README.md

Mini project for course in python calculating Mandelbrot fractals

build passing

Credit

- `mandelbrot` package is the work of Egon Kidmose.
- Originally forked from Ian Oszwald's python_template_with_config for use of `setuputils`, module structure etc.

Hand-in reading instructions

For the written part of the hand-in please refer to this document and comments in code, which are formatted for pydoc (After installing as described below: `python -c "import mandelbrot; help(mandelbrot)`).

Setup

Environment

Developed, tested and evaluated on a mix of Linux Mint 17.1 (Oracle VirtualBox), Linux Mint 17.2 (Oracle VirtualBox) and Ubuntu Server 12.04.5 LTS.

Requirements

- Python 2.7, 3.3 or 3.4
- cython
- Virtual environment (Optional but assumed in the given instructions)

```
sudo apt-get install -qq python-virtualenv
```

- Internet connection for installing components
- If matplotlib is not already installed, These build dependencies:

```
sudo apt-get install -qq build-essential python-dev python3-dev libpng12-dev libjpe
```

Expected to be OS independent, but as stated above I have developed and tested on Linux only.

Test

While the test strategy is discussed further below, unit tests are run on pushes on Travis CI and can be run locally with:

```
$ py.test mandelbrot
```

Installation instructions

```
$ virtualenv -p python3 env  # Create virtual env for isolation
$ source env/bin/activate  # Enter virtual environment
$ pip install cython # setup relies on python
$ python setup.py develop  # "develop" for symlinks to reflect code changes

# Alternatively, for an installation that will not be edited, without a virtual environ
$ sudo apt-get install -qq cython
$ sudo python setup.py install
```

Project structure

The project, contained in this folder contains the following import sub-folders:

- `mandelbrot` - The python module.
- `output` - Output of the algorithms; plots and data.
- `profiling` - Output of profiling tools.
- `setup.py` - The script for installing the `mandelbrot` module.

Methodologies

This section contains an outline of important methodologies used in this work.

As the very first choice it was decided to use the **Object Oriented Programming** (OOP) features of python. This was due to the similarity between the three task of implementing the same algorithm in different styles. Using OOP, a framework for doing the calculation was implemented, which is primarily represented by the `mandelbrot.MandelBrotCalculator` abstract base class which is inherited by the three implementations. This allows for all code except the calculation implementation to be shared by all three, adhering to the DRY principle (Don't Repeat Yourself). As for testing code, this also meant that the class interface was settled and tests for the second and third implementation required almost no additional implementation.

Test Driven Development (TDD) has been applied. In this project specifically the development process was formed by reiterating the following steps:

1. Decide on most important missing feature.
2. Implement a test for it and add it to the test suite.
3. Verify that it fails.

4. Implement feature to make test succeed.

Unit testing is highly coupled with the TDD approach, and implemented in this project using the python module `unittest` . A sub-module named `mandelbrot.tests` contains all tests and constants, allowing heavy reuse of test case as mentioned earlier. For a final distribution is would be relevant to move it out of the `mandelbrot` package.

Testing the output values of the given algorithm requires defining the expected ones. While it might be possible to define expectancy it is anticipated to be difficult, and it is expected that verifying the implementation might require solving the problem, leading to circular requirements. Instead it was assumed that the plot provided in the mini project description was correct and that I was able to compare it adequately with the output of the naive solution. Confident that the naive implementation was correct a collection of small output was calculated and stored in the `mandelbrot.tests.test_naive.Test` class. This is expected to catch a large majority of regressions in the naive implementation and allows testing on the same point of input space for the other implementation. During the development process these tests caught bugs and the result in the `output` folder is equivalent for all three implementation.

Git version control and the related service github have been used for the availability, ease of publishing and nice tools. Rather than implementing a full gitflow work flow the `master` branch has been used for stable code and `development` branch for unstable development. Implementing the full gitflow work flow is deemed overkill for a small, single contributor project. **Travis CI** is used for automated testing of every push, catching among other things incomplete commits, local remnants from previous builds and undiscovered dependencies incidentally fulfilled on my machine.

Documentation is written in markdown for this file as it integrates well with github while also readable in as plain text. For the comments in the python code the supported reST language is used.

Unimplemented ideas

The following is a list of ideas that also could have been interesting to apply:

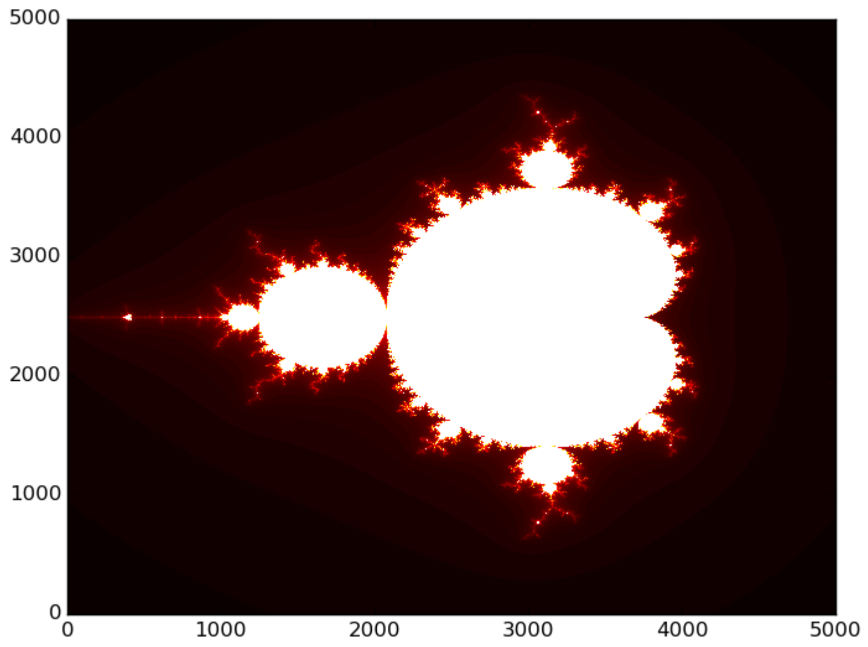
- Evaluate test coverage.
- Use logging facilities rather than direct writes to stdout.
- PEP8/pyflake evaluation for improved code style.

Results

Logs and transcripts

The raw outputs are available in the following places:

The `output` folder contains the plots produced by the algorithm in its various implementation along with the compressed result (As expected, These are identical across implementations).



`output/cli.log` contains logs for running the algorithms on the AAU job server. While this might be a noisy environment, the alternative was a virtual machine with too little resources to run the naive problem on a reasonably sized problem (Encountered exceptions due to memory not being available). It is assumed that the low utilisation across all cores means the interference from other task are limited, providing for roughly deterministic results. In the docstring of `mandelbrot.optimised.OptimisedCalculator` (`mandelbrot/optimised.pyx`) some step wise results are also included.

`profiling` contains the output of the profiles.

Discussion

Optimised implementation (Cython)

Adding cython to the naive implementation without any declarations we see roughly a speed-up of 200% (from 164ms to 81ms for 100x100 points), which is good compared to the price paid (Hardly any implementation to be done. Code reads like plain python except the file extension and a few lines in `setup.py` . The dependency on cython.).

Declaring all variables used in the loop provides approximately another 200% speed-up (from 81ms to 36 ms).

Finally rewriting to avoid a call to pythons built-in `abs()` in the innermost loop provides a little less speed up (from 36ms to 20ms).

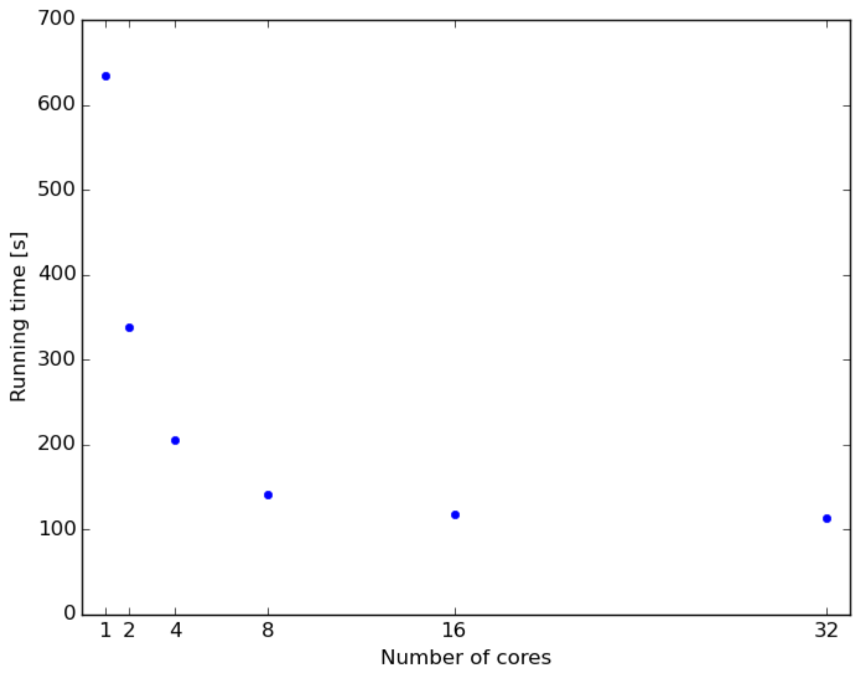
Overall, it seems suitable to use cython and declare variables to make the calculation four times as fast. The rewriting seems to provide a limited benefit compared to the impact on maintainability. All in all it depends on how badly the speedup is needed.

Vectorising and using numpy is expected to even more speed-up.

Parallel

The parallel implementation has been evaluated on the AAU job server at js3.es.aau.dk to allow usage of many cores. While the original intent was to make minimal changes to the naive implementation in order to parallelise it, the solution ended up including use of numpy arrays, due to the way multiprocessing works. It can therfore not be compared to the naive implementation.

From the figure below we see that adding one or three processes (assumed almost the same as a core) roughly provides a linear speed-up. For eight processes the gain is limited and from 16 to 32 we have hardly any improvement. We conclude that the linear component of the current implementation is roughly a sixth of the total.


Running time [s]
Number of cores