# paragami Documentation

***Release 0.33.post16+gbc72261***

## Contributors

**Mar 03, 2019**

# Contents

Parameter folding and flattening, parameter origami[1]: `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

[1] 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.

# CHAPTER 2

# Examples

## 2.1 Flattening and Folding With Covariance Matrices.

[1]:
```
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`.)

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

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:
[[1.2712968  0.74536048 0.33203184]
 [0.74536048 1.91869072 0.4602062 ]
 [0.33203184 0.4602062  1.58338   ]]

a_flat:
[1.2712968  0.74536048 0.33203184 0.74536048 1.91869072 0.4602062
 0.33203184 0.4602062  1.58338   ]
```

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:
[[1.2712968  0.74536048 0.33203184]
 [0.74536048 1.91869072 0.4602062 ]
 [0.33203184 0.4602062  1.58338   ]]

a_fold:
[[1.2712968  0.74536048 0.33203184]
 [0.74536048 1.91869072 0.4602062 ]
 [0.33203184 0.4602062  1.58338   ]]
```

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.

[4]:
```
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.

[5]:
```
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`.

[6]:
```
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.

[7]:
```
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 is 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.

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

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`.

[8]:
```
a_freeflat = a_pattern.flatten(a, free=True)
print('a:\n{}\n'.format(a))
print('a_freeflat:\n{}\n'.format(a_freeflat))
```

```
a:
[[1.2712968  0.74536048 0.33203184]
 [0.74536048 1.91869072 0.4602062 ]
 [0.33203184 0.4602062  1.58338   ]]

a_freeflat:
[0.12001874 0.66106306 0.19659043 0.2944803  0.21814513 0.18546238]
```

However, it transforms correctly back to a when folded.

[9]:
```
a_freefold = a_pattern.fold(a_freeflat, free=True)
print('a:\n{}\n'.format(a))
print('a_fold:\n{}\n'.format(a_freefold))
```

```
a:
[[1.2712968  0.74536048 0.33203184]
 [0.74536048 1.91869072 0.4602062 ]
 [0.33203184 0.4602062  1.58338   ]]

a_fold:
[[1.2712968  0.74536048 0.33203184]
 [0.74536048 1.91869072 0.4602062 ]
 [0.33203184 0.4602062  1.58338   ]]
```

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.

[10]:
```
# 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)
```

## 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 $X = (x_1, ..., x_N)$, where $x_n \in \mathbb{R}^3$, and write a loss function as

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

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 funciton 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 flatting the function get_loss, using
# the pattern a_pattern, with free parameterization, and the paramater
# 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 evalated 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 evalated 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

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

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
```

(continues on next page)

(continued from previous page)

```
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 $\mathrm{Cov}(A_{flat})$. The covariance between two elements of $A_{flat}$ corresponds to that between two elements of $A$. For example, using the notation given above,

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

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

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

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

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 $\frac{dA_{free}}{dA^T_{freeflat}}$ 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 erronously 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)

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 $\mathrm{Cov}(A_{flat})$ 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))
```

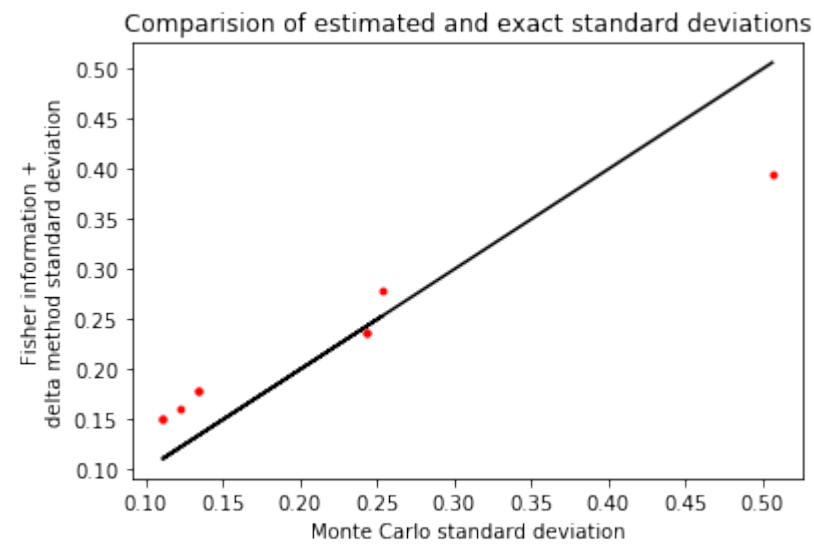
```
The marginal standard deviations of the elements of A:
[[0.15991362 0.15046908 0.17852051]
 [0.15046908 0.27980802 0.23681303]
 [0.17852051 0.23681303 0.39432762]]
```

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
```

(continues on next page)

(continued from previous page)

```
 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 ' +
      'so that you do not accidentally mix free and non-free flat values.')
dict_pattern_nodefault = paragami.PatternDict()
try:
    dict_pattern_nodefault.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.
```

## 2.4 Parameter containers.

Many models depend on several parameters of different types. In this notebook, we demonstrate flattening and folding "containers", which are patterns containing other patterns.

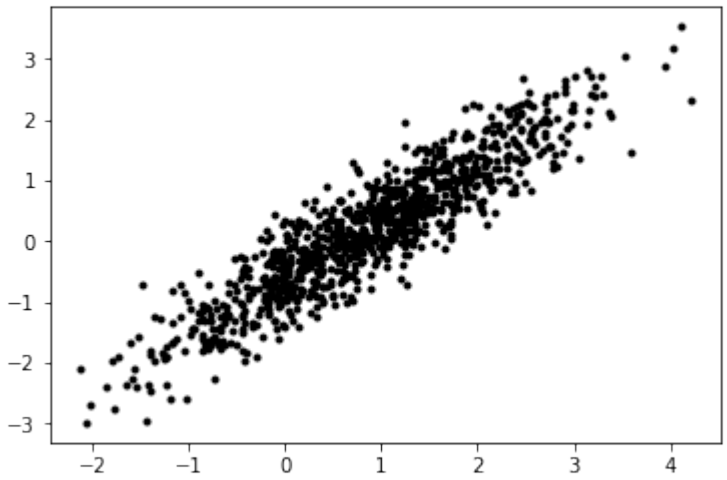
### 2.4.1 Parameter dictionaries.

[1]:
```
import autograd
import autograd.numpy as np
import example_utils
import matplotlib.pyplot as plt
import paragami
import time

%matplotlib inline
```

Let's consider a multivariate normal example. Define some parameters and draw some data.

[2]:
```
dim = 2
num_obs = 1000

mean_true = np.random.random(dim)
cov_true = np.random.random((dim, dim))
cov_true = 0.1 * np.eye(dim) + np.full((dim, dim), 1.0)

x = np.random.multivariate_normal(mean=mean_true, cov=cov_true, size=(num_obs, ))
plt.plot(x[:, 0], x[:, 1], 'k.')
```

[2]:
```
[<matplotlib.lines.Line2D at 0x7f6a3bd86dd8>]
```




A multivariate normal distribution depends on two parameters – a mean and a positive definite covariance matrix. In Python, we might store these parameters in a dictionary with a member for the mean and a member for the covariance.

This can be represented with a pattern dictionary, i.e., a `paragami.PatternDict` pattern. Each member of a pattern dictionary must itself be a pattern.

[3]:
```
mvn_pattern = paragami.PatternDict(free_default=True)
mvn_pattern['mean'] = paragami.NumericVectorPattern(length=dim)
mvn_pattern['cov'] = paragami.PSDSymmetricMatrixPattern(size=dim)
```

Flattening and folding work with pattern dictionaries just as with ordinary patterns. Note that folded pattern dicitonaries are `OrderedDict` by default.

[36]:
```
true_mvn_par = dict()
true_mvn_par['mean'] = mean_true
true_mvn_par['cov'] = cov_true

print('\nA dictionary of MVN parameters:\n{}'.format(
      true_mvn_par))

mvn_par_free = mvn_pattern.flatten(true_mvn_par)
print('\nA flat representation:\n{}'.format(
    mvn_pattern.flatten(true_mvn_par)))

print('\nFolding recovers the original parameters:\n{}'.format(
    mvn_pattern.fold(mvn_par_free)))
```

```
A dictionary of MVN parameters:
{'cov': array([[1.1, 1. ],
       [1. , 1.1]]), 'mean': array([0.87367236, 0.21280422])}

A flat representation:
[ 0.87367236  0.21280422  0.04765509  0.95346259 -0.82797896]

Folding recovers the original parameters:
OrderedDict([('mean', array([0.87367236, 0.21280422])), ('cov', array([[1.1, 1. ],
       [1. , 1.1]]))])
```

Parameter dictionaries are particularly convenient for optimization problems involving multiple parameters. A good

working style is to implement the loss function using named arguments and then wrap it using a lambda function.

To illustrate this, let us use `get_normal_log_prob` from `example_utils`.

[6]:
```
# ``example_utils.get_normal_log_prob`` returns the log probability of
# each datapoint x up to a constant.
def get_loss(x, sigma, mu):
    return -1 * np.sum(
        example_utils.get_normal_log_prob(x, sigma, mu))

get_free_loss = paragami.FlattenFunctionInput(
    lambda mvn_par: get_loss(x=x, sigma=mvn_par['cov'], mu=mvn_par['mean']),
    patterns=mvn_pattern,
    free=True)

print('Free loss:\t{}'.format(get_free_loss(mvn_par_free)))
print('Original loss:\t{}'.format(get_loss(x, true_mvn_par['cov'], true_mvn_par['mean
↪']))))
```

```
Free loss:      219.8143711666299
Original loss:  219.8143711666299
```

As with other parameters, `autograd` works with parameter dictionaries.

[7]:
```
get_free_loss_grad = autograd.grad(get_free_loss)
print(get_free_loss_grad(mvn_par_free))
```

```
[-59.66882221  78.5304     -24.49767189 -12.36127117  36.00719091]
```

### Pattern dictionaries containing pattern dictionaries.

Pattern dictionaries can contain pattern dictionaries (and so on).

[34]:
```
mvns_pattern = paragami.PatternDict(free_default=True)
mvns_pattern['mvn1'] = mvn_pattern
mvns_pattern['mvn2'] = mvn_pattern
mvns_pattern['ez'] = paragami.SimplexArrayPattern(array_shape=(1, ), simplex_size=2)

mvns_par = dict()
mvns_par['mvn1'] = true_mvn_par
mvns_par['mvn2'] = mvn_pattern.random()
mvns_par['ez'] = np.array([[0.3, 0.7]])

print('Folded mvns_par:\n{}'.format(mvns_par))
print('\nFree mvns_par:\n{}'.format(mvns_pattern.flatten(mvns_par)))
```

```
Folded mvns_par:
{'mvn1': {'cov': array([[1.1, 1. ],
       [1. , 1.1]]), 'mean': array([0.87367236, 0.21280422])}, 'mvn2':␣
↪OrderedDict([('mean', array([0.9666736 , 0.24848757])), ('cov', array([[4.77849291,␣
↪1.56784344],
       [1.56784344, 1.84355085]]))]), 'ez': array([[0.3, 0.7]])}

Free mvns_par:
[ 0.87367236  0.21280422  0.04765509  0.95346259 -0.82797896  0.9666736
  0.24848757  0.7820626   0.71722798  0.14226413  0.84729786]
```

Members of a pattern dictionary are just patterns, and can be used directly.

[37]:
```
mvn_par_free = mvns_pattern['mvn1'].flatten(true_mvn_par)
print('\nA flat representation of true_mvn_par:\n{}'.format(
    mvn_pattern.flatten(true_mvn_par)))
```

```
A flat representation of true_mvn_par:
[ 0.87367236  0.21280422  0.04765509  0.95346259 -0.82797896]
```

### Locking parameter dictionaries.

The meaning of a parameter dictionary changes as you add or assign elements.

[40]:
```
example_pattern = paragami.PatternDict(free_default=True)

example_pattern['par1'] = paragami.NumericVectorPattern(length=2)
print('The flat length of example_pattern with par1:\t\t\t{}'.format(
    example_pattern.flat_length()))

example_pattern['par2'] = paragami.NumericVectorPattern(length=3)
print('The flat length of example_pattern with par1 and par2:\t\t{}'.format(
    example_pattern.flat_length()))

example_pattern['par1'] = paragami.NumericVectorPattern(length=10)
print('The flat length of example_pattern with new par1 and par2:\t{}'.format(
    example_pattern.flat_length()))
```

```
The flat length of example_pattern with par1:                   2
The flat length of example_pattern with par1 and par2:          5
The flat length of example_pattern with new par1 and par2:      13
```

Sometime, you want to make sure the meaning of a parameter dictionary stays fixed. In order to prevent a pattern dictionary from having more elements added, you can `lock()` it.

[43]:
```
example_pattern.lock()
try:
    example_pattern['par3'] = paragami.NumericVectorPattern(length=4)
except ValueError as err:
    print('Adding a new pattern failed with the following error:\n{}'.format(err))

try:
    example_pattern['par1'] = paragami.NumericVectorPattern(length=4)
except ValueError as err:
    print('\nChanging an existing pattern failed with the following error:\n{}'.
↪format(err))
```

```
Adding a new pattern failed with the following error:
The dictionary is locked, and its values cannot be changed.

Changing an existing pattern failed with the following error:
The dictionary is locked, and its values cannot be changed.
```

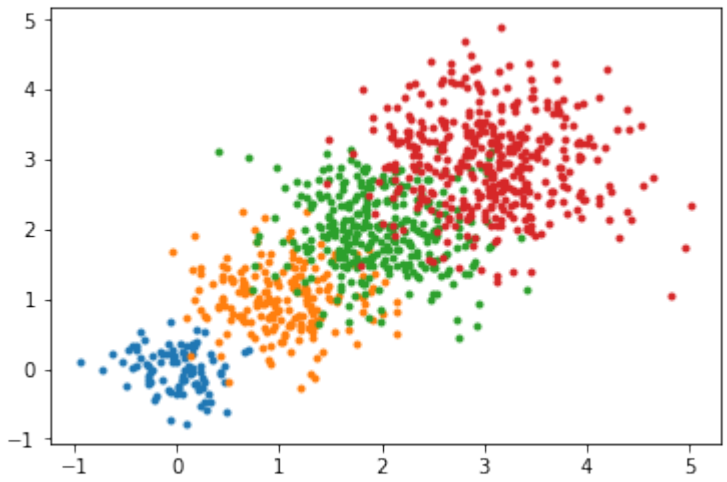
### 2.4.2 Parameter arrays.

Sometimes it is useful to have arrays of patterns. A classic use case is mixture distributions. Suppose that, for some $K$, and probabilities $\pi_k$ for $k = 1$ to $K$,

$$P(y_n|\pi, \mu_1, ..., \mu_K, \Sigma_1, ..., \Sigma_K) = \sum_{k=1}^{K} \pi_k \mathcal{N}(y_n|\mu_k, \Sigma_k) . \tag{2.1}$$

Then the random variable $y_n$ is distributed according to a mixture of normals. Let us define parameters and draw some data.

[64]:
```
num_components = 4
prob_true = np.arange(1, num_components + 1)
prob_true = prob_true / np.sum(prob_true)

k_true = np.random.choice(range(num_components), p=prob_true, size=num_obs)

means_true = np.array([
    np.full(dim, float(k)) for k in range(num_components) ])

covs_true = np.array([
    0.1 * (k + 1) * np.eye(dim) for k in range(num_components) ])

y = np.array([
    np.random.multivariate_normal(
        means_true[k_true[n], :],
        covs_true[k_true[n], :, :]) for n in range(num_obs) ])

for k in range(num_components):
    k_rows = k_true == k
    plt.plot(y[k_rows, 0], y[k_rows, 1], '.')
```




To define parameters for the mixture, we have defined arrays of shape `(num_components, )` containing the means and covariances. The means can be represented with an orderinary `NumericArrayPattern`. But to represent the array of covariances with `paragami`, we can use pattern arrays, i.e, `paragami.PatternArray`.

[79]:
```
mix_pattern = paragami.PatternDict(free_default=True)
mix_pattern['prob'] = paragami.SimplexArrayPattern(
    array_shape=(1, ), simplex_size=num_components)
```

(continues on next page)

(continued from previous page)

```
mix_pattern['means'] = paragami.NumericArrayPattern(shape=(num_components, dim))
mix_pattern['covs'] = paragami.PatternArray(
    array_shape=(num_components, ),
    base_pattern=paragami.PSDSymmetricMatrixPattern(size=dim))

true_mix_par = dict()
true_mix_par['means'] = means_true
true_mix_par['covs'] = covs_true
true_mix_par['prob'] = np.expand_dims(prob_true, axis=0)

true_mix_free = mix_pattern.flatten(true_mix_par)
print('Folding recovers the true mixture parameters:\n{}'.format(
    mix_pattern.fold(true_mix_free)))
```

```
Folding recovers the true mixture parameters:
OrderedDict([('prob', array([[0.1, 0.2, 0.3, 0.4]])), ('means', array([[0., 0.],
       [1., 1.],
       [2., 2.],
       [3., 3.]])), ('covs', array([[[0.1, 0. ],
        [0. , 0.1]],

       [[0.2, 0. ],
        [0. , 0.2]],

       [[0.3, 0. ],
        [0. , 0.3]],

       [[0.4, 0. ],
        [0. , 0.4]]]))])
```

Pattern arrays have some limitations. For one, they can only contain numeric types. This means you cannot have arrays of pattern dictionaries.

[76]:
```
example_pattern = paragami.PatternDict()
example_pattern['a'] = paragami.NumericVectorPattern(length=2)

try:
    paragami.PatternArray(array_shape=(2, ), base_pattern=example_pattern)
except NotImplementedError as err:
    print('Attempting to create a PatternArray of PatternDicts failed with the␣
↪error:\n{}'.format(
          err))
```

```
Attempting to create a PatternArray of PatternDicts failed with the error:
PatternArray does not support patterns whose folded values are not numpy.ndarray␣
↪types.
```

Also, under the hood, `PatternArray` types fold and flatten using a for loop over the array elements. For this reason, it is more effecient to use a different type if possible.

In particular, `SimplexArray` patterns will be more efficient than a `PatternArray` of `SimplexArray`, as the following example shows.

[108]:
```
import time

array_shape = (50, 10)
test_array = paragami.PatternArray(
    array_shape=array_shape,
```

(continues on next page)

(continued from previous page)

```
    base_pattern=paragami.SimplexArrayPattern(array_shape=(1, ), simplex_size=5))

test_simplex = paragami.SimplexArrayPattern(array_shape=array_shape, simplex_size=5)

simplex_val = test_simplex.random()
simplex_array_val = np.expand_dims(simplex_val, axis=2)

test_times = 10

simplex_time = time.time()
for i in range(test_times):
    test_simplex.flatten(simplex_val, free=True)
simplex_time = time.time() - simplex_time

array_time = time.time()
for i in range(test_times):
    test_array.flatten(simplex_array_val, free=True)
array_time = time.time() - array_time

print('Array time:\t{}\nSimplex time:\t{}'.format(array_time, simplex_time))
```

```
Array time:     0.30515265464782715
Simplex time:   0.0015544891357421875
```

## 2.5 Optimization Objectives.

When using `paragami`, a typical workflow is as follows:

1. Define an objective function.
2. Define parameter patterns for the input to the objective function.
3. Flatten the input of the objective function.
4. Use `autograd` to define derivatives of the flat objective.
5. Optimize.

To help reduce the boilerplate in step 4, `paragami` provides an `OptimizationObjective` class. In addition to defining all the `autograd` derivatives, optimization objectives also provide basic logging and progress displays.

[1]:
```
import autograd
import autograd.numpy as np
import scipy as sp
import example_utils
import matplotlib.pyplot as plt
import paragami
import time

%matplotlib inline
```

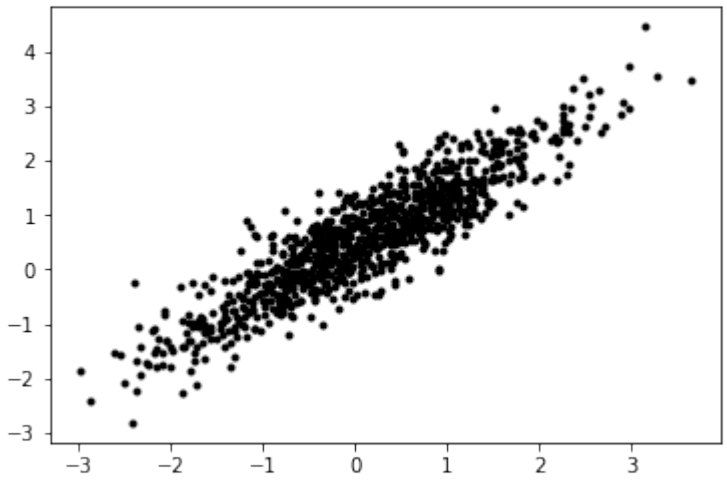
### 2.5.1 Problem setup.

Let us consider again the multivariate normal objective. The functionality in this section is described in previous notebooks.

First, define some parameters and draw data.

[2]:
```
dim = 2
num_obs = 1000

mean_true = np.random.random(dim)
cov_true = np.random.random((dim, dim))
cov_true = 0.1 * np.eye(dim) + np.full((dim, dim), 1.0)

x = np.random.multivariate_normal(mean=mean_true, cov=cov_true, size=(num_obs, ))
plt.plot(x[:, 0], x[:, 1], 'k.')
```

[2]:
```
[<matplotlib.lines.Line2D at 0x7f030ffdeda0>]
```




Define `paragami` patterns and a loss function.

[3]:
```
mvn_pattern = paragami.PatternDict(free_default=True)
mvn_pattern['mean'] = paragami.NumericVectorPattern(length=dim)
mvn_pattern['cov'] = paragami.PSDSymmetricMatrixPattern(size=dim)

# ``example_utils.get_normal_log_prob`` returns the log probability of
# each datapoint x up to a constant.
def get_loss(x, sigma, mu):
    return -1 * np.sum(
        example_utils.get_normal_log_prob(x, sigma, mu))

get_free_loss = paragami.FlattenFunctionInput(
    lambda mvn_par: get_loss(x=x, sigma=mvn_par['cov'], mu=mvn_par['mean']),
    patterns=mvn_pattern,
    free=True)

mvn_par = mvn_pattern.random()
mvn_free = mvn_pattern.flatten(mvn_par)
```

### Optimization objectives.

We can use `get_free_loss` to define an optimization objective and pass it to `scipy.optimize.minimize`.

[4]:
```
objective = paragami.OptimizationObjective(get_free_loss)

def Optimize(objective, x0=np.zeros(mvn_pattern.flat_length())):
    opt_result = sp.optimize.minimize(
        x0=x0,
        fun=objective.f,
        jac=objective.grad,
        hessp=objective.hessian_vector_product,
        method='trust-ncg')
    return opt_result
```

### 2.5.2 Controlling printing.

The frequency of printing can be controlled with `set_print_every()`, and the count can be reset with `reset()`.

[5]:
```
print('\nOptimizing:')
opt_result = Optimize(objective)
print(opt_result.message)

print('\nOptimizing printing less:')
objective.reset()
objective.set_print_every(5)
opt_result = Optimize(objective)
print(opt_result.message)

print('\nOptimizing without reset:')
opt_result = Optimize(objective)
print(opt_result.message)

print('\nOptimizing without printing:')
objective.set_print_every(0)
opt_result = Optimize(objective)
print(opt_result.message)
```

```
Optimizing:
Iter 0: f = 1290.22412134
Iter 1: f = 539.49481319
Iter 2: f = 449.15367362
Iter 3: f = 272.45823141
Iter 4: f = 255.05895262
Iter 5: f = 254.63015576
Iter 6: f = 254.47898697
Iter 7: f = 254.47819241
Iter 8: f = 254.47818845
Iter 9: f = 254.47818845
Optimization terminated successfully.

Optimizing printing less:
Iter 0: f = 1290.22412134
Iter 5: f = 254.63015576
Optimization terminated successfully.

Optimizing without reset:
Iter 10: f = 1290.22412134
```

(continues on next page)

(continued from previous page)

```
Iter 15: f = 254.63015576
Optimization terminated successfully.

Optimizing without printing:
Optimization terminated successfully.
```

### 2.5.3 Accessing derivatives.

The function value and derivatives are methods of the optimization objective. Under the hood, all the derivatives are calcualted with `autograd`.

[6]:
```
opt_x = opt_result.x

print('f() evaluates the objective:\n{} = {}'.format(
    objective.f(opt_x), get_free_loss(opt_x)))

grad = objective.grad(opt_x)
print('\ngrad() evaluates the gradient:\n{}'.format(grad))

print('\nhessian() evaluates the Hessian matrix:\n{}'.format(objective.hessian(opt_
↪x)))

print('\nhessian_vector_product() evaluates the Hessian-vector product:\n{}'.format(
    objective.hessian_vector_product(opt_x, grad)))
```

```
f() evaluates the objective:
254.47818844774238 = 254.47818844774238

grad() evaluates the gradient:
[-9.38462682e-07 -8.28401150e-07  1.12731491e-06 -9.35645267e-07
  3.23708426e-06]

hessian() evaluates the Hessian matrix:
[[ 4.80152064e+03 -4.37310150e+03  9.83363903e-06 -6.91901344e-06
  -1.47624300e-06]
 [-4.37310150e+03  4.90797543e+03 -8.10149773e-06  8.64973804e-06
   1.65680224e-06]
 [ 9.83363896e-06 -8.10149785e-06  6.30553475e+03 -4.59689665e+03
  -1.75268882e-06]
 [-6.91901349e-06  8.64973810e-06 -4.59689665e+03  4.90797543e+03
   1.87129286e-06]
 [-1.47624310e-06  1.65680237e-06 -1.75268281e-06  1.87129159e-06
   1.99999999e+03]]

hessian_vector_product() evaluates the Hessian-vector product:
[-8.83365609e-04  3.82200615e-05  1.14093879e-02 -9.77427408e-03
  6.47416850e-03]
```

### 2.5.4 Logging.

By default, an optimization objective does no logging. But by using `set_log_every` with a number larger than `0`, a log of the values is kept.

[7]:
```
print('\nOptimizing with logging:')
objective.reset()
objective.set_print_every(0)
objective.set_log_every(2)
opt_result = Optimize(objective)
print(opt_result.message)
```

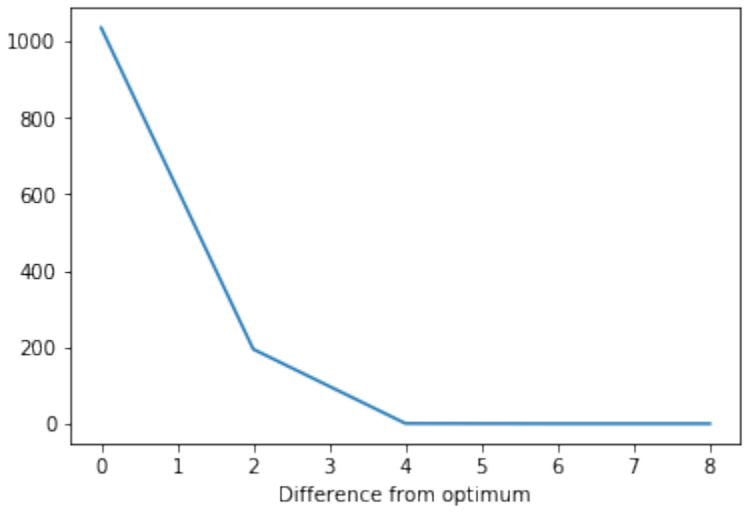
```
Optimizing with logging:
Optimization terminated successfully.
```

The log is contained in the attribute `optimization_log`. By default, each entry in the log is a tuple containing the iteration number, argument, and function value.

[8]:
```
for log_entry in objective.optimization_log:
    print(log_entry)

plt.plot(
    [ entry[0] for entry in objective.optimization_log],
    [ entry[2] - opt_result.fun for entry in objective.optimization_log])
plt.xlabel('Iteration')
plt.xlabel('Difference from optimum')
```

```
(0, array([0., 0., 0., 0., 0.]), 1290.2241213426291)
(2, array([ 0.09798938,  0.57638205,  0.10194361,  0.92848007, -1.16334881]), 449.
↪1536736241908)
(4, array([ 0.10983197,  0.58725075,  0.05844622,  0.94244548, -0.80874364]), 255.
↪05895261654913)
(6, array([ 0.13464741,  0.61326864,  0.04907768,  0.9356453 , -0.7953923 ]), 254.
↪47898697444145)
(8, array([ 0.13480492,  0.61343579,  0.04990915,  0.93661793, -0.79543076]), 254.
↪4781884477883)
```

[8]:
```
Text(0.5, 0, 'Difference from optimum')
```




The method `reset()` clears the log as well as the iteration count. Note that you can clear the log only using `reset_log()` and clear the iteration count only using `reset_iteration_count()`.

[9]:
```
objective.reset()
print(objective.optimization_log)
```

```
[]
```

### 2.5.5 Overloading.

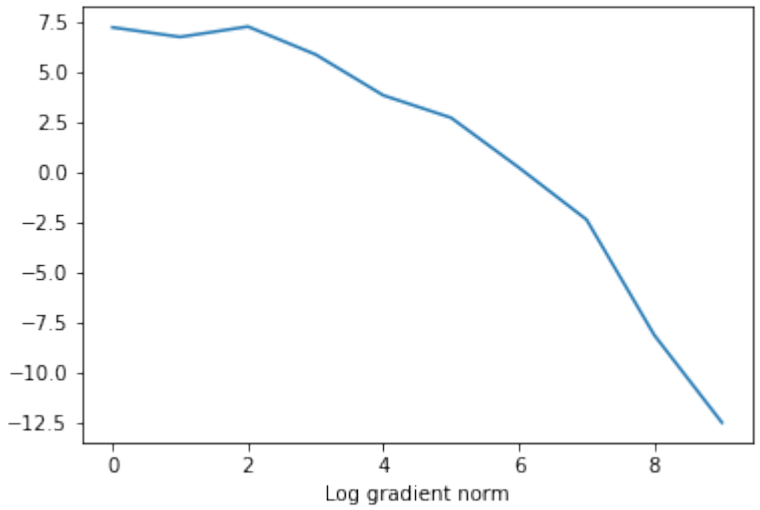
Both the logging and progress methods can be overloaded to provide custom behavior.

[17]:
```
class CustomObjective(paragami.OptimizationObjective):
    def print_value(self, num_f_evals, x, f_val):
        grad = objective.grad(x)
        print('Iteration {}: gradient norm = {:0.8f}'.format(
            num_f_evals, np.linalg.norm(grad)))

    def log_value(self, num_f_evals, x, f_val):
        grad = self.grad(x)
        self.optimization_log.append((num_f_evals, np.linalg.norm(grad)))

custom_objective = CustomObjective(get_free_loss)
```

[22]:
```
print('\nOptimizing with custom printing and logging:')
custom_objective.reset()
custom_objective.set_print_every(1)
custom_objective.set_log_every(1)
opt_result = Optimize(custom_objective)
print(opt_result.message)

plt.plot(
    [ entry[0] for entry in custom_objective.optimization_log],
    np.log([ entry[1] for entry in custom_objective.optimization_log]))
plt.xlabel('Iteration')
plt.xlabel('Log gradient norm')
```

```
Optimizing with custom printing and logging:
Iteration 0: gradient norm = 1325.81210303
Iteration 1: gradient norm = 822.69307418
Iteration 2: gradient norm = 1380.15778047
Iteration 3: gradient norm = 343.45510333
Iteration 4: gradient norm = 44.95265409
Iteration 5: gradient norm = 14.74702496
Iteration 6: gradient norm = 1.23658409
Iteration 7: gradient norm = 0.09246396
Iteration 8: gradient norm = 0.00029502
Iteration 9: gradient norm = 0.00000377
Optimization terminated successfully.
```

[22]:
```
Text(0.5, 0, 'Log gradient norm')
```


7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
0
2
4
6
8
Log gradient norm

# CHAPTER 3

# API

## 3.1 Pattern Parent Class

Every pattern described herin 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 val_freeflat / d 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*, *default_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 >= 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

| | |
|---|---|
| **array_shape: tuple of ints** | The shape of the array of simplexes, not including the simplex dimension. |
| **simplex_size: int** | The length of each simplex. |
| **shape: tuple of ints** | 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 intial 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

| | |
|---|---|
| **lock:** | 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 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`

**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 val_freeflat / d 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 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`

**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 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 val_freeflat / d 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*)
Get a function whose input has been preconditioned.

Throughout, the subscript `_c` will denote quantiites or funcitons 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 $x \to x_c = A^{-1}x$, 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

| | |
|---|---|
| **set_preconditioner:** | Set the preconditioner to a specified value. |
| **set_preconditioner_with_hessian:** | Set the preconditioner based on the Hessian of the objective at a point in the orginal domain. |
| **precondition:** | Convert from the original domain to the preconditioned domain. |
| **unprecondition:** | Convert from the preconditioned domain to the original domain. |

**__init__**(*original_fun*)

**Parameters**

**original_fun:** callable function of a single argument

**preconditioner:** The initial preconditioner.

**preconditioner_inv:** The inverse of the initial preconditioner.

**get_preconditioner**(*dim*)
Return a matrix representation of the preconditioner. This may be expensive.

**get_preconditioner_inv**(*dim*)
Return a matrix representation of the preconditioner. This may be expensive.

**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_functions**(*a_times*, *a_inv_times*)
Set the preconditioner with a functions that perform matrix multiplication.

**set_preconditioner_matrix**(*a*, *a_inv=None*)
Set the preconditioner with a matrix.

**set_preconditioner_with_hessian**(*x=None*, *hessian=None*, *ev_min=None*, *ev_max=None*)
Set the precoditioner 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 eigenvaluse of `hessian` that are less than `ev_min` to `ev_min` before taking the square root.

**ev_maxs: float** If not None, set eigenvaluse of `hessian` that are greater than `ev_max` to `ev_max` before taking the square root.

**Returns**

**Sets the precoditioner 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

| | |
|---|---|
| **f:** | The objective function with logging. |
| **grad:** | The gradient of the objective function. |
| **hessian:** | The Hessian of the objective function. |
| **hessian_vector_product:** | The Hessian vector product of the objective function. |
| **set_print_every:** | Set how often to display optimization progress. |
| **set_log_every:** | Set how often to log optimization progress. |
| **reset_iteration_count:** | Reset the number of iterations for the purpose of printing and logging. |
| **reset_log:** | Clear the log. |
| **reset:** | Run `reset_iteration_count` and `reset_log`. |
| **print_value:** | Display a function evaluation. |
| **log_value:** | 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(paragmi.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)
```

# Index

## Symbols



## A



## D



## E



## F

## G



## L



## N



## O



## P



## R



## S

## T



## U



## V