A gentle introduction to OMUSE: A Python framework for multiphysics simulations in Oceanography

Inti Pelupessy\textsuperscript{1,2} Ben van Werkhoven\textsuperscript{3} Arjen van Elteren\textsuperscript{2} Jan Viebahn\textsuperscript{4} Adam Candy\textsuperscript{5} Simon Portegies Zwart\textsuperscript{2} Henk Dijkstra\textsuperscript{1}

\textsuperscript{1}IMAU Utrecht \textsuperscript{2}Leiden Observatory \textsuperscript{3}NLeSc Amsterdam \textsuperscript{4}CWI Amsterdam \textsuperscript{5}TU Delft

IMUM 2016, 27-30 September 2016, OMP, Toulouse
What is OMUSE?

→ Oceanographic Multi-PURpose Software Environment
→ OMUSE is a Python environment for oceanographic numerical experiments

Goals:
- provide a homogeneous enviroment to run community codes
- enable new code couplings and interactions between components
- facilitate multi-physics and multi-scale simulations
Why OMUSE?

many excellent oceanographic codes have been written, so why OMUSE?

traditional monolithic codes present challenges:
- difficult to learn & use,
- difficult to maintain and adapt,
- difficult to couple with other models,
- difficult to extend with new physics

so why not build on the legacy of the oceanographic community and build a toolbox using existing codes?
History of OMUSE

- OMUSE build on AMUSE, started in the MODEST community

- development of predecessor and prototype around 2006: MUSE
- MUSE features retained in AMUSE: python based, 4 domains
- around 2009: more formal development started with funding from NOVA and later NWO,
- main development team in Leiden
- actively being used by 15+ groups worldwide
- 30+ publications, 8+ theses
History of OMUSE

- 2012 – 2013: discussions on wider applicability, call from the NLeSc for interdisciplinary projects, interest from Henk Dijkstra

- 2014: start of development @IMAU of OMUSE with funding from NLeSc, OMUSE main developers: Pelupessy (IMAU) & van Werkhoven (NLeSc)

- 2015 – 2016: current development of prototype & initial capability
from omuse.units import units
from omuse.community.qgmodel.interface import QGmodel
from amuse.io import read_set_from_file

input = read_set_from_file('initial_condition')

code = QGmodel()

code.parameters.dt = 0.5 | units.hour

code.grid.psi = input.psi

code.evolve_model(1. | units.day)

print code.grid.psi.max().in_(units.Sv/units.km)
OMUSE interface design
AMUSE & OMUSE design highlights

- python based:
  algorithmic flexibility and ease of programming
- remote function interfaces:
  built-in parallelism & separation of memory space, thread safety
- unit algebra module: units imposed
- automatic state handling
- object oriented interfaces
- error handling & stopping conditions
- testing integral part of AMUSE development:
  2000+ tests covering the base framework, support libraries and the community interfaces (>80% code coverage),
- test suite run daily on different (virtual) machines
Current status of OMUSE

initial set of codes currently in OMUSE:
- **QGmodel**: solves barotropic vorticity equation on rectangular cartesian grid
- **ADCIRC**: shallow water coastal model, solves 2D or 3D momentum equations
- **SWAN**: wave propagation model, implicit, solves spectral action balance equation
- **POP**: solves three-dimensional primitive equations for ocean dynamics
- **QGCM**: multi-layer QG solver, atmosphere + ocean
- under consideration: XBEACH, SELFE, Delft3D, ...
Current status of OMUSE

Development of support code:

- AMUSE framework support for different grid types
- grid transformations (e.g. dipolar, tripolar)
- remapping schemes
- triangulate package (building unstructured meshes)
- importers for netcdf data etc
- integrion of plotting libraries
- ext (utility functions, ..)
- unit support for 'oceanographic' specific units
OMUSE interface design

The interface to a code defines the way you talk to a code from python:

- interfaces are based on *physics* rather than *numerics*
- codes from the same domain use the *same interface*
- communicate *objects* rather than arrays
- *impose* the use of units
- model calling sequence in *state model*
- function calls are *remote*
- *stopping conditions* to detect events and guard integrity of the simulation results
example: Quasi-geostrophic model

- qgmodel code (Viebahn 2014), solves barotropic vorticity equation:

$$\frac{\partial}{\partial t} \nabla^2 \psi + J(\psi, \nabla^2 \psi) + \beta_0 \frac{\partial \psi}{\partial x} = \frac{1}{\rho_0 H} \left( \frac{\partial \tau_y}{\partial x} - \frac{\partial \tau_x}{\partial y} \right) - R_H \nabla^2 \psi + A_H \nabla^4 \psi$$

easy example, because:
- small number of variables and parameters
- simple, fast solver
- regular cartesian grid
Quasi-geostrophic model

brief steps of implementation of the QGModel interface:
→ make code library
→ define interface: parameters, model setters and getters, units
→ rewrite main into evolve_model
→ define state model & grid variables
→ write tests!

extra steps:
- change hardcoded wind model → interface wind
- add interface boundary conditions
- add fishpack Poisson solver (for portability)
Datamodel: Grid support

- OMUSE uses high level objects to describe state of a system: grids and particles sets
- these can reference memory storage, disk storage or the state of a community code
Datamodel: Grid remapping

→ abstraction for data transport: channels

(normal copy): channel.copy_attribute("density")

(functional transforms): channel.transform( target, function, src)

takes input attributes and transforms to (different) target attributes

remapping channels:
- remaps values between grids using a remapper object
- various remappers available:
  - interpolate, conservative
  - same semantics for usage.
What can you do with OMUSE?

- simplify setup and model runs,
- scripting simulations:
  - parameters searches
  - optimizations (e.g. MCMC)
  - event detection,
  - stoppage conditions
- 'online' data analysis
- cross verification: running problems with different codes and method
- coupling different codes to construct new solvers
Coupling codes in OMUSE
- the community code interfaces define a simple and homogeneous way of running codes,
- the interface provides read+write access to the state, forcings, boundary conditions etc. of a running code with very little overhead,
- code state is kept consistent by the interface,

→ the interface can be used to implement (explicit) couplings between different codes.
  + couplings can be formulated efficiently
  + couplings can be defined in a code agnostic way
  + coupling between codes running on different machines
  + easy to set up such that coupled code conform to interface spec.
- overhead of framework calls
ADCIRC/ SWAN: Hurricane Gustave example

data from: www.caseydietrich.com
(1) channel1=hurricane.grid.new_channel_to( swan.forcings )
( ) channel2=hurricane.grid.new_channel_to( adcirc.forcings )
( ) channel3=adcirc.nodes.new_channel_to( swan.forcings )
( ) channel4=swan.nodes.new_channel_to( adcirc.forcings )
(2) while time<tend:
(3) hurricane.evolve_model(time+dt/2)
(4) channel1.copy_attributes(["tau_x","tau_y"])
( ) channel2.copy_attributes(["vx","vy"])
(5) adcirc.evolve_model(time+dt/2)
( ) swan.evolve_model(time+dt/2)
(6) channel3.copy_attributes(["current_vx","current_vy"])
( ) channel4.copy_attributes(["wave_tau_x","wave_tau_y"])
in short, OMUSE...

easy to use:
- effortless using of different codes
- automation of unit conversions, state handling
- no learning different I/O formats, parameter files, etc

encourages reproducability:
- open source policies
- easy cross verification across different codes and numerical methods
- low barrier for communication of experiments: portable scripts
OMUSE distribution:

- source repository, soon also binary release:
  
  bitbucket.org/omuse/omuse

repository contains OMUSE specific code and open source community codes (all except ADCIRC)

- example script repository:
  
  bitbucket.org/omuse/omuse-examples

- AMUSE framework:
  
  www.amusecode.org

Code papers: Pelupessy et al. 2016, under GMD discussion
(http://www.geosci-model-dev-discuss.net/gmd-2016-178/)
Pelupessy et al. 2013, A&A 557, 84