(AMIBA)

Arbitrary Multidimensional Interpolation | and Benchmark Automation | Toolkit

Introduction and User Guide

Revision date: March 21, 2014
For software version: 0.1

Questions or Issues? Contact:

Dylan Rudolph

rudolph@chrec.org

Contents

1 Installation

Presently, all of the code is kept in the native Python `.py` form, and no compilation has been done. As a result, there is no installation process beyond extracting the archive containing the files. However, if you would like to test out the scripts for the sample benchmarks, make sure to re-build the sample benchmarks for your platform.

1.1 Requisite Software Packages

The minimum requirements are that the following are installed:

- A version of Python from the 2.7 branch.
- Numerical Python (numpy)
- Scientific Python (scipy)

These are among the most popular of all of the Python packages, so there should be no issue installing them on any common system configuration. Additionally, if performing benchmarks on a remote system by `SSH`, the following are requisite:

- An `SSH` implementation (*e.g.*, openssh).
- A means of doing password-less `SSH` login (*e.g.*, sshpass).

The provided examples for `SSH`-based remote benchmarking assume the above software, which limits them to use with the *nixes, but it is likely that non-*nix support for remote benchmarking is possible with different programs. In general, the benchmarking examples are made for *nix command-line calls, but the interpolation examples should be completely cross-platform.

1.2 Verifying the Software Configuration

The best way to verify the software configuration is to simply run one of the example scripts. The script that should work "out of the box" on all platforms is `example-a-p2.py`, and the others should run after the benchmarks are compiled for your platform. Nominally, `example-a-p2.py` will print out a few lines about estimations and variances at several points.

2 Features

This software does three things:

1. Interpolate the computation-times of arbitrary multi-dimensional functions
2. Automate the gathering of benchmark data

3. Automate the training of an interpolator

Each of these things has several modes of operation, which will be outlined below, and explained in detail in their respective sections.

2.1 Interpolation Techniques

In order of increasing complexity and, in general, accuracy, the following interpolation techniques are available:

- Nearest-neighbor
- Linear
- Radial basis functions
- Kriging

All of the interpolation techniques are available for any number of dimensions, with the exception of one-dimensional data, which is must be cast to two-dimensional data to be used. An example of how to do this is given in `example-c.py` for a dot-product benchmark.

2.2 Benchmark-Gathering Automation

The benchmark-gathering portion of this software works on batches of arguments presented to a single command-line call. For example, it would be possible automatically collect the output of

```
→ ./matrix_multiply 8 8
→ ./matrix_multiply 8 16
  ...
→ ./matrix_multiply 256 240
→ ./matrix_multiply 256 256
```

by specifying the `command` (`./matrix_multiply`), the `arguments` (`[[8, 8], [8, 16], ... [256, 240], [256, 256]]`), and some format specifiers concerning how to collect and parse the output. Also, there are a number of helper functions for generating these arguments for several common use-cases (*e.g.*, evenly-spaced grids).

2.3 Interpolator Training

By combining the interpolation and benchmarking software, iterative data gathering to improve the interpolators is possible. This is the least-developed feature, capable of only one mode, which is covered later.

3 Usage: Interpolation

Three of the interpolators, nearest-neighbor, linear, and radial basis functions, are very similar in structure. In fact, they are little more than wrappers for the SciPy interpolators of the same names. The Kriging interpolator, however, is home-grown, and requires a few more parameters than the other three.

The general structure of code to call any of the interpolators is to the effect of:

```
1. perform imports
2. create instance of benchmark and data classes
3. load points and values from a .bmark file
4. create an instance of, and initialize, the interpolator
5. perform interpolation at will
```

A simple example which does the above is included in `example-a-p2.py`. The only thing that should differ among the interpolators is the fourth step, so it should be a trivial modification to swap one interpolator for another.

For the non-Kriging interpolators, little to no configuration is required, but the relevant SciPy documentation explains the few parameters which are selectable. The Kriging interpolator requires three setup parameters:

- **The variogram:** a function which describes the degree of spatial correlation among the interpolation samples. It is specified by providing an instance of the Kriging class with a dictionary containing the parameters. Currently only exponential variograms are supported.
- **The underlying polynomial:** a number which describes the maximum degree of the polynomial which underlies the data. Currently only `0` and `1` are supported.
- **The number of neighbors to use:** the number of neighbors to use in the interpolation must be carefully selected. Too many neighbors can cause inaccuracy and slow interpolation. **Too few can result in the program to fail because it cannot resolve an estimation at a point.** This will manifest in an error to the effect of "singular matrix". Additionally, the higher the degree of the polynomial, the more neighbors are required.

There is also a difference in the return values of the different interpolators. All interpolators return two lists, and in every case the first list is the estimates (which is of the same length as the number of points that the interpolator was told to use). However, the second list returned is only meaningful for the Kriging interpolator. This second list contains the estimation variances at those points, which can be used to gauge uncertainty.

4 Usage: Benchmarking

Benchmarking is done by specifying:

- **The base command:** A string which specifies the portion of the call which does not change between iterations.
- **The arguments:** A list of list of numbers which specifies the changing parameters between iterations. Notably: only integer numbers are permitted.
- **The parsing parameters:** Each command line call is parsed by searching for a number between two strings. These two strings must be specified, along with a text name for the thing to be parsed.
- **Some Optional Parameters:** It is also possible to pass a few optional parameters, most of which are simply meta-data and do not affect the benchmarking process.

There are two classes of functions in the benchmarking class – those related to command line calls and those related to file management. The functions of the first kind include `call`, `batch`, and `single`. The function `call` is used for one-time command line calls, `batch` is used for running a number of argument-based calls, and `single` is used after a batch to append to the existing data. The functions of the second kind include `save_bmark`, `save_csv`, `load_bmark`, and `samples_from_bmark` – all of which are relatively self-explanatory and documented in the code.

5 Usage: Training

Training is an iterative combination of benchmarking and interpolation. Two examples are provided which demonstrate the training process, in the source files `example-a.py` and `examples-b.py`. Currently there is only one mode of training, called "budgeted random worst cost". This strategy is to the effect of "you have a set number of points you can add, the *budget*; choose these points by generating a random set of points and selecting the one with the highest cost, until you've expired your budget.". This strategy has two parameters:

- **The budget:** The number of points that can be added.
- **The evaluation points:** The number of random points to generate and select from.

The training process is composed of three steps: the setup, in which parameters are specified, the initialization, in which the initial points are gathered, and the training itself. See the source code for details of how these are done. Notably, the training class accepts instances of four other classes uses their underlying storage methods, but does not store more than a trivial amount of information itself.

