattened vector.

**free\_flat\_length** [*int*] The length of a free flattened vector.

**as\_dict** ()

Return a dictionary of attributes describing the pattern.

The dictionary should completely describe the pattern in the sense that if the contents of two patterns' dictionaries are identical the patterns should be considered identical.

If the keys of the returned dictionary match the arguments to **\_\_init\_\_**, then the default methods for **to\_json** and **from\_json** will work with no additional modification.

**empty** (*valid*)

Return an empty parameter in its folded shape.

### Parameters

**valid** [*bool*] Whether or folded shape should be filled with valid values.

### Returns

**folded\_val** [Folded value] A parameter value in its original folded shape.

**flat\_indices** (*folded\_bool, free=None*)

Get which flattened indices correspond to which folded values.

### Parameters

**folded\_bool** [Folded booleans] A variable in the folded shape but containing booleans. The elements that are **True** are the ones for which we will return the flat indices.

**free** [*bool*] Whether or not the flattened value is to be in a free parameterization. If not specified, the attribute **free\_default** is used.

### Returns

**indices** [*numpy.ndarray* (N,)] A list of indices into the flattened value corresponding to the **True** members of **folded\_bool**.

**flatten** (*folded\_val, free=None, validate\_value=None*)

Flatten a folded value into a flat vector.

### Parameters

**folded\_val** [Folded value] The parameter in its original folded shape.

**free** [*bool*, optional] Whether or not the flattened value is to be in a free parameterization. If not specified, the attribute `free_default` is used.

**validate\_value** [*bool*] Whether to check that the folded value is valid. If `None`, the pattern will employ a default behavior.

#### Returns

**flat\_val** [*numpy.ndarray*, (N, )] The flattened value.

**fold** (*flat\_val*, *free=None*, *validate\_value=None*)  
Fold a flat value into a parameter.

#### Parameters

**flat\_val** [*numpy.ndarray*, (N, )] The flattened value.

**free** [*bool*, optional.] Whether or not the flattened value is a free parameterization. If not specified, the attribute `free_default` is used.

**validate\_value** [*bool*, optional.] Whether to check that the folded value is valid. If `None`, the pattern will employ a default behavior.

#### Returns

**folded\_val** [Folded value] The parameter value in its original folded shape.

**freeing\_jacobian** (*folded\_val*, *sparse=True*)  
The Jacobian of the map from a flat free value to a flat value.

If the folded value of the parameter is `val`, `val_flat = flatten(val, free=False)`, and `val_freeflat = flatten(val, free=True)`, then this calculates the Jacobian matrix  $d \text{ val\_free} / d \text{ val\_freeflat}$ . For entries with no dependence between them, the Jacobian is taken to be zero.

#### Parameters

**folded\_val** [Folded value] The folded value at which the Jacobian is to be evaluated.

**sparse** [*bool*, optional] Whether to return a sparse or a dense matrix.

#### Returns

“*numpy.ndarray*“, (N, M) The Jacobian matrix  $d \text{ val\_free} / d \text{ val\_freeflat}$ . Consistent with standard Jacobian notation, the elements of `val_free` correspond to the rows of the Jacobian matrix and the elements of `val_freeflat` correspond to the columns.

#### See also:

`Pattern.unfreeing_jacobian`

**classmethod from\_json** (*json\_string*)  
Return a pattern from `json_string` created by `to_json`.

#### See also:

`Pattern.to_json`

**unfreeing\_jacobian** (*folded\_val*, *sparse=True*)  
The Jacobian of the map from a flat value to a flat free value.

If the folded value of the parameter is `val`, `val_flat = flatten(val, free=False)`, and `val_freeflat = flatten(val, free=True)`, then this calculates the Jacobian matrix  $d \text{ val\_freeflat} / d \text{ val\_free}$ . For entries with no dependence between them, the Jacobian is taken to be zero.

**Parameters**

**folded\_val** [Folded value] The folded value at which the Jacobian is to be evaluated.

**sparse** [*bool*, optional] If *True*, return a sparse matrix. Otherwise, return a dense numpy 2d array.

**Returns**

“**numpy.ndarray**“, (N, N) The Jacobian matrix  $d \text{ val\_freeflat} / d \text{ val\_free}$ . Consistent with standard Jacobian notation, the elements of *val\_freeflat* correspond to the rows of the Jacobian matrix and the elements of *val\_free* correspond to the columns.

**See also:**

`Pattern.freeing_jacobian`

**validate\_folded** (*folded\_val*, *validate\_value=None*)

Check whether a folded value is valid.

**Parameters**

**folded\_val** [Folded value] A parameter value in its original folded shape.

**validate\_value** [*bool*] Whether to validate the value in addition to the shape. The shape is always validated.

**Returns**

**is\_valid** [*bool*] Whether *folded\_val* is an allowable shape and value.

**err\_msg** [*str*]

### 3.3.2 Arrays of Patterns

**class** `paragami.pattern_containers.PatternArray` (*array\_shape*, *base\_pattern*, *free\_default=None*)

An array of a pattern (which is also itself a pattern).

The first indices of the folded pattern are the array and the final indices are of the base pattern. For example, if *shape*=(3, 4) and *base\_pattern* = *PSDSymmetricMatrixPattern*(*size*=5), then the folded value of the array will have shape (3, 4, 5, 5), where the entry *folded\_val*[*i*, *j*, :, :] is a 5x5 positive definite matrix.

Currently this can only contain patterns whose folded values are numeric arrays (i.e., *NumericArrayPattern*, *SimplexArrayPattern*, and *PSDSymmetricMatrixPattern*).

**\_\_init\_\_** (*array\_shape*, *base\_pattern*, *free\_default=None*)

**Parameters**

**array\_shape: tuple of int** The shape of the array (not including the base parameter)

**base\_pattern:** The base pattern.

**array\_shape** ()

The shape of the array of parameters.

This does not include the dimension of the folded parameters.

**as\_dict** ()

Return a dictionary of attributes describing the pattern.

The dictionary should completely describe the pattern in the sense that if the contents of two patterns' dictionaries are identical the patterns should be considered identical.

If the keys of the returned dictionary match the arguments to `__init__`, then the default methods for `to_json` and `from_json` will work with no additional modification.

**empty** (*valid*)

Return an empty parameter in its folded shape.

**Parameters**

**valid** [*bool*] Whether or folded shape should be filled with valid values.

**Returns**

**folded\_val** [Folded value] A parameter value in its original folded shape.

**flat\_indices** (*folded\_bool*, *free=None*)

Get which flattened indices correspond to which folded values.

**Parameters**

**folded\_bool** [Folded booleans] A variable in the folded shape but containing booleans. The elements that are `True` are the ones for which we will return the flat indices.

**free** [*bool*] Whether or not the flattened value is to be in a free parameterization. If not specified, the attribute `free_default` is used.

**Returns**

**indices** [*numpy.ndarray* (N,)] A list of indices into the flattened value corresponding to the `True` members of `folded_bool`.

**flat\_length** (*free=None*)

Return the length of the pattern's flattened value.

**Parameters**

**free** [*bool*, optional] Whether or not the flattened value is to be in a free parameterization. If not specified, `free_default` is used.

**Returns**

**length** [*int*] The length of the pattern's flattened value.

**flatten** (*folded\_val*, *free=None*, *validate\_value=None*)

Flatten a folded value into a flat vector.

**Parameters**

**folded\_val** [Folded value] The parameter in its original folded shape.

**free** [*bool*, optional] Whether or not the flattened value is to be in a free parameterization. If not specified, the attribute `free_default` is used.

**validate\_value** [*bool*] Whether to check that the folded value is valid. If `None`, the pattern will employ a default behavior.

**Returns**

**flat\_val** [*numpy.ndarray*, (N,)] The flattened value.

**fold** (*flat\_val*, *free=None*, *validate\_value=None*)

Fold a flat value into a parameter.

**Parameters**

**flat\_val** [*numpy.ndarray*, (N,)] The flattened value.

**free** [*bool*, optional.] Whether or not the flattened value is a free parameterization. If not specified, the attribute `free_default` is used.

**validate\_value** [*bool*, optional.] Whether to check that the folded value is valid. If `None`, the pattern will employ a default behavior.

#### Returns

**folded\_val** [Folded value] The parameter value in its original folded shape.

**freeing\_jacobian** (*folded\_val*, *sparse=True*)

The Jacobian of the map from a flat free value to a flat value.

If the folded value of the parameter is `val`, `val_flat = flatten(val, free=False)`, and `val_freeflat = flatten(val, free=True)`, then this calculates the Jacobian matrix  $d \text{ val\_free} / d \text{ val\_freeflat}$ . For entries with no dependence between them, the Jacobian is taken to be zero.

#### Parameters

**folded\_val** [Folded value] The folded value at which the Jacobian is to be evaluated.

**sparse** [*bool*, optional] Whether to return a sparse or a dense matrix.

#### Returns

“**numpy.ndarray**“, (**N**, **M**) The Jacobian matrix  $d \text{ val\_free} / d \text{ val\_freeflat}$ . Consistent with standard Jacobian notation, the elements of `val_free` correspond to the rows of the Jacobian matrix and the elements of `val_freeflat` correspond to the columns.

#### See also:

`Pattern.unfreeing_jacobian`

**classmethod from\_json** (*json\_string*)

Return a pattern from `json_string` created by `to_json`.

#### See also:

`Pattern.to_json`

**shape** ()

The shape of a folded value.

**unfreeing\_jacobian** (*folded\_val*, *sparse=True*)

The Jacobian of the map from a flat value to a flat free value.

If the folded value of the parameter is `val`, `val_flat = flatten(val, free=False)`, and `val_freeflat = flatten(val, free=True)`, then this calculates the Jacobian matrix  $d \text{ val\_freeflat} / d \text{ val\_free}$ . For entries with no dependence between them, the Jacobian is taken to be zero.

#### Parameters

**folded\_val** [Folded value] The folded value at which the Jacobian is to be evaluated.

**sparse** [*bool*, optional] If `True`, return a sparse matrix. Otherwise, return a dense numpy 2d array.

#### Returns

“**numpy.ndarray**“, (**N**, **N**) The Jacobian matrix  $d \text{ val\_freeflat} / d \text{ val\_free}$ . Consistent with standard Jacobian notation, the elements of `val_freeflat` correspond

to the rows of the Jacobian matrix and the elements of `val_free` correspond to the columns.

**See also:**

`Pattern.freeing_jacobian`

**validate\_folded** (*folded\_val*, *validate\_value=None*)

Check whether a folded value is valid.

**Parameters**

**folded\_val** [Folded value] A parameter value in its original folded shape.

**validate\_value** [*bool*] Whether to validate the value in addition to the shape. The shape is always validated.

**Returns**

**is\_valid** [*bool*] Whether *folded\_val* is an allowable shape and value.

**err\_msg** [*str*]

## 3.4 Function Wrappers

### 3.4.1 Flattening and folding the input and outputs of a function

**class** `paragami.function_patterns.TransformFunctionInput` (*original\_fun*, *patterns*, *free*, *original\_is\_flat*, *argnums=None*)

Convert a function of folded (or flattened) values into one that takes flattened (or folded) values.

**Examples**

```
mat_pattern = paragami.PSDSymmetricMatrixPattern(3)

def fun(offset, mat, kwoffset=3):
    return np.linalg.slogdet(mat + offset + kwoffset)[1]

flattened_fun = paragami.TransformFunctionInput(
    original_fun=fun, patterns=mat_pattern,
    free=True, argnums=1, original_is_flat=False)

# pd_mat is a matrix:
pd_mat = np.eye(3) + np.full((3, 3), 0.1)

# pd_mat_flat is an unconstrained vector:
pd_mat_flat = mat_pattern.flatten(pd_mat, free=True)

# These two functions return the same value:
print('Original: {}'.format(
    fun(2, pd_mat, kwoffset=3)))
print('Flat: {}'.format(
    flattened_fun(2, pd_mat_flat, kwoffset=3)))
```

**\_\_init\_\_** (*original\_fun*, *patterns*, *free*, *original\_is\_flat*, *argnums=None*)

**Parameters**

**original\_fun:** callable A function that takes one or more values as input.

**patterns:** ‘paragami.Pattern’ or list of ‘paragami.PatternPattern’ A single pattern or array of patterns describing the input to *original\_fun*.

**free:** ‘bool’ or list of ‘bool’ Whether or not the corresponding elements of *patterns* should use free or non-free flattened values.

**original\_is\_flat:** ‘bool’ If *True*, convert *original\_fun* from taking flat arguments to one taking folded arguments. If *False*, convert *original\_fun* from taking folded arguments to one taking flat arguments.

**argnums:** ‘int’ or list of ‘int’ The 0-indexed locations of the corresponding pattern in *patterns* in the order of the arguments fo *original\_fun*.

```
class paragami.function_patterns.FlattenFunctionInput (original_fun, patterns, free,
                                                         argnums=None)
```

A convenience wrapper of *paragami.TransformFunctionInput*.

See also:

*paragami.TransformFunctionInput*

```
__init__ (original_fun, patterns, free, argnums=None)
```

#### Parameters

**original\_fun:** callable A function that takes one or more values as input.

**patterns:** ‘paragami.Pattern’ or list of ‘paragami.PatternPattern’ A single pattern or array of patterns describing the input to *original\_fun*.

**free:** ‘bool’ or list of ‘bool’ Whether or not the corresponding elements of *patterns* should use free or non-free flattened values.

**original\_is\_flat:** ‘bool’ If *True*, convert *original\_fun* from taking flat arguments to one taking folded arguments. If *False*, convert *original\_fun* from taking folded arguments to one taking flat arguments.

**argnums:** ‘int’ or list of ‘int’ The 0-indexed locations of the corresponding pattern in *patterns* in the order of the arguments fo *original\_fun*.

```
class paragami.function_patterns.FoldFunctionInput (original_fun, patterns, free,
                                                         argnums=None)
```

A convenience wrapper of *paragami.TransformFunctionInput*.

See also:

*paragami.TransformFunctionInput*

```
__init__ (original_fun, patterns, free, argnums=None)
```

#### Parameters

**original\_fun:** callable A function that takes one or more values as input.

**patterns:** ‘paragami.Pattern’ or list of ‘paragami.PatternPattern’ A single pattern or array of patterns describing the input to *original\_fun*.

**free:** ‘bool’ or list of ‘bool’ Whether or not the corresponding elements of *patterns* should use free or non-free flattened values.

**original\_is\_flat:** ‘bool’ If *True*, convert *original\_fun* from taking flat arguments to one taking folded arguments. If *False*, convert *original\_fun* from taking folded arguments to one taking flat arguments.

**argnums: ‘int’ or list of ‘int’** The 0-indexed locations of the corresponding pattern in *patterns* in the order of the arguments fo *original\_fun*.

**class** `paragami.function_patterns.FlattenFunctionOutput` (*original\_fun*, *patterns*, *free*, *retnums=None*)

A convenience wrapper of *paragami.TransformFunctionOutput*.

**See also:**

`paragami.TransformFunctionOutput`

`__init__` (*original\_fun*, *patterns*, *free*, *retnums=None*)

#### Parameters

**original\_fun: callable** A function that returns one or more values.

**patterns: ‘paragami.Pattern’ or list of ‘paragami.PatternPattern’** A single pattern or array of patterns describing the return value of *original\_fun*.

**free: ‘bool’ or list of ‘bool’** Whether or not the corresponding elements of *patterns* should use free or non-free flattened values.

**original\_is\_flat: ‘bool’** If *True*, convert *original\_fun* from returning flat values to one returning folded values. If *False*, convert *original\_fun* from returning folded values to one returning flat values.

**retnums: ‘int’ or list of ‘int’** The 0-indexed locations of the corresponding pattern in *patterns* in the order of the return values of *original\_fun*.

**class** `paragami.function_patterns.FoldFunctionOutput` (*original\_fun*, *patterns*, *free*, *retnums=None*)

A convenience wrapper of *paragami.TransformFunctionOutput*.

**See also:**

`paragami.TransformFunctionOutput`

`__init__` (*original\_fun*, *patterns*, *free*, *retnums=None*)

#### Parameters

**original\_fun: callable** A function that returns one or more values.

**patterns: ‘paragami.Pattern’ or list of ‘paragami.PatternPattern’** A single pattern or array of patterns describing the return value of *original\_fun*.

**free: ‘bool’ or list of ‘bool’** Whether or not the corresponding elements of *patterns* should use free or non-free flattened values.

**original\_is\_flat: ‘bool’** If *True*, convert *original\_fun* from returning flat values to one returning folded values. If *False*, convert *original\_fun* from returning folded values to one returning flat values.

**retnums: ‘int’ or list of ‘int’** The 0-indexed locations of the corresponding pattern in *patterns* in the order of the return values of *original\_fun*.

**class** `paragami.function_patterns.FoldFunctionInputAndOutput` (*original\_fun*, *input\_patterns*, *input\_free*, *input\_argnums*, *output\_patterns*, *output\_free*, *output\_retnums=None*)

A convenience wrapper of *paragami.FoldFunctionInput* and *paragami.FoldFunctionOutput*.



See also:

`paragami.FoldFunctionInput`, `paragami.FoldFunctionOutput`

**\_\_init\_\_** (*original\_fun*, *input\_patterns*, *input\_free*, *input\_argnums*, *output\_patterns*, *output\_free*, *output\_retnums*=None)

Initialize self. See `help(type(self))` for accurate signature.

**class** `paragami.function_patterns.FlattenFunctionInputAndOutput` (*original\_fun*,  
*input\_patterns*,  
*input\_free*, *input\_argnums*,  
*output\_patterns*,  
*output\_free*, *output\_retnums*=None)

A convenience wrapper of `paragami.FlattenFunctionInput` and `paragami.FlattenFunctionOutput`.

See also:

`paragami.FlattenFunctionInput`, `paragami.FlattenFunctionOutput`

**\_\_init\_\_** (*original\_fun*, *input\_patterns*, *input\_free*, *input\_argnums*, *output\_patterns*, *output\_free*, *output\_retnums*=None)

Initialize self. See `help(type(self))` for accurate signature.

### 3.4.2 Preconditioning an objective function

**class** `paragami.optimization_lib.PreconditionedFunction` (*original\_fun*, *preconditioner*=None, *preconditioner\_inv*=None)

Get a function whose input has been preconditioned.

Throughout, the subscript `_c` will denote quantities or functions in the preconditioned space. For example, `x` will refer to a variable in the original space and `x_c` to the same variable after preconditioning.

Preconditioning means transforming  $x \rightarrow x_c = A^{-1}x$ , where the matrix  $A$  is the “preconditioner”. If  $f$  operates on  $x$ , then the preconditioned function operates on  $x_c$  and is defined by  $f_c(x_c) := f(Ax_c) = f(x)$ . Gradients of the preconditioned function are defined with respect to its argument in the preconditioned space, e.g.,  $f'_c = \frac{df_c}{dx_c}$ .

A typical value of the preconditioner is an inverse square root of the Hessian of  $f$ , because then the Hessian of  $f_c$  is the identity when the gradient is zero. This can help speed up the convergence of optimization algorithms.

#### Methods

<b>set_preconditioner:</b>	Set the preconditioner to a specified value.
<b>set_preconditioner_with_hessian:</b>	Set the preconditioner based on the Hessian of the objective at a point in the original domain.
<b>get_preconditioner:</b>	Return a copy of the current preconditioner.
<b>get_preconditioner_inv:</b>	Return a copy of the current inverse preconditioner.
<b>precondition:</b>	Convert from the original domain to the preconditioned domain.
<b>unprecondition:</b>	Convert from the preconditioned domain to the original domain.

**\_\_init\_\_** (*original\_fun*, *preconditioner*=None, *preconditioner\_inv*=None)

#### Parameters

**original\_fun:** callable function of a single argument

**preconditioner:** The initial preconditioner.

**preconditioner\_inv:** The inverse of the initial preconditioner.

**precondition** ( $x$ )

Multiply by the inverse of the preconditioner to convert  $x$  in the original domain to  $x_c$  in the preconditioned domain.

This function is provided for convenience, but it is more numerically stable to use `np.linalg.solve(preconditioner, x)`.

**set\_preconditioner\_with\_hessian** ( $x=None$ ,  $hessian=None$ ,  $ev\_min=None$ ,  $ev\_max=None$ )

Set the preconditioner to the inverse square root of the Hessian of the original objective (or an approximation thereof).

#### Parameters

**x: Numeric vector** The point at which to evaluate the Hessian of `original_fun`. If  $x$  is specified, the Hessian is evaluated with automatic differentiation. Specify either  $x$  or `hessian` but not both.

**hessian: Numeric matrix** The hessian of `original_fun` or an approximation of it. Specify either  $x$  or `hessian` but not both.

**ev\_min: float** If not `None`, set eigenvalue of `hessian` that are less than `ev_min` to `ev_min` before taking the square root.

**ev\_maxs: float** If not `None`, set eigenvalue of `hessian` that are greater than `ev_max` to `ev_max` before taking the square root.

#### Returns

Sets the preconditioner for the class and returns the Hessian with the eigenvalues thresholded by “`ev_min`” and “`ev_max`”.

**unprecondition** ( $x_c$ )

Multiply by the preconditioner to convert  $x_c$  in the preconditioned domain to  $x$  in the original domain.

### 3.4.3 An optimization objective class

**class** `paragami.optimization_lib.OptimizationObjective` (*objective\_fun*, *print\_every=1*,  
*log\_every=0*)

Derivatives and logging for an optimization objective function.

#### Attributes

**optimization\_log: Dictionary** A record of the optimization progress as recorded by `log_value`.

## Methods

<b>f:</b>	The objective function with logging.
<b>grad:</b>	The gradient of the objective function.
<b>hessian:</b>	The Hessian of the objective function.
<b>hessian_vector_product:</b>	The Hessian vector product of the objective function.
<b>set_print_every:</b>	Set how often to display optimization progress.
<b>set_log_every:</b>	Set how often to log optimization progress.
<b>reset_iteration_count:</b>	Reset the number of iterations for the purpose of printing and logging.
<b>reset_log:</b>	Clear the log.
<b>reset:</b>	Run <code>reset_iteration_count</code> and <code>reset_log</code> .
<b>print_value:</b>	Display a function evaluation.
<b>log_value:</b>	Log a function evaluation.

`__init__(objective_fun, print_every=1, log_every=0)`

### Parameters

**obj\_fun:** Callable function of one argumnet The function to be minimized.

**print\_every:** integer Print the optimization value every `print_every` iterations.

**log\_every:** integer Log the optimization value every `log_every` iterations.

`log_value(num_f_evals, x, f_val)`

Log the optimization progress. To create a custom log, overload this function. By default, the log is a list of tuples (`iteration`, `x`, `f(x)`).

### Parameters

**num\_f\_vals:** Integer The total number of function evaluations.

**x:** The current argument to the objective function.

**f\_val:** The value of the objective at `x`.

`num_iterations()`

Return the number of times the optimization function has been called, not counting any derivative evaluations.

`print_value(num_f_evals, x, f_val)`

Display the optimization progress. To display a custom update, overload this function.

### Parameters

**num\_f\_vals:** Integer The total number of function evaluations.

**x:** The current argument to the objective function.

**f\_val:** The value of the objective at `x`.

`reset()`

Reset the itreation count and clear the log.

`set_log_every(n)`

### Parameters

**n:** integer Log the objective function value every `n` iterations. If 0, do not log.

`set_print_every(n)`

### Parameters

**n: integer** Print the objective function value every *n* iterations. If 0, do not print any output.

## 3.5 Sensitivity Functions

**Warning:** These functions are deprecated. Please use the `vittles` package instead.

## 3.6 Writing and reading patterns and values from disk

### 3.6.1 Saving Folded Values with Patterns

Flattning makes it easy to save and load structured data to and from disk.

```
pattern = paragami.PatternDict()
pattern['num'] = paragami.NumericArrayPattern((1, 2))
pattern['mat'] = paragami.PSDSymmetricMatrixPattern(5)

val_folded = pattern.random()
extra = np.random.random(5)

outfile = tempfile.NamedTemporaryFile()
outfile_name = outfile.name
outfile.close()

paragami.save_folded(outfile_name, val_folded, pattern, extra=extra)

val_folded_loaded, pattern_loaded, data = \
    paragami.load_folded(outfile_name + '.npz')

# The loaded values match the saved values.
assert pattern == pattern_loaded
assert np.all(val_folded['num'] == val_folded_loaded['num'])
assert np.all(val_folded['mat'] == val_folded_loaded['mat'])
assert np.all(data['extra'] == extra)
```

`paragami.pattern_containers.save_folded(file, folded_val, pattern, **argk)`

Save a folded value to a file with its pattern.

Flatten a folded value and save it with its pattern to a file using `numpy.savez`. Additional keyword arguments will also be saved to the file.

#### Parameters

**file: String or file** Follows the conventions of `numpy.savez`. Note that the `npz` extension will be added if it is not present.

**folded\_val:** The folded value of a parameter.

**pattern:** A paragami pattern for the folded value.

`paragami.pattern_containers.load_folded(file)`

Load a folded value and its pattern from a file together with any additional data.

Note that `pattern` must be registered with `register_pattern_json` to use `load_folded`.

#### Parameters

**file: String or file** A file or filename of data saved with `save_folded`.

#### Returns

**folded\_val:** The folded value of the saved parameter.

**pattern:** The paragami pattern of the saved parameter.

**data:** The data as returned from `np.load`. Additional saved values will exist as keys of data.

## 3.6.2 Saving patterns

You can convert a particular pattern class to and from JSON using the `to_json` and `from_json` methods.

```
>>> pattern = paragami.NumericArrayPattern(shape=(2, 3))
>>>
>>> # ``pattern_json_string`` is a JSON string that can be written to a file.
>>> pattern_json_string = pattern.to_json()
>>>
>>> # ``same_pattern`` is identical to ``pattern``.
>>> same_pattern = paragami.NumericArrayPattern.from_json(pattern_json_string)
```

However, in order to use `from_json`, you need to know which pattern the JSON string was generated from. In order to decode generic JSON strings, one can use `get_pattern_from_json`.

```
>>> pattern_json_string = pattern.to_json()
>>> # ``same_pattern`` is identical to ``pattern``.
>>> same_pattern = paragami.get_pattern_from_json(pattern_json_string)
```

Before a pattern can be used with `get_pattern_from_json`, it needs to be registered with `register_pattern_json`. All the patterns in `paragami` are automatically registered, but if you define your own patterns they will have to be registered before they can be used with `get_pattern_from_json`.

`paragami.pattern_containers.get_pattern_from_json(pattern_json)`

Return the appropriate pattern from `pattern_json`.

The pattern must have been registered using `register_pattern_json`.

#### Parameters

**pattern\_json: String** A JSON string as created with a pattern's `to_json` method.

#### Returns

The pattern instance encoded in the “`pattern_json`” string.

`paragami.pattern_containers.register_pattern_json(pattern, allow_overwrite=False)`

Register a pattern for automatic conversion from JSON.

#### Parameters

**pattern: A Pattern class** The pattern to register.

**allow\_overwrite: Boolean** If true, allow overwriting already-registered patterns.

## Examples

```
>>> class MyCustomPattern(paragami.Pattern):
>>>     ... definitions ...
>>>
>>> paragami.register_pattern_json(paragami.MyCustomPattern)
>>>
>>> my_pattern = MyCustomPattern(...)
>>> my_pattern_json = my_pattern.to_json()
>>>
>>> # ``my_pattern_from_json`` should be identical to ``my_pattern``.
>>> my_pattern_from_json = paragami.get_pattern_from_json(my_pattern_json)
```

## Symbols

- `__init__()` (paragami.base\_patterns.Pattern method), 19
- `__init__()` (paragami.function\_patterns.FlattenFunctionInput method), 35
- `__init__()` (paragami.function\_patterns.FlattenFunctionInputAndOutput method), 37
- `__init__()` (paragami.function\_patterns.FlattenFunctionOutput method), 36
- `__init__()` (paragami.function\_patterns.FoldFunctionInput method), 35
- `__init__()` (paragami.function\_patterns.FoldFunctionInputAndOutput method), 37
- `__init__()` (paragami.function\_patterns.FoldFunctionOutput method), 36
- `__init__()` (paragami.function\_patterns.TransformFunctionInput method), 34
- `__init__()` (paragami.numeric\_array\_patterns.NumericArrayPattern method), 22
- `__init__()` (paragami.optimization\_lib.OptimizationObjective method), 39
- `__init__()` (paragami.optimization\_lib.PreconditionedFunction method), 37
- `__init__()` (paragami.pattern\_containers.PatternArray method), 31
- `__init__()` (paragami.pattern\_containers.PatternDict method), 29
- `__init__()` (paragami.psdmatrix\_patterns.PSDSymmetricMatrixPattern method), 24
- `__init__()` (paragami.simplex\_patterns.SimplexArrayPattern method), 26
- A**
- `array_shape()` (paragami.pattern\_containers.PatternArray method), 31
- `as_dict()` (paragami.base\_patterns.Pattern method), 19
- `as_dict()` (paragami.numeric\_array\_patterns.NumericArrayPattern method), 22
- `as_dict()` (paragami.pattern\_containers.PatternArray method), 31
- `as_dict()` (paragami.pattern\_containers.PatternDict method), 29
- `as_dict()` (paragami.psdmatrix\_patterns.PSDSymmetricMatrixPattern method), 24
- `as_dict()` (paragami.simplex\_patterns.SimplexArrayPattern method), 27
- B**
- `diag_lb()` (paragami.psdmatrix\_patterns.PSDSymmetricMatrixPattern method), 24
- E**
- `empty()` (paragami.base\_patterns.Pattern method), 19
- `empty()` (paragami.numeric\_array\_patterns.NumericArrayPattern method), 23
- `empty()` (paragami.pattern\_containers.PatternArray method), 32
- `empty()` (paragami.pattern\_containers.PatternDict method), 29
- `empty()` (paragami.psdmatrix\_patterns.PSDSymmetricMatrixPattern method), 24
- `empty()` (paragami.simplex\_patterns.SimplexArrayPattern method), 27
- `empty_bool()` (paragami.base\_patterns.Pattern method), 19
- F**
- `flat_indices()` (paragami.base\_patterns.Pattern method), 20
- `flat_indices()` (paragami.numeric\_array\_patterns.NumericArrayPattern method), 23
- `flat_indices()` (paragami.pattern\_containers.PatternArray method), 32
- `flat_indices()` (paragami.pattern\_containers.PatternDict method), 29
- `flat_indices()` (paragami.psdmatrix\_patterns.PSDSymmetricMatrixPattern method), 25
- `flat_indices()` (paragami.simplex\_patterns.SimplexArrayPattern method), 27

[flat\\_length\(\) \(paragami.base\\_patterns.Pattern method\), 20](#)  
[flat\\_length\(\) \(paragami.numeric\\_array\\_patterns.NumericArrayPattern method\), 23](#)  
[flat\\_length\(\) \(paragami.pattern\\_containers.PatternArray method\), 32](#)  
[flatten\(\) \(paragami.base\\_patterns.Pattern method\), 20](#)  
[flatten\(\) \(paragami.numeric\\_array\\_patterns.NumericArrayPattern method\), 23](#)  
[flatten\(\) \(paragami.pattern\\_containers.PatternArray method\), 32](#)  
[flatten\(\) \(paragami.pattern\\_containers.PatternDict method\), 29](#)  
[flatten\(\) \(paragami.psdmatrix\\_patterns.PSDSymmetricMatrixPattern method\), 25](#)  
[flatten\(\) \(paragami.simplex\\_patterns.SimplexArrayPattern method\), 27](#)  
[FlattenFunctionInput \(class in paragami.function\\_patterns\), 35](#)  
[FlattenFunctionInputAndOutput \(class in paragami.function\\_patterns\), 37](#)  
[FlattenFunctionOutput \(class in paragami.function\\_patterns\), 36](#)  
[fold\(\) \(paragami.base\\_patterns.Pattern method\), 20](#)  
[fold\(\) \(paragami.numeric\\_array\\_patterns.NumericArrayPattern method\), 23](#)  
[fold\(\) \(paragami.pattern\\_containers.PatternArray method\), 32](#)  
[fold\(\) \(paragami.pattern\\_containers.PatternDict method\), 30](#)  
[fold\(\) \(paragami.psdmatrix\\_patterns.PSDSymmetricMatrixPattern method\), 25](#)  
[fold\(\) \(paragami.simplex\\_patterns.SimplexArrayPattern method\), 27](#)  
[FoldFunctionInput \(class in paragami.function\\_patterns\), 35](#)  
[FoldFunctionInputAndOutput \(class in paragami.function\\_patterns\), 36](#)  
[FoldFunctionOutput \(class in paragami.function\\_patterns\), 36](#)  
[freeing\\_jacobian\(\) \(paragami.base\\_patterns.Pattern method\), 21](#)  
[freeing\\_jacobian\(\) \(paragami.pattern\\_containers.PatternArray method\), 33](#)  
[freeing\\_jacobian\(\) \(paragami.pattern\\_containers.PatternDict method\), 30](#)  
[from\\_json\(\) \(paragami.base\\_patterns.Pattern class method\), 21](#)  
[from\\_json\(\) \(paragami.pattern\\_containers.PatternArray class method\), 33](#)  
[from\\_json\(\) \(paragami.pattern\\_containers.PatternDict class method\), 30](#)  
[from\\_json\(\) \(paragami.simplex\\_patterns.SimplexArrayPattern class method\), 28](#)

**G**

[get\\_pattern\\_from\\_json\(\) \(in module paragami.pattern\\_containers\), 41](#)

**L**

[load\\_folded\(\) \(in module paragami.pattern\\_containers\), 40](#)  
[log\\_value\(\) \(paragami.optimization\\_lib.OptimizationObjective method\), 39](#)

**N**

[num\\_iterations\(\) \(paragami.optimization\\_lib.OptimizationObjective method\), 39](#)  
[NumericArrayPattern \(class in paragami.numeric\\_array\\_patterns\), 22](#)

**O**

[OptimizationObjective \(class in paragami.optimization\\_lib\), 38](#)

**P**

[Pattern \(class in paragami.base\\_patterns\), 19](#)  
[PatternArray \(class in paragami.pattern\\_containers\), 31](#)  
[PatternDict \(class in paragami.pattern\\_containers\), 28](#)  
[precondition\(\) \(paragami.optimization\\_lib.PreconditionedFunction method\), 38](#)  
[PreconditionedFunction \(class in paragami.optimization\\_lib\), 37](#)  
[pattern\\_value\(\) \(paragami.optimization\\_lib.OptimizationObjective method\), 39](#)  
[PSDSymmetricMatrixPattern \(class in paragami.psdmatrix\\_patterns\), 24](#)

**R**

[random\(\) \(paragami.base\\_patterns.Pattern method\), 21](#)  
[register\\_pattern\\_json\(\) \(in module paragami.pattern\\_containers\), 41](#)  
[reset\(\) \(paragami.optimization\\_lib.OptimizationObjective method\), 39](#)

**S**

[save\\_folded\(\) \(in module paragami.pattern\\_containers\), 40](#)  
[set\\_log\\_every\(\) \(paragami.optimization\\_lib.OptimizationObjective method\), 39](#)  
[set\\_preconditioner\\_with\\_hessian\(\) \(paragami.optimization\\_lib.PreconditionedFunction method\), 38](#)  
[set\\_print\\_every\(\) \(paragami.optimization\\_lib.OptimizationObjective method\), 39](#)  
[shape\(\) \(paragami.pattern\\_containers.PatternArray method\), 33](#)



shape() (paragami.psdmatrix\_patterns.PSDSymmetricMatrixPattern  
method), 25

SimplexArrayPattern (class in  
paragami.simplex\_patterns), 26

size() (paragami.psdmatrix\_patterns.PSDSymmetricMatrixPattern  
method), 25

## T

to\_json() (paragami.base\_patterns.Pattern method), 21

TransformFunctionInput (class in  
paragami.function\_patterns), 34

## U

unfreeing\_jacobian() (paragami.base\_patterns.Pattern  
method), 21

unfreeing\_jacobian() (paragami.pattern\_containers.PatternArray  
method), 33

unfreeing\_jacobian() (paragami.pattern\_containers.PatternDict  
method), 30

unprecondition() (paragami.optimization\_lib.PreconditionedFunction  
method), 38

## V

validate\_folded() (paragami.base\_patterns.Pattern  
method), 22

validate\_folded() (paragami.numeric\_array\_patterns.NumericArrayPattern  
method), 24

validate\_folded() (paragami.pattern\_containers.PatternArray  
method), 34

validate\_folded() (paragami.pattern\_containers.PatternDict  
method), 31

validate\_folded() (paragami.psdmatrix\_patterns.PSDSymmetricMatrixPattern  
method), 25

validate\_folded() (paragami.simplex\_patterns.SimplexArrayPattern  
method), 28