Testing

Programming good research code good

Patrick Mineault

Bulding around testing

Most scientists who write software constantly test their code. That is, if you are a scientist writing software, I am sure that you have tried to see how well your code works by running every new function you write, examining the inputs and the outputs of the function, to see if the code runs properly (without error), and to see whether the results make sense. Automated code testing takes this informal practice, makes it formal, and automates it, so that you can make sure that your code does what it is supposed to do, even as you go about making changes around it. (Ariel Rokem, Shablona README)

Demo

- Let's test `fib.py`

What can we test about `fib`?

- Correctness, e.g. $F(4) = 5$
- Edge cases, e.g. $F(0) = 1$, $F(-1) \rightarrow$ *error*
- Functional goals are achieved, e.g. caching works
- It's much easier to test decoupled code with no side effects
  - Forces you to write modular decoupled code

How can you decide what to test?

- If something caused a bug, test it
  - 70% of bugs will be old bugs that keep reappearing
- If you manually checked if procedure X yielded something reasonable results, write a test for it.

How can we test?

- `assert`
- Hide code behind `if __name__ == '__main__'`
- Test suite

assert

- `assert` throws an error if the assertion is False

```
assert -(7 // 2) == (-7 // 2)
```

- Great for inline tests
  - e.g. check whether the shape of a matrix is correct after a permute op

Hide code behind `if __name__ == '__main__'`

- Code behind `__name__ == '__main__'` is only run if you run the file as a script directly.
- Use this for lightweight tests in combination with `assert`.

```
if __name__ == '__main__':
    assert fib(4) == 5
```

Use a test suite

- Create a specialized file of tests that runs with the help of a runner.
- There's `pytest` and `unittest`.
- I use `unittest` because that's what I learned, and it's built in.

Basic template

```python
# test_something.py
import unittest

class MyTest(unittest.TestCase):
    def sample_test(self):
        self.assertTrue(True)

if __name__ == '__main__':
    unittest.main()
```

Run it

```
$ python test_something.py
```

Or using nose:

```
$ nosetests
```

A hierarchy of tests can be run with a runner

- Static tests (literally your editor parsing your code to figure if it will crash)
- Asserts
- Unit tests (what we just saw)
- Integration tests
- Smoke tests
- Regression tests
- E2E (literally a robot clicking buttons)

An integration test

- E.g. I write a spiffy function that fits a GLM with L1 regularization
- How do I test that?
  - I make up some test data
  - I run my model
  - It gives me the correct outputs

An integration test

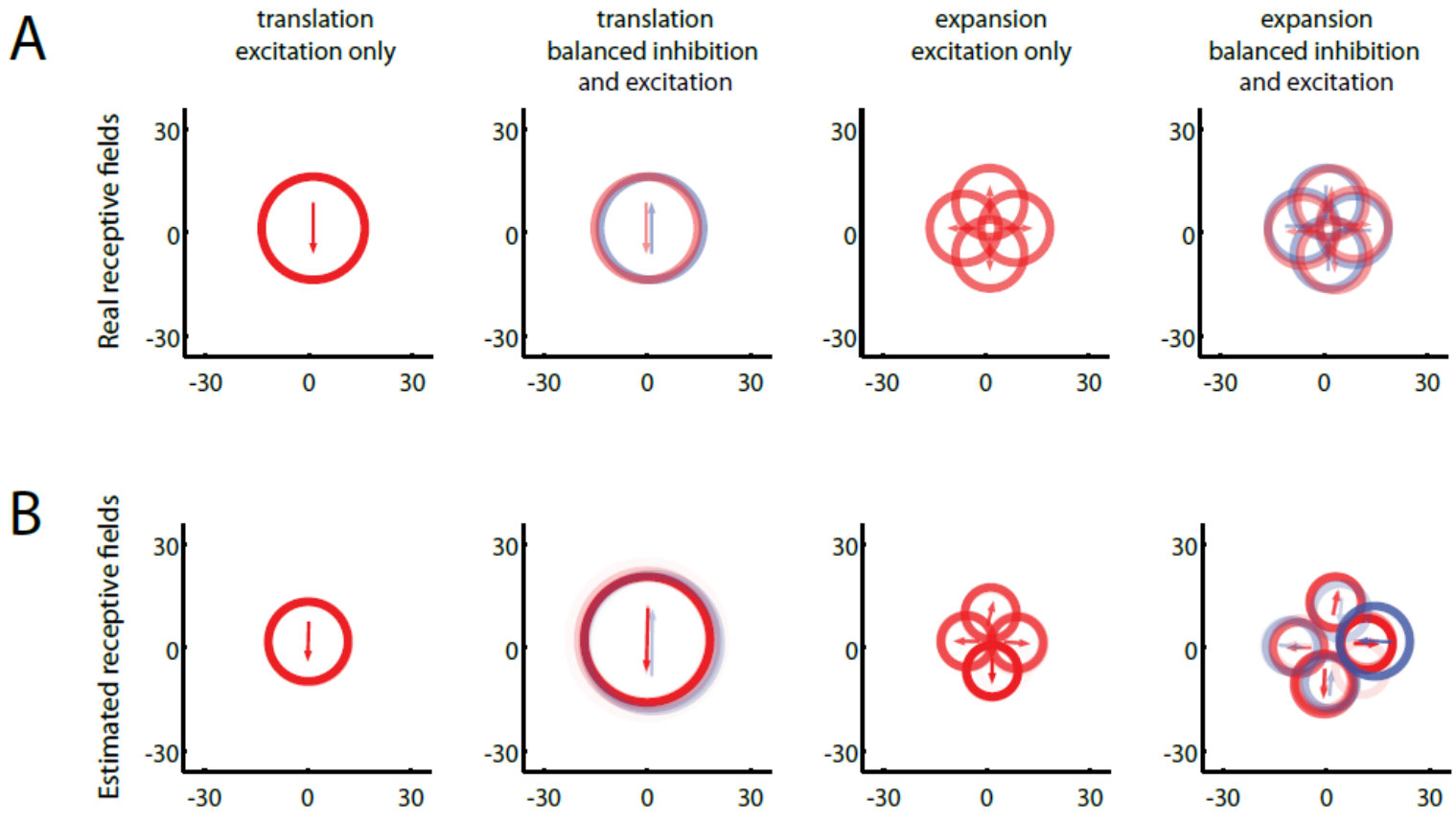

Figure 1: S1 in Mineault et al. 2011

Demo

- Let's code CKA tests

More things!

- Testing computational code has a very high returns:effort ratio, but. . .
- You can also test data loaders for correctness.
- You can also test data for correctness
- You can also test notebooks for correctness
- You can integrate your tests into Github

Lesson 3

- Test your code
- Free your WM from having to consider that a piece of code unrelated to the thing you care about is broken
- From lesson 1: much simpler to refactor code to make it tidy when you know you have a test scaffold which catch mistakes
- From lesson 2: you will have to decouple code to write tests
- Your 5-minute assignment: find a commented-out print statement in your code and replace it with `assert`