underworld Documentation

Author

Apr 14, 2018

Contents

CHAPTER 1

Modules

1.1 underworld module

Underworld2 is a python-friendly version of the Underworld geodynamics code which provides a programmable and flexible front end to all the functionality of the code running in a parallel HPC environment. This gives signficant advantages to the user, with access to the power of python libraries for setup of complex problems, analysis at runtime, problem steering, and coupling of multiple problems.

Underworld2 is integrated with the literate programming environment of the jupyter notebook system for tutorials and as a teaching tool for solid Earth geoscience.

Underworld is an open-source, particle-in-cell finite element code tuned for large-scale geodynamics simulations. The numerical algorithms allow the tracking of history information through the high-strain deformation associated with fluid flow (for example, transport of the stress tensor in a viscoelastic, convecting medium, or the advection of fine-scale damage parameters by the large-scale flow). The finite element mesh can be static or dynamic, but it is not contrained to move in lock-step with the evolving geometry of the fluid. This hybrid approach is very well suited to complex fluids which is how the solid Earth behaves on a geological timescale.

1.1.1 Module Summary

submodules:

underworld.function module

The function module contains the Function class, and related classes.

Function objects are constructed in python, but evaluated in C for efficiency. They provide a high level interface for users to compose model behaviour (such as viscosity), as well as a natural interface by which discrete data (such as meshvariables) may be utilised.

Module Summary

submodules:

underworld.function.branching module

The branching module provides functions which provide branching behaviour. Typically, these functions will select other user provided functions when certain conditions are met (with the condition also described by a function!).

Module Summary

classes:

underworld.function.branching.map	This function performs a map to other functions.
underworld.function.branching.conditional	This function provides 'if/elif' type conditional behaviour.

Module Details

classes:

class underworld.function.branching.**map**(*fn_key=None*, *mapping=None*, *fn_default=None*, **args*, ***kwargs*)

Bases: underworld.function._function.Function

This function performs a map to other functions. The user provides a python dictionary which maps unsigned integers keys to underworld functions. The user must also provide a key function. At evaluation time, the key function is evaluated first, with the outcome determining which function should finally be evaluated to return a value.

For a set of value functions $\{f_{v_0}, f_{v_1}, \ldots, f_{v_n}\}$, corresponding keys $\{k_0, k_1, \ldots, k_n\}$, and key function f_k, we have:

$$f(\mathbf{r}) = \begin{cases} f_{v_0}(\mathbf{r}), & \text{if } f_k(\mathbf{r}) = k_0 \\ f_{v_1}(\mathbf{r}), & \text{if } f_k(\mathbf{r}) = k_1 \\ \ldots & \\ f_{v_n}(\mathbf{r}), & \text{if } f_k(\mathbf{r}) = k_n \\ f_d(\mathbf{r}), & \text{otherwise} \end{cases}$$

As stated, the keys must be unsigned integers. The key function need not return an unsigned integer, but whatever value it returns **will** be cast to an unsigned integer so caution is advised.

The default function is optional, but if none is provided, and the key function evaluates to a value which is not within the user provide set of keys, an exception will be thrown.

Parameters

- **fn_key** (*underworld.function.Function (or convertible)*) – Function which returns integer key values. This function will be evaluated first to determine which function from the mapping is to be used.
- **mapping** (*dict (Function)*) – Python dictionary providing a mapping from unsigned

integer 'key' values to underworld 'value' functions. Note that the provided 'value' functions must return values of type 'double'.

- **fn_default** (underworld.function.Function *(or convertible)* *(optional)*) – Default function to be utilised when the key (returned by fn_key function) does not correspond to any key value in the mapping dictionary.

The following example sets different function behaviour inside and outside of a unit sphere. The unit sphere is represented by particles which record a swarm variable to determine if they are or not inside the sphere.

Example

Setup mesh, swarm, swarmvariable & populate

```
>>> import underworld as uw
>>> import underworld.function as fn
>>> import numpy as np
>>> mesh = uw.mesh.FeMesh_Cartesian(elementRes=(8,8),minCoord=(-1.0, -1.0),␣
↪maxCoord=(1.0, 1.0))
>>> swarm = uw.swarm.Swarm(mesh)
>>> svar = swarm.add_variable("int",1)
>>> swarm.populate_using_layout(uw.swarm.layouts.GlobalSpaceFillerLayout(swarm,
↪20))
```

For all particles in unit circle, set svar to 1

```
>>> svar.data[:] = 0
>>> for index, position in enumerate(swarm.particleCoordinates.data):
...     if position[0]**2 + position[1]**2 < 1.:
...         svar.data[index] = 1
```

Create a function which reports the value '1.' inside the sphere, and '0.' otherwise. Note that while we have only used constant value functions here, you can use any object of the class Function.

```
>>> fn_map = fn.branching.map(fn_key=svar, mapping={0: 0., 1:1.})
>>> np.allclose(np.pi, uw.utils.Integral(fn_map,mesh).evaluate(),rtol=2e-2)
True
```

Alternatively, we could utilise the default function to achieve the same result.

```
>>> fn_map = fn.branching.map(fn_key=svar, mapping={1: 1.}, fn_default=0.)
>>> np.allclose(np.pi, uw.utils.Integral(fn_map,mesh).evaluate(),rtol=2e-2)
True
```

class underworld.function.branching.**conditional**(*clauses*, **args*, ***kwargs*)

Bases: underworld.function._function.Function

This function provides 'if/elif' type conditional behaviour.

The user provides a list of tuples, with each tuple being of the form (fn_condition, fn_resultant). Effectively, each tuple provides a clause within the if/elif statement.

When evaluated, the function traverses the clauses, stopping at the first fn_condition which returns 'true'. It then executes the corresponding fn_resultant and returns the results.

If none of the provided clauses return a 'True' result, an exception is raised.

For a set of condition functions { fc_0, fc_1, … ,fc_n }, and corresponding resultant functions { fr_0, fr_1, … ,fr_n }, we have for a provided input f_in:

```
if   fc_0(f_in) :
    return fr_0(f_in)
elif fc_1(f_in) :
    return fr_1(f_in)
...
elif fc_n(f_in) :
    return fr_n(f_in)
else :
    raise RuntimeError("Reached end of conditional statement. At least one
                        of the clause conditions must evaluate to 'True'." );
```

Parameters **clauses** (*list*) – list of tuples, with each tuple being of the form (fn_condition, fn_resultant).

Example

The following example uses functions to represent a unit circle. Here a conditional function report back the value '1.' inside the sphere (as per the first condition), and '0.' otherwise.

```
>>> import underworld as uw
>>> import underworld.function as fn
>>> import numpy as np
>>> mesh = uw.mesh.FeMesh_Cartesian(elementRes=(16,16),minCoord=(-1.0, -1.0),␣
↪maxCoord=(1.0, 1.0))
>>> circleFn = fn.coord()[0]**2 + fn.coord()[1]**2
>>> fn_conditional = fn.branching.conditional( [ (circleFn < 1., 1. ),␣
↪                                              (         True, 0. ) ] )
>>> np.allclose(np.pi, uw.utils.Integral(fn_conditional,mesh).evaluate(),rtol=1e-
↪2)
True
```

underworld.function.exception module

This module provides functions which raise an exception when given conditions are encountered during function evaluations. Exception functions never modify query data.

Module Summary

classes:

underworld.function.exception.CustomException	This function allows you to set custom exceptions within your model.
underworld.function.exception.SafeMaths	This function checks if any of the following have been encountered during the evaluation of its subject function:

Module Details

classes:

class underworld.function.exception.**CustomException**(*fn_input*, *fn_condition*, *fn_print=None*, **args*, ***kwargs*)

Bases: underworld.function._function.Function

This function allows you to set custom exceptions within your model. You must pass it two functions: the first function is the pass through function, the second function is the required condition. You may also pass in a optional third function whose output will be printed if the condition evaluates to False.

A CustomException function will perform the following logic:

1. Evaluate the condition function.
2. If it evaluates to False, an exception is thrown and the simulation is halted. If a print function is provided, it will be evaluated and its results will be included in the exception message.
3. If it evaluates to True, the pass through function is evaluated with the result then being return.

Parameters

- **fn_passthrough** (underworld.function.Function) – The pass through function
- **fn_condition** (underworld.function.Function) – The condition function
- **fn_print** (underworld.function.Function) – The print function (optional).

Example

```
>>> import underworld as uw
>>> import underworld.function as fn
>>> one = fn.misc.constant(1.)
>>> passing_one = fn.exception.CustomException( one, (one < 2.) )
>>> passing_one.evaluate(0.) # constant function, so eval anywhere
array([[ 1.]])
>>> failing_one = fn.exception.CustomException( one, (one > 2.) )
>>> failing_one.evaluate(0.) # constant function, so eval anywhere
Traceback (most recent call last):
...
RuntimeError: CustomException condition function has evaluated to False for␣
↪current input!
```

Now with printing

```
>>> failing_one_by_five = fn.exception.CustomException( one, (one*5. > 20.),␣
↪one*5. )
>>> failing_one_by_five.evaluate(0.) # constant function, so eval anywhere
Traceback (most recent call last):
...
RuntimeError: CustomException condition function has evaluated to False for␣
↪current input!
Print function returns the following values (cast to double precision):
    ( 5 )
```

class underworld.function.exception.**SafeMaths**(*fn*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

This function checks if any of the following have been encountered during the evaluation of its subject function:

- Divide by zero
- Invalid domain was used for evaluation
- Value overflow errors
- Value underflow errors

If any of the above are encountered, and exception is thrown immediately.

Parameters **fn** (underworld.function.Function) – The function that is subject to the testing.

Example

```
>>> import underworld as uw
>>> import underworld.function as fn
>>> one = fn.misc.constant(1.)
>>> zero  = fn.misc.constant(0.)
>>> fn_dividebyzero = one/zero
>>> safedividebyzero = fn.exception.SafeMaths(fn_dividebyzero)
>>> safedividebyzero.evaluate(0.)  # constant function, so eval anywhere
Traceback (most recent call last):
...
RuntimeError: Divide by zero encountered while evaluating function.
```

underworld.function.tensor module

This module provides functions relating to tensor operations.

All Underworld2 functions return 1d array type objects. For tensor objects, the following convention is used:

Full tensors:

2D:

$$[a_{00}, a_{01},$$
$$a_{10}, a_{11}$$

3D:

$$[a_{00}, a_{01}, a_{02},$$
$$a_{10}, a_{11}, a_{12},$$
$$a_{20}, a_{21}, a_{22}$$

Symmetric tensors:

2D:

$$[a_{00}, a_{11}, a_{01}]$$

3D:

$$[a_{00}, a_{11}, a_{22}, a_{01}, a_{02}, a_{12}]$$

Module Summary

classes:

underworld.function.tensor.symmetric	This function calculates the symmetric part of a tensor and returns it as a symmetric tensor.
underworld.function.tensor.deviatoric	This function calculates the deviatoric stress tensor from the provided symmetric tensor.
underworld.function.tensor.antisymmetric	This function calculates the anti-symmetric part of a tensor, returning it as a tensor.
underworld.function.tensor.second_invariant	This function calculates the second invariant of (symmetric)tensor provided by the subject function.

Module Details

classes:

class underworld.function.tensor.**symmetric**(*fn*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

This function calculates the symmetric part of a tensor and returns it as a symmetric tensor. The function generated by this class returns objects of type SymmetricTensorType.

$$v_{ij} = \tfrac{1}{2}(u_{ij} + u_{ji})$$

Parameters **fn** (underworld.function.Function) – The function which provides the required tensor. This function must return objects of type TensorType.

class underworld.function.tensor.**deviatoric**(*fn*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

This function calculates the deviatoric stress tensor from the provided symmetric tensor. The function generated by this class returns objects of type SymmetricTensorType.

$$\tau_{ij} = \sigma_{ij} - \frac{\sigma_{kk}}{\delta_{ll}}\delta_{ij}$$

Parameters **fn** (underworld.function.Function) – The function which provides the required stress symmetric tensor. This function must return objects of type SymmetricTensorType.

class underworld.function.tensor.**antisymmetric**(*fn*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

This function calculates the anti-symmetric part of a tensor, returning it as a tensor. The function generated by this class returns objects of type TensorType.

$$v_{ij} = \tfrac{1}{2}(u_{ij} - u_{ji})$$

Parameters **fn** (underworld.function.Function) – The function which provides the required tensor. This function must return objects of type TensorType.

class underworld.function.tensor.**second_invariant**(*fn*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

This function calculates the second invariant of (symmetric)tensor provided by the subject function. The function generated by this class returns objects of type ScalarType.

$$u = \sqrt{\tfrac{1}{2}u_{ij}u_{ij}}$$

Parameters **fn** (underworld.function.Function) – The function which provides the required tensor. This function must return objects of type TensorType or SymmetricTensorType.

underworld.function.misc module

Miscellaneous functions.

Module Summary

classes:

underworld.function.misc.max	Returns the maximum of the results returned from its two argument function.
underworld.function.misc.constant	This function returns a constant value.
underworld.function.misc.min	Returns the minimum of the results returned from its two argument function.

Module Details

classes:

class underworld.function.misc.**max**(*fn1*, *fn2*, ***kwargs*)

Bases: underworld.function._function.Function

Returns the maximum of the results returned from its two argument function.

Parameters

- **fn1** (underworld.function.Function) – First argument function. Function must return a float type.
- **fn2** (underworld.function.Function) – Second argument function. Function must return a float type.

Example

```
>>> import underworld as uw
>>> import underworld.function as fn
>>> import numpy as np
>>> testpoints = np.array(([[ 0.0], [0.2], [0.4], [0.6], [0.8], [1.01], [1.2], [1.
↪4], [1.6], [1.8], [2.0],]))
```

Create which return identical results via different paths:

```
>>> fn_x = fn.input()[0]
>>> fn_x_minus_one = fn_x - 1.
>>> fn_one_minus_x = 1. - fn_x
```

Here we use 'max' and 'min' functions:

```
>>> fn_max = fn.misc.max(fn_one_minus_x,fn_x_minus_one)
>>> fn_min = fn.misc.min(fn_one_minus_x,fn_x_minus_one)
```

Here we use the conditional functions:

```
>>> fn_conditional_max = fn.branching.conditional( ( ( fn_x <= 1., fn_one_minus_x␣
↪), ( fn_x > 1., fn_x_minus_one ) ))
>>> fn_conditional_min = fn.branching.conditional( ( ( fn_x >= 1., fn_one_minus_x␣
↪), ( fn_x < 1., fn_x_minus_one ) ))
```

They should return identical results:

```
>>> np.allclose(fn_max.evaluate(testpoints),fn_conditional_max.
↪evaluate(testpoints))
True
>>> np.allclose(fn_min.evaluate(testpoints),fn_conditional_min.
↪evaluate(testpoints))
True
```

class underworld.function.misc.**constant**(*value*, **args*, ***kwargs*)

Bases: underworld.function._function.Function

This function returns a constant value.

Parameters **value** (*int, float, bool, iterable*) – The value the function should return. Note that iterable objects which contain valid types are permitted, but must be homogeneous in their type.

Example

```
>>> import underworld as uw
>>> import underworld.function as fn
>>> fn_const = fn.misc.constant( 3 )
>>> fn_const.evaluate(0.) # eval anywhere for constant
array([[3]], dtype=int32)
>>> fn_const = fn.misc.constant( (3,2,1) )
>>> fn_const.evaluate(0.) # eval anywhere for constant
array([[3, 2, 1]], dtype=int32)
>>> fn_const = fn.misc.constant( 3. )
>>> fn_const.evaluate(0.) # eval anywhere for constant
array([[ 3.]])
>>> fn_const = fn.misc.constant( (3.,2.,1.) )
>>> fn_const.evaluate(0.) # eval anywhere for constant
array([[ 3.,  2.,  1.]])
>>> fn_const = fn.misc.constant( True )
>>> fn_const.evaluate(0.) # eval anywhere for constant
array([[ True]], dtype=bool)
>>> fn_const = fn.misc.constant( (True,False,True) )
>>> fn_const.evaluate(0.) # eval anywhere for constant
array([[ True, False,  True]], dtype=bool)
```

value

value – constant value this function returns

class underworld.function.misc.**min**(*fn1*, *fn2*, ***kwargs*)

Bases: underworld.function._function.Function

Returns the minimum of the results returned from its two argument function.

Parameters

- **fn1** (underworld.function.Function) – First argument function. Function must return a float type.
- **fn2** (underworld.function.Function) – Second argument function. Function must return a float type.

Example

See the example provided for 'max' function.

underworld.function.analytic module

This module provides analytic solution functions to the Stokes flow equations.

Module Summary

classes:

underworld.function.analytic.SolDB3d	SolDB2d and solDB3d from:
underworld.function.analytic.SolM	
underworld.function.analytic.SolNL	
underworld.function.analytic.SolCx	The boundary conditions are free-slip everywhere on a unit domain.
underworld.function.analytic.SolKz	The boundary conditions are free-slip everywhere on a unit domain.
underworld.function.analytic.SolDB2d	SolDB2d and solDB3d from:
underworld.function.analytic.SolKx	The boundary conditions are free-slip everywhere on a unit domain.

Module Details

classes:

class underworld.function.analytic.**SolDB3d**(*Beta=4.0*, **args*, ***kwargs*)

Bases: underworld.function.analytic._SolBase

SolDB2d and solDB3d from:

Dohrmann, C.R., Bochev, P.B., A stabilized finite element method for the Stokes problem based on polynomial pressure projections, Int. J. Numer. Meth. Fluids 46, 183-201 (2004).

class underworld.function.analytic.**SolM**(*eta0=1.0*, *m=1*, *n=1.0*, *r=1.5*, **args*, ***kwargs*)

Bases: underworld.function.analytic._SolBase

class underworld.function.analytic.**SolNL**(*eta0=1.0*, *n=1.0*, *r=1.5*, **args*, ***kwargs*)

Bases: underworld.function.analytic._SolBase

class underworld.function.analytic.**SolCx**(*viscosityA=1.0*, *viscosityB=2.0*, *xc=0.25*, *nx=1*, **args*, ***kwargs*)

Bases: underworld.function.analytic._SolBase

The boundary conditions are free-slip everywhere on a unit domain. There is a viscosity jump in the x direction

at $x = xc$. The flow is driven by the following density perturbation:

$$\rho = \sin(nz\pi z)\cos(nx\pi x)$$

Parameters

- **viscosityA** (*float*) – Viscosty of region A.
- **viscosityB** (*float*) – Viscosty of region B.
- **xc** (*float*) – Location for viscosity jump.
- **nx** (*unsigned*) – Wavenumber in x axis.

class underworld.function.analytic.**SolKz**(*sigma=1.0*, *nx=1*, *nz=1.0*, *B=1.0*, **args*, ***kwargs*)

Bases: underworld.function.analytic._SolBase

The boundary conditions are free-slip everywhere on a unit domain. The viscosity varies exponentially in the z direction and is given by $\eta = \exp(2Bz)$. The flow is driven by the following density perturbation:

$$\rho = -\sigma \sin(nz\pi z)\cos(nx\pi x).$$

Parameters

- **sigma** (*float*) – Density perturbation amplitude.
- **nx** (*float*) – Wavenumber in x axis.
- **nz** (*unsigned*) – Wavenumber in axis.
- **B** (*float*) – Viscosity parameter

class underworld.function.analytic.**SolDB2d**(**args*, ***kwargs*)

Bases: underworld.function.analytic._SolBase

SolDB2d and solDB3d from:

Dohrmann, C.R., Bochev, P.B., A stabilized finite element method for the Stokes problem based on polynomial pressure projections, Int. J. Numer. Meth. Fluids 46, 183-201 (2004).

class underworld.function.analytic.**SolKx**(*sigma=1.0*, *nx=1.0*, *nz=1*, *B=1.1512925465*, **args*, ***kwargs*)

Bases: underworld.function.analytic._SolBase

The boundary conditions are free-slip everywhere on a unit domain. The viscosity varies exponentially in the x direction and is given by $\eta = \exp(2Bx)$. The flow is driven by the following density perturbation:

$$\rho = -\sigma \sin(nz\pi z)\cos(nx\pi x).$$

Parameters

- **sigma** (*float*) – Density perturbation amplitude.
- **nx** (*float*) – Wavenumber in x axis.
- **nz** (*unsigned*) – Wavenumber in axis.
- **B** (*float*) – Viscosity parameter

underworld.function.shape module

This module includes shape type functions. Shape functions generally define some geometric object, and return boolean values to indicate whether the queried locations are inside or outside the shape.

Module Summary

classes:

underworld.function.shape.Polygon	This function creates a polygon shape.

Module Details

classes:

class underworld.function.shape.**Polygon**(*vertices*, *fn=None*, **args*, ***kwargs*)

Bases: underworld.function._function.Function

This function creates a polygon shape. Note that this is strictly a 2d shape, and the third dimension of any query will be ignored. You may create a box type function if you wish to limit the shape extent in the third dimension.

You will need to use rotations to orient the polygon in other directions. Rotations functions will be available shortly (hopefully!).

Parameters

- **vertices** (*np.ndarray*) – This array provides the vertices for the polygon. Note that the order of the vertices is important. The polygon is defined by a piecewise linear edge joining the vertices in the order provided by the array. The final vertex and the initial vertex are joined to complete the polygon.
- **fn** (*underworld.function.Function, default=None*) – This is the input function. Generally it will not be required, but you may need to use (for example) to transform the incoming coordinates.

Example

In this example we will create a triangle shape and test some points.

```
>>> import underworld as uw
>>> import numpy as np
```

Create the array to define the triangle, and the function

```
>>> vertex_array = np.array( [(0.0,0.0),(0.5,1.0),(1.0,0.0)] )
>>> polyfn = uw.function.shape.Polygon(vertex_array)
```

Create some test points, and do a test evaluation

```
>>> test_array = np.array( [(0.0,0.9),(0.5,0.5),(0.9,0.2)] )
>>> polyfn.evaluate(test_array)
array([[False],
       [ True],
       [False]], dtype=bool)
```

underworld.function.rheology module

This module contains functions relating to rheological operations.

Module Summary

classes:

underworld.function.rheology.stress_limiting_viscosity	Returns a viscosity value which effectively limits the maximum fluid stress.

Module Details

classes:

class underworld.function.rheology.**stress_limiting_viscosity**(*fn_stress*, *fn_stresslimit*, *fn_inputviscosity*, **args*, ***kwargs*)

Bases: underworld.function._function.Function

Returns a viscosity value which effectively limits the maximum fluid stress. Where the stress invariant (as calculated using the provided fn_stress) is greater than the stress limit (as provided by the fn_stresslimit), the returned viscosity will affect a fluid stress at the stress limit. Otherwise, fn_inputviscosity is passed through.

Parameters

- **fn_stress** (underworld.function.Function) – Function which returns the current stress in the fluid. Function should return a symmetric tensor of floating point values.
- **fn_stresslimit** (underworld.function.Function) – Function which defines the stress limit. Function should return a scalar floating point value.
- **fn_inputviscosity** (underworld.function.Function) – Function which defines the non-yielded viscosity value. Function should return a scalar floating point value.

Example

Lets setup a simple shear type configuration but with a viscosity that increase vertically:

```
>>> import underworld as uw
>>> import underworld.function as fn
>>> mesh = uw.mesh.FeMesh_Cartesian(elementRes=(16,16), periodic=(True,False))
>>> velVar = uw.mesh.MeshVariable(mesh,2)
>>> pressVar = uw.mesh.MeshVariable(mesh.subMesh,1)
```

Simple shear boundary conditions:

```
>>> bot_nodes = mesh.specialSets["MinJ_VertexSet"]
>>> top_nodes = mesh.specialSets["MaxJ_VertexSet"]
>>> bc = uw.conditions.DirichletCondition(velVar, (top_nodes+bot_nodes,top_
↪nodes+bot_nodes))
>>> velVar.data[bot_nodes.data] = (-0.5,0.)
>>> velVar.data[top_nodes.data] = ( 0.5,0.)
```

Vertically increasing exponential viscosity:

```
>>> fn_visc = 1.
>>> stokesSys = uw.systems.Stokes(velVar,pressVar,fn_visc,conditions=[bc,])
```

Solve:

```
>>> solver = uw.systems.Solver(stokesSys)
>>> solver.solve()
```

Use the min_max function to determine a maximum stress:

```
>>> fn_stress =  2.*fn_visc*uw.function.tensor.symmetric( velVar.fn_gradient )
>>> fn_minmax_inv = fn.view.min_max(fn.tensor.second_invariant(fn_stress))
>>> ignore = fn_minmax_inv.evaluate(mesh)
>>> import numpy as np
>>> np.allclose(fn_minmax_inv.max_global(), 1.0, rtol=1e-05)
True
```

Now lets set the limited viscosity. Note that the system is now non-linear.

```
>>> fn_visc_limited = fn.rheology.stress_limiting_viscosity(fn_stress,0.5,fn_visc)
>>> stokesSys.fn_viscosity = fn_visc_limited
>>> solver.solve(nonLinearIterate=True)
```

Now check the stress:

```
>>> fn_stress = 2.*fn_visc_limited*uw.function.tensor.symmetric( velVar.fn_
↪gradient )
>>> fn_minmax_inv = fn.view.min_max(fn.tensor.second_invariant(fn_stress))
>>> ignore = fn_minmax_inv.evaluate(mesh)
>>> np.allclose(fn_minmax_inv.max_global(), 0.5, rtol=1e-05)
True
```

underworld.function.math module

This module provides math functions. All functions take functions (or convertibles) as arguments. These functions effectively wrap to the c++ standard template library equivalent. For example, the 'exp' class generates a function with uses std::exp(double).

All functions operate on and return 'double' type data (or 'float' from python).

Module Summary

classes:

underworld.function.math.pow	Power function.
underworld.function.math.cosh	Computes the hyperbolic cosine of its argument function.
underworld.function.math.acosh	Computes the inverse hyperbolic cosine of its argument function.
underworld.function.math.tan	Computes the tangent of its argument function (measured in radians).
underworld.function.math.asin	Computes the principal value of the arc sine of x, expressed in radians.
underworld.function.math.log	Computes the natural logarithm of its argument function.
underworld.function.math.atanh	Computes the inverse hyperbolic tangent of its argument function.

Continued on next page

Table 8 – continued from previous page

underworld.function.math.sqrt	Computes the square root of its argument function.
underworld.function.math.abs	Computes the absolute value of its argument function.
underworld.function.math.log10	Computes the base 10 logarithm of its argument function.
underworld.function.math.sin	Computes the sine of its argument function (measured in radians).
underworld.function.math.asinh	Computes the inverse hyperbolic sine of its argument function.
underworld.function.math.log2	Computes the base 2 logarithm of its argument function.
underworld.function.math.atan	Computes the principal value of the arc tangent of x, expressed in radians.
underworld.function.math.sinh	Computes the hyperbolic sine of its argument function.
underworld.function.math.cos	Computes the cosine of its argument function (measured in radians).
underworld.function.math.tanh	Computes the hyperbolic tangent of its argument function.
underworld.function.math.erf	Computes the error function of its argument function.
underworld.function.math.erfc	Computes the complementary error function of its argument function.
underworld.function.math.exp	Computes the exponent of its argument function.
underworld.function.math.acos	Computes the principal value of the arc cosine of x, expressed in radians.
underworld.function.math.dot	Dot product function.

Module Details

classes:

class underworld.function.math.**pow**(*fn1*, *fn2*, ***kwargs*)

Bases: underworld.function._function.Function

Power function. Raises fn1 to the power of fn2.

Parameters

- **fn1** (*underworld.function.Function (or convertible)*) – The base function.
- **fn2** (*underworld.function.Function (or convertible)*) – The power function.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = pow(_uw.function.input(),3.)
>>> np.allclose( func.evaluate(2.), math.pow(2.,3.) )
True
```

class underworld.function.math.**cosh**(*fn=None*, **args*, ***kwargs*)

Bases: underworld.function._function.Function

Computes the hyperbolic cosine of its argument function.

Parameters **fn** (*underworld.function.Function (or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = cosh()
>>> np.allclose( func.evaluate(0.1234), math.cosh(0.1234) )
True
```

class underworld.function.math.**acosh**(*fn=None*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

Computes the inverse hyperbolic cosine of its argument function.

Parameters **fn** (*underworld.function.Function (or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = acosh()
>>> np.allclose( func.evaluate(5.1234), math.acosh(5.1234) )
True
```

class underworld.function.math.**tan**(*fn=None*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

Computes the tangent of its argument function (measured in radians).

Parameters **fn** (*underworld.function.Function (or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = tan()
>>> np.allclose( func.evaluate(0.1234), math.tan(0.1234) )
True
```

class underworld.function.math.**asin**(*fn=None*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

Computes the principal value of the arc sine of x, expressed in radians.

Parameters **fn** (*underworld.function.Function (or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = asin()
```

(continues on next page)

(continued from previous page)

```
>>> np.allclose( func.evaluate(0.1234), math.asin(0.1234) )
True
```

class underworld.function.math.**log**(*fn=None*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

Computes the natural logarithm of its argument function.

Parameters **fn** (underworld.function.Function *(or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = log()
>>> np.allclose( func.evaluate(0.1234), math.log(0.1234) )
True
```

class underworld.function.math.**atanh**(*fn=None*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

Computes the inverse hyperbolic tangent of its argument function.

Parameters **fn** (underworld.function.Function *(or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = atanh()
>>> np.allclose( func.evaluate(0.1234), math.atanh(0.1234) )
True
```

class underworld.function.math.**sqrt**(*fn=None*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

Computes the square root of its argument function.

Parameters **fn** (underworld.function.Function *(or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = sqrt()
>>> np.allclose( func.evaluate(0.1234), math.sqrt(0.1234) )
True
```

class underworld.function.math.**abs**(*fn=None*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

Computes the absolute value of its argument function.

Parameters **fn** (*underworld.function.Function (or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = abs()
>>> np.allclose( func.evaluate(-0.1234), math.fabs(0.1234) )
True
```

class underworld.function.math.**log10**(*fn=None*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

Computes the base 10 logarithm of its argument function.

Parameters **fn** (*underworld.function.Function (or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = log10()
>>> np.allclose( func.evaluate(0.1234), math.log10(0.1234) )
True
```

class underworld.function.math.**sin**(*fn=None*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

Computes the sine of its argument function (measured in radians).

Parameters **fn** (*underworld.function.Function (or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = sin()
>>> np.allclose( func.evaluate(0.1234), math.sin(0.1234) )
True
```

class underworld.function.math.**asinh**(*fn=None*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

Computes the inverse hyperbolic sine of its argument function.

Parameters **fn** (*underworld.function.Function (or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = asinh()
>>> np.allclose( func.evaluate(5.1234), math.asinh(5.1234) )
True
```

class underworld.function.math.**log2**(*fn=None*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

Computes the base 2 logarithm of its argument function.

Parameters **fn** (*underworld.function.Function (or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = log2()
>>> np.allclose( func.evaluate(0.1234), math.log(0.1234,2) )
True
```

class underworld.function.math.**atan**(*fn=None*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

Computes the principal value of the arc tangent of x, expressed in radians.

Parameters **fn** (*underworld.function.Function (or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = atan()
>>> np.allclose( func.evaluate(0.1234), math.atan(0.1234) )
True
```

class underworld.function.math.**sinh**(*fn=None*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

Computes the hyperbolic sine of its argument function.

Parameters **fn** (*underworld.function.Function (or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = sinh()
```

(continues on next page)

(continued from previous page)

```
>>> np.allclose( func.evaluate(0.1234), math.sinh(0.1234) )
True
```

class underworld.function.math.**cos**(*fn=None*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

Computes the cosine of its argument function (measured in radians).

Parameters **fn** (underworld.function.Function *(or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = cos()
>>> np.allclose( func.evaluate(0.1234), math.cos(0.1234) )
True
```

class underworld.function.math.**tanh**(*fn=None*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

Computes the hyperbolic tangent of its argument function.

Parameters **fn** (underworld.function.Function *(or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = tanh()
>>> np.allclose( func.evaluate(0.1234), math.tanh(0.1234) )
True
```

class underworld.function.math.**erf**(*fn=None*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

Computes the error function of its argument function.

Parameters **fn** (underworld.function.Function *(or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = erf()
>>> np.allclose( func.evaluate(0.1234), math.erf(0.1234) )
True
```

class underworld.function.math.**erfc**(*fn=None*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

Computes the complementary error function of its argument function.

Parameters **fn** (*underworld.function.Function (or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = erfc()
>>> np.allclose( func.evaluate(0.1234), math.erfc(0.1234) )
True
```

class underworld.function.math.**exp**(*fn=None*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

Computes the exponent of its argument function.

Parameters **fn** (*underworld.function.Function (or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = exp()
>>> np.allclose( func.evaluate(0.1234), math.exp(0.1234) )
True
```

class underworld.function.math.**acos**(*fn=None*, **args*, ***kwargs*)
Bases: underworld.function._function.Function

Computes the principal value of the arc cosine of x, expressed in radians.

Parameters **fn** (*underworld.function.Function (or convertible) Default: None*) – The argument function. Default is None, in which case the input is processed directly.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> func = acos()
>>> np.allclose( func.evaluate(0.1234), math.acos(0.1234) )
True
```

class underworld.function.math.**dot**(*fn1*, *fn2*, ***kwargs*)
Bases: underworld.function._function.Function

Dot product function. Returns fn1.fn2. Argument functions must return values of identical size.

Parameters

- **fn1** (*underworld.function.Function (or convertible)*) – Argument function 1.

- **fn2** (underworld.function.Function *(or convertible)*) – Argument function 2.

Example

```
>>> from . import _systemmath as math
>>> import numpy as np
>>> input1 = (2.,3.,4.)
>>> input2 = (5.,6.,7.)
>>> func = dot( input1, input2 )
```

The function is constant, so evaluate anywhere:

```
>>> np.allclose( func.evaluate(0.), np.dot(input1,input2) )
True
```

underworld.function.view module

This module includes functions which provide views into the results of function queries. These functions never modify query data.

Module Summary

classes:

underworld.function.view.min_max	This function records the min & max result from a queried function.

Module Details

classes:

class underworld.function.view.**min_max**(*fn*, *fn_norm=None*, *fn_auxiliary=None*, **args*, ***kwargs*)

Bases: underworld.function._function.Function

This function records the min & max result from a queried function.

Note that this function simply records the min/max values encountered when it is evaluated. Therefore, if it has not been evaluated at all, the values returned via one of its methods ('min_local', 'min_global', etc) will simply be initialisation values.

For vector input types, this function will report on the magnitude of the vector.

Parameters

- **fn** (underworld.function.Function) – The primary function. If *fn_norm* is not provided, this is used to calculate the min_max. Results from this function are always passed back.
- **fn_norm** (underworld.function.Function) – This function returns a norm like quantity by which the min and max are determined. For example, where the primary func-

tion is a vector quantity, this function might calculate the magnitude of that vector. This function must return a scalar result, and must be provided where the primary function is non-scalar. See the example below for usage.

- **fn_auxiliary** (underworld.function.Function) – An auxiliary function which is evaluated at the location of the min/max. For example, often the coordinate where the min/max values occur are required, and so the user may pass in fn.input() as the auxiliary function to achieve this

Example

Create a simple function which returns two times its input:

```
>>> import underworld as uw
>>> import underworld.function as fn
>>> import numpy as np
>>> fn_simple = fn.input()[0]*2.
```

Let's wrap it with a min_max function:

```
>>> fn_minmax_simple = fn.view.min_max(fn_simple)
```

Now do an evaluation:

```
>>> fn_minmax_simple.evaluate(5.)
array([[ 10.]])
```

Since there's only been one evaluation, min and max values should be identical:

```
>>> fn_minmax_simple.min_global()
10.0
>>> fn_minmax_simple.max_global()
10.0
```

Do another evaluation:

```
>>> fn_minmax_simple.evaluate(-3.)
array([[-6.]])
```

Now check min and max again:

```
>>> fn_minmax_simple.min_global()
-6.0
>>> fn_minmax_simple.max_global()
10.0
```

Note that if we only evaluate the subject function, no min/max values are recorded:

```
>>> fn_simple.evaluate(3000.)
array([[ 6000.]])
>>> fn_minmax_simple.max_global()
10.0
```

Also note that for vector valued subject function, *fn_norm* must be provided:

```
>>> fn_vec = fn.input()*(1.,1.)
>>> fn_vec_mm = fn.view.min_max(fn_vec)
>>> fn_vec_mm.evaluate( 2. )
Traceback (most recent call last):
...
RuntimeError: Function provided to `min_max` class does not return scalar results.
 ↪ You must also provide a function which calculates the required norm like␣
 ↪quantity via the `fn_norm` parameter.
>>> fn_vec_mm = fn.view.min_max(fn_vec, fn_norm=fn.math.dot(fn_vec,fn_vec))
>>> fn_vec_mm.evaluate( 2. )
array([[ 2.,  2.]])
>>> fn_vec_mm.max_global()
8.0
>>> fn_vec_mm.evaluate( -1. )
array([[-1., -1.]])
>>> fn_vec_mm.min_global()
2.0
>>> fn_vec_mm.max_global()
8.0
```

To obtain the min/max values across a MeshVariable object, you will need to evaluate the function across all nodes of the MeshVariable:

```
>>> mesh = uw.mesh.FeMesh_Cartesian()
>>> meshvariable = uw.mesh.MeshVariable( mesh, 1 )
>>> meshvariable.data[:] = np.random.randint(100,size=meshvariable.data.shape)  #␣
 ↪init with random data
>>> fn_mv = fn.view.min_max(meshvariable)                          # create min_max␣
 ↪view wrapper
>>> ignore = fn_mv.evaluate(mesh)                                  # this call will␣
 ↪evaluate at all nodes
>>> np.allclose(fn_mv.min_local(),meshvariable.data.min())
True
>>> np.allclose(fn_mv.max_local(),meshvariable.data.max())
True
```

Note that when operating in parallel, the *min_global()* and *max_global()* methods are a good option for extracting discrete object global min/max values, as the numpy views will only report the local min/max values.

Also note that since min_max views only record results as they are evaluated, if the underlying subject function min/max values change, this will not be recorded by the min_max view until its evaluate encounters the new min/max values:

```
>>> meshvariable.data[3] = 1000                                    # change some␣
 ↪random value
>>> np.allclose(fn_mv.max_local(),meshvariable.data.max())         # check again, it␣
 ↪should be false
False
>>> ignore = fn_mv.evaluate(mesh)                                  # evaluate across␣
 ↪all nodes again
>>> np.allclose(fn_mv.max_local(),meshvariable.data.max())         # check again
True
```

Similarly, the view's min/max values are only updated when smaller/larger min/max values are encountered. So, if the underlying subject function's maximum (for example) is reduced, the view will not record this if its currently stored value exceeds the new maximum. A call to *reset()* is required:

```
>>> fn_mv.max_local()
1000.0
>>> meshvariable.data[3] = 500            # reduce max
>>> ignore = fn_mv.evaluate(mesh)         # evaluate across all nodes again
>>> fn_mv.max_local()                     # note that it still records old value
1000.0
>>> fn_mv.reset()                         # now re-init view's min/max
>>> ignore = fn_mv.evaluate(mesh)         # evaluate across all nodes again
>>> fn_mv.max_local()                     # it should now record new value
500.0
```

The auxiliary function allows you to obtain secondary information at the function minimum. One common use case would be to obtain a location where the min/max was obtained:

```
>>> fn_mv = fn.view.min_max(meshvariable, fn_auxiliary=fn.input())
>>> meshvariable.data[1] = 1000.0   # set second node to have the highest value
>>> ignore = fn_mv.evaluate(mesh)
>>> fn_mv.max_global()
1000.0
>>> np.allclose( mesh.data[1], fn_mv.max_global_auxiliary() )  # ensure max is␣
↪obtain at required mesh node.
True
```

max_global()
Returns the maximum value encountered across all processes.

Notes

This method must be called by collectively all processes.

Returns **double**

Return type maximum value

max_global_auxiliary()
Returns the results of the auxiliary function evaluated at the location corresponding to the primary function maximum. This method considers results across all processes (ie, globally).

Notes

This method must be called by collectively all processes.

Returns **FunctionIO**

Return type value at global maximum.

max_local()
Returns the max value encountered locally on the current process.

Returns **double**

Return type maximum value

max_local_auxiliary()
Returns the results of the auxiliary function evaluated at the location corresponding to the primary function maximum. This method only considers results on the current process.

Returns **FunctionIO**

Return type value at local maximum.

max_rank()
Returns the rank where the maximum occurs. Note that this method will return -1 until max_global has been called.

Returns int

Return type rank

min_global()
Returns the minimum value encountered across all processes.

Notes

This method must be called by collectively all processes.

Returns double

Return type minimum value

min_global_auxiliary()
Returns the results of the auxiliary function evaluated at the location corresponding to the primary function minimum. This method considers results across all processes (ie, globally).

Notes

This method must be called by collectively all processes.

Returns FunctionIO

Return type value at global minimum.

min_local()
Returns the minimum value encountered locally on the current process.

Returns double

Return type minimum value

min_local_auxiliary()
Returns the results of the auxiliary function evaluated at the location corresponding to the primary function minimum. This method only considers results on the current process.

Returns FunctionIO

Return type value at local minimum.

min_rank()
Returns the rank where the minimum occurs. Note that this method will return -1 until min_global has been called.

Returns int

Return type rank

reset()
Resets the minimum and maximum values.

classes:

underworld.function.Function	Objects which inherit from this class provide user definable functions within Underworld.
underworld.function.FunctionInput	Objects that inherit from this class are able to act as inputs to function evaluation from python.
underworld.function.coord	alias of `underworld.function._function.input`
underworld.function.input	This class generates a function which simply passes through its input.

Module Details

classes:

class underworld.function.**Function**(*argument_fns*, ***kwargs*)

Bases: `underworld._stgermain.LeftOverParamsChecker`

Objects which inherit from this class provide user definable functions within Underworld.

Functions aim to achieve a number of goals: * Provide a natural interface for mathematical behaviour description within python. * Provide a high level interface to Underworld discrete objects. * Allow discrete objects to be used in combination with continuous objects. * Handle the evaluation of discrete objects in the most efficient manner. * Perform all heavy calculations at the C-level for efficiency. * Provide an interface for users to evaluate functions directly within python, utilising numpy arrays for input/output.

__add__(*other*)

Operator overloading for '+' operation:

Fn3 = Fn1 + Fn2

Creates a new function Fn3 which performs additions of Fn1 and Fn2.

Returns **fn.add**

Return type Add function

Examples

```
>>> import misc
>>> import numpy as np
>>> three = misc.constant(3.)
>>> four  = misc.constant(4.)
>>> np.allclose( (three + four).evaluate(0.), [[ 7.]]  )  # note we can␣
↪evaluate anywhere because it's a constant
True
```

__and__(*other*)

Operator overloading for '&' operation:

Fn3 = Fn1 & Fn2

Creates a new function Fn3 which returns a bool result for the operation.

Returns **fn.logical_and**

Return type AND function

Examples

```
>>> import misc
>>> trueFn  = misc.constant(True)
>>> falseFn = misc.constant(False)
>>> (trueFn & falseFn).evaluate()
array([[False]], dtype=bool)
```

Notes

The '&' operator in python is usually used for bitwise 'and' operations, with the 'and' operator used for boolean type operators. It is not possible to overload the 'and' operator in python, so instead the bitwise equivalent has been utilised.

__div__(*other*)

Operator overloading for '/' operation:

Fn3 = Fn1 / Fn2

Creates a new function Fn3 which returns the quotient of Fn1 and Fn2.

Returns **fn.divide**

Return type Divide function

Examples

```
>>> import misc
>>> import numpy as np
>>> two  = misc.constant(2.)
>>> four = misc.constant(4.)
>>> np.allclose( (four/two).evaluate(0.), [[ 2.]]  )  # note we can evaluate␣
↪anywhere because it's a constant
True
```

__ge__(*other*)

Operator overloading for '>=' operation:

Fn3 = Fn1 >= Fn2

Creates a new function Fn3 which returns a bool result for the relation.

Returns **fn.greater_equal**

Return type Greater than or equal to function

Examples

```
>>> import misc
>>> import numpy as np
>>> two  = misc.constant(2.)
>>> (two >= two).evaluate()
array([[ True]], dtype=bool)
```

__getitem__(*index*)

Operator overloading for '[]' operation:

FnComponent = Fn[0]

Creates a new function FnComponent which returns the required component of Fn.

Returns **fn.at**

Return type component function

Examples

```
>>> import misc
>>> fn  = misc.constant((2.,3.,4.))
>>> np.allclose( fn[1].evaluate(0.), [[ 3.]]  )  # note we can evaluate␣
↪anywhere because it's a constant
True
```

__gt__(*other*)

Operator overloading for '>' operation:

Fn3 = Fn1 > Fn2

Creates a new function Fn3 which returns a bool result for the relation.

Returns **fn.greater**

Return type Greater than function

Examples

```
>>> import misc
>>> import numpy as np
>>> two  = misc.constant(2.)
>>> four = misc.constant(4.)
>>> (two > four).evaluate()
array([[False]], dtype=bool)
```

__le__(*other*)

Operator overloading for '<=' operation:

Fn3 = Fn1 <= Fn2

Creates a new function Fn3 which returns a bool result for the relation.

Returns **fn.less_equal**

Return type Less than or equal to function

Examples

```
>>> import misc
>>> import numpy as np
>>> two  = misc.constant(2.)
>>> (two <= two).evaluate()
array([[ True]], dtype=bool)
```

__lt__(*other*)

Operator overloading for '<' operation:

Fn3 = Fn1 < Fn2

Creates a new function Fn3 which returns a bool result for the relation.

Returns **fn.less**

Return type Less than function

Examples

```
>>> import misc
>>> import numpy as np
>>> two  = misc.constant(2.)
>>> four = misc.constant(4.)
>>> (two < four).evaluate()
array([[ True]], dtype=bool)
```

__mul__(*other*)

Operator overloading for '*' operation:

Fn3 = Fn1 * Fn2

Creates a new function Fn3 which returns the product of Fn1 and Fn2.

Returns **fn.multiply**

Return type Multiply function

Examples

```
>>> import misc
>>> import numpy as np
>>> three = misc.constant(3.)
>>> four  = misc.constant(4.)
>>> np.allclose( (three*four).evaluate(0.), [[ 12.]]  )  # note we can␣
↪evaluate anywhere because it's a constant
True
```

__neg__()

Operator overloading for unary '-'.

FnNeg = -Fn

Creates a new function FnNeg which is the negative of Fn.

Returns **fn.multiply**

Return type Negative function

Examples

```
>>> import misc
>>> import numpy as np
>>> four = misc.constant(4.)
>>> np.allclose( (-four).evaluate(0.), [[ -4.]]  )  # note we can evaluate␣
↪anywhere because it's a constant
True
```

__or__(*other*)

Operator overloading for 'I' operation:

Fn3 = Fn1 I Fn2

Creates a new function Fn3 which returns a bool result for the operation.

Returns **fn.logical_or**

Return type OR function

Examples

```
>>> import misc
>>> trueFn  = misc.constant(True)
>>> falseFn = misc.constant(False)
>>> (trueFn | falseFn).evaluate()
array([[ True]], dtype=bool)
```

Notes

The 'I' operator in python is usually used for bitwise 'or' operations, with the 'or' operator used for boolean type operators. It is not possible to overload the 'or' operator in python, so instead the bitwise equivalent has been utilised.

__pow__(*other*)

Operator overloading for '**' operation:

Fn3 = Fn1 ** Fn2

Creates a new function Fn3 which returns Fn1 to the power of Fn2.

Returns **fn.math.pow**

Return type Power function

Examples

```
>>> import misc
>>> import numpy as np
>>> two  = misc.constant(2.)
>>> four = misc.constant(4.)
>>> np.allclose( (two**four).evaluate(0.), [[ 16.]]  )  # note we can␣
↪evaluate anywhere because it's a constant
True
```

__radd__(*other*)

Operator overloading for '+' operation:

Fn3 = Fn1 + Fn2

Creates a new function Fn3 which performs additions of Fn1 and Fn2.

Returns **fn.add**

Return type Add function

Examples

```
>>> import misc
>>> import numpy as np
>>> three = misc.constant(3.)
>>> four  = misc.constant(4.)
>>> np.allclose( (three + four).evaluate(0.), [[ 7.]]  )  # note we can␣
↪evaluate anywhere because it's a constant
True
```

__rmul__(*other*)

Operator overloading for '*' operation:

Fn3 = Fn1 * Fn2

Creates a new function Fn3 which returns the product of Fn1 and Fn2.

Returns **fn.multiply**

Return type Multiply function

Examples

```
>>> import misc
>>> import numpy as np
>>> three = misc.constant(3.)
>>> four  = misc.constant(4.)
>>> np.allclose( (three*four).evaluate(0.), [[ 12.]]  )  # note we can␣
↪evaluate anywhere because it's a constant
True
```

__rsub__(*other*)

Operator overloading for '-' operation. Right hand version.

Fn3 = Fn1 - Fn2

Creates a new function Fn3 which performs subtraction of Fn2 from Fn1.

Returns **fn.subtract**

Return type RHS subtract function

Examples

```
>>> import misc
>>> import numpy as np
>>> four  = misc.constant(4.)
```

(continues on next page)

(continued from previous page)

```
>>> np.allclose( (5. - four).evaluate(0.), [[ 1.]]  )  # note we can evaluate␣
↪anywhere because it's a constant
True
```

__sub__ (*other*)

Operator overloading for '-' operation:

Fn3 = Fn1 - Fn2

Creates a new function Fn3 which performs subtraction of Fn2 from Fn1.

Returns **fn.subtract**

Return type Subtract function

Examples

```
>>> import misc
>>> import numpy as np
>>> three = misc.constant(3.)
>>> four  = misc.constant(4.)
>>> np.allclose( (three - four).evaluate(0.), [[ -1.]]  )  # note we can␣
↪evaluate anywhere because it's a constant
True
```

__xor__ (*other*)

Operator overloading for '^' operation:

Fn3 = Fn1 ^ Fn2

Creates a new function Fn3 which returns a bool result for the operation.

Returns **fn.logical_xor**

Return type XOR function

Examples

```
>>> import misc
>>> trueFn  = misc.constant(True)
>>> falseFn = misc.constant(False)
>>> (trueFn ^ falseFn).evaluate()
array([[ True]], dtype=bool)
>>> (trueFn ^ trueFn).evaluate()
array([[False]], dtype=bool)
>>> (falseFn ^ falseFn).evaluate()
array([[False]], dtype=bool)
```

Notes

The '^' operator in python is usually used for bitwise 'xor' operations, however here we always use the logical version, with the operation inputs cast to their bool equivalents before the operation.

static convert()

This method will attempt to convert the provided input into an equivalent underworld function. If the provided input is already of Function type, it is immediately returned. Likewise, if the input is of None type, it is also returned.

Parameters **obj** (*fn_like*) – The object to be converted. Note that if obj is of type None or Function, it is simply returned immediately. Where obj is of type int/float/double, a Constant type function is returned which evaluates to the provided object's value. Where obj is of type list/tuple, a function will be returned which evaluates to a vector of the provided list/tuple's values (where possible).

Returns

Return type Fn.Function or None.

Examples

```
>>> import underworld as uw
>>> import underworld.function as fn
```

```
>>> fn_const = fn.Function.convert( 3 )
>>> fn_const.evaluate(0.) # eval anywhere for constant
array([[3]], dtype=int32)
```

```
>>> fn_const == fn.Function.convert( fn_const )
True
```

```
>>> fn.Function.convert( None )
```

```
>>> fn1 = fn.input()
>>> fn2 = 10.*fn.input()
>>> fn3 = 100.*fn.input()
>>> vec = (fn1,fn2,fn3)
>>> fn_vec = fn.Function.convert(vec)
>>> fn_vec.evaluate([3.])
array([[   3.,   30.,  300.]])
```

evaluate(*inputData=None*, *inputType=None*)

This method performs evaluate of a function at the given input(s).

It accepts floats, lists, tuples, numpy arrays, or any object which is of class *FunctionInput*. lists/tuples must contain floats only.

FunctionInput class objects are shortcuts to their underlying data, often with performance advantages, and sometimes they are the only valid input type (such as using *Swarm* objects as an inputs to *SwarmVariable* evaluation). Objects of class *FeMesh*, *Swarm*, *FeMesh_IndexSet* and *VoronoiIntegrationSwarm* are also of class *FunctionInput*. See the Function section of the user guide for more information.

Results are returned as numpy array.

Parameters

- **inputData** (*float, list, tuple, ndarray,* underworld.function.FunctionInput) – The input to the function. The form of this input must be appropriate for the function being evaluated, or an exception will be thrown. Note that if no input is provided, function will be evaluated at *0.*

- **inputType** (*str*) – Specifies the type the provided data represents. Acceptable values are 'scalar', 'vector', 'symmetrictensor', 'tensor', 'array'.

Returns ndarray

Return type array of results

Examples

```
>>> from . import _systemmath as math
>>> import underworld.function.math as fnmath
>>> sinfn = fnmath.sin()
```

Single evaluation:

```
>>> np.allclose( sinfn.evaluate(math.pi/4.), [[ 0.5*math.sqrt(2.)]]  )
True
```

Multiple evaluations

```
>>> input = (0.,math.pi/4.,2.*math.pi)
>>> np.allclose( sinfn.evaluate(input), [[ 0., 0.5*math.sqrt(2.), 0.]]  )
True
```

Single MeshVariable evaluations

```
>>> mesh = uw.mesh.FeMesh_Cartesian()
>>> var = uw.mesh.MeshVariable(mesh,1)
>>> import numpy as np
>>> var.data[:,0] = np.linspace(0,1,len(var.data))
>>> result = var.evaluate( (0.2,0.5 ) )
>>> np.allclose( result, np.array([[ 0.45]]) )
True
```

Numpy input MeshVariable evaluation

```
>>> # evaluate at a set of locations.. provide these as a numpy array.
>>> count = 10
>>> # create an empty array
>>> locations = np.zeros( (count,2))
>>> # specify evaluation coodinates
>>> locations[:,0] = 0.5
>>> locations[:,1] = np.linspace(0.,1.,count)
>>> # evaluate
>>> result = var.evaluate(locations)
>>> np.allclose( result, np.array([[ 0.08333333],
↪              [ 0.17592593],                                                [ 0.
↪26851852],                                          [ 0.36111111],
↪                                    [ 0.4537037 ],
↪               [ 0.5462963 ],                                                [ 0.
↪63888889],                                          [ 0.73148148],
↪                                    [ 0.82407407],
↪               [ 0.91666667]])  )
True
```

Using the mesh object as a FunctionInput

```
>>> np.allclose( var.evaluate(mesh), var.evaluate(mesh.data))
True
```

evaluate_global(*inputData*, *inputType=None*)

This method attempts to evalute inputData across all processes, and then consolide the results on the root processor. This is most useful where you wish to evalute your functions using global coordinates which may span processes in a parallel simulation.

Note that this method does not currently support 'FunctionInput' class input data.

Due to the communications required for this method, a significant performance overhead may be encountered. The standard *evaluate* method should be used instead wherever possible.

Please see *evaluate* method for parameter details.

Notes

This method must be called collectively by all processes.

Returns

- *Only the root process gets the final results array. All other processes*
- *are returned None.*

integrate(*mesh*)

Perform an integral of this underworld function over the given mesh

Parameters **mesh** (*uw.mesh.FeMesh_Cartesian*) – Domain to perform integral over.

Examples

```
>>> mesh = uw.mesh.FeMesh_Cartesian(minCoord=(0.0,0.0), maxCoord=(1.0,2.0))
>>> fn_1 = uw.function.misc.constant(2.0)
>>> np.allclose( fn_1.integrate( mesh )[0], 4 )
True
```

```
>>> fn_2 = uw.function.misc.constant(2.0) * (0.5, 1.0)
>>> np.allclose( fn_2.integrate( mesh ), [2,4] )
True
```

class underworld.function.**FunctionInput**(**args*, ***kwargs*)

Bases: underworld._stgermain.LeftOverParamsChecker

Objects that inherit from this class are able to act as inputs to function evaluation from python.

underworld.function.**coord**

alias of underworld.function._function.input

class underworld.function.**input**(**args*, ***kwargs*)

Bases: underworld.function._function.Function

This class generates a function which simply passes through its input. It is the identity function. It is often useful when construct functions where the input itself needs to be accessed, such as to extract a particular component.

For example, you may wish to use this function when you wish to extract a particular coordinate component for manipulation. For this reason, we also provide an alias to this class called 'coord'.

Returns **fn.input**

Return type the input function

Examples

Here we see the input function simply passing through its input.

```
>>> infunc = input()
>>> np.allclose( infunc.evaluate( (1.,2.,3.) ), [ 1., 2., 3.] )
True
```

Often this behaviour is useful when we want to construct a function which operates on only a particular coordinate, such as a depth dependent density. We may wish to extract the z coordinate (in 2d):

```
>>> zcoord = input()[1]
>>> baseDensity = 1.
>>> density = baseDensity - 0.01*zcoord
>>> testCoord1 = (0.1,0.4)
>>> testCoord2 = (0.9,0.4)
>>> np.allclose( density.evaluate( testCoord1 ), density.evaluate( testCoord2 ) )
True
```

underworld.container module

Implementation relating to container objects.

Module Summary

classes:

underworld.container.IndexSet	The IndexSet class provides a set type container for integer values.
underworld.container.ObjectifiedIndexSet	This class simply adds an object to IndexSet data.

Module Details

classes:

class underworld.container.**IndexSet**(*size*, *fromObject=None*)

Bases: object

The IndexSet class provides a set type container for integer values. The underlying implementation is designed for memory efficiency. Index insertion and removal is a constant time operation.

Parameters

- **size** (*int*) – The size of the IndexSet. Note that the size corresponds to the maximum index value (plus 1) the set is required to hold, *NOT* the number of elements in the set. See IndexSet.size docstring for more information.
- **fromObject** (*iterable, array_like, IndexSet. Optional.*) – If pro-

vided, an IndexSet will be constructed using provided object's data. See 'add' method for more details on acceptable objects. If not provided, empty set is generated.

Examples

You can add items via the constructor:

```
>>> someSet = uw.container.IndexSet( 15, [3,14,2] )
>>> someSet
IndexSet([ 2,  3, 14])
```

Alternatively, create an empty set and add items as necessary:

```
>>> someSet = uw.container.IndexSet( 15 )
>>> someSet
IndexSet([])
>>> someSet.add(3)
>>> someSet
IndexSet([3])
>>> someSet.add( [2,11] )
>>> someSet
IndexSet([ 2,  3, 11])
```

Python operators are overloaded for convenience. Check class method details for full details.

AND (*indices*)

Logical AND operation performed with provided IndexSet.

Parameters **indices** (IndexSet) – IndexSet for which AND operation is performed. Note that provided set must be of type IndexSet.

Example

```
>>> someSet1 = uw.container.IndexSet( 15, [3,9,10] )
>>> someSet2 = uw.container.IndexSet( 15, [1,9,12] )
>>> someSet1.AND(someSet2)
>>> someSet1
IndexSet([9])
```

__add__ (*other*)

Operator overloading for `C = A + B`

Creates a new set C, then adds indices from A and B.

Returns **indexSet** – The new set (C).

Return type *IndexSet*

Example

```
>>> someSet1 = uw.container.IndexSet( 15, [3,9,10] )
>>> someSet2 = uw.container.IndexSet( 15, [1,9,12] )
>>> someSet1 + someSet2
IndexSet([ 1,  3,  9, 10, 12])
```

__and__(*other*)
Operator overloading for `C = A & B`

Creates a new set C, then adds indices from A, and performs AND logic with B.

Returns **indexSet** – The new set (C).

Return type *IndexSet*

Example

```
>>> someSet1 = uw.container.IndexSet( 15, [3,9,10] )
>>> someSet2 = uw.container.IndexSet( 15, [1,9,12] )
>>> someSet1 & someSet2
IndexSet([9])
```

__contains__(*index*)
Check if item is in IndexSet.

Parameters **index** (*unsigned int*) – Check if index is in IndexSet.

Returns **inSet** – True if item is in set, False otherwise.

Return type bool

Example

```
>>> someSet = uw.container.IndexSet( 15, [3,9,10] )
>>> 3 in someSet
True
```

__deepcopy__(*memo*)
Custom deepcopy routine required because python won't know how to copy memory owned by stgermain.

__iadd__(*other*)
Operator overloading for `A += B`

Adds indices from A and B.

Example

```
>>> someSet1 = uw.container.IndexSet( 15, [3,9,10] )
>>> someSet2 = uw.container.IndexSet( 15, [1,9,12] )
>>> someSet1 += someSet2
>>> someSet1
IndexSet([ 1,  3,  9, 10, 12])
```

__iand__(*other*)
Operator overloading for `A &= B`

Performs logical AND operation with A and B. Results are stored in A.

Example

```
>>> someSet1 = uw.container.IndexSet( 15, [3,9,10] )
>>> someSet2 = uw.container.IndexSet( 15, [1,9,12] )
>>> someSet1 &= someSet2
>>> someSet1
IndexSet([9])
```

__ior__(*other*)

Operator overloading for `A |= B`

Performs logical OR operation with A and B. Results are stored in A.

Example

```
>>> someSet1 = uw.container.IndexSet( 15, [3,9,10] )
>>> someSet2 = uw.container.IndexSet( 15, [1,9,12] )
>>> someSet1 |= someSet2
>>> someSet1
IndexSet([ 1,  3,  9, 10, 12])
```

__isub__(*other*)

Operator overloading for `A -= B`

Removes from A indices in B.

Example

```
>>> someSet1 = uw.container.IndexSet( 15, [3,9,10] )
>>> someSet2 = uw.container.IndexSet( 15, [1,9,12] )
>>> someSet1 -= someSet2
>>> someSet1
IndexSet([ 3, 10])
```

__or__(*other*)

Operator overloading for `C = A | B`

Creates a new set C, then adds indices from A, and performs OR logic with B.

Returns **indexSet** – The new set (C).

Return type *IndexSet*

Example

```
>>> someSet1 = uw.container.IndexSet( 15, [3,9,10] )
>>> someSet2 = uw.container.IndexSet( 15, [1,9,12] )
>>> someSet1 | someSet2
IndexSet([ 1,  3,  9, 10, 12])
```

__sub__(*other*)

Operator overloading for `C = A - B`

Creates a new set C, then adds indices from A, and removes those from B.

Returns **indexSet** – The new set (C).

Return type *IndexSet*

Example

```
>>> someSet1 = uw.container.IndexSet( 15, [3,9,10] )
>>> someSet2 = uw.container.IndexSet( 15, [1,9,12] )
>>> someSet1 - someSet2
IndexSet([ 3, 10])
```

add(*indices*)

Add item(s) to IndexSet.

Parameters **indices** (*unsigned int, ndarray, IndexSet, iterable object.*) – Index or indices to be added to the IndexSet. Ensure value(s) are integer and non-negative. An iterable object may also be provided, with numpy arrays and IndexSets being significantly more efficient.

Example

Create an empty set and add items as necessary:

```
>>> someSet = uw.container.IndexSet( 15 )
>>> someSet.add(3)
>>> someSet
IndexSet([3])
>>> 3 in someSet
True
>>> someSet.add([5,3,7,8])
>>> someSet
IndexSet([3, 5, 7, 8])
>>> someSet.add(np.array([10,11,3]))
>>> someSet
IndexSet([ 3,  5,  7,  8, 10, 11])
```

addAll()

Set all indices of set to added.

Example

```
>>> someSet = uw.container.IndexSet( 5 )
>>> someSet
IndexSet([])
>>> someSet.addAll()
>>> someSet
IndexSet([0, 1, 2, 3, 4])
```

clear()

Clear set. ie, set all indices to not included.

Example

```
>>> someSet = uw.container.IndexSet( 5, [1,2,3] )
>>> someSet
IndexSet([1, 2, 3])
>>> someSet.clear()
>>> someSet
IndexSet([])
```

count

Returns **int** – Returns the total number of members this set contains.

Return type member count

Example

```
>>> someSet = uw.container.IndexSet( 15, [3,9,10] )
>>> someSet.count
3
```

data

Returns the set members as a numpy array.

Note that only a numpy copy of the set is returned, and modifying this array is disabled (and would have no effect).

Returns Array containing IndexSet members.

Return type numpy.ndarray (uint32)

Example

```
>>> someSet = uw.container.IndexSet( 15, [3,9,10] )
>>> someSet.data
array([ 3,  9, 10], dtype=uint32)
```

invert()

Inverts the index set in place.

Example

```
>>> someSet = uw.container.IndexSet( 15, [1,3,5,7,9,11,13] )
>>> someSet.invert()
>>> someSet
IndexSet([ 0,  2,  4,  6,  8, 10, 12, 14])
```

remove(*indices*)

Remove item(s) from IndexSet.

Parameters **indices** (*unsigned int, ndarray, iterable object*) – Index or indices to be removed from the IndexSet. Ensure value(s) are integer and non-negative. An iterable object may also be provided, with numpy arrays being significantly more efficient.

Note that the 'remove' method can not be provided with an IndexSet object, as the 'add' object can.

Example

```
>>> someSet = uw.container.IndexSet( 15, [3,9,10] )
>>> someSet
IndexSet([ 3,  9, 10])
>>> someSet.remove(3)
>>> 3 in someSet
False
>>> someSet
IndexSet([ 9, 10])
>>> someSet.remove([9,10])
>>> someSet
IndexSet([])
```

size
The size of the IndexSet. Note that the size corresponds to the maximum index value (plus 1) the set is required to hold, *NOT* the number of elements in the set. So for example, a size of 16, would result in an IndexSet which can retain values between 0 and 15 (inclusive). Note also that the the IndexSet will require (size/8 + 1) bytes of memory storage.

class underworld.container.**ObjectifiedIndexSet**(*object=None*, **args*, ***kwargs*)
Bases: underworld.container._indexset.IndexSet

This class simply adds an object to IndexSet data. Usually this object will be the object for which the IndexSet data relates to.. For example, we can attach a Mesh object to an IndexSet containing mesh vertices.

__init__(*object=None*, **args*, ***kwargs*)
Class initialiser

Parameters

- **object** (*any, default=None*) – Object to tether to data
- **parent classes for further parameters.** (*See*) –

Returns **objectifiedIndexSet**

Return type *ObjectifiedIndexSet*

object
Object for which IndexSet data relates.

underworld.utils module

Various utility classes & functions.

Module Summary

functions:

underworld.utils.is_kernel	Function to determine if the script is being run in an ipython or jupyter notebook or in a regular python interpreter.

classes:

underworld.utils.SavedFileData	A class used to define saved data.
underworld.utils.Integral	The *Integral* class constructs the volume integral
underworld.utils.MeshVariable_Projection	This class provides functionality for projecting data from any underworld function onto a provided mesh variable.

Module Details

functions:

underworld.utils.**is_kernel**()

Function to determine if the script is being run in an ipython or jupyter notebook or in a regular python interpreter.

Return true if in ipython or Jupyter notebook, False otherwise.

classes:

class underworld.utils.**SavedFileData**(*pyobj*, *filename*)

Bases: object

A class used to define saved data.

Parameters

- **pyobj** (*object*) – python object saved data relates to.
- **filename** (*str*) – filename for saved data, full path

class underworld.utils.**Integral**(*fn*, *mesh=None*, *integrationType='volume'*, *surfaceIndexSet=None*, *integrationSwarm=None*, ***kwargs*)

Bases: underworld._stgermain.StgCompoundComponent

The *Integral* class constructs the volume integral

$$F_i = \int_V f_i(\mathbf{x})\,\mathrm{d}V$$

for some function f_i (specified by a *Function* object), over some domain V (specified by an *FeMesh* object), or the surface integral

$$F_i = \oint_\Gamma f_i(\mathbf{x})\,\mathrm{d}\Gamma$$

for some surface Γ (specified via an *IndexSet* object on the mesh).

Parameters

- **fn** (*uw.function.Function*) – Function to be integrated.
- **mesh** (*uw.mesh.FeMesh*) – The mesh over which integration is performed.

- **integrationType** (*str*) – Type of integration to perform. Options are "volume" or "surface".
- **surfaceIndexSet** (*uw.mesh.FeMesh_IndexSet*) – Must be provided where integrationType is "surface". This IndexSet determines which surface is to be integrated over. Note that surface integration over interior nodes is not currently supported.
- **integrationSwarm** (*uw.swarm.IntegrationSwarm (optional)*) – User provided integration swarm.

Notes

Constructor must be called by collectively all processes.

Example

Calculate volume of mesh:

```
>>> import underworld as uw
>>> mesh = uw.mesh.FeMesh_Cartesian(minCoord=(0.,0.), maxCoord=(1.,1.))
>>> volumeIntegral = uw.utils.Integral(fn=1.,mesh=mesh)
>>> np.allclose( 1., volumeIntegral.evaluate(), rtol=1e-8)
True
```

Calculate surface area of mesh:

```
>>> surfaceIntegral = uw.utils.Integral(fn=1., mesh=mesh, integrationType='surface
↪', surfaceIndexSet=mesh.specialSets["AllWalls_VertexSet"])
>>> np.allclose( 4., surfaceIntegral.evaluate(), rtol=1e-8)
True
```

evaluate()
Perform integration.

Notes

Method must be called collectively by all processes.

Returns **result** – Integration result. For vector integrals, a vector is returned.

Return type list of floats

maskFn
The integration mask used where surface integration is performed.

class underworld.utils.**MeshVariable_Projection**(*meshVariable=None*, *fn=None*, *voronoi_swarm=None*, *type=0*, ***kwargs*)

Bases: underworld._stgermain.StgCompoundComponent

This class provides functionality for projecting data from any underworld function onto a provided mesh variable.

For the variable $\mathbf{U} = \mathbf{u_a N_a}$ and function F, we wish to determine appropriate values for $\mathbf{u_a}$ such that $\mathbf{U} \simeq \mathbf{F}$.

Two projection methods are supported; weighted averages and weighted residuals. Generally speaking, weighted averages provide robust low order results, while weighted residuals give higher accuracy but spurious results for *difficult* functions F.

The weighted average method is defined as:

$$\mathbf{u_a} = \frac{\int_\Omega \mathbf{F} \mathbf{N_a} \partial\mathbf{\Omega}}{\int_\Omega \mathbf{N_a} \partial\mathbf{\Omega}}$$

The weighted residual method constructs an SLE which is then solved to determine the unknowns:

$$\mathbf{u_a} \int_\Omega \mathbf{N_a} \mathbf{N_b} \partial\mathbf{\Omega} = \int_\Omega \mathbf{F} \mathbf{N_b} \partial\mathbf{\Omega}$$

Parameters

- **meshVariable** (underworld.mesh.MeshVariable) – The variable you wish to project the function onto.
- **fn** (underworld.function.Function) – The function you wish to project.
- **voronoi_swarm** (underworld.swarm.Swarm) – Optional. If a voronoi_swarm is provided, voronoi type integration is utilised to integrate across elements. The provided swarm is used as the basis for the voronoi integration. If no swarm is provided, Gauss integration is used.
- **type** (*int, default=0*) – Projection type. 0:*weighted average*, 1:*weighted residual*

Notes

Constructor must be called collectively by all processes.

Examples

```
>>> import underworld as uw
>>> import numpy as np
>>> mesh = uw.mesh.FeMesh_Cartesian()
>>> U = uw.mesh.MeshVariable( mesh, 1 )
```

Lets cast a constant value onto this mesh variable

```
>>> const = 1.23456
>>> projector = uw.utils.MeshVariable_Projection( U, const, type=0 )
>>> np.allclose(U.data, const)
False
>>> projector.solve()
>>> np.allclose(U.data, const)
True
```

Now cast mesh coordinates onto a vector variable

```
>>> U_coord = uw.mesh.MeshVariable( mesh, 2 )
>>> projector = uw.utils.MeshVariable_Projection( U_coord, uw.function.coord(),␣
↪type=1 )
>>> projector.solve()
>>> np.allclose(U_coord.data, mesh.data)
True
```

Project one mesh variable onto another

```
>>> U_copy = uw.mesh.MeshVariable( mesh, 2 )
>>> projector = uw.utils.MeshVariable_Projection( U_copy, U_coord, type=1 )
>>> projector.solve()
>>> np.allclose(U_copy.data, U_coord.data)
True
```

Project the coords to the submesh (usually the constant mesh)

```
>>> U_submesh = uw.mesh.MeshVariable( mesh.subMesh, 2 )
>>> projector = uw.utils.MeshVariable_Projection( U_submesh, U_coord, type=1 )
>>> projector.solve()
>>> np.allclose(U_submesh.data, mesh.subMesh.data)
True
```

Create swarm, then project particle owning elements onto mesh

```
>>> U_submesh = uw.mesh.MeshVariable( mesh.subMesh, 1 )
>>> swarm = uw.swarm.Swarm(mesh)
>>> swarm.populate_using_layout(uw.swarm.layouts.GlobalSpaceFillerLayout(swarm,4))
>>> projector = uw.utils.MeshVariable_Projection( U_submesh, swarm.owningCell,␣
↪type=1 )
>>> projector.solve()
>>> np.allclose(U_submesh.data, mesh.data_elgId)
True
```

underworld.swarm module

This module contains routines relating to swarm type objects.

Module Summary

submodules:

underworld.swarm.layouts module

This module contains classes for populating swarms with particles across the domain.

Module Summary

classes:

underworld.swarm.layouts.ParticleLayoutAbstract	Abstract class.
underworld.swarm.layouts.PerCellSpaceFillerLayout	This layout fills the domain with particles in a quasi-random pattern.
underworld.swarm.layouts.GlobalSpaceFillerLayout	This layout fills the domain with particles in a quasi-random pattern.

Continued on next page

Table 14 – continued from previous page

underworld.swarm.layouts.PerCellGaussLayout	This layout populates the domain with particles located at gauss locations within each element of the swarm's associated finite element mesh.
underworld.swarm.layouts.PerCellRandomLayout	This layout fills the domain with particles in a random (per element) pattern.

Module Details

classes:

class underworld.swarm.layouts.**ParticleLayoutAbstract**(*swarm*, ***kwargs*)

Bases: underworld._stgermain.StgCompoundComponent

Abstract class. Children classes are responsible for populating swarms with particles, generally across the entire domain.

Parameters **swarm** (underworld.swarm.Swarm) – The swarm this layout will act upon

swarm

Returns Swarm this layout will act to fill with particlces.

Return type *underworld.swarm.Swarm*

class underworld.swarm.layouts.**PerCellSpaceFillerLayout**(*swarm*, *particlesPerCell*, ***kwargs*)

Bases: underworld.swarm.layouts._PerCellMeshParticleLayout

This layout fills the domain with particles in a quasi-random pattern. It utilises sobol sequences to generate per element particle locations which are more uniform than that achieved by a purely random generator.

Parameters

- **swarm** (underworld.swarm.Swarm) – The swarm this layout will act upon
- **particlesPerCell** (*int*) – The number of particles per element that this layout will generate.

Example

```
>>> import underworld as uw
>>> mesh = uw.mesh.FeMesh_Cartesian('Q1/dQ0', (1,1), (0.,0.), (1.,1.))
>>> swarm = uw.swarm.Swarm(mesh)
>>> layout = uw.swarm.layouts.PerCellSpaceFillerLayout(swarm,particlesPerCell=4)
>>> swarm.populate_using_layout(layout)
>>> swarm.particleLocalCount
4
>>> swarm.particleCoordinates.data
array([[ 0.5  ,  0.5  ],
       [ 0.25 ,  0.75 ],
       [ 0.75 ,  0.25 ],
       [ 0.375,  0.625]])
```

class underworld.swarm.layouts.**GlobalSpaceFillerLayout**(*swarm*, *particlesPerCell*, ***kwargs*)

Bases: *underworld.swarm.layouts.ParticleLayoutAbstract*

This layout fills the domain with particles in a quasi-random pattern. It utilises sobol sequences to generate global particle locations which are more uniform than that achieved by a purely random generator. This layout is mostly useful where populating particles across a rectangular domain.

Parameters

- **swarm** (underworld.swarm.Swarm) – The swarm this layout will act upon
- **particlesPerCell** (*float*) – The average number of particles per element that this layout will generate.

Example

```
>>> import underworld as uw
>>> mesh = uw.mesh.FeMesh_Cartesian('Q1/dQ0', (1,1), (0.,0.), (1.,1.))
>>> swarm = uw.swarm.Swarm(mesh)
>>> layout = uw.swarm.layouts.GlobalSpaceFillerLayout(swarm,particlesPerCell=4)
>>> swarm.populate_using_layout(layout)
>>> swarm.particleLocalCount
4
>>> swarm.particleCoordinates.data
array([[ 0.5  ,  0.5  ],
       [ 0.25 ,  0.75 ],
       [ 0.75 ,  0.25 ],
       [ 0.375,  0.625]])
```

class underworld.swarm.layouts.**PerCellGaussLayout**(*swarm*, *gaussPointCount*, ***kwargs*)

Bases: *underworld.swarm.layouts.ParticleLayoutAbstract*

This layout populates the domain with particles located at gauss locations within each element of the swarm's associated finite element mesh.

Parameters

- **swarm** (underworld.swarm.Swarm) – The swarm this layout will act upon
- **gaussPointCount** (*int*) – Per cell, the number of gauss points in each dimensional direction. Must take an int value between 1 and 5 inclusive.

Example

```
>>> import underworld as uw
>>> # choose mesh to coincide with global element
>>> mesh = uw.mesh.FeMesh_Cartesian('Q1/dQ0', (1,1), (-1.,-1.), (1.,1.))
>>> swarm = uw.swarm.Swarm(mesh)
>>> layout = uw.swarm.layouts.PerCellGaussLayout(swarm,gaussPointCount=2)
>>> swarm.populate_using_layout(layout)
>>> swarm.particleLocalCount
4
>>> swarm.particleCoordinates.data
array([[-0.57735027, -0.57735027],
       [ 0.57735027, -0.57735027],
       [-0.57735027,  0.57735027],
       [ 0.57735027,  0.57735027]])
>>> import math
```

(continues on next page)

(continued from previous page)

```
>>> # lets check one of these gauss points
>>> ( swarm.particleCoordinates.data[3][0] - math.sqrt(1./3.) ) < 1.e-10
True
```

class underworld.swarm.layouts.**PerCellRandomLayout**(*swarm*, *particlesPerCell*, *seed=13*, ***kwargs*)

Bases: underworld.swarm.layouts._PerCellMeshParticleLayout

This layout fills the domain with particles in a random (per element) pattern.

Parameters

- **swarm** (underworld.swarm.Swarm) – The swarm this layout will act upon
- **particlesPerCell** (*int*) – The number of particles per element that this layout will generate.
- **seed** (*int*) – Seed for random generator. Default is 13.

Example

```
>>> import underworld as uw
>>> mesh = uw.mesh.FeMesh_Cartesian('Q1/dQ0', (1,1), (0.,0.), (1.,1.))
>>> swarm = uw.swarm.Swarm(mesh)
>>> layout = uw.swarm.layouts.PerCellRandomLayout(swarm,particlesPerCell=4)
>>> swarm.populate_using_layout(layout)
>>> swarm.particleLocalCount
4
>>> swarm.particleCoordinates.data
array([[ 0.24261743,  0.67115852],
       [ 0.16116546,  0.70790335],
       [ 0.73160516,  0.08792286],
       [ 0.71953113,  0.15966135]])
```

classes:

underworld.swarm.PopulationControl	This class implements swarm population control mechanism.
underworld.swarm.SwarmVariable	The SwarmVariable class allows users to add data to swarm particles.
underworld.swarm.GaussIntegrationSwarm	Integration swarm which creates particles within an element at the Gauss points.
underworld.swarm.VoronoiIntegrationSwarm	Class for an IntegrationSwarm that maps to another Swarm
underworld.swarm.Swarm	The Swarm class supports particle like data structures.
underworld.swarm.IntegrationSwarm	Abstract class definition for IntegrationSwarms.
underworld.swarm.GaussBorderIntegrationSwarm	Integration swarm which creates particles within the boundary faces of an element, at the Gauss points.
underworld.swarm.SwarmAbstract	The SwarmAbstract class supports particle like data structures.

Module Details

classes:

class underworld.swarm.**PopulationControl**(*swarm*, *deleteThreshold=0.006*, *splitThreshold=0.25*, *maxDeletions=0*, *maxSplits=3*, *aggressive=False*, *aggressiveThreshold=0.8*, *particlesPerCell=None*, ***kwargs*)

Bases: underworld._stgermain.LeftOverParamsChecker

This class implements swarm population control mechanism. Population control acts on a per element basic, with a discrete voronoi algorithm is used to determine where particles should be added or removed.

Parameters

- **swarm** (underworld.swarm.Swarm) – The swarm for which population control should occur.
- **deleteThreshold** (*float*) – Particle volume fraction threshold below which particle is deleted. i.e if (particleVolume/elementVolume)<deleteThreshold, then the particle is deleted.
- **splitThreshold** (*float*) – Particle volume fraction threshold above which particle is split. i.e if (particleVolume/elementVolume)>splitThreshold, then the particle is split.
- **maxDeletions** (*int*) – maximum number of particle deletions per cell
- **maxSplits** (*int*) – maximum number of particles splits per cell
- **aggressive** (*bool*) – When enabled, this option will invoke aggressive population control in elements where particle counts drop below some threshold.
- **aggressiveThreshold** (*float*) – lower cell particle population threshold beyond which aggressive population control occurs. i.e if (cellParticleCount/particlesPerCell)<aggressiveThreshold, then aggressive pop control will occur. Note that this option is only valid if 'aggressive' is enabled.
- **particlesPerCell** (*int*) – This is the desired number of particles each element should contain. Note that this option is only valid if 'aggressive' is enabled.

Example

This simple example generates a swarm, then applies population control to split particles.

```
>>> import underworld as uw
>>> import numpy as np
>>> mesh = uw.mesh.FeMesh_Cartesian()
>>> swarm = uw.swarm.Swarm(mesh)
>>> swarm.populate_using_layout(uw.swarm.layouts.PerCellGaussLayout(swarm,4))
>>> population_control = uw.swarm.PopulationControl(swarm,deleteThreshold=0.,
↪splitThreshold=0.,maxDeletions=0,maxSplits=9999)
>>> population_control.repopulate()
>>> swarm.particleGlobalCount
512
```

repopulate()

This method repopulates the swarm.

class underworld.swarm.**SwarmVariable**(*swarm*, *dataType*, *count*, *writeable=True*, ***kwargs*)

Bases: underworld._stgermain.StgClass, underworld.function._function.Function

The SwarmVariable class allows users to add data to swarm particles. The data can be of type "char", "short", "int", "long, "float" or "double".

Note that the swarm allocates one block of contiguous memory for all the particles. The per particle variable datums is then interlaced across this memory block.

The recommended practise is to add all swarm variables before populating the swarm to avoid costly reallocations.

Swarm variables should be added via the add_variable swarm method.

Parameters

- **swarm** (underworld.swarm.Swarm) – The swarm of particles for which we wish to add the variable
- **dataType** (*str*) – The data type for the variable. Available types are "char", "short", "int", "long", "float" or "double".
- **count** (*unsigned*) – The number of values to be stored for each particle.
- **writeable** (*bool*) – Signifies if the variable should be writeable.

count

Returns Number of data items for this variable stored on each particle.

Return type int

data

Returns Numpy proxy array to underlying variable data. Note that the returned array is a proxy for all the *local* particle data. As numpy arrays are simply proxys to the underlying memory structures, no data copying is required.

Return type numpy.ndarray

Example

```
>>> # create mesh
>>> mesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(16,16),
↪ minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> # create empty swarm
>>> swarm = uw.swarm.Swarm(mesh)
>>> # add a variable
>>> svar = swarm.add_variable("int",1)
>>> # add particles
>>> swarm.populate_using_layout(uw.swarm.layouts.PerCellGaussLayout(swarm,2))
>>> swarm.particleLocalCount
1024
>>> len(svar.data)  # should be the same as particle local count
1024
>>> swarm.owningCell.data  # check particle owning cells/elements.
array([[  0],
       [  0],
       [  0],
       ...,
       [255],
```

(continues on next page)

(continued from previous page)

```
       [255],
       [255]], dtype=int32)
```

```
>>> # particle coords
>>> swarm.particleCoordinates.data[0]
array([ 0.0132078,  0.0132078])
>>> # move the particle
>>> with swarm.deform_swarm():
...     swarm.particleCoordinates.data[0] = [0.2,0.2]
>>> swarm.particleCoordinates.data[0]
array([ 0.2,  0.2])
```

dataType

Returns Data type for variable. Supported types are 'char', 'short', 'int', 'long', 'float' and 'double'.

Return type str

data_shadow

Returns Numpy proxy array to underlying variable shadow data.

Return type numpy.ndarray

Example

Refer to example provided for 'data' property(/method).

load(*filename*)
Load the swarm variable from disk. This must be called *after* the swarm.load().

Parameters **filename** (*str*) – The filename for the saved file. Relative or absolute paths may be used, but all directories must exist.

Notes

This method must be called collectively by all processes.

Example

Refer to example provided for 'save' method.

save(*filename*)
Save the swarm variable to disk.

Parameters

- **filename** (*str*) – The filename for the saved file. Relative or absolute paths may be used, but all directories must exist.
- **swarmHandle** (*uw.utils.SavedFileData , optional*) – The saved swarm file handle. If provided, a reference to the swarm file is made. Currently this doesn't provide any extra functionality.

Returns Data object relating to saved file. This only needs to be retained if you wish to create XDMF files and can be ignored otherwise.

Return type *underworld.utils.SavedFileData*

Notes

This method must be called collectively by all processes.

Example

First create the swarm, populate, then add a variable:

```
>>> mesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(16,16),
↪ minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> swarm = uw.swarm.Swarm(mesh)
>>> swarm.populate_using_layout(uw.swarm.layouts.PerCellGaussLayout(swarm,2))
>>> svar = swarm.add_variable("int",1)
```

Write something to variable

```
>>> import numpy as np
>>> svar.data[:,0] = np.arange(swarm.particleLocalCount)
```

Save to a file:

```
>>> ignoreMe = swarm.save("saved_swarm.h5")
>>> ignoreMe = svar.save("saved_swarm_variable.h5")
```

Now let's try and reload. First create a new swarm and swarm variable, and then load both:

```
>>> clone_swarm = uw.swarm.Swarm(mesh)
>>> clone_svar = clone_swarm.add_variable("int",1)
>>> clone_swarm.load("saved_swarm.h5")
>>> clone_svar.load("saved_swarm_variable.h5")
```

Now check for equality:

```
>>> import numpy as np
>>> np.allclose(svar.data,clone_svar.data)
True
```

```
>>> # clean up:
>>> if uw.rank() == 0:
...     import os;
...     os.remove( "saved_swarm.h5" )
...     os.remove( "saved_swarm_variable.h5" )
```

swarm

Returns The swarm this variable belongs to.

Return type *underworld.swarm.Swarm*

xdmf(*filename*, *varSavedData*, *varname*, *swarmSavedData*, *swarmname*, *modeltime=0.0*)

Creates an xdmf file, filename, associating the varSavedData file on the swarmSavedData file

Notes

xdmf contain 2 files: an .xml and a .h5 file. See http://www.xdmf.org/index.php/Main_Page This method only needs to be called by the master process, all other processes return quietly.

Parameters

- **filename** (*str*) – The output path to write the xdmf file. Relative or absolute paths may be used, but all directories must exist.
- **varname** (*str*) – The xdmf name to give the swarmVariable
- **swarmname** (*str*) – The xdmf name to give the swarm
- **swarmSavedData** (*underworld.utils.SaveFileData*) – Handler returned for saving a swarm. underworld.swarm.Swarm.save(xxx)
- **varSavedData** (underworld.utils.SavedFileData) – Handler returned from saving a SwarmVariable. underworld.swarm.SwarmVariable.save(xxx)
- **modeltime** (*float (default 0.0)*) – The time recorded in the xdmf output file

Example

First create the swarm and add a variable:

```
>>> mesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(16,16),
↪ minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> swarm = uw.swarm.Swarm( mesh=mesh )
>>> swarmLayout = uw.swarm.layouts.PerCellGaussLayout(swarm,2)
>>> swarm.populate_using_layout( layout=swarmLayout )
>>> swarmVar = swarm.add_variable( dataType="int", count=1 )
```

Write something to variable

```
>>> import numpy as np
>>> swarmVar.data[:,0] = np.arange(swarmVar.data.shape[0])
```

Save mesh and var to a file:

```
>>> swarmDat = swarm.save("saved_swarm.h5")
>>> swarmVarDat = swarmVar.save("saved_swarmvariable.h5")
```

Now let's create the xdmf file

```
>>> swarmVar.xdmf("TESTxdmf", swarmVarDat, "var1", swarmDat, "MrSwarm" )
```

Does file exist?

```
>>> import os
>>> if uw.rank() == 0: os.path.isfile("TESTxdmf.xdmf")
True
```

```
>>> # clean up:
>>> if uw.rank() == 0:
...     import os;
...     os.remove( "saved_swarm.h5" )
```

(continues on next page)

(continued from previous page)

```
...     os.remove( "saved_swarmvariable.h5" )
...     os.remove( "TESTxdmf.xdmf" )
```

class underworld.swarm.**GaussIntegrationSwarm**(*mesh*, *particleCount=None*, ***kwargs*)

Bases: underworld.swarm._integration_swarm.IntegrationSwarm

Integration swarm which creates particles within an element at the Gauss points.

Parameters

- **mesh** (underworld.mesh.FeMesh) – The FeMesh the swarm is supported by. See Swarm.mesh property docstring for further information.
- **particleCount** (*unsigned. Default is 3, unless Q2 mesh which takes default 5.*) – Number of gauss particles in each direction. Must take value in [1,5].

class underworld.swarm.**VoronoiIntegrationSwarm**(*swarm*, ***kwargs*)

Bases: underworld.swarm._integration_swarm.IntegrationSwarm, underworld.function._function.FunctionInput

Class for an IntegrationSwarm that maps to another Swarm

Note that this swarm can act as a function input. In this capacity, the fundamental function input type is the FEMCoordinate (ie, the particle local coordinate, the owning mesh, and the owning element). This input can be reduced to the global coordinate when returned within python. The FEMCoordinate particle representation is useful when deforming a mesh, as it is possible to deform the mesh, and then use the FEMCoordinate to reset the particles within the moved mesh.

Parameters **swarm** (underworld.swarm.Swarm) – The PIC integration swarm maps to this user provided swarm.

Example

This simple example checks that the true global coordiante, and that derived from the local coordinate, are close to equal. Note that the VoronoiIntegrationSwarm uses a voronoi centroid algorithm so we do not expect particle to exactly coincide.

```
>>> import underworld as uw
>>> import numpy as np
>>> mesh = uw.mesh.FeMesh_Cartesian()
>>> swarm = uw.swarm.Swarm(mesh)
>>> swarm.populate_using_layout(uw.swarm.layouts.PerCellGaussLayout(swarm,4))
>>> vswarm = uw.swarm.VoronoiIntegrationSwarm(swarm)
>>> vswarm.repopulate()
>>> np.allclose(swarm.particleCoordinates.data, uw.function.input().
↪evaluate(vswarm),atol=1e-1)
True
```

repopulate(*weights_calculator=None*)

This method repopulates the voronoi swarm using the provided global swarm. The weights are also recalculated.

Parameters **weights_calculator** (*underworld.swarm.Weights*) – The weights calculator for the Voronoi swarm. If none is provided, a default DVCWeights calculator is used.

class underworld.swarm.**Swarm**(*mesh*, *particleEscape=False*, ***kwargs*)

Bases: underworld.swarm._swarmabstract.SwarmAbstract, underworld.function._function.FunctionInput, underworld._stgermain.Save

The Swarm class supports particle like data structures. Each instance of this class will store a set of unique particles. In this context, particles are data structures which store a location variable, along with any other variables the user requests.

Parameters

- **mesh** (underworld.mesh.FeMesh) – The FeMesh the swarm is supported by. See Swarm.mesh property docstring for further information.
- **particleEscape** (*bool*) – If set to true, particles are deleted when they leave the domain. This may occur during particle advection, or when the mesh is deformed.

Example

Create a swarm with some variables:

```
>>> # First we need a mesh:
>>> mesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(16,16),␣
↪minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> # Create empty swarm:
>>> swarm = uw.swarm.Swarm(mesh)
>>> # Add a variable:
>>> svar = swarm.add_variable("char",1)
>>> # Add another:
>>> svar = swarm.add_variable("double",3)
>>> # Now use a layout to fill with particles
>>> swarm.particleLocalCount
0
>>> layout = uw.swarm.layouts.PerCellGaussLayout(swarm,2)
>>> swarm.populate_using_layout(layout)
>>> swarm.particleLocalCount
1024
>>> swarm.particleCoordinates.data[0]
array([ 0.0132078,  0.0132078])
>>> swarm.owningCell.data[0]
array([0], dtype=int32)
```

With particleEscape enabled, particles which are no longer within the mesh domain are deleted.

```
>>> mesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(16,16),␣
↪minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> swarm = uw.swarm.Swarm(mesh, particleEscape=True)
>>> swarm.particleLocalCount
0
>>> layout = uw.swarm.layouts.PerCellGaussLayout(swarm,2)
>>> swarm.populate_using_layout(layout)
>>> swarm.particleGlobalCount
1024
>>> with mesh.deform_mesh():
...     mesh.data[:] += (0.5,0.)
>>> swarm.particleGlobalCount
512
```

Alternatively, moving the particles:

```
>>> mesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(16,16),␣
↪minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> swarm = uw.swarm.Swarm(mesh, particleEscape=True)
>>> swarm.particleLocalCount
0
>>> layout = uw.swarm.layouts.PerCellGaussLayout(swarm,2)
>>> swarm.populate_using_layout(layout)
>>> swarm.particleGlobalCount
1024
>>> with swarm.deform_swarm():
...     swarm.particleCoordinates.data[:] -= (0.5,0.)
>>> swarm.particleGlobalCount
512
```

add_particles_with_coordinates(*coordinatesArray*)

This method adds particles to the swarm using particle coordinates provided using a numpy array.

Note that particles with coordinates NOT local to the current processor will be reject/ignored.

Parameters **coordinatesArray** (*numpy.ndarray*) – The numpy array containing the coordinate of the new particles. Array is expected to take shape n*dim, where n is the number of new particles, and dim is the dimensionality of the swarm's supporting mesh.

Returns Array containing the local index of the added particles. Rejected particles are denoted with an index of -1.

Return type numpy.ndarray

Example

```
>>> mesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(4,4),␣
↪minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> swarm = uw.swarm.Swarm(mesh)
>>> import numpy as np
>>> arr = np.zeros((5,2))
>>> arr[0] = [0.1,0.1]
>>> arr[1] = [0.2,0.1]
>>> arr[2] = [0.1,0.2]
>>> arr[3] = [-0.1,-0.1]
>>> arr[4] = [0.8,0.8]
>>> swarm.add_particles_with_coordinates(arr)
array([ 0,  1,  2, -1,  3], dtype=int32)
>>> swarm.particleLocalCount
4
>>> swarm.particleCoordinates.data
array([[ 0.1,  0.1],
       [ 0.2,  0.1],
       [ 0.1,  0.2],
       [ 0.8,  0.8]])
```

deform_swarm(***kwds*)

Any particle location modifications must occur within this python context manager. This is necessary as it is critical that certain internal objects are updated when particle locations are modified.

Parameters **update_owners** (*bool, default=True*) – If this is set to False, particle ownership (which element owns a particular particle) is not updated at the conclusion of the

context manager. This is often necessary when both the mesh and particles are advecting simutaneously.

Example

```
>>> mesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(16,16),
↪ minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> swarm = uw.swarm.Swarm(mesh)
>>> layout = uw.swarm.layouts.PerCellGaussLayout(swarm,2)
>>> swarm.populate_using_layout(layout)
>>> swarm.particleCoordinates.data[0]
array([ 0.0132078,  0.0132078])
```

Attempted modification without using deform_swarm() should fail:

```
>>> swarm.particleCoordinates.data[0] = [0.2,0.2]
Traceback (most recent call last):
...
ValueError: assignment destination is read-only
```

Within the deform_swarm() context manager, modification is allowed:

```
>>> with swarm.deform_swarm():
...     swarm.particleCoordinates.data[0] = [0.2,0.2]
>>> swarm.particleCoordinates.data[0]
array([ 0.2,  0.2])
```

fn_particle_found()

This function returns True where a particle is able to be found using the provided input to function evaluation.

Returns The function object.

Return type *underworld.function.Function*

Example

Setup some things:

```
>>> import numpy as np
>>> mesh = uw.mesh.FeMesh_Cartesian(elementRes=(32,32))
>>> swarm = uw.swarm.Swarm(mesh, particleEscape=True)
>>> layout = uw.swarm.layouts.PerCellGaussLayout(swarm,2)
>>> swarm.populate_using_layout(layout)
```

Now, evaluate the fn_particle_found function on the swarm.. all should be true

```
>>> fn_pf = swarm.fn_particle_found()
>>> fn_pf.evaluate(swarm).all()
True
```

Evalute at arbitrary coord... should return False

```
>>> fn_pf.evaluate( (0.3,0.9) )
array([[False]], dtype=bool)
```

Now, lets get rid of all particles outside of a circle, and look to obtain pi/4. First eject particles:

```
>>> with swarm.deform_swarm():
...     for ind,coord in enumerate(swarm.particleCoordinates.data):
...         if np.dot(coord,coord)>1.:
...             swarm.particleCoordinates.data[ind] = (99999.,99999.)
```

Now integrate and test

```
>>> incirc = uw.function.branching.conditional( ( (fn_pf,1.),(True,0.)  ) )
>>> np.isclose(uw.utils.Integral(incirc, mesh).evaluate(),np.pi/4.,rtol=2e-2)
array([ True], dtype=bool)
```

load(*filename*, *try_optimise=True*, *verbose=False*)

Load a swarm from disk. Note that this must be called before any SwarmVariable members are loaded.

Parameters

- **filename** (*str*) – The filename for the saved file. Relative or absolute paths may be used.
- **try_optimise** (*bool, Default=True*) – Will speed up the swarm load time but warning - this algorithm assumes the previously saved swarm data was made on an identical mesh and mesh partitioning (number of processors) with respect to the current mesh. If this isn't the case then the reloaded particle ordering will be broken, leading to an invalid swarms. One can disable this optimisation and revert to a brute force algorithm, much slower, by setting this option to False.
- **verbose** (*bool*) – Prints a swarm load progress bar.

Notes

This method must be called collectively by all processes.

Example

Refer to example provided for 'save' method.

particleGlobalCount

Returns The global number (across all processes) of particles in the swarm.

Return type int

save(*filename*)

Save the swarm to disk.

Parameters **filename** (*str*) – The filename for the saved file. Relative or absolute paths may be used, but all directories must exist.

Returns Data object relating to saved file. This only needs to be retained if you wish to create XDMF files and can be ignored otherwise.

Return type *underworld.utils.SavedFileData*

Notes

This method must be called collectively by all processes.

Example

First create the swarm, and populate with layout:

```
>>> mesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(16,16),
↪ minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> swarm = uw.swarm.Swarm(mesh)
>>> swarm.populate_using_layout(uw.swarm.layouts.PerCellGaussLayout(swarm,2))
```

Save to a file:

```
>>> ignoreMe = swarm.save("saved_swarm.h5")
```

Now let's try and reload. First create an empty swarm, and then load:

```
>>> clone_swarm = uw.swarm.Swarm(mesh)
>>> clone_swarm.load( "saved_swarm.h5" )
```

Now check for equality:

```
>>> import numpy as np
>>> np.allclose(swarm.particleCoordinates.data,clone_swarm.
↪particleCoordinates.data)
True
```

```
>>> # clean up:
>>> if uw.rank() == 0:
...     import os;
...     os.remove( "saved_swarm.h5" )
```

shadow_particles_fetch()

When called, neighbouring processor particles which have coordinates within the current processor's shadow zone will be communicated to the current processor. Ie, the processor shadow zone is populated using particles that are owned by neighbouring processors. After this method has been called, particle shadow data is available via the *data_shadow* handles of SwarmVariable objects. This data is read only.

Note that you will need to call this whenever neighbouring information has potentially changed, for example after swarm advection, or after you have modified a SwarmVariable object.

Any existing shadow information will be discarded when this is called.

Notes

This method must be called collectively by all processes.

update_particle_owners()

This routine will update particles owners after particles have been moved. This is both in terms of the cell/element the the particle resides within, and also in terms of the parallel processor decomposition (particles belonging on other processors will be sent across).

Users should not generally need to call this as it will be called automatically at the conclusion of a deform_swarm() block.

Notes

This method must be called collectively by all processes.

Example

```
>>> mesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(16,16),
↪ minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> swarm = uw.swarm.Swarm(mesh)
>>> swarm.populate_using_layout(uw.swarm.layouts.PerCellGaussLayout(swarm,2))
>>> swarm.particleCoordinates.data[0]
array([ 0.0132078,  0.0132078])
>>> swarm.owningCell.data[0]
array([0], dtype=int32)
>>> with swarm.deform_swarm():
...     swarm.particleCoordinates.data[0] = [0.1,0.1]
>>> swarm.owningCell.data[0]
array([17], dtype=int32)
```

class underworld.swarm.**IntegrationSwarm**(*mesh*, ***kwargs*)

Bases: underworld.swarm._swarmabstract.SwarmAbstract

Abstract class definition for IntegrationSwarms.

All IntegrationSwarms have the following SwarmVariables from this class:

1. **localCoordVariable** [double (number of particle, dimensions)] For local element coordinates of the particle
2. **weightVariable** [double (number of particles)] For the integration weight of each particle

particleWeights

Returns Swarm variable recording the weight of the swarm particles.

Return type *underworld.swarm.SwarmVariable*

class underworld.swarm.**GaussBorderIntegrationSwarm**(*mesh*, *particleCount=None*, ***kwargs*)

Bases: underworld.swarm._integration_swarm.GaussIntegrationSwarm

Integration swarm which creates particles within the boundary faces of an element, at the Gauss points.

See parent class for parameters.

class underworld.swarm.**SwarmAbstract**(*mesh*, ***kwargs*)

Bases: underworld._stgermain.StgCompoundComponent

The SwarmAbstract class supports particle like data structures. Each instance of this class will store a set of unique particles. In this context, particles are data structures which store a location variable, along with any other variables the user requests.

Parameters **mesh** (underworld.mesh.FeMesh) – The FeMesh the swarm is supported by. See Swarm.mesh property docstring for further information.

add_variable(*dataType*, *count*)

Add a variable to each particle in this swarm. Variables can be added at any point. Removal of variables is however not currently supported. See help(SwarmVariable) for further information.

Parameters

- **dataType** (*str*) – The data type for the variable. Available types are “char”, “short”, “int”, “float” or “double”.
- **count** (*unsigned*) – The number of values to be stored for each particle.

Returns The newly created swarm variable.

Return type *underworld.swarm.SwarmVariable*

Example

```
>>> # first we need a mesh
>>> mesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(16,16),
↪ minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> # create swarm
>>> swarm = uw.swarm.Swarm(mesh)
>>> # add a variable
>>> svar = swarm.add_variable("char",1)
>>> # add another
>>> svar = swarm.add_variable("double",3)
>>> # add some particles
>>> swarm.populate_using_layout(uw.swarm.layouts.PerCellGaussLayout(swarm,2))
>>> # add another variable
>>> svar = swarm.add_variable("double",5)
```

globalId

Returns Swarm variable recording a particle global identifier. Not yet implemented.

Return type *underworld.swarm.SwarmVariable*

mesh

Returns Supporting FeMesh for this Swarm. All swarms are required to be supported by mesh (or similar) objects, which provide the data structures necessary for efficient particle locating/tracking, as well as the necessary mechanism for the swarm parallel decomposition.

Return type *underworld.mesh.FeMesh*

owningCell

Returns Swarm variable recording the owning cell of the swarm particles. This will usually correspond to the owning element local id.

Return type *underworld.swarm.SwarmVariable*

particleCoordinates

Returns Swarm variable recording the coordinates of the swarm particles.

Return type *underworld.swarm.SwarmVariable*

particleLocalCount

Returns Number of swarm particles within the current processes local space.

Return type int

populate_using_layout(*layout*)

This method uses the provided layout to populate the swarm with particles. Usually layouts add particles across the entire domain. Available layouts may be found in the swarm.layouts module. Note that Layouts can only currently be used on empty swarms. Also note that all numpy arrays associated with swarm variables must be deleted before a layout can be applied.

Parameters **layout** (underworld.swarm.layouts.ParticleLayoutAbstract) – The layout which determines where particles are created and added.

Example

```
>>> # first we need a mesh
>>> mesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(16,16),
↪ minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> # create swarm
>>> swarm = uw.swarm.Swarm(mesh)
>>> # add populate
>>> swarm.populate_using_layout(uw.swarm.layouts.PerCellGaussLayout(swarm,2))
```

stateId

Returns Swarm state identifier. This is incremented whenever the swarm is modified.

Return type int

variables

Returns List of swarm variables associated with this swarm.

Return type list

underworld.mesh module

Implementation relating to meshing.

Module Summary

classes:

underworld.mesh.FeMesh_IndexSet	This class ties the FeMesh instance to an index set, and stores other metadata relevant to the set.
underworld.mesh.FeMesh	The FeMesh class provides the geometry and topology of a finite element discretised domain.
underworld.mesh.FeMesh_Cartesian	This class generates a finite element mesh which is topologically cartesian and geometrically regular.
underworld.mesh.MeshVariable	The MeshVariable class generates a variable supported by a finite element mesh.

Module Details

classes:

class underworld.mesh.**FeMesh_IndexSet**(*object*, *topologicalIndex=None*, **args*, ***kwargs*)

Bases: underworld.container._indexset.ObjectifiedIndexSet, underworld.function._function.FunctionInput

This class ties the FeMesh instance to an index set, and stores other metadata relevant to the set.

Parameters

- **object** (underworld.mesh.FeMesh) – The FeMesh instance from which the IndexSet was extracted.

- **topologicalIndex** (*int*) – Mesh topological index for which the IndexSet relates. See docstring for further info.

Example

```
>>> amesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(4,4),␣
↪minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> iset = uw.libUnderworld.StgDomain.RegularMeshUtils_CreateGlobalMaxISet( amesh.
↪_mesh )
>>> uw.mesh.FeMesh_IndexSet( amesh, topologicalIndex=0, size=amesh.nodesGlobal,␣
↪fromObject=iset )
FeMesh_IndexSet([ 4,  9, 14, 19, 24])
```

topologicalIndex

Returns

The topological index for the indices. The mapping is: 0 - vertex 1 - edge 2 - face 3 - volume

Return type int

class underworld.mesh.**FeMesh**(*elementType*, *generator=None*, ***kwargs*)

Bases: underworld._stgermain.StgCompoundComponent, underworld.function._function.FunctionInput

The FeMesh class provides the geometry and topology of a finite element discretised domain. The FeMesh is implicitly parallel. Some aspects may be local or global, but this is generally handled automatically.

A number of element types are supported.

Parameters

- **elementType** (*str*) – Element type for FeMesh. See FeMesh.elementType docstring for further info.
- **generator** (*underworld.mesh.MeshGenerator*) – Generator object which builds the FeMesh. See FeMesh.generator docstring for further info.

add_post_deform_function(*function*)

Adds a function function to be executed after mesh deformation is applied.

Parameters **function** (*FunctionType*) – Python (not underworld) function to be executed. Closures should be used where parameters are required.

add_pre_deform_function(*function*)

Adds a function function to be executed before mesh deformation is applied.

Parameters **function** (*FunctionType*) – Python (not underworld) function to be executed. Closures should be used where parameters are required.

add_variable(*nodeDofCount*, *dataType='double'*, ***kwargs*)

Creates and returns a mesh variable using the discretisation of the given mesh.

To set / read nodal values, use the numpy interface via the 'data' property.

Parameters

- **dataType** (*string*) – The data type for the variable. Note that only 'double' type variables are currently supported.
- **nodeDofCount** (*int*) – Number of degrees of freedom per node the variable will have

Returns The newly created mesh variable.

Return type *underworld.mesh.MeshVariable*

Example

```
>>> linearMesh  = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0',␣
↪elementRes=(16,16), minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> scalarFeVar = linearMesh.add_variable( nodeDofCount=1, dataType="double" )
>>> q0field     = linearMesh.subMesh.add_variable( 1 )  # adds variable to␣
↪secondary elementType discretisation
```

data

Numpy proxy array proxy to underlying object vertex data. Note that the returned array is a proxy for all the *local* vertices, and it is provided as 1d list.

As these arrays are simply proxys to the underlying memory structures, no data copying is required.

Note that this property returns a read-only numpy array as default. If you wish to modify mesh vertex locations, you are required to use the deform_mesh context manager.

If you are modifying the mesh, remember to modify any submesh associated with it accordingly.

Returns The data proxy array.

Return type numpy.ndarray

Example

```
>>> import underworld as uw
>>> someMesh = uw.mesh.FeMesh_Cartesian( elementType='Q1', elementRes=(2,2),␣
↪minCoord=(-1.,-1.), maxCoord=(1.,1.) )
>>> someMesh.data.shape
(9, 2)
```

You can retrieve individual vertex locations

```
>>> someMesh.data[1]
array([ 0., -1.])
```

You can modify these locations directly, but take care not to tangle the mesh! Mesh modifications must occur within the deform_mesh context manager.

```
>>> with someMesh.deform_mesh():
...     someMesh.data[1] = [0.1,-1.1]
>>> someMesh.data[1]
array([ 0.1, -1.1])
```

data_elementNodes

Returns Array specifying the nodes (global node id) for a given element (local element id). NOTE: Length is local size.

Return type numpy.ndarray

data_elgId

Returns Array specifying global element ids. Length is domain size, (local+shadow).

Return type numpy.ndarray

data_nodegId

Returns Array specifying global node ids. Length is domain size, (local+shadow).

Return type numpy.ndarray

deform_mesh(***kwds*)

Any mesh deformation must occur within this python context manager. Note that certain algorithms may be switched to their irregular mesh equivalents (if not already set this way). This may have performance implications.

Any submesh will also be appropriately updated on return from the context manager, as will various mesh metrics.

Parameters **isRegular** (*bool*) – The general assumption is that the deformed mesh will no longer be regular (orthonormal), and more general but less efficient algorithms will be selected via this context manager. To over-ride this behaviour, set this parameter to True.

Example

```
>>> import underworld as uw
>>> someMesh = uw.mesh.FeMesh_Cartesian()
>>> with someMesh.deform_mesh():
...     someMesh.data[0] = [0.1,0.1]
>>> someMesh.data[0]
array([ 0.1,  0.1])
```

elementType

Returns Element type for FeMesh. Supported types are “Q2”, “Q1”, “dQ1”, “dPc1” and “dQ0”.

Return type str

elementsDomain

Returns Returns the number of domain (local+shadow) elements on the mesh

Return type int

Example

```
>>> someMesh = uw.mesh.FeMesh_Cartesian( elementType='Q1', elementRes=(2,2),␣
↪minCoord=(-1.,-1.), maxCoord=(1.,1.) )
>>> someMesh.elementsDomain
4
```

elementsGlobal

Returns Returns the number of global elements on the mesh

Return type int

Example

```
>>> someMesh = uw.mesh.FeMesh_Cartesian( elementType='Q1', elementRes=(2,2),␣
↪minCoord=(-1.,-1.), maxCoord=(1.,1.) )
>>> someMesh.elementsGlobal
4
```

elementsLocal

Returns Returns the number of local elements on the mesh

Return type int

Example

```
>>> someMesh = uw.mesh.FeMesh_Cartesian( elementType='Q1', elementRes=(2,2),␣
↪minCoord=(-1.,-1.), maxCoord=(1.,1.) )
>>> someMesh.elementsLocal
4
```

generator

Getter/Setter for the mesh MeshGenerator object.

Returns Object which builds the mesh. Note that the mesh itself may be a generator, in which case this property will return the mesh object iself.

Return type underworld.mesh.MeshGenerator

load(*filename*)

Load the mesh from disk.

Parameters **filename** (*str*) – The filename for the saved file. Relative or absolute paths may be used, but all directories must exist.

Notes

This method must be called collectively by all processes.

If the file data array is the same length as the current mesh global size, it is assumed the file contains compatible data. Note that this may not be the case where for example where you have saved using a 2*8 resolution mesh, then loaded using an 8*2 resolution mesh.

Provided files must be in hdf5 format, and use the correct schema.

Example

Refer to example provided for 'save' method.

nodesDomain

Returns Returns the number of domain (local+shadow) nodes on the mesh.

Return type int

nodesGlobal

Returns Returns the number of global nodes on the mesh

Return type int

Example

```
>>> someMesh = uw.mesh.FeMesh_Cartesian( elementType='Q1', elementRes=(2,2),␣
↪minCoord=(-1.,-1.), maxCoord=(1.,1.) )
>>> someMesh.nodesGlobal
9
```

nodesLocal

Returns Returns the number of local nodes on the mesh.

Return type int

reset()

Reset the mesh.

Templated mesh (such as the DQ0 mesh) will be reset according to the current state of their geometryMesh.

Other mesh (such as the Q1 & Q2) will be reset to their post-construction state.

Notes

This method must be called collectively by all processes.

save(*filename*)

Save the mesh to disk

Parameters **filename** (*string*) – The name of the output file.

Returns Data object relating to saved file. This only needs to be retained if you wish to create XDMF files and can be ignored otherwise.

Return type *underworld.utils.SavedFileData*

Notes

This method must be called collectively by all processes.

Example

First create the mesh:

```
>>> mesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(16,16),
↪ minCoord=(0.,0.), maxCoord=(1.,1.) )
```

Save to a file (note that the 'ignoreMe' object isn't really required):

```
>>> ignoreMe = mesh.save("saved_mesh.h5")
```

Now let's try and reload. First create new mesh (note the different spatial size):

```
>>> clone_mesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0',␣
↪elementRes=(16,16), minCoord=(0.,0.), maxCoord=(1.5,1.5) )
```

Confirm clone mesh is different from original mesh:

```
>>> import numpy as np
>>> np.allclose(mesh.data,clone_mesh.data)
False
```

Now reload using saved file:

```
>>> clone_mesh.load("saved_mesh.h5")
```

Now check for equality:

```
>>> np.allclose(mesh.data,clone_mesh.data)
True
```

```
>>> # clean up:
>>> if uw.rank() == 0:
...     import os;
...     os.remove( "saved_mesh.h5" )
```

specialSets

Returns This dictionary stores a set of special data sets relating to mesh objects.

Return type dict

Example

```
>>> import underworld as uw
>>> someMesh = uw.mesh.FeMesh_Cartesian( elementType='Q1', elementRes=(2,2),␣
↪minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> someMesh.specialSets.keys()
['MaxI_VertexSet', 'MinI_VertexSet', 'AllWalls_VertexSet', 'MinJ_VertexSet',
↪'MaxJ_VertexSet', 'Empty']
>>> someMesh.specialSets["MinJ_VertexSet"]
FeMesh_IndexSet([0, 1, 2])
```

class underworld.mesh.**FeMesh_Cartesian**(*elementType='Q1/dQ0'*, *elementRes=(4, 4)*, *minCoord=(0.0, 0.0)*, *maxCoord=(1.0, 1.0)*, *periodic=None*, *partitioned=True*, ***kwargs*)

Bases: underworld.mesh._mesh.FeMesh, underworld.mesh._mesh.CartesianMeshGenerator

This class generates a finite element mesh which is topologically cartesian and geometrically regular. It is possible to directly build a dual mesh by passing a pair of element types to the constructor.

Refer to parent classes for parameters beyond those below.

Parameters

- **elementType** (*str*) – Mesh element type. Note that this class allows the user to (optionally) provide a pair of elementTypes for which a dual mesh will be created. The submesh is accessible through the 'subMesh' property. The primary mesh itself is the object returned by this constructor.
- **elementRes** (*list,tuple*) – List or tuple of ints specifying mesh resolution. See CartesianMeshGenerator.elementRes docstring for further information.

- **minCoord** (*list, tuple*) – List or tuple of floats specifying minimum mesh location. See CartesianMeshGenerator.minCoord docstring for further information.
- **maxCoord** (*list, tuple*) – List or tuple of floats specifying maximum mesh location. See CartesianMeshGenerator.maxCoord docstring for further information.
- **periodic** (*list, tuple*) – List or tuple of bools, specifying mesh periodicity in each direction.
- **partitioned** (*bool*) – If false, the mesh is not partitioned across entire processor pool. Instead mesh is entirely owned by processor which generated it.

Examples

To create a linear mesh:

```
>>> import underworld as uw
>>> someMesh = uw.mesh.FeMesh_Cartesian( elementType='Q1', elementRes=(16,16),␣
↪minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> someMesh.dim
2
>>> someMesh.elementRes
(16, 16)
```

Alternatively, you can create a linear/constant dual mesh

```
>>> someDualMesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(16,
↪16), minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> someDualMesh.elementType
'Q1'
>>> subMesh = someDualMesh.subMesh
>>> subMesh.elementType
'DQ0'
```

To set / read vertex coords, use the numpy interface via the 'data' property.

integrate(*fn*)

Perform a domain integral of the given underworld function over this mesh

Parameters **mesh** (*uw.mesh.FeMesh_Cartesian*) – Domain to perform integral over.

Examples

```
>>> mesh = uw.mesh.FeMesh_Cartesian(minCoord=(0.0,0.0), maxCoord=(1.0,2.0))
>>> fn_1 = uw.function.misc.constant(2.0)
>>> np.allclose( mesh.integrate( fn_1 )[0], 4 )
True
```

```
>>> fn_2 = uw.function.misc.constant(2.0) * (0.5, 1.0)
>>> np.allclose( mesh.integrate( fn_2 ), [2,4] )
True
```

subMesh

Returns Returns the submesh where the object is a dual mesh, or None otherwise.

Return type *underworld.mesh.FeMesh*

class underworld.mesh.**MeshVariable**(*mesh*, *nodeDofCount*, *dataType='double'*, ***kwargs*)

Bases: underworld._stgermain.StgCompoundComponent, underworld.function._function.Function, underworld._stgermain.Save, underworld._stgermain.Load

The MeshVariable class generates a variable supported by a finite element mesh.

To set / read nodal values, use the numpy interface via the 'data' property.

Parameters

- **mesh** (underworld.mesh.FeMesh) – The supporting mesh for the variable.
- **dataType** (*string*) – The data type for the variable. Note that only 'double' type variables are currently supported.
- **nodeDofCount** (*int*) – Number of degrees of freedom per node the variable will have.

See property docstrings for further information.

Example

For example, to create a scalar meshVariable:

```
>>> linearMesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(16,
↪16), minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> scalarFeVar = uw.mesh.MeshVariable( mesh=linearMesh, nodeDofCount=1, dataType=
↪"double" )
```

or a vector meshvariable can be created:

```
>>> vectorFeVar = uw.mesh.MeshVariable( mesh=linearMesh, nodeDofCount=3, dataType=
↪"double" )
```

copy(*deepcopy=False*)

This method returns a copy of the meshvariable.

Parameters **deepcopy** (*bool*) – If True, the variable's data is also copied into new variable.

Returns The mesh variable copy.

Return type *underworld.mesh.MeshVariable*

Example

```
>>> mesh = uw.mesh.FeMesh_Cartesian()
>>> var = uw.mesh.MeshVariable(mesh,2)
>>> import math
>>> var.data[:] = (math.pi,math.exp(1.))
>>> varCopy = var.copy()
>>> varCopy.mesh == var.mesh
True
>>> varCopy.nodeDofCount == var.nodeDofCount
True
>>> import numpy as np
>>> np.allclose(var.data,varCopy.data)
False
>>> varCopy2 = var.copy(deepcopy=True)
```

(continues on next page)

(continued from previous page)

```
>>> np.allclose(var.data,varCopy2.data)
True
```

data

Numpy proxy array to underlying variable data. Note that the returned array is a proxy for all the *local* nodal data, and is provided as 1d list. It is possible to change the shape of this numpy array to reflect the cartesian topology (where appropriate), though again only the local portion of the decomposed domain will be available, and the shape will not necessarily be identical on all processors.

As these arrays are simply proxys to the underlying memory structures, no data copying is required.

Returns The proxy array.

Return type numpy.ndarray

Example

```
>>> linearMesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0',␣
↪elementRes=(16,16), minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> scalarFeVar = uw.mesh.MeshVariable( mesh=linearMesh, nodeDofCount=1,␣
↪dataType="double" )
>>> scalarFeVar.data.shape
(289, 1)
```

You can retrieve individual nodal values

```
>>> scalarFeVar.data[100]
array([ 0.])
```

Likewise you can modify nodal values

```
>>> scalarFeVar.data[100] = 15.333
>>> scalarFeVar.data[100]
array([ 15.333])
```

dataType

Returns Data type for variable. Supported types are 'double'.

Return type str

fn_gradient

Returns a Function for the gradient field of this meshvariable.

Note that for a scalar variable T, the gradient function returns an array of the form:

$$[\frac{\partial T}{\partial x}, \frac{\partial T}{\partial y}, \frac{\partial T}{\partial z}]$$

and for a vector variable v:

$$[\frac{\partial v_x}{\partial x}, \frac{\partial v_x}{\partial y}, \frac{\partial v_x}{\partial z}, \frac{\partial v_y}{\partial x}, \frac{\partial v_y}{\partial y}, \frac{\partial v_y}{\partial z}, \frac{\partial v_z}{\partial x}, \frac{\partial v_z}{\partial y}, \frac{\partial v_z}{\partial z}]$$

Returns The gradient function.

Return type *underworld.function.Function*

load (*filename*, *interpolate=False*)
Load the MeshVariable from disk.

Parameters

- **filename** (*str*) – The filename for the saved file. Relative or absolute paths may be used, but all directories must exist.
- **interpolate** (*bool*) – Set to True to interpolate a file containing different resolution data. Note that a temporary MeshVariable with the file data will be build on **each** processor. Also note that the temporary MeshVariable can only be built if its corresponding mesh file is available. Also note that the supporting mesh mush be regular.

Notes

This method must be called collectively by all processes.

If the file data array is the same length as the current variable global size, it is assumed the file contains compatible data. Note that this may not be the case where for example where you have saved using a 2*8 resolution mesh, then loaded using an 8*2 resolution mesh.

Provided files must be in hdf5 format, and use the correct schema.

Example

Refer to example provided for 'save' method.

mesh

Returns Supporting FeMesh for this MeshVariable.

Return type *underworld.mesh.FeMesh*

nodeDofCount

Returns Degrees of freedom on each mesh node that this variable provides.

Return type int

save (*filename*, *meshHandle=None*)
Save the MeshVariable to disk.

Parameters

- **filename** (*string*) – The name of the output file. Relative or absolute paths may be used, but all directories must exist.
- **meshHandle** (*uw.utils.SavedFileData , optional*) – The saved mesh file handle. If provided, a link is created within the mesh variable file to this saved mesh file. Important for checkpoint when the mesh deforms.

Notes

This method must be called collectively by all processes.

Returns Data object relating to saved file. This only needs to be retained if you wish to create XDMF files and can be ignored otherwise.

Return type *underworld.utils.SavedFileData*

Example

First create the mesh add a variable:

```
>>> mesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(16,16),
↪ minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> var = uw.mesh.MeshVariable( mesh=mesh, nodeDofCount=1, dataType="double" )
```

Write something to variable

```
>>> import numpy as np
>>> var.data[:,0] = np.arange(var.data.shape[0])
```

Save to a file (note that the 'ignoreMe' object isn't really required):

```
>>> ignoreMe = var.save("saved_mesh_variable.h5")
```

Now let's try and reload.

```
>>> clone_var = uw.mesh.MeshVariable( mesh=mesh, nodeDofCount=1, dataType=
↪"double" )
>>> clone_var.load("saved_mesh_variable.h5")
```

Now check for equality:

```
>>> np.allclose(var.data,clone_var.data)
True
```

```
>>> # clean up:
>>> if uw.rank() == 0:
...     import os;
...     os.remove( "saved_mesh_variable.h5" )
```

syncronise()

This method is often necessary when Underworld is operating in parallel.

It will syncronise the mesh variable so that it is consistent with it's parallel neighbours. Specifically, the shadow space of each process obtains the required data from neighbouring processes.

xdmf(*filename*, *fieldSavedData*, *varname*, *meshSavedData*, *meshname*, *modeltime=0.0*)

Creates an xdmf file, filename, associating the fieldSavedData file on the meshSavedData file

Notes

xdmf contain 2 files: an .xml and a .h5 file. See http://www.xdmf.org/index.php/Main_Page This method only needs to be called by the master process, all other processes return quiely.

Parameters

- **filename** (*str*) – The output path to write the xdmf file. Relative or absolute paths may be used, but all directories must exist.
- **varname** (*str*) – The xdmf name to give the field
- **meshSavedData** (*underworld.utils.SaveFileData*) – Handler returned for saving a mesh. underworld.mesh.save(xxx)
- **meshname** (*str*) – The xdmf name to give the mesh

- **fieldSavedData** (*underworld.SavedFileData*) – Handler returned from saving a field. underworld.mesh.save(xxx)
- **modeltime** (*float*) – The time recorded in the xdmf output file

Example

First create the mesh add a variable:

```
>>> mesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(16,16),
↪ minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> var = uw.mesh.MeshVariable( mesh=mesh, nodeDofCount=1, dataType="double" )
```

Write something to variable

```
>>> import numpy as np
>>> var.data[:,0] = np.arange(var.data.shape[0])
```

Save mesh and var to a file:

```
>>> meshDat = mesh.save("saved_mesh.h5")
>>> varDat = var.save("saved_mesh_variable.h5")
```

Now let's create the xdmf file

```
>>> var.xdmf("TESTxdmf", varDat, "var1", meshDat, "meshie" )
```

Does file exist?

```
>>> import os
>>> if uw.rank() == 0: os.path.isfile("TESTxdmf.xdmf")
True
```

Clean up:

```
>>> if uw.rank() == 0:
...     import os;
...     os.remove( "saved_mesh_variable.h5" )
...     os.remove( "saved_mesh.h5" )
...     os.remove( "TESTxdmf.xdmf" )
```

underworld.systems module

This module contains routines relating to differential system.

Module Summary

submodules:

underworld.systems.sle module

Module Summary

classes:

underworld.systems.sle.ConstitutiveMatrixTerm	
underworld.systems.sle.PreconditionerMatrixTerm	
underworld.systems.sle.AssembledVector	Vector object, generally assembled as a result of the FEM framework.
underworld.systems.sle.AssemblyTerm	
underworld.systems.sle.AssembledMatrix	Matrix object, generally assembled as a result of the FEM framework.
underworld.systems.sle.VectorAssemblyTerm	
underworld.systems.sle.MatrixAssemblyTerm_NA_i__NB_i__Fn	
underworld.systems.sle.LumpedMassMatrixVectorTerm	
underworld.systems.sle.VectorAssemblyTerm_NA_i__Fn_i	
underworld.systems.sle.GradientStiffnessMatrixTerm	
underworld.systems.sle.MatrixAssemblyTerm_NA__NB__Fn	
underworld.systems.sle.EqNumber	The SolutionVector manages the numerical solution vectors used by Underworld's equation systems.
underworld.systems.sle.AdvDiffResidualVectorTerm	
underworld.systems.sle.SolutionVector	The SolutionVector manages the numerical solution vectors used by Underworld's equation systems.
underworld.systems.sle.MatrixAssemblyTerm	
underworld.systems.sle.VectorSurfaceAssemblyTerm_NA__Fn__ni	Build an assembly term for a surface integral.
underworld.systems.sle.VectorAssemblyTerm_NA__Fn	
underworld.systems.sle.VectorAssemblyTerm_NA_j__Fn_ij	Build an assembly term for a spatial gradient, used for the viscoelastic force term.

Module Details

classes:

class underworld.systems.sle.**ConstitutiveMatrixTerm**(*fn_visc1=None*, *fn_visc2=None*, *fn_director=None*, ***kwargs*)
Bases: underworld.systems.sle._assemblyterm.MatrixAssemblyTerm

class underworld.systems.sle.**PreconditionerMatrixTerm**(*assembledObject=None*, ***kwargs*)
Bases: underworld.systems.sle._assemblyterm.MatrixAssemblyTerm

class underworld.systems.sle.**AssembledVector**(*meshVariable*, *eqNum*, ***kwargs*)
Bases: underworld.systems.sle._svector.SolutionVector

Vector object, generally assembled as a result of the FEM framework.

See parent class for parameters.

petscVector
petscVector (swig petsc vector) – Underlying PETSc vector object.

class underworld.systems.sle.**AssemblyTerm**(*integrationSwarm*, *extraInfo=None*, ***kwargs*)
Bases: underworld._stgermain.StgCompoundComponent

class underworld.systems.sle.**AssembledMatrix**(*rowVector*, *colVector*, *rhs=None*, *rhs_T=None*, *assembleOnNodes=False*, ***kwargs*)
Bases: underworld._stgermain.StgCompoundComponent

Matrix object, generally assembled as a result of the FEM framework.

Parameters

- **meshVariableRow** (underworld.mesh.MeshVariable) – MeshVariable object for matrix row.
- **meshVariableCol** (underworld.mesh.MeshVariable) – MeshVariable object for matrix column.

class underworld.systems.sle.**VectorAssemblyTerm**(*assembledObject*, ***kwargs*)
Bases: underworld.systems.sle._assemblyterm.AssemblyTerm

class underworld.systems.sle.**MatrixAssemblyTerm_NA_i__NB_i__Fn**(*fn*, ***kwargs*)
Bases: underworld.systems.sle._assemblyterm.MatrixAssemblyTerm

class underworld.systems.sle.**LumpedMassMatrixVectorTerm**(*assembledObject*, ***kwargs*)
Bases: underworld.systems.sle._assemblyterm.VectorAssemblyTerm

class underworld.systems.sle.**VectorAssemblyTerm_NA_i__Fn_i**(*fn*, *mesh=None*, ***kwargs*)
Bases: underworld.systems.sle._assemblyterm.VectorAssemblyTerm

class underworld.systems.sle.**GradientStiffnessMatrixTerm**(*assembledObject=None*, ***kwargs*)
Bases: underworld.systems.sle._assemblyterm.MatrixAssemblyTerm

class underworld.systems.sle.**MatrixAssemblyTerm_NA__NB__Fn**(*fn*, *mesh*, ***kwargs*)
Bases: underworld.systems.sle._assemblyterm.MatrixAssemblyTerm

class underworld.systems.sle.**EqNumber**(*meshVariable*, *removeBCs=True*, ***kwargs*)
Bases: underworld._stgermain.StgClass

The SolutionVector manages the numerical solution vectors used by Underworld's equation systems. Interface between meshVariables and systems.

Parameters **meshVariable** (*uw.mesh.MeshVariable*) – MeshVariable object for which this equation numbering corresponds.

Example

```
>>> linearMesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(4,4),
↪ minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> tField = uw.mesh.MeshVariable( linearMesh, 1 )
>>> teqNum = uw.systems.sle.EqNumber( tField )
```

class underworld.systems.sle.**AdvDiffResidualVectorTerm**(*velocityField*, *diffusivity*, *sourceTerm*, ***kwargs*)

Bases: underworld.systems.sle._assemblyterm.VectorAssemblyTerm

class underworld.systems.sle.**SolutionVector**(*meshVariable*, *eqNumber*, ***kwargs*)

Bases: underworld._stgermain.StgCompoundComponent

The SolutionVector manages the numerical solution vectors used by Underworld's equation systems. Interface between meshVariables and systems.

Parameters

- **meshVariable** (underworld.mesh.MeshVariable) – MeshVariable object for which this SLE vector corresponds.
- **eqNumber** (underworld.systems.sle.EqNumber) – Equation numbering object corresponding to this vector.

Example

```
>>> linearMesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(4,4),
↪ minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> tField = uw.mesh.MeshVariable( linearMesh, 1 )
>>> eqNum = uw.systems.sle.EqNumber( tField )
>>> sVector = uw.systems.sle.SolutionVector(tField, eqNum )
```

class underworld.systems.sle.**MatrixAssemblyTerm**(*assembledObject=None*, ***kwargs*)

Bases: underworld.systems.sle._assemblyterm.AssemblyTerm

class underworld.systems.sle.**VectorSurfaceAssemblyTerm_NA__Fn__ni**(*nbc*, *integrationSwarm=None*, *surfaceGaussPoints=2*, ***kwargs*)

Bases: underworld.systems.sle._assemblyterm.VectorAssemblyTerm

Build an assembly term for a surface integral.

Parameters

- **nbc** (underworld.conditions.NeumannCondition) – See uw.conditions.NeumannCondition for details

- **integrationSwarm** (underworld.swarm.GaussBorderIntegrationSwarm) – Optional integration swarm to be used for numerical integration.
- **surfaceGaussPoints** (*int*) – The number of quadrature points per element face to use in surface integration. Will be used to create a GaussBorderIntegrationSwarm in the case the 'swarm' input is 'None'.

class underworld.systems.sle.**VectorAssemblyTerm_NA__Fn**(*fn*, *mesh=None*, ***kwargs*)
Bases: underworld.systems.sle._assemblyterm.VectorAssemblyTerm

class underworld.systems.sle.**VectorAssemblyTerm_NA_j__Fn_ij**(*fn*, *mesh=None*, ***kwargs*)
Bases: underworld.systems.sle._assemblyterm.VectorAssemblyTerm

Build an assembly term for a spatial gradient, used for the viscoelastic force term.

Parameters

- **fn** (underworld.function.Function) – Function is a vector of size 3 (dim=2) or 6 (dim=3) representing a symetric tensor
- **mesh** (*uw.mesh.FeMesh_Cartesian*) –

functions:

underworld.systems.Solver	This method simply returns a necessary solver for the provided system.

classes:

underworld.systems.Stokes	This class provides functionality for a discrete representation of the Stokes flow equations.
underworld.systems.SteadyStateHeat	This class provides functionality for a discrete representation of the steady state heat equation.
underworld.systems.SteadyStateDarcyFlow	This class provides functionality for a discrete representation of the steady state darcy flow equation.
underworld.systems.TimeIntegration	Abstract class for integrating numerical objects (fields, swarms, etc.) in time.
underworld.systems.AdvectionDiffusion	This class provides functionality for a discrete representation of an advection-diffusion equation.
underworld.systems.SwarmAdvector	Objects of this class advect a swarm through time using the provided velocity field.

Module Details

functions:

underworld.systems.**Solver**(*eqs*, *type='BSSCR'*, **args*, ***kwargs*)
This method simply returns a necessary solver for the provided system.

classes:

class underworld.systems.**Stokes**(*velocityField*, *pressureField*, *fn_viscosity*, *fn_bodyforce=None*, *fn_one_on_lambda=None*, *fn_lambda=None*, *fn_source=None*, *voronoi_swarm=None*, *conditions=[]*, *_removeBCs=True*, *_fn_viscosity2=None*, *_fn_director=None*, *fn_stresshistory=None*, *_fn_stresshistory=None*, *_fn_v0=None*, *_fn_p0=None*, *_callback_post_solve=None*, ***kwargs*)

Bases: underworld._stgermain.StgCompoundComponent

This class provides functionality for a discrete representation of the Stokes flow equations.

Specifically, the class uses a mixed finite element method to construct a system of linear equations which may then be solved using an object of the underworld.system.Solver class.

The underlying element types are determined by the supporting mesh used for the 'velocityField' and 'pressureField' parameters.

The strong form of the given boundary value problem, for f, g and h given, is

$$\sigma_{ij,j} + f_i = 0 \qquad \text{in } \Omega \tag{1.1}$$

$$u_{k,k} + \frac{p}{\lambda} = H \qquad \text{in } \Omega \tag{1.2}$$

$$u_i = g_i \qquad \text{on } \Gamma_{g_i} \tag{1.3}$$

$$\sigma_{ij} n_j = h_i \qquad \text{on } \Gamma_{h_i} \tag{1.4}$$

(1.5)

where,

- $\sigma_{i,j}$ is the stress tensor
- u_i is the velocity,
- p is the pressure,
- f_i is a body force,
- λ is a bulk viscosity,
- H is the compressible equation source term,
- g_i are the velocity boundary conditions (DirichletCondition)
- h_i are the traction boundary conditions (NeumannCondition).

The problem boundary, Γ, admits the decompositions $\Gamma = \Gamma_{g_i} \cup \Gamma_{h_i}$ where $\emptyset = \Gamma_{g_i} \cap \Gamma_{h_i}$. The equivalent weak form is:

$$\int_\Omega w_{(i,j)} \sigma_{ij} \, d\Omega = \int_\Omega w_i \, f_i \, d\Omega + \sum_{j=1}^{n_{sd}} \int_{\Gamma_{h_j}} w_i \, h_i \, d\Gamma$$

where we must find u which satisfies the above for all w in some variational space.

Parameters

- **velocityField** (underworld.mesh.MeshVariable) – Variable used to record system velocity.
- **pressureField** (underworld.mesh.MeshVariable) – Variable used to record system pressure.

- **fn_viscosity** (underworld.function.Function) – Function which reports a viscosity value. Function must return scalar float values.
- **fn_bodyforce** (underworld.function.Function, *Default = None*) – Function which reports a body force for the system. Function must return float values of identical dimensionality to the provided velocity variable.
- **fn_lambda** (*Removed use, fn_one_on_lambda instead*) –
- **fn_minus_one_on_lambda** (underworld.function.Function, *Default = None*) – Function which defines a non solenoidal velocity field via the relationship div(velocityField) = -fn_minus_one_on_lambda * pressurefield + fn_source When this is left as None a incompressible formulation of the stokes equation is formed, ie, div(velocityField) = 0. fn_minus_one_on_lambda is incompatible with the 'penalty' stokes solver, ensure a 'penalty' equal to 0 is used when fn_minus_one_on_lambda is used. By default this is the case.
- **fn_source** (underworld.function.Function, *Default = None*) – Function which defines a non solenoidal velocity field via the relationship div(velocityField) = -fn_minus_one_on_lambda * pressurefield + fn_source. fn_minus_one_on_lambda is incompatible with the 'penalty' stokes solver, ensure the 'penalty' of 0, is used when fn_lambda is used. By default this is the case.
- **fn_stresshistory** (underworld.function.Function, *Default = None*) – Function which defines the stress history term used for viscoelasticity. Function is a vector of size 3 (dim=2) or 6 (dim=3) representing a symetric tensor.
- **voronoi_swarm** (underworld.swarm.Swarm) – If a voronoi_swarm is provided, voronoi type numerical integration is utilised. The provided swarm is used as the basis for the voronoi integration. If no voronoi_swarm is provided, Gauss integration is used.
- **conditions** (underworld.conditions.SystemCondition) – Numerical conditions to impose on the system. This should be supplied as the condition itself, or a list object containing the conditions.

Notes

Constructor must be called by collectively all processes.

eqResiduals
Returns the stokes flow equations' residuals from the latest solve. Residual calculations use the matrices and vectors of the discretised problem. The residuals correspond to the momentum equation and the continuity equation.

Returns r1 is the momentum equation residual r2 is the continuity equation residual

Return type (r1, r2) - 2 tuple of doubles

Notes

This method must be called collectively by all processes.

fn_bodyforce
The body force function. You may change this function directly via this property.

fn_one_on_lambda
A bulk viscosity parameter

fn_source
The volumetric source term function. You may change this function directly via this property.

fn_viscosity
The viscosity function. You may change this function directly via this property.

stokes_callback
Return the callback function used by this system

velocity_rms()
Calculates RMS velocity as follows

$$v_{rms} = \sqrt{\frac{\int_V (\mathbf{v}.\mathbf{v})\,\mathrm{d}V}{\int_V \mathrm{d}V}}$$

class underworld.systems.**SteadyStateHeat**(*temperatureField*, *fn_diffusivity*, *fn_heating=0.0*, *voronoi_swarm=None*, *conditions=[]*, *_removeBCs=True*, ***kwargs*)
Bases: underworld._stgermain.StgCompoundComponent

This class provides functionality for a discrete representation of the steady state heat equation.

The class uses a standard Galerkin finite element method to construct a system of linear equations which may then be solved using an object of the underworld.system.Solver class.

The underlying element types are determined by the supporting mesh used for the 'temperatureField'.

The strong form of the given boundary value problem, for f, h and h given, is

$$\begin{aligned} q_i &= -\alpha\, u_{,i} && (1.6)\\ q_{i,i} &= f && (1.7)\\ u &= g && \text{on } \Gamma_g \quad (1.8)\\ -q_i n_i &= h && \text{on } \Gamma_h \quad (1.9)\\ & && (1.10) \end{aligned}$$

where, α is the diffusivity, u is the temperature, f is a source term, g is the Dirichlet condition, and h is a Neumann condition. The problem boundary, Γ, admits the decomposition $\Gamma = \Gamma_g \cup \Gamma_h$ where $\emptyset = \Gamma_g \cap \Gamma_h$. The equivalent weak form is:

$$-\int_\Omega w_{,i}\, q_i\, d\Omega = \int_\Omega w\, f\, d\Omega + \int_{\Gamma_h} w\, h\, d\Gamma$$

where we must find u which satisfies the above for all w in some variational space.

Parameters

- **temperatureField** (underworld.mesh.MeshVariable) – The solution field for temperature.
- **fn_diffusivity** (underworld.function.Function) – The function that defines the diffusivity across the domain.
- **fn_heating** (underworld.function.Function) – A function that defines the heating across the domain. Optional.
- **voronoi_swarm** (underworld.swarm.Swarm) – If a voronoi_swarm is provided, voronoi type numerical integration is utilised. The provided swarm is used as the basis for the voronoi integration. If no voronoi_swarm is provided, Gauss integration is used.
- **conditions** (underworld.conditions.SystemCondition) – Numerical conditions to impose on the system. This should be supplied as the condition itself, or a list object containing the conditions.

Notes

Constructor must be called collectively by all processes.

Example

Setup a basic thermal system:

```
>>> linearMesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(4,4),
↪ minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> tField = uw.mesh.MeshVariable( linearMesh, 1 )
>>> topNodes = linearMesh.specialSets["MaxJ_VertexSet"]
>>> bottomNodes = linearMesh.specialSets["MinJ_VertexSet"]
>>> tbcs = uw.conditions.DirichletCondition(tField, topNodes + bottomNodes)
>>> tField.data[topNodes.data] = 0.0
>>> tField.data[bottomNodes.data] = 1.0
>>> tSystem = uw.systems.SteadyStateHeat(temperatureField=tField, fn_
↪diffusivity=1.0, conditions=[tbcs])
```

Example with non diffusivity:

```
>>> k = tField + 1.0
>>> tSystem = uw.systems.SteadyStateHeat(temperatureField=tField, fn_
↪diffusivity=k, conditions=[tbcs])
>>> solver = uw.systems.Solver(tSystem)
>>> solver.solve()
Traceback (most recent call last):
...
RuntimeError: Nonlinearity detected.
Diffusivity function depends on the temperature field provided to the system.
Please set the 'nonLinearIterate' solve parameter to 'True' or 'False' to␣
↪continue.
>>> solver.solve(nonLinearIterate=True)
```

fn_diffusivity
The diffusivity function. You may change this function directly via this property.

fn_heating
The heating function. You may change this function directly via this property.

class underworld.systems.**SteadyStateDarcyFlow**(*pressureField*, *fn_diffusivity*, *fn_bodyforce=None*, *voronoi_swarm=None*, *conditions=[]*, *velocityField=None*, *swarmVarVelocity=None*, *_removeBCs=True*, ***kwargs*)

Bases: underworld._stgermain.StgCompoundComponent

This class provides functionality for a discrete representation of the steady state darcy flow equation.

The class uses a standard Galerkin finite element method to construct a system of linear equations which may then be solved using an object of the underworld.system.Solver class.

The underlying element types are determined by the supporting mesh used for the 'pressureField'.

The strong form of the given boundary value problem, for f, q and h given, is

$$
\begin{aligned}
q_i &= \kappa\left(-u_{,i} + S_i\right) && (1.11)\\
q_{i,i} &= f && (1.12)\\
u &= q && (1.13)\\
-q_i n_i &= h && (1.14)
\end{aligned}
\tag{1.15}
$$

where,

- κ is the diffusivity, u is the pressure,
- S is a flow body-source, for example due to gravity,
- f is a source term, q is the Dirichlet condition, and
- h is a Neumann condition.

The problem boundary, Γ, admits the decomposition $\Gamma = \Gamma_q \cup \Gamma_h$ where $\emptyset = \Gamma_q \cap \Gamma_h$. The equivalent weak form is:

$$
-\int_\Omega w_{,i}\, q_i \, d\Omega = \int_\Omega w\, f \, d\Omega + \int_{\Gamma_h} w\, h \, d\Gamma
$$

where we must find u which satisfies the above for all w in some variational space.

Parameters

- **pressureField** (underworld.mesh.MeshVariable) – The solution field for pressure.
- **fn_diffusivity** (underworld.function.Function) – The function that defines the diffusivity across the domain.
- **fn_bodyforce** (underworld.function.Function) – A function that defines the flow body-force across the domain, for example gravity. Must be a vector. Optional.
- **voronoi_swarm** (underworld.swarm.Swarm) – A swarm with just one particle within each cell should be provided. This avoids the evaluation of the velocity on nodes and inaccuracies arising from diffusivity changes within cells. If a swarm is provided, voronoi type numerical integration is utilised. The provided swarm is used as the basis for the voronoi integration. If no voronoi_swarm is provided, Gauss integration is used.
- **conditions** (underworld.conditions.SystemCondition) – Numerical conditions to impose on the system. This should be supplied as the condition itself, or a list object containing the conditions.
- **velocityField** (underworld.mesh.MeshVariable) – Solution field for darcy flow velocity. Optional.
- **swarmVarVelocity** (*undeworld.swarm.SwarmVariable*) – If a swarm variable is provided, the velocity calculated on the swarm will be stored. This is the most representative velocity data object, as the velocity calculation occurs on the swarm, away from mesh nodes. Optional.

Notes

Constructor must be called collectively by all processes.

fn_bodyforce
The heating function. You may change this function directly via this property.

fn_diffusivity
The diffusivity function. You may change this function directly via this property.

class underworld.systems.**TimeIntegration**(*order*, ***kwargs*)
Bases: underworld._stgermain.StgCompoundComponent

Abstract class for integrating numerical objects (fields, swarms, etc.) in time.

The integration algorithm is a modified Runge Kutta method that only evaluates midpoint information varying in space - using only the present timestep solution. The order of the integration used can be 1,2,4

Parameters **order** (*int {1,2,4}*) – Defines the numerical order 'in space' of the Runge Kutta like integration scheme.

dt
Time integrator timestep size.

time
Time integrator time value.

class underworld.systems.**AdvectionDiffusion**(*phiField*, *phiDotField*, *velocityField*, *fn_diffusivity*, *fn_sourceTerm=None*, *conditions=[]*, *_allow_non_q1=False*, ***kwargs*)
Bases: underworld._stgermain.StgCompoundComponent

This class provides functionality for a discrete representation of an advection-diffusion equation.

The class uses the Streamline Upwind Petrov Galerkin SUPG method to integrate through time.

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

Parameters

- **phiField** (underworld.mesh.MeshVariable) – The concentration field, typically the temperature field
- **phiDotField** (underworld.mesh.MeshVariable) – A MeshVariable that defines the initial time derivative of the phiField. Typically 0 at the beginning of a model, e.g. phiDotField.data[:]=0 When using a phiField loaded from disk one should also load the phiDotField to ensure the solving method has the time derivative information for a smooth restart. No dirichlet conditions are required for this field as the phiField degrees of freedom map exactly to this field's dirichlet conditions, the value of which ought to be 0 for constant values of phi.
- **velocityField** (underworld.mesh.MeshVariable) – The velocity field.
- **fn_diffusivity** (underworld.function.Function) – A function that defines the diffusivity within the domain.
- **fn_sourceTerm** (underworld.function.Function) – A function that defines the heating within the domain. Optional.
- **conditions** (underworld.conditions.SystemCondition) – Numerical conditions to impose on the system. This should be supplied as the condition itself, or a list object containing the conditions.

Notes

Constructor must be called by collectively all processes.

get_max_dt()
Returns a numerically stable timestep size for the current system. Note that as a default, this method returns a value one half the size of the Courant timestep.

Returns The timestep size.

Return type float

integrate(*dt*)
Integrates the advection diffusion system through time, dt Must be called collectively by all processes.

Parameters **dt** (*float*) – The timestep interval to use

class underworld.systems.**SwarmAdvector**(*velocityField*, *swarm*, *order=2*, ***kwargs*)
Bases: underworld.systems._timeintegration.TimeIntegration

Objects of this class advect a swarm through time using the provided velocity field.

Parameters

- **velocityField** (underworld.mesh.MeshVariable) – The MeshVariable field used for evaluating the velocity field that advects the swarm particles
- **swarm** (underworld.swarm.Swarm) – Particle swarm that will be advected by the given velocity field

integrate(*dt*, *update_owners=True*)
Integrate the associated swarm in time, by dt, using the velocityfield that is associated with this class

Parameters

- **dt** (*double*) – The timestep to use in the intergration
- **update_owners** (*bool*) – If this is set to False, particle ownership (which element owns a particular particle) is not updated after advection. This is often necessary when both the mesh and particles are advecting simutaneously.

Example

```
>>> import underworld as uw
>>> import numpy as np
>>> import numpy.testing as npt
>>> from underworld import function as fn
>>> dim=2;
>>> elementMesh = uw.mesh.FeMesh_Cartesian(elementType="Q1/dQ0",␣
↪elementRes=(9,9), minCoord=(-1.,-1.), maxCoord=(1.,1.))
>>> velocityField = uw.mesh.MeshVariable( mesh=elementMesh, nodeDofCount=dim )
>>> swarm = uw.swarm.Swarm(mesh=elementMesh)
>>> particle = np.zeros((1,2))
>>> particle[0] = [0.2,-0.2]
>>> swarm.add_particles_with_coordinates(particle)
array([0], dtype=int32)
>>> velocityField.data[:]=[1.0,1.0]
>>> swarmAdvector = uw.systems.SwarmAdvector(velocityField=velocityField,␣
↪swarm=swarm, order=2)
>>> dt=1.0
>>> swarmAdvector.integrate(dt)
>>> npt.assert_allclose(swarm.particleCoordinates.data[0], [1.2,0.8], rtol=1e-
↪8, verbose=True)
```

underworld.conditions module

Implementation relating to system conditions.

Module Summary

classes:

underworld.conditions.NeumannCondition	This class defines Neumann conditions for a differential equation.
underworld.conditions.DirichletCondition	The DirichletCondition class provides the required functionality to imposed Dirichlet conditions on your differential equation system.
underworld.conditions.SystemCondition	

Module Details

classes:

class underworld.conditions.**NeumannCondition**(*variable*, *indexSetsPerDof=None*, *fn_flux=None*, *nodeIndexSet=None*)

Bases: underworld.conditions._conditions.SystemCondition

This class defines Neumann conditions for a differential equation. Neumann conditions specifiy a field's flux along a boundary.

As such the user specifices the field's flux as a uw.Function and the nodes where this flux is to be applied - similar to uw.conditions.DirichletCondtion

Parameters

- **fn_flux** (underworld.function.Function) – Function which determines flux values.
- **variable** (underworld.mesh.MeshVariable) – The variable that describes the discretisation (mesh & DOFs) for 'indexSetsPerDof'
- **indexSetsPerDof** (*list, tuple,* IndexSet) – The index set(s) which flag nodes/DOFs as Neumann conditions. Note that the user must provide an index set for each degree of freedom of the variable above. So for a vector variable of rank 2 (say Vx & Vy), two index sets must be provided (say VxDofSet, VyDofSet).

Example

Basic setup and usage of Neumann conditions:

```
>>> linearMesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(4,4),
↪ minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> velocityField = uw.mesh.MeshVariable( linearMesh, 2 )
>>> velocityField.data[:] = [0.,0.]  # set velocity zero everywhere, which will␣
↪of course include the boundaries.
>>> myFunc = (uw.function.coord()[1],0.0)
```

(continues on next page)

(continued from previous page)

```
>>> bottomWall = linearMesh.specialSets["MinJ_VertexSet"]
>>> tractionBC = uw.conditions.NeumannCondition(variable=velocityField, fn_
↪flux=myFunc, indexSetsPerDof=(None,bottomWall) )
```

fn_flux
Get the underworld.Function that defines the flux

class underworld.conditions.**DirichletCondition**(*variable*, *indexSetsPerDof*)
Bases: underworld.conditions._conditions.SystemCondition

The DirichletCondition class provides the required functionality to imposed Dirichlet conditions on your differential equation system.

The user is simply required to flag which nodes/DOFs should be considered by the system to be a Dirichlet condition. The values at the Dirichlet nodes/DOFs is then left untouched by the system.

Parameters

- **variable** (underworld.mesh.MeshVariable) – This is the variable for which the Dirichlet condition applies.
- **indexSetsPerDof** (*list, tuple,* IndexSet) – The index set(s) which flag nodes/DOFs as Dirichlet conditions. Note that the user must provide an index set for each degree of freedom of the variable. So for a vector variable of rank 2 (say Vx & Vy), two index sets must be provided (say VxDofSet, VyDofSet).

Notes

Note that it is necessary for the user to set the required value on the variable, possibly via the numpy interface.

Constructor must be called collectively all processes.

Example

Basic setup and usage of Dirichlet conditions:

```
>>> linearMesh = uw.mesh.FeMesh_Cartesian( elementType='Q1/dQ0', elementRes=(4,4),
↪ minCoord=(0.,0.), maxCoord=(1.,1.) )
>>> velocityField = uw.mesh.MeshVariable( linearMesh, 2 )
>>> velocityField.data[:] = [0.,0.]  # set velocity zero everywhere, which will␣
↪of course include the boundaries.
>>> IWalls = linearMesh.specialSets["MinI_VertexSet"] + linearMesh.specialSets[
↪"MaxI_VertexSet"]  # get some wall index sets
>>> JWalls = linearMesh.specialSets["MinJ_VertexSet"] + linearMesh.specialSets[
↪"MaxJ_VertexSet"]
>>> freeSlipBC = uw.conditions.DirichletCondition(velocityField, (IWalls,JWalls)␣
↪)  # this will give free slip sides
>>> noSlipBC = uw.conditions.DirichletCondition(velocityField, (IWalls+JWalls,
↪IWalls+JWalls) )  # this will give no slip sides
```

class underworld.conditions.**SystemCondition**(*variable*, *indexSetsPerDof*)
Bases: underworld._stgermain.StgCompoundComponent

indexSetsPerDof
See class constructor for details.

variable
See class constructor for details.

functions:

underworld.matplotlib_inline	This function simply enables Jupyter Notebook inlined matplotlib results.
underworld.nProcs	Returns the number of processes being utilised by the simulation.
underworld.rank	Returns the rank of the current process.
underworld.barrier	Creates an MPI barrier.

1.1.2 Module Details

functions:

underworld.**matplotlib_inline**()
This function simply enables Jupyter Notebook inlined matplotlib results. This function should be called at the start of your notebooks as a replacement for the Jupyter Notebook *%matplotlib inline* magic. It provides the same functionality, however it allows notebooks to be converted to python without having to explicitly remove these calls.

underworld.**nProcs**()
Returns the number of processes being utilised by the simulation.

Returns Number of processors.

Return type unsigned

underworld.**rank**()
Returns the rank of the current process.

Returns Rank of current process.

Return type unsigned

underworld.**barrier**()
Creates an MPI barrier. All processes wait here for others to catch up.

1.2 glucifer module

The glucifer module provides visualisation algorithms for Underworld.

Visualisation data is generated in parallel, with each processes generating the necessary data for its part of the domain. This data is written into a data file.

Actual rendering is performed in serial using the LavaVu rendering engine.

glucifer provides many flexible rendering options, including client-server based operation for remote usage. Users may choose to renderer outputs to raster images, or save a database file for later rendering. For those working in the Jupyter environment, glucifer will inline rendered images or even interactive webgl frames (still experimental).

1.2.1 Module Summary

submodules:

glucifer.objects module

Module Summary

classes:

glucifer.objects.ColourBar	The ColourBar drawing object draws a colour bar for the provided colour map.
glucifer.objects.VectorArrows	This drawing object class draws vector arrows corresponding to the provided vector field.
glucifer.objects.Points	This drawing object class draws a swarm of points.
glucifer.objects.Sampler	The Sampler class provides functionality for sampling a field at a number of provided vertices.
glucifer.objects.Surface	This drawing object class draws a surface using the provided scalar field.
glucifer.objects.Volume	This drawing object class draws a volume using the provided scalar field.
glucifer.objects.Mesh	This drawing object class draws a mesh.
glucifer.objects.Contours	This drawing object class draws contour lines in a cross section using the provided scalar field.
glucifer.objects.ColourMap	The ColourMap class provides functionality for mapping colours to numerical values.
glucifer.objects.CrossSection	This drawing object class defines a cross-section plane, derived classes plot data over this cross section
glucifer.objects.Drawing	This is the base class for all drawing objects but can also be instantiated as is for direct/custom drawing.
glucifer.objects.IsoSurface	This drawing object class draws an isosurface using the provided scalar field.

Module Details

classes:

class glucifer.objects.**ColourBar**(*colourMap*, **args*, ***kwargs*)

Bases: *glucifer.objects.Drawing*

The ColourBar drawing object draws a colour bar for the provided colour map.

Parameters **colourMap** (glucifer.objects.ColourMap) – Colour map for which the colour bar will be drawn.

class glucifer.objects.**VectorArrows**(*mesh*, *fn*, *resolution=[16, 16, 16]*, *autoscale=True*, **args*, ***kwargs*)

Bases: glucifer.objects._GridSampler3D

This drawing object class draws vector arrows corresponding to the provided vector field.

See parent class for further parameter details. Also see property docstrings.

Parameters

- **mesh** (underworld.mesh.FeMesh) – Mesh over which vector arrows are rendered.
- **fn** (underworld.function.Function) – Function used to determine vectors to render. Function should return a vector of floats/doubles of appropriate dimensionality.
- **arrowHead** (*float*) – The size of the head of the arrow. If > 1.0 is ratio to arrow radius If in range [0.,1.] is ratio to arrow length
- **scaling** (*float*) – Scaling for entire arrow.
- **autoscale** (*bool*) – Scaling based on field min/max
- **glyphs** (*int*) – Type of glyph to render for vector arrow. 0: Line, 1 or more: 3d arrow, higher number => better quality.
- **resolution** (*list (unsigned)*) – Number of samples in the I,J,K directions.

class glucifer.objects.**Points**(*swarm*, *fn_colour=None*, *fn_mask=None*, *fn_size=None*, *colourVariable=None*, *colourBar=True*, **args*, ***kwargs*)

Bases: *glucifer.objects.Drawing*

This drawing object class draws a swarm of points.

See parent class for further parameter details. Also see property docstrings.

Parameters

- **swarm** (underworld.swarm.Swarm) – Swarm which provides locations for point rendering.
- **fn_colour** (underworld.function.Function) – Function used to determine colour to render particle. This function should return float/double values.
- **fn_mask** (underworld.function.Function) – Function used to determine if a particle should be rendered. This function should return bool values.
- **fn_size** (underworld.function.Function) – Function used to determine size to render particle. This function should return float/double values.

class glucifer.objects.**Sampler**(*mesh*, *fn*, **args*, ***kwargs*)

Bases: *glucifer.objects.Drawing*

The Sampler class provides functionality for sampling a field at a number of provided vertices.

Parameters

- **vertices** (*list, array*) – List of vertices to sample the field at, either a list or numpy array
- **mesh** (underworld.mesh.FeMesh) – Mesh over which the values are sampled
- **fn** (underworld.function.Function) – Function used to get the sampled values.

class glucifer.objects.**Surface**(*mesh*, *fn*, *drawSides='xyzXYZ'*, *colourBar=True*, **args*, ***kwargs*)

Bases: *glucifer.objects.CrossSection*

This drawing object class draws a surface using the provided scalar field.

See parent class for further parameter details. Also see property docstrings.

Parameters

- **mesh** (underworld.mesh.FeMesh) – Mesh over which cross section is rendered.
- **fn** (underworld.function.Function) – Function used to determine values to render.

- **drawSides** (*str*) – Sides (x,y,z,X,Y,Z) for which the surface should be drawn. For example, "xyzXYZ" would render the provided function across all surfaces of the domain in 3D. In 2D, this object always renders across the entire domain.

class glucifer.objects.**Volume**(*mesh, fn, resolution=[64, 64, 64], colourBar=True, *args, **kwargs*)

Bases: glucifer.objects._GridSampler3D

This drawing object class draws a volume using the provided scalar field.

See parent class for further parameter details. Also see property docstrings.

Parameters

- **mesh** (underworld.mesh.FeMesh) – Mesh over which object is rendered.
- **fn** (underworld.function.Function) – Function used to determine colour values. Function should return a vector of floats/doubles of appropriate dimensionality.
- **resolution** (*list (unsigned)*) – Number of samples in the I,J,K directions.

class glucifer.objects.**Mesh**(*mesh*, *nodeNumbers=False*, *segmentsPerEdge=1*, **args*, ***kwargs*)

Bases: *glucifer.objects.Drawing*

This drawing object class draws a mesh.

See parent class for further parameter details. Also see property docstrings.

Parameters

- **mesh** (underworld.mesh.FeMesh) – Mesh to render.
- **nodeNumbers** (*bool*) – Bool to determine whether global node numbers should be rendered.
- **segmentsPerEdge** (*unsigned*) – Number of segments to render per cell/element edge. For higher order mesh, more segments are useful to render mesh curvature correctly.

class glucifer.objects.**Contours**(*mesh*, *fn*, *labelFormat=''*, *unitScaling=1.0*, *interval=0.33*, *limits=(0.0, 0.0)*, **args*, ***kwargs*)

Bases: *glucifer.objects.CrossSection*

This drawing object class draws contour lines in a cross section using the provided scalar field.

See parent class for further parameter details. Also see property docstrings.

Parameters

- **mesh** (underworld.mesh.FeMesh) – Mesh over which cross section is rendered.
- **fn** (underworld.function.Function) – Function used to determine values to render.
- **labelFormat** (*str*) – Format string (printf style) used to print a contour label, eg: " %g K"
- **unitScaling** – Scaling factor to apply to value when printing labels
- **interval** (*float*) – Interval between contour lines
- **limits** (*tuple, list*) – User defined minimum and maximum limits for the contours. Provided as a tuple/list of floats (minValue, maxValue). If none is provided, the limits will be determined automatically.

class glucifer.objects.**ColourMap**(*colours='diverge'*, *valueRange=None*, *logScale=False*, *discrete=False*, ***kwargs*)

Bases: underworld._stgermain.StgCompoundComponent

The ColourMap class provides functionality for mapping colours to numerical values.

Parameters

- **colours** (*str, list*) – List of colours to use for drawing object colour map. Provided as a string or as a list of strings. Example, "red blue", or ["red", "blue"]
- **valueRange** (*tuple, list*) – User defined value range to apply to colour map. Provided as a tuple of floats (minValue, maxValue). If none is provided, the value range will be determined automatically.
- **logScale** (*bool*) – Bool to determine if the colourMap should use a logarithmic scale.
- **discrete** (*bool*) – Bool to determine if a discrete colour map should be used. Discrete colour maps do not interpolate between colours and instead use nearest neighbour for colouring.

class glucifer.objects.**CrossSection**(*mesh, fn, crossSection='', resolution=[100, 100, 1], colourBar=True, offsetEdges=None, onMesh=False, *args, **kwargs*)

Bases: *glucifer.objects.Drawing*

This drawing object class defines a cross-section plane, derived classes plot data over this cross section

See parent class for further parameter details. Also see property docstrings.

Parameters

- **mesh** (underworld.mesh.FeMesh) – Mesh over which cross section is rendered.
- **fn** (underworld.function.Function) – Function used to determine values to render.
- **crossSection** (*str*) – Cross Section definition, eg. z=0.
- **resolution** (*list (unsigned)*) – Surface sampling resolution.
- **onMesh** (*boolean*) – Sample the mesh nodes directly, as opposed to sampling across a regular grid. This flag should be used in particular where a mesh has been deformed.

crossSection

crossSection (str) – Cross Section definition, eg;: z=0.

class glucifer.objects.**Drawing**(*name=None, colours=None, colourMap='', colourBar=False, valueRange=None, logScale=False, discrete=False, *args, **kwargs*)

Bases: underworld._stgermain.StgCompoundComponent

This is the base class for all drawing objects but can also be instantiated as is for direct/custom drawing.

Note that the defaults here are often overridden by the child objects.

Parameters

- **colours** (*str, list.*) – See ColourMap class docstring for further information
- **colourMap** (glucifer.objects.ColourMap) – A ColourMap object for the object to use. This should not be specified if 'colours' is specified.
- **opacity** (*float*) – Opacity of object. If provided, must take values from 0. to 1.
- **colourBar** (*bool*) – Bool to determine if a colour bar should be rendered.
- **valueRange** (*tuple, list*) – See ColourMap class docstring for further information
- **logScale** (*bool*) – See ColourMap class docstring for further information

- **discrete** (*bool*) – See ColourMap class docstring for further information

colourBar
colourBar (object) – return colour bar of drawing object, create if doesn't yet exist.

colourMap
colourMap (object) – return colour map of drawing object

label(*text*, *pos=(0.0, 0.0, 0.0)*, *font='sans'*, *scaling=1*)
Writes a label string

Parameters

- **text** (*str*) – label text.
- **pos** (*tuple*) – X,Y,Z position to place the label.
- **font** (*str*) – label font (small/fixed/sans/serif/vector).
- **scaling** (*float*) – label font scaling (for "vector" font only).

line(*start=(0.0, 0.0, 0.0)*, *end=(0.0, 0.0, 0.0)*)
Draws a line

Parameters

- **start** (*tuple*) – X,Y,Z position to start line
- **end** (*tuple*) – X,Y,Z position to end line

point(*pos=(0.0, 0.0, 0.0)*)
Draws a point

Parameters **pos** (*tuple*) – X,Y,Z position to place the point

vector(*position=(0.0, 0.0, 0.0)*, *vector=(0.0, 0.0, 0.0)*)
Draws a vector

Parameters

- **position** (*tuple*) – X,Y,Z position to centre vector on
- **vector** (*tuple*) – X,Y,Z vector value

class glucifer.objects.**IsoSurface**(*mesh, fn, fn_colour=None, resolution=[64, 64, 64], colourBar=True, isovalue=None, *args, **kwargs*)
Bases: *glucifer.objects.Volume*

This drawing object class draws an isosurface using the provided scalar field.

See parent class for further parameter details. Also see property docstrings.

Parameters

- **mesh** (underworld.mesh.FeMesh) – Mesh over which object is rendered.
- **fn** (underworld.function.Function) – Function used to determine surface position. Function should return a vector of floats/doubles.
- **fn_colour** (underworld.function.Function) – Function used to determine colour of surface.
- **resolution** (*list (unsigned)*) – Number of samples in the I,J,K directions.
- **isovalue** (*float*) – Isovalue to plot.
- **isovalues** (*list of float*) – List of multiple isovalues to plot.

classes:

glucifer.Store	The Store class provides a database which stores gLucifer drawing objects as they are rendered in figures.
glucifer.Figure	The Figure class provides a window within which gLucifer drawing objects may be rendered.

1.2.2 Module Details

classes:

class glucifer.**Store**(*filename=None*, *split=False*, ***kwargs*)

Bases: underworld._stgermain.StgCompoundComponent

The Store class provides a database which stores gLucifer drawing objects as they are rendered in figures. It also provides associated routines for saving and reloading this database to external files

In addition to parameter specification below, see property docstrings for further information.

Parameters

- **filename** (*str*) – Filename to use for a disk database, default is in memory only unless saved.
- **split** (*bool*) – Set to true to write a separate database file for each timestep visualised
- **view** (*bool*) – Set to true and pass filename if loading a saved database for revisualisation

Example

Create a database:

```
>>> import glucifer
>>> store = glucifer.Store()
```

Optionally provide a filename so you don't need to call save later (no extension)

```
>>> store = glucifer.Store('myvis')
```

Pass to figures when creating them (Providing a name allows you to revisualise the figure from the name)

```
>>> fig = glucifer.Figure(store, name="myfigure")
```

When figures are rendered with show() or save(imgname), they are saved to storage If you don't need to render an image but still want to store the figure to view later, just call save() without a filename

```
>>> fig.save()
```

Save the database (only necessary if no filename provided when created)

```
>>> dbfile = store.save("myvis")
```

empty()

Empties all the cached drawing objects

save (*filename*)
Saves the database to the provided filename.

Parameters **filename** (*str*) – Filename to save file to. May include an absolute or relative path.

class glucifer.**Figure**(*store=None*, *name=None*, *figsize=None*, *boundingBox=None*, *facecolour='white'*, *edgecolour='black'*, *title=''*, *axis=False*, *quality=1*, **args*, ***kwargs*)
Bases: dict

The Figure class provides a window within which gLucifer drawing objects may be rendered. It also provides associated routines for image generation and visualisation.

In addition to parameter specification below, see property docstrings for further information.

Parameters

- **store** (glucifer.Store) – Database to collect visualisation data, this may be shared among figures to collect their data into a single file.
- **name** (*str*) – Name of this figure, optional, used for revisualisation of stored figures.
- **resolution** (*tuple*) – Image resolution provided as a tuple.
- **boundingBox** (*tuple*) – Tuple of coordinate tuples defining figure bounding box. For example ((0.1,0.1), (0.9,0.9))
- **facecolour** (*str*) – Background colour for figure.
- **edgecolour** (*str*) – Edge colour for figure.
- **title** (*str*) – Figure title.
- **axis** (*bool*) – Bool to determine if figure axis should be drawn.
- **quality** (*unsigned*) – Antialiasing oversampling quality. For a value of 2, the image will be rendered at twice the resolution, and then downsampled. Setting this to 1 disables antialiasing, values higher than 3 are not recommended..
- **properties** (*str*) – Further properties to set on the figure.

Example

Create a figure:

```
>>> import glucifer
>>> fig = glucifer.Figure()
```

We need a mesh

```
>>> import underworld as uw
>>> mesh = uw.mesh.FeMesh_Cartesian()
```

Add drawing objects:

```
>>> fig.Surface( mesh, 1.)
```

Draw image (note, in a Jupyter notebook, this will render the image within the notebook).

```
>>> fig.show()
<IPython.core.display.HTML object>
```

Save the image

```
>>> imgfile = fig.save("test_image")
```

Clean up:

```
>>> if imgfile:
...     import os;
...     os.remove( imgfile )
```

append(*drawingObject*)
Add a drawing object to the figure.

Parameters **drawingObject** (glucifer.objects.Drawing) – The drawing object to add to the figure

axis
axis – Axis enabled if true.

clear() → None. Remove all items from D.

close_viewer()
Close the viewer.

edgecolour
edgecolour – colour of figure border.

facecolour
facecolour – colour of face background.

objects
objects – list of objects to be drawn within the figure.

open_viewer(*args=[]*, *background=True*)
Open the external viewer.

properties
properties (dict) – visual properties of viewport, passed to LavaVu to control rendering output of figure.

When using the property setter, new properties are set, overwriting any duplicate keys but keeping existing values otherwise.

resolution
resolution (tuple(int,int)) – size of window in pixels.

save(*filename=None*, *size=(0, 0)*, *type='Image'*)
Saves the generated image to the provided filename or the figure to the database.

Parameters

- **filename** (*str*) – Filename to save file to. May include an absolute or relative path.
- **(tuple(int,int))** (*size*) – If omitted, simply saves the figure data without generating an image
- **type** (*str*) – Type of visualisation to save ('Image' or 'WebGL').

Returns **filename** – The final filename (including extension) used to save the image will be returned. Note that only the root process will return this filename. All other processes will not return anything.

Return type str

script (*cmd=None*)
Append to or get contents of the saved script.

Parameters **cmd** (*str*) – Command to add to script.

send_command (*cmd*, *retry=True*)
Run command on an open viewer instance.

Parameters **cmd** (*str*) – Command to send to open viewer.

show (*type='Image'*)
Shows the generated image inline within an ipython notebook.

Parameters

- **type** (*str*) – Type of visualisation to display ('Image' or 'WebGL').
- **IPython is installed, displays the result image or WebGL content inline** (*If*) –
- **IPython is not installed, this method will call the default image/web** (*If*) –
- **routines to save the result with a default filename in the current directory** (*output*) –

step
step (int) – current timestep

title
title – a title for the image.

viewer (*new=False*, **args*, ***kwargs*)
Return viewer instance.

Parameters **new** (*boolean*) – If True, a new viewer instance will always be returned Otherwise the existing instance will be used if available

window (**args*, ***kwargs*)
Open an inline viewer.

This returns a new LavaVu instance to display the figure and opens it as an interactive viewing window.

genindex

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

W

X