Tutorial: Software testing in scientific programming

David W.H. Swenson

E-CAM Extended Software Development Workshop
Leiden, Netherlands; 17 August 2017

Talk materials: https://gitlab.e-cam2020.eu/dwhswenson/software_testing

Funded by the EU (Project n°676531)

Outline

- Software testing: the basics
- Hands-on with the nose package and coverage
- Test helpers in OPS
- Test design

Software testing isn't rocket science

Why should you write tests?

- Um, so your code has fewer bugs? (Duh)
- Testable code is better-designed code
- Don't break things that do work (see regression testing)
- The tests *are* the API

Why don't you write tests?

- Because you're lazy

Why don't you write tests?

- Because you're in a rush to get to publication, and writing tests takes time
- Because scientific software is complicated, and so it is hard to write simple tests for it

Regression Testing and Continuous Integration

- “Regression tests:” Run tests after every change. Goal: Don't introduce new bugs.
- “Continuous integration:” Merge branches to master after every change. Goal: Don't let code diverge too far.

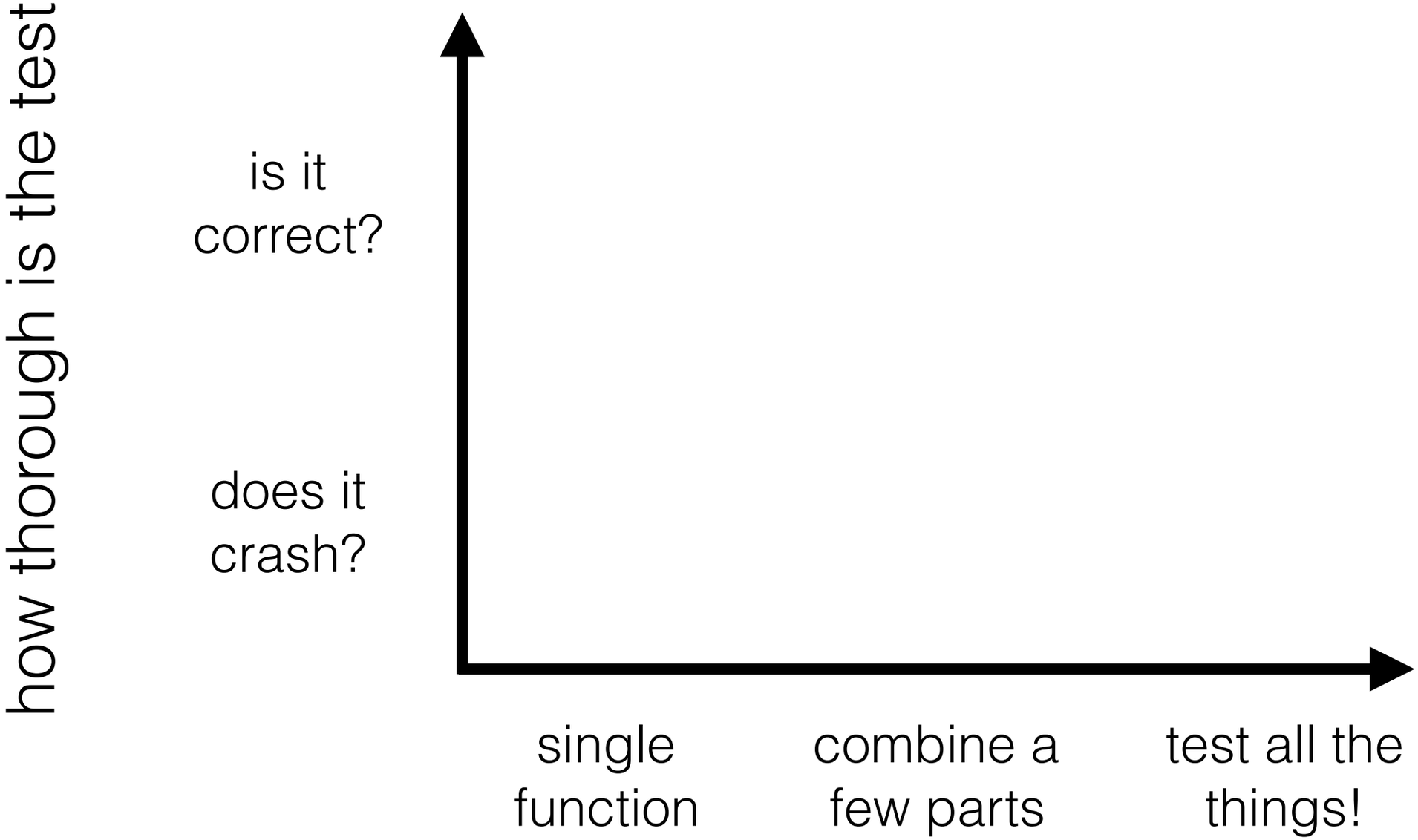
Closely related concepts. Once you open a PR to OPS, we attempt to merge your changes to master (CI). If that is successful, we run our test suite (RT).

Test Driven Development

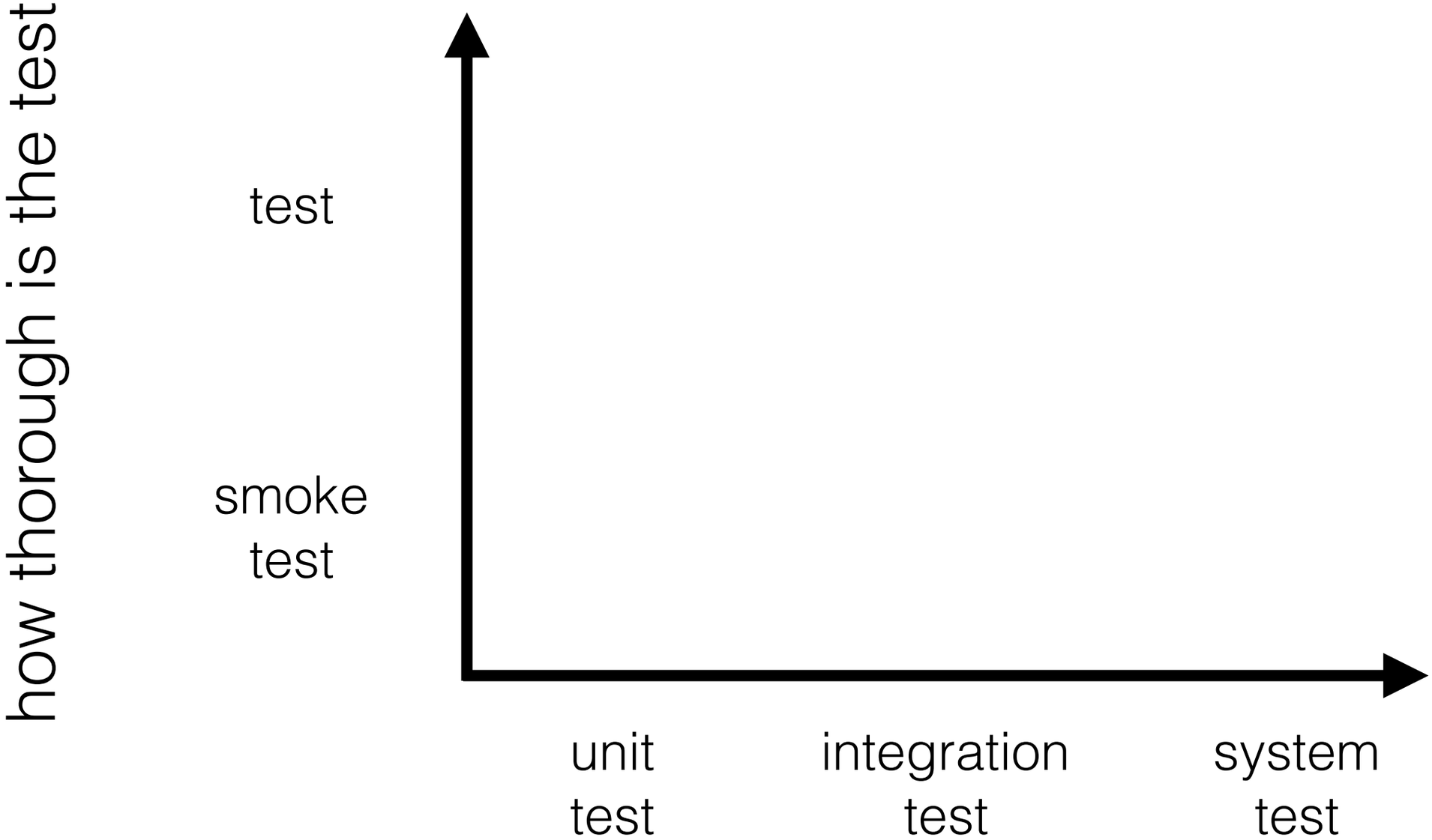
*Write the tests **FIRST**!*

1. Add a test
2. Run tests to see if new test fails
3. Write the code
4. Run tests
5. Refactor

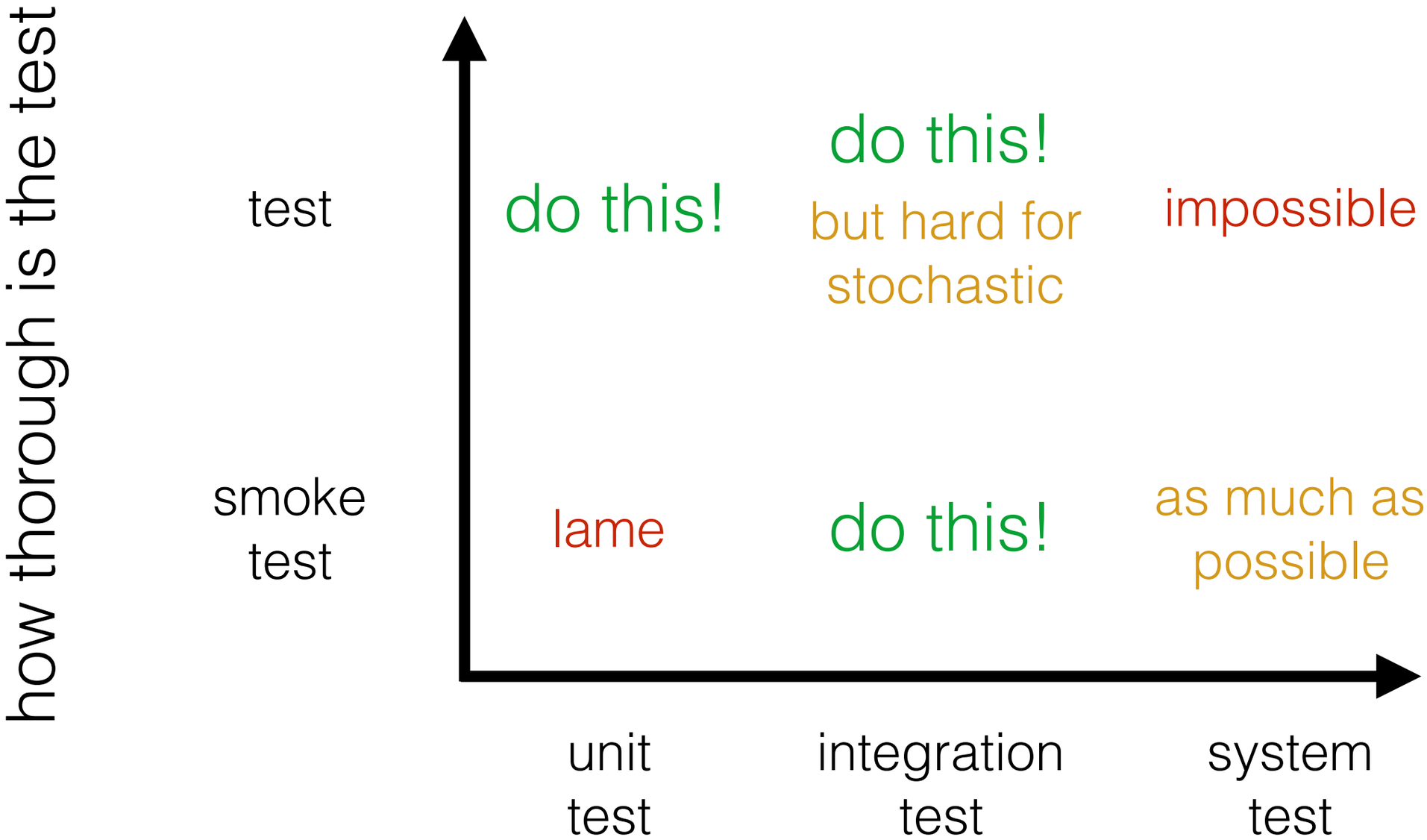
Terminology



Terminology



Terminology



True Unit Tests

- Test only a small part of the code
- Each test should be independent!

```
def square_it(number):
    return number * number

def first_n_squares(n):
    return [square_it(i+1)
            for i in range(n)]
```

```
def test_square_it():
    assert_equal(square_it(1), 1)
    assert_equal(square_it(2), 4)
    assert_equal(square_it(8), 64)

def test_first_n_squares():
    # without running square_it?
```

- OPS allows small integration tests (not independent) in place of true unit tests

Mocks

- How do you write a unit test, if your function depends on other functions?

- Think about the shooting move: how can you test that your shooting works without testing your engine?

- Answer: create something that acts like an engine, but is faster/predictable! A "mock engine"

- There are fancy tools for mocks; we use simple toy subclasses

OPS Tests

Primary tests (unit and basic integration):

- located in `openpathsampling/tests/`
- run using **`nosetests`** (from `root`, `openpathsampling/`, or `openpathsampling/tests/`)

IPython notebook tests (system and integration):

- located in `examples/`
- run using **`examples/ipynbtests.sh`**

Testing with nose

Various testing platforms

- `unittest`: comes with python
- `pytest`: powerful external package; what most people use now
- `nose`: older package; not actively developed

OpenPathSampling uses `nose` (although that may change)

`coverage`: check which lines are visited by tests

Autodiscovery of Tests

Preface stuff with `test` (or `Test`) and `nose` will automatically find the tests and run them.

```
# in test_module.py

def test_function():
    # bare test function
    pass

class TestClass(object):  # can inherit from object or UnitTest
    def test_method(self):
        # test method within class
        pass
```

Useful asserts

- `assert_equal`
- `assert_almost_equal`
- `assert_true`
- `assert_false`
- `assert_in`
- `assert_is`
- `assert_is_instance`
- `assert_is_not`
- `assert_regexp_matches`
- `assert_list_equal`
- `assert_tuple_equal`
- `assert_set_equal`
- `assert_greater`
- `assert_greater_equal`
- `assert_less`
- `assert_less_equal`

... and many more!

Test Structure Mirrors Code

```
# in file module.py
# this is the main code


def some_function():
    pass

class MyClass(object):
    def method(self):
        pass
```

```
# in file test_module.py
# tests for module.py
from module import *


def test_some_function():
    pass

class TestMyClass(object):
    def test_method(self):
        pass
```

Fixtures

Some parts of tests (especially setup) are often repeated

Fixtures allow you to group this boilerplate code together

```
def setup():
    print "Module-level setup"

def teardown():
    print "Module-level teardown"

class TestClass(object):
    def setup(self):
        print "  Class-level setup"

    def teardown(self):
        print "  Class-level teardown"
```

Coverage

Good testing requires quality (good tests)
and quantity (test all the code)

```
72            n_innermost = len(mover.innermost_ensembles)
73            if n_innermost != 1:
74                raise ValueError(
75                    mistis_err_str + "Mover " + str(mover) + " does not "
76                    + "have exactly one innermost ensemble. Found "
77                    + str(len(mover.innermost_ensembles)) + "."
78                )
79
80        if flux_pairs is None:
81            # get flux_pairs from network
82            flux_pairs = []
83            minus_ens_to_trans = self.network.special_ensembles['minus']
84            for minus_ens in self.network.minus_ensembles:
85                n_trans = len(minus_ens_to_trans[minus_ens])
86                if n_trans > 1:  # pragma: no cover
87                    # Should have been caught be the previous ValueError. If
88                    # you hit this, something unexpected happened.
89                    raise ValueError(mistis_err_str + "Ensemble "
90                                     + repr(minus_ens) + " connects "
91                                     + str(n_trans) + " transitions.")
```

Minimal check:
Does every line of code get run during tests?

Live Demo!

- Raising exceptions (Jupyter notebook)
- Running nosetests
- Tests and coverage (simple example)
- Tests and coverage (you do it)

OPS Development Cycle

- Start code locally
- Run tests locally (nosetests, examples/ipynbtests.sh)
- Open pull request
  - Automatically run nose tests, then ipynb tests
  - Automatically check coverage of nose tests

Test Helpers in OPS

make_1d_trajectory

```
import openpathsampling.tests.test_helpers as ops_test

traj = ops_test.make_1d_traj([0.5, 1.5, 0.7])

print traj.xyz
# [[[ 0.5  0.   0. ]]
#  [[ 1.5  0.   0. ]]
#  [[ 0.7  0.   0. ]]]
```

CalvinistDynamics

```
ensemble = paths.LengthEnsemble(3)
input_sample = paths.Sample(trajectory=traj,
                            ensemble=ensemble,
                            replica=0)
```

```
engine = ops_test.CalvinistDynamics(
    predestination=[1.5, 2.0, 3.0]
)
```

```
fwd_shooter = paths.ForwardShootMover(
    ensemble=ensemble,
    selector=paths.UniformSelector(),
    engine=engine
)
trials, details = fwd_shooter(input_sample)
```

CalvinistDynamics

```
print details
# {'initial_trajectory': Trajectory[3],
#  'shooting_snapshot': <ToySnapshot at 0x111770190>}
```

```
print trials[0].trajectory.xyz
# [[[ 0.5  0.   0. ]]
#  [[ 1.5  0.   0. ]]
#  [[ 2.   0.   0. ]]]
```

Pseudo-Mocks: Simple engines

```
import openpathsampling.engines.toy as toys
pes = toys.LinearSlope(m=[0.0], c=[0.0])
integ = toys.LeapfrogVerletIntegrator(dt=0.1)
```

A perfectly flat PES, with a perfectly predictable integrator.

Simple integration tests that are properly testable!

Test Design

Test Design Brainstorm

- Engine tests: compare with other examples
- Collective Variable tests: other examples
- PathMovers: Use deterministic input
- PathSimulators: often use pseudo-mock engines, but make sure it isn't expensive