# miniff

*Release 0.1.4*

**miniff authors**

**Aug 10, 2021**

# CONTENTS:

`miniff` uses neural networks to compute energies and forces of atomic systems. Read more about the method from Behler and Parrinello. Preferred installation method is `pip`

```
pip install miniff
```

The source code is hosted at Quantum Tinkerer. Report bugs here. BSD license.

CHAPTER

ONE

# INTRODUCTION

## 1.1 Glossary

- **atom**, **point**: a vector in real (typically, 3D) space defining the location and the metadata associated with this location. This gives us information about an individual small object in space.

- **structure**, **cell**, **box**: a set of *points* enclosed into a parallelogram and representing a snapshot of atoms in media (molecular, solid, liquid, etc.). A **box** is a way to describe how multiple *atoms* share the same space. This can simply be a list of coordinates, for example. Related classes: `miniff.kernel.Cell`, `miniff.kernel.CellImages`.

- **potential**, **interaction**, **spring**: a protocol (a function) taking atomic coordinates of two or three *atoms* (optionally, matching a set of conditions) and producing a single floating-point number. For example, **potentials** may describe how strongly atoms interact with each other.

- **atomic environment**: a single *point* picked in a *structure*. **Atomic environment** is a very abstract way of telling which *interactions* are important and how *atoms* are grouped by these interactions.
- **partial energy**, **atomic energy**, **potential energy**: a sum of all *interaction* values the chosen atom participates in. The **potential energy** value may be subject to double-counting issues when a single *potential* is shared between many atoms. Related classes: `miniff.potentials.LocalPotential`, `miniff.ml.NNPotential`.
- **machine learning (ML) potential**: a variant of the *partial energy* where the sum is replaced by a more complex process involving machine learning techniques.
- **(total) energy**: a sum of all *atomic energies* defining the cumulative energy accumulated in the *structure* and originating from attractions and repulsions of individual *atoms*. Predicting **total energy** from **structures** is the primary purpose of this package.
- **(total) energy landscape**: *total energy* as a function of one or more parameters of a *structure*. It is simply a way to look at the *total energy* as a scalar function of many variables.
- **(total) energy minimum**: a *structure* and the corresponding *total energy* minimum of the *potential landscape*. Finding *potential energy minimum* is one of the primary purposes of the *total energy* description.
- **charge**: a scalar belonging to *atomic* metadata with the properties of *partial energy*. **Charges** are used to treat long-range *potentials* which cannot be described by *atomic environments*. **Charges** are not necessary physical (Coulomb) charges: they may also be electronegativities or any other scalar property of an *atom*.
- **force**, **stress**: negative gradients of the *total energy* with respect to coordinates describing *structures*. *Forces* indicate the direction in a multidimensional space where the *total energy* becomes smaller.

- **descriptors**: a special sort of *potential energy* intended to describe *atomic environments*. Descriptors are typically defined by *local environments*. Unlike *potentials*, **descriptor** functions usually have a simple smooth form.

CHAPTER

TWO

# USING MINIFF

## 2.1 As an executable

Standard workflows in `miniff` can be accessed by running:

```
python -m miniff jobs.yml
```

where `jobs.yml` lists job parameters. For example:

```
fit:
  prepare:
    fn_cells: fit-structures.json
  run:
    n_epochs: 10
    save: potentials.pt

test-direct:
  prepare:
    fn_cells: test-structures.json
    potentials: potentials.pt
```

Root arguments specify what kind of workflow to perform: `fit`, `test-direct`, and others. For each workflow, arguments are stored in three section corresponding to `Workflow.__init__`, `Workflow.prepare`, and `Workflow.run`. If either of the section is absent the corresponding workflow stage will still run with the default arguments. If multiple workflows are specified, they will be invoked in the order they are present in the job file.

### 2.1.1 Arguments

The available arguments are specified in documentation of `miniff.ml_util`.

## 2.2 As a library

`miniff` is a python library with a plain modular structure. Depending on the needs, it can be used integrally or by individual pieces.

### 2.2.1 Workflows

TBD

### 2.2.2 Dataset construction

TBD

### 2.2.3 Descriptors and potentials

TBD

CHAPTER

THREE

# TUTORIALS

## 3.1 Toy function fitting

`miniff` is built around the idea of using neural-network fits to describe inter-atomic interactions. This is done in three basic steps:

1. constructing the dataset;
2. fitting the data (= finding optimal map parameters);
3. using the fit.

This tutorial demonstrates the first two steps in the context of simple scalar function fitting. We will pick a function and will try to perform a least-squares fit. Then, we will repeat the procedure with the help of `miniff.ml` and `miniff.ml_util` modules to introduce the key concepts of dataset construction and fitting in `miniff`.

### 3.1.1 The toy problem

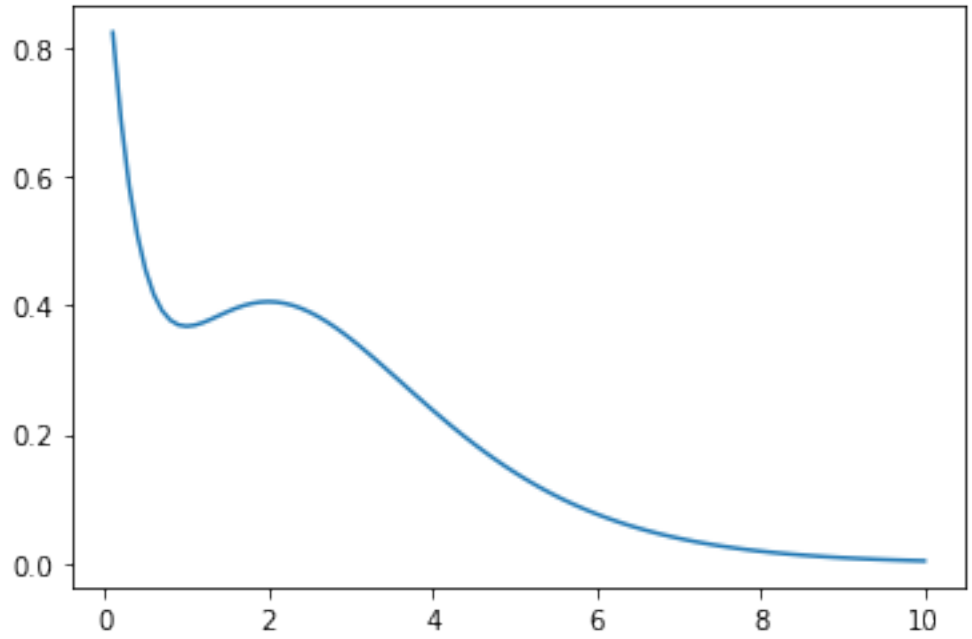
Neural-network fitting is useful for interpolating black-box functions which otherwise require intensive computations. For pedagogical reasons, let us instead try to fit a simple analytic function defined for positive `r` and resembling some features of realistic interatomic potentials:

$$f(r) = \left(r^2 - r + 1\right) e^{-r}.$$

Let's first plot the function:

```
import numpy as np
from matplotlib import pyplot

def f(r):
    return (r ** 2 - r + 1) * np.exp(-r)

r = np.linspace(0.1, 10, 100)
f_r = f(r)
pyplot.plot(r, f_r)
```

```
[<matplotlib.lines.Line2D at 0x7f777a4ae4d0>]
```




The function has two extrema at `r = 1, 2` and decays rapidly: these are the features which we will attempt to reproduce.

When fitting, we only have access to discretized sampling values `f_r`, but not to the functional form `f(r)`. In principle, we may still use `f(r)` as a black box to produce more sampling points but we consider this operation to be computationally expensive, thus, to only be used *before* the fitting.

### 3.1.2 1. Constructing the dataset

The fitting data (or the dataset) includes two pieces of information: blackbox outputs and descriptors. They can be also seen as right-hand and left-hand sides of the map `g(r)->f(r)` respectively, where `f(r)` are previously computed blackbox function values `f_r`.

#### What are descriptors?

The purpose of descriptors `g(r)` is to describe values of `r` where `f(r)` was computed in a physically reasonable way. Can we use values of `g(r)=r` directly? Yes, but `g(r)` play the role of pre-conditioners: we input some intuition about how `f(r)` would most reasonably look like while leaving particular details to be determined during the fitting process.

Additionally, we allow the blackbox function to have an arbitrary number of inputs such as

$$f\left(\vec{r}\right) = \sum_i \left(r_i^2 - r_i + 1\right) e^{-r_i}.$$

In this case we may want to construct the dataset from vectors of a non-constant dimension `r = ([0], [0.1, 0.2], [2.2, 3, 2], ...)`. To employ neural network machinery, however, we usually need a constant number of inputs. This can be achieved by using a descriptor of the form

$$g\left(\vec{r}\right) = \sum_i r_i$$

in place of `r`. Importantly, the above descriptor also respect the permutation symmetry of `f(r)`: `f(x, y) = f(y, x)`, `g(x, y) = g(y, x)`. Thus, the follow-up fitting process does not need to deduce this symmetry in a hard way numerically.

### How to choose descriptors?

- `g(r)` should include as much intuition as possible (symmetries, asymptotic behavior, ...);
- `g(r)` should be simple enough functions fast to compute.

Here we choose the following family of descriptors which obviously satisfies both recommendations:

$$g_i(r) = r^i e^{-r}.$$

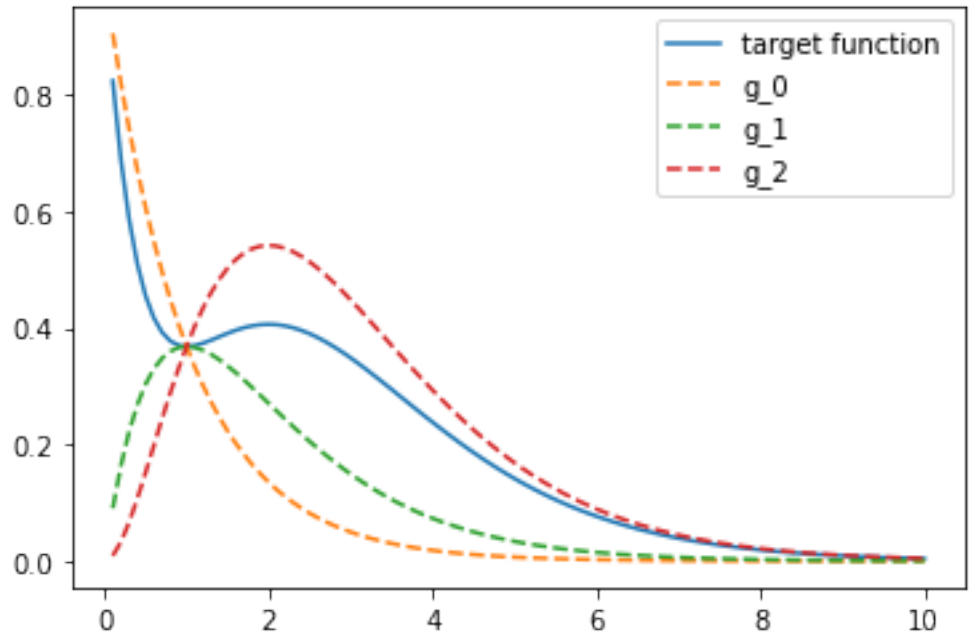
We also know the exact form of the map to be fitted:

$$f(r) = g_2(r) - g_1(r) + g_0(r).$$

So, to complete the dataset we compute `g_0, g_1, g_2` at previously chosen sampling points.

```
def g(r, i):
    return r ** i * np.exp(-r)

g_r = np.array(tuple(g(r, i) for i in range(3)))
pyplot.plot(r, f_r, label="target function")
for i, _g in enumerate(g_r):
    pyplot.plot(r, _g, ls="--", label=f"g_{i}")
pyplot.legend()
```

```
<matplotlib.legend.Legend at 0x7f77498d4450>
```



### 3.1.3 2. Fitting the data

For this example whe know there exists a linear combination of descriptors perfectly reproducing the black-box function

$$f(r) = \sum_i a_i \cdot g_i(r).$$

Thus, the least-squares fit will find map parameters `a_i`.

```
gf_map, res, rank, s = np.linalg.lstsq(g_r.T, f_r)
print(gf_map)

from numpy import testing
testing.assert_allclose(gf_map, (1, -1, 1))
```

```
[ 1. -1.  1.]
```

According to the last formula in the previous section, the answer is `[1, -1, 1]`.

### 3.1.4 miniff setting

The above solution can be also obtained from `miniff` with the same workflow.

#### 1. Constructing the dataset (miniff)

`miniff.ml` contains handy pipelines for dataset construction and pre-processing. `PerCellDataset` and `PerPointDataset` are the two key containers where descriptors and target function values are stored. Some data pieces (such as descriptors, but also atomic charges and more) can be assigned to individual points (atoms) and are stored in `PerPointDataset`. Other data pieces (such as total interatomic energies) are cumulative values that are stored in `PerCellDataset`. Due to the nature of force-field fitting, there can be only one `PerCellDataset` but one or more ``PerPointDataset``s.

In this case we store `f_r` into the total interaction energy slot of `PerCellDataset` to benefit from the energy fitting pipeline.

```
import torch
from miniff import ml

rhs = ml.PerCellDataset(
    energy=torch.tensor(f_r, dtype=torch.float32)[:, None],
)
```

All dataset pieces have to be `torch` tensors of a fixed shape. In this case "energy" is required to be a column array with the second dimension equal to 1: otherwise a `ValueError` will be raised.

Descriptors are stored in `PerPointDataset` as a standard.

```
lhs = ml.PerPointDataset(
    features=torch.tensor(g_r.T, dtype=torch.float32)[:, None, :],
    mask=torch.ones(len(r), 1),
)
```

Here, we additionally specify the mask which weights individual data points. Again, the dimensions of input tensors are well-defined. As a final step, we collect the two pieces of fitting data into the `Dataset` object.

```
dataset = ml.Dataset(rhs, lhs)
```

At this point all tensor shapes are checked and we are ready to set up the fitting.

### 2. Fitting the data (miniff)

First, we need to define the functional form of the map. In this case the map is linear which can be declared using `torch.nn.Linear`.

$$f = W \cdot g,$$

where $W$ stands for weights.

```
nn = torch.nn.Linear(in_features=3, out_features=1, bias=False)
```

Here, `3` stands for the descriptor count, `1` is output count (single energy), `bias=False` tells that no constant is added to the linear form.

To obtain the best fit we use energy fit optimisation pipeline defined in `miniff.ml_util`. First, we define the closure to be energy difference closure. In general, closure defines the loss function to optimize but the corresponding `ml_util` object also includes default optimization settings.

```
from miniff import ml_util

tolerance = dict(
    tolerance_change=1e-14,
    tolerance_grad=1e-14,
)

closure = ml_util.simple_energy_closure([nn], dataset,
    optimizer_kwargs=tolerance)
```

Let's check that the initial (random) guess is non-optimal.

```
print("Initial loss:", closure.loss().loss_value.item())
```

```
Initial loss: 0.010355422273278236
```

Finally, let's run the optimization and check the loss function value becomes small.

```
closure.optimizer_init()
closure.optimizer_step()  # LBFGS algorithm by default
loss = closure.last_loss.loss_value.item()
print("Final loss:", loss)
assert loss < 1e-8
```

```
Final loss: 1.784633621844346e-15
```

Parameters are now stored in the neural network module.

```
weights = nn.weight.detach().squeeze().numpy()
print(weights)
testing.assert_allclose(weights.squeeze(), [1, -1, 1], atol=1e-3)
```

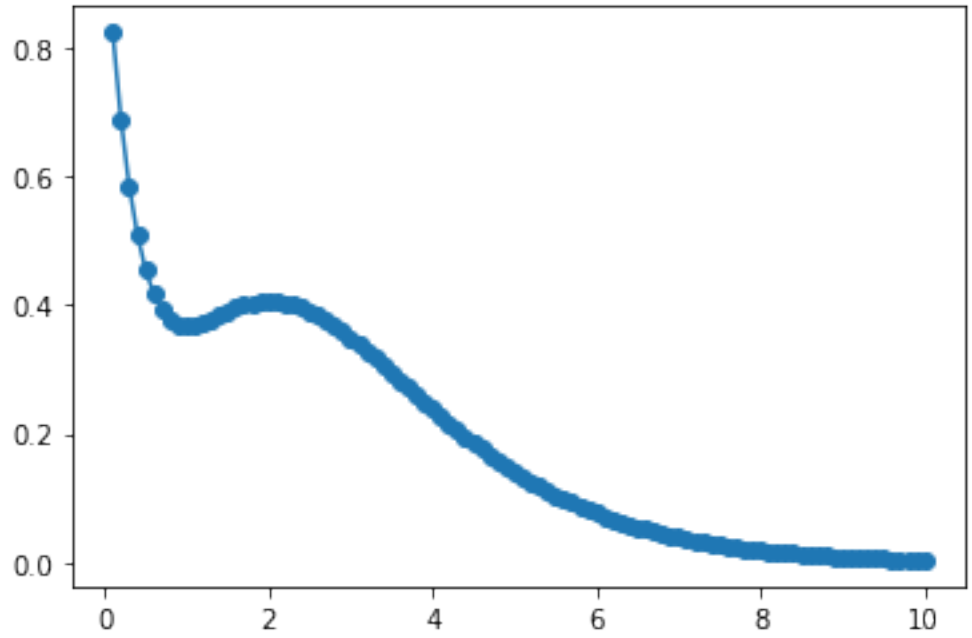
```
[ 1.0000002 -1.0000002  1.0000001]
```

### Checking the result

Finally, let's check whether the fit is reasonable by plotting it.

```
def nn_fit(r, descriptor_fns, nn):
    # Compute descriptors
    descriptors = np.array(tuple(g(r) for g in descriptor_fns))
    # Compute the map
    result = nn(torch.tensor(descriptors.T, dtype=torch.float32))
    return result.detach().numpy().squeeze()

from functools import partial

f_r_test = nn_fit(
    r,
    tuple(partial(g, i=i) for i in range(3)),
    nn,
)
pyplot.plot(r, f_r)
pyplot.scatter(r, f_r_test)
```

```
<matplotlib.collections.PathCollection at 0x7f76dffe0c50>
```




The function `nn_fit` demonstrates how to propagate the map forward: one needs to (1) compute descriptors as a function of input `r` and (2) to feed them to the neural network.

### 3.1.5 Advanced

It does not make sense to use anything but linear map for this particular problem with a known exact solution. However, if we chose to, for example, reduce the descriptors to `g_0` only or to chose different descriptors then the linear map becomes a crude approximation. In this case one typically switches to a more general neural-network setting with non-linear activation layers helping to map various curvature features in the dataset. One can use `torch` machinery for this or just otherwise increase `n_layers` when initializing the map. With `n_layers=2`, for example, one sigmoid activation layer is inserted between two linear layers.

The use of `ml_util` functions is entirely optional. They aim to provide default optimization setting to start with: loss functions (2-norm), optimizers (LBFGS), descriptor sets (Behler recipe). You are free to modify and replace these components at the point when you feel that the default choice is no longer optimal.

## 3.2 Computing descriptors

Computing descriptors is an important part of both constructing datasets and using neural-network fits.

### 3.2.1 The problem

Consider a unit box in 3 dimensions. `n = 100` atoms are present in the box; their cartesian coordinates are written in a `[100 x 3]` matrix `r` such that `0 <= r <= 1`.

```
import numpy as np

n = 100
a = 0.3
r = np.random.rand(100, 3)
```

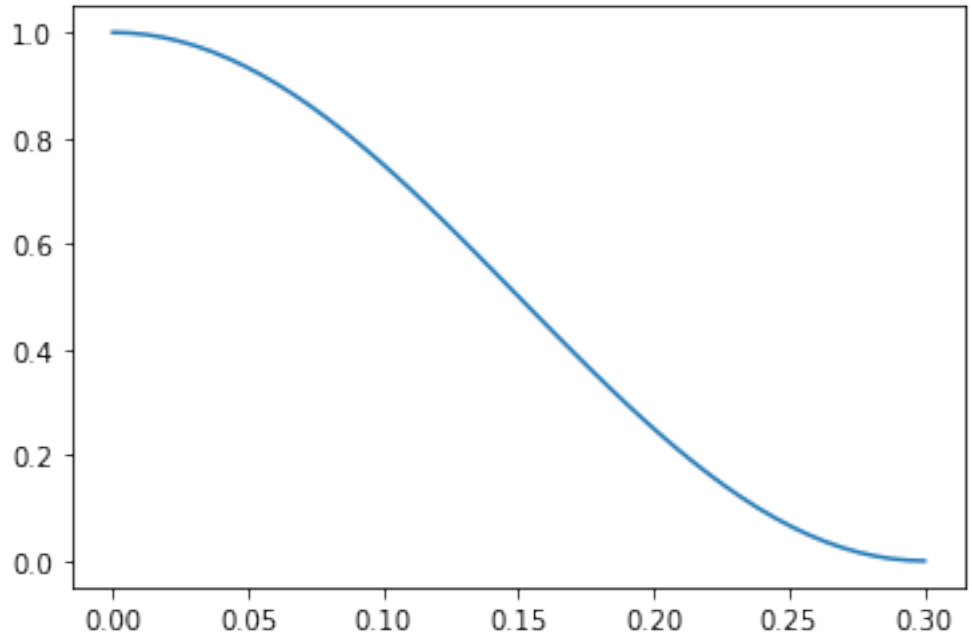
Our task is to count atomic neighbors inside a shell of size `a = 0.3` for each atom. This is what a typical descriptor looks like: it counts neighbors in shells around each atom.

To make a continuous function out of neighbor count let's introduce a kernel function of neighbor distance `r` smoothly decaying from a unit value at `r = 0` down to zero at cutoff value `r = a`. Let's take a half-period of cosine function for this example.

$$g(r) = \frac{1 + \cos\frac{\pi r}{a}}{2} \quad r < a$$

```
def g(r):
    return (1 + np.cos(np.pi * r / a)) / 2 * (r < a)

from matplotlib import pyplot
x = np.linspace(0, a)
pyplot.plot(x, g(x))
```

```
[<matplotlib.lines.Line2D at 0x7fc0cd9424d0>]
```




Instead of computing neighbors directly let's compute the cumulative value of the kernel function for all neighbors.

$$n_i = \sum_j g(r_{ij})$$

### 3.2.2 Computing naively

We start with computing all pair distances between atoms.

```
mdist = np.linalg.norm(r[:, None, :] - r[None, :, :], axis=-1)
```

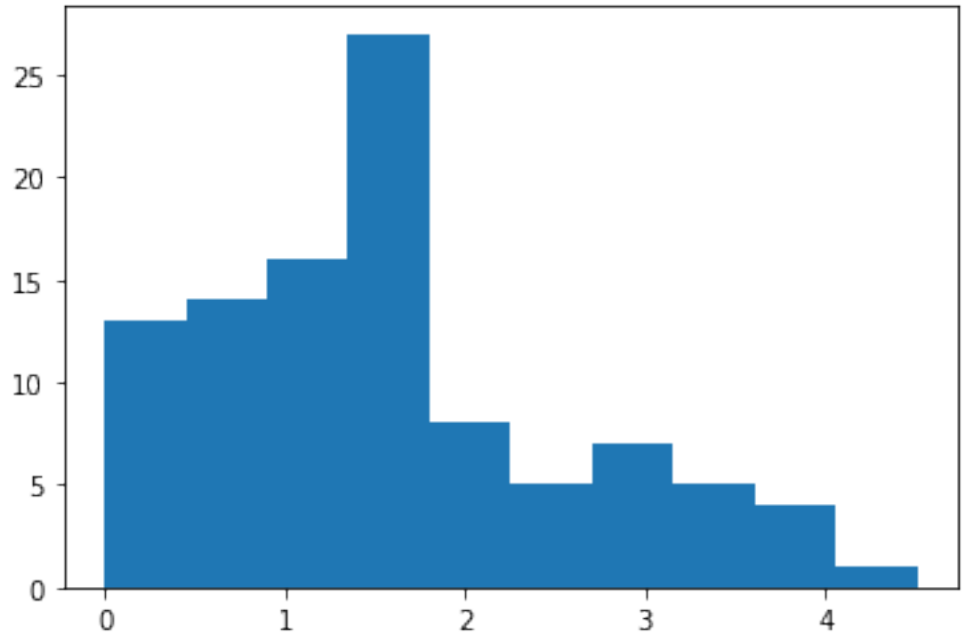
Then, we reduce pair distances using the kernel function.

```
result = g(mdist).sum(axis=0) - 1
```

Finally, we can plot the distribution of neighbors across all points.

```
from matplotlib import pyplot
pyplot.hist(result)
```

```
(array([13., 14., 16., 27.,  8.,  5.,  7.,  5.,  4.,  1.]),
 array([0.        , 0.45074515, 0.90149029, 1.35223544, 1.80298058,
        2.25372573, 2.70447087, 3.15521602, 3.60596116, 4.05670631,
        4.50745146]),
 <BarContainer object of 10 artists>)
```




This approach is straightforward but not efficient. First, matrix `mdist` includes information about very distant atoms we simply discard. Second, we can only perform pair reductions this way: counting atomic triples within the same sphere of radius `a` will require a tensor with three dimensions raising memory requirements by a factor of `n = 100`. Instead, `miniff` uses a more efficient approach with a quasi-linear complexity scaling.

### 3.2.3 Computing neighbors with miniff

`miniff` provides an interface to computing neighbors efficiently. It uses `KDTree` from `scipy` to compute neighbor lists and `cython` functions to iterate over the list and to reduce it to the desired quantities. As such, there are two key steps.

#### 1. Compute neighbor lists

The primary purpose of `miniff.kernel` module is to perform neighbor list construction using `compute_images` method and `CellImages` class. To do it, we first construct a unit cell object with atomic coordinates.

```
from miniff.kernel import Cell, compute_images

cell = Cell(np.eye(3), r, ['a'] * len(r))
```

The second step is to compute its images (neighbors) given the cutoff value and other periodicity information.

```
images = compute_images(cell, cutoff=a, pbc=False)
print(len(images.distances.data))
```

```
794
```

As a result, `images.distances` contains neighbor distance data.

### 2. Reduce

In principle, neighbor counts can be deduced from the sparse matrix `images.distances` directly with previously defined method `g`. A more conventional workflow is to pick a reduction function from `miniff.potentials` module. Specifically, `miniff.potentials.sigmoid_descriptor_family` provides the following reduction function:

$$n_i = \sum_j \frac{1}{1 + e^{\frac{r_{ij} - r_0}{dr}}} \cdot \frac{1 + \cos\frac{\pi r_{ij}}{a}}{2},$$

where `r_ij` is the distance matrix, `r0`, `dr` and `a` are constant parameters. By setting `dr` to a small value and `r0` to `2a` we simplify the expression to the second term only

$$n_i \approx \sum_j \frac{1 + \cos\frac{\pi r_{ij}}{a}}{2},$$

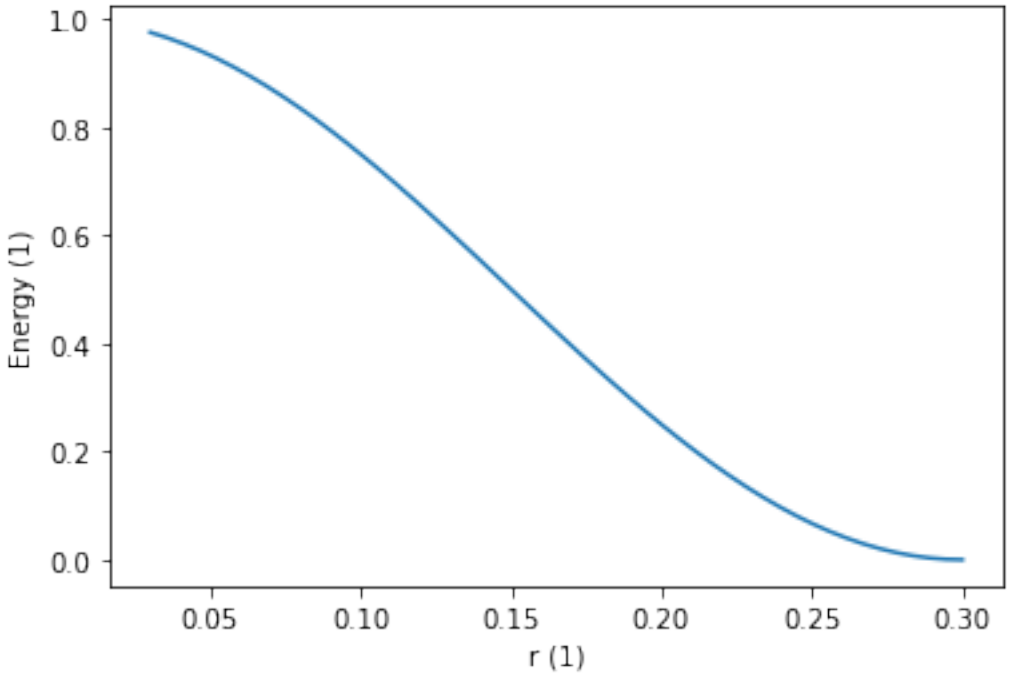
resembling the kernel above. To instantiate the descriptor we simply call the factory object the specified parameters.

```
from miniff.potentials import sigmoid_descriptor_family

descriptor = sigmoid_descriptor_family(r0=10 * a, dr=a / 10, a=a)
```

Let's visualise the radial form of the descriptor.

```
from miniff.presentation import plot_potential_2

plot_potential_2(descriptor, energy_unit="1", length_unit="1")
```

```
<AxesSubplot:xlabel='r (1)', ylabel='Energy (1)'>
```




To perform the reduction we simply feed the descriptor to `images.eval`.

```
result_miniff = images.eval(descriptor.copy(tag="a-a"), "kernel")
```

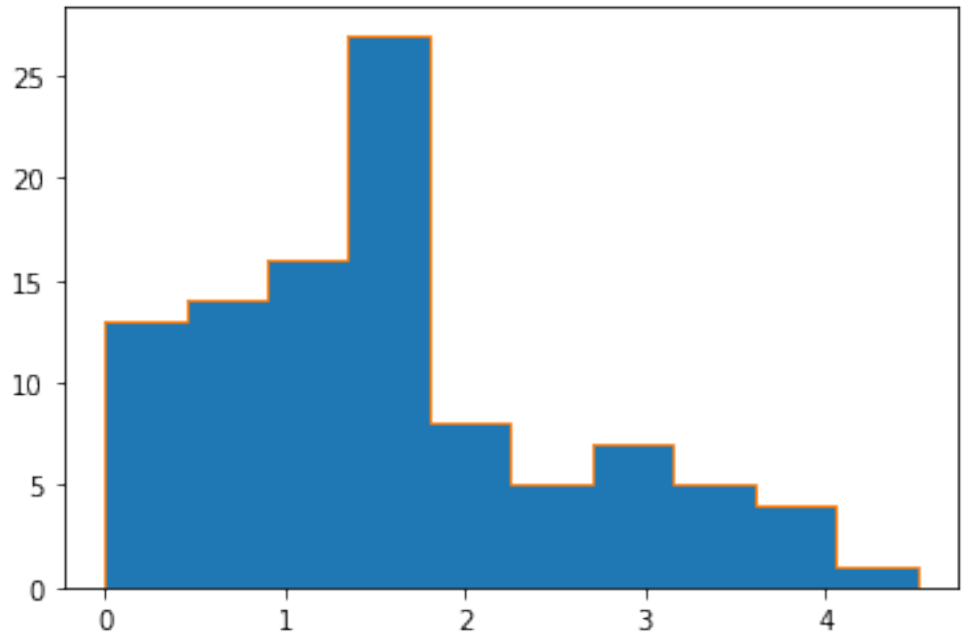
Not that `descriptor.copy(tag="a-a")` makes a copy of the descriptor with the `descriptor.tag = "a-a"`. The tag is required by `images.eval` to distinguish between species and possible pairs. For example, having two kinds of species in the box `"a"` and `"b"` we may reduce several kinds of pairs such as `"a-a"` or `"a-b"`.

Additionally, we specify the function we want to compute. The same descriptor may provide multiple functions at once: `"kernel"` stands for the descriptor value while `"kernel_gradient"` efficiently computes descriptor gradients.

Finally, let's plot and compare the result.

```
_, bins, _ = pyplot.hist(result)
pyplot.hist(result_miniff, bins, histtype="step")
```

```
(array([13., 14., 16., 27.,  8.,  5.,  7.,  5.,  4.,  1.]),
 array([0.        , 0.45074515, 0.90149029, 1.35223544, 1.80298058,
        2.25372573, 2.70447087, 3.15521602, 3.60596116, 4.05670631,
        4.50745146]),
 [<matplotlib.patches.Polygon at 0x7fc065414290>])
```




### 3.2.4 Advanced

`miniff.potentials` includes several most common reduction functions. But it is also possible to define yours using `miniff.potentials.general_pair_potential_family`. It accepts two methods: `f(r)` and `df_dr(r)`. As `f` is a python function called multiple times during the reduction, it will not benefit from `cython` optimizations and parallelism. Nevertheless, it is a handy prototyping tool.

## 3.3 Classical potentials

`miniff` is a minimal implementation of classical force fields: this tutorial demonstrates how to implement force fields for bulk Silicon. More exactly, we will reproduce Fig. 1 of Stillinger and Weber (1985) where they propose and benchmark a specific form of the classical potential.

### 3.3.1 The problem

Stillinger and Weber propose to use a sum of the following pair and a triple potentials to model chemical forces between atoms in Silicon, Eqs. 2.3 and 2.5.

$$f_2(r) = A(Br^{-p} - r^{-q}) \cdot \exp[(r-a)^{-1}]$$

$$f_3(r_1, r_2, \theta) = \lambda \exp[\gamma(r_1 - a)^{-1} + \gamma(r_2 - a)^{-1}] \times (\cos\theta + \cos\theta_0)^2$$

Fig. 1 presents how the sum of these potentials scales with atomic density for different types of cubic cells.

### 3.3.2 Computing with miniff

Stillinger-Weber potential form is readily available in `miniff.potentials`. To instantiate potentials we specify parameter values from the manuscript: `A`, `B`, `p`, `q`, `a`, , .

```
from miniff.potentials import sw2_potential_family, sw3_potential_family

si_gauge_a = 7.049556227
si_gauge_b = 0.6022245584
si_p = 4
si_q = 0
si_a = 1.8
si_l = 21
si_gamma = 1.2

sigma = 1  # length unit
epsilon = 0.5  # energy unit: fixes double counting

si2 = sw2_potential_family(
    gauge_a=si_gauge_a, gauge_b=si_gauge_b,
    a=si_a, p=si_p, q=si_q,
    epsilon=epsilon, sigma=sigma)

si3 = sw3_potential_family(
    l=si_l, gamma=si_gamma, cos_theta0=-1./3, a=si_a,
    epsilon=epsilon, sigma=sigma)
```

In addition, `epsilon` and `sigma` provide the energy and length scales, respectively. The second step is to construct unit cells with different symmetries: simple cubic (SC), body-centered cubic (BCC), face-centered cubic (FCC), and diamond. For this, we use `kernel.Cell`. We do not bother about lattice parameters for the moment: we will recompute them from the density of atoms.

```
from miniff.kernel import Cell
import numpy as np
```

(continues on next page)

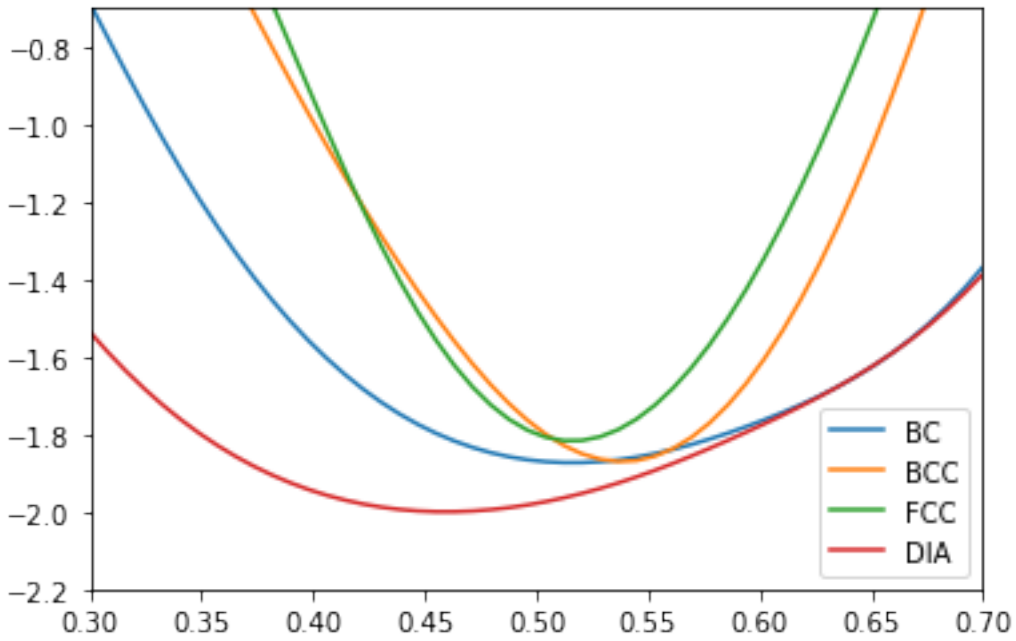
(continued from previous page)

```
# simple cubic: one atom in a cubic box
c_sc = Cell(np.eye(3), np.zeros((1, 3)), ["si"], meta=dict(tag="BC"))

# BCC: one additional atom at the center of the box
c_bcc = Cell(np.eye(3), [[0, 0, 0], [.5, .5, .5]], ["si", "si"], meta=dict(tag="BCC"))

# FCC: three additional atoms at box face centers
c_fcc = Cell(np.eye(3), [[0, 0, 0], [.5, .5, 0], [.5, 0, .5], [0, .5, .5]],
    ["si"] * 4, meta=dict(tag="FCC"))

# diamond: rhombic unit cell with two atoms
c_dia = Cell([[0, 1, 1], [1, 0, 1], [1, 1, 0]], [[0, 0, 0], [.25, .25, .25]],
    ["si"] * 2, meta=dict(tag="DIA"))

cells = [c_sc, c_bcc, c_fcc, c_dia]
```

Finally, we apply a uniform strain to these cells and compute the total energy as a function of the atomic density. The `kernel.profile_directed_strain` batches multiple energy computations into a single function.

```
from miniff.kernel import profile_directed_strain
from matplotlib import pyplot

density = np.linspace(0.3, 0.7)  # target density values from the plot

for c in cells:
    d = c.size / c.volume  # actual density
    e = profile_directed_strain([si2, si3], c, (d / density) ** (1./3), (1, 1, 1))
    pyplot.plot(density, e / c.size, label=c.meta["tag"])

pyplot.ylim(-2.2, -0.7)
pyplot.xlim(0.3, 0.7)
pyplot.legend()
```

```
<matplotlib.legend.Legend at 0x7fcf331b8f50>
```




### 3.3.3 Tips

`presentation.plot_strain_profile` combines `profile_directed_strain` with plotting routines and constructs a similar figure.

CHAPTER

FOUR

# MINIFF DESIGN DOCUMENTATION

## 4.1 Structure

The project includes several modules performing energy and gradients computations, potential parameter optimization through machine learning, simple geometry optimization and presentation utilities.

- `potentials`: a core module implementing smooth classical interatomic potentials and descriptors. The module includes `LocalPotentialFamily`: a factory for constructing parameterized `LocalPotential` objects which, in turn, include all necessary data and interfaces to compute atomic energies and gradients. Several pre-built potential forms are provided through instantiating `LocalPotentialFamily`: for example, `lj_potential_family` (Lenard-Jones pair potential) or `behler5_descriptor_family` (Behler type 5 angular descriptor).
- `kernel`: implements a `CellImages` class which computes neighbor information from atomic coordinate data and prepares contiguous data buffers which can be processed by `LocalPotential``s. ``CellImages` is implemented with 3D periodic boundary conditions in mind to model amorphous materials. However, it also supports molecular systems without any overhead. `kernel` also implements a key interface `kernel.eval` which, for a given structure and a list of potentials, computes total energy and gradients.
- `ml`: implements machine learning potentials.
  - `ml.Dataset` is the key container for atomic descriptors, energies and gradients. It ensures that all dataset pieces are `torch.Tensor``s compatible with each other. ``ml.Dataset` includes two large blocks of data, namely one `ml.PerCellDataset` block with target energies and energy gradients, and one or more ``ml.PerPointDataset``s with descriptor information.
  - `ml.Normalization` implements normalization of datasets in a physically reasonable way.
  - `ml.learn_cauldron` is a typical entry point for dataset creation which includes reasonable default values.
  - `ml.SequentialSoleEnergyNN` is Behler et al. suggestion for the neural network potential form.
  - `ml.forward_cauldron` is the core routine for machine-learning optimization. It combines dataset and neural-network models to produce the energy and gradients prediction.
  - `ml.NNPotential` is a neural-network potential subclassing `potentials.LocalPotential`.

  `miniff.ml` is built around `pytorch`.
- `ml_util`: includes reasonable recipes for optimizing neural networks from `ml`.
  - `ml_util.simple_energy_closure` provides defaults for running the optimization with *LBFGS*.
  - `ml_util.*Workflow` are workflow classes for optimizing neural-network potentials. These classes accumulate and re-distribute many parameters related to the dataset organization, potential form and optimizatin process.
- `dyn`: a toy dynamics module implementing the search of local minima and atomic dynamics.

- `presentation`: various handy plotting routines to present potentials and visualize machine learning optimization process.

## 4.2 Deployment

*miniff* can be deployed on high-performance computing (HPC) clusters.

### 4.2.1 Parallelism

- `miniff` takes a full advantage of GPU parallelism in `pytorch`. Please note that it is often not enough to install pre-built bundles of `pytorch` as they support only a limited set of (very recent) GPU drivers. If your HPC hardware does not feature those you have several options:
  - It is best to ask your HPC support team for a suitable `pytorch` build specifically for the HPC machine. Such builds may be available through `module` script or other ways to manage the runtime environment on the cluster: please investigate such options first.
  - The second possibility is to use an older `pytorch` version which bundles kernels for older GPUs. `miniff` does its best to support a wide range of `pytorch` versions but you have to test the compatibility manually in your case.
  - The last possibility is to build `pytorch` manually. This is the most tedious approach, thus, not recommended for unexperienced users.
- OpenMP threading support is present in potential and gradient computations. This may be useful for computing energies and gradients in large atomic systems. The number of threads is controlled by usual means such as `OMP_NUM_THREADS` environment variable. For small atomic systems ~`100` atoms up to 2-4 threads are beneficial: make sure your parallel cluster setup is reasonable.

CHAPTER

FIVE

# INDICES AND TABLES

- genindex
- modindex
- search