Groundwater

Methods Note/

Scripting MODFLOW Model Development Using Python and FloPy

by M. Bakker[1], V. Post[2,3], C. D. Langevin[4], J. D. Hughes[4], J. T. White[5], J. J. Starn[6], and M. N. Fienen[7]

Abstract

Graphical user interfaces (GUIs) are commonly used to construct and postprocess numerical groundwater flow and transport models. Scripting model development with the programming language Python is presented here as an alternative approach. One advantage of Python is that there are many packages available to facilitate the model development process, including packages for plotting, array manipulation, optimization, and data analysis. For MODFLOW-based models, the FloPy package was developed by the authors to construct model input files, run the model, and read and plot simulation results. Use of Python with the available scientific packages and FloPy facilitates data exploration, alternative model evaluations, and model analyses that can be difficult to perform with GUIs. Furthermore, Python scripts are a complete, transparent, and repeatable record of the modeling process. The approach is introduced with a simple FloPy example to create and postprocess a MODFLOW model. A more complicated capture-fraction analysis with a real-world model is presented to demonstrate the types of analyses that can be performed using Python and FloPy.

Introduction

The construction of a groundwater model can be a complex task depending on the size and geometry of the domain and temporal variability of the processes simulated. Models are commonly constructed with a graphical user interface (GUI). Various GUIs have been developed, with different levels of complexity and sophistication. GUIs generally consist of an interactive environment in which the modeler can construct a model grid, populate the model with hydraulic properties and boundary conditions, and run and postprocess the model results. GUIs provide a level of ease and intuitiveness that is much greater than direct manipulation of the input files, and have therefore become the *de facto* standard for the construction of numerical groundwater models.

In this paper, an alternative approach is presented for developing, running, and postprocessing groundwater models. The approach is based on the development of Python scripts. Python is an interpreted, object-oriented programming language that has gained widespread popularity in science and engineering (Pérez et al. 2011); a short introduction to object-oriented programming in a groundwater setting is given by Bakker and Kelson (2009). Python is a high-level programming language, which means it has a more powerful syntax and a more complete set of data structures than low-level languages (Fortran or C, for example). In a high-level language, complex tasks can be achieved with a few lines of readable code. In addition to the core Python language, there is an extensive library of Python packages for just about any type of scientific analysis. Robust libraries are available for working with arrays (Numpy; Oliphant

[1]Corresponding author: Water Resources Section, Delft University of Technology, Delft, Netherlands; mark.bakker@tudelft.nl
[2]Flinders University, Adelaide, South Australia.
[3]Currently at Federal Institute for Geosciences and Natural Resources, BGR, Hannover, Germany.
[4]U.S. Geological Survey, Reston, VA.
[5]U.S. Geological Survey, Texas Water Science Center, Austin, TX.
[6]U.S. Geological Survey, East Hartford, CT.
[7]U.S. Geological Survey Wisconsin Water Science Center, Middleton, WI.

Article Impact Statement: Python/FloPy scripts are a powerful approach to build and analyze MODFLOW-based models and are a full record of the entire modeling process.

Received October 2015, accepted January 2016.

doi: 10.1111/gwat.12413

2006), making publication-quality graphics (Matplotlib; Hunter 2007), optimization and statistics (Scipy; Jones et al. 2001), working with geospatial information (Fiona; Gillies 2014, Shapely; Gillies 2013), and performing data analysis (Pandas; McKinney 2012). These packages, together with the interactive IPython environment (Pérez and Granger 2007), form the core of what is called the Scipy Stack and are at the heart of exploratory computing with Python. Python itself, the Scipy Stack, and a long list of other packages are open-source software, and can be downloaded and used for free.

The approach presented in this paper is implemented in FloPy, a Python package written by the authors for developing, running, and postprocessing models that are part of the MODFLOW family of codes (i.e., MODFLOW; Harbaugh 2005, MODPATH; Pollock 2012, MT3DMS; Zheng and Wang 1999, and SEAWAT; Langevin et al. 2008). FloPy provides functionality for creating new models as well as working with existing models. The concept is that the user writes a script to construct, run, and postprocess the groundwater model. The script sets Python variables to define the grid, hydrogeological parameters, initial conditions, boundary conditions, solver parameters, and other information required by the model. Python packages for reading and processing geospatial information (e.g., GIS-based shapefiles and rasters) can also be employed to facilitate incorporation of property and boundary data from a variety of sources. Once the model information is defined, FloPy can write input files for the specified models (MODFLOW, MT3DMS, etc.). The script can then be used to run the model, read the model results, including binary model output, and make any type of customized plot. Once model results are read, many other analyses using Python functionality and packages are available.

The approach of using scripts to develop groundwater models is clearly different from using a GUI. Some might argue that it is antiquated because it requires the user to write syntactically correct computer code. There are no well-defined menu bars for performing common tasks, nor is there a main graphic panel showing the model grid, boundary conditions, and model results. In short, the approach is reminiscent of decades-old Unix and DOS batch scripting that so many practicing hydrogeologists have tried to forget. This paper shows that Python's easy-to-read syntax and powerful features justify the rejuvenation of the script-based approach. First, FloPy is introduced with a basic example of one-dimensional flow, followed by a discussion of the power, flexibility, and benefits behind the scripting approach, which is demonstrated by a more complicated example.

A Basic FloPy Example

The use of FloPy requires a basic familiarity with MODFLOW and its packages, see, for example, Harbaugh (2005), as well as a basic understanding of object-oriented code design. MODFLOW itself consists of packages that are labeled with three letter acronyms in the MODFLOW documentation (e.g., BAS, LPF, WEL). These should not be confused with Python packages. A FloPy script for a MODFLOW model commonly consists of at least the following eight steps:

1. Import the FloPy package.
2. Create a MODFLOW model object.
3. Define the model setup, including the discretization, active model cells, starting heads, hydraulic properties, and layer types.
4. Add packages to simulate features of the flow system (e.g., wells, recharge, or rivers).
5. Define the solver that MODFLOW uses to obtain a head solution.
6. Define what output MODFLOW needs to save.
7. Generate MODFLOW input files and call MODFLOW to obtain a solution.
8. Read the (binary) output files for display and further analysis.

As a first simple example, consider steady, one-dimensional, unconfined flow between two long canals with fixed water levels equal to 20 m; the centers of the canals are 2000 m apart. The bottom of the aquifer is at elevation 0 m, and the top of the aquifer at elevation 50 m. The hydraulic conductivity is 10 m/d and the groundwater recharge is 1 mm/d. Two long ditches run parallel to the canals, one at 500 m from the left canal and one at 500 m from the right canal. Both ditches have an extraction rate of 1 m^3/m/d.

The eight steps, as listed above, for developing, running, and postprocessing this groundwater model are as follows[1]:

1. Import the MODFLOW and utilities subpackages of FloPy and give them the aliases `fpm` and `fpu`, respectively

```
import numpy as np
import flopy.modflow as fpm
import flopy.utils as fpu
```

2. Create a MODFLOW model object. Here, the MODFLOW model object is stored in a Python variable called `model`, but this can be an arbitrary name. This object name is important as it will be used as a reference to the model in the remainder of the FloPy script. In addition, a `modelname` is specified when the MODFLOW model object is created. This `modelname` is used for all the files that are created by FloPy for this model.

```
model = fpm.Modflow(modelname = 'gwexample')
```

3. The discretization of the model is specified with the discretization file (DIS) of MODFLOW. The aquifer is divided into 201 cells of length 10 m and width 1 m. The first input of the discretization package is the name of the model object. All other input arguments are self explanatory.

[1]Computer-specific setup, such as specification of the directory that contains the MODFLOW executable, is omitted here.

```
fpm.ModflowDis(model, nlay=1,
               nrow=1, ncol=201, delr=10,
               delc=1, top=50, botm=0)
```

Active cells and the like are defined with the Basic package (BAS), which is required for every MODFLOW model. It contains the `ibound` array, which is used to specify which cells are active (value is positive), inactive (value is 0), or fixed head (value is negative). The `numpy` package (aliased as `np`) can be used to quickly initialize the `ibound` array with values of 1, and then set the `ibound` value for the first and last columns to −1. The `numpy` package (and Python, in general) uses zero-based indexing and supports negative indexing so that row 1 and column 1, and row 1 and column 201, can be referenced as [0, 0], and [0, −1], respectively. Although this simulation is for steady flow, starting heads still need to be specified. They are used as the head for fixed-head cells (where `ibound` is negative), and as a starting point to compute the saturated thickness for cases of unconfined flow.

```
ibound = np.ones((1, 201))
ibound[0, 0] = ibound[0, -1] = -1
fpm.ModflowBas(model, ibound=ibound, strt=20)
```

The hydraulic properties of the aquifer are specified with the layer properties flow (LPF) package (alternatively, the block centered flow (BCF) package may be used). Only the hydraulic conductivity of the aquifer and the layer type (`laytyp`) need to be specified. The latter is set to 1, which means that MODFLOW will calculate the saturated thickness differently depending on whether or not the head is above the top of the aquifer.

```
fpm.ModflowLpf(model, hk=10, laytyp=1)
```

4. Aquifer recharge is simulated with the Recharge package (RCH) and the extraction of water at the two ditches is simulated with the Well package (WEL); the length of each ditch normal to the plane of flow is equal to 1 m (`delc = 1`). The latter requires specification of the layer, row, column, and injection rate of the well for each stress period. The layers, rows, columns, and the stress period are numbered (consistent with Python's zero-based numbering convention) starting at 0. The required data are stored in a Python dictionary (`lrcQ` in the code below), which is used in FloPy to store data that can vary by stress period. The `lrcQ` dictionary specifies that two wells (one in cell 1, 1, 51 and one in cell 1, 1, 151), each with a rate of −1 m^3/d, will be active for the first stress period. Because this is a steady-state model, there is only one stress period and therefore only one entry in the dictionary.

```
fpm.ModflowRch(model, rech=0.001)
lrcQ = { 0: [[0, 0, 50, -1], [0, 0, 150, -1]]}
fpm.ModflowWel(model, stress_period_data=lrcQ)
```

5. The preconditioned conjugate-gradient (PCG) solver, using the default settings, is specified to solve the model.

```
fpm.ModflowPcg(model)
```

6. The frequency and type of output that MODFLOW writes to an output file is specified with the output control (OC) package. In this case, the budget is printed and heads are saved (the default), so no arguments are needed.

```
fpm.ModflowOc(model)
```

7. Finally the MODFLOW input files are written (eight files for this model) and the model is run. This requires, of course, that MODFLOW is installed on your computer and FloPy can find the executable in your path.

```
model.write_input()
model.run_model()
```

8. After MODFLOW has responded with the positive `Normal termination of simulation`, the calculated heads can be read from the binary output file. First, a file object is created. As the modelname used for all MODFLOW files was specified as `gwexample` in step 1, the file with the heads is called `gwexample.hds`. FloPy includes functions to read data from the file object, including heads for specified layers or time steps, or head time series at individual cells. For this simple mode, all computed heads are read.

```
hfile = fpu.HeadFile('gwexample.hds')
h = hfile.get_data(totim=1.0)
```

The heads are now stored in the Python variable `h`. FloPy includes powerful plotting functions to plot the grid, boundary conditions, head, etc. This functionality is demonstrated later. For this simple one-dimensional example, a plot is created with the matplotlib package, resulting in the plot shown in Figure 1.

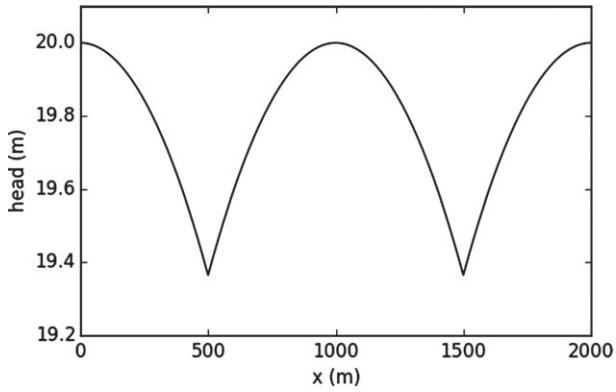

Figure 1. Steady, one-dimensional, unconfined flow between two canals (at $x = 0$ and $x = 2000$) with areal recharge and two ditches (at $x = 500$ and $x = 1500$) where water is extracted.

MODFLOW input files contain much more information than is shown in the script discussed above. The reason more information is not required is that FloPy has default settings for nearly every MODFLOW input parameter. The default value is used when a value is not specified. For example, upon creation of the discretization file, one of the input variables is whether flow is simulated as steady state or transient. In this case, flow is steady, which is the default, so it does not need to be specified. For transient flow, the additional argument `steady = False` needs to be specified (and the storage coefficient needs to be specified as one of the hydraulic properties). FloPy also

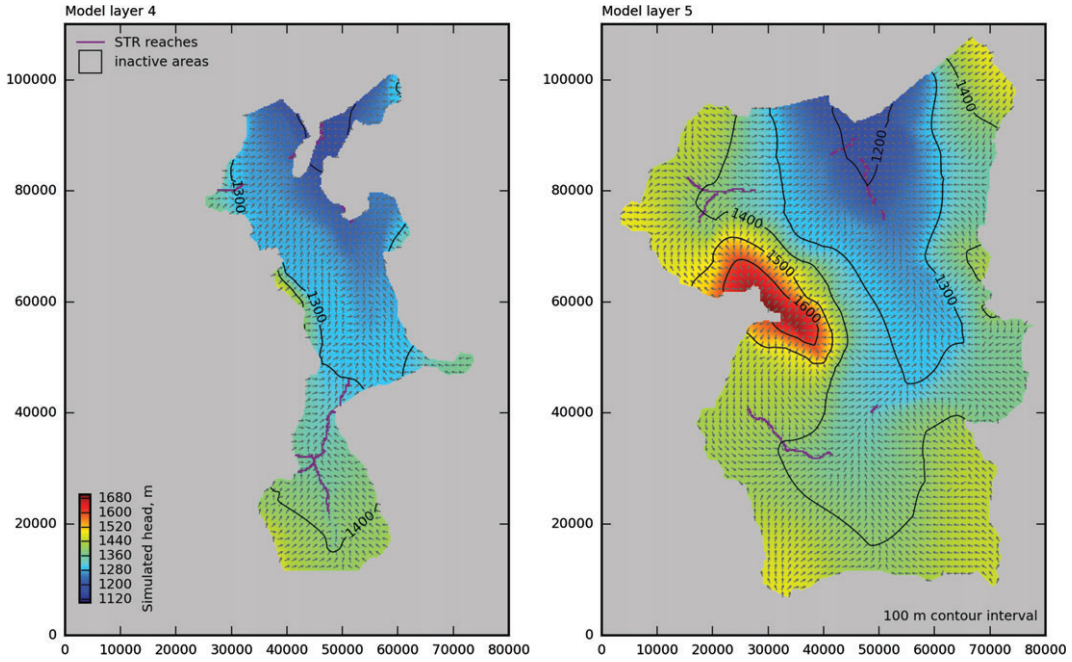

Figure 2. FloPy generated map showing inactive parts of the model, stream boundary conditions, and simulated head and groundwater-flow directions in model layers 4 and 5 of the Upper San Pedro Basin model of Pool and Dickinson (2007). Groundwater-flow directions are shown for every fourth model cell.

has a sophisticated approach for handling arrays that result in MODFLOW input files that are as small as possible. This flexibility allows entire arrays to be specified with a single value, as was used above when the hydraulic conductivity of all cells was specified with a single number (`hk = 10`).

Advantages of Scripting Groundwater Model Development

As was shown in the previous example, use of FloPy to create MODFLOW models requires a certain familiarity with Python, as well as MODFLOW and its packages. For the example presented above, input was relatively simple and could be entered by hand, but for more complicated models, groundwater-related data are likely stored in a variety of file formats. Python packages exist to read and modify data from virtually any source, whether they need to be scraped from websites, read from shapefiles, or retrieved directly from a spreadsheet or database. All that is required is familiarity with the Python package for the specific task, and some lines of Python code to import and convert the data into a format that FloPy can handle. Once loaded into Python, model results can be written to a large number of formats, including image files, shapefiles, georeferenced images, or NetCDF files.

Python scripts can be written to automatically rediscretize a model both in time and in space, in order to analyze how a prediction of interest is sensitive to model resolution. Determination of optimal model resolution is not common practice in groundwater modeling, because it is inefficient to do with existing GUI-based software. Python scripts are ideally suited for analyses that require running the model repeatedly using slightly different input such as sensitivity/uncertainty analysis, parameter estimation, drawdown analysis, and capture analysis. Python packages exist for a variety of parameter estimation and uncertainty analysis methods including Levenberg-Marquardt, Markov Chain Monte Carlo, Simulated Annealing, and Differential Evolution. For a drawdown analysis or capture-fraction analysis, groundwater pumping needs to be added to each cell in a model, one at a time, after which the drawdown or water budget is evaluated and stored.

Besides the unparalleled flexibility of using Python scripts for groundwater model development, the most important advantage is that a script forms a record of the entire model construction process (e.g., Bakker 2014), which makes it transparent and reproducible. Models can easily be shared, web-based repositories can be used for version control, and multiple variants of the model can be created that share the same base data.

An Example of the Use of Python Scripts and FloPy for Analysis

Leake et al. (2010) presented the capture-fraction method to compute which fraction of pumpage comes from which source. This method requires the addition of groundwater pumping and the calculation of water budget changes relative to a base case simulation for each cell in a model layer and is straightforward to automate using FloPy, as shown in the following example.

Leake et al. (2010) presented a capture-fraction analysis for the Upper San Pedro Basin in southeastern Arizona, USA, and northern Sonora, Mexico, based on

a simplified version of the steady-state predevelopment model of Pool and Dickinson (2007). The San Pedro River and associated riparian areas run in a narrow north–south trending band along the axis of the basin. The model has 5 layers, 440 rows, and 320 columns with a constant horizontal grid spacing of 250 m × 250 m. Head-dependent boundaries in the model include streams, drains, and evapotranspiration. A constant inflow was applied to upstream reaches of the stream network. Recharge and constant-head boundary conditions were also applied in the model. The model includes a steady-state stress period and a 100-year transient stress period with 100 time steps of 1 year each.

FloPy was used to load the existing MODFLOW-2000 (Harbaugh et al. 2000) model datasets (developed by Leake et al. 2010) into Python, and rerun the steady-state model using MODFLOW-2005 (Harbaugh 2005). FloPy plot functions were used to plot the boundary conditions, simulated head and specific discharge, and the inactive parts of the model (Figure 2).

A short example of Python code using FloPy plot functions to create a map showing inactive model cells and simulated head and groundwater-flow directions is shown in Figure 3, where it is assumed that the FloPy Modflow object is aliased `fpm`, the flopy plotting package is aliased `fpp`, simulated heads are stored in the array `h`, and the flow right face and front face are stored in the variables `frf` and `fff`, respectively. An existing MODFLOW model is loaded into FloPy using the NAM file (line 1). An instance of the FloPy model map object (`mm`) is created for model layer 4 (model layer 3 in FloPy, as numbering starts at 0) using discretization information contained in the FloPy model object (line 2). FloPy model map objects contain routines for creating many common model plots, such as contour plots, color flood plots, vector plots, and plots of model boundaries. Simulated heads, inactive model areas, stream boundary conditions, and groundwater-flow directions are added to the model map object (lines 3–8). Specification of `masked values=[−888,−999]` in line 3 masks head values in inactive and dry model cells, respectively. Groundwater-flow directions are plotted for every fifth cell (`istep = 5` and `jstep = 5`) because of the large number of grid cells in the model, and the `normalize=True` keyword plots groundwater-flow directions instead of specific discharge rates (lines 6–8). When the plot is complete, it is displayed on the screen with a standard matplotlib call (line 9).

```
1 model = fpm.load('DG.nam')
2 mm = fpp.ModelMap(model=ml, layer=3)
3 mm.plot_array(h, masked_values=[-888, -999])
4 mm.plot_inactive()
5 mm.plot_bc(ftype='STR')
6 mm.plot_discharge(frf[0], fff[0],
7                    istep=5, jstep=5,
8                    normalize=True)
9 plt.show()
```

Figure 3. Python code to load the Upper San Pedro Basin model and generate a map showing inactive model cells and simulated head and groundwater-flow directions for model layer 3 (zero based). Similar code was used to create the maps shown in Figure 2.

A function `cf_model` was written to add a well to any cell, and to compute the combined fraction of the well discharge that comes from drains, streams, or a reduction of evapotranspiration during the simulation period of 100 years (Figure 4). The six input arguments are the FloPy model object `model` for the Upper San Pedro Basin model, `layer`, `row`, `col` of the model cell to which the well is added, the simulated unstressed head-dependent boundary flow `base`, and the simulated pumping rate `Q`.

The `cf model` function removes the existing well package (there are no other wells in the model), creates

```
1  def cf_model(model, layer, row, col, base, Q):
2      model.remove_package('WEL')
3      lrcQ = {1: [[layer, row, col, Q]]}
4      wel = fpm.ModflowWel(model=model, stress_period_data=lrcQ)
5      wel.write_file()
6      model.run_model(silent=True)
7      hfile = fpu.HeadFile('DG.hds', precision='double')
8      cfile = fpu.CellBudgetFile('DG.cbc', precision='double')
9      step_period_list = hfile.get_kstpkper()
10     h = hfile.get_ts((layer, row, col))
11     cap_frac = np.zeros((len(step_period_list)))
12     for tstep, step_period in enumerate(step_period_list):
13         if h[tstep, 1] == model.lpf.hdry:
14             capfrac[tstep] = np.nan
15         else:
16             v1 = cfile.get_data(kstpkper=step_period,
17                                 text='DRAINS', full3D=True)
18             v2 = cfile.get_data(kstpkper=step_period,
19                                 text='STREAM LEAKAGE', full3D=True)
20             v3 = cfile.get_data(kstpkper=step_period, text='ET',
21                                 full3D=True)
22             cap_frac[tstep] = ((v1[0].sum() + v2[0].sum() +
23                                 v3[0].sum()) - base) / (-Q)
24     return cap_frac
```

Figure 4. Python function to compute capture fraction of a well in a specific cell.

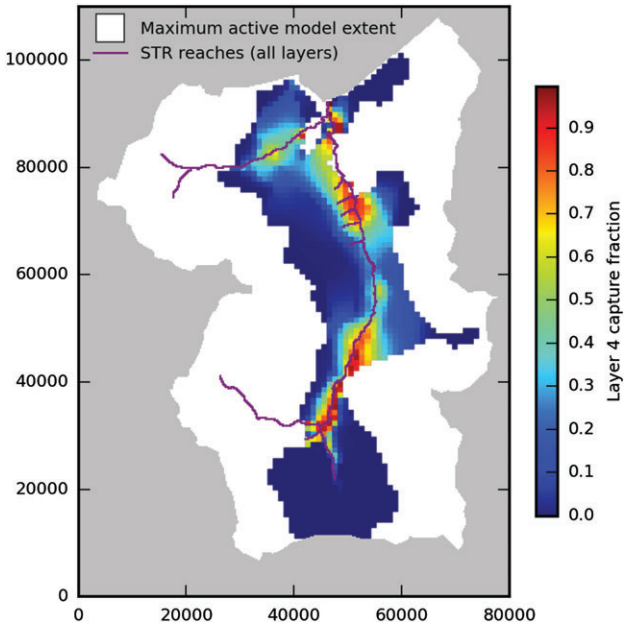

Figure 5. FloPy generated map showing the computed capture fraction of water from head-dependent boundaries as a function of well location in the Upper San Pedro Basin model layer corresponding to the lower basin fill after 10 years of pumpage. The maximum areal extent of the active model domain and the location of stream boundary conditions in all model layers are also shown.

a dictionary specifying the location and discharge of the well for stress period 1 (similar to the previous example), adds the new well package to the model, and writes the new well package file (lines 2–5). Next, the modified model is run and file objects are created for the head file and the cell budget file (lines 6–8). The model name is DG, and the precision is specified as double, because use of a double precision version of MODFLOW-2005 improved model convergence. A list of time steps and stress periods is read from the head file object, the heads are read for all saved time steps at the specified cell, and an empty array is created for the capture fraction at all saved time steps (lines 9–11). Finally, for each saved time step, the net flux to all head dependent boundaries are summed and the capture fraction is computed and returned by the function (lines 12–24). The function returns not-a-number (`np.nan`) for time steps where the layer, row, and column location passed to the function is dry (lines 13–14).

With the capture-fraction calculation available as a function call, it is straightforward to embed the analysis in a loop that iterates over every cell (or a subset of cells) in a model layer to develop an array of transient capture-fraction values for each time step in stress period 2. The computed capture fraction for the lower basin fill (model layer 4) after 10 years of pumping is shown in Figure 5.

Because of the large number of grid cells in the model, pumping locations were only considered in active cells and in every fourth row and every fourth column in model layer 4. This subset of model cells required a total of 1530 model runs and approximately 10 hours to complete the capture-fraction analysis. Parallelization of the capture-fraction analysis using the `multiprocessing` Python package and three cores reduced the analysis time to approximately 4 hours (representing a speedup of 2.4). Further speedup could easily be attained by wrapping this scripted model in a high-throughput run manager (e.g., Fienen and Hunt 2015) and distributing over many machines in a way that would be difficult with a GUI.

Conclusion

Use of Python and FloPy allows programmatic creation and postprocessing of MODFLOW-based models. A programmatic modeling approach facilitates analyses that can be difficult or impossible to complete using currently available GUIs, especially when many model runs with slightly different input are required, such as for parameter estimation, uncertainty analysis, drawdown analysis, and capture-fraction analysis. (Any of these analyses can be built into a GUI by the GUI developers, of course.) Use of a high-level programming language such as Python allows relatively complicated tasks to be achieved with a few lines of readable code.

The first FloPy example presented in this paper shows how to create and postprocesses a simple MODFLOW model. The capture-fraction analysis using the Upper San Pedro Basin model presented in this paper demonstrates the utility of advanced model development using Python scripts. More than 1500 model runs were automatically executed and postprocessed using the Python script developed for the analysis. The final script serves as a record of the steps performed and can be distributed along with the original data and reproduced by other hydrogeologists.

FloPy is a community based open source project hosted on https://github.com/modflowpy/flopy. The scripts discussed in this paper and the supporting files are available on the FloPy website. Installation instructions, documentation, as well as examples demonstrating how to use FloPy to create and postprocess MODFLOW-based models can also be found on the FloPy website.

Acknowledgments

The authors welcome additions, suggestions, and assistance from the groundwater community, and thank all past contributors for their work. Any use of trade, product, or firm names is for descriptive purposes only and does not imply endorsement by the U.S. Government.

References

Bakker, M. 2014. Python scripting: The return to programming. *Ground Water* 52, no. 6: 821–822.

Bakker, M., and V.A. Kelson. 2009. Writing analytic element programs in python. *Ground Water* 47, no. 6: 828–834.

Fienen, M.N., and R.J. Hunt. 2015. High-throughput computing versus high-performance computing for groundwater applications. *Ground Water* 53, no. 2: 180–184.

Gillies, S. 2014. The Fiona user manual. http://toblerity.org/fiona/manual.html (accessed January 16, 2016).

Gillies, S. 2013. The shapely user manual. http://toblerity.org/shapely/manual.html (accessed January 16, 2016).

Harbaugh, A. 2005. *MODFLOW-2005, the U.S. Geological Survey Modular Ground-Water Model—The Ground-Water Flow Process*. U.S. Geological Survey Techniques and Methods, Book 6, Chapter A16, variously paged. Reston, Virginia: USGS.

Harbaugh, A., Banta, E., Hill, M., and McDonald, M. 2000. *MODFLOW-2000, the U.S. Geological Survey modular ground-water model—User guide to modularization concepts and the Ground-Water Flow Process*. U.S. Geological Survey Open-File Report 00-92, 121 p. USGS.

Hunter, J. 2007. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering* 9, no. 3: 90–95.

Jones, E., Oliphant, T., Peterson, P., et al. (2001). SciPy: Open source scientific tools for Python. http://www.scipy.org/ (accessed September 23, 2015).

Langevin, C., Thorne, D., Dausman, A., Sukop, M., and Guo, W. 2008. *SEAWAT Version 4: A Computer Program for Simulation of Multi-Species Solute and Heat Transport*. U.S. Geological Survey Techniques and Methods, Book 6, Chapter A22, 39 p. Reston, Virginia: USGS.

Leake, S.A., H.W. Reeves, and J.E. Dickinson. 2010. A new capture fraction method to map how pumpage affects surface water flow. *Ground Water* 48, no. 5: 690–700.

McKinney, W. 2012. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. Sebastopol, California: O'Reilly Media Sebastopol, CA.

Oliphant, T.E. 2006. *Guide to NumPy*. USA: Trelgol Publishing.

Pérez, F., and B.E. Granger. 2007. IPython: A system for interactive scientific computing. *Computing in Science & Engineering* 9, no. 3: 21–29.

Pérez, F., B.E. Granger, and J.D. Hunter. 2011. Python: An ecosystem for scientific computing. *Computing in Science & Engineering* 13, no. 2: 13–21.

Pollock, D. (2012). *User Guide for MODPATH Version 6–A Particle-Tracking Model for MODFLOW*. U.S. Geological Survey Techniques and Methods, Book 6, Chapter A41, 58 p. Reston, Virginia: USGS.

Pool, D.R., and Dickinson, J.E. (2007). *Ground-water flow model of the Sierra Vista subwatershed and Sonoran portions of the Upper San Pedro Basin, Southeastern Arizona, United States, and Northern Sonora, Mexico*. U.S. Geological Survey Scientific Investigations Report 2006-5228, 49 p. USGS.

Zheng, C., and Wang, P.P. (1999). *MT3DMS: A modular three-dimensional multispecies transport model for simulation of advection, dispersion and chemical reactions of contaminants in groundwater systems*. U.S. Army Engineer Research and Development Center Contract Report SERDP-99-1, Vicksburg, MS, 202 p. USACE.