In [22]:

```
import underworld as uw
import math
from underworld import function as fn
import glucifer
import numpy as np
import os
```

In [23]:

```
outputPath = os.path.join(os.path.abspath("."),"output/")

if uw.rank()==0:
    if not os.path.exists(outputPath):
        os.makedirs(outputPath)
uw.barrier()
```

In [24]:

```
xRes = 192
yRes = 48
boxLength = 4.0
boxHeight = 1.0
```

In [25]:

```
mesh = uw.mesh.FeMesh_Cartesian( elementType = ("Q1/dQ0"),
                                 elementRes  = (xRes, yRes),
                                 minCoord    = (0.,0.),
                                 maxCoord    = (boxLength, boxHeight),
                                 periodic    =  [True, False] )

velocityField  = uw.mesh.MeshVariable( mesh=mesh,         nodeDofCount=2 )
pressureField  = uw.mesh.MeshVariable( mesh=mesh.subMesh, nodeDofCount=1 )
```

In [26]:

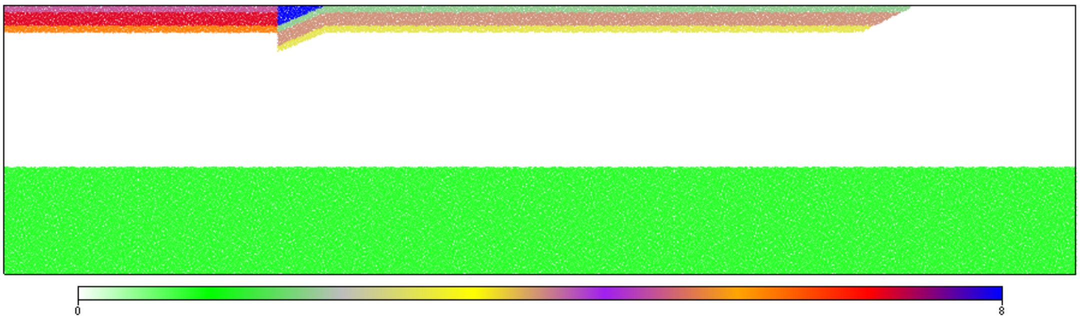
```
swarm = uw.swarm.Swarm( mesh=mesh )
materialVariable   = swarm.add_variable( dataType="int", count=1 )
swarmLayout = uw.swarm.layouts.GlobalSpaceFillerLayout( swarm=swarm, particlesPerCell=2
0 )
swarm.populate_using_layout( layout=swarmLayout )
```

In [27]:

```
# intialise the 'materialVariable' data to represent two different materials.
upperMantleIndex   = 0
lowerMantleIndex   = 1
upperSlabIndex     = 2
lowerSlabIndex     = 3
coreSlabIndex      = 4
upperOverslabIndex = 5
lowerOverslabIndex = 6
coreOverslabIndex  = 7
weakZoneIndex      = 8

# Initial material layout has a flat lying slab at 15/ degree perturation with over-rid
ing plate
lowerMantleY       = 0.4
slabLowerShape     = np.array( [ (1.2,0.925), (3.25,0.925), (3.20,0.900), (1.2,0.900),
(1.02,0.825), (1.02,0.850) ] )
slabCoreShape      = np.array( [ (1.2,0.975), (3.35,0.975), (3.25,0.925), (1.2,0.925),
(1.02,0.850), (1.02,0.900) ] )
slabUpperShape     = np.array( [ (1.2,1.000), (3.40,1.000), (3.35,0.975), (1.2,0.975),
(1.02,0.900), (1.02,0.925) ] )
overslabLowerShape = np.array([(0,0.925), (1.02,0.925), (1.02,0.899), (0,0.899)])
overslabCoreShape  = np.array([(0,0.975), (1.02,0.975), (1.02,0.925), (0,0.925)])
overslabUpperShape = np.array([(0,1.000), (1.02,1.000), (1.02,0.975), (0,0.975)])
weakZoneShape      = np.array( [ (1.02, 0.925), (1.02, 1.000), (1.2, 1.000)])

slabLower = fn.shape.Polygon( slabLowerShape )
slabUpper = fn.shape.Polygon( slabUpperShape )
slabCore  = fn.shape.Polygon( slabCoreShape  )
overslabLower = fn.shape.Polygon( overslabLowerShape )
overslabUpper = fn.shape.Polygon( overslabUpperShape )
overslabCore  = fn.shape.Polygon( overslabCoreShape  )
weakZone      = fn.shape.Polygon( weakZoneShape )

# initialise everything to be upper mantle material
materialVariable.data[:] = upperMantleIndex

# change material index if the particle is not upper mantle

for index in range ( len(swarm.particleCoordinates.data) ):
    coord = swarm.particleCoordinates.data[index][:]
    if coord[1] < lowerMantleY:
        materialVariable.data[index]     = lowerMantleIndex
    if slabCore.evaluate(tuple(coord)):
             materialVariable.data[index] =coreSlabIndex
    if slabUpper.evaluate(tuple(coord)):
             materialVariable.data[index] = upperSlabIndex
    if slabLower.evaluate(tuple(coord)):
            materialVariable.data[index]  = lowerSlabIndex
    if overslabLower.evaluate(tuple(coord)):
            materialVariable.data[index]  =  lowerOverslabIndex
    if overslabUpper.evaluate(tuple(coord)):
            materialVariable.data[index]  =  upperOverslabIndex
    if overslabCore.evaluate(tuple(coord)):
            materialVariable.data[index]  =  coreOverslabIndex
    elif weakZone.evaluate(tuple(coord)):
            materialVariable.data[index]  =  weakZoneIndex
```

In [28]:

```
store = glucifer.Store('output/overriding plate')
figParticle = glucifer.Figure( store, figsize=(960,300), name="Particles")
figParticle.append( glucifer.objects.Points(swarm, materialVariable, pointSize=2, colou
rs='white green gray yellow purple orange red blue'))
figParticle.show()
```



In [29]:

```
upperMantleViscosity =    1.0
lowerMantleViscosity =  100.0
slabViscosity        =  500.0
coreViscosity        =  500.0
weakZoneViscosity    =  400.0

# The yielding of the upper slab is dependent on the strain rate.
strainRate_2ndInvariant = fn.tensor.second_invariant(
                            fn.tensor.symmetric(
                            velocityField.fn_gradient))
cohesion = 0.06
vonMises = 0.5 * cohesion/ (strainRate_2ndInvariant+1.0e-18)

# The upper slab viscosity is the minimum of the 'slabViscosity' or the 'vonMises'
slabYieldvisc = fn.exception.SafeMaths( fn.misc.min(vonMises, slabViscosity) )

# Viscosity function for the materials
viscosityMap = { upperMantleIndex    : upperMantleViscosity,
                 lowerMantleIndex    : lowerMantleViscosity,
                 weakZoneIndex       : weakZoneViscosity,
                 upperSlabIndex      : slabYieldvisc,
                 lowerSlabIndex      : slabYieldvisc,
                 coreSlabIndex       : coreViscosity,
                 upperOverslabIndex : slabYieldvisc,
                 lowerOverslabIndex : slabYieldvisc,
                 coreOverslabIndex  : coreViscosity}
viscosityMapFn = fn.branching.map( fn_key = materialVariable, mapping = viscosityMap)
```

In [30]:

```
mantleDensity = 0.0
slabDensity   = 1.0

densityMap = { upperMantleIndex   : mantleDensity,
               lowerMantleIndex   : mantleDensity,
               upperSlabIndex     : slabDensity,
               lowerSlabIndex     : slabDensity,
               coreSlabIndex      : slabDensity,
               upperOverslabIndex : slabDensity,
               coreOverslabIndex  : slabDensity,
               lowerOverslabIndex : slabDensity,
               weakZoneIndex      : slabDensity}
densityFn  = fn.branching.map( fn_key = materialVariable, mapping = densityMap )

# Define our vertical unit vector using a phython tuple
z_hat = ( 0.0, 1.0 )

# now create a buoyancy force vector
buoyancyFn = -1.0 * densityFn * z_hat
```

In [31]:

```
# Set initial conditions
velocityField.data[:] = [0.,0.]
pressureField.data[:] = 0.

# send boundary condition information to underworld
iWalls = mesh.specialSets["MinJ_VertexSet"] + mesh.specialSets["MaxI_VertexSet"]
jWalls = mesh.specialSets["MinJ_VertexSet"] + mesh.specialSets["MaxJ_VertexSet"]
bottomWall = mesh.specialSets["MinJ_VertexSet"]

periodicBC = uw.conditions.DirichletCondition( variable         = velocityField,
                                               indexSetsPerDof  = ( bottomWall, jWalls)
)
```

In [32]:

```
# initial linear slab viscosity setup
stokes = uw.systems.Stokes(    velocityField = velocityField,
                               pressureField = pressureField,
                               voronoi_swarm = swarm,
                               conditions    = periodicBC,
                               fn_viscosity  = viscosityMapFn,
                               fn_bodyforce  = buoyancyFn )
# create solver and solve
solver = uw.systems.Solver(stokes)
```

In [33]:

```
# use "lu" direct solve if running in serial
if(uw.nProcs()==1):
    solver.set_inner_method("lu")
```

In [34]:

```
advector = uw.systems.SwarmAdvector( swarm=swarm, velocityField=velocityField, order=2
)
```

In [35]:

```
# the root mean square Velocity
velSquared = uw.utils.Integral( fn.math.dot(velocityField,velocityField), mesh )
area = uw.utils.Integral( 1.,mesh )
Vrms = math.sqrt( velSquared.evaluate()[0]/area.evaluate()[0] )
```

In [36]:

```
#Plot of Velocity Magnitude
figVelocityMag = glucifer.Figure(store, figsize=(960,300))
figVelocityMag.append( glucifer.objects.Surface(mesh, fn.math.sqrt(fn.math.dot(velocity
Field,velocityField))) )

#Plot of Strain Rate, 2nd Invariant
figStrainRate = glucifer.Figure(store, figsize=(960,300))
figStrainRate.append( glucifer.objects.Surface(mesh, strainRate_2ndInvariant, logScale=
True) )

#Plot of particles viscosity*
figViscosity = glucifer.Figure(store, figsize=(960,300))
figViscosity.append( glucifer.objects.Points(swarm, viscosityMapFn, pointSize=2) )

#Plot of particles stress invariant
figStress = glucifer.Figure( store, figsize=(960,300) )
figStress.append( glucifer.objects.Points(swarm, 2.0*viscosityMapFn*strainRate_2ndInvar
iant, pointSize=2, logScale=True) )
```

In [37]:

```
time = 0. # Initial time
step = 0  # Initial timestep
maxSteps = 30 # Maximum timesteps
steps_output = 10  # output every 10 timesteps
```

In [38]:

```
# define an update function
def update():
    # Retrieve the maximum possible timestep for the advection system.
    dt = advector.get_max_dt()
    # Advect using this timestep size
    advector.integrate(dt)
    return time+dt, step+1
```

In [39]:

```
while step < maxSteps:
    # Solve non linear Stokes system
    solver.solve(nonLinearIterate=True)
    # output figure to file at intervals = steps_output
    if step % steps_output == 0 or step == maxSteps-1:
        #Important to set the timestep for the store object here or will overwrite prev
ious step
        store.step = step
        figParticle.save(     outputPath + "particle"    + str(step).zfill(4) )
        figVelocityMag.save(  outputPath + "velocityMag" + str(step).zfill(4) )
        figStrainRate.save(   outputPath + "strainRate"  + str(step).zfill(4) )
        figViscosity.save(    outputPath + "viscosity"   + str(step).zfill(4) )
        figStress.save(       outputPath + "stress"      + str(step).zfill(4) )

        Vrms = math.sqrt( velSquared.evaluate()[0]/area.evaluate()[0] )
        print 'step = {0:6d}; time = [1:.3e]; Vrms = {2:.3e}'.format(step,time,Vrms)

    # update
    time,step = update()
```

```
step =      0; time = [1:.3e]; Vrms = 1.097e-05
step =     10; time = [1:.3e]; Vrms = 1.152e-05
step =     20; time = [1:.3e]; Vrms = 1.165e-05
step =     29; time = [1:.3e]; Vrms = 1.175e-05
```

In [40]:

```
import glucifer
saved = glucifer.Viewer('output/overriding plate')
```

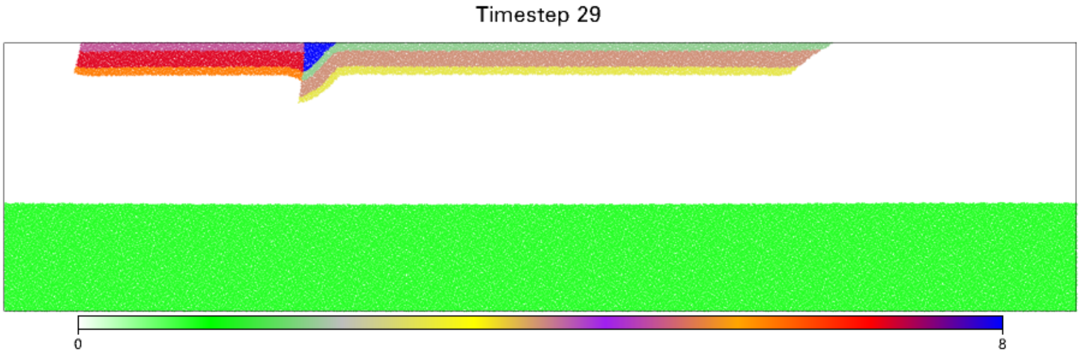
In [41]:

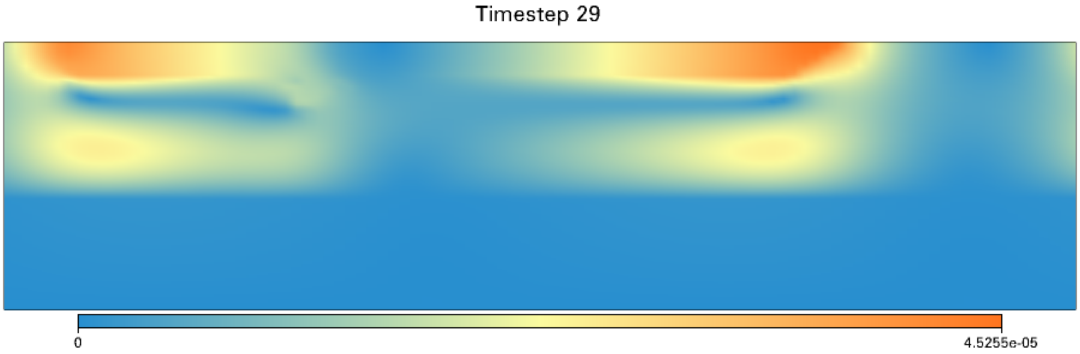
```
figs = saved.figures
steps= saved.steps
print("Saved database '%s'" % (saved.filename))
print(" - %d figures : %s"  % (len(figs), str(figs.keys())))
print(" - %d timesteps (final = %d) : %s" % (len(steps), steps[-1], steps))
```

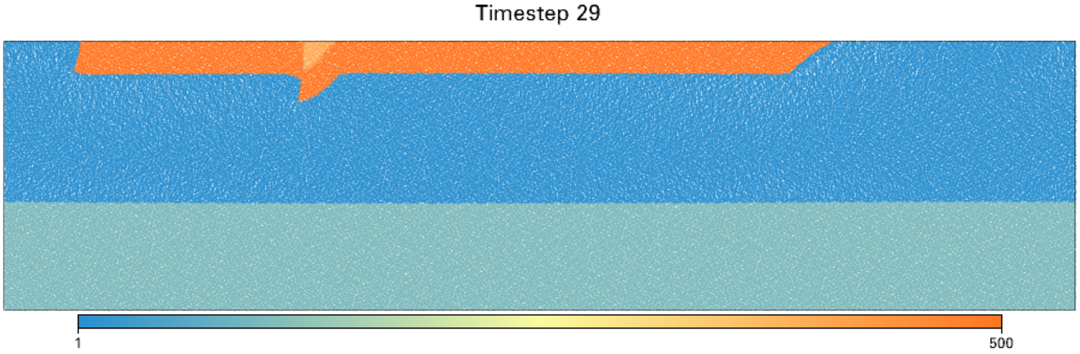
```
Saved database 'output/overriding plate.gldb'
 - 5 figures : ['Particles', 'Figure_5', 'Figure_7', 'Figure_6', 'Figure_
8']
 - 4 timesteps (final = 29) : [0, 10, 20, 29]
```

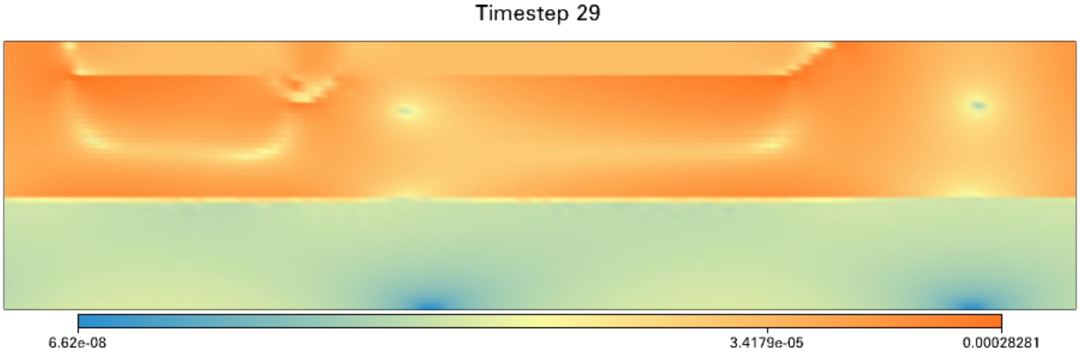
In [42]:

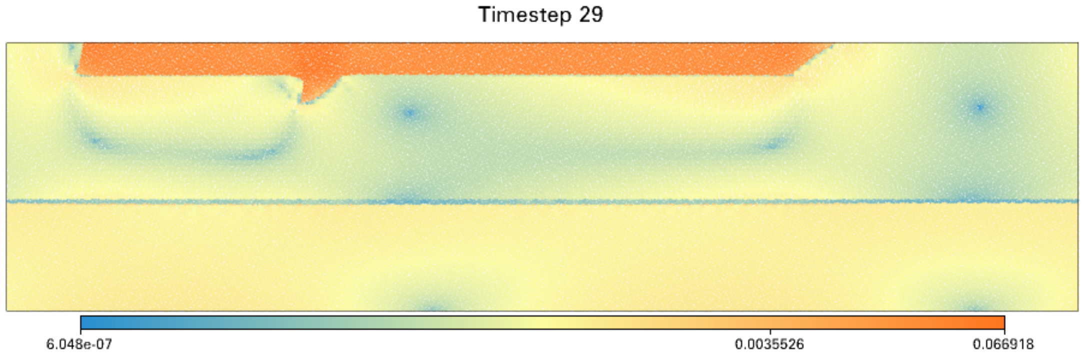
```
#Re-visualise the final timestep
saved.step = steps[-1]
for name in saved.figures:
    saved.figure(name)
    saved["quality"] = 2
    saved["title"] = "Timestep ##"
    saved.show()
```


Timestep 29
0
8


Timestep 29
0
4.5255e-05


Timestep 29
1
500


Timestep 29
6.62e-08
3.4179e-05
0.00028281


Timestep 29
6.048e-07
0.0035526
0.066918