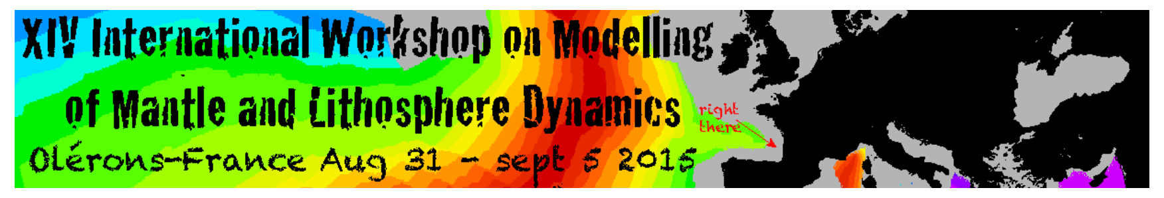

XIV International Workshop on Modelling
of Mantle and Lithosphere Dynamics
Olérons-France Aug 31 - sept 5 2015
right there

# Modelling in tomorrrow's technological landscape - Unveiling Underworld2

Quenette S. (1), Moresi L. (2), Mansour J. (1), Revote J. (1)
(1) Monash eResearch Centre, Monash University, Australia
(2) School of Earth Sciences, University of Melbourne, Australia

## Abstract

Understanding of mantle and lithosphere dynamics, distinct or coupled, attracts certain modes of scientific discovery. It is not only engineering - as the constitutive laws for minerals under these conditions are difficult to measure in the laboratory. It is not only geology - as the precise characterisation of the Earth is impossible without interpretations on sensor observations. Further it is intrinsically multi-scale, where chemical and physical effects at the centimetre scale effect structures as broad as plates and mantle flow. The mode of “modelling” dominates our discovery process. Do we understand how this mode will continue in the changing technological landscape?

Over the period of two decades ago to one decade ago, increased accessibility to personal computing led to a golden age of Earth dynamics discovery. Fundamental processes were contributed by many, all relying on computation of numerical systems of scale or complexity that required a computer. Invariably we must thank the innovation of Moore’s Law - over 50years of sustained 50%-compounded yearly growth in computing capability - for enabling such computing at this time.

Increasingly a sole phd student could no longer write their own code in isolation and from scratch. Despite the readily available computing power, the total model required had become sufficiently complex that collaboration about codes became necessary. About a decade ago, the very first versions of software Underworld was released. And along with other codes, a second golden age was born, where many discoveries about 3-dimensional effects together with processes across scales have arisen. Hence innovations of Underworld were enabled by software for complexity - allowing more expertise and more libraries to readily contribute. Underworld in particular focused on distributed parallel computing, increasingly complex numerical methods, and increasingly complicated physics. It is by no means perfect, but has pioneered avenues of methods and discoveries.

Today, Moore’s Law is ailing, and the only man-made innovation that is remotely close to it is the Internet of Things. Sensor capabilities are an honourable second (approximately 25%-compounded yearly growth over the last couple of decades). Together with increasing storage technologies, they are fuelling the data-deluge, and in-turn, data-driven scientific discovery (clearly being enjoyed by the geophysical disciplines). They are also fuelling organisational and asset (code in our case) permeability. We are no longer needing just massive amounts of computing for the complex numerical system, but an eco system of computing that enables rapid experimentation and high throughput on data. In short, increasingly innovation at large will drive towards codes and environments that assimilate with data, and codes and environments that have accessible insides (rather than those that are one monolithic box or function).

Here we unveil Underworld2, a cloud ready, python-based code for mantle and lithosphere dynamics discovery, spanning tutorials, data assimilation and in-line analysis. We hope that nothing is lost from Underworld1 but that Underworld, and its subparts, are accessible to the researcher with data.

## Modelling and big data

In the past, developing "f" was  our only focus...

$$y = \underset{\underset{\text{"big"}}{\uparrow}}{f}\ (x)$$

But now, developing "f" and how it is assimilated are both a focus ...

$$\underset{\underset{\text{"big"}}{\uparrow}}{y_o} \equiv y_f = \underset{\underset{\text{"big"}}{\uparrow}}{f}\ (x)$$

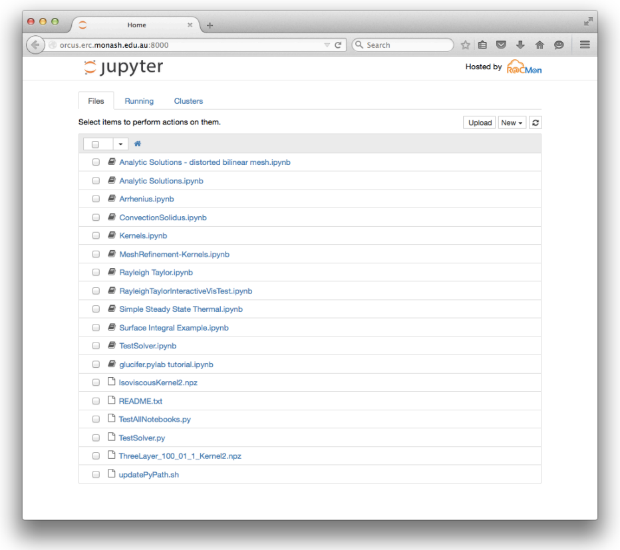
## Underworld2

Underword2 is based on Underword(1), and hence inherit the capabilities of 3D parallel geodynamics problems.

However, it introduces a python interface that replaces the XML-based model configuration and parameter input system.

glucifer is now not only - render inline with compute and render via viewer, it now has a HTML5 version, meaning it is available on the web.

Using docker and ipython/jupyter-notebooks, Underworld2 is now available on the cloud, with documented problem tutorials. (see http://orcus.erc.monash.edu.au:8000)



Select items to perform actions on them.

- Analytic Solutions - distorted bilinear mesh.ipynb
- Analytic Solutions.ipynb
- Arrhenius.ipynb

### Rayleigh Taylor Instability Benchmark

This notebook implements the isoviscous thermochemical convection benchmark from van Keken et al (1997).

$$\nabla \cdot (\eta \nabla \dot{\varepsilon}) - \nabla p = (Ra_T T + Ra_\Gamma \Gamma)\,\hat{\mathbf{z}}$$

$$\nabla \cdot \mathbf{v} = 0$$

The thermal and compositional evolution is controlled by advection and (thermal) diffusion

$$\frac{DT}{Dt} = \nabla^2 T$$

$$\frac{D\Gamma}{Dt} = 0$$

Thermal and compositional Rayleigh numbers are defined by

$$Ra_T = \frac{g\rho\alpha\Delta T h^3}{\kappa\eta_r}; \quad Ra_\Gamma = \frac{g\Delta\rho_\Gamma h^3}{\kappa\eta_r}$$

van Keken, P. E., S. D. King, H. Schmeling, U. R. Christensen, D. Neumeister, and M. P. Doin (1997), A comparison of methods for the modeling of thermochemical convection, J. Geophys. Res., 102(B10), 22477, doi:10.1029/97JB01353.

In [1]:
```
import underworld as uw
import math
from underworld import function as fn
import glucifer.pylab as plt
```

In [2]:
```
dim = 2
```

In [3]:
```
# create mesh objects
elementMesh = uw.mesh.FeMesh_Cartesian( elementType=("linear","constant"),
                                         elementRes=(64,64),
                                           minCoord=(0.,0.),
                                           maxCoord=(0.9142,1.) )

linearMesh   = elementMesh
constantMesh = elementMesh.subMesh
```

In [4]:
```
# create fevariables
velocityField = uw.fevariable.FeVariable( feMesh=linearMesh,   nodeDofCount
pressureField = uw.fevariable.FeVariable( feMesh=constantMesh, nodeDofCount
```

In [5]:
```
# Initialise data.. Note that we are also setting boundary conditions here
velocityField.data[:] = [0.,0.]
pressureField.data[:] = 0.
```

In [6]:
```
# Get list of special sets.
# These are sets of vertices on the mesh. In this case we want to set them
linearMesh.specialSets.keys()
```

Out[6]:
```
['MaxI_VertexSet',
 'MinI_VertexSet',
 'AllWalls',
 'MinJ_VertexSet',
 'MaxJ_VertexSet',
 'Empty']
```

In [7]:
```
# Get the actual sets
#
#  HJJJJJJH
#  I      I
#  I      I
#  I      I
#  HJJJJJJH
#
#  Note that H = I & J

# Note that we use operator overloading to combine sets
IWalls = linearMesh.specialSets["MinI_VertexSet"] + linearMesh.specialSets[
JWalls = linearMesh.specialSets["MinJ_VertexSet"] + linearMesh.specialSets[
```

In [8]:
```
# You can view the contents of the sets directly
IWalls
```

Out[8]:
```
FeMesh_IndexSet([   0,   64,   65,  129,  130,  194,  195,  259,  260,  3
24,  325,
```

In [9]:
```
JWalls
```

Out[9]:
```
FeMesh_IndexSet([   0,    1,    2,    3,    4,    5,    6,    7,    8,
 9,   10,
```

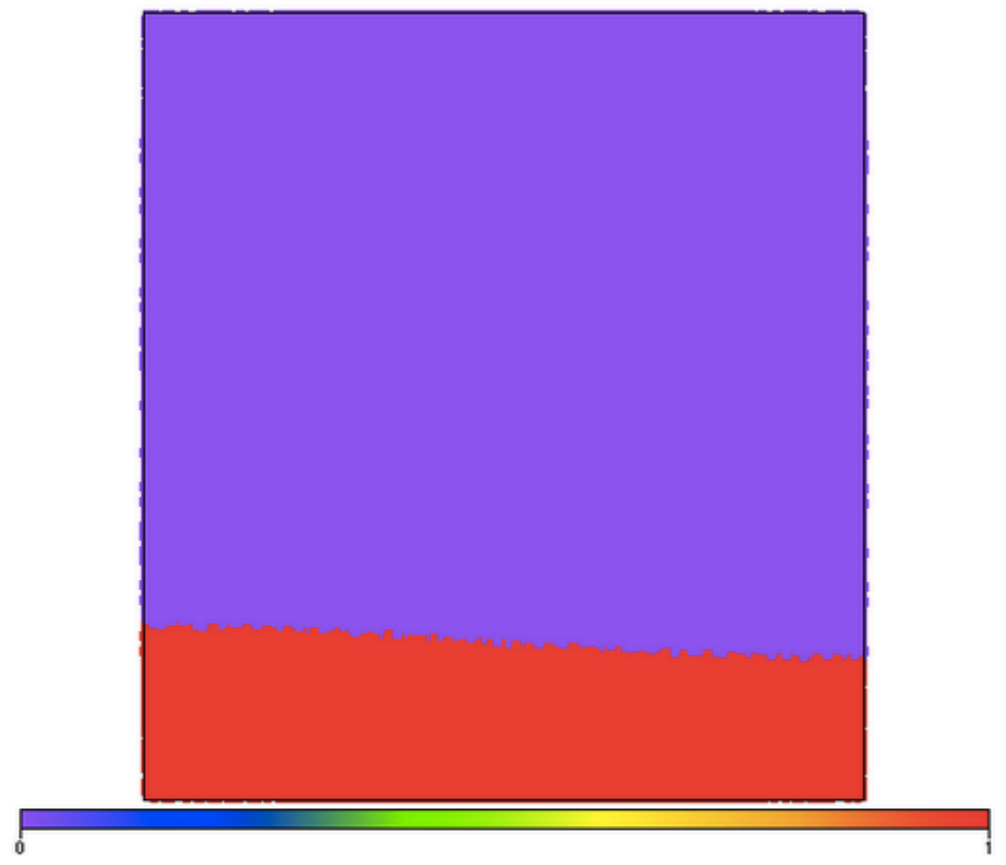
In [10]:
```
# Now setup the dirichlet boundary condition
# Note that through this object, we are flagging to the system
# that these nodes are to be considered as boundary conditions.
# Also note that we provide a tuple of sets.. One for the Vx, one for Vy.
AllWalls = IWalls + JWalls

freeslipBC = uw.conditions.DirichletCondition(     variable=velocityField,
                                              nodeIndexSets=(AllWalls,JWall
```

In [11]:
```
# We create swarms of particles which can advect, and which may determine '
gSwarm = uw.swarm.Swarm( feMesh=elementMesh )

# Now we add a data variable which will store an index to determine materia
materialVariable = gSwarm.add_variable( dataType="char", count=1 )

# Layouts are used to populate the swarm across the whole domain
# Create the layout object
layout = uw.swarm.layouts.GlobalSpaceFillerLayout( swarm=gSwarm, particlesP
# Now use it to populate.
gSwarm.populate_using_layout( layout=layout )
```

In [12]:
```
# Lets initialise the 'materialVariable' data to represent two different ma
materialHeavyIndex = 0
materialLightIndex = 1

# Now let's initialize the materialVariable with the required perturbation
import math
wavelength = 1.8284
amplitude  = 0.02
offset     = 0.2
k = 2.*math.pi / wavelength
coordinate = fn.input()
materialVariable.data[:] = fn.branching.conditional(
    [ ( offset + amplitude*fn.math.cos( k*coordinate[0] ) > coordinate[1] ,
      (                                                            True ,
```

In [13]:
```
# visualise
fig1 = plt.Figure()
fig1.Points( swarm=gSwarm, colourVariable=materialVariable, pointSize=5.0 )
fig1.show()
```

Out[13]:



In [14]:
```
# We create some functions here.
# The Map function allows as to create 'per material' type behaviour.
# It requires a 'keyFunc', which is first evaluate to determine the key
# into the map, and then once the key is determine, we retrieve the value (
# it maps to, and evaluate that.

# Here we set a viscosity value of '1.' for both materials
viscosityMapFn = fn.branching.map( keyFunc = materialVariable,
                         mappingDict = { materialLightIndex:1., materialHea
# Here we set a density of '0.' for the lightMaterial, and '1.' for the hea
densityFn      = fn.branching.map( keyFunc = materialVariable,
                         mappingDict = { materialLightIndex:0., materialHea
# Define our gravity using a python tuple (this will be automatically conve
gravity = ( 0.0, -1.0 )
# now create a buoyancy force vector.. the gravity tuple is converted to a
# here via operator overloading
buoyancyFn = gravity*densityFn
```

In [15]:
```
# Setup a stokes system
# For PIC style integration, we include a swarm for the a PIC integration s
# For gauss integration, simple do not include the swarm. Nearest neighbour
stokesPIC = uw.systems.Stokes(velocityField=velocityField,
                              pressureField=pressureField,
                              swarm=gSwarm,
                              conditions=[freeslipBC,],
                              viscosityFn=viscosityMapFn,
                              bodyForceFn=buoyancyFn )
```

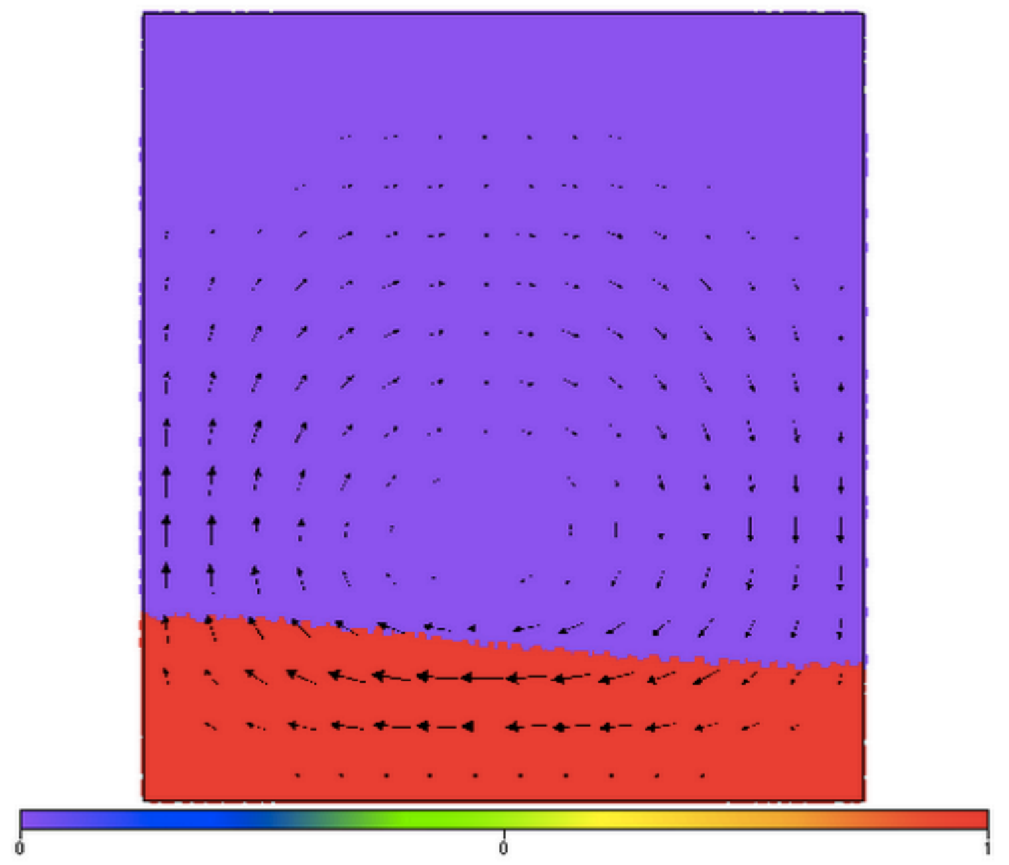
In [16]:
```
# Create advector objects to advect the swarms. We specify second order int
advector = uw.systems.SwarmAdvector( swarm=gSwarm, velocityField=velocityFi
# Also create some integral objects which are used to calculate statistics.
v2sum_integral  = uw.utils.Integral( feMesh=linearMesh, fn=fn.math.dot(velc
volume_integral = uw.utils.Integral( feMesh=linearMesh, fn=1. )
```

In [17]:
```
# Stepping. Initialise time and timestep.
time = 0.
step = 0
# Perform 3 steps
while step<3:
    # Get solution for initial configuration
    stokesPIC.solve()
    # Retrieve the maximum possible timestep for the advection system.
    dt = advector.get_max_dt()
    # Advect using this timestep size
    advector.integrate(dt)
    # Calculate the RMS velocity
    v2sum = v2sum_integral.integrate()
    volume = volume_integral.integrate()
    vrms = math.sqrt(v2sum[0]/volume[0])
    print 'step =',step, 'time =', time, 'vrms = ', vrms
    # Increment
    time += dt
    step += 1
```

```
step = 0 time = 0.0 vrms =  0.00018493383262
step = 1 time = 17.0335742504 vrms =  0.00021780294623
step = 2 time = 31.4523449745 vrms =  0.000254973326893
```

In [18]:
```
fig1 = plt.Figure()
fig1.Points( swarm=gSwarm, colourVariable=materialVariable, pointSize=5.0 )
fig1.VectorArrows( velocityField, elementMesh, lengthScale=100, arrowHeadSi
fig1.show()
```

Out[18]:



### Hotplate (steady state thermal)

This notebook implements the heat flow eqautions over (a) a homogenous hot plate, (b) over materials of non-constant properties, and (c) over many materials.

Problem (a) implements

$$\nabla \cdot q = -A$$

where

$$\underline{q} = \kappa \nabla(T)$$

In [1]:
```
import underworld as uw
import glucifer.pylab as plt
import underworld.function as fn
```

In [2]:
```
mesh = uw.mesh.FeMesh_Cartesian('linear', (32,32), (-1.,-1.), (1.,1.))
temperatureField = uw.fevariable.FeVariable(mesh,1)
```

In [3]:
```
mesh.specialSets.keys()
```

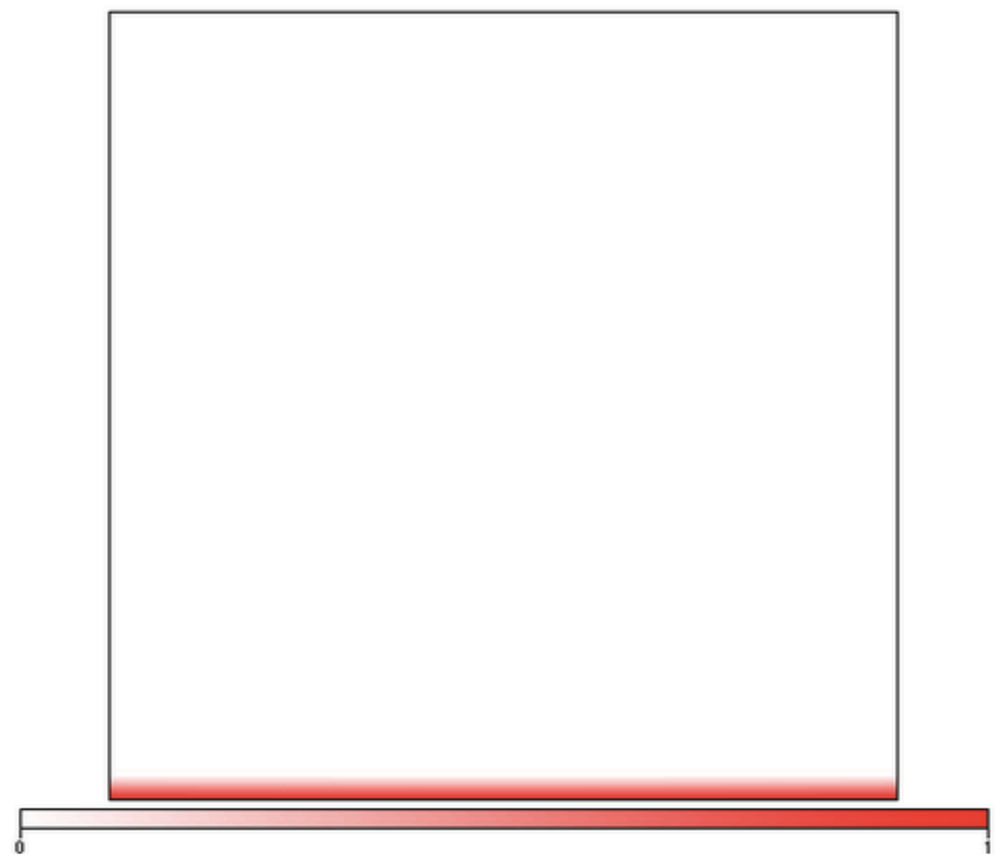
Out[3]:
```
['MaxI_VertexSet',
 'MinI_VertexSet',
 'AllWalls',
 'MinJ_VertexSet',
 'MaxJ_VertexSet',
 'Empty']
```

In [4]:
```
# declare which nodes are to be considered as boundary nodes
topNodes = mesh.specialSets["MaxJ_VertexSet"]
bottomNodes = mesh.specialSets["MinJ_VertexSet"]
conditions = uw.conditions.DirichletCondition(temperatureField, topNodes +
```

In [5]:
```
# init tempfield to zero everywhere
temperatureField.data[:] = 0.
# setup required values on boundary nodes
temperatureField.data[topNodes.data]    = 0.
temperatureField.data[bottomNodes.data] = 1.
```

In [6]:
```
# lets take a look
fig = plt.Figure()
fig.Surface(temperatureField,mesh, colours=['white','red'])
fig.show()
```

Out[6]:

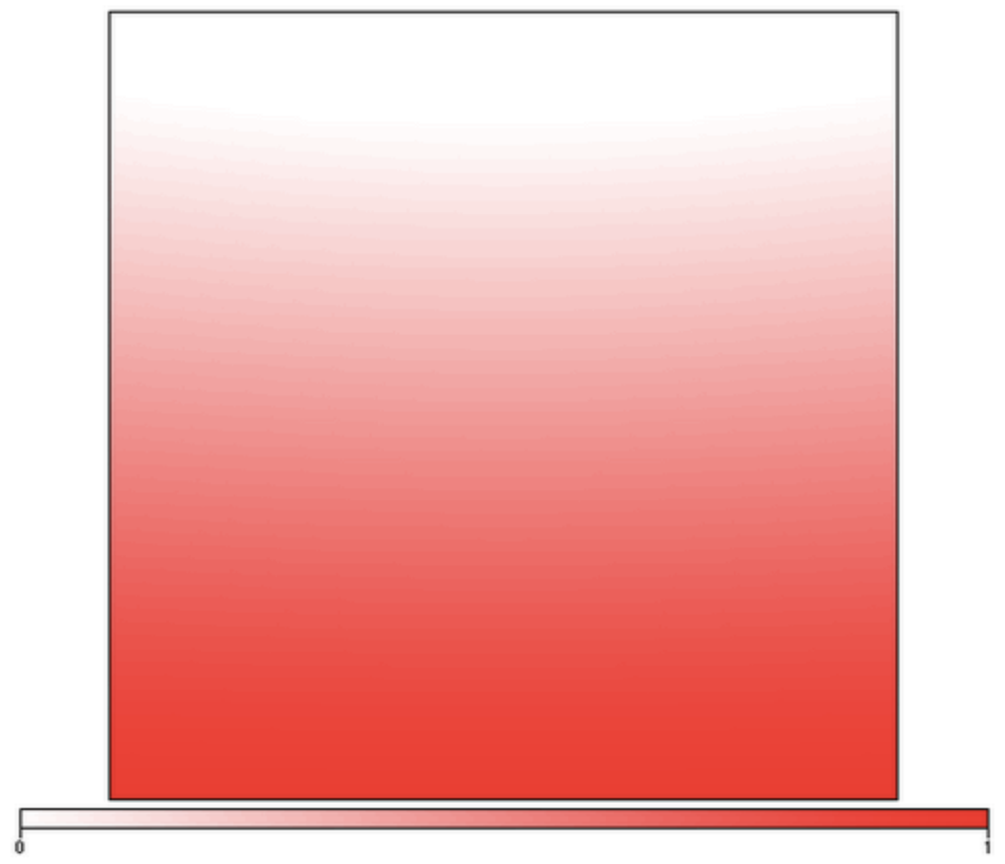


In [7]:
```
# setup system
steadyStateThermal = uw.systems.Thermal(temperatureField,1.,conditions=[con
```

In [8]:
```
# solve!
steadyStateThermal.solve()
```

In [9]:
```
# now lets look
fig = plt.Figure()
fig.Surface(temperatureField,mesh, colours=['white','red'])
fig.show()
```

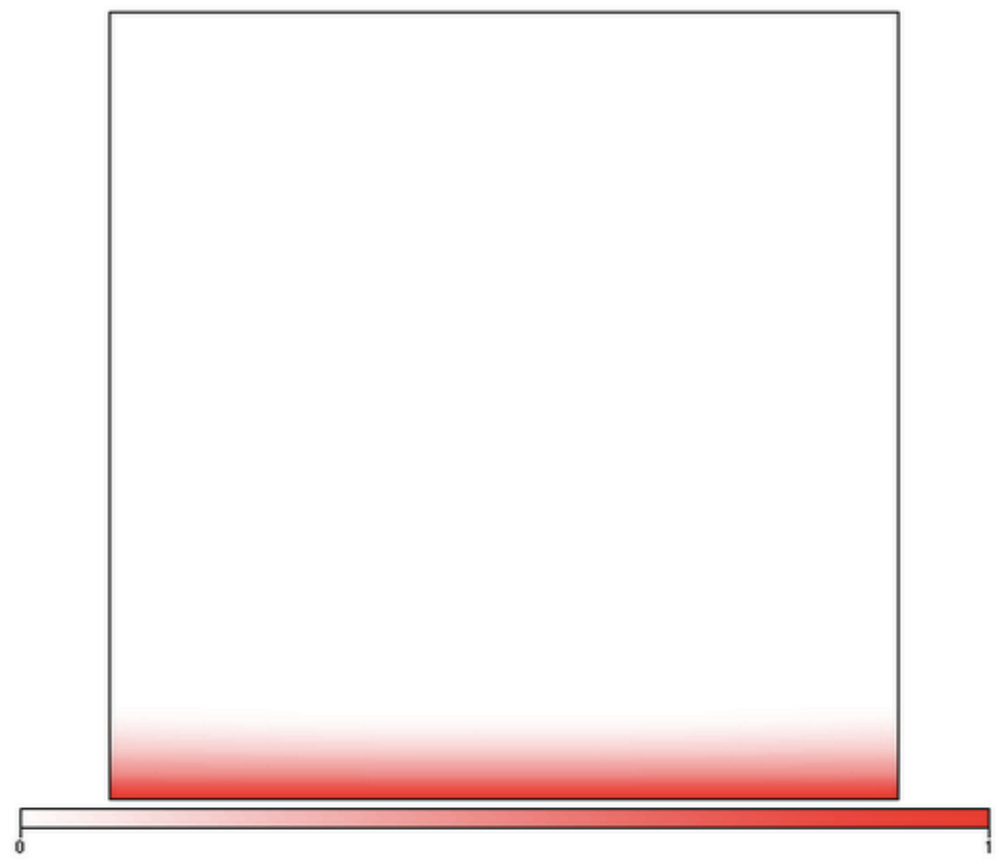
Out[9]:



In [10]:
```
# wonderous!  now lets change the conductivity function to a function of he
coord = fn.input()
steadyStateThermal.conductivityFn = fn.math.exp(10.*coord[1])
```

In [11]:
```
steadyStateThermal.solve()
fig = plt.Figure()
fig.Surface(temperatureField,mesh, colours=['white','red'])
fig.show()
```

Out[11]:



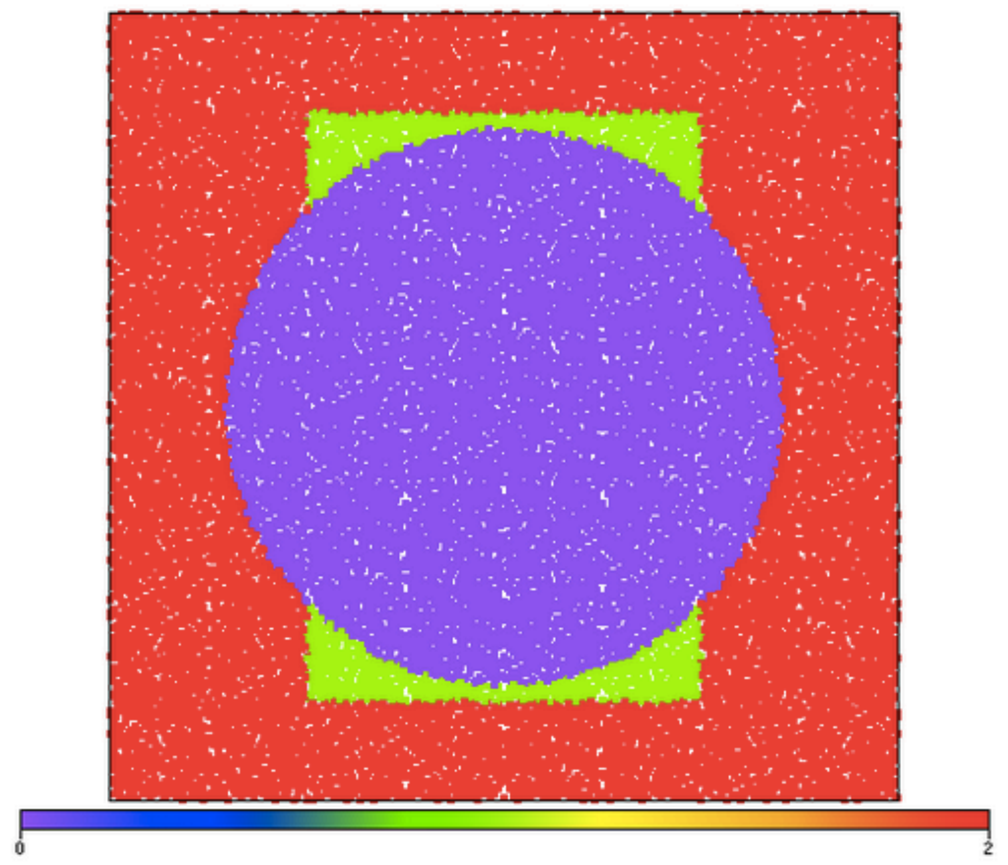
In [12]:
```
# ok, lets introduce a swarm to set some materials
swarm = uw.swarm.Swarm(mesh)
```

In [13]:
```
# lets create a variable which will track the material index
index = swarm.add_variable('char',1)
# add particles
layout = uw.swarm.layouts.GlobalSpaceFillerLayout(swarm,20)
swarm.populate_using_layout(layout)
```

In [14]:
```
# create some misc shapes
index.data[:] = fn.branching.conditional( [
            ( coord[0]*coord[0] + coord[1]*coord[1] < 0.5
            ( (fn.math.abs(coord[0]) < 0.5) & (fn.math.abs(coord[1])< 0.75)
            ( True
    ] ).evaluate(swarm)
```

In [15]:
```
fig = plt.Figure()
fig.Points(swarm,index,pointSize=4.0)
fig.show()
```

Out[15]:



In [16]:
```
# now lets map them to conductivities
materialFunction = fn.branching.map(keyFunc=index, mappingDict={0:100., 1:1.
```

In [17]:
```
steadyStateThermal.conductivityFn = materialFunction
steadyStateThermal.solve()
fig = plt.Figure()
fig.Surface(temperatureField,mesh, colours=['white','red'])
fig.show()
```

Out[17]: