Oily Python: a Reservoir Engineering Perspective

PyAr – November 17, 2012

Andrea Gavana

Maersk Oil

andrea.gavana@gmail.com
andrea.gavana@maerskoil.com
Outline

✓ What reservoir engineers do
✓ Data pre-processing and number crunching – xlrd and numpy
✓ 2D visualizations – matplotlib
✓ 3D visualizations – VTK, mayavi and NetworkX
✓ Integration with the reservoir numerical simulator – f2py
✓ Automation and N-D interpolation – Python and scipy
✓ Graphical user interfaces (GUIs) – wxPython

Presentation samples: http://www.infinity77.net/pycon/oily.zip
What We Do

✓ Using all sorts of real-life measurements:
  • Man-made seismic waves
  • Detailed record of the geologic formations penetrated by a well (logs)
  • Rock properties, oil/water/gas content in the reservoir rock
  • Pressure/temperature vs. depth in a well
  • Oil/water/gas production rates measured at the well
  • ... and many others ...

✓ A reservoir engineer:
  • Builds a 3D numerical model representing the reservoir and runs time-dependent fluid flow simulations
  • Tries to calibrate that model, i.e., match the simulated results with the real data
  • Using the calibrated model, tries to predict the future performances of the field
What We Do – Complications

- Located underground: we can’t go and see what’s in there
- Sheer areal size – hard to accurately model numerically
- Huge amount of data to pre-process and integrate
- Each simulation can easily generate 100 GB of results to analyze
Data Pre-Processing

“When fed with garbage data, a simulator is a machine that calculates meaningless results with incredible precision.”

✓ A big part of the job is to ensure that the input data makes sense
  • Measurements come from many, unrelated sources
  • Data frequency – both in time and depth – varies wildly
  • Deep and thorough data checking needs to be carried out

✓ Dense visual representations of the input data are fundamental
  • Nothing beats seeing an image of your data to spot errors
  • Automatic filters and data adjustments (via Python code) are inherently limited

✓ Cleaned, sensible data can then be used to feed the simulation
  • One possible source of errors has been removed
Data Pre-Processing – *xlrd*

- Part of the data comes in Excel format (sigh...) – I am no friend with Excel

- *xlrd* is a great, multi-platform Python package to read Excel files
  - Fast as a rabbit – faster than Excel itself
  - Works around many Excel bugs (especially *datetime*-related)

```python
# Open the Excel file
book = xlrd.open_workbook('example_1.xls')
# Get the first worksheet
sheet = book.sheet_by_index(0)

# Allocate an empty numpy array
values = numpy.zeros((sheet.nrows, 3))

# Loop over all the Excel sheet rows
for row in range(1, sheet.nrows):
    # Get the well name
    well_name = sheet.cell(row, 0).value

    # Column B should be a date...
    cell_type = sheet.cell(row, 1).ctype
    cell_value = sheet.cell(row, 1).value

    if cell_type == xlrd.XL_CELL_DATE:
        # It's a date!
        date = xlrd.xldate_as_tuple(cell_value, book.datemode)
        date = datetime.date(*date[0:3])

    # Store production data into a numpy array
    for col in range(3):
        values[row, col] = sheet.cell(row, col+2).value
```

- Smoothly handles different cell types (empty, text, number, boolean, etc...)
- Various Excel-errors handling (#REF!, #DIV/0!, #VALUE!, etc...)
- Info on cell fonts, formats, formulae
- It’s the base of *XLSGrid* (an AGW widget in *wxPython*) 😊

Oily sample: *xlrd_1.py*
Number Crunching and I/O

Task of the day

✓ Quality check of the electrical measurements on a well (logs)
✓ Depth-based data at 15cm intervals (well length can be more than 10Km)
✓ Free format text file with variable-length headers
  • Data is organized in columns
✓ We only care about depth, rock property and water content
  • All other data is discarded
✓ Unphysical values must be filtered out ($X < 0$ or $X > 1$)
✓ Cleaned data is then exported in another format
  1. Keeping original depth intervals (15cm)
  2. Averaging rock property and water content every 6m
Number Crunching and I/O

Problem size and available resources

- 860 wells, 4.9 GB of data scattered over a network
- Python 2.7 on Windows Vista:
  - CPU @ 3.46 GHz, 64 bit architecture
  - 16 cores, 96 GB or RAM

**Oily sample:** *numpy_1.py*
Number Crunching and I/O – *numpy*

- `loadtxt` is very handy and fast
- Returns a 2D *numpy* array
- Supports a wide range of file formats by tweaking its keyword arguments
- Fast and intuitive operations on N-D arrays
- `savetxt` is as handy and as fast as `loadtxt`
- A moving average implementation is a 2-liner with *numpy*

```python
# We skip the first 43 rows of the text file
skip = 43

# Column 0 = Depth
# Column 8 = Rock property
# Column 13 = Water content
columns = (0, 8, 13)

# 1. Load the data using numpy.loadtxt
data = numpy.loadtxt('log.prn', skiprows=skip, usecols=columns)

# 2. Filter out the bad values for rock property
# and water content
rock_water = data[:, 1]
rock_water[rock_water < 0] = -999
rock_water[rock_water > 1] = -999

data[:, 1:] = rock_water

# 3. Save the filtered data to a new file
numpy.savetxt('log_out.prn', data, fmt='%-15s')

# 4. Moving average every 20ft - 6m
# a. Set negative (default) values to NaN
averaged = numpy.where(data < 0, numpy.NaN, data)

# Pre-allocate a matrix for the averaged values
out_averaged = numpy.zeros((5, averaged.shape[1]))

for col in xrange(averaged.shape[1]):
    out_averaged[:, col] = moving_average(averaged[:, col], 40)
```
Final results and performances

 ✓ Looped through all the files in 6.5 minutes

 ✓ Can we do better?
   • Yes we can – go parallel with the multiprocessing module
   • The task is easily parallelizable: one file at a time

 ✓ Windows is less suited to parallel stuff than other platforms (no os.fork())

 ✓ Nevertheless, this approach gives stupendous speed gains

 ✓ If I am I/O-bound... I don’t care
Number Crunching and I/O – *numpy*

Petrel RE Reference Project - Logs Processing Time

- Processing time (minutes)
- Processing speed (files/s)
- Memory consumption - RAM (GB)
2D Visualizations

“A picture is worth a thousand words.”

- We produce visualizations for every data type in our datasets
  - Visual inspection is a powerful solution to spot errors
  - Everyone in the team has a chance to analyze the data
  - Often provide new insights on how to better integrate the data

- The generated plots contain as much information as possible

- *matplotlib* is the Python package of choice
  - Almost limitless customizations of plots
  - Very high plot quality and wide range of plot types
  - Easy integration with GUI toolkits (*wxPython, Qt, PyGtk, TkInter*)
2D Visualizations – matplotlib

Reservoir Fluids
- Gas
- Gas/Oil
- Oil
- Oil/Water
- Gas/Water
- Water

TVDSS
- Pressure
- Mobility
- Trajectory
- Bad Test

WELL (01-Jan-2100)

True Vertical Depth - TVDSS (ft)

Measured Depth (ft)

Pressure (psia)

Mobility (mD/psi)

[TVD_{TVD} - TVD_{ROP}] = 3.3 ft (0.11%) 
\rho_{fluid} = 0.34 \text{ psi/ft} 
Weight_{mud} = 10.0 \text{ ppg} 
RT_{direction} = N/A 
T = 124.3 \text{ °F}
2D Visualizations – *matplotlib*

```python
from mpl_toolkits.axes_grid1 import host_subplot
import mpl_toolkits.axisartist as AA
import matplotlib.pyplot as plt

host = host_subplot(111, axes_class=AA.Axes)
plt.subplots_adjust(right=0.75)

par1 = host.twinx()
par2 = host.twinx()

new_fixed_axis = par2.get_grid_helper().new_fixed_axis
par2.axis['right'] = new_fixed_axis(loc='right', axes=par2, offset=(60, 0))
par2.axis['right'].toggle(all=True)

fig = plt.figure()
ax = fig.add_subplot(111)

colors = ['r', 'g', 'b', 'm', 'y']

for i in range(5):
    start, end = 10*i, 10*(i+1)
    ax.axvspan(start, end, color=colors[i], alpha=0.1)

    reservoir = 'Reservoir %d' % (i+1)
    ax.text(10*i+5, 8, reservoir, fontweight='bold',
            bbox=dict(fc='w', ec='k'), zorder=100,
            ha='center')

plt.show()
```

- Multiple independent Y-axis
- Axis location, ticks, colors, labels, etc... can be tweaked
- *axisartist* supports curvilinear axis as well

Oily sample: *matplotlib_1.py*

- *axhspan* adds a horizontal span (rectangle) across the axis
- *axvspan* is its vertical friend

Oily sample: *matplotlib_2.py*
2D Visualizations – *matplotlib*

```python
fig = plt.figure()
ax = fig.add_subplot(111)

collLabels = ['Event', 'Date', 'Top (ft)', 'Bottom (ft)']

# No row labels
rowLabels = ['', '']

cellText = [['Perforation', '01-Jan-2020', '300', '400'],
            ['Squeeze' , '01-Aug-2030', '0' , '300']]

table = ax.table(cellText=cellText, rowLabels=rowLabels,
                 colLabels=collLabels, bbox=(0.1, 0.7, 0.8, 0.2))

table.auto_set_font_size(False)

plt.show()
```

✓ Tables are a useful addition to *matplotlib* plots

✓ Exact formatting, colors and font may sometimes be hard to get right

Oily sample: *matplotlib_3.py*

```python
# Make a square figure
fig = plt.figure(figsize=(6, 6))

# Add polar axes
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8], polar=True)

# Make some data up
r = numpy.arange(0, 3.0, 0.01)
theta = 2*numpy.pi*r
ax.plot(theta, r, color='$\text{#ee8d18}$', lw=3)
ax.set_rmax(2.0)
ax.grid(True)

plt.show()
```

✓ Polar plots are not widely used in the oil industry

✓ They can be a great tool to analyze a well trajectory

Oily sample: *matplotlib_4.py*
2D Visualizations – *matplotlib*

Drilling Schedule Forecast

- **Reservoirs**
  - Red: Res 1
  - Orange: Res 2
  - Yellow: Res 3
  - Light Blue: Res 4
  - Dark Blue: Res 5
  - Maroon: Res 6

- **Rig 1**
- **Rig 2**
- **Rig 3**
- **Rig 4**
- **Rig 5**
- **Rig 6**

**Date**

[Image of bar charts showing drilling schedule for different rigs and reservoirs]
2D Visualizations – *matplotlib*

```python
fig = plt.figure()
ax = fig.add_subplot(111)
ax.broken_barh([(110, 30), (150, 10)], (10, 9), facecolors='blue')
ax.broken_barh([(10, 50), (100, 20), (130, 10)], (20, 9), facecolors=('red', 'yellow', 'green'))
ax.set_ylim(5, 35)
ax.set_xlim(0, 200)
ax.set_xlabel('Drilling Time (days)')
ax.set_xticks([15, 25])
ax.set_yticklabels(['Rig 1', 'Rig 2'])
ax.grid(True)
ax.annotate('Rig stopped', (61, 25),
            xytext=(0.6, 0.9), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            fontsize=16, ha='right', va='top')
plt.show()
```

✔️ *broken_barh* is the perfect tool to draw drilling schedules

✔️ Similar plots can be obtained by using multiple calls to *ax.barh()*

✔️ Axis annotations add useful info about the data being displayed

Oily sample: *matplotlib_5.py*

I’ll use this occasion to remember John Hunter, the creator of *matplotlib* (1968-2012)
“There's something that 3D gives to the picture that takes you into another land and you stay there and it's a good place to be...”

✓ Most commercial software handle 3D stuff with no effort

✓ 3D visualization in Python is used only for specific, niche problems
  • Simulation results of well production at a specific depth
  • Double-checking input data for the simulation
  • Visualize a relationship between wells, area, reservoir and a project

✓ VTK and mayavi are the most widely used 3D rendering Python packages
  • Scale fairly well on big 3D datasets
  • VTK can easily be integrated in a GUI window (wxPython, Qt, PyGtk, etc...)
  • VTK figures can be saved as VRML files to let the colleagues play with them
3D Visualizations – VTK

- 3D reservoir model, 500,000 cells (VTK unstructured grid)
- We easily go up to 10 million cells, interaction is still smooth
3D Visualizations – VTK

✓ VTK unstructured grids require explicit point and cell representations
✓ 3D Cells can be seen as distorted hexahedrons

```python
# matrix is a (8*Nx*Ny*Nz, 3) 2D numpy array
vtk_pts = array2vtkPoints(matrix)

# Create vtk data
grid = vtk.vtkUnstructuredGrid()
grid.SetPoints(vtk_pts)

# Create cells
ids = numpy.arange(0, 8*nx*ny*nz, dtype=numpy.float32)
ids = numpy.reshape(ids, (nx*ny*nz, 8))
cells = array2vtkCellArray(ids)

# Assign cells to unstructured grid
grid.SetCells(12, cells)

# Actually create the unstructured grid
ugrid = vtk.vtkExtractUnstructuredGrid()
ugrid.SetInput(grid)

ugrid = ugrid.GetOutput()
ugrid.Update()
```

✓ Special techniques exists to handle very large datasets
✓ Coincident points can be merged (faster rendering)
✓ Highlighted functions are available in the `array_handler.py` module as part of the distributed samples
✓ These functions ease the transition between `numpy` arrays and `VTK` arrays

Oily sample: `vtk_1.py`
3D Visualizations – VTK

- Spheres identify a producing interval in a well
- Colors represent the produced fluid (oil, water, gas)
- Spherical slices shows the relative abundance of each fluid
- Each sphere can be “picked”, i.e. selected with the mouse, to display more data
- Time based animation are possible
3D Visualizations – VTK

- `vtkPolyData` can represent vertices, lines, polygons etc...
- `vtkTubeFilter` is a very good way to represent wells in a 3D space
- The well name caption “actor” follows the user view while she interacts with the VTK window
- Highlighted functions are available in the `array_handler.py` module as part of the distributed samples

```python
# x, y, z coordinates of a well trajectory
points = numpy.array(points)
line = [range(len(points))]

# Create the vtk data for the trajectory
vtk_pts = array2vtkPoints(points)
vtk_lines = array2vtkCellArray(line)
poly = vtk.vtkPolyData()
poly.SetPoints(vtk_pts)
poly.SetLines(vtk_lines)

# A filter that generates tubes around lines
profileTubes = vtk.vtkTubeFilter()
# Set the tube radius and resolution
profileTubes.SetRadius(radius)
profileTubes.SetNumberOfSides(20)
profileTubes.SetInput(poly)

# Map vtkPolyData to graphics primitives
wellMapper = vtk.vtkPolyDataMapper()
wellMapper.SetInput(profileTubes.GetOutput())

# Create an "actor" for the well
wellActor = vtk.vtkActor()
wellActor.SetMapper(wellMapper)

# Create a caption "actor" for the well name
textActor = vtk.vtkCaptionActor2D()
textActor.SetCaption(wellName)
```

Oily sample: `vtk_2.py`
3D Visualizations – *NetworkX and mayavi*

- Visualize relationships between wells, areas, reservoirs and projects
- Shows dependencies between wells and undeveloped areas
- 3D version of a *GraphViz* inheritance diagram
- Particularly useful when a project contains 1000s of wells

Oily sample: `mayavi_1.py`
Integration with the Simulator

“Fast as a rabbit, dumb as a stone.”

✓ The reservoir simulator can easily generate 100 GB of results per simulation
✓ Each result set is made of 5-8 interesting files
  • Results are stored in heavily compressed, unformatted binary files
  • These files are generated by a Fortran-based simulator
  • File structure is relatively simple and straightforward
✓ We can use Python to extract the simulation results from these files
  • Performances are generally poor (code is slow)
  • Does not scale well when files are big
✓ Can we write a small Fortran routine and interface it with Python to read these large, binary files?
  • Enter `f2py`
Integration with the Simulator – f2py

✔ Fortran to Python interface generator

✔ Connects the two languages:
  • Creates Python C/API modules from Fortran 77/90/95
  • Works directly on Fortran sources
  • Automatically handles the difference in the data storage order of multi-dimensional Fortran and numpy arrays

✔ Requires a Fortran compiler installed – supports many major compilers, such as gfortran, Intel IVF, Absoft, NAG, etc...

```bash
f2py -c fortran_file.f90 -m py_module
```

✔ Now every Fortran subroutine/function in fortran_file.f90 is accessible in Python by importing py_module
Integration with the Simulator – *f2py*

**Fortran Compilers and f2py vs. Pure-Python**

- Intel IVF
- G95
- GFortran
- Pure-Python

Processing Time (seconds) vs. Binary File Size (MB)
Automation and N-D Interpolation

“Besides black art, there is only automation and mechanization.”

Task of the day

✓ We have 16,000 new simulations available (sensitivities)
  • Each of them represents a unique combination of 13 parameters (oil gravity, rock properties, distance between wells etc...)
  • Simulation results could give insights on the numerical model sensitivity to the parameters variations

✓ The 13 parameters form a discrete set of known data points

✓ Use a f2py-generated module to read results from all the simulations

✓ Use interpolation to estimate results at intermediate values of the parameters
  • scipy offers multi-dimensional interpolation/extrapolation capabilities
  • scipy.interpolate.rbf: uses Radial Basis Function interpolation of N-dimensional scattered data

Oily sample: scipy_1.py
Automation and N-D Interpolation – *scipy*

FOPT @ 30 years vs Well spacing

**Oil API**
- Value 1
- Value 2
- Value 3
- Value 4
- Value 5
- Value 6
- Value 7
- Value 8
- Value 9
- Value 10
- Value 11

**Extrapolation**

**Interpolation**
Graphical User Interfaces

“A picture is worth a thousand words. An interface is worth a thousand pictures.”

✓ User interfaces are an obvious choice when it comes to sharing your findings with non-Pythonistas colleagues

✓ Although many high quality GUI frameworks are available...

✓ **wxPython** is *the* tool I use
  - Almost effortlessly integrate with *matplotlib* and *VTK* (2D and 3D)
  - Easy to build practical, responsive and sexy user interfaces
  - GUIs look (and are) native, whatever the platform
  - Number of widgets available far surpass all other toolkits

✓ Distribution to colleagues is done via *py2exe / PyInstaller* and *InnoSetup* to generate a standard Windows installer
Graphical User Interfaces

Task of the week/month

✓ Create a GUI that evaluates the quality of a calibrated reservoir model

✓ Calibration is good when simulation results are close to measurements (shaded area)

✓ Errors in the calibration are measured by different formulas such as:

\[
Error = \frac{1}{N} \sqrt{\sum_{i=1}^{N} \omega_i \left( \frac{s_i - o_i}{o_i} \right)^2}
\]

✓ The GUI should allow the user to explore the numerical calculations and to quickly plot the simulation results against the measurements
Graphical User Interfaces

Complications

✓ Number of data points: 17 years of historical measurements
✓ Number of wells and simulation time steps (thousands)
✓ The user would like to be able to:
  • Filter out values outside a user-defined date window (per well)
  • Apply a custom multiplier to some of the measurements
  • Exclude some values if a well has been closed for more than X days in a month
  • Modify the error function if a well has been using some gas to ease production
  • Many, many other customizations...
✓ The GUI puts together the power of numpy, f2py, matplotlib, scipy, multiprocessing and wxPython to deliver all that and much more 😊
Graphical User Interfaces
Graphical User Interfaces

Final outcome

✓ We have a fast, practical and nice GUI to examine the quality of model calibration
✓ Colleagues can independently run the GUI and examine the results
✓ Multiple simulations can be analyzed and compared
✓ The interface automagically exports *matplotlib* figures for all the wells and Excel reports (and it does it on multiple processors...)
  • Findings and insights can easily be shared outside the team
  • Consistent, fixed (and beautiful) format for pictures in reports and documents
✓ We have the source code ☺ – any modification is embarrassingly fast
Graphical User Interfaces
The reservoir simulator we use is called ECLIPSE

- It’s keyword-based – you enter inputs in a text file with keywords and sub-keywords
- 1983: first release of ECLIPSE (ECL’s Implicit Program for Simulation Engineering)
- ECLIPSE currently handles ≈1,600 keywords
- On average, each keyword has 3 switches/sub-keywords (≈4,200 in total)
- No editor with syntax highlighting, error checking capabilities and integrated help system exists for the input files (after 30 years!!)

How about a wxPython-based editor with all these capabilities?

- The wxStyledTextCtrl (Scintilla-based) already provides excellent syntax highlighting for various programming languages
- wxPython 2.9 contains powerful HTML viewing capabilities (via wx.html2 module)
- The ECLIPSE input files syntax is very similar to the programming language Lua
Another GUI: *DeckEd*

- *DeckEd* is a text editor based on *wxStyledTextCtrl*
- Syntax highlighting for the reservoir simulator ECLIPSE and more than 60 other programming languages (Python, C++, Java, HTML, PHP, Ruby, etc...)
- Integrated help for the reservoir simulator keywords and sub-keywords
- Runtime monitoring of simulation status and progress
- Runtime error checking for ECLIPSE input files keywords
- Plugin-based architecture – you can add a Python debugger, a spell checker, a code browser, etc...
Graphical User Interfaces

- Keyword Tree
- Real-Time Error Checking
- Integrated Help
- Open Files List
Graphical User Interfaces

- Alphabetical Keyword List
- Real-Time Keyword Help
- Directory Tree
- Keyword Usage Examples
Conclusions

✓ Many, many more examples of the usage of Python in the oil industry that I couldn’t show

✓ Python is becoming increasingly popular amongst reservoir engineers
  • Automation improves working effectiveness a hundredfold
  • Beauty and elegance of the language – easy to grasp even for newcomers

✓ Third-party packages add great value to the standard library:
  • matplotlib – plot customization and unbeatable figure quality
  • numpy and scipy – fast numerical manipulation of multi-dimensional arrays
  • f2py – when you need Fortran raw speed with Python elegance
  • VTK and mayavi – scalable 3D visualization
  • wxPython – the glue to keep all the above together in a nice, point-and-click GUI

✓ Presentation samples: http://www.infinity77.net/pycon/oily.zip