Amp Documentation

Release 0.4

Andrew A. Peterson, Alireza Khorshidi

February 24, 2017

Contents

Welcome to Amp documentation! This project is being developed at Brown University in the School of Engineering, primarily by **Andrew Peterson** and **Alireza Khorshidi**, and is released under the GNU General Public License. This is a relatively new project, so things are constantly changing!

(Note that Amp is built off of our previous project, Neural. You may still find Neural at our bitbucket page.)

Table of Contents:

CHAPTER 1

Introduction

Amp is an open-source package designed to easily bring machine-learning to atomistic calculations. This allows one to predict (or really, interpolate) calculations on the potential energy surface, by first building up a regression representation of a "train set" of atomic images. Amp calculator works by first learning from any other calculator (usually quantum mechanical calculations) that can provide energy and forces as a function of atomic coordinates. In theory, these predictions can take place with arbitrary accuracy approaching that of the original calculator.

Amp is designed to integrate closely with the Atomic Simulation Environment (ASE). As such, the interface is in pure python, although several compute-heavy parts of the underlying codes also have fortran versions to accelerate the calculations. The close integration with ASE means that any calculator that works with ASE - including EMT, GPAW, DACAPO, VASP, NWChem, and Gaussian - can easily be used as the parent method.

CHAPTER 2

Installation

AMP is python-based and is designed to integrate closely with the Atomic Simulation Environment (ASE). In its most basic form, it has few requirements:

- Python, version 2.7 is recommended
- ASE
- NumPy (>= 1.9 for saving data in ".db" database format)
- SciPy

Install ASE

We always test against the latest version (svn checkout) of ASE, but slightly older versions (>=3.9.0) are likely to work as well. Follow the instructions at the ASE website. ASE itself depends upon python with the standard numeric and scientific packages. Verify that you have working versions of NumPy and SciPy. We also recommend matplotlib in order to generate plots.

Check out the code

As a relatively new project, it may be preferable to use the development version rather than "stable" releases, as improvements are constantly being made and features added. We run daily unit tests to make sure that our development code works as intended. We recommend checking out the latest version of the code via the project's bitbucket page. If you use git, check out the code with:

```
$ cd ~/path/to/my/codes
$ git clone git@bitbucket.org:andrewpeterson/amp.git
```

where you should replace '~/path/to/my/codes' with wherever you would like the code to be located on your computer. If you do not use git, just download the code as a zip file from the project's download page, and extract it into '~/path/to/my/codes'. Please make sure that the folder '~/path/to/my/codes/amp' includes the script '__init__.py' as well as the folders 'descriptor', 'regression', ... At the download page, you can also find historical numbered releases.

Set the environment

You need to let your python version know about the existence of the amp module. Add the following line to your '.bashrc' (or other appropriate spot), with the appropriate path substituted for '~/path/to/my/codes':

```
$ export PYTHONPATH=~/path/to/my/codes:$PYTHONPATH
```

You can check that this works by starting python and typing the below command, verifying that the location listed from the second command is where you expect:

```
>>> import amp
>>> print(amp.__file__)
```

Recommended step: Fortran modules

The code is designed to work in pure python, which makes it is easier to read, develop, and debug. However, it will be annoyingly slow unless you compile the associated fortran modules which speed up some crucial parts of the code. The compilation of Fortran codes and integration with the python parts is accomplished with the command 'f2py', which is part of NumPy. A Fortran compiler will also be necessary on your system; a reasonable open-source option is GNU Fortran, or gfortran. This complier will generate Fortran modules (.mod). gfortran will also be used by f2py to generate extension module 'fmodules.so' on Linux or 'fmodules.pyd' on Windows. In order to prepare the extension module take the following steps:

- Compile regression Fortran subroutines inside the regression folder by:

```
$ cd ~/path/to/my/codes/regression/
$ gfortran -c neuralnetwork.f90
```

- Move the module 'regression.mod' created in the last step, to the parent directory by:

```
$ mv regression.mod ../
```

- Compile the main Fortran subroutines in the parent directory in companian with the descriptor and regression subroutines by something like:

```
$ f2py -c -m fmodules main.f90 descriptor/gaussian.f90 regression/neuralnetwork.f90
```

or on a Windows machine by:

```
$ f2py -c -m fmodules main.f90 descriptor/gaussian.f90 regression/neuralnetwork.f90 --fcompiler=gnu95
```

If you update the code and your fmodules extension is not updated, an exception will be raised, telling you to re-compile.

Recommended step: Run the tests

We include tests in the package to ensure that it still runs as intended as we continue our development; we run these tests on the latest build every night to try to keep bugs out. It is a good idea to run these tests after you install the package to see if your installation is working. The tests are in the folder *tests*; they are designed to run with nose. If you have nose installed, run the commands below:

```
$ mkdir /tmp/amptests
$ cd /tmp/amptests
$ nosetests ~/path/to/my/codes/amp/tests
```

CHAPTER 3

How to use Amp?

Amp can look at the atomic system in two different schemes. In the first scheme, the atomic system is looked at as a whole. The consequent potential energy surface approximation is valid only for systems with the same number of atoms and identities. In other words the learned potential is size-dependent. On the other hand, in the second scheme, Amp looks at local atomic environments, ending up with a learned potential which is size-independent, and can be simultaneously used for systems with different sizes. The user should first determine which scheme is of interest.

Initializing Amp

The calculator as well as descriptors and regressions should be first imported by:

```
>>> from amp import Amp
>>> from amp.descriptor import *
>>> from amp.regression import *
```

Then Amp is initiated with a descriptor and a regression algorithm, e.g.:

```
>>> calc = Amp(descriptor=Gaussian()
                   regression=NeuralNetwork())
```

The values of arguments shown above are the default values in the current release of Amp. A size-dependent scheme can be simply taken by `descriptor=None`. Optional arguments for initializing Amp can be reviewed at main methods.

Starting from old parameters

The parameters of the calculator are saved after training so that the calculator can be re-established for future calculations. If the previously trained parameter file is named 'old.json', it can be introduced to Amp to take those values with something like:

```
>>> calc = Amp(load='old.json', label='new')
```

The label ('new') is used as a prefix for any output from use of this calculator. In general, the calculator is written to not overwrite your old settings, but it is a good idea to have a different name for this label to avoid accidentally overwriting your carefully trained calculator's files!

Training Amp

Training a new calculator instance occurs with the train method, which is given a list of train images and desired values of root mean square error (rmse) for forces and atomic energy as below:

```
>>> calc.train(images, energy_goal=0.001, force_goal=0.005)
```

Calling this method will generate output files where you can watch the progress. Note that this is in general a computationally-intensive process! These two parameters as well as other optional parameters are described at.

In general, we plan to make the ASE database the recommended data type. However, this is a work in progress over at ASE.

Growing your train set

Say you have a nicely trained calculator that you built around a train set of 10,000 images, but you decide you want to add another 1,000 images to the train set. Rather than starting training from scratch, it can be faster to train from the previous optimum. This is fairly simple and can be accomplished as:

```
>>> calc = Amp(load='old_calc', label='new_calc')
>>> calc.train(all_images)
```

In this case, 'all_images' contains all 11,000 images (the old set **and** the new set).

If you are training on a large set of images, a building-up strategy can be effective. For example, to train on 100,000 images, you could first train on 10,000 images, then add 10,000 images to the set and re-train from the previous parameters, etc. If the images are nicely randomized, this can give you some confidence that you are not training inside too shallow of a basin. The first images set, which you are starting from, had better be representative of the verity of all images you will add later.

Using trained Amp

The trained calculator can now be used just like any other ASE calculator, e.g.:

```
>>> atoms = ...
>>> atoms.set_calculator(calc)
>>> energy = atoms.get_potential_energy()
```

where 'atoms' is an atomic configuration not necessarily included in the train set.

Complete example

The script below shows an example of training Amp. This script first creates some sample data using a molecular dynamics simulation with the cheap EMT calculator in ASE. The images are then randomly divided into "train" and "test" data, such that cross-validation can be performed. Then Amp is trained on the train data. After this, the predictions of the Amp are compared to the parent data for both the train and test set. This is shown in a parity plot, which is saved to disk.

```
from ase.lattice.surface import fcc110
from ase import Atom, Atoms
from ase.constraints import FixAtoms
from ase.calculators.emt import EMT
```

```
from ase.md import VelocityVerlet
from ase.md.velocitydistribution import MaxwellBoltzmannDistribution
from ase import units
from ase import io

from amp.utilities import randomize_images
from amp import Amp
from amp.descriptor import *
from amp.regression import *

#######################################################################


def test():

    # Generate atomic system to create test data.
    atoms = fcc110('Cu', (2, 2, 2), vacuum=7.)
    adsorbate = Atoms([Atom('H', atoms[7].position + (0., 0., 2.)),
                       Atom('H', atoms[7].position + (0., 0., 5.))])
    atoms.extend(adsorbate)
    atoms.set_constraint(FixAtoms(indices=[0, 2]))
    calc = EMT()  # cheap calculator
    atoms.set_calculator(calc)

    # Run some molecular dynamics to generate data.
    trajectory = io.Trajectory('data.traj', 'w', atoms=atoms)
    MaxwellBoltzmannDistribution(atoms, temp=300. * units.kB)
    dynamics = VelocityVerlet(atoms, dt=1. * units.fs)
    dynamics.attach(trajectory)
    for step in range(50):
        dynamics.run(5)
    trajectory.close()

    # Train the calculator.
    train_images, test_images = randomize_images('data.traj')

    calc = Amp(descriptor=Behler(),
               regression=NeuralNetwork())
    calc.train(train_images, energy_goal=0.001, force_goal=None)

    # Plot and test the predictions.
    import matplotlib
    matplotlib.use('Agg')
    from matplotlib import pyplot

    fig, ax = pyplot.subplots()

    for image in train_images:
        actual_energy = image.get_potential_energy()
        predicted_energy = calc.get_potential_energy(image)
        ax.plot(actual_energy, predicted_energy, 'b.')

    for image in test_images:
        actual_energy = image.get_potential_energy()
        predicted_energy = calc.get_potential_energy(image)
        ax.plot(actual_energy, predicted_energy, 'r.')

    ax.set_xlabel('Actual energy, eV')
```

```
    ax.set_ylabel('Amp energy, eV')

    fig.savefig('parityplot.png')

##############################################################################

if __name__ == '__main__':
    test()
```

Note that most of the script is just creating the train set or analyzing the results – the actual Amp initialization and training takes place on just two lines in the middle of the script. The figure that was produced is shown below. As both the generated data and the initial guess of the training parameters are random, this will look different each time this script is run.

CHAPTER 4

Theory

According to Born-Oppenheimer approximation, the potential energy of an atomic configuration can be assumed as a function of only nuclei positions. Potential energy is in general a very complicated function that in theory can be found by directly solving the Schrodinger equation. However, in practice, exact analytical solution to the many-body Schrodinger equation is very difficult (if not impossible). Taking into account this fact, the idea is then to approximate the potential energy with a regression model:

$$\mathbf{R} \xrightarrow{\text{regression}} E = E(\mathbf{R}),$$

where $\mathbf{R}$ is the position of atoms in the system.

Atomic representation of potential energy

In order to have a potential function which is simultaneously applicable to systems of different sizes, the total potential energy of the system can to be broken up into atomic energy contributions:

$$E(\mathbf{R}) = \sum_{\text{atom}=1}^{N} E_{\text{atom}}(\mathbf{R}).$$

The above expansion can be justified by assembling the atomic configuration by bringing atoms close to each other one by one. Then the atomic energy contributions can be approximated using a regression method:

$$\mathbf{R} \xrightarrow{\text{regression}} E_{\text{atom}} = E_{\text{atom}}\left(\mathbf{R}\right).$$

Descriptor

A better interpolation can be achieved if an appropriate symmetry function $\mathbf{G}$ of atomic coordinates, approximating the functional dependence of local energetics, is used as the input of regression operator:

$$\mathbf{R} \xrightarrow{\mathbf{G}} \mathbf{G}\left(\mathbf{R}\right) \xrightarrow{\text{regression}} E_{\text{atom}} = E_{\text{atom}}\left(\mathbf{G}\left(\mathbf{R}\right)\right).$$

Gaussian

Gaussian descriptor $\mathbf{G}$ as a function of pair-atom distances and three-atom angles, has been suggested by Behler [1], and is implemented within Amp. Radial fingerprint of Gaussian type captures interaction of atom i with all atoms j as

the sum of Gaussians with width η and center R_s,

$$G_i^I = \sum_{\substack{\text{atoms j within } R_c \\ \text{distance of atom i} \\ j \neq i}} e^{-\eta(R_{ij}-R_s)^2/R_c^2} f_c\left(R_{ij}\right).$$

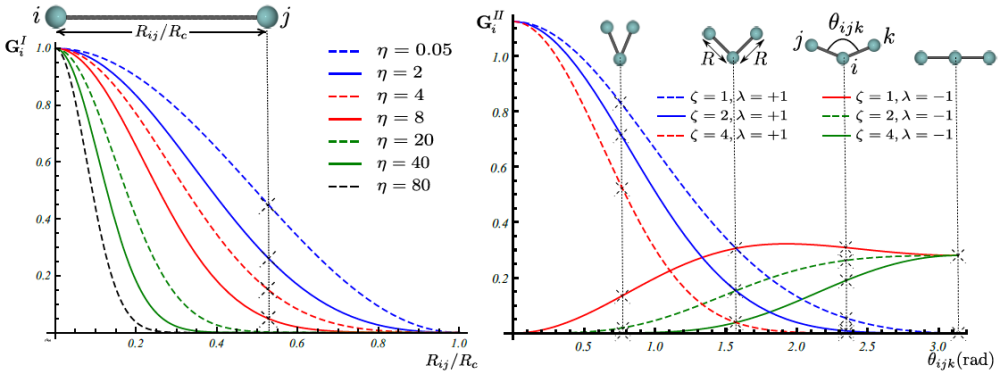
The next type is the angular fingerprint accounting for three-atom interactions. Gaussian angular fingerprint is computed for all triplets of atoms i, j, and k by summing over the cosine values of the angles $\theta_{ijk} = \cos^{-1}\left(\frac{\mathbf{R}_{ij}.\mathbf{R}_{ik}}{R_{ij}R_{ik}}\right)$, ($\mathbf{R}_{ij} = \mathbf{R}_i - \mathbf{R}_j$), centered at atom i, according to

$$G_i^{II} = 2^{1-\zeta} \sum_{\substack{\text{atoms j, k within } R_c \\ \text{distance of atom i} \\ j,\, k \neq i \\ (j \neq k)}} \left(1 + \lambda \cos\theta_{ijk}\right)^{\zeta} e^{-\eta\left(R_{ij}^2 + R_{ik}^2 + R_{jk}^2\right)/R_c^2} f_c\left(R_{ij}\right) f_c\left(R_{ik}\right) f_c\left(R_{jk}\right),$$

with parameters λ, η, and ζ. The function $f_c\left(R_{ij}\right)$ in the above equations is the cutoff function defining the energetically relevant local environment with value one at $R_{ij} = 0$ and zero at $R_{ij} = R_c$, where R_c is the cutoff radius. In order to have a continuous force-field, the cutoff function $f_c\left(R_{ij}\right)$ as well as its first derivative should be continuous in $R_{ij} \in [0, \infty)$. One possible expression for such a function as proposed by Behler [1] is

$$f_c\left(R_{ij}\right) == \begin{cases} 0.5\left(1 + \cos\left(\pi \dfrac{R_{ij}}{R_c}\right)\right) & \text{for } R_{ij} \leq R_c, \\ 0 & \text{for } R_{ij} > R_c. \end{cases}$$

Figure below shows how components of fingerprints $\mathbf{G}_i^I$ and $\mathbf{G}_i^{II}$ change with, respectively, distance R_{ij} between pair atoms i and j and valence angle θ_{ijk} between triplet of atoms i, j, and k with central atom i:



Zernike

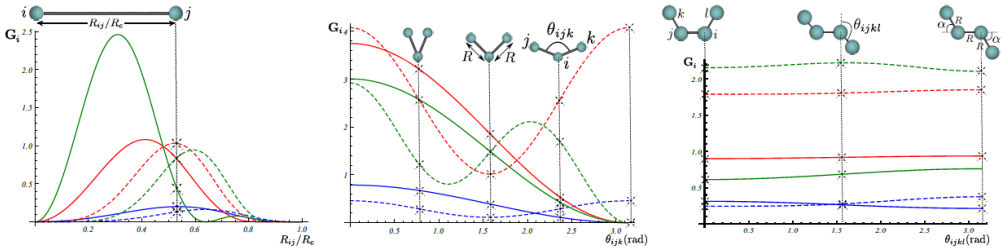
Three-dimensional Zernike descriptor is also available inside Amp, and can be used as the atomic environment descriptor. Zernike-type descriptor has been previously used in the machine-learning community extensively, but it has been suggested here as the first time for representing chemical local environment. Zernike moments are basically a tensor product between spherical harmonics complete and orthogonal on the surface of unit sphere, and Zernike polynomials complete and orthogonal within the unit sphere. Zernike descriptor components for each integer degree are then defined as the norm of Zernike moments with the same corresponding degree. For more details on the Zernike descriptor the reader is referred to the nice paper of Novotni and Klein [2].

Inspired by Bartok et. al. [3], to represent the local chemical environment of atom i, an atomic density function $\rho_i(\mathbf{r})$ is defined for each atomic local environment as the sum of delta distributions shifted to atomic positions:

$$\rho_i(\mathbf{r}) = \sum_{\substack{\text{atoms j within } R_c \\ \text{distance of atom i} \\ j \neq i}} \eta_j \delta\left(\mathbf{r} - \mathbf{R}_{ij}\right) f_c\left(\|\mathbf{R}_{ij}\|\right),$$

Next components of Zernike descriptor are computed from Zernike moments of the above atomic density destribution for each atom i.

Figure below show how components of Zernike descriptor vary with pair-atom distance, three-atom angle, and four-atom dehidral angle. It is important to note that components of the Gaussian descriptor discussed above are non-sensitive to the four-atom dehidral angle of the following figure.



(a) versus pair-atom distance R_{ij}. **(b)** versus valence angle θ_{ijk} for $R = R_c/2$. **(c)** versus dihedral angle θ_{ijkl} for $R = R_c/2$ and $\alpha = \pi/3$.

Bispectrum

Bispectrum of four-dimensional spherical harmonics have been suggested by Bartok et al. [3] to be invariant under rotation of local atomic environment. In this approach, the atomic density distribution defined above is first mapped onto the surface of unit sphere in four dimensions. Consequently, Bartok et al. have shown that the bispectrum of this mapping can be used as atomic environment descriptor. We refer the reader to the original paper [3] for mathematical details. Worth to mention that this approach of describing local environment is also available inside Amp.

Regression Model

The general purpose of the regression model $x \xrightarrow{\text{regression}} y$ with input x and output y is to approximate the function $y = f(x)$ by using sample train data points (x_i, y_i). The intent is to later use the approximated f for input data x_j (other than x_i in the train data set), and make predictions for y_j. Typical regression models include, but are not limited to, Gaussian processes, support vector regression, and neural network.

Neural network model

Neural network is basically a very simple model of how the nervous system processes information. The first mathematical model was developed in 1943 by McCulloch and Pitts [4] for classification purposes; biological neurons either send or do not send a signal to the neighboring neuron. The model was soon extended to do linear and nonlinear regression, by replacing the binary activation function with a continuous function. The basic functional unit of a neural network is called “node”. A number of parallel nodes constitute a layer. A feed-forward neural network consists of at least an input layer plus an output layer. When approximating the PES, the output layer has just one neuron representing the potential energy. For a more robust interpolation, a number of “hidden layers” may exist in the neural network as well; the word “hidden” refers to the fact that these layers have no physical meaning. A schematic of a typical feed-forward neural network is shown below. In each node a number of inputs is multiplied by the corresponding weights and summed up with a constant bias. An activation function then acts upon the summation and an output is generated. The output is finally sent to the neighboring neuron in the next layer. Typically used activation functions are hyperbolic tangent, sigmoid, Gaussian, and linear function. The unbounded linear activation function is particularly useful in the last hidden layer to scale neural network outputs to the range of reference values. For our purpose, the output of neural network represents energy of atomic system.

References:

1. "Atom-centered symmetry functions for constructing high-dimensional neural network potentials", J. Behler, J. Chem. Phys. 134(7), 074106 (2011)
2. "Shape retrieval using 3D Zernike descriptors", M. Novotni and R. Klein, Computer-Aided Design 36(11), 1047–1062 (2004)
3. "Gaussian approximation potentials: The accuracy of quantum mechanics, without the electrons", A.P. Bart'ok, M.C. Payne, R. Kondor and G. Csanyi, Physical Review Letters 104, 136403 (2010)
4. "A logical calculus of the ideas immanent in nervous activity", W.S. McCulloch, and W.H. Pitts, Bull. Math. Biophys. 5, 115–133 (1943)

CHAPTER 5

Parallel Computing

As the number of atoms corresponding to data points (images) or the number of data points themselves in the train data set increases, serial computing takes a long time, and thus, resorting to parallel computing becomes inevitable. Two common approaches of parallel computing are either to parallelize over multiple threads or to parallelize over multiple processes. In the former approach, threads usually have access to the same memory area, which can lead to conflicts in case of improper synchronization. But in the second approach, separate memory locations are allocated to each spawned process which avoids any possible interference between sub-tasks. Amp takes the second approach via "multiprocessing", a built-in module in Python 2.6 (and above) standard library. For details about multiprocessing module, the reader is directed to multiprocessing documentation. Only local concurrency (multi-computing over local cores) is possible in the current release of Amp. Amp starts with serial computing, but when arriving to calculating atomic fingerprints and their derivatives, multiple processes are spawned each doing a sub-task on part of the atomic configuration, and writing the results in temporary JSON files. Temporary files are next unified and saved in the script directory. If Fortran modules are utilized, data of all images including real energies and forces, atomic identities, and calculated values of fingerprints (as well as their derivatives and neighbor lists if force-training) are then reshaped to rectangular arrays. Portions of the reshaped data corresponding to each process are sent to the copy of Fortran modules at each core. Partial cost function and its derivative with respect to variables are calculated on each core, and the calculated values are returned back to the main python code via multiprocessing "Queues". The main python code then adds up partial values. Amp automatically finds the number of requested cores, prints it out at the top of "train-log.txt" script, and makes use of all of them. A schematic of how parallel computing is designed in Amp is shown in the below figure. When descriptor is None, no fingerprint operation is carried out.

CHAPTER 6

Development

This page contains standard practices for developing Amp, focusing on repositories and documentation.

Repositories and branching

The main Amp repository lives on bitbucket, andrewpeterson/amp. We employ a branching model where the *master* branch is the main development branch, containing day-to-day commits from the core developers and honoring merge requests from others From time to time, we create a new branch that corresponds to a release. This release branch contains only the tagged release and any bug fixes.

Contributing

You are welcome to contribute new features, bug fixes, better documentation, etc. to Amp. If you would like to contribute, please create a private fork and a branch for your new commits. When it is ready, send us a merge request. We follow the same basic model as ASE; please see the ASE documentation for complete instructions.

It is also a good idea to send us an email if you are planning something complicated.

Documentation

This documentation is built with sphinx. On your own computer, you can build a copy with a command like:

```
$ sphinx-build . /tmp/ampdocs/
```

List of All Methods:

CHAPTER 7

Main

This module is the main part of Amp package.

Module contents

CHAPTER 8

Descriptor

Methods for describing local atomic environment are included in this module.

Module contents

CHAPTER 9

Regression

This module includes methods for interpolating energies and forces with local atomic environment.

Module contents

CHAPTER 10

Utilities

This module contains utilities for use with various aspects of the Amp calculator.

Module contents

CHAPTER 11

Analysis

Tools for analysis of output exist here.

Module contents

CHAPTER 12

Indices and tables

- genindex
- modindex
- search