

Underworld2 Cheat Sheet

Where to find us

Underworld homepage & blog:

<http://www.underworldcode.org/>

Underworld codebase:

<https://github.com/underworldcode/underworld2>

Issue tracker:

<https://github.com/underworldcode/underworld2/issues>

Follow us on Facebook! :

<https://www.facebook.com/underworldcode/>

Useful Docker Commands

Launch a container running Jupyter notebook instance. Port 8888 is published to make the notebook available from host browser (usually at <http://localhost:8888>):

```
$ docker run -p 8888:8888 underworldcode/underworld2
```

Run an Jupyter notebook mapping the current directory to a container directory (/workspace/user_data), and also publishing port 8888:

```
$ docker run -v $PWD:/workspace/user_data -p 8888:8888 \
    underworldcode/underworld2
```

Run an Underworld script stored in your current local directory:

```
$ docker run -v $PWD:/workspace underworldcode/underworld2 \
    python model.py
```

Run a local Underworld script in parallel:

```
$ docker run -v $PWD:/workspace underworldcode/underworld2 \
    mpirun -np 2 python model.py
```

Update your Underworld2 image:

```
$ docker pull underworldcode/underworld2
```

Note: Ctrl-\ can usually be used to terminate a running docker instance.

Jupyter Notebooks

Enter: Enter edit mode.

Esc: Leave edit mode.

Cursor Up/Down: Move to the above/below cell (when not in edit mode).

Ctrl-Enter: Execute current cell.

Shift-Enter: Execute current cell and move to next.

Tab: (After you've commenced typing) Autocomplete.

Shift-Tab: (When inside function parenthesis) List function parameters.

Mesh and MeshVariables

Mesh and MeshVariable objects form the basis for numerical PDE solutions.

uw.mesh.FeMesh_Cartesian:

This class generates a regular cartesian mesh.

Useful methods:

add_variable(): Add a MeshVariable to the mesh.

data: (Property) Access mesh vertex coordinate data.

deform_mesh(): (Context manager) Deform mesh within this manager.

reset(): Reset the mesh to its original configuration.

specialSets: (Dict) Dictionary of special vertex sets associated with the mesh.

uw.mesh.MeshVariable:

The MeshVariable class adds data at each vertex of the mesh.

Useful methods:

data: (Property) Access variable data.

Example:

```
import underworld as uw
mesh = uw.mesh.FeMesh_Cartesian(elementRes=( 4, 4 ),
                                  minCoord=( 0., 0. ),
                                  maxCoord=( 1., 1. ))

meshvar = mesh.add_variable(1)
meshvar.data[:] = 0. # initialise data to zero
with mesh.deform_mesh(): # deform mesh
    mesh.data[0] = (-0.1,-0.1)
```

Swarms and SwarmVariables

Swarms define arbitrarily located points which may be used to define complex geometries for your dynamics.

uw.swarm.Swarm:

The Swarm class provides a container for particles.

Useful methods:

add_particles_with_coordinates(): Populate the swarm using provided coordinates array.

add_variable(): Adds a SwarmVariable to the swarm. Returns the SwarmVariable object.

data: (Property) Handle to the swarm's particle coordinates SwarmVariable.

deform_swarm(): (Context manager) Explicitly move swarm particles within this manager.

particleGlobalCount: (Property) Returns the swarm global particle count.

particleLocalCount: (Property) Returns the swarm local particle count.

populate_using_layout(): Populate the swarm globally using provided layout object.

uw.swarm.SwarmVariable:

The SwarmVariable class adds data to each particle. Note that you will usually create swarm variables via the **add_variable()** method on your swarm object.

Useful methods:

data: (Property) Access the swarm variables underlying data.

Example:

```
import underworld as uw
mesh = uw.mesh.FeMesh_Cartesian()
swarm = uw.swarm.Swarm(mesh)
svar = swarm.add_variable("int",1)
layout = uw.swarm.layouts.PerCellSpaceFillerLayout(swarm,20)
swarm.populate_using_layout(layout)
```

Systems and Conditions

The systems and conditions modules houses PDE related classes.

uw.systems.SteadyStateHeat:

This class implements FEM to constructs an SLE representation of a steady state heat equation of the form:

$$\nabla(k\nabla)T = h$$

uw.systems.Stokes:

This class implements FEM to constructs an SLE representation of a Stokes type system of the form:

$$\tau_{ij,j} - p_{,i} + f_i = 0$$

uw.systems.Solver:

This class returns a solver appropriate for the provide SLE object.

Useful methods:

solve(): Solve the system.

uw.systems.AdvectionDiffusion:

This class constructs an SUPG implementation of an Advection-Diffusion type system of the form:

$$\frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla \phi = \nabla(k\nabla \phi)$$

Useful methods:

get_max_dt(): Returns a CFL type timestep size.

integrate(): Integrate forward in time for provided time interval.

uw.systems.SwarmAdvecter:

This class implements a time integration scheme to advect swarm particles using a provided velocity.

Useful methods:

get_max_dt(): Returns a CFL type timestep size.

integrate(): Integrate forward in time for provided time interval.

uw.conditions.DirichletCondition:

This class implements a Dirichlet condition at the specified nodes.

uw.conditions.NeumannCondition:

This class implements a Neumann condition at the specified nodes.

Example:

```
import underworld as uw
mesh = uw.mesh.FeMesh_Cartesian()
temp_var = mesh.add_variable( 1 )
bISet = mesh.specialSets["MinJ_VertexSet"] # grab bottom set
tISet = mesh.specialSets["MaxJ_VertexSet"] # grab top set
bcs = uw.conditions.DirichletCondition(temp_var,
    indexSetsPerDof=bISet+tISet)
temp_var.data[bISet.data] = 1. # set bottom BC value
temp_var.data[tISet.data] = 0. # set top BC value
thermal_system = uw.systems.SteadyStateHeat(temp_var,
    fn_diffusivity=1.,
    conditions=[bcs,])

solver = uw.systems.Solver(thermal_system)
solver.solve()
```

Functions

Functions provide a high level interface to your modelling data and allow you to define model behaviours. Functions are overloaded with the +,-,*,-,[] and ** operators.

uw.function.Function:

This is an abstract class. Any class which inherits from this class is able to behave as a function object.

Useful methods:

evaluate(): Evaluate the function at the provided coordinate or coordinate array. Returns an array of results.

uw.function.coord:

This function returns the coordinate at the evaluation position.

uw.function.analytic:

This module contains various analytic solutions functions.

uw.function.branching:

This module contains various branching functions.

uw.function.exception:

This module contains functions which raise exceptions when things go wrong.

uw.function.math:

This module contains elementary mathematical functions.

uw.function.rheology:

This module contains a function which implements a stress limiting viscosity.

uw.function.shape:

This module contains a Polygon shape.

uw.function.tensor:

This module contains functions for tensor relations.

Visualisation

Underworld provides tools for visualisation in the **visualisation** submodule.

visualisation.Figure:

The Figure class is the basic container object for visualisation.

Useful methods:

show(): Show the rendered image.

save_image(): Save a rendered image to disk.

append(): Append a drawing object.

clear(): Clear the figure of drawing objects.

visualisation.objects.Drawing:

Subclasses of this class provide the ingredients you will compose your visualisations with.

This is an abstract class so you will never use it directly.

visualisation.objects.Mesh:

Draw the provided mesh object.

visualisation.objects.Surface:

Draws the provided function across a mesh surface.

visualisation.objects.Points:

Draws the provided swarm, using functions to determine point attributes.

visualisation.objects.VectorArrows:

Draws the vector arrows across a mesh corresponding to a provided vector function.

visualisation.objects.Volume:

Performs a volume render of the provided function within the provided mesh.

Example:

```
import underworld as uw
import underworld.visualisation as vis
fig = vis.Figure()
mesh = uw.mesh.FeMesh_Cartesian() # create something to draw
fig.append( vis.objects.Mesh(mesh) )
fig.show()
```