automan Documentation

Release 0.5.dev0

Prabhu Ramachandran

Jun 09, 2019

Contents:

CHAPTER 1

Overview

`automan` is an open source, Python-based automation framework for numerical computing.

It is designed to automate the drudge work of managing many numerical simulations. As an automation framework it does the following:

- helps you organize your simulations.
- helps you orchestrate running simulations and then post-processing the results from these.
- helps you reuse code for the post processing of your simulation data.
- execute all your simulations and post-processing with one command.
- optionally distribute your simulations among other computers on your network.

This greatly facilitates reproducibility. Automan is written in pure Python and is easy to install.

This document should help you use automan to improve your productivity. If you are interested in a more detailed article about automan see the automan paper a draft of which is available here: https://arxiv.org/abs/1712.04786

1.1 Installation

The easiest way to install `automan` is with pip:

```
$ pip install automan
```

If you wish to run the latest version that has not been relesed you may clone the git repository:

```
$ git clone https://github.com/pypr/automan

$ cd automan
```

And then run:

```
$ python setup.py develop
```

If you just want to run the latest version and do not have `git` you can do this:

```
$ pip install https://github.com/pypr/automan/zipball/master
```

Once this is done, move on to the next section that provides a gentle tutorial introduction to using automan.

1.2 Citing `automan`

If you find automan useful and wish to cite it you may use the following article:

- Prabhu Ramachandran, "automan: A Python-Based Automation Framework for Numerical Computing," in *Computing in Science & Engineering*, vol. 20, no. 5, pp. 81-97, 2018. doi:10.1109/MCSE.2018.05329818

You can find a draft of the article here: https://arxiv.org/abs/1712.04786

1.3 Changelog

1.3.1 0.5

- Release date: still in development.

1.3.2 0.4

- Release date: 26th November, 2018.
- Support for inter Problem/Simulation/Task dependencies.
- Print more useful messages when running tasks.
- Fix bug with computing the available cores.
- Improve handling of cases when tasks fail with errors.
- Fix a few subtle bugs in the task runner.
- Minor bug fixes.

1.3.3 0.3

- Release date: 5th September, 2018.
- Complete online documentation and examples.
- Much improved and generalized cluster management with support for conda and edm in addition to virtualenvs.
- Support multiple projects that use different bootstrap scripts.
- Better testing for the cluster management.
- Do not rewrite the path to python executables and run them as requested by the user.
- Removed any lingering references or use of `pysph`.
- Change the default root to `automan` instead of `pysph_auto`.

- Support filtering cases with a callable.
- Fix bug where a simulation with an error would always be re-run.
- Fix bug caused due to missing `--nfs` option to automator CLI.

1.3.4 0.2

- Release date: 28th August, 2017.
- First public release of complete working package with features described in the paper.

CHAPTER 2

A tutorial on using `automan`

Automan is best suited for numerical computations that take a lot of time to run. It is most useful when you have to manage the execution of many of these numerical computations. Very often one needs to process the output of these simulations to either compare the results and then assemble a variety of plots or tables for a report or manuscript.

The numerical simulations can be in the form of any executable code, either in the form of a binary or a script. In this tutorial we will assume that the programs being executed are in the form of Python scripts.

There are three general recommendations we make for your numerical programs to be able to use automan effectively.

1. They should be configurable using command line arguments.
2. They should generate their output files into a directory specified on the command line.
3. Any post-processed data should be saved into an easy to load datafile to facilitate comparisons with other simulations.

None of these is strictly mandatory but we strongly recommend doing it this way.

2.1 A simple example

In this tutorial we take a rather trivial program to execute just to illustrate the basic concepts. Our basic program is going to simply calculate the square of a number passed to it on the command line. Here is how the code looks:

```
from __future__ import print_function
import sys
x = float(sys.argv[1])
print(x, x*x)
```

You can execute this script like so:

```
$ python square.py 2.0
2.0 4.0
```

Yay, it works!

Note: If you want to run these examples, they are included along with the example files in the automan source code. The files should be in `examples/tutorial` (Browse online).

This example does not produce any output files and doesn't really take any configuration arguments. So we don't really need to do much about this.

Now let us say we want to automate the execution of this script with many different arguments, let us say 2 different values for now (we can increase it later).

We can now do this with automan. First create a simple script for this, we could call it `automate1.py` (this is just a convention, you could call it anything you want). The code is as shown below:

```
from automan.api import Problem, Automator


class Squares(Problem):
    def get_name(self):
        return 'squares'

    def get_commands(self):
        return [
            ('1', 'python square.py 1', None),
            ('2', 'python square.py 2', None),
        ]

    def run(self):
        self.make_output_dir()


automator = Automator(
    simulation_dir='outputs',
    output_dir='manuscript/figures',
    all_problems=[Squares]
)
automator.run()
```

Note the following:

- The `Squares` class derives from *`automan.automation.Problem`*. This encapsulates all the simulations where we wish to find the square of some number.
- The `get_name` method returns a subdirectory for all the outputs of this problem.
- The `get_commands` returns a list of tuples of the following form, `(directory_name, command, job_info)`. In this case we don't pass any job information and we'll get to this later. Notice that the two commands specified are essentially what we'd have typed on the terminal.
- The `run` command does nothing much except create a directory. For now let us leave this alone.

Let us execute this to see what it does:

```
$ python automate1.py

Writing config.json
4 tasks pending and 0 tasks running
```

(continues on next page)

(continued from previous page)

```
Running task CommandTask with output directory: 2 ...
Starting worker on localhost.
Job run by localhost
Running python square.py 2

Running task CommandTask with output directory: 1 ...
Job run by localhost
Running python square.py 1
2 tasks pending and 2 tasks running
Running task Problem named squares...

Running task <automan.automation.RunAll object at 0x10628d908>...
Finished!
```

So the script executes and seems to have run the requested computations. Let us see the output directories:

```
$ tree
.
├── automate1.py
├── config.json
├── manuscript
│   └── figures
│       └── squares
├── outputs
│   └── squares
│       ├── 1
│       │   ├── job_info.json
│       │   ├── stderr.txt
│       │   └── stdout.txt
│       └── 2
│           ├── job_info.json
│           ├── stderr.txt
│           └── stdout.txt
└── square.py

7 directories, 9 files
```

Let us summarize what just happened:

- The two commands we asked for were executed and the respective outputs of each invocation were placed into `outputs/squares/1` and `outputs/squares/2`. Notice that there are `stdout.txt, stderr.txt` and a `job_info.json` file here too.
- A manuscript directory called `manuscript/figures/squares` is created.
- There is also a new `config.json` that we can safely ignore for now.

Let us see the contents of the files in the outputs directory:

```
$ cat outputs/squares/1/stdout.txt
1.0 1.0

$ cat outputs/squares/1/job_info.json
{"start": "Fri Aug 24 01:11:46 2018", "end": "Fri Aug 24 01:11:46 2018", "status":
↪"done", "exitcode": 0, "pid": 20381}
```

As you can see, the standard output has the output of the command. The `job_info.json` has information about the actual execution of the code. This is very useful in general.

Thus automan has executed the code, organized the output directories and collected the standard output and information about the execution of the tasks.

Now, let us run the automation again:

```
$ python automate1.py
0 tasks pending and 0 tasks running
Finished!
```

It does not re-run the code as it detects that everything is complete.

2.2 Adding some post-processing

Let us say we want to create a plot that either compares the individual runs or assembles the runs into a single plot or collects the data into a single file. We can easily do this by adding more code into the `run` method of our `Squares` class. Let us also add a couple of more computations.

```
from automan.api import Problem, Automator


class Squares(Problem):
    def get_name(self):
        return 'squares'

    def get_commands(self):
        return [
            ('1', 'python square.py 1', None),
            ('2', 'python square.py 2', None),
            ('3', 'python square.py 3', None),
            ('4', 'python square.py 4', None),
        ]

    def run(self):
        self.make_output_dir()
        data = []
        for i in ('1', '2', '3', '4'):
            stdout = self.input_path(i, 'stdout.txt')
            with open(stdout) as f:
                data.append(f.read().split())

        output = self.output_path('output.txt')
        with open(output, 'w') as o:
            o.write(str(data))


if __name__ == '__main__':
    automator = Automator(
        simulation_dir='outputs',
        output_dir='manuscript/figures',
        all_problems=[Squares]
    )

    automator.run()
```

Let us examine this code a little carefully:

- In `get_commands`, we have simply added two more cases.

- In `run`, we have added some simple code to just iterate over the 4 directories we should have and then read the standard output into a list and finally we write that out into an `output.txt` file.
- We also moved the `automator` object creation and execution so we can import our `automate2.py` script if we wish to.

The two new methods you see here are `self.input_path(...)` which makes it easy to access any paths inside the simulation directories and the `self.output_path(...)` which does the same but inside the output path. Let us see what these do, inside the directory containing the `automate2.py` if you start up a Python interpreter (IPython would be much nicer), you can do the following:

```
>>> import automate2
>>> squares = automate2.Squares(
...     simulation_dir='outputs',
...     output_dir='manuscript/figures/'
... )

>>> squares.input_path('1')
'outputs/squares/1'

>>> squares.input_path('1', 'stdout.txt')
'outputs/squares/1/stdout.txt'

>>> squares.output_path('output.txt')
'manuscript/figures/squares/output.txt'
```

As you can see, these are just conveniences for finding the input file paths and the output file paths. Now let us run this new script:

```
$ python automate2.py
0 tasks pending and 0 tasks running
Finished!
```

Whoa! That doesn't seem right? What happens is that since the last time we ran the automate script, it created the output, it assumes that there is nothing more to do as the final result (the `manuscript/figures/squares`) was successfully created so it does not run anything new. If you do this:

```
$ python automate2.py -h
```

You'll see an option `-f` which basically redoes the post-processing by removing any old plots, so let us try that:

```
$ python automate2.py -f
4 tasks pending and 0 tasks running
...
Finished!
```

Now it actually ran just the new simulations (you can see the commands in the output it prints), it will not re-run the already executed cases. Now let us see if the output is collected:

```
$ cat manuscript/figures/squares/output.txt
[['1.0', '1.0'], ['2.0', '4.0'], ['3.0', '9.0'], ['4.0', '16.0']]
```

So what automan did was to execute the newly added cases and then executed our post-processing code in the `run` method to produce the output.

Building on this we have a slightly improved script, called `automate3.py`, which makes a plot:

```
from automan.api import Problem, Automator
from matplotlib import pyplot as plt
import numpy as np


class Squares(Problem):
    def get_name(self):
        return 'squares'

    def get_commands(self):
        commands = [(str(i), 'python square.py %d' % i, None)
                    for i in range(1, 8)]
        return commands

    def run(self):
        self.make_output_dir()
        data = []
        for i in range(1, 8):
            stdout = self.input_path(str(i), 'stdout.txt')
            with open(stdout) as f:
                values = [float(x) for x in f.read().split()]
                data.append(values)

        data = np.asarray(data)
        plt.plot(data[:, 0], data[:, 1], 'o-')
        plt.xlabel('x')
        plt.ylabel('y')
        plt.savefig(self.output_path('squares.pdf'))


automator = Automator(
    simulation_dir='outputs',
    output_dir='manuscript/figures',
    all_problems=[Squares]
)
automator.run()
```

This version simplifies the command generation by using a list-comprehension, so reduces several lines of code. It then makes a matplotlib plot with the collected data. Let us run this:

```
$ python automate3.py -f
5 tasks pending and 0 tasks running
...
Finished!

$ ls manuscript/figures/squares/
squares.pdf
```

As you can see, the old `output.txt` is gone and our plot is available.

Note: This example requires that you have matplotlib and NumPy installed.

If you wanted to change the plotting style in any way, you can do so and simply re-run `python automate3.py -f` and it will only regenerate the final plot without re-executing the actual simulations.

So what if you wish to re-run any of these cases? In this case you will need to manually remove the particular simulation (or even all of them). Let us try this:

```
$ rm -rf outputs/squares/3

$ python automate3.py -f
3 tasks pending and 0 tasks running
...
Finished!
```

It will just run the missing case and re-generate the plots.

While this may not seem like much, we've fully automated our simulation and analysis.

2.3 Doing things a bit better

The previous section demonstrated the basic ideas so you can get started quickly. Our example problem was very simple and only produced command line output. Our next example is a simple problem in the same directory called `powers.py`. This problem is also simple but supports a few command line arguments and is as follows:

```
import argparse
import os

import numpy as np


def compute_powers(r_max, power):
    """Compute the powers of the integers upto r_max and return the result.
    """
    result = []
    for i in range(0, r_max + 1):
        result.append((i, i**power))
    x = np.arange(0, r_max + 1)
    y = np.power(x, power)
    return x, y


def main():
    p = argparse.ArgumentParser()
    p.add_argument(
        '--power', type=float, default=2.0,
        help='Power to calculate'
    )
    p.add_argument(
        '--max', type=int, default=10,
        help='Maximum integer that we must raise to the given power'
    )
    p.add_argument(
        '--output-dir', type=str, default='.',
        help='Output directory to generate file.'
    )
    opts = p.parse_args()

    x, y = compute_powers(opts.max, opts.power)

    fname = os.path.join(opts.output_dir, 'results.npz')
    np.savez(fname, x=x, y=y)

```

(continues on next page)

(continued from previous page)

```
39
40 if __name__ == '__main__':
41     main()
```

Again, the example is very simple, bulk of the code is parsing command line arguments. There are three arguments the code can take:

- `--power power` specifies the power to be computed.
- `--max` specifies the largest integer in sequence whose power is to be computed
- `--output-dir` is the directory where the output should be generated.

When executed, the script will create a `results.npz` file which contains the results stored as NumPy arrays. This example also requires that matplotlib and NumPy be installed. Let us run the code:

```
$ python powers.py

$ ls results.npz
results.npz

$ python powers.py --power 3.5
```

On a Python interpreter we can quickly look at the results:

```
$ python

>>> import numpy as np
>>> data = np.load('results.npz')
>>> data['x']
>>> data['y']
```

This looks about right, so let us move on to see how we can automate running several cases of this script in a better way than what we had before. We will continue to automate the previous `squares.py` script. This shows you how you can use automan incrementally as you add more cases. We only show the lines that are changed and do not reproduce the `Squares` problem class in the listing below.

```
31 from automan.api import Simulation
32
33
34 class Powers(Problem):
35     def get_name(self):
36         return 'powers'
37
38     def setup(self):
39         base_cmd = 'python powers.py --output-dir $output_dir'
40         self.cases = [
41             Simulation(
42                 root=self.input_path(str(i)),
43                 base_command=base_cmd,
44                 power=float(i)
45             )
46             for i in range(1, 5)
47         ]
48
49     def run(self):
50         self.make_output_dir()
```

(continues on next page)

(continued from previous page)

```
        for case in self.cases:
            data = np.load(case.input_path('results.npz'))
            plt.plot(
                data['x'], data['y'],
                label=r'$x^{{%.2f}}$' % case.params['power']
            )
        plt.grid()
        plt.xlabel('x')
        plt.ylabel('y')
        plt.legend()
        plt.savefig(self.output_path('powers.pdf'))


if __name__ == '__main__':
    automator = Automator(
        simulation_dir='outputs',
        output_dir='manuscript/figures',
        all_problems=[Squares, Powers]
    )
    automator.run()
```

To see the complete file see automate4.py. The key points to note in the code are the following:

- As before `get_name()` simply returns a convenient name where the outputs will be stored.
- A new `setup()` method is used and this creates an instance attribute called `self.cases` which is a list of cases we wish to simulate. Instead of using strings in the `get_commands` we simply setup the `cases` and no longer need to create a `get_commands`. We discuss `Simulation` instances in greater detail below.
- The `run()` method is similar except it uses the `cases` attribute and some conveniences of the simulation objects for convenience.

The *`automan.automation.Simulation`* instances we create are more general purpose and are very handy. A simulation instance's first argument is the the output directory and the second is a basic command to execute. It takes a third optional argument called `job_info` which specifies the number of cores and threads to use and we discuss this later. For now let us ignore it. In addition any keyword arguments one passes to this are automatically converted to command line arguments. Let us try to create one of these on an interpreter to see what is going on:

```
>>> from automan.api import Simulation
>>> s = Simulation(root='some_output/dir/blah',
...                base_command='python powers.py', power=3.5)
>>> s.name
'blah'
>>> s.command
'python powers.py --power=3.5'
>>> s.params
{'power': 3.5}
```

Notice that the name is simply the basename of the root path. You will see that additional keyword argument `power=3.5` is converted to a suitable command line argument. This is done by the `Simulation.get_command_line_args` method and can be overridden if you wish to do something different. The `Simulation.params` attribute simply stores all the keyword arguments so you could use it later while post-processing.

Now we want that each execution of this command produces output into the correct directory. We could either roll this into the `base_command` argument by passing the correct output directory or there is a nicer way to do this using the magic `$output_dir` argument that is automatically set the output directory when the command is executed, for example:

```
>>> from automan.api import Simulation
>>> s = Simulation(root='some_output/dir/blah',
...                base_command='python powers.py --output-dir $output_dir', power=3.
↪5)
>>> s.command
'python powers.py --output-dir $output_dir --power=3.5'
```

Note that the magic variable is not substituted at this point but later when the program is about to be executed.

Given these details, the code in the `run` method should be fairly straightforward to understand. Note that this organization of our code has made us maximize reuse of our plotting code. The `case.params` attribute is convenient when post-prprocessing. One can also filter the cases using the `filter_cases` function that is provided by `automan`. We discuss this later.

The last change to note is that we add the `Powers` class to the `Automator`'s `all_problems` and we are done.Let us now run this:

```
$ python automate4.py
6 tasks pending and 0 tasks running
...
Finished!
```

This only executes the new cases from the `Powers` class and makes the plot in `manuscript/figures/powers/powers.pdf`.

Using *`automan.automation.Simulation`* instances allows us to parametrize simulations with the keyword arguments. In addition, it is handy while post-processing. We can also subclass the `Simulation` instance to customize various things or share code.

There are a few more conveniences that automan provides that are useful while post-processing and these are discussed below.

2.4 Filtering and comparing cases

`automan` provides a couple of handy functions to help filter the different cases based on the parameters or the name of the cases. One can also make plots for a collection of cases and compare them easily.

- The *`filter_cases()`* function takes a sequence of cases and any additional keyword arguments with parameter values and filters out the cases having those parameter values. For example from our previous example in the `Powers` class, if we do the following:

  ```
  filtered_cases = filter_cases(cases, power=2)
  ```

  will return a list with a single case which uses `power=2`. This is very handy. This function can also be passed a callable which returns `True` for any acceptable case. For example:

  ```
  filter_cases(cases, lambda x: x.params['power'] % 2)
  ```

  will return all the cases with odd powers.

- The *`filter_by_name()`* function filters the cases whose names match the list of names passed. For example:

  ```
  filter_by_name(cases, ['1', '4'])
  ```

  will return the two simulations whose names are equal to `'1'` or `'4'`.

- The *compare_runs()* function calls a method or callable with the given cases. This is very handy to make comparison plots.

With this information you should be in a position to automate your computational simulations and analysis.

Next we look at setting up additional remote computers on which we can execute our computations.

2.5 Using additional computational resources

Wouldn't it be nice if we could easily run part of the simulations on one or more remote computers? `automan` makes this possible. Let us see how with our last example.

Let us first remove all the generated outputs and files so we can try this:

```
$ rm -rf outputs/ manuscript/figures config.json
```

Running the simulations on a remote machine requires a few things:

- the computer should be running either Mac OS or Linux/Unix.
- you should have an account on the computer, and be able to `ssh` into it without a password (see article on password-less ssh.
- the computer should have a working basic Python interpreter.
- on Linux the remote `libpython*.so` should be built as a shared library. If you are building Python from source you must do `./configure --enable-shared` or use a packaged Python like conda/miniconda/edm.
- Make sure that the python interpreter is in the $PATH and that the library directory is in $LD_LIBRARY_PATH – again this applies only for a Python binary that you have built by yourself.
- while running commands from the host computer, remove the following from the remote computer's .bashrc:

```
case $- in
*i*) ;;
*) return;;
esac
```

For more complex dependencies, you need to make sure the remote machine has the necessary software.

Assuming you have these requirements on a computer accessible on your network you can do the following:

```
$ python automate4.py -a host_name
[...]
```

Where `host_name` is either the computer's name or IP address. This will print a lot of output and attempt to setup a virtual environment on the remote machine. If it fails, it will print out some instructions for you to fix.

If this succeeds, you can now simply use the automation script just as before and it will now run some of the code on the remote machine depending on its availability. For example:

```
$ python automate4.py
14 tasks pending and 0 tasks running

Running task CommandTask with output directory: 4 ...
Starting worker on localhost.
Job run by localhost
Running python powers.py --output-dir outputs/powers/4 --power=4.0
```

(continues on next page)

(continued from previous page)

```
Running task CommandTask with output directory: 3 ...
Starting worker on 10.1.10.242.
Job run by 10.1.10.242
Running python powers.py --output-dir outputs/powers/3 --power=3.0
...
```

Note that you can add new machines at any point. For example you may have finished running a few simulations already and are simulating a new problem that you wish to distribute, you can add a new machine and fire the automation script and it will use it for the new simulations.

When you add a new remote host `automan` does the following:

- Creates an `automan` directory in the remote machine home directory (you can set a different home using `python automate4.py -a host --home other_home`.)
- Inside this directory it copies the current project directory, `tutorial` in the present case.
- It then copies over a `bootstrap.sh` and `update.sh` and runs the `bootstrap.sh` script. These scripts are inside a `.automan/` directory on your localhost and you may edit these if you need to.

The bootstrap code does the following:

- It creates a virtualenv called `tutorial` on this computer using the system Python and puts this in `automan/envs/tutorial`.
- It then activates this environment, installs `automan` and also runs any `requirements.txt` if they exist in the tutorial directory.

If for some reason this script fails, you may edit it on the remote host and re-run it.

When executing the code, automan copies over the files from the remote host to your computer once the simulation is completed and also deletes the output files on the remote machine.

If your remote computer shares your file-system via nfs or so, you can specify this when you add the host as follows:

```
$ python automate4.py -a host_sharing_nfs_files --nfs
```

In this case, files will not be copied back and forth from the remote host.

Now lets say you update files inside your project you can update the remote hosts using:

```
$ python automate4.py -u
```

This will update all remote workers and also run the `update.sh` script on all of them. It will also copy your local modifications to the scripts in `.automan`. It will then run any simulations.

Lets say you do not want to use a particular host, you can remove the entry for this in the `config.json` file.

When `automan` distributes tasks to machines, local and remote, it needs some information about the task and the remote machines. Recall that when we created the `Simulation` instances we could pass in a `job_info` keyword argument. The `job_info` is an optional dictionary with the following optional keys:

- `'n_core'`: the number of cores that this simulation requires. This is used for scheduling tasks. For example if you set `n_core=4` and have a computer with only 2 cores, automan will not be able to run this job on this machine at all. On the other hand if the task does indeed consume more than one core and you set the value to one, then the scheduler will run the job on a computer with only one core available.
- `'n_thread'`: the number of threads to use. This is used to set the environment variable `OMP_NUM_THREADS` for OpenMP executions.

As an example, here is how one would use this:

```
Simulation(root=self.input_path('3.5'),
           base_command='python powers.py',
           job_info=dict(n_core=1, n_thread=1),
           power=3.5
)
```

This job requires only a single core. So when automan tries to execute the job on a computer it looks at the load on the computer and if one core is free, it will execute the job.

If for some reason you are not happy with how the remote computer is managed and wish to customize it, you can feel free to subclass the *automan.cluster_manager.ClusterManager* class. You may pass this in to the *automan.automation.Automator* class as the `cluster_manager_factory` and it will use it. This is useful if for example you wish to use conda or some other tool to manage the Python environment on the remote computer.

We provide two simple environment managers one is a based on anaconda's conda and the other is on Enthought's edm, the following contains details on how to use them.

A simple *automan.conda_cluster_manager.CondaClusterManager* which will setup a remote computer so long as it has conda on it. If your project directory has an `environments.yml` and/or a `requirements.txt` it will use those to setup the environment. This is really a prototype and you may feel free to customize this. To use the conda cluster manager you could do the following in the tutorial example:

```
from automan.api import CondaClusterManager

automator = Automator(
    simulation_dir='outputs',
    output_dir='manuscript/figures',
    all_problems=[Squares, Powers],
    cluster_manager_factory=CondaClusterManager
)
automator.run()
```

A simple *automan.edm_cluster_manager.EDMClusterManager* which will setup a remote computer so long as it has edm on it. If your project directory has an `bundled_envs.json` and/or a `requirements.txt` it will use those to setup the environment. You can change the file names by accessing `ENV_FILE` class variable . By default this assumes the edm executable location to be in `~/.edm` to change this point the `EDM_ROOT` variable to the correct location relative to ~ (the current user home folder) not including the symbol ~. To use the edm cluster manager you could do the following in the tutorial example:

```
from automan.api import EDMClusterManager

automator = Automator(
    simulation_dir='outputs',
    output_dir='manuscript/figures',
    all_problems=[Squares, Powers],
    cluster_manager_factory=EDMClusterManager
)
automator.run()
```

You may also subclass these or customize the bootstrap code and use that.

A complete example of each of these is available in the `examples/edm_conda_cluster` directory that you can see here https://github.com/pypr/automan/tree/master/examples/edm_conda_cluster

The README in the directory tells you how to run the examples.

2.6 Specifying simulation dependencies

New in version 0.4: Specifying simulation dependencies was added in the 0.4 version.

There are times when one simulation uses the output from another and you wish to execute them in the right order. This can be quite easily achieved. Here is a simple example from the test suite that illustrates this:

```
class MyProblem(Problem):
    def setup(self):
        cmd = 'python -c "import time; print(time.time())"'
        s1 = Simulation(self.input_path('1'), cmd)
        s2 = Simulation(self.input_path('2'), cmd, depends=[s1])
        s3 = Simulation(self.input_path('3'), cmd, depends=[s1, s2])
        self.cases = [s1, s2, s3]
```

Notice the extra keyword argument, `depends=` which specifies a list of other simulations. In the above case, we could have also used `self.cases = [s3]` and that would have automatically picked up the other simulations.

When this problem is run, `s1` will run first followed by `s2` and then by `s3`. Note that this will only execute `s1` once even though it is declared as a dependency for two other simulations. This makes it possible to easily define inter-dependent tasks/simulations. In general, the dependencies could be any *automan.automation.Simulation* or *automan.automation.Task* instance.

Internally, these simulations create suitable task instances that support dependencies see *automan.automation.CommandTask*

2.7 Specifying inter-problem dependencies

New in version 0.4: Specifying problem dependencies was added in the 0.4 version.

Sometimes you may have a situation where one problem depends on the output of another. These may be done by overriding the `Problem.get_requires` method. Here is an example from the test suite:

```
class A(Problem):
    def get_requires(self):
        cmd = 'python -c "print(1)"'
        ct = CommandTask(cmd, output_dir=self.sim_dir)
        return [('task1', ct)]

class B(Problem):
    def get_requires(self):
        # or return Problem instances ...
        return [('a', A(self.sim_dir, self.out_dir))]

class C(Problem):
    def get_requires(self):
        # ... or Problem subclasses
        return [('a', A), ('b', B)]
```

Normally, the `get_requires` method automatically creates tasks from the simulations specified but in the above example we show a case (problem `A`) where we explicitly create command tasks. In the above example, the problem `B` depends on the problem `A` and simply returns an instance of `A`. On the other hand `C` only returns the problem class and not an instance. This shows how one can specify inter problem dependencies.

Note that if the problem performs some simulations (by setting `self.cases`), you should call the parent method (via `super`) and add your additional dependencies to this.

Also note that the dependencies are resolved based on the "outputs" of a task. So two tasks with the same outputs are treated as the same. This is consistent with the design of automan where each simulation's output goes in its own directory.

2.8 Using docker

It should be possible to use automan from within a Docker container. This can be done either by specifying commands to be run within suitable `docker run` invocations. Alternatively, one can install automan and run scripts within the docker container and this will work correctly.

One could use docker on the remote computers also but this is not yet fully tested.

2.9 Learning more

If you wish to learn more about automan you may find the following useful:

- Read the automan paper or a draft of the paper.
- The paper mentions another manuscript which was fully automated using automan, the sources for this are at https://gitlab.com/prabhu/edac_sph/ and this demonstrates a complete real-world example of using automan to automate an entire research paper.
- Olivier Mesnard has created a nice example as part of the review of this paper that can be seen here: https://github.com/mesnardo/automan-example the example also nicely shows how automan can be used from within a docker container for a completely reproducible workflow.

CHAPTER 3

Reference documentation

3.1 Reference Documentation

Autogenerated from doc strings using sphinx's autodoc feature.

3.1.1 Main automation module

class automan.automation.**Automator**(*simulation_dir*, *output_dir*, *all_problems*, *cluster_manager_factory=None*)

Main class to automate a collection of problems.

This processess command line options and runs all tasks with a scheduler that is configured using the `config.json` file if it is present. Here is typical usage:

```
>>> all_problems = [EllipticalDrop]
>>> automator = Automator('outputs', 'figures', all_problems)
>>> automator.run()
```

The class also creates a *automan.cluster_manager.ClusterManager* instance and integrates the cluster management features as well. This allows a user to automate their results across a collection of remote machines accessible only by ssh.

run(*argv=None*)

Start the automation.

class automan.automation.**CommandTask**(*command*, *output_dir*, *job_info=None*, *depends=None*)

Convenience class to run a command via the framework. The class provides a method to run the simulation and also check if the simulation is completed. The command should ideally produce all of its outputs inside an output directory that is specified.

clean()

Clean out any generated results.

This completely removes the output directory.

complete()
Should return True/False indicating success of task.

job

output()
Return list of output paths.

requires()
Return iterable of tasks this task requires.

It is important that one either return tasks that are idempotent or return the same instance as this method is called repeatedly.

run(*scheduler*)
Run the task, using the given scheduler.

Using the scheduler is optional but recommended for any long-running tasks. It is safe to raise an exception immediately when running the task but for long running tasks, the exception will not matter and the *complete* method should do.

class automan.automation.**Problem**(*simulation_dir*, *output_dir*)
This class represents a numerical problem or computational problem of interest that needs to be solved.

The class helps one run a variety of commands (or simulations), and then assemble/compare the results from those in the *run* method. This is perhaps easily understood with an example. Let us say one wishes to run the elliptical drop example problem with the standard SPH and TVF and compare the results and their convergence properties while also keep track of the computational time. To do this one will have to run several simulations, then collect and process the results. This is achieved by subclassing this class and implementing the following methods:

- *get_name(self)*: returns a string of the name of the problem. All results and simulations are collected inside a directory with this name.
- *get_commands(self)*: returns a sequence of (directory_name, command_string, job_info, depends) tuples. These are to be executed before the *run* method is called.
- *get_requires(self)*: returns a sequence of (name, task) tuples. These are to be exeuted before the *run* method is called.
- *run(self)*: Processes the completed simulations to make plots etc.

See the *EllipticalDrop* example class below to see a full implementation.

clean()
Cleanup any generated output from the analysis code. This does not clean the output of any nested commands.

get_commands()
Return a sequence of (name, command_string, job_info_dict) or (name, command_string, job_info_dict, depends).

The name represents the command being run and is used as a subdirectory for generated output.

The command_string is the command that needs to be run.

The job_info_dict is a dictionary with any additional info to be used by the job, these are additional arguments to the *automan.jobs.Job* class. It may be None if nothing special need be passed.

The depends is any dependencies this simulation has in terms of other simulations/tasks.

get_name()
Return the name of this problem, this name is used as a directory for the simulation and the outputs.

get_outputs()
Get a list of outputs generated by this problem. By default it returns the output directory (as a single element of a list).

get_requires()
Return a sequence of tuples of form (name, task).

The name represents the command being run and is used as a subdirectory for generated output.

The task is a *automan.automation.Task* instance.

input_path(**args*)
Given any arguments, relative to the simulation dir, return the absolute path.

make_output_dir()
Convenience to make the output directory if needed.

output_path(**args*)
Given any arguments relative to the output_dir return the absolute path.

run()
Run any analysis code for the simulations completed. This is usually run after the simulation commands are completed.

setup()
Called by init, so add any initialization here.

task_cls
alias of *CommandTask*

class automan.automation.**PySPHProblem**(*simulation_dir*, *output_dir*)

task_cls
alias of *PySPHTask*

class automan.automation.**PySPHTask**(*command*, *output_dir*, *job_info=None*, *depends=None*)
Convenience class to run a PySPH simulation via an automation framework.

This task automatically adds the output directory specification for pysph so users to not need to add it.

class automan.automation.**RunAll**(*simulation_dir*, *output_dir*, *problem_classes*, *force=False*, *match=''*)
Solves a given collection of problems.

requires()
Return iterable of tasks this task requires.

It is important that one either return tasks that are idempotent or return the same instance as this method is called repeatedly.

class automan.automation.**Simulation**(*root*, *base_command*, *job_info=None*, *depends=None*, ***kw*)
A convenient class to abstract code for a particular simulation. Simulation objects are typically created by `Problem` instances in order to abstract and simulate repetitive code for a particular simulation.

For example if one were comparing the elliptical_drop example, one could instantiate a Simulation object as follows:

```
>>> s = Simlation('outputs/sph', 'pysph run elliptical_drop')
```

One can pass any additional command line arguments as follows:

```
>>> s = Simlation(
...     'outputs/sph', 'pysph run elliptical_drop', timestep=0.005
... )
>>> s.command
'pysph run elliptical_drop --timestep=0.001'
>>> s.input_path('results.npz')
'outputs/sph/results.npz'
```

The extra parameters can be used to filter and compare different simulations. One can define additional plot methods for a particular subclass and use these to easily plot results for different cases.

One can also pass any additional parameters to the *automan.jobs.Job* class via the job_info kwarg so as to run the command suitably. For example:

```
>>> s = Simlation('outputs/sph', 'pysph run elliptical_drop',
...               job_info=dict(n_thread=4))
```

The object has other methods that are convenient when comparing plots. Along with the `compare_cases`, `filter_cases` and `filter_by_name` this is an extremely powerful way to automate and compare results.

command

data

get_command_line_args()

get_labels(*labels*)

input_path(**args*)
Given any arguments, relative to the simulation dir, return the absolute path.

kwargs_to_command_line(*kwargs*)

render_parameter(*param*)
Return string to be used for labels for given parameter.

class automan.automation.**SolveProblem**(*problem*, *match=''*, *force=False*)
Solves a particular *Problem*. This runs all the commands that the problem requires and then runs the problem instance's run method.

The match argument is a string which when provided helps run only a subset of the requirements for the problem.

The force argument specifies that the problem should be cleaned, so as to re-run any post-processing.

complete()
Should return True/False indicating success of task.

If the task was just executed (in this invocation) but failed, raise any Exception that is a subclass of Exception as this signals an error to the task execution engine.

If the task was executed in an earlier invocation of the automation, then just return True/False so as to be able to re-run the simulation.

output()
Return list of output paths.

requires()
Return iterable of tasks this task requires.

It is important that one either return tasks that are idempotent or return the same instance as this method is called repeatedly.

run(*scheduler*)
Run the task, using the given scheduler.

Using the scheduler is optional but recommended for any long-running tasks. It is safe to raise an exception immediately when running the task but for long running tasks, the exception will not matter and the *complete* method should do.

class automan.automation.**Task**
Basic task to run. Subclass this to do whatever is needed.

This class is very similar to luigi's Task class.

complete()
Should return True/False indicating success of task.

If the task was just executed (in this invocation) but failed, raise any Exception that is a subclass of Exception as this signals an error to the task execution engine.

If the task was executed in an earlier invocation of the automation, then just return True/False so as to be able to re-run the simulation.

output()
Return list of output paths.

requires()
Return iterable of tasks this task requires.

It is important that one either return tasks that are idempotent or return the same instance as this method is called repeatedly.

run(*scheduler*)
Run the task, using the given scheduler.

Using the scheduler is optional but recommended for any long-running tasks. It is safe to raise an exception immediately when running the task but for long running tasks, the exception will not matter and the *complete* method should do.

class automan.automation.**TaskRunner**(*tasks*, *scheduler*)
Run given tasks using the given scheduler.

add_task(*task*)

run(*wait=5*)
Run the tasks that were given.

Wait for the given amount of time to poll for completed tasks.

Returns the number of tasks that had errors.

class automan.automation.**WrapperTask**
A task that wraps other tasks and is done when all its requirements are done.

complete()
Should return True/False indicating success of task.

If the task was just executed (in this invocation) but failed, raise any Exception that is a subclass of Exception as this signals an error to the task execution engine.

If the task was executed in an earlier invocation of the automation, then just return True/False so as to be able to re-run the simulation.

automan.automation.**compare_runs**(*sims*, *method*, *labels*, *exact=None*)
Given a sequence of Simulation instances, a method name, the labels to compare and an optional method name for an exact solution, this calls the methods with the appropriate parameters for each simulation.

Parameters

sims: sequence Sequence of *Simulation* objects.

method: str or callable Name of a method on each simulation method to call for plotting. Or a callable which is passed the simulation instance and any kwargs.

labels: sequence Sequence of parameters to use as labels for the plot.

exact: str or callable Name of a method that produces an exact solution plot or a callable that will be called.

automan.automation.**filter_by_name**(*cases*, *names*)
Filter a sequence of Simulations by their names. That is, if the case has a name contained in the given *names*, it will be selected.

automan.automation.**filter_cases**(*runs*, *predicate=None*, ***params*)
Given a sequence of simulations and any additional parameters, filter out all the cases having exactly those parameters and return a list of them.

One may also pass a callable to filter the cases using the *predicate* keyword argument. If this is not a callable, it is treated as a parameter. If *predicate* is passed though, the other keyword arguments are ignored.

automan.automation.**key_to_option**(*key*)
Convert a dictionary key to a valid command line option. This simply replaces underscores with dashes.

automan.automation.**kwargs_to_command_line**(*kwargs*)
Convert a dictionary of keyword arguments to a list of command-line options. If the value of the key is None, no value is passed.

Examples

```
>>> sorted(kwargs_to_command_line(dict(some_arg=1, something_else=None)))
['--some-arg=1', '--something-else']
```

automan.automation.**linestyles**()
Cycles over a set of possible linestyles to use for plotting.

3.1.2 Low-level job management module

class automan.jobs.**Job**(*command*, *output_dir*, *n_core=1*, *n_thread=1*, *env=None*)

clean(*force=False*)

get_info()

get_stderr()

get_stdout()

join()

pretty_command()

run()

status()

substitute_in_command(*basename*, *substitute*)
Replace occurrence of given basename with the substitute.

This is useful where the user asks to run [‘python’, ‘script.py’] and we wish to change the ‘python’ to a specific Python. Normally this is not needed as the PATH is set to pick up the right Python. However, in the rare cases where this rewriting is needed, this method is available.

to_dict()

class automan.jobs.**JobProxy**(*worker*, *job_id*, *job*)

clean(*force=False*)

copy_output(*dest*)

free_cores()

get_info()

get_stderr()

get_stdout()

run()

status()

total_cores()

class automan.jobs.**LocalWorker**

clean(*job_id*, *force=False*)

copy_output(*job_id*, *dest*)

get_config()

get_info(*job_id*)

get_stderr(*job_id*)

get_stdout(*job_id*)

run(*job*)
Runs the job and returns a JobProxy for the job.

status(*job_id*)
Returns status of the job.

class automan.jobs.**RemoteWorker**(*host*, *python*, *chdir=None*, *testing=False*, *nfs=False*)

clean(*job_id*, *force=False*)

copy_output(*job_id*, *dest*)

free_cores()

get_config()

get_info(*job_id*)

get_stderr(*job_id*)

get_stdout(*job_id*)

run(*job*)
Runs the job and returns a JobProxy for the job.

status(*job_id*)
Returns status of the job.

total_cores()

class automan.jobs.**Scheduler**(*root='.'*, *worker_config=()*, *wait=5*)

add_worker(*conf*)

load(*fname*)

save(*fname*)

submit(*job*)

class automan.jobs.**Worker**

can_run(*n_core*)
Returns True if the worker can run a job with the required cores.

clean(*job_id*, *force=False*)

copy_output(*job_id*, *dest*)

free_cores()

get_info(*job_id*)

get_stderr(*job_id*)

get_stdout(*job_id*)

run(*job*)
Runs the job and returns a JobProxy for the job.

status(*job_id*)
Returns status of the job.

total_cores()

automan.jobs.**free_cores**()

automan.jobs.**serve**(*channel*)
Serve the remote manager via execnet.

automan.jobs.**total_cores**()

3.1.3 Cluster management module

Code to bootstrap and update the project so a remote host can be used as a worker to help with the automation of tasks.

This requires ssh/scp and rsync to work on all machines.

This is currently only tested on Linux machines.

exception automan.cluster_manager.**BootstrapError**

class automan.cluster_manager.**ClusterManager**(*root='automan'*, *sources=None*, *config_fname='config.json'*, *exclude_paths=None*, *testing=False*)
The cluster manager class.

This class primarily helps setup software on a remote worker machine such that it can run any computational jobs from the automation framework.

The general directory structure of a remote worker machine is as follows:

```
remote_home/            # Could be ~
    automan/            # Root of automation directory (configurable)
        envs/           # python virtual environments for use.
        my_project/     # Current directory for specific projects.
```

The project directories are synced from this machine to the remote worker.

The "my_project" is the root of the directory with the automation script and this should contain the required sources that need to be executed. One can use a list of source directories which will be copied over but it is probably most convenient to put it all in the root of the project directory to keep everything self-contained.

The *ClusterManager* class manages these remote workers by helping setup the directories, bootstrapping the Python virtualenv and also keeping these up-to-date as project directory is changed on the local machine.

The class therefore has two primary public methods,

1. *add_worker(self, host, home, nfs)* which adds a new worker machine by bootstrapping the machine with the software and the appropriate source directories.
2. *update()*, which keeps the directory and software up-to-date.

The class variables BOOTSTRAP and UPDATE are the content of scripts uploaded to these machines and should be extended by users to do what they wish.

The class creates a `config.json` in the current working directory that may be edited by a user. It also creates a directory called `.{self.root}` which defaults to `.automan`. The bootstrap and update scripts are put here and may be edited by the user for any new hosts.

One may override the *_get_python, _get_helper_scripts*, and *_get_bootstrap_code, _get_update_code* methods to change this to use other package managers like edm or conda. See the conda_cluster_manager for an example.

BOOTSTRAP = '#!/bin/bash\n\nset -e\nif hash virtualenv 2>/dev/null; then\n virtualenv

UPDATE = '#!/bin/bash\n\nset -e\nsource envs/{project_name}/bin/activate\n# Run any re

add_worker(*host*, *home*, *nfs*)

cli(*argv=None*)
This is just a demonstration of how this class could be used.

create_scheduler()
Return a *automan.jobs.Scheduler* from the configuration.

update(*rebuild=True*)

class automan.conda_cluster_manager.**CondaClusterManager**(*root='automan'*, *sources=None*, *config_fname='config.json'*, *exclude_paths=None*, *testing=False*)

BOOTSTRAP = '#!/bin/bash\n\nset -e\nCONDA_ROOT={conda_root}\nENV_FILE="{project_name}/

CONDA_ROOT = 'miniconda3'

UPDATE = '#!/bin/bash\n\nset -e\nCONDA_ROOT={conda_root}\nENV_FILE="{project_name}/env

class automan.edm_cluster_manager.**EDMClusterManager**(*root='automan'*, *sources=None*, *config_fname='config.json'*, *exclude_paths=None*, *testing=False*)

```
BOOTSTRAP = '#!/bin/bash\n\nset -e\nENV_FILE="{project_name}/{env_file}"\n\nif hash ed
EDM_ROOT = '.edm'
ENV_FILE = 'bundled_env.json'
UPDATE = '#!/bin/bash\n\nset -e\nENV_FILE="{project_name}/{env_file}"\n\nif hash edm 2
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

G

I

J

K

L

M

O

P

R

S

T

U

W