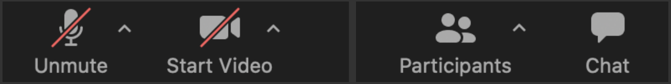
Workflows with Programmable Filter & Source in ParaView

Alex Razoumov
alex.razoumov@westgrid.ca

Zoom controls

- Please mute your microphone and camera unless you have a question
- To ask questions at any time, type in Chat, or Unmute to ask via audio
- Raise your hand in Participants



- This talk is being recorded
  - your name might appear in the recording (if you want, you can change it)
  - video will be posted at `https://westgrid.github.io/trainingMaterials` under one of the top-menu topics
- Email training@westgrid.ca

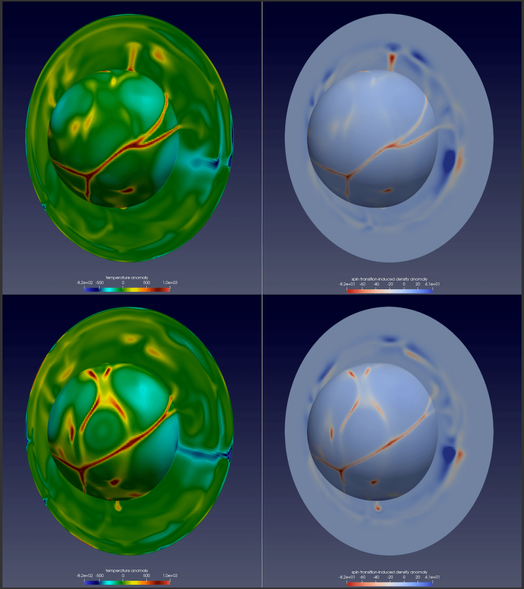
Upcoming winter/spring webinars

- 10 confirmed webinars
- `https://bit.ly/wg2021a` for up-to-date schedule and registration

2021 IEEE Vis Contest

https://scivis2021.netlify.app

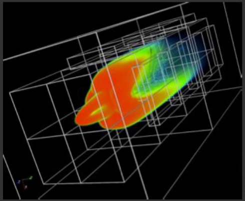
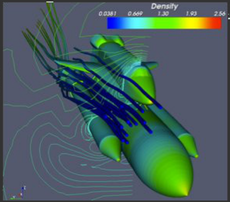
- Co-hosting 2021 SciVis Contest with IEEE Vis
- Dataset: 3D simulation of Earth's mantle convection covering 500 Myrs of geological time
- Contest is open to anyone (no research affiliation necessary), dataset available now
- Wanted: pretty pictures + problem-specific analysis of descending / rising flows
- July 31, 2021 - deadline for Contest entry submissions

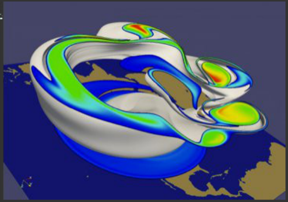

temperature anomaly
-8.2e+02 -500 0 500 1.0e+03
spin transition-induced density anomaly
-8.2e+01 -60 -40 -20 0 20 4.1e+01
temperature anomaly
-8.2e+02 -500 0 500 1.0e+03
spin transition-induced density anomaly
-8.2e+01 -60 -40 -20 0 20 4.1e+01

ParaView

http://www.paraview.org and https://github.com/Kitware/ParaView

- Started in 2000 as a collaboration between Los Alamos National Lab and Kitware Inc., later joined by Sandia National Labs and other partners; first public release in 2002
- Available as source and pre-compiled binary for Linux/Mac/Windows
- To visualize extremely large datasets on distributed memory machines
- Both interactive and Python scripting
- Client-server for remote interactive visualization
- Uses MPI for distributed-memory parallelism on HPC clusters
- ParaView is based on VTK (Visualization Toolkit)
  - not the only VTK-based open-source scientific renderer, e.g. VisIt, MayaVi (Python + numpy + scipy + VTK), an of course a number of Kitware's own tools besides ParaView are based on VTK
  - VTK can be used from C++, Python, and now JavaScript as a standalone renderer



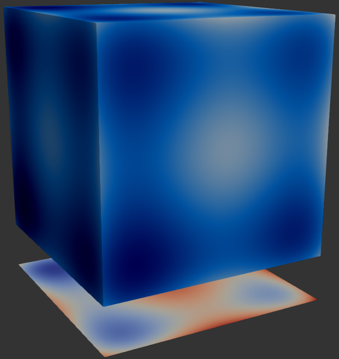
Advanced ParaView topics

- VTK programming (overlap with other topics)
- ✓ Web-based 3D scientific visualization: ParaViewWeb, VTK.js, ParaView Glance, ParaView Lite, Visualizer
- ParaView Cinema
- Catalyst in-situ visualization library
- AMR (multi-resolution) & multi-block datasets
- ✓ CPU-based ray tracing / photorealistic rendering with OSPRay
- ✎ Ray tracing on CUDA-supported cards with Nvidia OptiX
- ✎ Scalable 3D volumetric visualization on GPU clusters with NVIDIA IndeX
- ✓ Remote / parallel interactive visualization
- ✓ Batch visualization
- ✓ Advanced scripting
- Programmable Filter & Source in ParaView
- Writing ParaView plugins
- ✓ Advanced animation
- Using ParaView with special hardware: HMDs, stereo projectors, Looking Glass

Past webinar recordings at `http://bit.ly/vispages`

Modifying VTK objects

Let's say we want to plot a projection of a cubic dataset along one of its principal axes, or do some other transformation for which there is no filter



- Calculator / Python Calculator filter cannot modify the geometry . . .

Programmable Filter

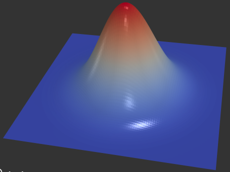
1. Apply **Programmable Filter** to a ParaView pipeline object
2. Select OutputDataSetType (either Same as Input, or one of provided VTK data types)
3. In the Script box, write Python code: input from a pipeline object → output
   - use `inputs[0].Points[:,0:3]` and/or `inputs[0].PointData['array name']` to compute your output: points, cells, and data arrays
   - some objects pass multiple `inputs[:]`
   - either use the default output for your selected OutputDataSetType, or create your own custom output object
4. Depending on output's type, might need to describe it in the RequestInformation Script box
5. Hit Apply

Programmable Source

Same as Programmable Filter but without an input

1. build an object programmatically
2. read and process data from a file

Let's create a 2D Gaussian $f(\vec{r}) = e^{-\frac{|\vec{r}-\vec{r}_0|^2}{2\sigma^2}}$, $\vec{r} \in \mathbb{R}^2$ centered at (0,0)



1. Apply Sources | **Plane** at 100^2 resolution
2. Apply **Python Calculator** with the Gaussian for Point Data

```
p = exp(-(inputs[0].Points[:,0]**2+inputs[0].Points[:,1]**2)/(2*0.02))
```

3. Apply **Programmable Filter**
   - set Output Data Set Type = Same as Input
   - this will create the same discretization (Points) as its input, without the fields (PointData)
4. Let's print some info

```
print(type(output))    # paraview.vtk.numpy_interface.dataset_adapter.PolyData
print(dir(output))
print(output.Points.shape)    # actually a 3D dataset
print(output.Points)          # but all z-coordinates are 0s
```

5. Paste into Script

```
output.Points[:,2] = 0.5*inputs[0].PointData["p"]    # set the output's z-axis
dataArray = 0.3 + inputs[0].PointData["p"]
output.PointData.append(dataArray, "pnew")       # add the new data array to our output
```

6. Display in 3D, colour with `pnew`

- A scalar field is stored as a flat, 1D array over 3D points

```
print(output.PointData["pnew"].shape)
```

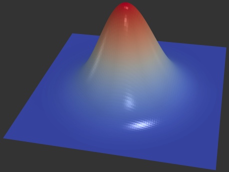
- Multiple ways to access data, e.g. these two lines point to the same array

```
print(output.PointData["pnew"])
print(output.GetPointData().GetArray("pnew"))
```

- The same for points

```
print(output.GetNumberOfPoints())
print(output.GetPoint(1005))
print(output.Points[1005,:])
```

Same workflow without the Python Calculator



Let's simplify our script

1. Apply Sources | **Plane** at 100^2 resolution
2. Apply **Programmable Filter**
   - set Output Data Set Type = Same as Input
   - this will create the same discretization (Points) as its input, without the fields (PointData)
3. Paste into Script

```
output.Points[:,2] = 0.5*exp(-(inputs[0].Points[:,0]**2+
                               inputs[0].Points[:,1]**2)/(2*0.02))
dataArray = 0.3 + 2*output.Points[:,2]
output.PointData.append(dataArray, "pnew")    # add the new data array to our output
```

4. Display in 3D, colour with `pnew`

Programmable Source

1. Let's switch to Programmable Source
2. Try different OutputDataSetType (no Same as Input)
3. `print(type(output))` via Script

Read CSV file into a table

1. Apply **Programmable Source**
   - set Output Data Set Type = vtkTable
2. Paste into Script code from `readCSV2table.py`

```
import numpy as np
data = np.genfromtxt("/Users/razoumov/Documents/01-webinar/tabulatedGrid.txt",
                     dtype=None, names=True, delimiter=',', autostrip=True)
for name in data.dtype.names:    # go through all 4 columns
   array = data[name]            # this is a numpy array
   output.RowData.append(array, name)
```

3. Pass tabular data through **Table To Structured Grid**
   - set data extent in each dimension
   - specify `x/y/z` columns
   - colour with `scalar`

Read CSV file directly to Cartesian mesh

1. Apply **Programmable Source**
   - set Output Data Set Type = vtkImageData
2. Paste into Script 1st code from `readCSV2image.py`

```
import numpy as np
data = np.genfromtxt("/Users/razoumov/Documents/01-webinar/tabulatedGrid.txt",
                     dtype=None, names=True, delimiter=',', autostrip=True)

nx = round(data['x'].shape[0]**(1./3.))
ny = round(data['y'].shape[0]**(1./3.))
nz = round(data['z'].shape[0]**(1./3.))

output.SetDimensions(nx,ny,nz)
output.SetOrigin(0,0,0)
output.SetSpacing(.1,.1,.1)
output.SetExtent(0,nx-1,0,ny-1,0,nz-1)
output.AllocateScalars(vtk.VTK_FLOAT,1)

vtk_data_array = vtk.util.numpy_support.numpy_to_vtk(num_array=data['scalar'].ravel(),
                                                     deep=True, array_type=vtk.VTK_FLOAT)
vtk_data_array.SetNumberOfComponents(1)
vtk_data_array.SetName("density")
output.GetPointData().SetScalars(vtk_data_array)
```

Read CSV file directly to Cartesian mesh (cont.)

3. The Image (Uniform Rectilinear Grid) array is properly created (check the Information tab!), but nothing shows up ...

Read CSV file directly to Cartesian mesh (cont.)

3. The Image (Uniform Rectilinear Grid) array is properly created (check the Information tab!), but nothing shows up ...

   Need to tell the ParaView pipeline about the dimensionality of our vtkImageData!

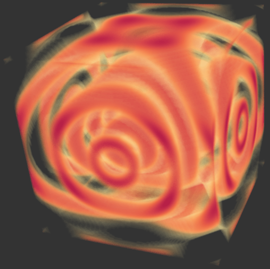
4. Paste into Script (Request Information) 2nd code from `readCSV2image.py`

```
from paraview import util
nx, ny, nz = 10, 10, 10
util.SetOutputWholeExtent(self, [0,nx-1,0,ny-1,0,nz-1])
```

Write a 3D function directly into Cartesian mesh: point by point

1. Apply **Programmable Source**, set Output Data Set Type = vtkImageData
2. Paste 1st code from `volumeImage.py`

```
from numpy import linspace, sin, sqrt

n = 100
output.SetDimensions(n,n,n)
output.SetOrigin(0,0,0)
output.SetSpacing(.1,.1,.1)
output.SetExtent(0,n-1,0,n-1,0,n-1)
output.AllocateScalars(vtk.VTK_FLOAT,1)

x = linspace(-7.5,7.5,n)
y, z = x.reshape(n,1), x.reshape(n,1,1)
data = ((sin(sqrt(y*y+x*x)))**2-0.5)/(0.001*(y*y+x*x)+1.)**2 + \
    ((sin(sqrt(z*z+y*y)))**2-0.5)/(0.001*(z*z+y*y)+1.)**2 + 1.

rho = vtk.vtkFloatArray()
rho.SetName("density")
rho.SetNumberOfComponents(1)
rho.SetNumberOfTuples(n**3)
point = 0
for i in range(n):
    for j in range(n):
        for k in range(n):
            rho.SetValue(point,data[i][k][j])
            point += 1

output.GetPointData().AddArray(rho)
```



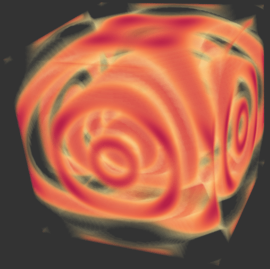
Script (Request Information):

```
from paraview import util
n = 100
util.SetOutputWholeExtent(self,
            [0,n-1,0,n-1,0,n-1])
```

Write a 3D function directly into Cartesian mesh: numpy → VTK

1. Apply **Programmable Source**, set Output Data Set Type = vtkImageData
2. Paste 2nd code from `volumeImage.py`

```
from numpy import linspace, sin, sqrt

n = 100
output.SetDimensions(n,n,n)
output.SetOrigin(0,0,0)
output.SetSpacing(.1,.1,.1)
output.SetExtent(0,n-1,0,n-1,0,n-1)
output.AllocateScalars(vtk.VTK_FLOAT,1)

x = linspace(-7.5,7.5,n)
y, z = x.reshape(n,1), x.reshape(n,1,1)
data = ((sin(sqrt(y*y+x*x)))**2-0.5)/(0.001*(y*y+x*x)+1.)**2 + \
    ((sin(sqrt(z*z+y*y)))**2-0.5)/(0.001*(z*z+y*y)+1.)**2 + 1.

vtk_data_array = vtk.util.numpy_support.numpy_to_vtk(
    num_array=data.ravel(), deep=True, array_type=vtk.VTK_FLOAT)
vtk_data_array.SetNumberOfComponents(1)
vtk_data_array.SetName("density")
output.GetPointData().SetScalars(vtk_data_array)
```

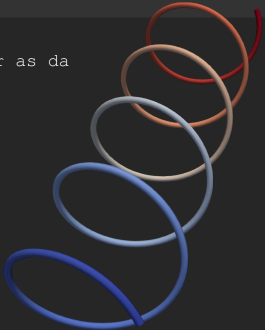


Script (Request Information):

```
from paraview import util
n = 100
util.SetOutputWholeExtent(self,
            [0,n-1,0,n-1,0,n-1])
```

Single helix source example from ParaView docs

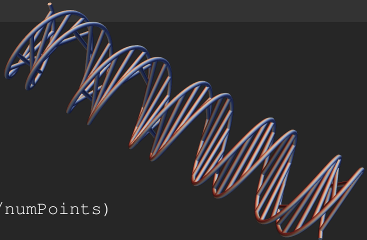
1. Apply **Programmable Source**, set Output Data Set Type = vtkPolyData
2. Paste code from `singleHelix.py`



```
import numpy as np
import vtk.numpy_interface.algorithms as alg, vtk.numpy_interface.dataset_adapter as da

numPoints, length, rounds = 300, 8.0, 5

index = np.arange(0, numPoints, dtype=np.int32)    # 0, ..., numPoints-1
phi = rounds * 2 * np.pi * index / numPoints
x, y, z = index * length / numPoints, np.sin(phi), np.cos(phi)
coordinates = alg.make_vector(x, y, z)    # numpy array (numPoints,3)

output.Points = coordinates                # set point coordinates
output.PointData.append(phi, 'angle')     # append a scalar field on points

pointIds = vtk.vtkIdList()
pointIds.SetNumberOfIds(numPoints)
for i in range(numPoints):    # define a single polyline connecting all the points in order
   pointIds.SetId(i, i)       # point i in the line is formed from point i in vtkPoints

output.Allocate(1, 1)      # allocate space for one vtkPolyLine 'cell' to the vtkPolyData object
output.InsertNextCell(vtk.VTK_POLY_LINE, pointIds)    # add this 'cell' to the vtkPolyData object
```

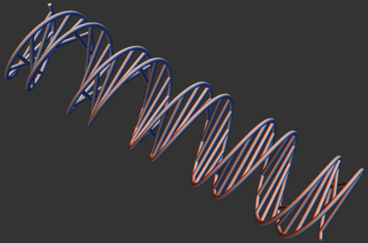
Double helix source example from ParaView docs

1. Apply **Programmable Source**, set Output Data Set Type = vtkPolyData
2. Paste code from `doubleHelix.py`



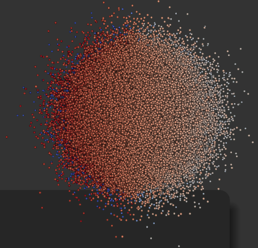
```
import numpy as np

numPoints, length, rounds, phaseShift = 300, 8.0, 5, np.pi/1.5

points = vtk.vtkPoints()    # this will store the points for the Helix
for i in range(0, numPoints):
   x = i * length / numPoints
   y, z = np.sin(i*rounds*2*np.pi/numPoints), np.cos(i*rounds*2*np.pi/numPoints)
   points.InsertPoint(i,x,y,z)                 # 1st helix
   y, z = np.sin(i*rounds*2*np.pi/numPoints + phaseShift), np.cos(i*rounds*2*np.pi/numPoints + phaseShift)
   points.InsertPoint(i+numPoints,x,y,z)    # 2nd helix

output.SetPoints(points)          # add these points to vtkPolyData

helix1 = vtk.vtkPolyLine()    # cell forming the 1st helix
helix2 = vtk.vtkPolyLine()    # cell forming the 2nd helix

helix1.GetPointIds().SetNumberOfIds(numPoints)
helix2.GetPointIds().SetNumberOfIds(numPoints)
for i in range(0,numPoints):
   helix1.GetPointIds().SetId(i, i)               # 1st helix: point i from point i in vtkPoints
   helix2.GetPointIds().SetId(i, i+numPoints)    # 2nd helix: point i from point i+numPoints in vtkPoints
```

Double helix source example from ParaView docs (cont.)



```
links = range(2,numPoints,5)         # a link at every 5th helix point pair starting from the 3rd pair
output.Allocate(2+len(links), 1)     # allocate space for 2 helices and 60 links: each is a single cell

output.InsertNextCell(helix1.GetCellType(), helix1.GetPointIds())    # add 1st helix 'cell'
output.InsertNextCell(helix2.GetCellType(), helix2.GetPointIds())    # add 2nd helix 'cell'

for i in links:
   link = vtk.vtkLine()                                 # add a line
   link.GetPointIds().SetId(0, i)                          # connecting point i (1st helix)
   link.GetPointIds().SetId(1, i+numPoints)                # and point i+numPoints (2nd helix)
   output.InsertNextCell(link.GetCellType(), link.GetPointIds())    # add the link 'cell'
```

Programmatically generated point cloud

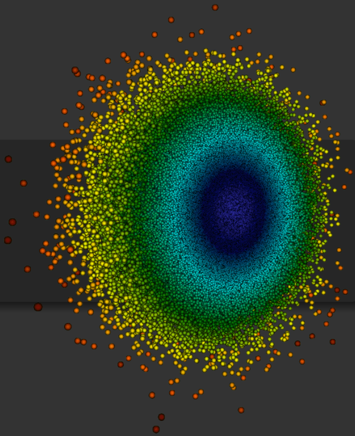


1. Apply **Programmable Source**, set Output Data Set Type = vtkPolyData
2. Paste code from `pointCloud.py`

```
from numpy import abs, random, sin, cos, pi, arcsin

numPoints = int(1e6)
r = abs(random.randn(numPoints))                # 1D array drawn from a normal (Gaussian) distribution
theta = arcsin(2.*random.rand(numPoints)-1.)    # use 1D array drawn from a uniform [0,1) distribution
phi = 2.*pi*random.rand(numPoints)              # use 1D array drawn from a uniform [0,1) distribution

x = r*cos(theta)*sin(phi)
y = r*cos(theta)*cos(phi)
z = r*sin(theta)
coordinates = vtk.numpy_interface.algorithms.make_vector(x, y, z)    # numpy array (numPoints,3)

output.Points = coordinates                  # set point coordinates
output.PointData.append(r, "r")              # append a scalar field on points
output.PointData.append(phi, "phi")          # append another scalar field on points
```

3. Points are not visible in ParaView $\Rightarrow$ either (a) switch to Point Gaussian representation, or (b) in Script's output create a single cell without connections (next slide)

Programmatically generated point cloud



Create a single cell without connections:

```
pointIds = vtk.vtkIdList()
pointIds.SetNumberOfIds(numPoints)
for p in range(numPoints):
    pointIds.SetId(p, p)
output.Allocate(1)       # allocate space for a single cell
output.InsertNextCell(vtk.VTK_POLY_VERTEX, pointIds)
```

4. Colour by radius
5. Apply **Clip**
6. Optionally switch back to Point Gaussian representation

The following two pieces of code produce the same result:

```
output.Points = coordinates
```

```
points = vtk.vtkPoints()
points.SetData(vtk.numpy_interface.dataset_adapter.
                numpyTovtkDataArray(coordinates, 'Points')) # set coordinates
output.SetPoints(points)     # add these points to vtkPolyData
```

Projection to a plane: point by point

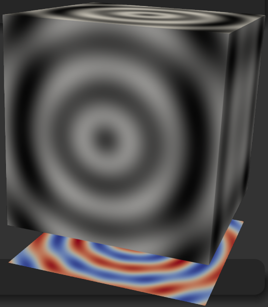
1. To a 3D dataset, apply **Programmable Filter**, set Output Data Set Type = vtkUnstructuredGrid
2. Paste code from `projectionUnstructured.py`

```
numPoints = inputs[0].GetNumberOfPoints()    # also inputs[0].GetNumberOfCells()
side = round(numPoints**(1./3.))
layer = side*side
rho = inputs[0].PointData['density']    # 1D flat array; also inputs[0].GetPointData().GetArray('density')

points = vtk.vtkPoints()        # create vtkPoints instance, to contain 100^2 points in the projection
proj = vtk.vtkDoubleArray()     # create the projection array
proj.SetName('projection')
for i in range(layer):          # loop through 100x100 points
    x, y = inputs[0].GetPoint(i)[0:2]
    z, column = -20., 0.
    for j in range(side):
        column += rho.GetValue(i+layer*j)
    points.InsertNextPoint(x,y,z)          # also points.InsertPoint(i,x,y,z)
    proj.InsertNextValue(column)           # add value to this point

output.SetPoints(points)                   # add points to vtkUnstructuredGrid
output.GetPointData().SetScalars(proj)     # add projection array to these points
```

Projection to a plane: point by point (cont.)

3. Add cells to our vtkUnstructuredGrid

```
quad = vtk.vtkQuad()                   # create a cell
output.Allocate(side, side)            # allocate space for side^2 'cells'
for i in range(side-1):
    for j in range(side-1):
        quad.GetPointIds().SetId(0,i+j*side)
        quad.GetPointIds().SetId(1,(i+1)+j*side)
        quad.GetPointIds().SetId(2,(i+1)+(j+1)*side)
        quad.GetPointIds().SetId(3,i+(j+1)*side)
        output.InsertNextCell(vtk.VTK_QUAD, quad.GetPointIds())
```



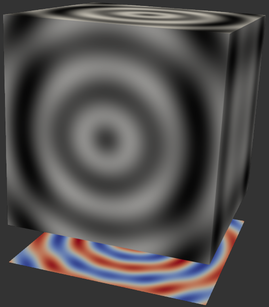
Check out an alternative implementation (same results) in `projectionUnstructured2.py` with:

```
output = self.GetOutput()    # different output type => different code
```

Projection to a plane: numpy → VTK

1. To a 3D dataset, apply **Programmable Filter**, set Output Data Set Type = vtkImageData
2. Paste code from `projectionImage.py`

```
numPoints = inputs[0].GetNumberOfPoints()
side = round(numPoints**(1./3.))
layer = side*side
rho = inputs[0].PointData['density']
           # also inputs[0].GetPointData().GetArray('density')

output.SetOrigin(inputs[0].GetPoint(0)[0],
                 inputs[0].GetPoint(0)[1], -20.)
output.SetSpacing(1.0, 1.0, 1.0)
output.SetDimensions(side, side, 1)
output.SetExtent(0,99,0,99,0,1)
output.AllocateScalars(vtk.VTK_FLOAT, 1)

rho3D = rho.reshape(side, side, side)
vtk_data_array = vtk.util.numpy_support.
          numpy_to_vtk(rho3D.sum(axis=2).ravel(),
                       deep=True, array_type=vtk.VTK_FLOAT)
vtk_data_array.SetNumberOfComponents(1)
vtk_data_array.SetName("projection")
output.GetPointData().SetScalars(vtk_data_array)
```



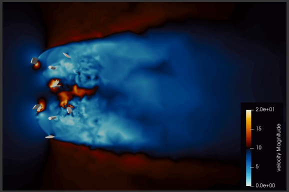
Script (Request Information):

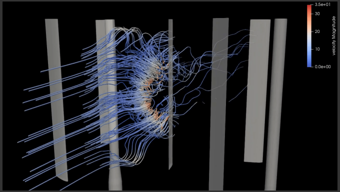
```
from paraview import util
n = 100
util.SetOutputWholeExtent(self,
               [0,n-1,0,n-1,0,0])
```

Saving workflow

- Save as a ParaView state file $\Rightarrow$ Python code with appear inside an XML element
- Save as a Python state file $\Rightarrow$ Python code with appear inside `programmableFilter1.Script` variable

Toy animations





2017 Visualize This contest: Jarno van der Kolk, UofOttawa

1. Toy 3D conceptual animation of rotating blades done entirely in ParaView
2. Variation of cross-section along the vertical direction + nice colour scheme for showing the wind speed
3. Velocity streamlines with colour showing the air speed
4. Q-criterion isosurfaces for vorticity
5. Pressure field on the blades

On presenter's laptop:

```
paraview -state=/Users/razoumov/Documents/01-webinar/jarno/landscape.pvsm
```

- Workflow suggestions
  - use `print`
  - ParaView *will crash* when you don't allocate objects properly inside Programmable Filter/Source
- Links
  - `https://www.paraview.org/Wiki/Python_Programmable_Filter`
  - `https://www.paraview.org/Wiki/ParaView/Simple_ParaView_3_Python_Filters`
  - `https://docs.paraview.org/en/latest/ReferenceManual/pythonProgrammableFilter.html`
  - `https://pyscience.wordpress.com/2014/09/06/numpy-to-vtk-converting-your-numpy-arrays-to-vtk-arrays-and-files`
  - first half of the talk by Jean M. Favre (Swiss National Supercomputing Centre) `https://youtu.be/aPKVrrzYGdo`
- Future topic: converting Programmable Filter's code to a plugin
  - code its own custom GUI Properties using Python Decorators
  - load your plugin, then use it as a *normal* Filter/Source

Questions?