ParaViewによる可視化中級編:
スクリプティングを中心に

担当: 大嶋拓也 (新潟大学)

オープンCAEシンポジウム2011
2011年12月1日(木)

Agenda

本講習の目的

- ParaViewにおけるPythonスクリプティングの概要を学ぶ

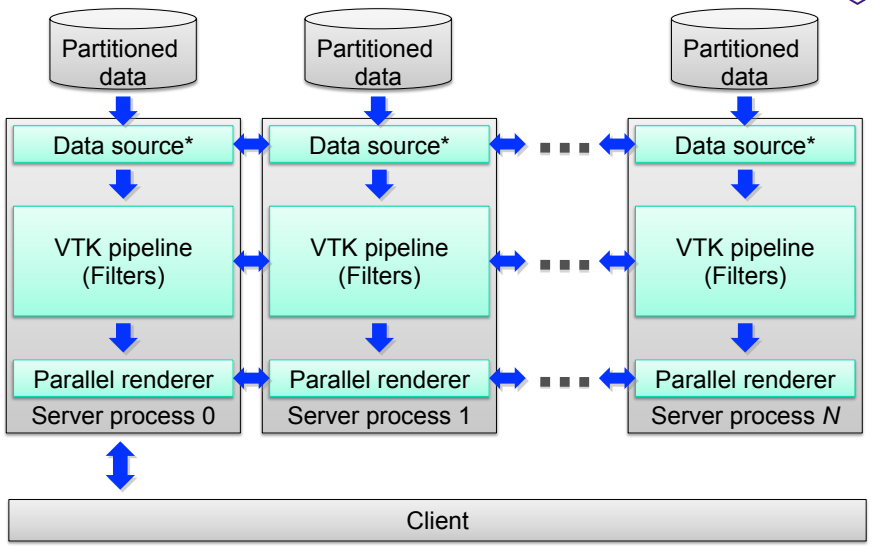
Topics

- 序: ParaViewのクライアント・サーバモデル
  - 当該モデルにおけるPython適用の整理
    - プログラミング・インターフェイス: 3種類
    - クライアント・サーバ形態: 4形態
    - Pythonの適用可能箇所: 9箇所
- 本編: 9箇所それぞれのPythonスクリプティングを概説

序: ParaViewのクライアント・サーバモデル

- 当該モデルにおけるPython適用の整理

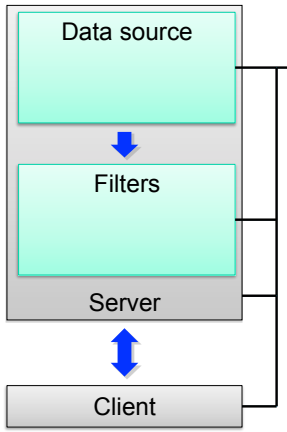
ParaViewのクライアント・サーバ・モデル



*Data Source: データファイルリーダ、またはFilters→Sourcesの各項目

ParaViewにおけるPythonの適用(1): 概要

- クライアント・サーバ・モデルにおけるPythonの適用形態は多様
  - 典型的には4ケース(次々スライド以降)



- それぞれにおいてPythonの適用が可能
- 3種のプログラミング・インターフェイス
  - Python Calculator
  - VTK Python
  - ParaView Python

- 目的に応じて、どの部分でPythonスクリプティングを使用すべきか見極める

ParaViewにおけるPythonの適用(2): インターフェイス

- Python Calculator ：パイプライン内でのデータ演算に使用、下記VTK Pythonの簡略化記法。サーバ側で実行される

- VTK Python ：VTKの数百のクラスをPythonでラッピングしたもの。パイプライン内でのデータ演算、生成に使用。主にサーバ側で実行される(例外: `servermanager.Fetch()`)

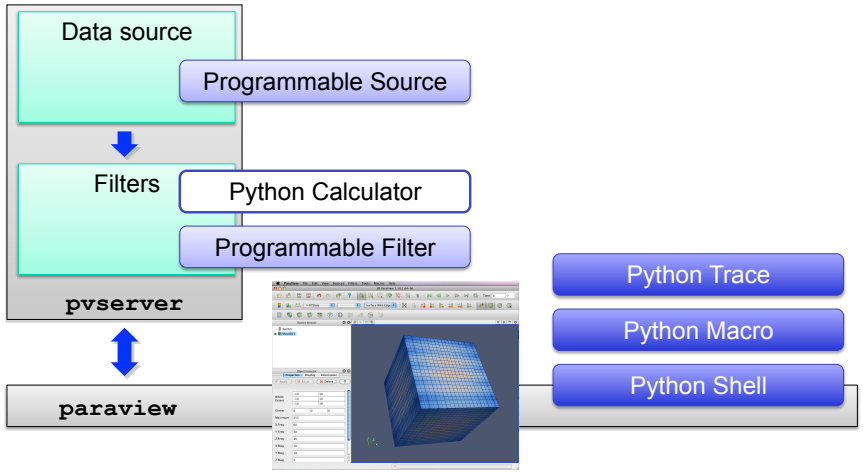
```
import vtk
```

- ParaView Python ：ParaView全体の実行制御に使用。主にクライアント側で実行される(例外: **`pvbatch`**, CoProcessing)

```
import paraview
```

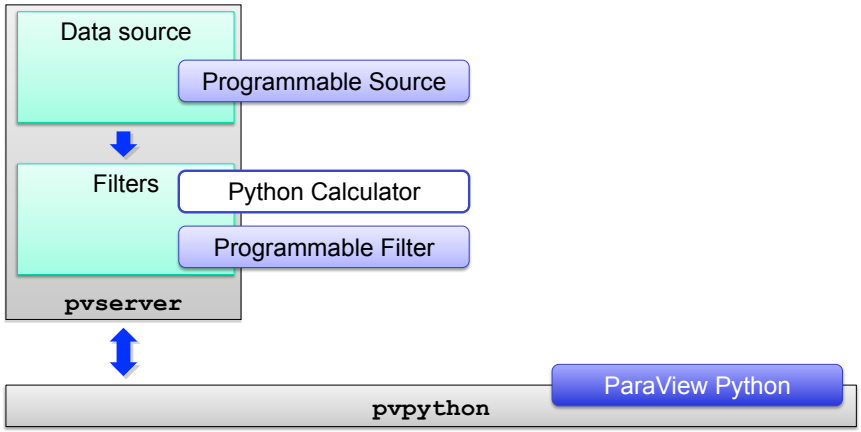

ParaViewにおけるPythonの適用(3): **`paraview`**

- GUIクライアントとParaViewサーバによる、通常の利用形態



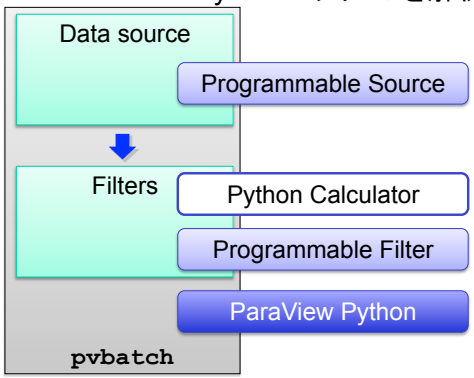
ParaViewにおけるPythonの適用(4): **`pvpython`**

- **`pvpython`**: ParaView Python readyなPythonインタプリタ
- **`pvpython`**はクライアントとして動作



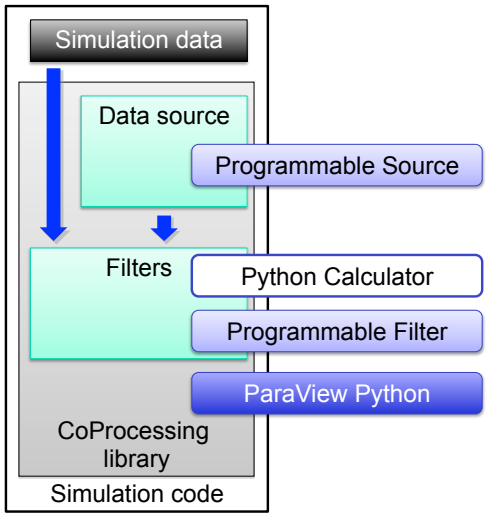
ParaViewにおけるPythonの適用(5): pvbatch

- **pvbatch**: ParaView Python readyなPythonインタプリタ
- **pvbatch**はサーバとして動作、クライアント無し
- 並列実行の場合は、**pvbatch**を`mpirun`で並列実行させる
- プロセス0がPythonスクリプトを解釈、1以降は0の指示により動作



ParaViewにおけるPythonの適用(6): CoProcessing

- シミュレーション・コードに、ParaViewをライブラリとして組み込む

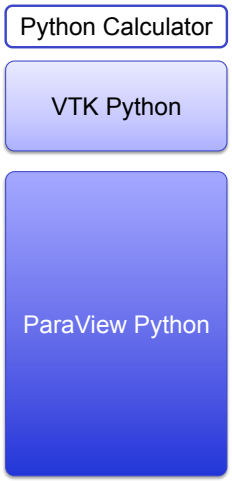


- シミュレーション・コードの対応作業が必要
- ディスクI/Oを介さず、全てのデータをオン・メモリで処理
- 大規模計算に最適
- 全プロセスがPythonスクリプトを解釈 (Symmetric MPI mode)

ParaViewにおけるPythonの適用(7): 一覧

- 以上まとめると

1. Python Calculator
2. Programmable Filter
3. Programmable Source
4. Python Trace
5. Python Macro
6. Python Shell
7. **pvpython**
8. **pvbatch**
9. CoProcessing



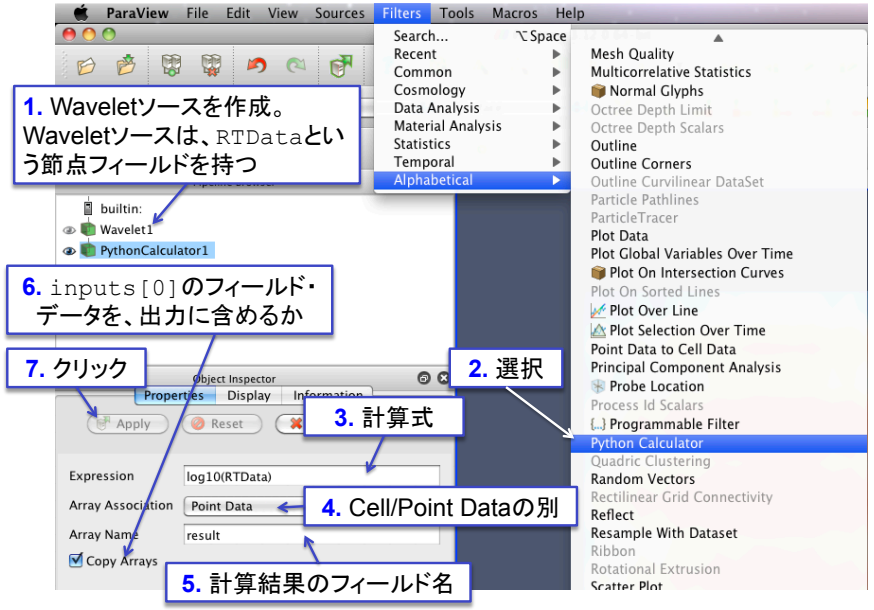
- 以下、順に説明

1. Python Calculator

Python Calculator (1): 概要

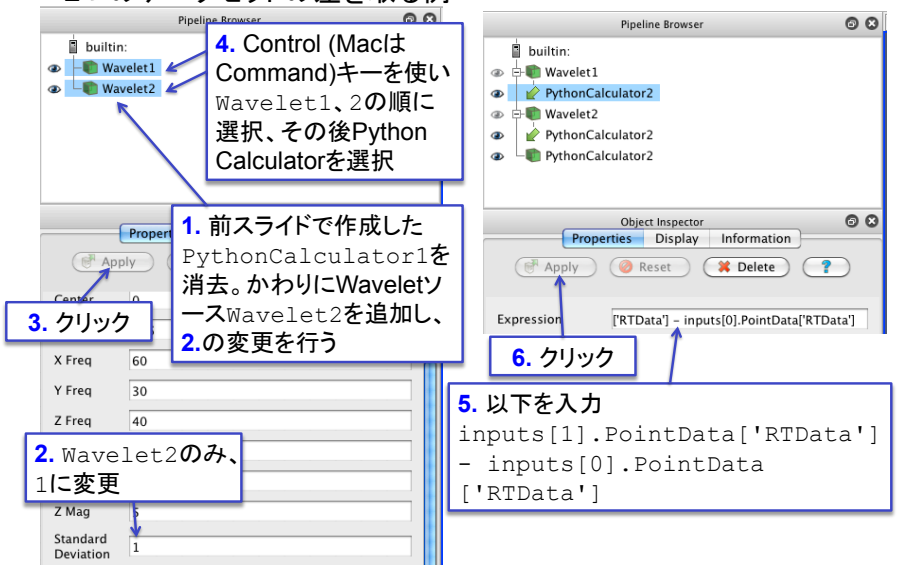
- フィルタの一種、Programmable Filterの簡易版
- 最も簡単なPythonスクリプティング (VTK Pythonの簡略化記法)
- フィールド・データの一様な演算に便利
- `abs`, `cross`, `curl`, ...など関数一覧、詳細な使い方:
  - http://paraview.org/Wiki/ParaView/Users_Guide/Python_Calculator
- 複数の入力間の演算も可能
  - `inputs[番号]`: 入力データセットの選択
    - `.PointData['フィールド名']`: 節点フィールドの選択
    - `.CellData['フィールド名']`: セルフィールドの選択
    - `.Points`: 節点座標

Python Calculator (2): 使用例1 フィールドに関数を適用



Python Calculator (3): 使用例2 データセット間の差分

- 2つのデータセットの差を取る例



Python Calculator (4): 注意点

- 出力メッシュは、`inputs[0]`のものとなる
- フィールド・データのサイズは、全ての入力データセットで等しい必要がある
  - 通常は、同一のメッシュ(形状・節点数・セル数・トポロジ)に対して使用する
  - 並列実行時には、全てのデータセットがプロセス間で同じように配分されている必要がある
- フィールド・データはNumPy配列として扱われる
  - NumPyと同様な演算が可能

2. Programmable Filter

Programmable Filter (1): 概要

- ParaViewフィルタに用意されていない機能を、Pythonによって自分で作成できる
- サーバ側で実行される
  - サーバの並列実行時は並列実行される
- プログラミング・インターフェイス: 以下の混在が可能
  - Python Calculator
  - VTK Python: VTKの知識が必要
- VTKデータ表現とNumPyデータ表現の相互変換が可能
  - NumPyによる演算が可能

Programmable Filter (2): 例1 Python Calculator記法

- Python Calculatorの例2(データセット間の差分)と同じ処理を、Python Calculator記法によって記述
- `Wavelet1`、`2`ともに選択後、Programmable Filterを選択
- Script: 欄に以下を入力、Apply

```
idi0 = inputs[0]
idi1 = inputs[1]
diff = idi1.PointData['RTData'] - idi0.PointData['RTData']
output.PointData.append(diff, "result")
```

- フィールド・データはNumPy配列として扱われる
- より高度な処理を行いたい(メッシュも操作したいetc)
  - VTK Python: 次スライド

Programmable Filter (3): 例2 VTK Python

- 同じ処理を、VTK Pythonによって記述

```
input0 = self.GetInputDataObject(0, 0)
input1 = self.GetInputDataObject(0, 1)

data0 = input0.GetPointData().GetArray('RTData')
data1 = input1.GetPointData().GetArray('RTData')

diff = vtk.vtkFloatArray()
diff.SetNumberOfTuples(data0.GetNumberOfTuples())
diff.SetName('result')

for i in range(data0.GetNumberOfTuples()):
    diff.SetValue(i, data1.GetValue(i) - data0.GetValue(i))

output = self.GetOutputDataObject(0)
output.GetPointData().AddArray(diff)
```

- Pythonの`for`ループは効率が悪い、処理内容がわかりにくい
  - VTKデータ・NumPy配列相互変換: 次スライド

Programmable Filter (4): 例3 NumPyとの相互変換

- 同じ処理を、VTKデータ・NumPy相互変換を用いて記述

```
from paraview.vtk import dataset_adapter as DA

input0 = self.GetInputDataObject(0, 0)
input1 = self.GetInputDataObject(0, 1)

data0 = input0.GetPointData().GetArray('RTData')
data1 = input1.GetPointData().GetArray('RTData')

data0np = DA.vtkDataArrayToVTKArray(data0) # VTK to NumPy
data1np = DA.vtkDataArrayToVTKArray(data1) # VTK to NumPy

diffnp = data1np - data0np # NumPy operation!

diff = DA.numpyTovtkDataArray(diffnp) # NumPy to VTK
diff.SetName('result')

output = self.GetOutputDataObject(0)
output.GetPointData().AddArray(diff)
```


3. Programmable Source

Programmable Source (1): 概要・例 ポリゴンメッシュ生成

- Sources→Programmable Source
  - Programmable FilterのSource版
- 四角形1つのポリゴンメッシュ生成例

```
pdo = self.GetPolyDataOutput()
pdo.Allocate()

pts = vtk.vtkPoints()
pts.InsertNextPoint(-1, -1, 0)
pts.InsertNextPoint(1, -1, 0)
pts.InsertNextPoint(1, 1, 0)
pts.InsertNextPoint(-1, 1, 0)
pdo.SetPoints(pts)

ptIds = vtk.vtkIdList()
ptIds.InsertNextId(0)
ptIds.InsertNextId(1)
ptIds.InsertNextId(2)
ptIds.InsertNextId(3)
pdo.InsertNextCell(vtk.VTK_QUAD, ptIds)
```

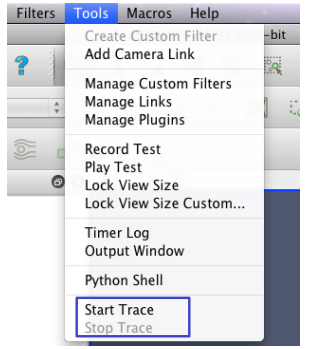

Programmable Source (2): パイプラインのクリア

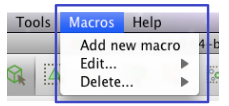
- ここで、ツールバー上のDisconnectボタンをクリックして、パイプラインをクリアして下さい

4. Python Trace
5. Python Macro

Python Trace / Python Macro (1): 概要

- ParaView GUI上で行った操作を、ParaView Pythonスクリプトとして書き出す
- 書き出されたスクリプトは、Macros...メニューに登録して実行できる
  - 操作の自動化: ParaView Pythonを知らなくても、バッチ処理スクリプトを作れる
  - 書き出されたコードを読んで、ParaView Pythonを学習



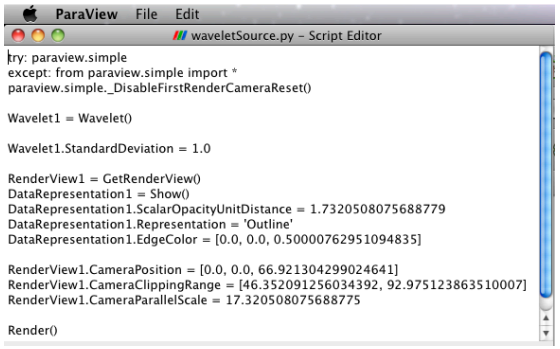


Python Trace / Python Macro (2): 例 ソースの作成

- ソースを1つ生成する例

1. Tools→Start Traceを選択
2. Sources→Waveletを選択、Standard Deviationを1とし、Apply
3. Tools→Stop Traceを選択

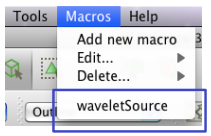
- スクリプトが生成され、スクリプト・エディタに表示される



Python Trace / Python Macro (3): 例 ソースの作成

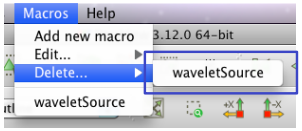
- File→Saveを選択、適当なフォルダに保存
  - ParaView Pythonスクリプトとして保存される

- File→Save As Macro...を選択、ダイアログのデフォルトフォルダに保存
  - ファイルの内容はFile→Saveによる保存ファイルと同一
  - Macrosメニューに登録される
  - ツールバーにも登録される



Python Trace / Python Macro (4): 例 ソースの作成

- 登録されたマクロの削除: Macros→Delete...→削除したいマクロを選択

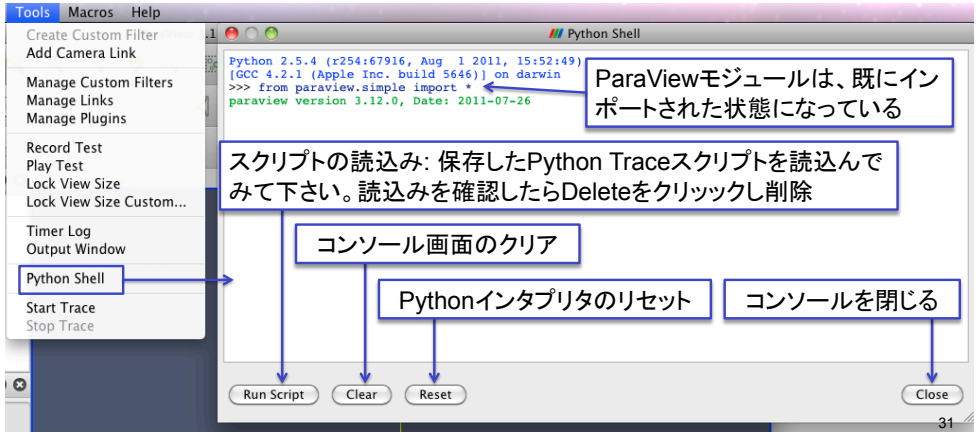


- ここで、ツールバー上のDisconnectボタンをクリックして、パイプラインをクリアして下さい

6. Python Shell

Python Shell (1): 概要

- Tools→Python Shellを選択: **`paraview`**上のPythonコンソール
- ParaView Pythonを実行可能
- スクリプトを手で入力しながら実行 (↑↓キーで入力履歴を辿れる)



Python Shell (2): モジュールのインポート

- ParaView Python
  - `paraview.simple`: ひととおりの制御を行う
  - `paraview.servermanager`: 高度な制御を行う
  - `paraview.vtk`: ParaView Python/VTK Pythonの混在
- ここでは主に`paraview.simple`モジュールを扱う
- `paraview.simple`モジュールのインポート

```
from paraview.simple import *
```

  - Python Console上では不要
  - それ以外では必要
- 以下のようにすると、どちらの場合も対応できる

```
try: paraview.simple
except: from paraview.simple import *
```

Python Shell (3): ソースの作成

- ソースの作成

```
wavelet1 = Wavelet()
```

  - 全ソースとフィルタについて、生成のための関数がある
  - 関数名は、GUI上に表示される名前からホワイトスペースや特殊文字を除いたもの
- ソースの作成と同時に、プロパティを設定できる

```
wavelet2 = Wavelet(StandardDeviation=1)
```

  - GUI上のプロパティパネルに、設定値が反映される
- 作成されたWaveletオブジェクトの実体はサーバ上に存在
  - ParaView Pythonは代理オブジェクト(プロキシ)を操作

Python Shell (4): ソースのプロパティ

- プロパティ名も、基本的にGUIのプロパティ・パネルに表示される名前からホワイトスペースや特殊文字を除いたもの
  - 例外多数
- プロパティー覧

```
dir(wavelet2)
help(wavelet2)
```

  - ただし、ParaViewが内部的に使用するプロパティも多い
- ユーザレベルで設定するプロパティの一覧

```
wavelet2.ListProperties()
```

- 具体的な設定方法は、Python Traceでの確認がおすすめ

Python Shell (5): オブジェクトの削除

- オブジェクトのパイプラインからの削除

```
Delete(wavelet2)
```

  - ソース、フィルタ、レプレゼンテーション、レンダー・ビュー等、全てこれで削除できる
- さらに、メモリ上から完全に削除するには

```
del wavelet2
```

- 以下がエラーになることで、オブジェクトの削除を確認

```
wavelet2
```

- 後の演習を続けるため、再度`wavelet2`を作成(↑キーを使用)

```
wavelet2 = Wavelet(StandardDeviation=1)
```

  - パイプライン・ブラウザ上では、`Wavelet3`

Python Shell (6): レプレゼンテーションの作成

- アクティブなソース(直前に作成したソース)のレプレゼンテーションを作成し、可視状態にする

```
drw2 = Show()
```

- レプレゼンテーションをSurface With Edgesにする

```
drw2.Representation='Surface With Edges'
```

- `RTData`の最小値−最大値範囲で色付けする

```
drw2.ColorArrayName='RTData'
r = drw2.Input.PointData[drw2.ColorArrayName].GetRange()
drw2.LookupTable = MakeBlueToRedLT(r[0], r[1])
```

- レプレゼンテーションには、多くのプロパティがある。詳細は

```
dir(drw2)
help(drw2)
```

  - または、Python Trace

Python Shell (7): レンダー・ビューの作成

- ビューをレンダリングし、レンダー・ビューを取得する

```
rv = Render()
```

- ビューのサイズを変更する (Python Consoleでは効かない)

```
rv.ViewSize = [1024, 768]
```

- 時系列データに対して、現在の時刻を指定する(後で使用)

```
rv.ViewTime = 0.0
```

- レンダー・ビューには、多くのプロパティがある(特にカメラ関係)。
  詳細は

```
dir(rv)
help(rv)
```

  - または、Python Trace

Python Shell (8): 可視・不可視状態の制御

- 指定したソースを可視状態にする

```
Show(wavelet1)
Render()
```

- アクティブなソース(直前に作成したソース)を不可視状態にする

```
Hide()
```

- 指定したソースを不可視状態にする

```
Hide(wavelet1)
Render()
```

- アクティブでないソースには、`Show()`/`Hide()`の後に`Render()`が必要

Python Shell (9): アクティブなソース、ソース一覧

- アクティブなソースを取得する

```
GetActiveSource()
```

- アクティブなソースを変更する

```
SetActiveSource(wavelet1)
```

  - パイプライン・ブラウザ上のハイライトも変更される

- ソースの一覧を取得する

```
GetSources()
```

  - 一覧は`('ソース名', 'ID')`のタプルをキーとするPythonの辞書形式リストになっている。リストからいずれかの要素を抽出するには、例えば以下のようにする(IDは状況により異なる)

```
GetSources()[('Wavelet1', '291')]
```


Python Shell (10): 1入力フィルタの作成・設定

- アクティブなソースに対して、フィルタを作成

```
threshold = Threshold(ThresholdRange=[0, 150])
Show()
Render()
```

- しきい値を変更して、フィルタを再度実行

```
threshold.ThresholdRange = [200, 250]
Render()
```

- フィルタの入力を変更

```
threshold.Input = wavelet2
Render()
```

  - フィルタの作成時にも、`Input`キーワード引数によってアクティブなソース以外を指定可能

Python Shell (11): 複数入力フィルタの作成・設定

- Python Calculatorの例2(データセット間の差分)を、ParaView Pythonによって記述
- wavelet1、wavelet2を入力としてPython Calculatorを作成

```
pycalc = PythonCalculator(Input=[wavelet1, wavelet2])
```

- 計算式スクリプトをExpressionプロパティに設定

```
pycalc.Expression="inputs[1].PointData['RTData'] - inputs
[0].PointData['RTData']"
```

  - Programmable Source/Filterでも、同様にScriptプロパティにPythonスクリプトを設定可能 (Python Trace参照)

- レプレゼンテーションを設定

```
drpc = Show()
drpc.ColorArrayName = 'result'
r = drpc.Input.PointData[drpc.ColorArrayName].GetRange()
drpc.LookupTable = MakeBlueToRedLT(r[0], r[1])
Render()
```


Python Shell (12): 画像の保存

- 現在のビューの画像を、適当なフォルダにview.pngのファイル名で保存(パスは適宜読み替えて下さい)

```
WriteImage('/Users/ohshima/Desktop/view.png')
```

- view.pngが作成されているのを確認したら、Disconnectをクリックしてパイプラインをクリアして下さい

Python Shell (13): データファイルの読込み1

- ParaViewData内のフォルダに移動して、以降のフォルダ・パス入力を削減(パスは適宜読み替えて下さい)

```
import os
os.chdir('/Users/ohshima/Desktop/ParaViewData-3.12.0/Data')
```

- データフォーマットをParaViewに認識させる場合
  - 主に拡張子で判断される

```
can = OpenDataFile('can.ex2')
Show()
rv = Render()
```


Python Shell (13): データファイルの読込み2

- データ・リーダを明示的に指定する場合
  - ファイル名を1つだけ指定するリーダ

```
box = ExodusIIReader(FileName='box.ex2')
```

  - 複数のファイルを時系列として認識するリーダ

```
mc = XMLStructuredGridReader(FileName=['multicomb_0.vts',
'multicomb_1.vts'])
```

    - LegacyVTKReaderでは、FileNameでなくFileNames

- 今回はこれらのデータを使わないので、確認したら消去

```
Delete(box)
del box
Delete(mc)
del mc
```

Python Shell (14): 時系列データのアニメーション

- アニメーション作成は、AnimateReader()を使うのが簡単

```
AnimateReader(can, filename='../can.avi')
```

  - 連番PNGファイルにするなら
    - 40以上のファイルが生成されるので注意

```
AnimateReader(can, filename='../can.png')
```

- または、時刻ステップのリストが、リーダのTimestepValuesプロパティに格納されているのを利用

```
can.TimestepValues
```

  - レンダー・ビューのViewTimeを操作すれば良い

```
for i in range(len(can.TimestepValues)):
    rv.ViewTime = can.TimestepValues[i]
    Render()
    WriteImage('../can_' + str(i) + '.png')
```


Python Shell (15): VTK Pythonの使用

- ビルトイン・サーバ・モードの場合に限り、ParaView Pythonでサーバ上のVTKオブジェクトの取得が可能

```
canvtk = servermanager.Fetch(can)
```

  - canvtkは、VTKの実体オブジェクト(プロクシではない)
  - VTK Pythonによる操作が可能

```
can.GetClassName()
canvtk.GetClassName()
canvtk.GetNumberOfBlocks()
```

- Python ShellのCloseをクリックして、Python Shellを閉じて下さい

7. pvpython

8. pvbatch

pvpython / pvbatch

- **pvpython** / **pvbatch**: ParaViewモジュールをimportできるよう、モジュール検索パス(sys.path)を設定済みのPythonインタプリタ
- **pvpython**: クライアントとして動作
  - pvserverへ接続

```
c = Connect('serverhost', 11111)
```

- **pvbatch**: サーバとして動作するので、サーバへの接続は不要。**pvbatch**自体をmpirunで並列実行させる

```
mpirun -n 2 pvbatch batchScript.py
```

  - Pythonスクリプトを解釈・実行するのはプロセス0のみ、プロセス1以降はプロセス0の指示により動作

9. CoProcessing

CoProcessing (1): 概要

- シミュレーション・コードにParaViewをライブラリとしてリンク
- シミュレーション・コードのメモリ上のデータをCoProcessing APIを通じて受取り、ディスクI/Oを介さず処理を行う
- 処理結果として、以下を出力
  - パイプライン処理後のデータセット(VTK XML形式)
  - 可視化画像
- 可視化パイプラインは、Pythonスクリプトで表現する
  - CoProcessing特有のコードが追加されたParaView Python
    - 付属プラグイン(CoProcessingPlugin)により、GUI上のパイプラインをスクリプトとして書出し可能

CoProcessing (2): シミュレーション・コード側の対応1

- シミュレーション・コード側では以下を追加
  - C++の場合。ヘッダ・ファイルのインクルードは省略
- セットアップ部分
  - MPI、コプロセッサ初期化、Pythonスクリプトの読込み

```
MPI_Init(&argc, &argv);
vtkCPProcessor *processor = vtkCPProcessor::New();
processor->Initialize();
vtkCPPythonScriptPipeline *pipeline =
    vtkCPPythonScriptPipeline::New();
// Load Python script
if(pipeline->Initialize(pythonScriptFileName) == 0)
  {
  // Error handling
  }
processor->AddPipeline(pipeline);
pipeline->Delete();
vtkCPDataDescription *dataDesc = vtkCPDataDescription::New();
dataDesc->AddInput("input");
```


CoProcessing (3): シミュレーション・コード側の対応2

- 時刻ステップごとのループ内

```
dataDesc->SetTimeData(timeValue, timeStep);
if(processor->RequestDataDescription(dataDesc) != 0)
  {
  vtkCPInputDataDescription *inputDesc
      = dataDesc->GetInputDescriptionByName("input");
  inputDesc->SetGrid(dataObject);
  inputDesc->SetWholeExtent(wholeExtent);
  processor->CoProcess(dataDesc);
  }
```

  - `dataObject`: コード内部のメッシュ表現をVTKデータオブジェクトに変換したもの。変換コードは、適宜作成する必要がある
    - 構造格子: おおむね簡単
    - 非構造格子: 場合によっては面倒
  - `SetWholeExtent()`は、構造格子のみ必要

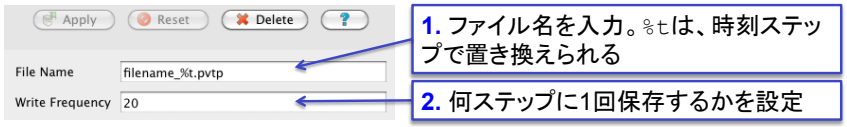
CoProcessing (4): シミュレーション・コード側の対応3

- 終了処理

```
dataDesc->Delete();
processor->Finalize();
processor->Delete();
MPI_Finalize();
```


CoProcessing (5): Pythonスクリプトの書出し1

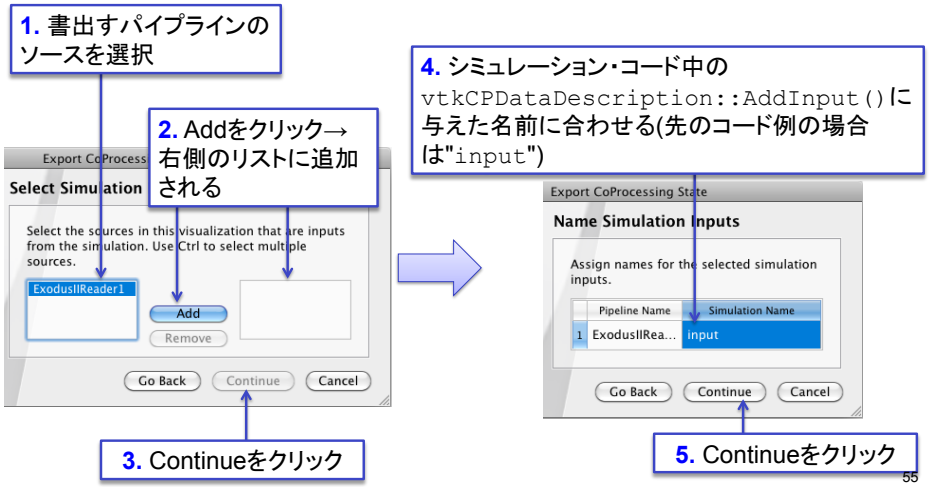
- 適当なデータを使って、パイプラインを作成する
  - Python Shellから引き続き、Sliceを作成、Apply
- Tools→Manage Pluginsを選択、CoProcessingPluginを選択、Load Selectedをクリック、Closeをクリック
  - "CoProcessing"、"Writers"メニューが追加される
- パイプライン処理後のデータセットを出力する場合、パイプライン末尾に"Writer"メニューからライタを追加(無くても良い)



- CoProcessing→Export Stateを選択、Continueをクリック

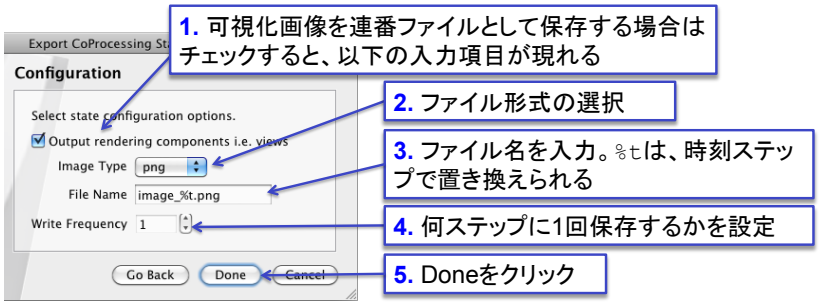
CoProcessing (6): Pythonスクリプトの書出し2

- 書出すパイプラインのソースを選択
- パイプラインに名前をつける



CoProcessing (7): Pythonスクリプトの書出し3

- 可視化画像保存の設定



- Doneをクリックするとファイルダイアログが現れるので、ファイル名を付けて保存する(状況により保存されない不具合有り)
- スクリプトはPythonモジュールとして読み込まれるので、Pythonモジュールとして有効な名前を付けること

CoProcessing (8): 備考

- CoProcessing≒シミュレーション・コードに埋込まれた**pvbatch**
  - 相違点: Pythonスクリプトは全プロセスで実行される
    - Symmetric MPI mode
- シミュレーション・コードのMPI並列実行時は、パイプライン処理も並列動作
  - 案外、実行時間上は負荷にならない
- シミュレーション・コードの内部データとCoProcessing入力データ間のコピーを、なるべく避ける
  - CoProcessing入力データは変更されないことを利用
- (実行時間・メモリ使用量的に)重いフィルタは極力避け、注意深くパイプライン構築

まとめ

まとめ

- ParaViewにおけるPythonの広範な活用を概説
  - Python Calculator、Programmable Filter、Programmable Source、Python Trace、Python Macro、Python Shell、**pvpython**、**pvbatch**、CoProcessing
- ParaViewのPythonスクリプティング機能は多彩
  - 全貌の把握、適材適所の見極めが難しい
  - 本講習が指針となれば幸いです