

Best Practice in Software Design

By Edward Smith
www.edwardsmith.co.uk

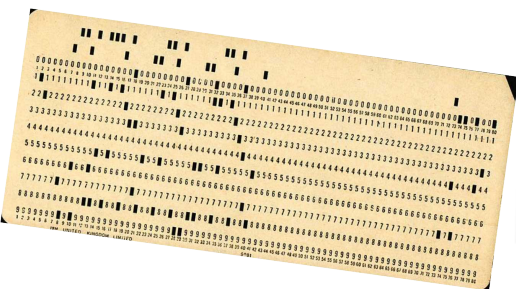
Overview

- Research Software Engineering
 - Introduction
 - Programming to an interface
- Testing and version control
 - Minimal examples of testing an interface
 - TDD and continuous integration
- Examples from my work with CPL library
 - Quick overview of what it does/challenges
 - Unit testing and deployment examples
- Frustrations and discussion points

My Background

- Researcher and software engineer (moving to Brunel as lecturer)
 - Civil, Mech & Chem Eng at IC (EPSRC, dCSE and eCSE funding)
 - About 11 years of programming experience
 - Software Sustainability Institute fellow (www.software.ac.uk)
 - Taught Python at Imperial
 - Organised a workshop on Continuous Integration for HPC
 - Involved with RSE at Imperial and several UK RSE conferences
 - Answer questions on Stackoverflow (7k+ reputation)
- I want to promote best practice to prevent people going through the same process I did

Excel → MATLAB → FORTRAN → Fortran → Python → C++




```
66 GO TO 77 ; . . . import numpy as np
IF (E) THEN ; x = np.array([1,2,3])
77 END IF
```

for (auto &f : objs)
f->run(x);

Research Software Engineering

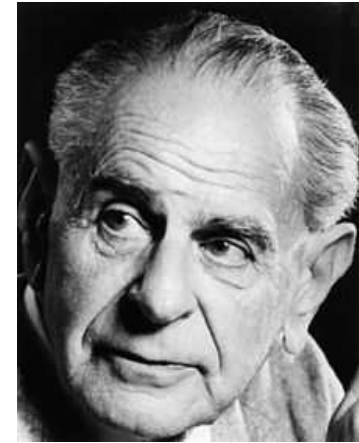
Research Software Engineering

- Software sustainability institute (SSI) is a great place to start!  Software Sustainability Institute
- Improving software is more than just learning Python, Git and Docker
- Many researchers work on legacy software and believe it's the way to code
- Nothing wrong with Fortran – no bad languages, just bad code

Scientific Method for Scientific Computing

“In so far as a scientific statement speaks about reality, it must be falsifiable; and in so far as it is not falsifiable, it does not speak about reality”

Karl Popper



The Scientific Method

- Create a theory that explains reality
- Present to the scientific community
- It is valid until a single counter example is found

“In so far as a scientific **software** speaks about reality, it must be **testable**; and in so far as it is not **testable**, it does not speak about reality”

The Software Design Method

- Create **software** that explains reality
- Make it **open** to the scientific community
- It is only valid until a single **bug** is found

```

1 import unittest
2
3 class tests(unittest.TestCase):
4     def test_square(self):
5         self.assertEqual(square(2.), 4.)
6     def test_cube(self):
7         self.assertEqual(cube(2.), 8.)
8
9 unittest.main()
10

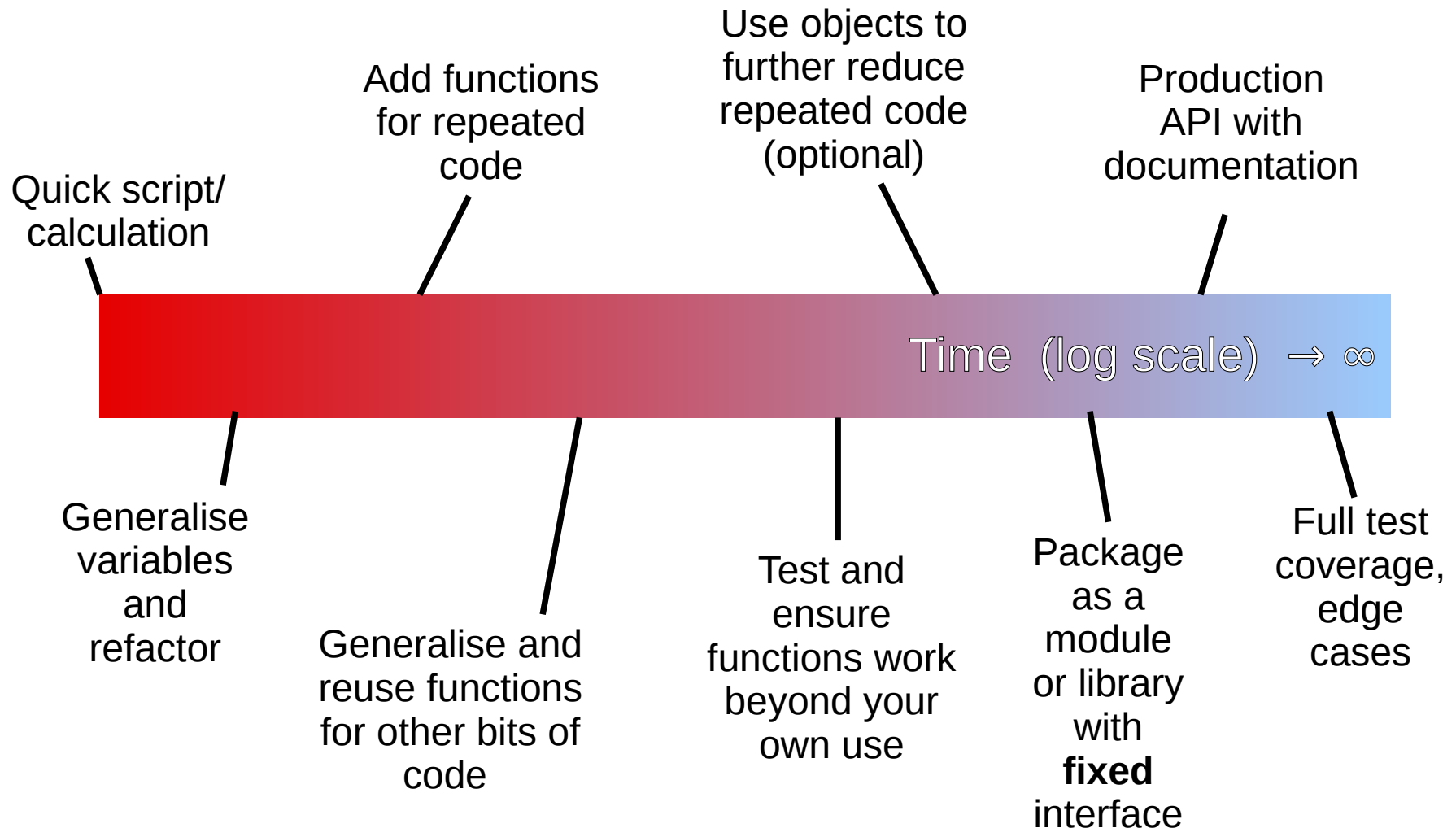
```

The Engineering Method for Software Engineering

- Scientific software is a prototype
 - Quick to develop with novel parts
 - Bespoke components developed by a small team who understand everything
 - Generally not reliable (but fast)
- Software Engineering is mass production
 - Slow to develop
 - Standardised components with clear interface
 - Reliability from rigorous testing procedure validation



A Pragmatic Developer Spectrum



Programming to an Interface

- The one thing I'd wish I'd understood earlier when writing code
 - Consider a USB port, you can use the same cable anywhere
 - Standardisation of interfaces built the modern world



Functional Interface

- The inputs to a function and returned output are like a contract with the user, 'give me this and I will give you that'
 - Take inputs in some format
 - Return output in some format
 - This hides the complexity from the user, you only need to know the format of the function or class
- When releasing software, version number systems are based around this
 - From v1.0 to v1.1 the interface stays the same
 - If major number changes, e.g. v1.1 to v2.0, the interface has changed and is no longer backward compatible

$$y = f(x)$$

Functional Interface

- Functions are like a contract with the user, here we take in the file name and return the data from the file

```
def square(a):  
    return a**2
```

TAKES A NUMBER AND RETURNS ITS SQUARE

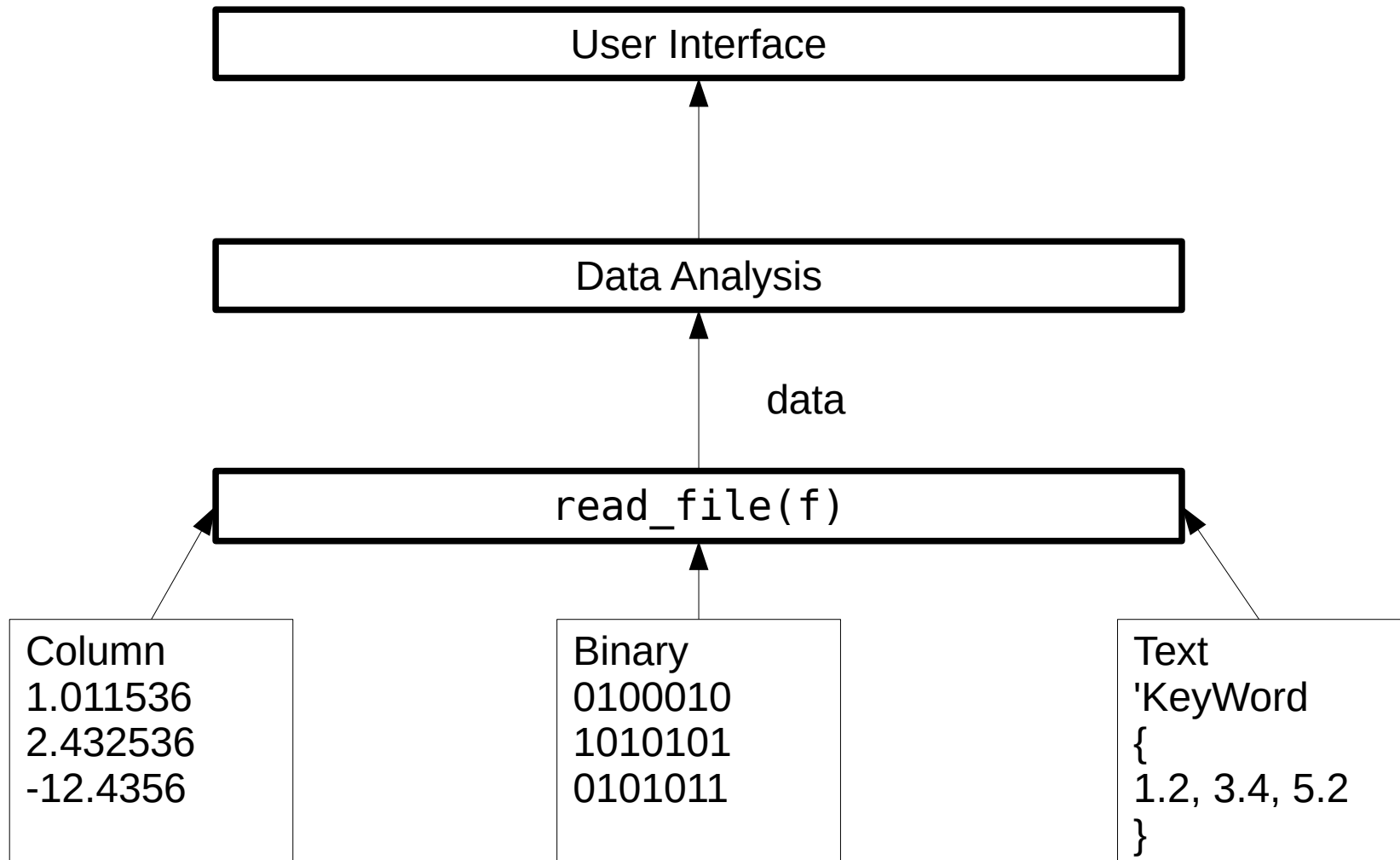
```
#Iterate through files
```

```
for f in files:  
    data = read_file(f)
```

TAKES A FILENAME AND RETURNS ITS CONTENTS

- We aim to design functions so for a given input we get back a consistent output

Examples of an Interface



Programming to an Interface

Focus your design on what the code is doing, not how it does it

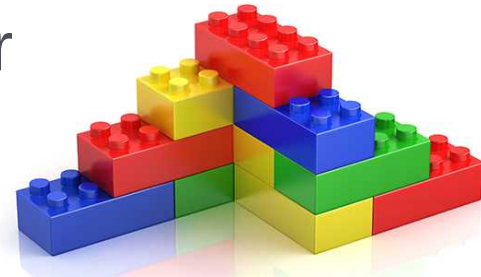
Interface here is used in the general sense of the allowed inputs to a **function**, object, program (command line/file) or GUI.
Often called an application program interface (API)

Other Examples of an Interface

- Most linux commands
 - `convert picture.png picture.jpg` (command line)
- BLAS and LAPACK (purely functional)
 - Basic matrix algebra and linear algebra packages, scipy uses these libraries extensively (faster than optimised code)
- Message Passing Interface – MPI (mostly functional)
 - Provides communication between processes on supercomputers through send and receive functions
- Objects with an abstract base class (Object Oriented)
 - Inheritance defines interface instead of providing functionality
 - Design patterns aim to codify experience

Advantages of an Interface

- Makes it very clear what your software does
- Allows tests to be designed using expected functionality
- Forces you to think carefully about modular design, like lego blocks
- Code can be redesigned or refactored with no impact provided the interface is the same
- Enables teamwork by allowing clear division of responsibility
- By limiting available functionality, you can ensure use is not outside of intended range
- More likely to be used in other peoples' software



Testing and Version Control

Unit Testing

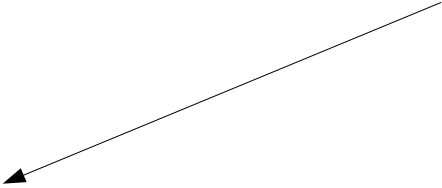
- Functions are like a contract with the user, unit tests aim to ensure these work as expected

```
function square(a)  
    return a**2  
end function
```

```
assertAlmostEqual(square(2.0), 4.0)
```

```
assertEqual(square(2), 4)
```

Assert raises an error if the logical statement is not true. Note, finite precision arithmetic so non exact

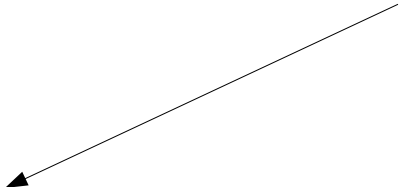


Unit Testing in Python (unittest/Pytest)

```
import unittest
```

```
def square(a):  
    return a**2
```

Required format for
unittest (test class
inherits from
unittest.TestCase
base class)



```
class test_square(unittest.TestCase):
```


```
    def test_float(self):
```

```
        self.assertAlmostEqual(square(2.), 4.)
```

```
    def test_int(self):
```


```
        self.assertEqual(square(2), 4)
```

Assert raises an error if
the logical statement is
not true. Note, finite
precision arithmetic so
non exact



```
unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

arg-is-ignored to
avoid an error in
jupyter notebooks



A minimal example with Travis CI is available

https://github.com/edwardsmith999/python_example_project

Unit Testing in C++ (gtest)

- A testing framework for C++ code which can be packaged as a set of header files

- Macros to allow easy naming of testing functions

- Range of assert test

- Build up/tear down

- Used with Google mocks

```
1  #include "gtest/gtest.h"
2  #include "gmock/gmock.h"
3  #include <math.h>
4
5  // The fixture for testing class Foo.
6  class Example_Test : public ::testing::Test {
7  protected:
8
9      Example_Test()
10     virtual ~Example_Test()
11 };
12
13 float square(float x){
14     return pow (x, 2.0);
15 }
16
17 TEST_F(Example_Test, test_square) {
18     ASSERT_DOUBLE_EQ(4.0, square(2.0));
19 }
```

- A minimal example with Travis CI is available

https://github.com/edwardsmith999/cpp_example_project

Unit Testing in Fortran (FRUIT)

- A complete package with a driver written in Ruby
- Personally I use just the fruit.f90 file from /fruit_3.4.3/src/

gfortran fruit.f90 fruit_test.f90

- Allows assert and summaries

```

1  module fns
2
3      contains
4
5      function square(x)
6          double precision :: x, square
7          square = x**2
8      end function square
9
10 end module fns
11
12 program test
13     use fns
14     use fruit
15     implicit none
16
17     double precision, parameter :: tol=1e-12
18
19     !Unit testing in Fortran
20     call init_fruit
21     call assert_equals(4.d0, square(2.d0), tol)
22     call fruit_summary
23     call fruit_finalize
24
25 end program test
26

```

- A minimal example with Travis CI

https://github.com/edwardsmith999/fortran_example_project

Version Control

- Once you have some code, put it into a code repository
 - Backup in case you lose your computer
 - Access to code from home, work and anywhere else.
 - Allows you to keep a clear history of code changes
 - Only reasonable option when working together on a code
- Three main repositories are Git, Mercurial and Subversion.
- Most common is Git while Subversion is often disregarded due to centralised model.
- Range of "free" services for hosting – Github (Imperial subscription), Bitbucket, Gitlab, CloudForge, etc
- Or you can host your own

Version Control

- Git is the most popular version control (Github a hosting site)
 - `git clone http://www.github/repo/loc ./out` Clone directory to out
 - `git log` Check history of commits
 - `git diff` Check changes made by user since last
 - `git pull` Get latest changes from origin (fetch+merge)
 - `git add` Add changes to staging area
 - `git commit -m "Log message"` Commit changes with message
 - `git push` Push changes to origin
 - `git branch` Create a branch of the code

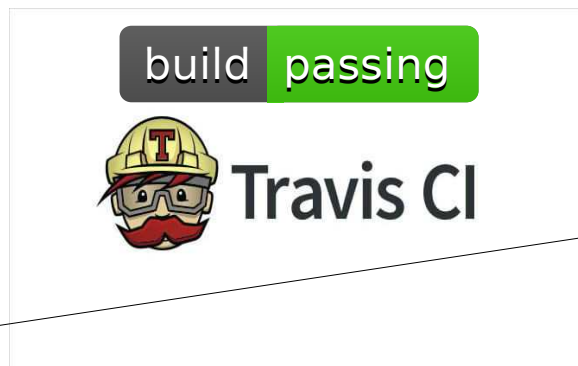


Automated Testing with Travis and

- When you commit code to version control, your tests are run automatically. If a change breaks the test, you get emailed
- This (along with deployment) is known as continuous integration

.travis.yml

```
os: linux
language: python
python:
  - 2.7
  - 3.6
script:
  - make test
```



Makefile

```
default: test
  echo "Default"
test:
  pytest test_fns.py
```

- Script here is Travis CI linked to github, free for open source
- Many other options including Jenkins, buildbot, circleCI, gitlab, ANVIL (STFC) or setup your own local solution using scripts

**BUT IN MY EXPERIENCE – IF YOU DON'T AUTOMATE IT
IT DOESN'T GET RUN!!**

Best Practice – Test Driven Development

- Work out what you want the software to do
- Write tests first to define the desired functionality – Test Driven Development (TDD)
- Develop functions or classes to pass these test scripts

Unit Testing and TDD in Python

```
import unittest
```

```
def square(a):  
    pass
```

Function initial empty
and written to satisfy
required functionality

```
class test_square(unittest.TestCase):
```

```
    def test_float(self):
```

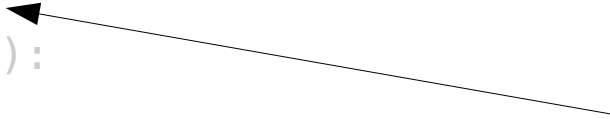
```
        self.assertAlmostEqual(square(2.), 4.)
```

```
    def test_int(self):
```

```
        self.assertEqual(square(2), 4)
```

```
unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

Assert raises an error if
the logical statement is
not true. Note, finite
precision arithmetic so
non exact




Unit Testing and TDD in Python

```
import unittest
```

```
def square(a):
```

```
    return a**2
```

Write a function
which passes both
tests



```
class test_square(unittest.TestCase):
```

```
    def test_float(self):
```

```
        self.assertAlmostEqual(square(2.), 4.)
```

```
    def test_int(self):
```

```
        self.assertEqual(square(2), 4)
```

```
unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

Unit Testing and TDD in Python

```
import unittest

def square(a):
    return a**2

def cube(a):
    pass
```

↑

Add a new test
to define the new
function cube

```
class test_square(unittest.TestCase):
    def test_float(self):
        self.assertAlmostEqual(square(2.), 4.)
    def test_int(self):
        self.assertEqual(square(2), 4)

class test_cube(unittest.TestCase):
    def test_float(self):
        self.assertAlmostEqual(cube(2.), 8.)
    def test_int(self):
        self.assertEqual(cube(2), 8)

unittest.main(argv=['first-arg-is-ignored'],
               exit=False)
```

Unit Testing and TDD in Python

```
import unittest

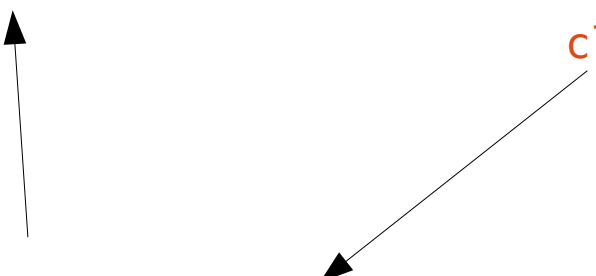
def square(a):
    return a**2

def cube(a):
    return a**3

class test_square(unittest.TestCase):
    def test_float(self):
        self.assertEqual(square(2.), 4.)
    def test_int(self):
        self.assertEqual(square(2), 4)

class test_cube(unittest.TestCase):
    def test_float(self):
        self.assertEqual(cube(2.), 8.)
    def test_int(self):
        self.assertEqual(cube(2), 8)

unittest.main(argv=['first-arg-is-ignored'],
exit=False)
```



We have written
cube to pass the
test

Testing in Scientific Code

- Much easier said than done
 - Thinking of meaningful tests is often far from easy
 - Test driven development (TDD) best with simple aims
- Academic suicide?
 - Serious investment of time; whole teams for this in companies
 - There is no reward mechanism for reliable software
- General advice now is something is better than nothing
 - Probably cannot meet software engineering standards
 - But can aim for a level of falsifiability to please Popper
 - Academics already do testing/verification
 - Need better unit-testing and automation with continuous integration (CI).

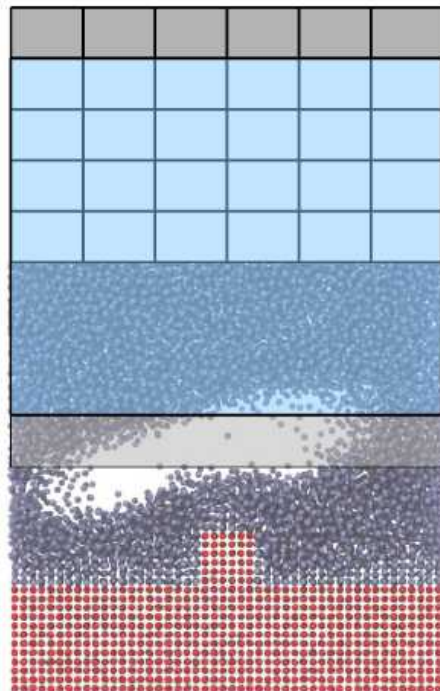
A Real Example

CPL Library

- We are coupling two separate codes to run together
 - Grid based Computational Fluid Dynamics (CFD)
 - Particles e.g. Molecular Dynamics (MD)
 - Both use MPI and require a complex setup
- CPL library is a shared library
 - Codes built separately
 - Exchange information through minimal interface of send/recv functions
- This is good because it
 - Allows separate testing of CPL library and both codes
 - Maintains scope of both codes
 - Promotes optimal scaling

CPL library

- Domain Decomposition (MD near wall, CFD for remaining domain)

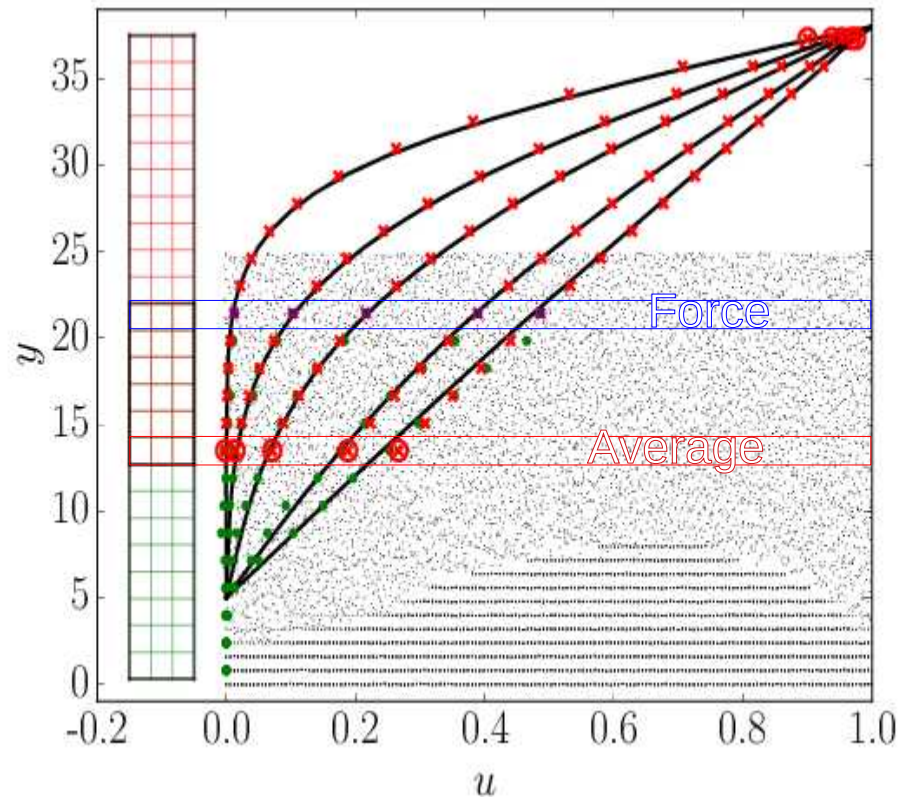


CFD
Region

Overlap
Region

MD
Region

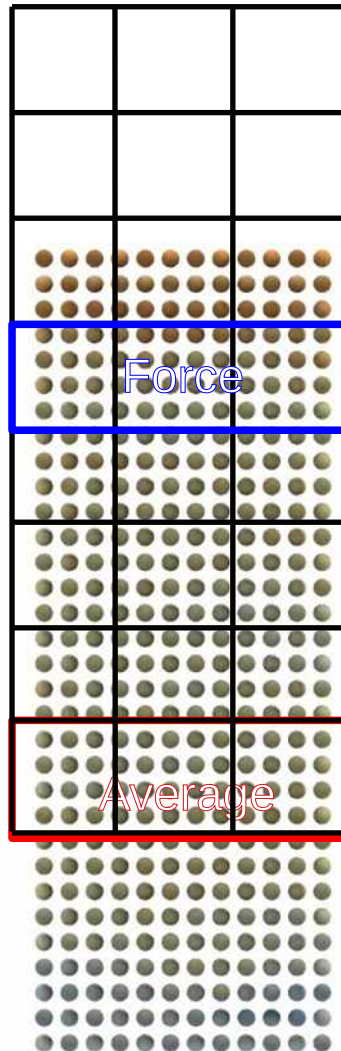
Black lines are the analytical
solution for Couette flow



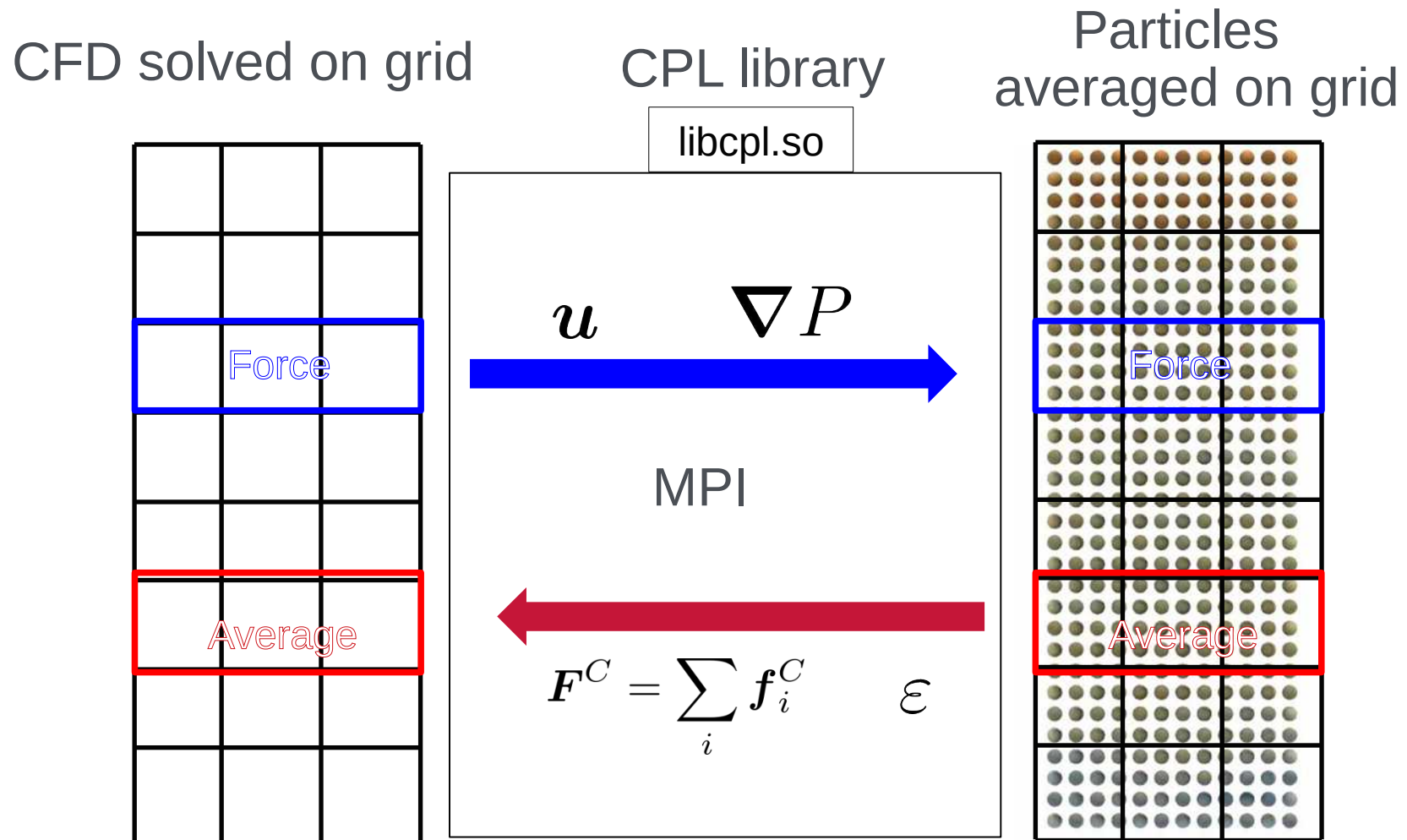
CPL LIBRARY

www.cpl-library.org

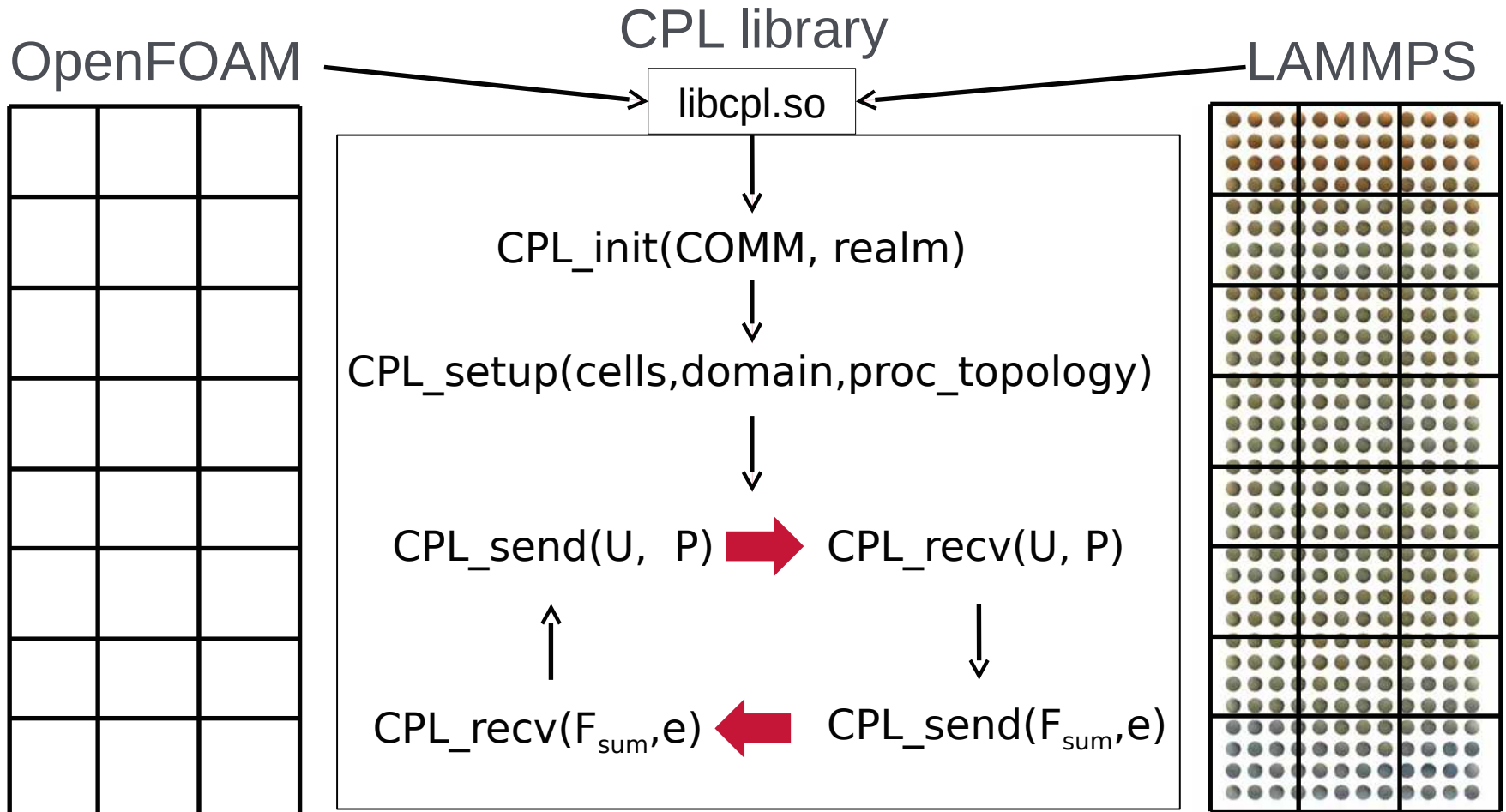
A Tale of Two Grids



A Tale of Two Grids



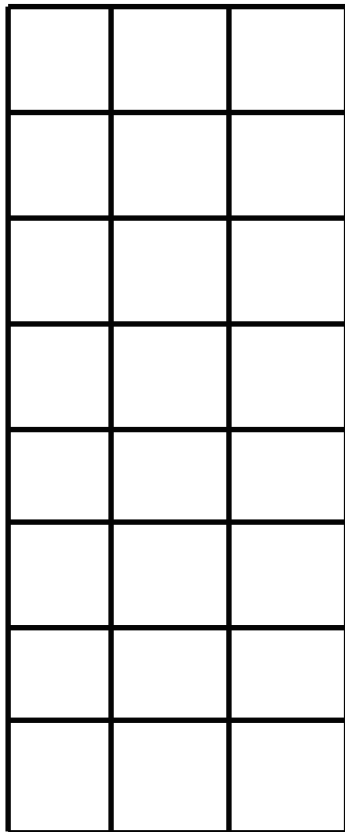
A Tale of Two Grids



Two codes sharing a communicator **`mpirun -n 4 ./cfd.exe : -n 48 ./dem.exe`**

CPL Mocks - A Tale of Two Grids

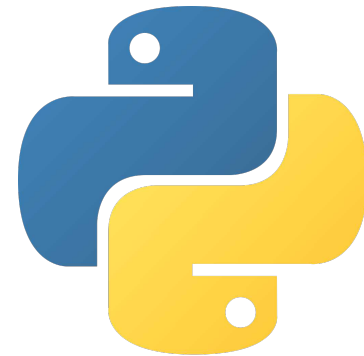
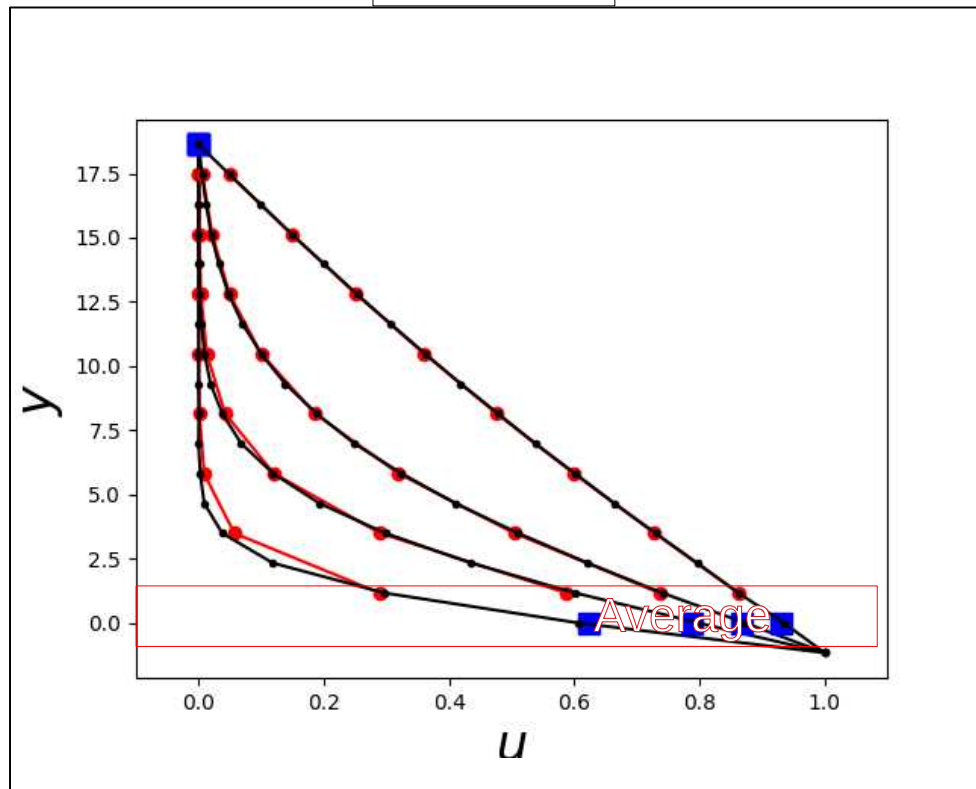
Any CFD e.g.
OpenFOAM



CPL library

libcpl.so

CPL Mock



Two codes sharing a communicator

`mpiexec -n 4 ./cfd.exe : -n 48 ./mock.py`

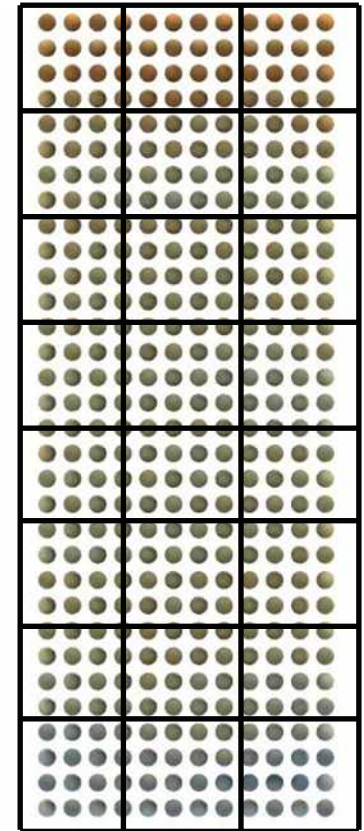
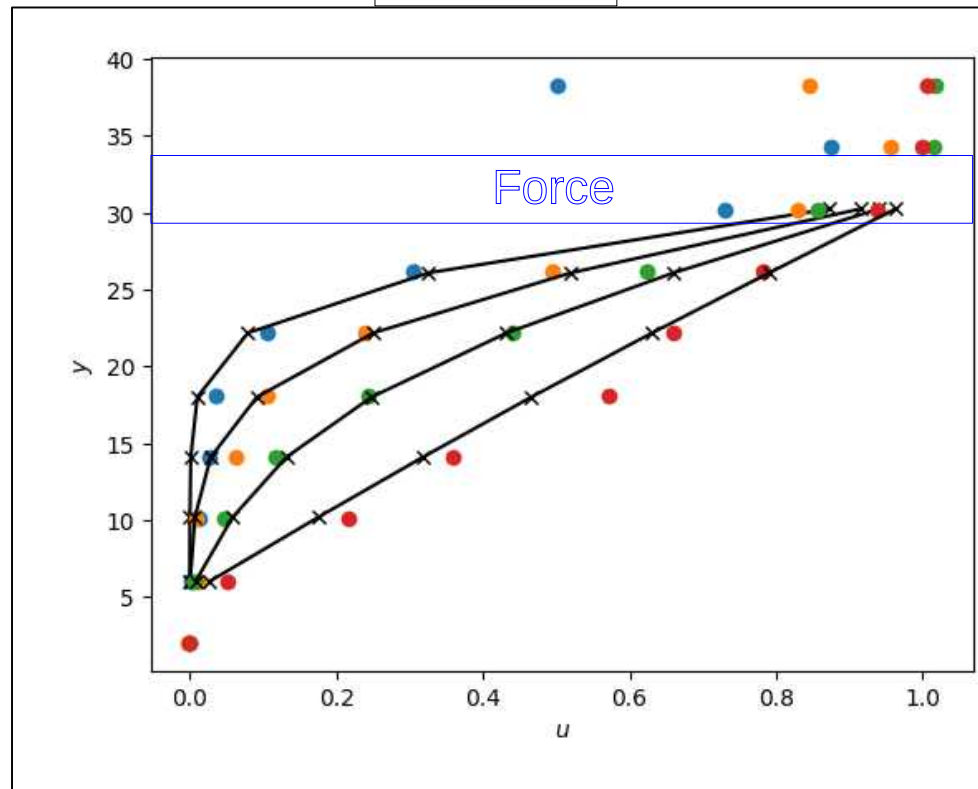
CPL Mocks - A Tale of Two Grids

CPL Mock

CPL library

libcpl.so

Any DEM e.g.
LAMMPS

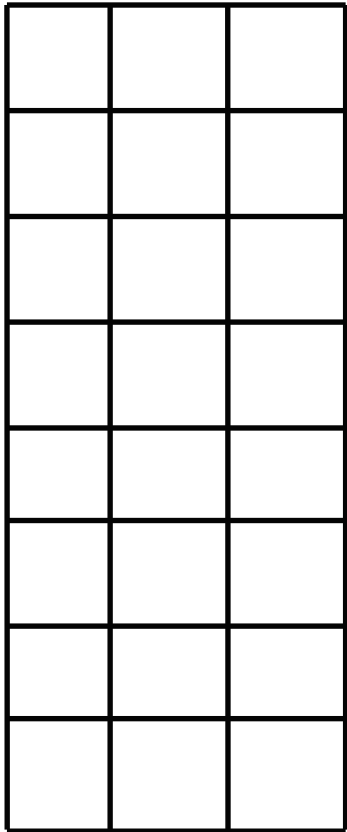


Two codes sharing a communicator

`mpirun -n 4 ./mock.py : -n 48 ./dem.exe`

A Tale of Two Grids

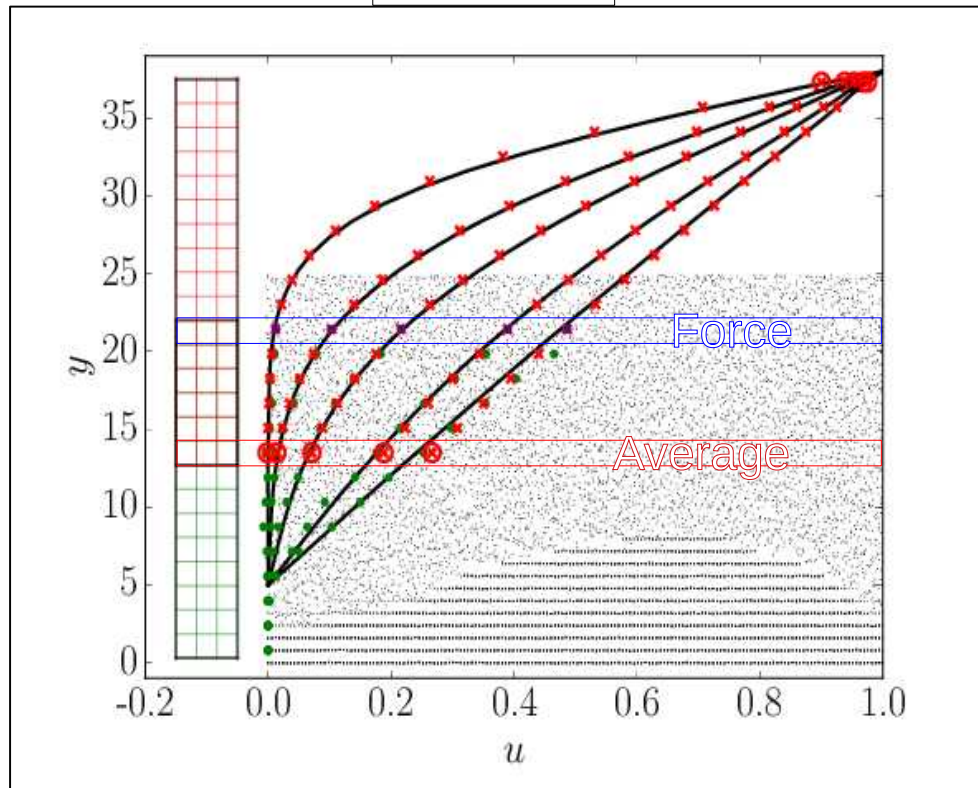
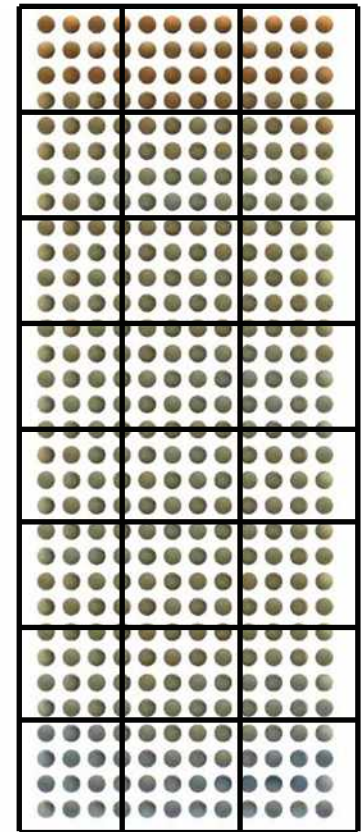
OpenFOAM



CPL library

libcpl.so

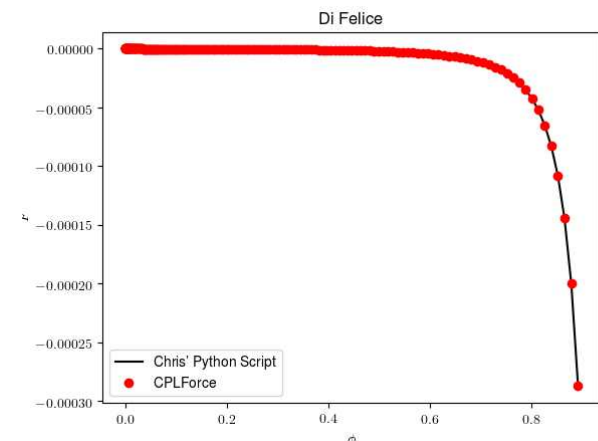
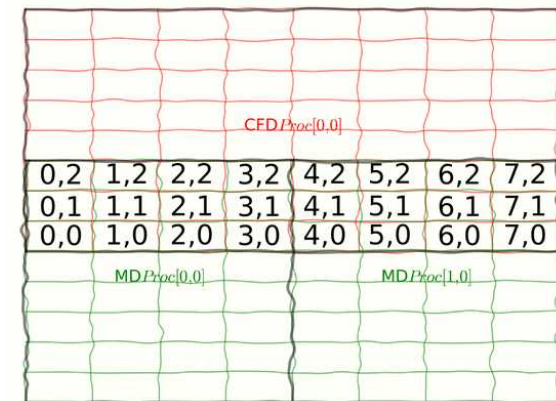
LAMMPS



Two codes sharing a communicator **`mpirun -n 4 ./cfd.exe : -n 48 ./dem.exe`**

CPL Unit Tests

- A range of possible topologies are tested (pytest)
 - Checks cell numbers for a wide range of processor topologies using MPI
 - Test unsupported cases raise expected errors
- Test mix of Fortran, C++ and Python codes (pytest)
 - Examples from website
 - Test for memory leaks with valgrind
- Range of utilities grids, fields and forces (gtest offline without MPI)
 - 4D array system in Fortran, C++ and Python
 - Methods to average particle system
 - A range of forces tested against literature



How to Test Parallel Code?

- We test parallel case on a range of different processor topologies
 - pytest parameterize to create all possibilities
 - Serial code uses subprocess (i.e. python starting another program) to create possible MPI runs with mpiexec

```
@pytest.mark.parametrize("cfdprocs, mdprocs, err_msg", [
    ((2, 2, 3), (2, 2, 3), ""),
    ((3, 2, 2), (3, 2, 2), ""),
    ((2, 3, 2), (2, 3, 2), ""),
    ((4, 4, 6), (4, 4, 6), ""),
    ((4, 6, 4), (4, 6, 4), ""),
    ((6, 4, 4), (6, 4, 4), "")])
def test_mapcells(prepare_config_fix, cfdprocs, mdprocs, err_msg):
    MD_PARAMS = {"lx": 24.0, "ly": 24.0, "lz": 24.0,
                  "which_test": "cell_test"}
    MD_PARAMS["npx"], MD_PARAMS["npz"] = mdprocs
```


Automated Testing

- Travis CI script is slightly more complex but ensures both build (on fresh linux) and test works as expected

```
# http://travis-ci.org/Crompulence/cpl-library
os: linux
sudo: required
language: python
python:
  - 2.7
env:
  - MPI=mpich3 GCC_VERSION=5
before_install:
  - sh ./make/travis/travis-install-gcc.sh
  ...
  - export MPI_DIR=$MPI_BUILD_DIR/$MPI
install:
  - export PATH=$MPI_DIR/bin:$PATH
  - make
script:
  - make test-all
```

build passing



Travis CI

build failing

Deployment Through Docker/Singularity

- Use Dockerfile and convert to singularity as more supported and Docker is clearer to me than the singularity scientific file system
 - Start from a container with choice of linux, I use Ubuntu 16.04
 - Script with each command adding another layer (group each RUN)

start from base

```
FROM ubuntu:16.04
MAINTAINER Edward Smith
<edward.smith05@imperial.ac.uk>
```

#Install compilers, mpi (with ssh)

```
RUN apt-get update && apt-get install -y \
    gcc \
    gfortran \
    git-core \
    build-essential \
    mpich \
    openssh-server \
    ...
```

#Clone code from github

```
RUN git clone
https://github.com/Crompulence/cpl-
library.git /cpl-library
WORKDIR /cpl-library
```

#Install CPL library

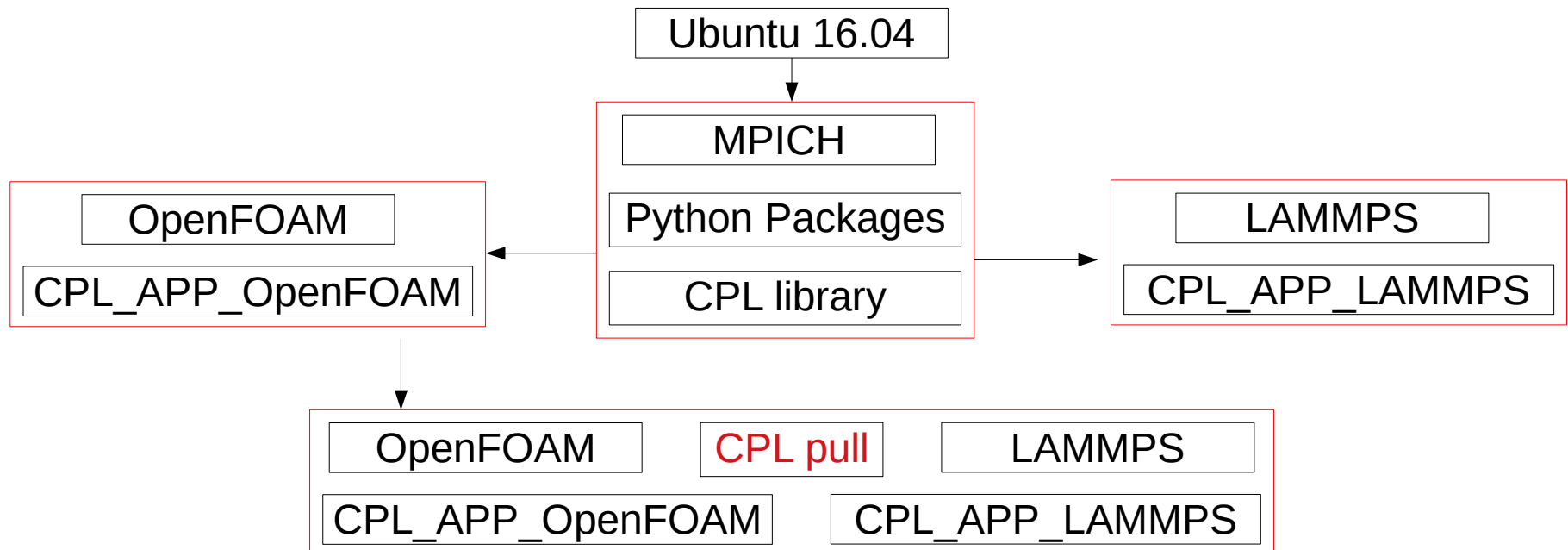
```
RUN make PLATFORM=gcc
```

#Add to the path

```
ENV CPL_PATH=/cpl-library
ENV CPL_BIN_PATH="$CPL_PATH/bin"
ENV PATH=${CPL_BIN_PATH}:$PATH
```

Deployment Through Docker/Singularity

- Use Dockerfile and convert to singularity as more supported and Docker is clearer to me than the singularity scientific file system
- Start from a container with choice of linux, I use Ubuntu 16.04
- Script with each command adding another layer (group each RUN)
- Rebuild when github changes automated using DockerHub



Running Coupled Docker or Singularity

- The use of containers for coupling allows a variety of run options, with **Docker** you start a container

```
sudo docker run -it --name cplrun cpllibrary/cpl-openfoam-lammps
```

- then run coupled cases inside the container which has inputs files

```
mpirun -n 1 CPLSediFOAM -case ./openfoam/ -parallel &  
mpirun -n 1 Imp_cpl < lammps/fcc.in
```

- With **Singularity**, mpirun is outside the container so we start an executable in each container and they couple through MPI

```
singularity pull docker://cpllibrary/cpl-openfoam-lammps
```

- executables from container, input files from local directory

```
mpirun -n 1 singularity exec cpl-openfoam-lammps.simg \  
CPLSediFOAM -case ./openfoam/ -parallel &  
mpirun -n 1 singularity exec cpl-openfoam-lammps.simg \  
Imp_cpl < lammps/fcc.in
```

More detail: www.cpl-library.org/wiki

Frustrations – Testing and Deployment

- Much more time needed to develop unit tested software
 - Maintenance burden of tests can be more than code
 - Only as good as the tests you can devise
 - Relies on everyone fixing broken builds as they occur
- Issue of deployment still tricky as MPI is a tough dependency and we must build OpenFOAM and LAMMPS (with patching)
 - Building from source is different on each platform (especially supercomputers with old libc)
 - Docker and Singularity work only if fully embraced by users

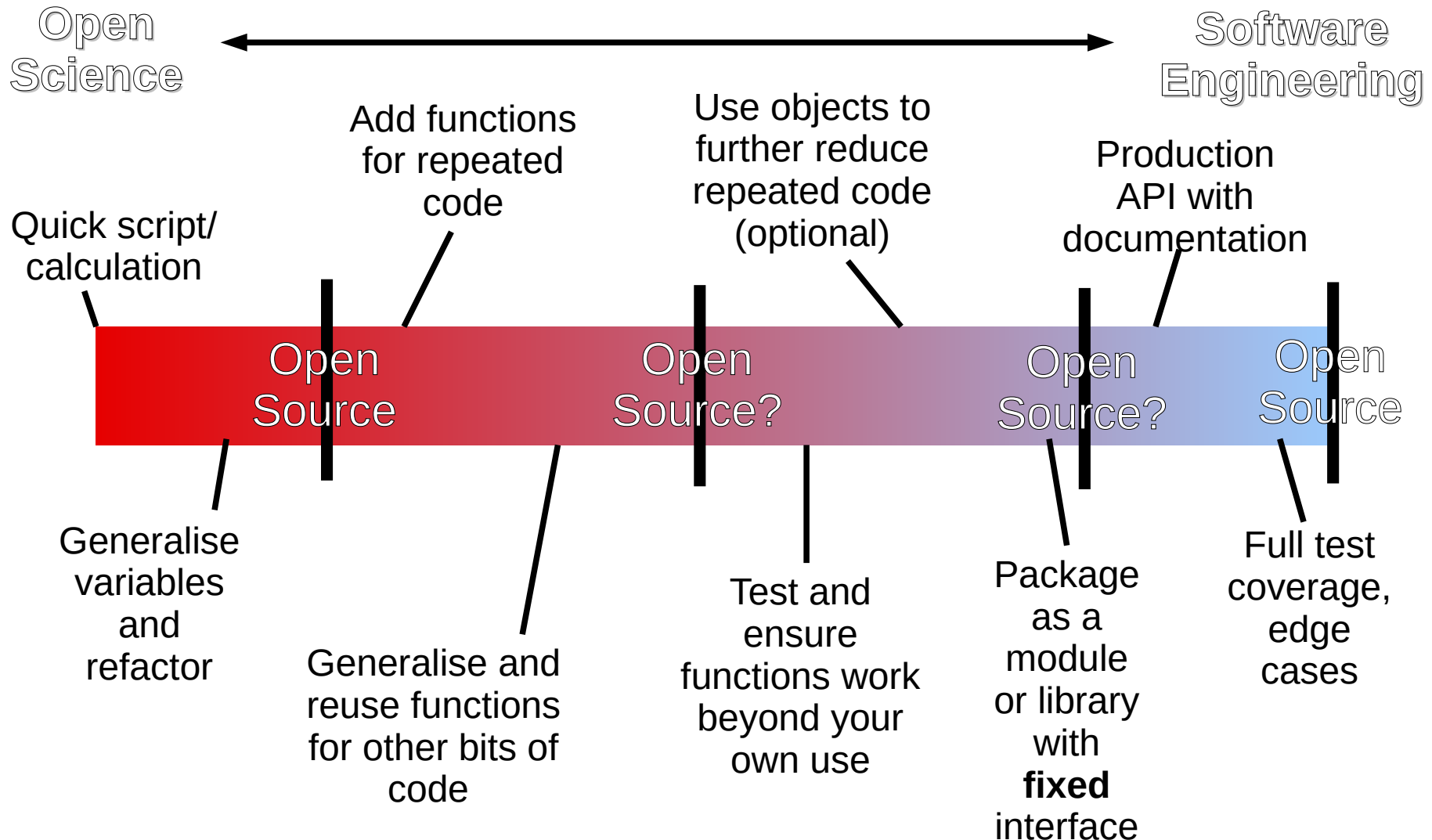
Frustrations - Tools

- Git for academic workflow may be overkill
 - Previously used subversion, fine for a few people
 - Scientific workflow often involve many computers (work, home, cx1, cx2, archer, etc..), branches add to this complexity and you end up centralised anyway
- Supporting Dockerhub and Travis continuous deployment is a full time job
 - You need to support changes for any upstream packages, if a repository is offline, can trigger an error
 - Travis as a solution is frustrating – cannot run locally so often waste time trying to figure a simple error by committing to github

Frustrations - Science

- Failed to deliver a scientific results
 - Run out of time, need results, collaborators now using semi-commercial code which gives clear results
 - The problem is not software but the science, so little time spent on this (c.f. hashing out code)
 - Scientific prototype may be better, then RSE rewriting?
- Scientific problems have physical validation cases
 - Unit test are less meaningful if the whole model works?
 - Chaotic behaviour means repeatability often impossible
 - Testing noisy data vs. analytical solutions is imprecise and very difficult to automate in a non-brittle way

Pin the Open Source on the Project



Summary

- Research Software Engineering
 - Introduction
 - Programming to an interface
- Testing and version control
 - Minimal examples of testing an interface
 - TDD and continuous integration
- Examples from my work with CPL library
 - Quick overview of what it does/challenges
 - Unit testing and deployment examples
- Frustrations and discussion points

Questions/Discussion

- How much time should be invested in software engineering for an academic project? Led by PI?
- When should code be made open source? To validate scripts (scientific) or only when very polished (RSE)
- Is a CI Github/Travis workflow the best one? What about local testing or CI on supercomputers?
- Is Docker or singularity (now commercial) a good solution for deployment, or is there a better one?

Extra Slides

CFD (OpenFOAM)

- Libraries of field objects which perform differential operations on themselves (you write your own solver)

```

Info<< "\nStarting time loop\n" << endl;
while (runTime.loop())
{
    Info<< "Time = " << runTime.timeName() << nl << endl;
    #include "CourantNo.H"

    // Momentum predictor

    fvVectorMatrix UEqn
    (
        fvm::ddt(U)
        + fvm::div(phi, U)
        - fvm::laplacian(nu, U)
    );

    if (piso.momentumPredictor())
    {
        solve(UEqn == -fvc::grad(p));
    }

    // --- PISO loop
    while (piso.correct())
    {
        volScalarField rAU(1.0/UEqn.A());
        volVectorField HbyA(constrainHbyA(rAU*UEqn.H(), U, p));
        surfaceScalarField phiHbyA
        (
            "phiHbyA",
            fvc::flux(HbyA)
            + fvc::interpolate(rAU)*fvc::ddtCorr(U, phi)
        );
        adjustPhi(phiHbyA, U, p);

        // Update the pressure BCs to ensure flux consistency
        constrainPressure(p, U, phiHbyA, rAU);

        // Non-orthogonal pressure corrector loop
        while (piso.correctNonOrthogonal())
        {
            // Pressure corrector

            fvScalarMatrix pEqn
            (
                fvm::laplacian(rAU, p) == fvc::div(phiHbyA)
            );
            pEqn.setReference(pRefCell, pRefValue);
            pEqn.solve(mesh.solver(p.select(piso.finalInnerIter())));
        }
    }
}

```

Momentum predictor: momentum equation is solved with old values of pressure

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot (\mathbf{U}\mathbf{U}) - \nabla \cdot (\nu \nabla \mathbf{U}) = -\nabla p$$

$$rAU: \frac{1}{a_p} \quad \text{Extracts the diagonal coefficients of the matrix for } \mathbf{U}$$

$$HbyA: \frac{\mathbf{H}(\mathbf{U})}{a_p} \quad \text{Performs the product between off-diagonal matrix and diagonal}$$

$$\phi HbyA: \left(\frac{\mathbf{H}(\mathbf{U})}{a_p} \right)_f \cdot \mathbf{S}_f \quad \text{Interpolates HbyA from cell centers to face centers and evaluates fluxes}$$

$$\nabla \cdot \left(\frac{1}{a_p} \nabla p \right) = \nabla \cdot \left(\frac{\mathbf{H}(\mathbf{U})}{a_p} \right)_f \quad \text{Defines Poisson equation for pressure}$$

Shifts pressure values to match **pRefValue** at **pRefCell** (see fvSolution)

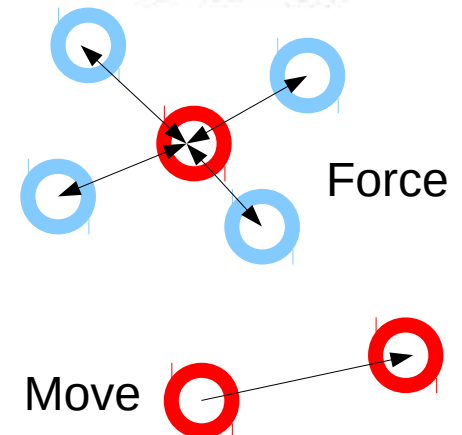
Solves Poisson equation for pressure

from notes by
Mirco Maginini

MD (LAMMPS)

- Solve Newton's laws for N interacting molecules
 - Add up all forces F_i on mol i
 - Move i by integrating $F_i = m_i a_i$
- LAMMPS uses "hooks"
 - User additions are designed as an object with set interface
 - Functions `pre_force` or `end_of_step` can be defined

```
loop over N timesteps:  
    fix->initial_integrate()  
    fix->post_integrate()  
    fix->pre_exchange()  
    domain->pbcb()  
    comm->exchange()  
    fix->pre_force()  
    pair->compute()  
    fix->post_force()  
    fix->final_integrate()  
    fix->end_of_step()
```



Deployment Through Anaconda

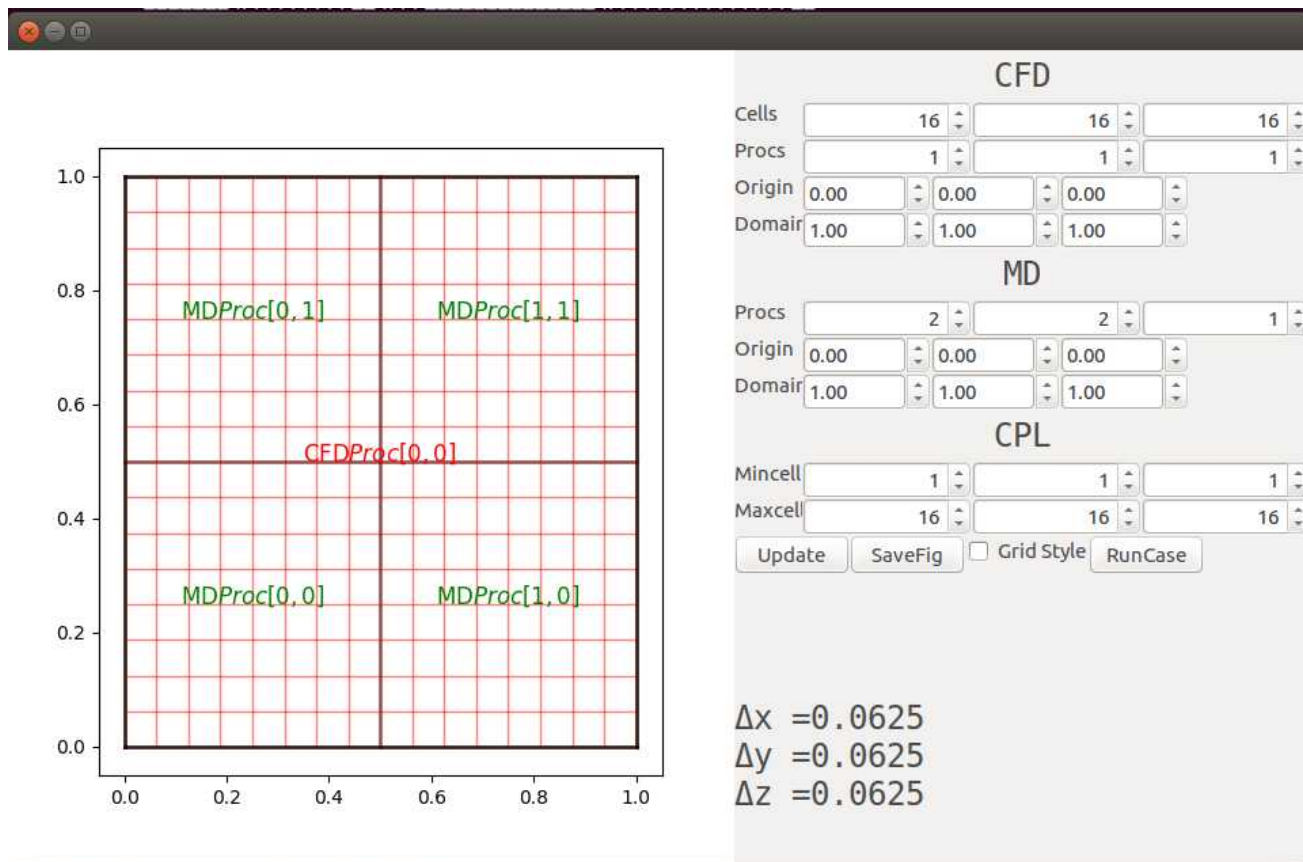
- A package manager for scientific computing and Python
- OpenFOAM takes 8+ hrs to build from source
- Packaged on virtual machine for compatibility with linux on ARCHER, CX1, CX2 and supercomputer in Texas

```
conda_install.sh x
1 wget https://repo.continuum.io/miniconda/Miniconda2-latest-Linux-x86_64.sh
2
3 MYPWD=${PWD}
4 bash Miniconda2-latest-Linux-x86_64.sh -bf -p $MYPWD/miniconda
5 export PATH=$PATH:$MYPWD/miniconda/bin
6
7 conda create -n cplrun python=2.7 --no-default-packages
8 source activate cplrun
9 conda install -c edu159 -y cplpy
10 conda install -c edu159 -y cplapp-openfoam
11 conda install -c edu159 -y cplapp-lammpsdev
12 source $CONDA_PREFIX/opt/OpenFOAM-3.0.1/etc/bashrc
13
```

This work is mainly by Eduardo Ramos-Fernandez

Test Different Topologies / Communication

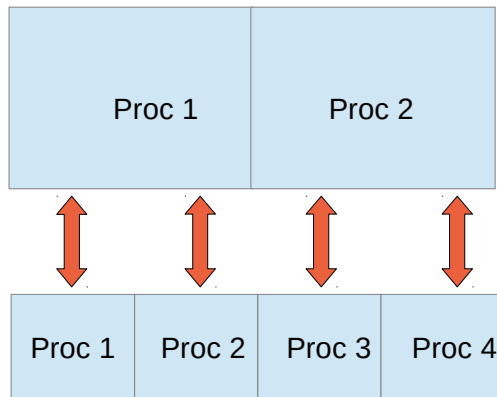
- See also `utils/design_topology/` for a gui to try your own, run with `python cpl_gridsetup.py`



Scaling on Supercomputers

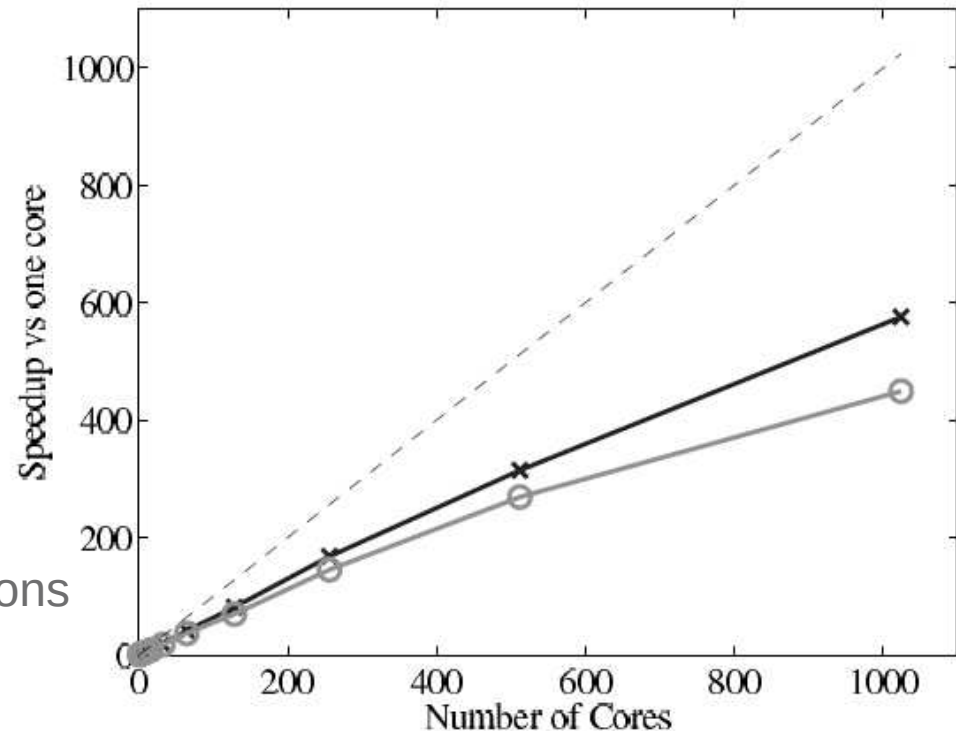
Developed for linking of particle and continuum code

Previous focus on scalability
(for supercomputers)



All MPI
communications
are local

Maintains separate scope of
each code by linking shared
library



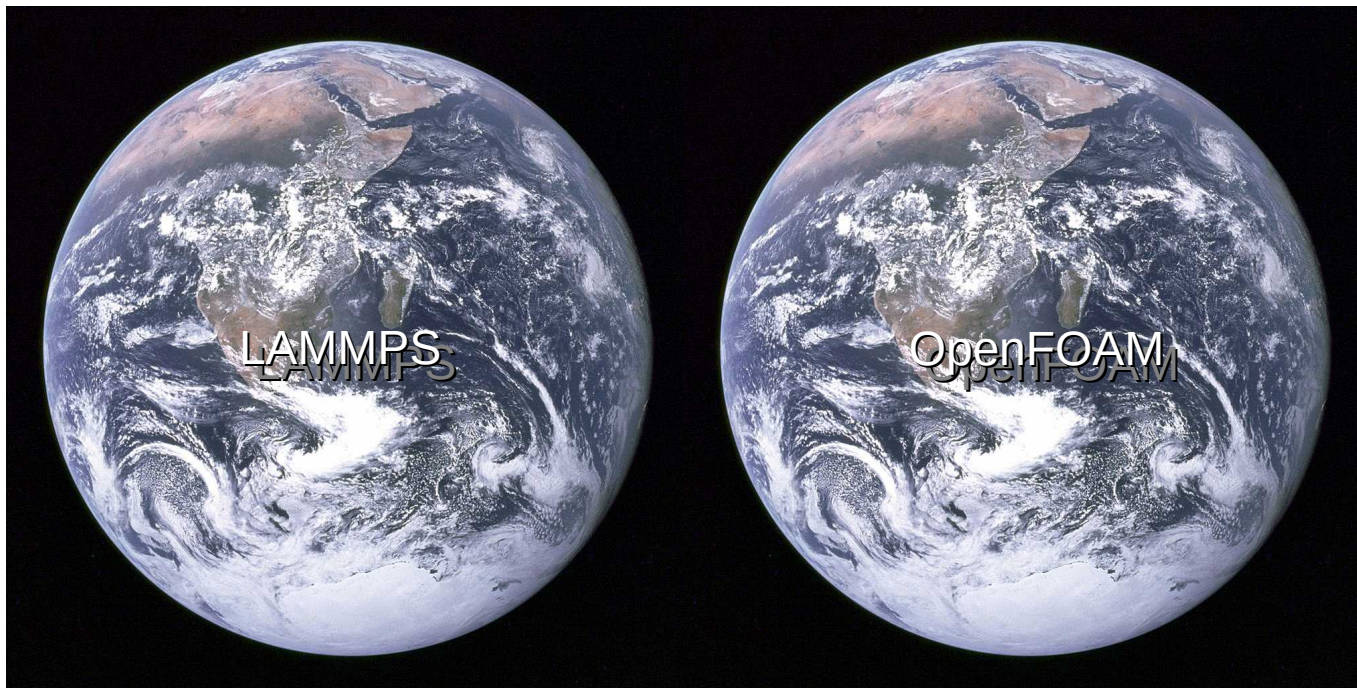
(a) Parallel speedup of the MD solver only (x), coupled code (o) against the ideal speedup (—)

Weak scaling

- Particle only x
- Particle Coupled o

MPI

- Communicators are used to determine which processes communicate
 - `MPI_send(data, size, MPI_COMM)`
- The default that contains all communicators is `MPI_COMM_WORLD`
- Fine to use this as long as there is only ever one world



Minimal Examples Online

https://github.com/edwardsmith999/python_example_project

Branch: master New pull request Create new file Upload files Find file Clone or download

edwardsmith999 committed on GitHub Update README.md Latest commit 9a9a464 3 minutes ago

.travis.yml	Removed Python 2.6 from tests	6 minutes ago
Makefile	Py.test causes weird import bug due to __init__ in folder, added make...	8 minutes ago
README.md	Update README.md	3 minutes ago
__init__.py	Minimal example of functions, Number class and travis test	16 minutes ago
number_class.py	Minimal example of functions, Number class and travis test	16 minutes ago
number_fns.py	Minimal example of functions, Number class and travis test	16 minutes ago
test_number_class.py	Minimal example of functions, Number class and travis test	16 minutes ago
test_number_fns.py	Minimal example of functions, Number class and travis test	16 minutes ago

- `git clone https://github.com/edwardsmith999/python_example_project.git`
- `git clone https://github.com/edwardsmith999/cpp_example_project.git`
- `git clone https://github.com/edwardsmith999/fortran_example_project.git`

An example with an Object

- A number class which includes methods to get square and cube

```
class Number():  
    def __init__(self, a):  
        self.a = a  
    def square(self):  
        pass  
    def cube(self):  
        pass
```

Testing an Object

```
import unittest

class Number():
    def __init__(self, a):
        self.a = a
    def square(self):
        pass
    def cube(self):
        pass

    Desired class functionality is DEFINED by the tests

class test_number(unittest.TestCase):
    def test_float(self):
        n = Number(2.)
        self.assertEqual(n.square(), 4.)
        self.assertEqual(n.cube(), 8.)
    def test_int(self):
        n = Number(2)
        self.assertEqual(n.square(), 4)
        self.assertEqual(n.cube(), 8)

unittest.main(argv=['first-arg-is-ignored'],
                exit=False)
```

Class methods empty
and must be written to
pass tests

Testing an Object

```
import unittest

class Number():
    def __init__(self, a):
        self.a = a
    def square(self):
        return self.a**2
    def cube(self):
        return self.a**3

    Desired class functionality is DEFINED by the tests

class test_number(unittest.TestCase):
    def test_float(self):
        n = Number(2.)
        self.assertEqual(n.square(), 4.)
        self.assertEqual(n.cube(), 8.)
    def test_int(self):
        n = Number(2)
        self.assertEqual(n.square(), 4)
        self.assertEqual(n.cube(), 8)

unittest.main(argv=['first-arg-is-ignored'],
                exit=False)
```

Class methods written in order to satisfy required functionality