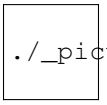
FD Modeling Course Groundwater Flow

Release 0.01

Theo Olsthoorn

May 24, 2017

Contents:




```
Finite Difference Models are derived and implemented completely in Python.
The theory and construction of these models can be used in their own right
or may serve as a thorough introduction in groundwater modeling with available
codes especially with MODFLOW, MT3DMS, MODPATH and SEAWAT.
At the end of this course we have built from ground on a powerful 3D steady state␣
↪and transient finite difference groundwater code completely in Python functions␣
↪and also a powerful 3D particle tracking funcion capable of tracking millions of␣
↪particles simultaneously. We also have seen the versatile use of this code. The␣
↪finite difference model functions are compatible with MODFLOW and MODPATH. The␣
↪only limitation is that the finite difference functions allow just fixed-head␣
↪and prescribed flow boundaries. This is to limit clutter and keep the finite-
↪difference model functions in this course as lean and simple to use as possible,␣
↪but the full theory is presented in the first chapter. This limitation is not a␣
↪problem in most cases as, it's easy to model so-called general-head boundaries␣
↪by setting appropriate conductivities.
For more advanced finite difference modeling and use of more specific packages,␣
↪one should use the USGS codes MODFLOW, MODPATH, MT3DMS and SEAWAT directly. A␣
↪Matlab interface as developed and used by me and my students for the last 8␣
↪years in many projects is available under project mfLab on SourceForge.org. A␣
↪Python interface is available under the name PyFlow as developed by the USGS.
```

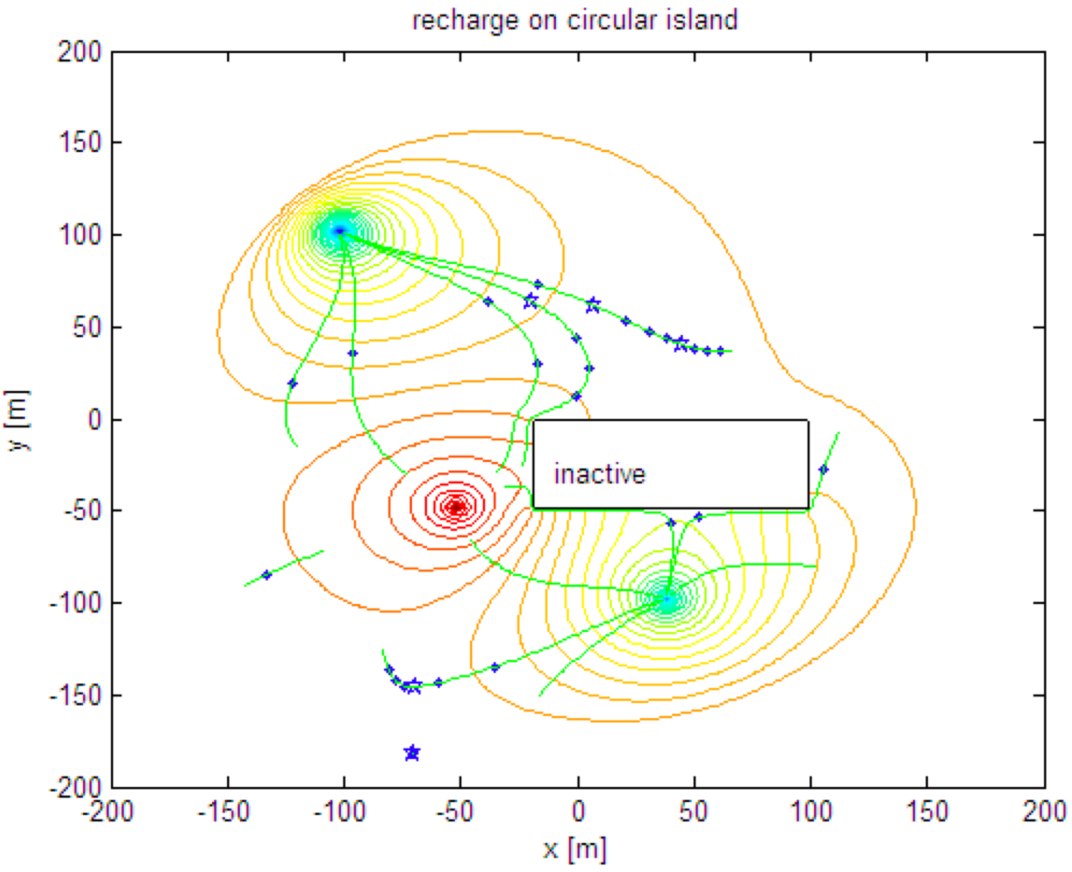
CHAPTER 1

Finite Difference Groundwater Modeling in Python

Previously Matlab-based graduate course at TUDelft

Prof. dr.ir. T.N.Olsthoorn

Dec 31, 2016, 24 May 2016



Abstract ABSTRACT

This syllabus explains the theory behind numerical groundwater modeling and how to make your own finite difference groundwater models in Python. The theory is equally well applicable to other computations and computer language environments like Octave, Scilab and Python. This syllabus aims at providing in-depth insight in numerical modeling of groundwater. It is also base for exercises in the master course CT5440, Geohydrology 2, of the TU-Delft. Although the structure is kept general, and, therefore applicable also to other times of models like finite element models and even surface water flow models, its focus is on finite difference models.

During the course, the student will build his or her own finite difference model in Python. The student will see how flat, axially symmetric, 3D, steady-state and transient models are related. He will also learn how initial and boundary conditions are introduced. Special attention is given to effective treatment of fixed-head boundaries. The models are small Python functions, elegant yet powerful, i.e. capable of simulating simple and small as well as complex and and large groundwater flow problems.The examples serve to demonstrate some things what may be done as well to verify their accuracy including some pitfalls and how to avoid them.

A real world modeling project is generally preceded by a stage where insight is gained into the answers to be provided and the structure and processes relevant in the system to be modeled. In a subsequent step, one or more conceptual models will be made to simulate groundwater behavior under a number of stresses of various types in terms of heads and flows that force the groundwater in the system. Such stresses are surface water elevations, recharge, evaporation, pumping and drainage. The questions to be answered in combination with the relevant complexity of the system also determine the detail of the model mesh to be used, both in space and time. Much time is generally spent on acquiring input and putting it into the form in which the model can use it. Nowadays, much information is often directly drawn from databases and already filled GIS systems, including remotely sensed data such a rain radar. However, one must remain very critical regarding the relevance and correctness of each data item with respect to the modeling problem at hand. In the end, the modeler is responsible for the outcomes, not the model or the computer. The results and predictions often stand at the basis of decisions that will affect livelihoods of people as well as habitats of plants and animals. Lack of time prohibits dealing with with such extended real-world problems in this course. Insight into the internal behavior of the model and the ability to verify its outcomes are more relevant to the engineer, and therefore, is the focus of this syllabus.

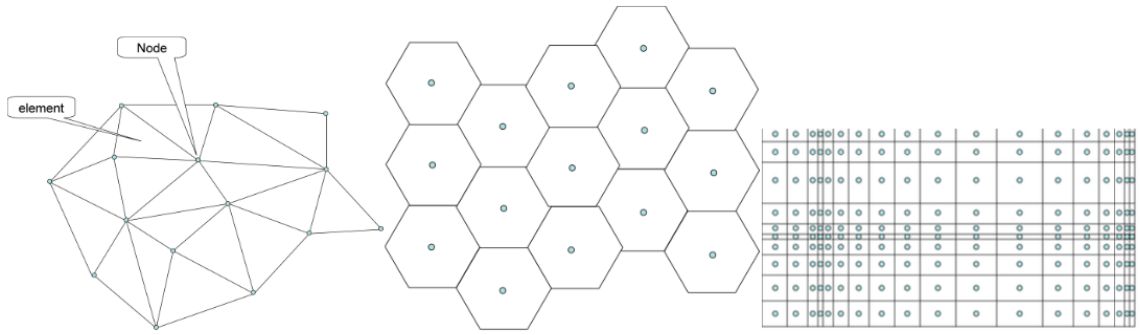


Fig. 1.1: mesh1

Figure: Different model meshes (grid). Left: a finite element triangular network with the nodes at the element corners. Middle: a hexagonal finite difference network with nodes in the center of hexagonal cells. Right: a rectangular finite difference network with nodes in the center of the cells. Area properties are generally specified for elements in the finite element method and for cells in the finite difference method. Heads and flows are generally specified at the nodes of the finite element method and at the cell centers of the finite difference method.

Because MODFLOW, the open-source groundwater model of the United States Geological Survey, is worldwide the most used groundwater model, we'll stay close to its approaches and terminology so that the MODFLOW manual will look familiar to the student. MODFLOW is a fully-implicit 3D finite difference model written in FORTRAN. It can be downloaded together with its manual and source code from HTTP://water.USGS.gov/ogw/modflow.

The Python environment is far more expressive and, from that point of view more powerful than FORTRAN meaning we can set-up a powerful MODFLOW-like model in Python within a few tens of lines of code in way

we can fully understand; MODFLOW requires thousands of lines of FORTRAN that are difficult to grasp unless you are a software engineer with expertise in FORTRAN and modeling at the same time. Python has the power to build a model line by line, interactively, while testing each part of the code immediately on screen, supported by its very powerful debugger, which points at the location where a problem occurred and allows full inspection of the circumstances that caused it.

Next to modeling, Python is also a powerful environment to visualize modeling results. Therefore, outside Python no additional packages are required. Some Python knowledge has, of course, to be acquired during the course. There exist very good Python books, documentation and tutorials on the web; vertually any question related to Python can be "googled" to find useful ansers..

CHAPTER 2

Numerical groundwater modeling

We will start with a general description of groundwater modeling and then derive an actual numerical model, which will finely be converted into a finite difference model by choosing the network and the way the so-called conductance between model cells are computed. The general overview that follows is valid for all kinds of numerical models. We will follow the general approach as long as possible because it provides the best insight with the least clutter.

Numerical models divide the space to be modeled into an often large number subspaces, called elements in the Finite Element Method (FDM) or cells in the Finite Difference Method (FDM). The properties of each element or cell are specified and generally taken constant within. In the FEM, the heads will be computed at the nodes, whereas in the FDM they will be computed at the cell centers. In the FEM, flows will be computed between the nodes, whereas in the FDM they will be computed at the cell faces between adjacent elements. In the FDM the governing partial differential equation directly discretized on the grid, which takes the form of a water balance equation of each cell and, hence, for the model as a whole. The FEM requires that the partial differential equation integrated over each element is satisfied. Solving the model means adjusting all non-fixed nodal or cell heads such that the water balance over all cells and elements are satisfied simultaneously. The FEM and FDM generally lead to different grid shapes, see [fig:Different-model-meshes]. The elements associated with the FEM may be of arbitrary shape, while the shape in the FDM is generally more limited to regular hexagons or rectangular for instance. However, the newest version of MODFLOW, MODFLOW-USG which stands for "Un Structured Grid", brings finite elements and finite differences much closer together by allowing arbitrarily shaped grids, but this is considered beyond the scope of this syllabus.

To stay close to MODFLOW, we will make a finite difference model with rectangular or block-shaped cells in which the properties of the subsurface are assumed constant and at the center of which the heads are computed. The flows are computed at the cell faces, i.e. between adjacent cells.

Although it is straightforward to derive a full 3D finite difference model from the onset, we start with a 2D model for simplicity, where we divide the subsurface into Ny rows and Nx columns. The cell sizes thus defined may vary from column to column and from row to row. The thickness in the z-direction may vary if desired.. This configuration is shown in right-hand picture of figure [fig:Different-model-meshes]. This approach is easy to understand and easy to implement.

The finial result of any of the possible derivations of the model equations, no matter if they are for a finite element model or a finite difference model, comes down to a system of equations, each of which is the water balance for a node or cell of that model. This system of equations represents all nodal water balances. Solving the model is fulfilling these water balances for all models simultaneously. This is achieved computing the unknown heads in the nodes/cells that make all nodal/cell water balances match simultaneously.

The FDM is derived by directly writing down equations for the water balance for the nodes; the FEM takes a more general approach by requiring the governing partial differential equation, which is the water balance on

infinitesimal scale, to be optimally fulfilled within all of the elements. The the FEM is more complicated in deriving its equations and setting up the model, but the bonus is more flexibility in element shapes.

In the end, any numerical groundwater model yields a set of water balances, one for each node. This is true for the FEM, the FDM as it is for any surface water model. In all such models the the space between nodes is replaced by links model types differ only in the way how this is done. In any case, the number of equations, as well as the number of unknowns, equals the number of non-fixed nodes, equals the number of water balances. A finite difference model of 300 rows, 300 columns and 10 layers thus has 0.9 million equations and the same order of unknowns.

Figure [fig:Model-node] shows some of the nodes (or cell centers) of an arbitrary finite element or finite difference model. For one node or cell, with index number i, the adjacent nodes are shown to which it is directly connected, that is, they share one element edge in the FEM or one cell face in the FDM (or one canal or river section in a surface water model). The only difference between these types of models is the way in which the connections are computed. So most of the discussion about modeling and model construction can be done without bothering about these specific details, which is the line followed in this syllabus, because it is most general. For the sake of simplicity whenever the word node is used it can be read as a node in the FEM or equally as a cell center in the FDM.

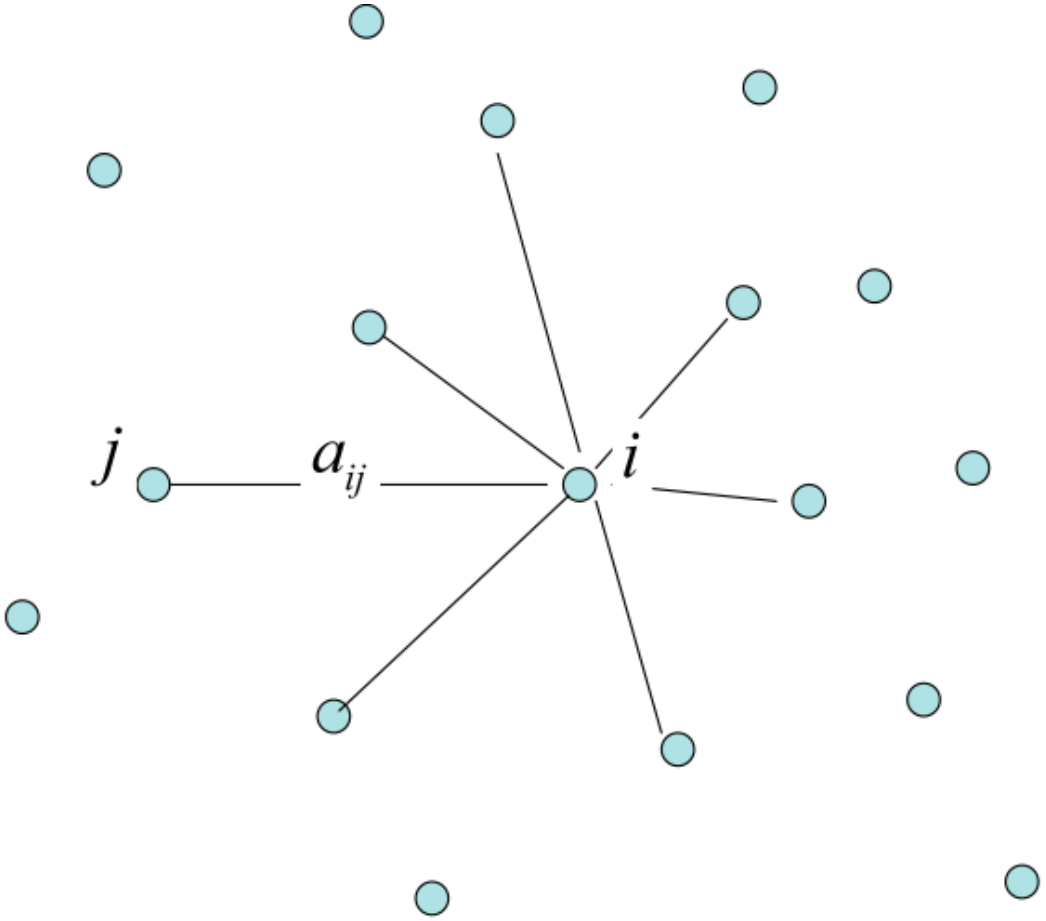


Figure: A model node with its surrounding connected neighbors

Just as general is, that the flow Q_{ij} from node i in the direction of adjacent node j with heads ϕ_i and ϕ_j respectively, is described by

$$Q_{ij} = C_{ij}\left(\phi_i - \phi_j\right)$$

$$= \frac{1}{R_{ij}}\left(\phi_i - \phi_j\right)$$

C_{ij} [(L3/T)/L]} or [L2/T] is called the "conductance" and its reciprocal is the "resistance" [L/(L^3/T)]. The conductance comprises the properties of the area between the connected nodes and their distance. In case the conductance is not constant, as is the case in a surface water model or in a groundwater model with a water table in which the transmissivity is not known a priori, this flow must be computed iteratively.

The physical meaning of the conductance is obvious: it is the flow of water $[L3/T]$ from node i to node j in case the head difference $\phi_i - \phi_j$ [L] equals 1 [L]. The actual dimension depends on the system used, i.g meters and days or feet and hours.

The steady state water balance of an arbitrary node i in the numerical model is described by the following equation

$$v \sum_{j=i,\, j\neq i}^{j=N} Q_{ij} = Q_i$$

Where Q_i is the inflow to the node or cell from the outside world and the left hand side is the combined outflow from the node or cell to all its neighbors. Hence inflow from the outside world into the model is taken positive. The left -hand side thus represents the flow from node i through the model towards its connected neighbors. We will deal with transient models later.

The nodal inflow Q_i, is the sum of all inflows of water from the outside world into node i minus the extractions of water from node i to the the outside world. Therefore, Q_i combines recharge, injections, extractions, leakage, drainage and so on, summed over and integrated over the space attributed to the node (FEM) or cell (FDM).

Using conductances, the nodal water balance becomes:

$$\sum_{j=1,\, j\neq i}^{N} C_{ij}\,(\phi_i - \phi_j) = Q_i$$

Notice that i and j run over all the nodes of the model. This equations expresses that node i may be connected to any or all other nodes of the model no matter how far apart. Of course, in an ordinary model each node is only connected to its direct neighbors. Therefore, most of the conductances C_{ij} are zero. In case a node has n connected neighbors, only $n + 1$ of these conductances are non-zero for each node. Therefore, of a model with N nodes has $N \times N$ possible connections of which N with node i. These connections, and hence, conductances, potentially fill a matrix of N rows and N columns. Notice that a finite difference model with a grid consisting of N_x rows by N_y columns and N_z layers, ha $N = N_x \times N_y \times N_z$ cells, and, therefore, this $N \times N$ array can easily exceed the memory capacity of any available computer. For instance, a model having "only" 300 rows, 300 columns and 10 layers has $N = 0.9$ million cells and hence the $N \times N$ array of possible conductances has $N^2 = 0.81 \times 10^{12}$ entries. With, with 4 bytes per value to be stored this would require a computer memory of 3×10^{12} bytes or about 3 terabyte. This is huge for any internal computer memory. However, if we only store the non-zero values, then the maximum number of conductance to be stored it tremendously reduced. In a 3D finite difference model the maximum number of connected neighbors of any cell is 7. This implies that the number of non-zero values can be no more than $7 \times N$, i.e. $7 \times 4 \times 0.81 \times 10^6 \approx$ 30 Mb in the example model. This memory storage peanuts on even a modern PC with for instance 8 GB internal memory. In fact the array of conductances is extremely sparse. In this case the fraction of non-zero values is at most $7 \times N/N^2 = 7/N \approx 10^{-5}$ or 0.001%.. We will therefore make use of this sparsety when storing the system matrix and solving the model, because, if we do not do this, our computer could not even handle a small size model!

Writing out the above balance equation yields

$$-C_{i1}\phi_1 - C_{i2}\phi_2 - \ldots + \left(\sum_{j=i,\, j\neq i}^{j=N} C_{ij} \right) \phi_{ii} \ldots - C_{i,\,N-1}\phi_{N-1} - C_{i,\,N}\phi_N = Q_i$$

or

$$-C_{i1}\phi_1 - C_{i2}\phi_2 - \ldots + C_{ii}\phi_{ii} \ldots - C_{i,\,N-1}\phi_{N-1} - C_{i,\,N}\phi_N = Q_i$$

where

$$C_{ii} = -\sum_{j=1, j\neq i}^{N} C_{ij}$$

The physical meaning of diagonal matrix element C_{ij} is the amount of water flowing from node i to all its adjacent nodes if the head in node i is exactly 1 m higher than that of its neighbors.

Equation [eq:nodal-water-balance-with-conductances] can be written compactly as follows:

$$\sum_{j=1}^{N} C_{ij}\phi_j = Q_i$$

where the sum taken over all matrix elements in a row equals zero

$$\sum_{j=1}^{N} C_{ij} = 0$$

which means that the flow from node i to node j with $\phi_i - \phi_j = 1$ equals the flow from node j to node i when $\phi_j - \phi_i = 1$. Under special circumstances, this may not be true, in which case the model is non-linear and needs to be solved iteratively.

Equation [eq:system-equation-as-sum] is equivalent to the matrix equation

$$\mathbf{C\Phi} = \mathbf{Q}$$

With $\mathbf{C}$ the square coefficient or system matrix, which holds the conductances $-C_{ij}$, $i \neq j$ and C_{ii} as defined in equation [eq:Cii]. In a 3D finite difference model, both i and j may take values from 1 to $N_x \times N_y \times N_z$. Therefore, the size of $\mathbf{C}$ in such a model is $N_x \times N_y \times N_z$ rows by $N_x \times N_y \times N_z$ columns, which potentially is huge. Φ is the column vector of still unknown heads at the nodes or cell centers (its size is $1 \times N_x N_y N_z$) and $\mathbf{Q}$ the column vector net nodal or cell inflows from the outside world, which has the same size as Φ..

To fill the system matrix, we have to compute the conductances between all connected nodes and put their value into the matrix at location specified by i and j. That is, $-C_{ij}$ goes to row i and column j, $i \neq j$. When done, the coefficients for the diagonal, C_{ii}, are computed by taking the negative sum of the no-diagonal elements in line i of the matrix, which representing node i.

Before deriving the expressions for the conductances, and hence, the how to compute the elements in the system matrix, we consider the model's boundary conditions.

To prevent having to deal with the zeros in over 99% of the system matrix, Python's `scipy.sparse` module offers sparse matrices and sparse matrix functions. These sparse matrices work exactly like ordinary matrices but they store only the non-zero elements. `Scipy.sparse` also offers sparse matrix functions that know how to handle sparse matrices and how to deal only with the non-zeros elements. It is the sparse matrices that make computing of large numerical models feasible on a PC.

Boundary conditions connect the model to the outside world, by linking nodes to heads outside the model or by specifying inflows and extractions, which can be of any type including wells, drainage, recharge and evaporation. Model nodes can also be linked to an outside head through a conductance $\hat{C}$ or a resistance $R = 1/\hat{C}$. Such lines turn out to be a mixture of a fixed head and a fix flow boundary.

Exchange between model nodes and the outside world through flows is quite trivial: all net inflows to (negative if outflows from) the outside world, whatever their type, are directly added to the the inflow at the right-hand size of equation [eq:Model-equation]; i.e. all given inflows minus outflows to node i are added to Q_i in vector $\mathbf{Q}$.

The other types of boundary condition deal with heads, such that the flow between the outside world and the model node is driven by the head difference, and, therefore, is a priori unknown. We treat this in a general way, i.e. by writing out how fixed heads in the outside world connect to nodes of the model through a conductance $\hat{C}$. Heads that are fixed directly at a node of the model, i.e. fixed heads, become a limiting case in which the conductance approaches ∞ or the resistance approaches zero. These heads can and will be handled separately in a way that speeds up the model and stabilizes it.

Consider flow $Q_{ex,\,i}$ into node i from from a water body in the external to the model. Let the head in that water body be fixed and equal to h_i while the head ϕ_i in the model at node i is unknown. This flow through the conductance $\hat{C}_i$ between node and outside world equals

$$Q_{ex,\,i} = \hat{C}_i \left(h_i - \phi_i\right)$$

This flow can be simply added to the right-hand side of the model equation to give

$$\sum_{j=i,\,j\neq i}^{N} -C_{ij}\phi_j + C_{ii}\phi_i = Q_i + \hat{C}_i \left(h_i - \phi_i\right)$$

in which the diagonal C_{ii} was taken out of the matrix for clarity (notice the sum indices).

Equation [eq:Qex] represents a net inward flow , just like the given inflow Q_i.

This way, each model node may be connected to the outside world having arbitrary fixed heads (lakes, rivers and so on).

The constant part , $\hat{C}_i h_i$, works exactly like a fixed inflow. The variable part, $C_i \phi_i$, may be put to the left-hand side of the equation to yield

$$\sum_{j=i,\, j\neq i}^{N} -C_{ij}\phi_j + \left(C_{ii} + \hat{C}_i\right)\phi_i = Q_i + C_i h_i$$

This boils down to adding $\hat{C}_i$ to the diagonal matrix entry, $C_{ii} \rightarrow C_{ii} + \hat{C}_i$.

In matrix form for direct in use in Python, using the subscript $\mathbf{ghb}$ to indicate general head boundary

$$\left(\mathbf{C} + diag\left(\hat{\mathbf{C}}_{\mathbf{ghb}}\right)\right)\mathbf{\Phi} = \mathbf{Q} + \hat{\mathbf{C}}_{\mathbf{ghb}} \cdot \mathbf{h}$$

Where $diag\left(\hat{\mathbf{C}}_{\mathbf{g}}\right)$ is an $N \times N$ diagonal matrix with the elements $\hat{C}_i$. This is indeed equivalent to adding $\hat{C}_i$ to the diagonal elements C_{ii}. Notice that $\hat{C}_i \neq 0$ only where general head boundaries exist, but they can be associated with any cell in the model.

The boundary conditions explained in this section are so-called general head boundaries. In Modflow jargon they are abbreviated to GHB. Truly fixed-head boundaries are dealt with further down.

Modflow has two other variants of these general head boundaries: called drains (abbreviated to DRN) and rivers (abbreviated to RIV). DRNs differ form GHBs in that they only discharge when the head in the model is above the user-specified drain elevation. RIVs differ from GHBs in that the head difference that drives the flow from the river to the connected model node is limited to the water depth of the river; if the head in the model node declines below the river bottom, the river bottom is used instead of the head as explained below.

Drains and rivers thus make the model non-linear as they imply a switch, i.e. cut off or curtail flow depending on the head in the model. Such non-linearities are dealt with using iterative matrix solvers, so that the flows can be updated during the solution process. We will ignore iterative solvers in Python even when the model is non-linear and use a standard (sparse) matrix solver repeatedly when needed, until convergence is achieved. This mostly works faster.

As said above, drains work as general head boundaries as long as the head is above the drain elevation. When the head declines to below the local drain elevation, the flow is set to zero. For the DRN cells we thus need to specify a drain elevation, i.e. a vector $\mathbf{h_{drn}}$ next to the drain conductances $\hat{\mathbf{C}}_{\mathbf{drn}}$. Of course, $\hat{C}_{\mathbf{drn,i}} \neq 0$ only for cells that have drains connected.

The switch may be implemented as a using Boolean vector $\mathbf{b_{drn}}$ which contains true (or 1) for all cells where $\Phi > \mathbf{h_{drn}}$ and false (0) otherwise:

$$\mathbf{b_{drn}} = (\Phi > \mathbf{h_{drn}})$$

Hence, the drains are implemented as follows:

$$\left(\mathbf{C} + diag\left(\hat{\mathbf{C}}_{\mathbf{drn}} \cdot \mathbf{b_{drn}}\right)\right)\Phi = \mathbf{Q} + \hat{\mathbf{C}}_{\mathbf{drn}} \cdot \mathbf{b_{drn}} \cdot \mathbf{h_{drn}}$$

Notice that in Python, a Boolean true becomes 1 if used in arithmetic operations and false then becomes zero. The Boolean vector in the above equation should therefore be read as a vector of ones an zeros.

River boundaries are also general head boundaries as long as the head remains above the bottom of the river. When it falls below the river bottom, h_B, the infiltration is assumed to pass through the unsaturated zone without suction from the fallen head. So, for an arbitrary river node:

$$Q_{riv} = \hat{C}_R\left(h_{riv} - \phi\right), \quad \phi > h_{bot}$$

$$Q_{riv} = \hat{C}_{riv}\left(h_{riv} - h_B\right), \quad \phi \leq h_{bot}$$

writing $b_{riv} = \phi > h_{bot}$ and $\neg b_{riv} = \neg (\phi > h_{bot}) = \phi \leq h_{bot}$

or

$$Q_{riv} = \hat{C}_{riv} (h_{riv} - \phi) b_{riv} + \hat{C}_{riv} (h_{riv} - h_{bot}) \neg b_{riv}$$

In Python where $\neg b_{riv} = 1 - b_{riv}$ this reduces to

$$Q_{riv} = \hat{C}_{riv} (h_{riv} - \phi) b_{riv} + \hat{C}_{riv} (h_{riv} - h_{riv}) - \hat{C}_{riv} (h_{riv} - h_{bot}) b_{riv}$$

$$= \hat{C}_{riv} (h_{riv} - h_{bot}) + \hat{C}_{riv} (h_{bot} - \phi) b_{riv}$$

Therefore MODFLOW-type rivers can be implemented as follows

$$\left(\mathbf{C} + diag \left(\hat{\mathbf{C}}_{\mathbf{riv}} \cdot \mathbf{b}_{\mathbf{riv}} \right) \right) \Phi = \mathbf{Q} + \hat{\mathbf{C}}_{\mathbf{riv}} \left(\mathbf{h}_{\mathbf{riv}} - (1 - \mathbf{b}_{\mathbf{riv}}) \mathbf{h}_{\mathbf{bot}} \right)$$

Combining the three previous sections, the model equation with all general head, drain and river boundaries then becomes:

$$\left(\mathbf{C} + diag \left(\hat{\mathbf{C}}_{\mathbf{ghb}} + \hat{\mathbf{C}}_{\mathbf{drn}} \cdot \mathbf{b}_{\mathbf{drn}} + \hat{\mathbf{C}_{\mathbf{riv}}} \cdot \mathbf{b}_{\mathbf{riv}} \right) \right) \Phi = RHS$$

$$RHS = \mathbf{Q} + \hat{\mathbf{C}}_{\mathbf{ghb}} \cdot \mathbf{h}_{\mathbf{ghb}} + \hat{\mathbf{C}}_{\mathbf{drn}} \cdot \mathbf{h}_{\mathbf{drn}} \cdot \mathbf{b}_{\mathbf{drn}} + \hat{\mathbf{C}}_{\mathbf{riv}} \cdot \left(\mathbf{h}_{\mathbf{riv}} - (1 - \mathbf{b}_{\mathbf{riv}}) \mathbf{h}_{bot} \right)$$

Equation [eq:system-equation-head-boundaries] specifies the complete model from which the heads may be solved directly in Python using the appropriate function (see actual Python code in subsequent chapters).

Combining for simplicity the contribution from the different head boundary conditions under **:raw-latex:'\mathbf{\hat{C}}'** and h, different, the solution of equation [eq:system-equation-head-boundaries] simplifies to:

$$\Phi = \left(\mathbf{C} + diag \left(\hat{\mathbf{C}} \right) \right) \setminus \left(\mathbf{Q} + \hat{\mathbf{C}} \cdot \mathbf{h} \right)$$

where the backslash is "Matlab language" means: solve this set of equations for the unknowns at the left, but don't necessarily invert the matrix left of the \ for computation efficiency reasons.

The column vector $\hat{\mathbf{C}} \cdot \mathbf{h}$ contains therefore the elements $c_i h_i$.

The latter system equation ([eq:system-equation]), which solves for the unknown heads Φ and includes the boundary conditions, represents the complete model .

Once the heads are computed by [eq:system-equation], we may calculate the net inflow of all the nodes or nodes by the matrix multiplication [eq:Model-equation], which must be zero when summed over the entire model

$$\sum \mathbf{Q}_{in} = 0$$

This is an easy check of correct implementation.

We may compute the inflow from all external fixed-head sources (negative if the flow is outward) from

$$\mathbf{Q}_{FH} = \mathbf{C}\Phi - \mathbf{Q}$$

Above we used so-called general-head boundaries, i.e. fixed heads in the outside world that connect with the model through a conductance. The general head boundaries were extended to specific forms, i.e. drains and river boundaries. However, most models also define fixed-head boundaries as nodes in which the heads are directly prescribed and need not to be computed at all..

One way to deal with fixed-head boundaries is through the use of a very large conductance in combination with general head boundaries, i.e. $\hat{C}_i \to \infty$, i.e. say $\hat{C}_i = \Gamma = 10^{10}$ or so) with Γ here representing an infinite value of $\hat{C}_i$.

Then for the fixed-head nodes we have

$$\sum_{j=1,\, j \neq i}^{N} -C_{ij} \phi_j + (C_{ii} + \Gamma) \phi_i = Q_i + \Gamma h_i$$

Because $\Gamma \to \infty$ and so $\Gamma \gg |C_{ii}|$, then by dividing the left and right hand side by Γ, yields

$$\phi_i = h_i$$

This may be all what is needed to fix heads in given nodes. It works well in Python. However, it is inefficiency and the system matrix may become unstable leading to very high condition values with the risk of inaccurate results. But normally no difficulties occur and the results are very accurate. Below we show a better, far more efficient and surely accurate method.

Differentiating between active and inactive cells is common in finite difference modeling with regular grids. Inactive cells represent a part of the grid that does not take part of the model. It might represent bedrock with no groundwater at all. The active cells are the cells for which the heads are unknown and must be computed. Then there is a third category of cells, namely the cells with a fixed head. To differentiate between these three categories of cells, MODFLOW uses its IBOUND array as a three-way Boolean. The IBOUND array has the same shape and number of cells as the grid and contains integers (whole numbers). It is interpreted as follows:

Cells with a value > 0 are active cells with unknown heads.

Cells with a value equal to zero are inactive and therefore excluded from the model

Cells with a value < 0 have a fixed head. The head values are taken from the array with STRTHD values.

The IBOUND array may be just just as a 3-way Boolean, but often also as a zone-array indicating the position of certain features. This is because from the point of view of the model on only thing that matters is whether the IBOUND value of a cell is less than, equal to or larger than zero.

In Python obtaining a Boolean array of active cells, inactive cells or fixed head cell can be done as follows, where we use the `ravel()` method to flatten the 3D shape of the IBOUND array into a long vector: Iact = IBOUND.ravel()>0 Iinact= IBOUND.ravel()==0 Ifh = IBOUND.ravel()<0 These three Boolean arrays have the same possibly 3D shape as the IBOUND array and, therefore as the grid of the model. We can make it column vectors in the way we will use them Iact = IBOUND.ravel()>0 Iinact = IBOUND.ravel()==0 Ifh = IBOUND.revel()<0 If we don't wand a Boolean vector but rather the actual indices of the cells concerned, place find() around the previous expressions: Iac = where(IBOUND.ravel()>0) Iinact= where(IBOUND.ravel()==0) Ifh = where(IBOUND.ravel()<0) Which shows how flexible Python is.

Rather than using an arbitrary large conductance to implement fixed heads as explained above, we may directly implement them in a way that improves the condition of the matrix and reduces the computing time, because the fixed heads nodes do not have to be computed at all; the more cells with fixed heads, the smaller the computational effort and the faster the model will be.

Let the model be described by the system equation as before, and let us ignore general head, drain and river boundaries here just for simplicity and reduce the length of writing in the derivation:

$$\mathbf{C}\Phi = \mathbf{Q}$$

First of all, we may kick out the inactive cells, which could substantially reduce the size of the model. The only cells relevant to our model are the union of the fixed head and the active cells, which is the set of cells that are not inactive. Let $\mathbf{I_{inact}}$ be the set of inactive cells, i.e. a vector of Boolean values indicating such cells, let further $\mathbf{I_{fh}}$ be the vector of Booleans indicating fixed-head cells and let $\mathbf{I_{act}}$ be the vector of Booleans that indicates the active cells, then using :raw-latex:‘neg ‘to mean "not" we have:

$$\neg\mathbf{I_{inact}} = \mathbf{I_{fh}} \cap \mathbf{I_{act}}$$

where $\neg\mathbf{I_{inact}}$ is the vector of booleans in our model that matter, i.e. where groundwater exists.

Hence the reduced model without inactive cells is

$$\mathbf{C}\left(\neg\mathbf{I_{inact}}, \neg\mathbf{I_{inact}}\right)\Phi\left(\neg\mathbf{I_{inact}}\right) = Q\left(\neg\mathbf{I_{inact}}\right)$$

This expresses that we use only those rows and columns of the system matrix and vectors that represent either fixed head or active cells.

Then $\mathbf{C}\left(\neg\mathbf{I_{inact}}, \mathbf{I_{fh}}\right)$ represents all the columns of the system matrix in that will be multiplied by a fixed head, i.e. the heads represented by the vector $\Phi\left(\mathbf{I_{fh}}\right)$. Hence, the matrix-vector multiplication $\mathbf{C}\left(\neg\mathbf{I_{inact}}, \mathbf{I_{fh}}\right)\Phi\left(\mathbf{I_{fh}}\right)$

yields a constant vector for all relevant cells, which may be placed directly at the right-hand side of the matrix equation, leaving the remaining columns $\mathbf{C}\left(\neg\mathbf{I}_{\mathbf{inact}}, \mathbf{I}_{\mathbf{act}}\right)$ untouched at the left hand side. The system equation then becomes:

$$\mathbf{C}\left(\neg\mathbf{I}_{\mathbf{inact}}, \mathbf{I}_{\mathbf{act}}\right)\Phi\left(\mathbf{I}_{\mathbf{act}}\right) = \mathbf{Q}\left(\mathbf{I}_{\mathbf{act}}\right) - \mathbf{C}\left(\neg\mathbf{I}_{\mathbf{inact}}, \mathbf{I}_{\mathbf{fh}}\right)\cdot\Phi\left(\mathbf{I}_{\mathbf{fh}}\right)$$

Because we only have to compute the heads at nodes indexed by $\mathbf{I}_{\mathbf{act}}$, we get a further reduced system of equations. The rows corresponding to the fixed heads may also be eliminated as the fixed heads need not to be computed at all. This results in the following matrix equation:

$$\mathbf{C}\left(\mathbf{I}_{\mathbf{act}}, \mathbf{I}_{\mathbf{act}}\right)\Phi\left(\mathbf{I}_{\mathbf{act}}\right) = \mathbf{Q}\left(\mathbf{I}_{\mathbf{act}}\right) - \mathbf{C}\left(\mathbf{I}_{\mathbf{act}}, \mathbf{I}_{\mathbf{fh}}\right)\cdot\Phi\left(\mathbf{I}_{\mathbf{fh}}\right)$$

Hence, the right-hand side contains the constants and the left-hand side the remaining equations (rows and columns) with the unknown heads. Again, the result is a smaller model that is computationally faster and also better conditioned the the original. The more cells have a fixed head, the faster the model will be.

We use Matlab's backslash (\) operator (called left division) here for convenience of expressing in this math that this this set of linear equations can be directly solved (see acutual python code in subsequent chapters):

$$\Phi\left(\mathbf{I}_{\mathbf{act}}\right) = \mathbf{C}\left(\mathbf{I}_{\mathbf{act}}, \mathbf{I}_{\mathbf{act}}\right) \setminus \left(\mathbf{Q}\left(\mathbf{I}_{\mathbf{act}}\right) - \mathbf{C}\left(\mathbf{I}_{\mathbf{act}}, \mathbf{I}_{\mathbf{fh}}\right)\cdot\Phi\left(\mathbf{I}_{\mathbf{fh}}\right)\right)$$

and where

$$\Phi\left(\mathbf{I}_{\mathbf{fh}}\right)$$

are known beforehand.

With all heads now known, we can compute the nodal inflow of all nodes, including the fixed-head nodes by

$$\mathbf{Q}\left(\neg\mathbf{I}_{\mathbf{inact}}\right) = \mathbf{C}\left(\neg\mathbf{I}_{\mathbf{inact}}, \neg\mathbf{I}_{\mathbf{inact}}\right)\Phi\left(\neg\mathbf{I}_{\mathbf{inact}}\right)$$

This leaves the flows in the inactive cells untouched; they remain whatever they were.

In Python code this would look like the following snippet Phi = STRTHD.ravel(); Q = zeros(IBOUND.shape); Iact = IBOUND.ravel()>0; Iinact = IBOUnD.ravel()==0; Ifh = IBOUND.ravel()<0; Phi[Iact] = C[Iact][Iact] Q[Iact]-C[Iact][Ifh]*Phi[Ifh] Q[!Iinact]= C[!Iinact][!Iinact]*Phi[!Iinact]

In [ ]:

CHAPTER 3

Finite difference modeling

Prof. dr.ir.T.N.Olsthoorn

Heemstede, Sept. 2016, 24 May 2017

Approach

In this chapter we set-up a 3D steady-state finite difference model from scratch. We do this by computing a numerical groundwater problem step by step, by hand, using finite difference, building up the pieces of the model, which we will assemble in the next chapter.

Setup of the model by specifying its dimensions

The 3D steady state FDM will be based on a regular grid consisting of rows an columns and layers. The column widths and the row heigts are constant on a per column and per row basis, but the layer thickness can vary on a cell by cell basis. The grid of a full 3D model will thus be specified in general by a vector of x cell boundary coordinates, a vector y row boundary coordinates and a full 3D array of cell top and bottom coordinates.

Notice that the arrays are interpreted as [z, y, x] or [layer row col]. This is a convenience in Python where when Phi is a 3D array of the shape of the grid [Nz, Ny, Nz] we have

Phi[k].shape is [Ny, Nx], the entire layer number i. Phi[k][j] = Nx, the entire row j of layer i. Phi[k][j][i] = the head in cell [k, j, i] which is the same as Phi[k, j, i]

```
In [1]: import numpy as np

        # specify a rectangular grid
        x = np.arange(-1000., 1000., 25.)
        y = np.arange(-1000., 1000., 25.) # backward, i.e. first row grid line has highest y
        z = np.arange(-100., 0., 20.)  # backward, i.e. from top to bottom
```

From these coordinates we obtain the number of cells along each axis and the cell sizes and

```
In [3]: # as well as the number of cells along the three axes
        Nx = len(x)-1
        Ny = len(y)-1
        Nz = len(z)-1
```

```
sz = (Nz,Ny,Nx)  # the shape of the model
Nod = np.prod(sz) # total number of cells in the model

# from this we have the width of columns, rows and layers
dx = np.diff(x).reshape(1, 1, Nx)
dy = np.diff(y).reshape(1, Ny,1)
dz = np.abs(np.diff(z)).reshape(Nz, 1,1)
```

IBOUND array - telling which cells are active and which have a prescribed head

Let's first specify which of the cells have their head prescrided and which cells are inactive. We have to tackle inactive cells early to make sure their conductance is made zero (in case there conductivities might be specified as non-zeros).

We do that my means of a so-called boundary array IBOUND (MODFLOW terminology), which is an integer array of the shape of the model grid that tells which cells have a prescribed head, which cells are inactive (i.e. which cells does not take part of the computation, such as cells that represent impermeable rock) and for which cells the head should be computed.

- IBOUND > 0, means heads will be computed
- IBOUND == 0, means cells are inactive
- IBOUND <0 , means heads prescribed

In this particular example we specify that the vertical zx plane at the last row of the model will have prescribed heads equal to zero.

```
In [7]: IBOUND = np.ones(sz)
        IBOUND[:,-1,:] = -1  # last row of model heads are prescribed
        IBOUND[:, 40:45, 20:70]=0 # these cells are inactive
```

This boundary array makes it easy telling which cells cells are active (head computed), inactive, and fixed-head.

```
In [8]: active = (IBOUND>0).reshape(Nod)  # active is now a vector of booleans of length Nod
        inact  = (IBOUND==0).reshape(Nod) # dito for inact
        fxhd   = (IBOUND<0).reshape(Nod)  # dito for fxhd
```

Cell conductancies: defining the ease of flow between adjacent cells

The first thing to define based on the properties of the cells is the flow resistance of each cell in the 3 grid directions, `x`, `y` and `z`. For that we need the cell sizes from the coordinates and the hydraulic conductivities in the `x`, `y` and `z` direction. The latter are given as full 3D arrays `kx`, `ky`, `kz` whose shapes correspond to that of the model mesh.

```
In [9]: k = 10.0 # m/d uniform conductivity
        kx = k * np.ones(sz) # [L/T] 3D kx array
        ky = k * np.ones(sz) # [L/T] 3D ky array with same values as kx
        kz = k * np.ones(sz) # [L/T] 3D kz array with same values as kx
```

The flow resistances for each cell is the head loss across opposite cell faces due to a unit flux through the cell along the axis perperndicular to them. These resistances are cell properties that can immediately be computed for the entire grid of the model. Because we always need the resistance between the cell center and its outer faces, we use the factor 0.5 (flow over half the lenght of the cell in each direction)

```
In [10]: # half cell flow resistances
         Rx = 0.5 * dx / (dy * dz) / kx  # [T/L2], flow resistance half cell in x-direction
         Ry = 0.5 * dy / (dz * dx) / ky  # same in y-direction
         Rz = 0.5 * dz / (dx * dy) / kz  # same in z-direction
```

Make inactive cells inactive by setting their resistance to np.Inf (infinite):

```
In [11]: Rx = Rx.reshape(Nod,); Rx[inact] = np.Inf; Rx=Rx.reshape(sz)
         Ry = Ry.reshape(Nod,); Ry[inact] = np.Inf; Ry=Ry.reshape(sz)
         Rz = Rz.reshape(Nod,); Rz[inact] = np.Inf; Rz=Rz.reshape(sz)
```

From this we compute the conductance between each pair of adjacent cells across their connecting cell face. The conductance is just the reverse of the resistance of the two connected half cells. This resistance is the sum of the resistances of the two connected half cells because these resistances are placed in series with respect to the flow.

```
In [12]: # conductances between adjacent cells
         Cx = 1 / (Rx[:, :, :-1] + Rx[:, :,1:]) # [L2/T] in x-direction
         Cy = 1 / (Ry[:, :-1,:] + Ry[:, 1:,:]) # idem in y-direction
         Cz = 1 / (Rz[:-1,:,:] + Rz[1:,:,:]) # idem in z-direction
```

Setting up the system matrix - set of water balance equations

The system matrix has size of (`Nod`, `Nod`) allowing a connection between each pair of cells. Of course only cells that share their cell face are connected in reality. In a 3D model this means that each cell is connected to its 6 neighbors instead of to all other cells in the model. This means that most of the matrix entries will be zero.

To be able to indentify adjacent cells we generate cell numbers in an array that has the size of the model grid:

```
In [14]: NOD = np.arange(Nod).reshape(sz) # this is a full 3D array of node numbers (cell numbers
```

With this array it's easy to identify adjacent cells by their cell number. Thus we generate arrays with the cel numbers of right hand neigbor of the cells (east neighbor), the left hand neighbor (the west neigbor), the north neighbor, south neighbor, the top neighbor and the bottom neighbor as follows

```
In [15]: IE = NOD[:, :, 1:] # numbers of the eastern neighbors of each cell
         IW = NOD[:, :, :-1] # same western neighbors
         IN = NOD[:, :-1,:] # same northern neighbors
         IS = NOD[:, 1:,:] # southern neighbors
         IT = NOD[:-1,:,:] # top neighbors
         IB = NOD[1:,:,:] # bottom neighbors
```

Notice that the shape of the `IE` and `IW` is the same as that of `Cx`, the size of `IN` and `IS` is the same as that of `Cy` and the size of `IT` and `IB` is the same as that of `Cx`.

To put the conductances into the system matrix we need their row and column indices together with their value, so that we can say `a[j,i] = value`. Because we have the numbers of adjacent cells in the arrays `IE`, `IW` etc, we can immediately place all the system matrix coefficiencts at the place into a sparse matrix.

```
In [16]: import scipy.sparse as sp

         R = lambda x : x.ravel() # define short hand for x.ravel()

         # notice the call signature:
         #      csc_matrix( (data, (row_index, col_index) ), (M,N)); This is a tuple within tuple
         A = sp.csc_matrix(( np.concatenate(( R(Cx), R(Cx), R(Cy), R(Cy), R(Cz), R(Cz)) ),\
                             (np.concatenate(( R(IE), R(IW), R(IN), R(IS), R(IB), R(IT)) ),\
                              np.concatenate(( R(IW), R(IE), R(IS), R(IN), R(IT), R(IB)) ),\
                           )),(Nod,Nod))
```

We now have to define the diagonal elements of the system matrix `A`, i.e. the values `a[i,i]` for `i`=[0:`Nod`].

These are just the negative sum of the row coefficients. Hence we sum `A` over the second axis (`axis`=1) to get them in a [`Nod`,1] sized vector. (Notice stat sparace matrix derived vectors keep their orientation, contrary to vectors obained from numpy arrays, which produce dimensionless vectors).

Generate the diagonal values:

```
In [17]: # to use the vector of diagonal values int a call of sp.diags() we need to have it aa a
         # standard nondimensional numpy vector.
         # To get this:
         # - first turn the matrix obtained by A.sum(axis=1) into a np.array by np.array( .. )
```

```
        # - then take the whole column to loose the array orientation (to get a dimensionless nu
        adiag = np.array(-A.sum(axis=1))[:,0]
```

Then generate a diagonal array from these values, that we can add it to `A`, to complete `A`.

```
In [18]: Adiag = sp.diags(adiag)
```

More complex alternative: Generate diagonal array by calling the csr_matrix constructor:

Adiag = sp.csr_matrix((adiag,(np.arange(Nod),np.arange(Nod))),(Nod,Nod))

Boundary conditions

For this chapter we only use fixed flow and fixed head boundary conditions.

Fixed flows

Fixed flow boundary conditions are specified by an 3D array of the size of the grid. Each values specifies the inflow for the corresponding cell (injections are positive). Cells without a specified flow are, in fact, cells where the specified flow is zero. Hence the fixed-flows array is a full 3D array with flow values that are zero where no flow enters or leaves the cells and have non-zero values elsewhere.

For this example, we specify a single extraction of Q=-1200 m3/d in cell [30,25,2]:

```
In [19]: FQ = np.zeros(sz)     # all flows zero. Note sz is the shape of the model grid
         FQ[2, 30, 25] = -1200  # [m3/d] extraction in this cell
```

The righ-hand size of the matrix equation to be solved, the vector RHS, contains the flows. So we can generate it by assignment of `FQ` and converting it to a numpy vector

```
In [20]: RHS = FQ.reshape(Nod)
```

See further down how we use `RHS` for only the active and non-fixed head rows.

The next step is to add fixed head boundary conditions.

Fixed heads

Fixed heads are known heads. This implies that in the set of equations that represent the model, i.e

$A \times Phi = RHS$

Some of the Phis are prescribed and should not be computed as defined by `IBOUND` and contained in the boolean vectors `active`, `fxhd` and `inact` specified and computed above.

Now that we know which cell have fixed heads, we can multiply out these heads with the corresponding columns of the system matrix, which yields a vector of constant values with dimension flow [m3/d] that can be added to the fixed flow vector in the `RHS` vector. The `RHS` vector is now the sum of the `FQ` and the contribution from the fixed heads.

Notice that the fixed heads will be obtained from the given array `HI` of the initial heads, where the head in the cells where `IBOUND`>0 correspond with the fixed heads.

```
In [21]: HI = np.zeros(sz)
```

We reshape `FQ` and `HI` to a column vector to allow matrix multiplication

```
In [22]: RHS = FQ.reshape(Nod,1) - A[:,fxhd].dot(HI.reshape(Nod,1)[fxhd])
```

We have now the complete `RHS` of the matrix equation to solve:

$A \times Phi = RHS$

Solving the matrix equation for the unknown heads

We use the sparse matrix solver in module `scipy.sparse.linalg` to compute the unknown heads.

```
In [23]: from scipy.sparse.linalg import spsolve # import with from to use its short name
```

Of course we only need the active rows and columns of `A` and the active rows from `RHS`.

But first allocate a full-fledged vector of heads to store the result.

```
In [24]: Phi = HI.flatten()
```

Then compute the unknown heads (i.e. the active cels only).

Remark: If we want to select a submatrix from `A` defiend by a given vectors of row and column indices, we can do so in sequence: Rows (I) first, columns (J) next, like so:

$A[I][:, J]$

which we apply in the next line

```
In [25]:  Phi[active] = spsolve( (A+Adiag)[active][:,active] ,RHS[active] )
```

At this point we solved the problem and now have the heads for all cells in the vector Phi.

We didn't touch the rows and columns that are inactive. So the heads of these inactive cells whatever they are in `HI` are know still in `Phi`. Just to make sure we detect them and won't use them, set them to `NaN` (Not a Number).

```
In [26]: Phi[inact] = np.NaN
```

Finally we reshape the head vector to that of the model grid.

```
In [27]: Phi=Phi.reshape(sz) # reshape vector Phi to 3D shape of the grid
```

```
In [28]: Phi # show Phi
```

```
Out[28]: array([[[ 1.77107246,  1.77132861,  1.77183761, ...,  1.56399507,
                   1.56320139,  1.56280385],
                 [ 1.77081631,  1.77107575,  1.77159133, ...,  1.56360172,
                   1.56280525,  1.56240631],
                 [ 1.77030071,  1.77056677,  1.77109553, ...,  1.56281367,
                   1.56201158,  1.56160982],
                 ...,
                 [ 0.05529699,  0.05524583,  0.0551436 , ...,  0.03419641,
                   0.03426788,  0.03430386],
                 [ 0.02763571,  0.02761016,  0.02755909, ...,  0.01708962,
                   0.01712512,  0.01714299],
                 [ 0.        ,  0.        ,  0.        , ...,  0.        ,
                   0.        ,  0.        ]],

                [[ 1.77107246,  1.77132861,  1.77183761, ...,  1.56399507,
                   1.56320139,  1.56280385],
                 [ 1.77081631,  1.77107575,  1.77159133, ...,  1.56360172,
                   1.56280525,  1.56240631],
                 [ 1.77030071,  1.77056677,  1.77109553, ...,  1.56281367,
                   1.56201158,  1.56160982],
                 ...,
                 [ 0.05529699,  0.05524583,  0.0551436 , ...,  0.03419641,
                   0.03426788,  0.03430386],
                 [ 0.02763571,  0.02761016,  0.02755909, ...,  0.01708962,
                   0.01712512,  0.01714299],
                 [ 0.        ,  0.        ,  0.        , ...,  0.        ,
                   0.        ,  0.        ]],

                [[ 1.77107246,  1.77132861,  1.77183761, ...,  1.56399507,
                   1.56320139,  1.56280385],
                 [ 1.77081631,  1.77107575,  1.77159133, ...,  1.56360172,
                   1.56280525,  1.56240631],
                 [ 1.77030071,  1.77056677,  1.77109553, ...,  1.56281367,
```

```
       1.56201158,  1.56160982],
     ...,
     [ 0.05529699,  0.05524583,  0.0551436 , ...,  0.03419641,
       0.03426788,  0.03430386],
     [ 0.02763571,  0.02761016,  0.02755909, ...,  0.01708962,
       0.01712512,  0.01714299],
     [ 0.        ,  0.        ,  0.        , ...,  0.        ,
       0.        ,  0.        ]],

    [[ 1.77107246,  1.77132861,  1.77183761, ...,  1.56399507,
       1.56320139,  1.56280385],
     [ 1.77081631,  1.77107575,  1.77159133, ...,  1.56360172,
       1.56280525,  1.56240631],
     [ 1.77030071,  1.77056677,  1.77109553, ...,  1.56281367,
       1.56201158,  1.56160982],
     ...,
     [ 0.05529699,  0.05524583,  0.0551436 , ...,  0.03419641,
       0.03426788,  0.03430386],
     [ 0.02763571,  0.02761016,  0.02755909, ...,  0.01708962,
       0.01712512,  0.01714299],
     [ 0.        ,  0.        ,  0.        , ...,  0.        ,
       0.        ,  0.        ]]])
```

Plotting the heads as contours

Import the required plotting module and setup the plot.

```
In [29]: %matplotlib notebook
         import matplotlib.pyplot as plt # combines namespace of numpy and pyplot
```

For coordinates of the cells use their centers.

```
In [30]: xm = 0.5 * (x[:-1] + x[1:]) # [L] coordinates of column centers
         ym = 0.5 * (y[:-1] + y[1:]) # [L] coordinates of row centers
         layer = 2 # contours for this layer
         nc = 50   # number of contours in total
```

Plot the results using plt functions like in Matlab

```
In [31]: plt.xlabel('x [m]')
         plt.ylabel('y [m]')
         plt.title("Contours (%d in total) of the head in layer %d with inactive section" % (nc,
         plt.contour(xm, ym, Phi[layer], nc)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Out[31]: <matplotlib.contour.QuadContourSet at 0x10e3410f0>
```

- The white area is the aquifer part that was defined as inactive (impervious).
- The red trough is the well location.
- The two flanks and the front sides are closed (no FQ and no fixed head).
- The head at the back side is prescribed and maintained at zero.

Conclusion

In this chapter we have developed, from scratch, a full 3D finite difference model for which fixed heads, fixed flows and inactive subareas are be prescribed. While developing we also computed a concrete example of which the head contours were finally shown.

To turn this model into a general Python function that can compute arbitrary 3D steady-state models of this kind, we only have to gather the developed lines and put them under a function definition, add some help strings (a doc string) and add error checking for convenience, while the data for this specific case, like the grid dimensions (`x`,`y`,`z`) the conductivities (`kx`,`ky`,`kz`), the prescribed flows `FQ` and initial heads `HI` plus the boundary array `IBOUND` that tells which cells have fixed heads and which are inactive, are to be passed as in user-given arguments at the function call like so:

Phi = fdm3(x,y,z,kx,ky,kz,FQ,IH,IBOUND) # function that solves arbitrary 3D steady steate finite difference model

We do this in the next section.

CHAPTER 4

A finite difference model as a Python function

Prof. dr.ir.T.N.Olsthoorn

Heemstede, Sept. 2016, May 2017

Generalize the finite difference model into a callable function

The 3D finite difference computation in the previous chapter can be generalized by putting the generic parts in a callable Python function and turning the the user or case specific data into the arguments by which the function is called like so:

Phi = fdm3(x, y, z, kx, ky, kz, FQ, IH, IBOUND)

Doing so and adding some input error checking for convenience leads to the following code, that is save to disk for future use:

```
In [1]: %%writefile fdm_a.py
        # write the function in this cell to disk as file fdm.py

        import numpy as np
        #import pdb  # in case we need to debug this function

        def fdm3(x, y, z, kx, ky, kz, FQ, HI, IBOUND):
            '''Returns computed heads of steady state 3D finite difference grid.

            Steady state 3D Finite Difference Model that computes the heads a 3D ndarray.

            Parameters
            ----------
            `x` : ndarray, shape: Nx+1, [L]
                `x` coordinates of grid lines perpendicular to rows, len is Nx+1
            `y` : ndarray, shape: Ny+1, [L]
                `y` coordinates of grid lines along perpendicular to columns, len is Ny+1
            `z` : ndarray, shape: Nz+1, [L]
                `z` coordinates of layers tops and bottoms, len = Nz+1
            `kx`, `ky`, `kz` : ndarray, shape: (Ny, Nx, Nz) [L/T]
                hydraulic conductivities along the three axes, 3D arrays.
            `FQ` : ndarray, shape: (Ny, Nx, Nz), [L3/T]
                prescrived cell flows (injection positive, zero of no inflow/outflow)
            `IH` : ndarray, shape: (Ny, Nx, Nz), [L]
```

```
        initial heads. `IH` has the prescribed heads for the cells with prescribed head.
    `IBOUND` : ndarray of int, shape: (Ny, Nx, Nz), dim: [-]
        boundary array like in MODFLOW with values denoting
        * IBOUND>0  the head in the corresponding cells will be computed
        * IBOUND=0  cells are inactive, will be given value NaN
        * IBOUND<0  coresponding cells have prescribed head

    Returns
    -------
    `Phi` : ndarray, shape: (Ny, Nx, Nz), [L]
        the 3D array with the final heads with `NaN` at inactive cells.

    TO 160905
    '''

    import numpy as np
    import scipy.sparse as sp
    from scipy.sparse.linalg import spsolve # to use its short name
#    pdb.set_trace()
    x = np.sort(np.array(x))       # enforce ascending
    y = np.sort(np.array(y))[::-1] # enforce descending
    z = np.sort(np.array(z))[::-1] # enforce descending

    # as well as the number of cells along the three axes
    SHP = Nz, Ny, Nx = len(z) - 1, len(y)-1, len(x)-1

    Nod = np.prod(SHP)

    if Nod == 0:
        raise AssertationError("Nx, Ny and Nz must be >= 1")

    # assert correct shape of input arrays
    if kx.shape != SHP:
        raise AssertionError("shape of kx {0} differs from that of model {1}".format(kx.sh
    if ky.shape != SHP:
        raise AssertionError("shape of ky {0} differs from that of model {1}".format(ky.sh
    if kz.shape != SHP:
        raise AssertionError("shape of kz {0} differs from that of model {1}".format(kz.sh

    # from this we have the width of columns, rows and layers
    dx = np.abs(np.diff(x).reshape(1, 1, Nx)) # enforce positive
    dy = np.abs(np.diff(y).reshape(1, Ny, 1)) # enforce positive
    dz = np.abs(np.diff(z)).reshape(Nz, 1, 1) # enforce positive

    active = (IBOUND >0).reshape(Nod,) # boolean vector denoting the active cells
    inact  = (IBOUND==0).reshape(Nod,) # boolean vector denoting inacive cells
    fxhd   = (IBOUND <0).reshape(Nod,) # boolean vector denoting fixed-head cells

    # half cell flow resistances
    Rx = 0.5 * dx / (dy * dz) / kx
    Ry = 0.5 * dy / (dz * dx) / ky
    Rz = 0.5 * dz / (dx * dy) / kz

    # set flow resistance in inactive cells to infinite
    Rx = Rx.reshape(Nod,); Rx[inact] = np.Inf; Rx=Rx.reshape(SHP)
    Ry = Ry.reshape(Nod,); Ry[inact] = np.Inf; Ry=Ry.reshape(SHP)
    Rz = Rz.reshape(Nod,); Rz[inact] = np.Inf; Rz=Rz.reshape(SHP)

    # conductances between adjacent cells
    Cx = 1 / (Rx[:, :, :-1] + Rx[:, :, 1:])
    Cy = 1 / (Ry[:, :-1, :] + Ry[:, 1:, :])
    Cz = 1 / (Rz[:-1, :, :] + Rz[1:, :, :])
```

```
    NOD = np.arange(Nod).reshape(SHP)

    IE = NOD[:, :, 1:]  # east neighbor cell numbers
    IW = NOD[:, :, :-1] # west neighbor cell numbers
    IN = NOD[:, :-1, :] # north neighbor cell numbers
    IS = NOD[:, 1:, :]  # south neighbor cell numbers
    IT = NOD[:-1, :, :] # top neighbor cell numbers
    IB = NOD[1:, :, :]  # bottom neighbor cell numbers

    R = lambda x : x.ravel()  # generate anonymous function R(x) as shorthand for x.ravel

    # notice the call  csc_matrix( (data, (rowind, coind) ), (M,N))  tuple within tupple
    A = sp.csc_matrix(( -np.concatenate(( R(Cx), R(Cx), R(Cy), R(Cy), R(Cz), R(Cz)) ),\
                         (np.concatenate(( R(IE), R(IW), R(IN), R(IS), R(IB), R(IT)) ),\
                          np.concatenate(( R(IW), R(IE), R(IS), R(IN), R(IT), R(IB)) ),\
                       )),(Nod,Nod))

    # to use the vector of diagonal values in a call of sp.diags() we need to have it aa a
    # standard nondimensional numpy vector.
    # To get this:
    # - first turn the matrix obtained by A.sum(axis=1) into a np.array by np.array( .. )
    # - then take the whole column to loose the array orientation (to get a dimensionless
    adiag = np.array(-A.sum(axis=1))[:,0]

    Adiag = sp.diags(adiag)  # diagonal matrix with a[i,i]

    RHS = FQ.reshape(Nod,1) - A[:,fxhd].dot(HI.reshape(Nod,1)[fxhd]) # Right-hand side vec

    Phi = HI.flatten() # allocate space to store heads

    Phi[active] = spsolve( (A + Adiag)[active][:,active] ,RHS[active] ) # solve heads at a

    Phi[inact] = np.NaN # put NaN at inactive locations

    return Phi.reshape(SHP) # reshape vector to 3D size of original model
Overwriting fdm_a.py
```

```
In [2]: import numpy as np # we always need this
        import fdm_a # import fdm_a module for use

        from importlib import reload  # we need reload if we edited the file fdm_a.py
        reload(fdm_a)  # if edited, must reload it

        fdm3 = fdm_a.fdm3  # for convenience create a local name to just write fdm3
```

This function should be saved in a .py file e.g. “fdm3.py” so that it can be used as a module that can be imported by other users or programs or scripts.

Apply the model

To apply the model we specify coofdinates, conductivities, prescribed flows and prescribed heads and the `IBOUND` array. Then we call the function with the proper arguments. After the function computed the heads, we’ll show them by a contour plot.

Generate input to run the model with

The lines below are case specific and could be placed in a script that can be run to set up the model afterw which the model is called from the script with the proper arguments

```
In [3]: # specify a rectangular grid
        x = np.arange(-1000., 1000., 25.)
        y = np.arange(-1000., 1000., 25.) # backward, i.e. first row grid line has highest y
        z = np.arange(-100., 0., 20.)  # backward, i.e. from top to bottom

        SHP = (len(z)-1, len(y)-1, len(x)-1)

        k = 10.0 # m/d uniform conductivity
        kx = k * np.ones(SHP) # [L/T] 3D kx array
        ky = k * np.ones(SHP) # [L/T] 3D ky array with same values as kx
        kz = k * np.ones(SHP) # [L/T] 3D kz array with same values as kx

        FQ = np.zeros(SHP)     # all flows zero. Note sz is the shape of the model grid
        FQ[2, 30, 25] = -1200  # [m3/d] extraction in this cell

        HI = np.zeros(SHP) # initial heads

        IBOUND = np.ones(SHP)
        IBOUND[:, -1, :] = -1  # last row of model heads are prescribed
        IBOUND[:, 40:45, 20:70]=0 # these cells are inactive
```

Call the function with the correct arguments

```
In [4]: Phi = fdm3( x, y, z, kx, ky, kz, FQ, HI, IBOUND)
```

Visualization of the results: plot heads as contours

Import the required plotting module and setup the plot and run `%matplotlib notebook` to allow plots beeing shown inside the notebook.

```
In [5]: %matplotlib notebook
        import matplotlib.pyplot as plt # combines namespace of numpy and pyplot
```

```
In [6]: xm = 0.5 * (x[:-1] + x[1:]) # cell center coordinates
        ym = 0.5 * (y[:-1] + y[1:]) # same
        layer = 2 # contours for this layer
        nc = 50   # number of contours in total

        plt.xlabel('x [m]')
        plt.ylabel('y [m]')
        plt.title("Contours (%d in total) of the head in layer %d with inactive section" % (nc, l
        plt.contour(xm, ym, Phi[layer], nc)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Out[6]: <matplotlib.contour.QuadContourSet at 0x10e619240>
```

- The white area is the aquifer part that was defined as inactive (impervious).
- The red trough is the well location.
- The two flanks and the front sides are closed (no FQ and no fixed head).
- The head at the back side is prescribed and maintained at zero.

Conclusion

We have developed from scratch a full 3D steady-state finite difference mode for which fixed heads, fixed flows and inactive subareas can be prescribed.

This model is general, it can be called as a function and allows solving a wide range of different 3D steady-state groundwater flow problems in a finite difference setting.

The function should be stored in a module (a .py file) and can then be imported from this module, like so

import mymodule

from mymodule import fdm3

Examples

Here we set-up a few simple examples that we can readily verify analytically. The examples are 2D and can easily be computed with our 3D model.

In [7]:
```
# basic imports

import numpy as np
%matplotlib notebook
import matplotlib.pyplot as plt # combines namespace of numpy and pyplot

import fdm_a
fdm3 = fdm_a.fdm3    # could also use from fdm_a import fdm3 (sets fdm3 as convenient loca
```

Example 1: flow between 2 fixed bounaries

Consider the flow in an aquifer with constant transmissivity `kD=1000` m2/d, with fixed head `h=0` at the left and the right boundaries at `x=-L/2` and `x=+L/2` respectively, subject to uniform precipitation `N`.

The analytical solution for this case is

$h = **:raw-latex:'\frac 'N {2 kD} ((:raw-latex:'**frac *L 2) :sup:'2-x2*) $

The model will consist of 1 layer, is `L=500` m wide, with cells of `dx=10` m and is subject to `N=0.01` m/d recharge.

In order to put the fixed heads at the outer cell centers exactly at $\pm -L/2$, we make the width of the outer cells very small, 0.001 m say.

In [8]:
```
# constants
N = 0.01 # m/d, precipitaion suplus
L = 500  # m,   width of the cross section
k = 10   # m/d, aquifer conductivity
D = 100  # m/d, aquifer thickness
L = 1000 # m,   width of the cross section
kD = k * D # m2/d, aquifer transmissivity
```

In [9]:
```
# model grid
x = np.hstack((-L/2-0.001, np.linspace(-L/2, L/2, 51), L/2+0.001))
y = np.array([-0.5, 0.5]) # make the model 1m wide
z = np.array([-D, 0])    # use thickenss of model

Nz, Ny, Nx = SHP = (len(z) -1, len(y)-1, len(x)-1)

# cell center coordinates
xm = .5 * (x[:-1] + x[1:])
ym = .5 * (y[:-1] + y[1:])
zm = .5 * (z[:-1] + z[1:])

# cell size
dx = np.abs(np.diff(x));
dy = np.abs(np.diff(y));
dz = np.abs(np.diff(z));
```

```
In [10]: # boundary array
         IBOUND = np.ones(SHP)
         IBOUND[:, :, [0, -1]] = -1  # fix head of left and right boundary

         # conductivitys array
         K = k  * np.ones(SHP)

         # prescribed flow = precipitation surplus
         FQ = np.ones(SHP)
         FQ[0, :, :] = dx * N

         # Initial heads, all zero
         HI = np.zeros(SHP)

In [11]: # run the model to compute the heads
         Phi = fdm3(x, y, z, K, K, K, FQ, HI, IBOUND)
```

Vizualisation of the computed heads:

```
In [12]: %matplotlib notebook
         import matplotlib.pyplot as plt # combines namespace of numpy and pyplot

         plt.figure()
         plt.xlabel('x [m]')
         plt.ylabel('head [m]')
         plt.title('Head between two fixed boundaries and constant precipitation {0} m/d'.format(N
         plt.plot(xm, Phi.ravel(), 'bo-', label='numeric')

         # now add the analytical solution
         fi = N/(2*kD) * ((L/2)**2 - x**2)
         plt.plot(x, fi, 'xr', label='analytic')
         plt.legend()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Out[12]: <matplotlib.legend.Legend at 0x1073e4c50>
```

This shows that the analytical and numerical solutions match. Notice, however, that we have made the outer two cells very narrow, so that the location of the `x[0]` and `xm[0]` and `x[-1]` and `xm[-1]` are almost indistinguishable, so that the position of the boundary are essentially the same for the numerical and the analytical model. (Just make the outer two cells as wide as the other cells to see the difference.

The 3D model was used here as a 1D model along the `x`-axis. It should be evident that our 3D numerical model is very flexible in modeling 1D, 2D as well as 3D situations.

Example 2, semi-confined flow (mazure case)

Assume the case where we have a water body with constant head $h = h_w$ that at `x=0` is in direct contact with an semi aquifer that is charaterized with constant transmissivity `kD` convered by a semi-confining layer with constant resistance `c` above which we have a head that is maintained at a constant level $h = h_p$. This problem was solved by Marzure around 1930 when he studied groundwater flow from the IJssel Lake to a new adjacent polder with a 5 m lower maintained water level.

The analytical solution for the head in the regional aquifer, that is, below the confining layer is

$ **:raw-latex:‘\phi ‘- h_p = (:raw-latex:‘**phi‘_0 - h_p) „ **:raw-latex:‘\exp ‘(-x / :raw-latex:‘**lambda‘) $

with $\lambda = \sqrt{kDc}$

Let’s work out this case to compare the analytical result with our model.

```
In [13]: # import numpy as np, assumed numpy has been loaded above
```

```
k = 25 # m/d conductivity of regional aquifer
D = 50 # m   thickness of regional aquifer
d = 10 # m   thickness of confining layer
dtop = 0.001 # m dummy thickness of layer in which the head is maintained
c = 500.0 # d  vertical hydraulic resistance of confining layer
hw = -0.4   # m fixed IJssel lake elevation
hp = -5.0 # m fixed polder water elevation.

kD = k*D
lam = np.sqrt(kD * c)
```

In [14]:
```
x = np.hstack((-0.001, np.linspace(0, 5000, 100)))
y = np.array([-0.5, 0.5])
grs = 0 # m,  ground surface elevation
z = np.array([grs-D-d-dtop, grs-d-dtop, grs-dtop, grs])
```

In [16]:
```
Nz, Ny, Nx = SHP = len(z)-1, len(y)-1, len(x)-1
IBOUND = np.ones(SHP)
IBOUND[0, :, :] = -1 # fixed head maintained in top layer (above confining layer)
IBOUND[-1, :, 0] = -1 # fixed head in aquifer at x=0

HI = np.zeros(SHP)
HI[-1, :, 0] = hw # fixed level of IJssel Lake
HI[ 0, :, :] = hp # fixed polder level

FQ = np.zeros(SHP) # no fixed flows

kAquif = k;
kConf  = d/c;
kTop   = 100;  #immaterial

K = np.array([kTop, kConf, kAquif]).reshape(Nz, 1, 1) * np.ones(SHP)
```

In [17]:
```
Phi = fdm3(x, y, z, K, K, K, FQ, HI, IBOUND)
```

In [18]:
```
plt.figure()
plt.xlabel('x [m]')
plt.ylabel('head [m]')
plt.title('Head between two fixed boundaries and constant precipitation {0} m/d'.format(N

xm = 0.5 * (x[:-1] + x[1:])
plt.plot(xm, Phi[-1][0], 'bo-')

# now add the analytical solution
fi = hp + (hw - hp) * np.exp(-x/lam)
plt.plot(x, fi, 'xr-')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Out[18]:
```
[<matplotlib.lines.Line2D at 0x10ef718d0>]
```

As can be seen, the analytical and the numeical model match. Notice that here we made the first cell width very small, so that the fixed head in the first cell falls almost exactly at zero. Had we not done so, then the fixed head of the numerical model would have been at `xm[0] = 50` i.e. in the center of the first cell, while the fixed head in the analytical solution would still be at exactly `x = 0`, which would lead to a difference between the two solutions. Just try this by setting the width of the first model cell equal to 100 m instead of 0.001 m.

Another issue is that the numerical model is closed at the right-hand side, i.e. where `x = 5000 m`. In this example, this distance is so large that the effect of the left boundary is almost zero, so that the difference between the analytical solution and the numerical model is not visibile in the graph. Just try to make the right-hand side of the model less far to see the difference.

Of course one could easily fix the right-hand boundary as well as the left-hand one, to obtain a solution that is bounded by fixed heads on either side. It's easy to derive the analytical solution for that case as well and to verify

it with the numerical model. This is left as an exercise to the student.

The analytical solution for the the same head at either side of the cross section is this

$:raw-latex:*phi ‘- h_p = (h_w - h_p) :raw-latex:*frac {cosh (frac x lambda) } {cosh (frac {L,/2} lambda)}‘ $

Also notice that we have now a 3 layer model, one row, a large number of columns and 3 layers. We simply could make the model full 3D by taking more than one row. It should also be clear that the properties of the layers can be varies arbitrarily on a cell by cell basis.

Example 3, same case, but using only two layers

Instead of 3 layers, we may use only two and fix the head in the semi confining layer. Because of the derivation of our the finite difference method water flows between cell centers. This means that, when we use the semi confing layer to also prescribe the head, this head is prescribed at the center of the layer, and the water flows vertically from or to the center of that layer from the bottom of that layer, so that only half its thickness is active for vertical flow. For the model layout, it implies that we should put the center of the layer at the elevation of ground surface and only take the vertical flow through half the layer into account. This is worked out below.

```
In [19]: # aquifer
         k = 25 # m/d conductivity of regional aquifer
         D = 50 # m   thickness of regional aquifer
         d = 20 # m   thickness of confining layer
         c = 500.0 # d  vertical hydraulic resistance of confining layer
         hw = -0.4   # m fixed IJssel lake elevation
         hp = -5.0 # m fixed polder water elevation.

         kD = k*D
         lam = np.sqrt(kD * c)

In [20]: x = np.hstack((-0.001, np.linspace(0, 5000, 100)))
         y = np.array([-0.5, 0.5])
         grs = 0 # m,  ground surface elevation
         z = np.array([grs-D-d, grs-d, grs+d])

In [21]: Nz, Ny, Nx = SHP = len(z)-1, len(y)-1, len(x)-1
         IBOUND = np.ones(SHP)
         IBOUND[0, :, :]  = -1 # fixed head maintained in top layer (above confining layer)
         IBOUND[-1, :, 0] = -1 # fiex head in aquifer at x=0

         HI = np.zeros(SHP)
         HI[-1, :, 0] = hw # fixed level of IJssel Lake
         HI[ 0, :, :] = hp # fixed polder level

         FQ = np.zeros(SHP) # no fixed flows

         kAquif = k;
         kConf  = d/c;
         kTop   = 100;  #immaterial

         K = np.array([kConf, kAquif]).reshape(Nz, 1, 1) * np.ones(SHP)

In [22]: Phi = fdm3(x, y, z, K, K, K, FQ, HI, IBOUND)

In [24]: # %matplotlib notebook  # already done above
         # import matplotlib.pyplot as plt # combines namespace of numpy and pyplot

         plt.figure()
         plt.xlabel('x [m]')
         plt.ylabel('head [m]')
         plt.title('Head between two fixed boundaries and constant precipitation {0} m/d'.format(

         xm = 0.5 * (x[:-1] + x[1:])
         plt.plot(xm, Phi[-1][0], 'bo-')
```

```
        # now add the analytical solution
        fi = hp + (hw - hp) * np.exp(-x/lam)
        plt.plot(x, fi, 'xr-')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Out[24]: [<matplotlib.lines.Line2D at 0x10f501588>]
```

As can be seen, we get same result as before, but now with only two layers, of which the top one represents the semi confined layer in which center the head is prescribed.

Cicular island with recharge

This examples uses a flat model (one layer) and places a fixed head outside the boundary of the island.

```
In [47]: x = np.linspace(-1000., +1000., 200)
         y = np.linspace(-1000., +1000., 200)
         z = np.array([0., -100.])

         xm = 0.5 * (x[:-1] + x[1:])
         ym = 0.5 * (y[:-1] + y[1:])
         zm = 0.5 * (z[:-1] + z[1:])

         Nx = len(xm)
         Ny = len(ym)
         Nz = len(zm)

         SHP = (Nz, Ny, Nx)

         ZM, YM, XM = np.meshgrid(zm, ym, xm, indexing='ij') # full 3D arrays of cell center coor

         DX = np.abs(np.diff(x).reshape((1, 1, Nx)) * np.ones(SHP))  # column width (3D aray)
         DY = np.abs(np.diff(y).reshape((1, Ny, 1)) * np.ones(SHP))  # row widths (3D array)

         x0 = 0.; y0 = 0. # center of the island

         RM = np.sqrt((XM - x0)**2 + (YM - y0)**2).reshape(SHP) # distance to center

         R = 750.0 # [m] radius of the island

         IBOUND = np.ones(SHP)
         IBOUND[RM>R] = -1

         k0 = 10 # [m/d]
         k  = k0 * np.ones(SHP) # uniform conductivity

         kD = float(k0 * np.abs(np.diff(z)))

         rch = 0.01 # [m/d] recharge rate
         FQ  = rch * DX * DY # [m3/d] cell inflows
         IH  = np.zeros(SHP) # [m] initial heads

         Phi = fdm3(x, y, z, k, k, k, FQ, IH, IBOUND) # run model, return heads

         # plot the heads a contours
         plt.figure()
         plt.xlabel('x [m]'); plt.ylabel('y [m]')
         plt.title('Circular island with radius R={0} m and recharge N = {1} m/d'.format(R,N))
         plt.contourf(xm,ym,Phi[0],50)
```

```
        # plot the heads along the cross section and compare with analytical solution
        plt.figure()
        plt.xlabel('x [m]'); plt.ylabel('y [m]')
        plt.title('Heads through the center of the island')
        centerRow = int(np.floor(Ny/2))
        plt.plot(xm, Phi[0, centerRow, :], label='Numeric')
        Island = np.logical_and(xm >= -R, xm <= R)
        plt.plot(xm[Island], N/(4 * kD) * (R**2 - xm[Island] ** 2), label='Analytic')
        plt.legend()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Out[47]: <matplotlib.legend.Legend at 0x10f52fa20>
```

Notice the difference between the analytic and numeric solution. This difference depends on the size of the number of cells that are used. The finer the model, the better the agreement. A small agreement remains because the boundary of the numeric model is not perfectly circular. We will a develop and use an axisymmetric model in the next chapter.

Circular polder

This circular polder is also a circular island with fixed boundary at distance R from its center, but instead of a uniform recharge we have leakage through a semi-confining top layer that we will simulate using an extra layer.

Within the island we have a fixed head above the confining top layer. As an extra example we can choose to extend the polder to not be surrounded by open water with a fixed head but by other land where the head is fixed above the confing layer but at a different value.

Much of the set-up below is copied from the previous example.

```
In [46]: import scipy  # we need bessel functions

         I0 = scipy.special.i0 # bessel function
         I1 = scipy.special.i1 # same
         K0 = scipy.special.k0 # same
         K1 = scipy.special.k1 # same

         z0=   0. # m, reference elevation, ground surface
         d =  10. # m, thickness of confining layer
         D = 100. # m, thickness of regional aquifer

         R  = 750. # m, radius of island
         x0 = 0.; y0 = 0. # center of the island

         hp = -2.0  # m, maintained head r<=R
         hl =  0.50 # m, maintained head r>R

         c  = 250. # d, vertical hydraulic resistance of confining layer
         k0 =  d/c # m/d vertical conductivity of confining layer
         k1 = 10. # m/d, conductivity of regional layer
         kD = k1 * D # m2/d transmissivity of regional aquifer
         lam = np.sqrt(kD * c) # m, spreading or characteristic length of aquifer system

         # The grid, coordinates and size
         x = np.linspace(-2000., +2000., 200)
         y = np.linspace(-2000., +2000., 200)
         z = z0 - np.array([0, d, d+D])
```

```
xm = 0.5 * (x[:-1] + x[1:])
ym = 0.5 * (y[:-1] + y[1:])
zm = 0.5 * (z[:-1] + z[1:])

Nx = len(xm)
Ny = len(ym)
Nz = len(zm)

SHP = (Nz, Ny, Nx)

ZM, YM, XM = np.meshgrid(zm, ym, xm, indexing='ij') # full 3D arrays of cell center coor

DX = np.abs(np.diff(x).reshape((1, 1, Nx)) * np.ones(SHP))  # column width (3D aray)
DY = np.abs(np.diff(y).reshape((1, Ny, 1)) * np.ones(SHP))  # row widths (3D array)

RM = np.sqrt((XM - x0)**2 + (YM - y0)**2).reshape(SHP) # distance to center
in_polder = RM[0, :, :]<=R
in_land   = np.logical_not(in_polder)

IBOUND = np.ones(SHP)
IBOUND[0, :, :] = -1  # maintain head everywhere in confining layer

k = np.ones(SHP) # uniform conductivity
k[ 0, :, :] = k0/2  # m/d, top layer
k[-1, :, :] = k1  # m/d, bottom layer

rch = 0.01 # [m/d] recharge rate
FQ = np.zeros(SHP)# [m3/d] cell inflows

IH = np.zeros(SHP) # [m] initial heads
H  = np.ones((Ny, Nx)) # m, maintained head in top layer
H[in_land]  = hl # outside R
H[in_polder]= hp # inside R

IH[0, :, :] = H  # use in initial heads

Phi = fdm3(x, y, z, k, k, k, FQ, IH, IBOUND) # run model, return heads

# plot the heads a contours
plt.figure()
plt.xlabel('x [m]'); plt.ylabel('y [m]')
plt.title('Circular polder with surrounding land, semi confined regional aquifer')
plt.contourf(xm,ym,Phi[-1],50) # contour bottom layer

# plot the heads along the cross section and compare with analytical solution
plt.figure()
plt.xlabel('x [m]'); plt.ylabel('y [m]')
plt.title('Heads through the center of the polder and land')
centerRow = int(np.floor(Ny/2))

plt.plot(xm, Phi[-1, centerRow, :], label='Numeric')
Island = np.logical_and(xm>=-R, xm<=R)

# Analytical solution
RL = R/lam # shorthand

# head at x==R
xPold = xm[np.abs(xm) <= R] - x0 # inside polder
xLand = xm[np.abs(xm) >  R] - y0 # outside polder

# head a x==R, analytical
phiR = (hp * I1(RL) / I0(RL) + hl* K1(RL) / K0(RL)) / (I1(RL) / I0(RL) + K1(RL) / K0(RL)
```

```
        phiPold  = hp + (phiR - hp) * I0(np.abs(xPold) / lam) / I0(RL) # h in polder, r<=R
        phiLand  = hl + (phiR - hl) * K0(np.abs(xLand) / lam) / K0(RL) # h outside polder, r> R

        plt.plot(xPold, phiPold, 'r-', label='Poldm analytic')
        plt.plot(xLand, phiLand, 'g-', label='Land analytic')

        plt.legend()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Out[46]: <matplotlib.legend.Legend at 0x10dd7d710>
```

The analytical solution matches the numerical one except near the edges of the model, where the model is closed and the analytical solution is not. The numerical solution, therefore, becomes horizontal near the boundary, indicating no flow across it. One also notices the distortion of the circular color field in the first figure due to the boundaries of the square shape of our model. Notice that the only boundary conditions in this model are prescribed heads in the top layer, which are different inside and outside the radius `R`.

One nicely see both analytical solutions, one for the land outside the polder (green) and one for the land inside teh polder (red).

There is a small difference between the numerical and analytical solution in the center. Again, this may be due to the fact that the circular polder can't be perfectly circular due to the rectangular cells of the model. But the larger the number of cells, the smaller the difference will be.

It is important to note that we used half the vertical conductivity of the top layer. This is because in the model, water flows vertically from the center of the toplayer, where the head is fixed to the bottom, that is, it passes only half the cell. To keep the resistance of this lower half of the cell equal to the given vertical hydraulic resisttance, we have to use $ k0 = (d/2) / c $ instead of $ k0 = d/c $

More efficient coordinates

One sees that to set up a model, there area quite some lines that deal with the grid coordinates. This can be done a lot more efficient with a Grid class, the instances of which are invoked with the grid coordinates, after which a large number of variables can be requested from it, variables that are computed upon request.

CHAPTER 5

Computing flows with the finite difference model

Prof. dr.ir.T.N.Olsthoorn

Heemstede, Sept. 2016

Adding flows to the output of fdm3

Until now, the 3D finite difference model was used to compute only the heads in the cell centers. It should, however been clear that the model has all the information on board to also compute the flows. In fact, each line in the system matrix is a cell's water balance and computes flows over all faces of the cell in question including the total net inflow of each cell. The equation

$ Q_x = -C_x :raw-latex:‘times ‘,, diff (Phi, axis=1) $

is the flow across all cell faces perpendicular to the `x-axis`. Likewise

$ Q_y = +C_y :raw-latex:‘times ‘, , diff (Phi, axis=0) $

is the flow across all cell faces perpendicular to the `y-axis` in direction of this axis, and

$ Q_z = +C_z :raw-latex:‘times ‘, , diff(Phi, axis=2) $

is the upward vertical flow

In fact, we can readily compute these flow across cell faces within our model as is shown below.

We can also compute the total net inflow of all individual cells from the matrix multiplication

$ Q = A **:raw-latex:‘\times ‘:raw-latex:‘**Phi ‘$

where `A` is the system matrix with the conductances and Φ is the complete vector of heads, including the fixed heads, as all computed and prescribed heads are known after the model has been solved.

Flow output of the model

Because we have all the information to compute all flows at our disposition inside the function when computing the heads, we can, and perhaps should, at the same time compute these important flow arrays:

- `Q` : ndarray shape (`Ny`, `Nx` , `Nz`) [L3/T] # total net inflow of cells

- Qx : ndarray shape (Ny, Nx-1, Nz) [L3/T] # flow across cell faces in x-direction
- Qy : ndarray shape (Ny-1, Nx, Nz) [L3/T] # flow across cell faces in y-direction
- Qz : ndarray shape (Ny, Nx, Nz-1) [L3/T] # flow across cell faces in z-direction

We gather the computed arrays of Phi, Q, Qx, Qy and Qz in an single named tuple output named Out, from which the individual arrays may be addressed by their name like so:

Out.Phi, Out.Q, Out.Qx, Out.Qy and Out.Qz

Flows in the cell centers

The flows at across the cell iterfaces and the net inflow for each cell is uniquely defined by the equations given above. But this is not true for the velocity or specific discharge, because the cell face area jumps at the cell itnerface. So we mat have a different specific discharge perpendicular to the cell face just to the right and to the left of it.

Therefore, such specific discharge (the Darcy velocity) are generally computed for the cell centers by averaging the total flow at two opposite cell faces and deviding by the cross section perpendicular to the cell face. This vector is unique, yet not completely defined for the case when we also have an extraction in the cell. But this does not geneally or necessarly disturb the picture of velocity vectors much. So we will use this method further down.

We have written a function quivdata to extract the necessary coordinates and flows or velocities for displaying them as arrows with the matplotlib function quiver, see example further down.

Some more additions been made to the model:

- Applying the function unique to x, y and z input vectors to prevent duplicates. The function unique is defined in the module fdm. It allows entering coordinates in arbitrary order as they will be sorted and double values will be eliminated according to some user definable tolerance.
- Keeping track of the axis directions. We demand that x must be ascending, while y and z must be descending. This way columns (x) run from left to right when printing, the top row is the one with the largest y-coordinate and the top layer is the one with the highest z-coordinate. Enforcing these directions make interpreting scrolled or printed output intuitive and ensures the correct sign of flows. That is, a postive Qx, Qy and Qz always points into the direction of higher coordinates.

The adapted model is shown below.

fdm3 model with computation of flows

The model fdm3 is written to the module fdm.py together with the functions quivdata and unique

In [1]:
```
%%writefile fdm_b.py

        import numpy as np
        import pdb
        import scipy.sparse as sp
        from scipy.sparse.linalg import spsolve # to use its short name
        from collections import namedtuple

        class InputError(Exception):
            pass

        def quivdata(Out, x, y, z=None, iz=0):
            """Returns coordinates and velocity components to show velocity with quiver

            Compute arrays to display velocity vectors using matplotlib's quiver.
            The quiver is always drawn in the xy-plane for a specific layer. Default iz=0
```

```
    Parameters
    ----------
    `Out` : namedtuple holding arrays `Qx`, `Qy`, `Qz` as defined in `fdm3`
        `Qx` : ndarray, shape: (Ny, Nx-1, Nz), [L3/T]
            Interfacial flows in finite difference model in x-direction from `fdm3'
        `Qy` : ndarray, shape: (Ny-1, Nx, Nz), [L3/T]
            Interfacial flows in finite difference model in y-direction from `fdm3`
        `Qz` : ndarray, shape: (Ny, Nx, Nz-1), [L3/T]
            Interfacial flows in finite difference model in z-direction from `fdm3`
    `x` : ndarray, [m]
        Grid line coordinates of columns
    'y' : ndarray, [m]
        Grid line coordinates of rows
    `z` : ndaray [L] | int [-]
        If z == None, then iz must be given (default = 0)
        If z is an ndarray vector of floats
            z will be interpreted as the elvations of uniform layers.
            iz will be ignored
        If z is a full 3D ndarray of floats
            z will be interpreted as the elevations of the tops and bottoms of all cells.
            iz will be ignored
    `iz` : int [-]
            iz is ignored if z ~= None
            iz is the number of the layer for which the data are requested,
            and all output arrays will be 2D for that layer.

    Returns
    -------
    `Xm` : ndarray, shape: (Nz, Ny, Nx), [L]
        x-coordinates of cell centers
    `Ym` : ndarray, shape: (Nz, Ny, Nx), [L]
        y-coodinates of cell centers
    `ZM` : ndarray, shape: (Nz, Ny, Nx), [L]
        `z`-coordinates at cell centers
    `U` : ndarray, shape: (Nz, Ny, Nx), [L3/d]
        Flow in `x`-direction at cell centers
    `V` : ndarray, shape: (Nz, Ny, Nx), [L3/T]
        Flow in `y`-direction at cell centers
    `W` : ndarray, shape: (Nz, Ny, Nx), [L3/T]
        Flow in `z`-direction at cell centers.

    """
    Ny = len(y)-1
    Nx = len(x)-1

    xm = 0.5 * (x[:-1] + x[1:])
    ym = 0.5 * (y[:-1] + y[1:])

    X, Y = np.meshgrid(xm, ym) # coordinates of cell centers

    # Flows at cell centers
    U = np.concatenate((Out.Qx[iz, :, 0].reshape((1, Ny, 1)), \
                        0.5 * (Out.Qx[iz, :, :-1].reshape((1, Ny, Nx-2)) +\
                               Out.Qx[iz, :, 1: ].reshape((1, Ny, Nx-2))), \
                        Out.Qx[iz, :, -1].reshape((1, Ny, 1))), axis=2).reshape((Ny,Nx))
    V = np.concatenate((Out.Qy[iz, 0, :].reshape((1, 1, Nx)), \
                        0.5 * (Out.Qy[iz, :-1, :].reshape((1, Ny-2, Nx)) +\
                               Out.Qy[iz, 1:,  :].reshape((1, Ny-2, Nx))), \
                        Out.Qy[iz, -1, :].reshape((1, 1, Nx))), axis=1).reshape((Ny,Nx))
    return X, Y, U, V


def unique(x, tol=0.0001):
```

```
    """return sorted unique values of x, keeping ascending or descending direction"""
    if x[0]>x[-1]:  # vector is reversed
        x = np.sort(x)[::-1]  # sort and reverse
        return x[np.hstack((np.diff(x) < -tol, True))]
    else:
        x = np.sort(x)
        return x[np.hstack((np.diff(x) > +tol, True))]


def fdm3(x, y, z, kx, ky, kz, FQ, HI, IBOUND):
    '''Steady state 3D Finite Difference Model returning computed heads and flows.

    Heads and flows are returned as 3D arrays as specified under output parmeters.

    Parameters
    ----------
    `x` : ndarray,[L]
        `x` coordinates of grid lines perpendicular to rows, len is Nx+1
    `y` : ndarray, [L]
        `y` coordinates of grid lines along perpendicular to columns, len is Ny+1
    `z` : ndarray, [L]
        `z` coordinates of layers tops and bottoms, len = Nz+1
    `kx`, `ky`, `kz` : ndarray, shape: (Ny, Nx, Nz), [L/T]
        hydraulic conductivities along the three axes, 3D arrays.
    `FQ` : ndarray, shape: (Ny, Nx, Nz), [L3/T]
        prescrived cell flows (injection positive, zero of no inflow/outflow)
    `IH` : ndarray, shape: (Ny, Nx, Nz), [L]
        initial heads. `IH` has the prescribed heads for the cells with prescribed head.
    `IBOUND` : ndarray, shape: (Ny, Nx, Nz) of int
        boundary array like in MODFLOW with values denoting
        * IBOUND>0  the head in the corresponding cells will be computed
        * IBOUND=0  cells are inactive, will be given value NaN
        * IBOUND<0  coresponding cells have prescribed head

    outputs
    -------
    `Out` : namedtuple containing heads and flows:
        `Out.Phi` : ndarray, shape: (Ny, Nx, Nz), [L3/T]
            computed heads. Inactive cells will have NaNs
        `Out.Q`   : ndarray, shape: (Ny, Nx, Nz), [L3/T]
            net inflow in all cells, inactive cells have 0
        `Out.Qx   : ndarray, shape: (Ny, Nx-1, Nz), [L3/T]
            intercell flows in x-direction (parallel to the rows)
        `Out.Qy`  : ndarray, shape: (Ny-1, Nx, Nz), [L3/T]
            intercell flows in y-direction (parallel to the columns)
        `Out.Qz`  : ndarray, shape: (Ny, Nx, Nz-1), [L3/T]
            intercell flows in z-direction (vertially upward postitive)
        the 3D array with the final heads with `NaN` at inactive cells.

    TO 160905
    '''

    # define the named tuple to hold all the output of the model fdm3
    Out = namedtuple('Out',['Phi', 'Q', 'Qx', 'Qy', 'Qz'])
    Out.__doc__ = """fdm3 output, <namedtuple>, containing fields Phi, Qx, Qy and Qz\n \
                    Use Out.Phi, Out.Q, Out.Qx, Out.Qy and Out.Qz"""

    x = unique(x)
    y = unique(y)[::-1]  # unique and descending
    z = unique(z)[::-1]  # unique and descending

    # as well as the number of cells along the three axes
    SHP = Nz, Ny, Nx = len(z)-1, len(y)-1, len(x)-1
```

```
Nod = np.prod(SHP)

if Nod == 0:
    raise AssetError(
        "Grid shape is (Ny, Nx, Nz) = {0}. Number of cells in all 3 direction must al

if kx.shape != SHP:
    raise AssertionError("shape of kx {0} differs from that of model {1}".format(kx.sh
if ky.shape != SHP:
    raise AssertionError("shape of ky {0} differs from that of model {1}".format(ky.sh
if kz.shape != SHP:
    raise AssertionError("shape of kz {0} differs from that of model {1}".format(kz.sh

# from this we have the width of columns, rows and layers
dx = np.abs(np.diff(x)).reshape(1, 1, Nx)
dy = np.abs(np.diff(y)).reshape(1, Ny, 1)
dz = np.abs(np.diff(z)).reshape(Nz, 1, 1)

active = (IBOUND>0).reshape(Nod,)  # boolean vector denoting the active cells
inact  = (IBOUND==0).reshape(Nod,) # boolean vector denoting inacive cells
fxhd   = (IBOUND<0).reshape(Nod,)  # boolean vector denoting fixed-head cells

# half cell flow resistances
Rx = 0.5 * dx / (dy * dz) / kx
Ry = 0.5 * dy / (dz * dx) / ky
Rz = 0.5 * dz / (dx * dy) / kz

# set flow resistance in inactive cells to infinite
Rx = Rx.reshape(Nod,); Rx[inact] = np.Inf; Rx=Rx.reshape(SHP)
Ry = Ry.reshape(Nod,); Ry[inact] = np.Inf; Ry=Ry.reshape(SHP)
Rz = Rz.reshape(Nod,); Rz[inact] = np.Inf; Rz=Rz.reshape(SHP)

# conductances between adjacent cells
Cx = 1 / (Rx[:, :, :-1] + Rx[:, :, 1:])
Cy = 1 / (Ry[:, :-1, :] + Ry[:, 1:, :])
Cz = 1 / (Rz[:-1, :, :] + Rz[1:, :, :])

NOD = np.arange(Nod).reshape(SHP)

IE = NOD[:, :, 1:]  # east neighbor cell numbers
IW = NOD[:, :, :-1] # west neighbor cell numbers
IN = NOD[:, :-1, :] # north neighbor cell numbers
IS = NOD[:, 1:, :]  # south neighbor cell numbers
IT = NOD[:-1, :, :] # top neighbor cell numbers
IB = NOD[1:, :, :]  # bottom neighbor cell numbers

R = lambda x : x.ravel()  # generate anonymous function R(x) as shorthand for x.ravel

# notice the call  csc_matrix( (data, (rowind, coind) ), (M,N))  tuple within tupple
# also notice that Cij = negative but that Cii will be postive, namely -sum(Cij)
A = sp.csc_matrix(( -np.concatenate(( R(Cx), R(Cx), R(Cy), R(Cy), R(Cz), R(Cz)) ),\
                    (np.concatenate(( R(IE), R(IW), R(IN), R(IS), R(IB), R(IT)) ),\
                     np.concatenate(( R(IW), R(IE), R(IS), R(IN), R(IT), R(IB)) ),\
                  )),(Nod,Nod))

# to use the vector of diagonal values in a call of sp.diags() we need to have it aa a
# standard nondimensional numpy vector.
# To get this:
# - first turn the matrix obtained by A.sum(axis=1) into a np.array by np.array( .. )
# - then take the whole column to loose the array orientation (to get a dimensionless
adiag = np.array(-A.sum(axis=1))[:,0]
```

```
            Adiag = sp.diags(adiag)  # diagonal matrix with a[i,i]

            RHS = FQ.reshape(Nod,1) - A[:,fxhd].dot(HI.reshape(Nod,1)[fxhd]) # Right-hand side ve

            Out.Phi = HI.flatten() # allocate space to store heads

            Out.Phi[active] = spsolve( (A+Adiag)[active][:,active] ,RHS[active] ) # solve heads a

            # net cell inflow
            Out.Q   = (A+Adiag).dot(Out.Phi).reshape(SHP)

            # set inactive cells to NaN
            Out.Phi[inact] = np.NaN # put NaN at inactive locations

            # reshape Phi to shape of grid
            Out.Phi = Out.Phi.reshape(SHP)

            #Flows across cell faces
            Out.Qx =  -np.diff( Out.Phi, axis=2) * Cx
            Out.Qy =  +np.diff( Out.Phi, axis=1) * Cy
            Out.Qz =  +np.diff( Out.Phi, axis=0) * Cz

            return Out # all outputs in a named tuple for easy access
```

```
Overwriting fdm_b.py
```

These functions have now been saved in the file "fdm_b.py". This file is a module, which can be imported, after which the functions in it can be used in programs and scripts.

```
In [2]: import fdm_b
        from importlib import reload
        reload(fdm_b)
```

```
Out[2]: <module 'fdm_b' from '/Users/Theo/GRWMODELS/python/FDM_course/fdm_b.py'>
```

Application of the model

Generate input to run the model with (same example)

```
In [3]: import numpy as np
        reload(fdm_b) # make sure we reload because we edit the file regularly

        # specify a rectangular grid
        x = np.arange(-1000.,  1000.,  25.)
        y = np.arange( 1000., -1000., -25.)
        z = np.array([20, 0 ,-10, -100.])

        Nz, Ny, Nx = SHP = len(z)-1, len(y)-1, len(x)-1

        k = 10.0 # m/d uniform conductivity
        kx = k * np.ones(SHP)
        ky = k * np.ones(SHP)
        kz = k * np.ones(SHP)

        IBOUND = np.ones(SHP)
        IBOUND[:, -1, :] = -1  # last row of model heads are prescribed
        IBOUND[:, 40:45, 20:70]=0 # inactive

        FQ = np.zeros(SHP)     # all flows zero. Note SHP is the shape of the model grid
        FQ[1, 30, 25] = -1200  # extraction in this cell

        HI = np.zeros(SHP)
```

Call the function with the correct arguments

```
In [4]: Out = fdm_b.fdm3( x, y, z, kx, ky, kz, FQ, HI, IBOUND)

        # when not using some of the parameters
        #Phi, _, _, _ = fdm.fdm3( x, y, z, kx, ky, kz, FQ, HI, IBOUND) # ignore them using the _
        #Phi, _Qx, _Qy, _Qz = fdm.fdm3( x, y, z, kx, ky, kz, FQ, HI, IBOUND)  # make them private
```

Visualization of the results: plot heads as contours

Import the required plotting module and setup the plot.

```
In [5]: %matplotlib notebook
        import matplotlib.pyplot as plt # combines namespace of numpy and pyplot
```

```
In [6]: print('Out.Phi.shape = {0}'.format(Out.Phi.shape))
        print('Out.Q.shape = {0}'.format(Out.Q.shape))
        print('Out.Qx.shape = {0}'.format(Out.Qx.shape))
        print('Out.Qy.shape = {0}'.format(Out.Qy.shape))
        print('Out.Qz.shape = {0}'.format(Out.Qz.shape))
```

```
Out.Phi.shape = (3, 79, 79)
Out.Q.shape = (3, 79, 79)
Out.Qx.shape = (3, 79, 78)
Out.Qy.shape = (3, 78, 79)
Out.Qz.shape = (2, 79, 79)
```

```
In [7]: xm = 0.5 * (x[:-1] + x[1:])
        ym = 0.5 * (y[:-1] + y[1:])

        layer = 2 # contours for this layer
        nc = 50   # number of contours in total

        plt.xlabel('x [m]')
        plt.ylabel('y [m]')
        plt.title("Contours (%d in total) of the head in layer %d with inactive section" % (nc, l
        plt.contour(xm, ym, Out.Phi[layer], nc)

        #plt.quiver(X, Y, U, V) # show velocity vectors
        X, Y, U, V = fdm_b.quivdata(Out, x, y, iz=0)
        plt.quiver(X, Y, U, V)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Out[7]: <matplotlib.quiver.Quiver at 0x10e69d898>
```

- The white area is the aquifer part that was defined as inactive (impervious).
- The red trough is the well location.
- The two flanks and the front sides are closed (no FQ and no fixed head).
- The head at the back side is prescribed and maintained at zero.

Notice that the quidata function uses the computed flow arrays Out.Qx and Out.Qy. We can verfiy the net inflow for all cells by printing or contouring the values.

```
In [8]: print('\nSum of the net inflow over all cells is sum(Q) = {0:g} [m3/d]\n'.format(np.sum(Ou
```

```
Sum of the net inflow over all cells is sum(Q) = 4.80952e-10 [m3/d]
```

It can be seen that the sum of `Out.Q` over all cells is indeed zero (almost).

```
In [9]: print('\nThe indivdual values for the top layer are shown here:')
        print(np.round(Out.Q[:,:,0].T,2))
```

```
The indivdual values for the top layer are shown here:
[[ -0.     0.     0.  ]
 [  0.     0.    -0.  ]
 [  0.     0.     0.  ]
 [  0.    -0.    -0.  ]
 [ -0.    -0.     0.  ]
 [ -0.    -0.     0.  ]
 [  0.    -0.     0.  ]
 [ -0.    -0.    -0.  ]
 [ -0.     0.    -0.  ]
 [ -0.     0.     0.  ]
 [  0.     0.    -0.  ]
 [ -0.     0.    -0.  ]
 [  0.     0.     0.  ]
 [ -0.     0.     0.  ]
 [  0.     0.     0.  ]
 [ -0.     0.    -0.  ]
 [  0.    -0.     0.  ]
 [ -0.     0.    -0.  ]
 [  0.     0.     0.  ]
 [  0.     0.    -0.  ]
 [ -0.     0.    -0.  ]
 [ -0.     0.     0.  ]
 [ -0.     0.    -0.  ]
 [  0.     0.     0.  ]
 [ -0.     0.    -0.  ]
 [  0.     0.     0.  ]
 [ -0.     0.     0.  ]
 [ -0.     0.    -0.  ]
 [  0.     0.    -0.  ]
 [ -0.     0.     0.  ]
 [  0.     0.     0.  ]
 [ -0.     0.     0.  ]
 [  0.     0.     0.  ]
 [ -0.     0.    -0.  ]
 [  0.     0.    -0.  ]
 [  0.    -0.     0.  ]
 [ -0.     0.    -0.  ]
 [ -0.     0.    -0.  ]
 [  0.    -0.     0.  ]
 [  0.     0.    -0.  ]
 [  0.    -0.    -0.  ]
 [ -0.    -0.    -0.  ]
 [  0.     0.     0.  ]
 [ -0.     0.    -0.  ]
 [ -0.     0.    -0.  ]
 [ -0.     0.     0.  ]
 [  0.     0.     0.  ]
 [ -0.     0.    -0.  ]
 [ -0.     0.     0.  ]
 [ -0.     0.    -0.  ]
 [ -0.    -0.     0.  ]
 [ -0.     0.    -0.  ]
 [ -0.     0.     0.  ]
 [ -0.     0.     0.  ]
 [ -0.     0.    -0.  ]
 [ -0.     0.     0.  ]
 [  0.    -0.    -0.  ]
 [ -0.     0.    -0.  ]
 [ -0.     0.     0.  ]]
```

```
 [ -0.      0.      0.  ]
 [ -0.      0.     -0.  ]
 [ -0.      0.      0.  ]
 [ -0.      0.      0.  ]
 [ -0.      0.      0.  ]
 [  0.     -0.      0.  ]
 [ -0.      0.     -0.  ]
 [  0.      0.      0.  ]
 [ -0.      0.      0.  ]
 [  0.      0.     -0.  ]
 [  0.      0.     -0.  ]
 [ -0.      0.      0.  ]
 [ -0.      0.     -0.  ]
 [  0.      0.      0.  ]
 [ -0.     -0.      0.  ]
 [ -0.     -0.      0.  ]
 [  0.      0.     -0.  ]
 [  0.     -0.     -0.  ]
 [ -0.      0.      0.  ]
 [ 16.58    1.84    3.68]]
```

The non-zero cells correspond to cells with prescribed flows or cells with prescribed heads, for which the netto inflows are computed using all computed and prescribed heads.

```
In [10]: plt.figure(); plt.title('Q of cells in top layer [m3/d]')
         plt.imshow(Out.Q[0, :, :], interpolation='None')
         plt.colorbar()

         plt.figure(); plt.title('Q of cells in second layer [m3/d]')
         plt.imshow(Out.Q[1, :, :], interpolation='None')
         plt.colorbar()

         plt.figure(); plt.title('Q of cells in bottom layer with well [m3/d]')
         plt.imshow(Out.Q[1, :, :], interpolation='None')
         plt.colorbar()
```

```
<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Out[10]: <matplotlib.colorbar.Colorbar at 0x10f2e9748>
```

Notice:

- that the coordinates are the cell numbers, the model is (79, 79, 3).
- the colorbars are different for each figure due to aut-selection.
- the flows in the 3rd layer with the well varies indeed between -1200 (well extraction) and 0
- Further only the lower boundary is colored, i.e.non-zero, which is due to the fact that the heads are prescribed along that boundary and, therefore, the flows are computed. Positive values mean water is flowing from the outside world into the model.
- The zone with inactive cells are not visible, because the flow in those cells are zero as is the case for other cells with no prescribed flows or heads.

Conclusion

We have extended the 3D steady state finite difference model to not only compute the heads but also the flows between the cells and the net inflow of the cells. The latter are for the computed heads as well as for the prescribed heads. Inactive cells will have a head `NaN` (Not a Number) and will have flow zero.

We also now have the convenience function to extract the flows at the center of all cells for visualization using the `matplotlib` function `quiver()`.

In [ ]:

CHAPTER 6

A Grid class to deal with any finite difference model grid

Prof. dr.ir.T.N.Olsthoorn

Heemstede, Sept. 2016, 24 May 2017

Using a Grid class to handle spatial information regarding the grid

As we saw we often have to compute values of the grid, like the `xm`, `dx`, `Nx`, `Ny`, `shape` etc. A much better, more convenient and far less error-prone method of dealing with anything that has to do with the spacial dimensions of the finite difference grid is to define a Grid class, whose instances are created with the grid coordinates like so

```
gr = Grid(x, y, z)
```

after which any spatial information can be obtained from the actual Grid instanceuse, called gr in this example.

Requesting values then work like this:

```
gr.Nx        # int, len(x)-1
gr.shape     # tuple (Nt, Nx, Nz)
gr.xm        # ndarray, (len(x)-1
gr.Xm        # ndarray, (Ny, Nx) of x-coordinates of cell centers
gr.XM        # ndarray, (Nz, Ny, Nx) of x-coordinates of cell centers
gr.YM        # ndarray, (Nz, Ny, Nx) of y-coordinates of cell centers
gr.xm[3:10] # indexing gr.xm
gr.shape     # tuple, (Nz, Ny, Nx)
gr.area      # scalar, total area of the model
gr.Area      # ndarray, (Ny, Nx)
gr.volume    # scalar total volume of the model
gr.Volume    # ndarray, (Nz, Ny, Nx)
```

A large number of spacial or grid-specific variables, in fact, anything that can be computed from the coordinates can then be obtained.

The Grid class will take care of error checking and house-keeping. It can even be told to interpret the grid as axially symmetric

```
gr = Grid(x, y, z, axial=True)
```

It can guarantee a minimum layer thickness like so

```
gr = Grid(x, y, z, min_dz=0.001)
```

No matter if z used to invoke the Grid was specified as a vector telling the elevation of the tops and bottoms of uniform layers, or if z is a full-fledged 3D ndarray telling the top and bottom of all cells, Grid will handle it, in any case yielding a full 3D array of tops and bottoms when requested

```
gr.Z # full 3D array: shape (Ny, Nx, Nz+1)
```

and any other like grid related quantities that one may think of.

All grid related information is then contained in the grid object, where the z needs not be limited to a vector but may be a full 3D array of the tops and bottoms of all cells, so that each cell column can have elevations different from its neighbors. This approach is definitely much more flexible. Also, the grid can carry out all necessary error checking behing the scene which is effective as well.

The Grid class also has methods,like

```
A = gr.const(v)
```

This generates a ndarray of the size of the model with all values `v` if `v` is a scalar or with the values cells in layer `i` the value `v[i]` if `v` is a vector of length `gr.Nz`.

```
gr.plot(linespec)
```

will plot itself using the specified linespec, i.e. combination of color and linetype like used in `matplotilb.plot`.

Additionally, other functions like `fdm3` can be adapted to simply accept a `Grid` object as input instead of individual `x`, `y` and `z` coordinates. This requires less preparation and less clutting insize fdm3, while error checking then is delegated to the Grid object.

You can learn about the Grid class by introspection or by simply loading it in an editor and studying how it was implemented. Simply typing

```
Grid?
```

Provides the help from its docstring.

Grid-adapted model fdm3

The module fdm_b of the previous chapter contains the functions `unique()`, `quivdata()` and `fdm3`. This model is copied below but only the functions `fdm3` and `quivdata` have been adapted to deal with the grid object for its grid information.

The new module will be save as `fdm_c.py`

```
In [1]: %%writefile fdm_c.py

        import numpy as np
        import pdb
        import scipy.sparse as sp
        from scipy.sparse.linalg import spsolve # to use its short name
        from collections import namedtuple

        class InputError(Exception):
            pass

        def quivdata(Out, gr, iz=0):
            """Returns coordinates and velocity components to show velocity with quiver

            Compute arrays to display velocity vectors using matplotlib's quiver.
```

```
    The quiver is always drawn in the xy-plane for a specific layer. Default iz=0

    Parameters
    ----------
    `Out` : namedtuple holding arrays `Qx`, `Qy`, `Qz` as defined in `fdm3`
        `Qx` : ndarray, shape: (Nz, Ny, Nx-1), [L3/T]
            Interfacial flows in finite difference model in x-direction from `fdm3'
        `Qy` : ndarray, shape: (Nz, Ny-1, Nx), [L3/T]
            Interfacial flows in finite difference model in y-direction from `fdm3`
        `Qz` : ndarray, shape: (Nz-1, Ny, Nx), [L3/T]
            Interfacial flows in finite difference model in z-direction from `fdm3`
    `gr` : `grid_object` generated by Grid
    `iz` : int [-]
            iz is the number of the layer for which the data are requested,
            and all output arrays will be 2D for that layer.
            if iz==None, then all outputs will be full 3D arrays and cover all layers
            simultaneously

    Returns
    -------
    `Xm` : ndarray, shape: (Nz, Ny, Nx), [L]
        x-coordinates of cell centers
    `Ym` : ndarray, shape: (Nz, Ny, Nx), [L]
        y-coodinates of cell centers
    `ZM` : ndarray, shape: (Nz, Ny, Nx), [L]
        `z`-coordinates at cell centers
    `U` : ndarray, shape: (Nz, Ny, Nx), [L3/d]
        Flow in `x`-direction at cell centers
    `V` : ndarray, shape: (Nz, Ny, Nx), [L3/T]
        Flow in `y`-direction at cell centers
    `W` : ndarray, shape: (Nz, Ny, Nx), [L3/T]
        Flow in `z`-direction at cell centers.

    """

    X, Y = np.meshgrid(gr.xm, gr.ym) # coordinates of cell centers

    shp = (gr.Ny, gr.Nx) # 2D tuple to select a single layer

    # Flows at cell centers
    U = np.concatenate((Out.Qx[iz, :, 0].reshape((1, gr.Ny, 1)), \
                        0.5 * (Out.Qx[iz, :, :-1].reshape((1, gr.Ny, gr.Nx-2)) +\
                               Out.Qx[iz, :, 1: ].reshape((1, gr.Ny, gr.Nx-2))), \
                        Out.Qx[iz, :, -1].reshape((1, gr.Ny, 1))), axis=2).reshape(shp)
    V = np.concatenate((Out.Qy[iz, 0, :].reshape((1, 1, gr.Nx)), \
                        0.5 * (Out.Qy[iz, :-1, :].reshape((1, gr.Ny-2, gr.Nx)) +\
                               Out.Qy[iz, 1:,  :].reshape((1, gr.Ny-2, gr.Nx))), \
                        Out.Qy[iz, -1, :].reshape((1, 1, gr.Nx))), axis=1).reshape(shp)
    return X, Y, U, V


def unique(x, tol=0.0001):
    """return sorted unique values of x, keeping ascending or descending direction"""
    if x[0]>x[-1]:  # vector is reversed
        x = np.sort(x)[::-1]  # sort and reverse
        return x[np.hstack((np.diff(x) < -tol, True))]
    else:
        x = np.sort(x)
        return x[np.hstack((np.diff(x) > +tol, True))]


def fdm3(gr, kx, ky, kz, FQ, HI, IBOUND):
    '''Steady state 3D Finite Difference Model returning computed heads and flows.
```

```
Heads and flows are returned as 3D arrays as specified under output parmeters.

Parameters
----------
'gr' : `grid_object`, generated by gr = Grid(x, y, z, ..)
`kx`, `ky`, `kz` : ndarray, shape: (Nz, Ny, Nx), [L/T]
    hydraulic conductivities along the three axes, 3D arrays.
`FQ` : ndarray, shape: (Nz, Ny, Nx), [L3/T]
    prescrived cell flows (injection positive, zero of no inflow/outflow)
`IH` : ndarray, shape: (Nz, Ny, Nx), [L]
    initial heads. `IH` has the prescribed heads for the cells with prescribed head.
`IBOUND` : ndarray, shape: (Nz, Ny, Nx) of int
    boundary array like in MODFLOW with values denoting
    * IBOUND>0  the head in the corresponding cells will be computed
    * IBOUND=0  cells are inactive, will be given value NaN
    * IBOUND<0  coresponding cells have prescribed head

outputs
-------
`Out` : namedtuple containing heads and flows:
    `Out.Phi` : ndarray, shape: (Nz, Ny, Nx), [L3/T]
        computed heads. Inactive cells will have NaNs
    `Out.Q`   : ndarray, shape: (Nz, Ny, Nx), [L3/T]
        net inflow in all cells, inactive cells have 0
    `Out.Qx   : ndarray, shape: (Nz, Ny, Nx-1), [L3/T]
        intercell flows in x-direction (parallel to the rows)
    `Out.Qy`  : ndarray, shape: (Nz, Ny-1, Nx), [L3/T]
        intercell flows in y-direction (parallel to the columns)
    `Out.Qz`  : ndarray, shape: (Nz-1, Ny, Nx), [L3/T]
        intercell flows in z-direction (vertially upward postitive)
    the 3D array with the final heads with `NaN` at inactive cells.

TO 160905
'''

# define the named tuple to hold all the output of the model fdm3
Out = namedtuple('Out',['Phi', 'Q', 'Qx', 'Qy', 'Qz'])
Out.__doc__ = """fdm3 output, <namedtuple>, containing fields Phi, Qx, Qy and Qz\n \
               Use Out.Phi, Out.Q, Out.Qx, Out.Qy and Out.Qz"""

if kx.shape != gr.shape:
    raise AssertionError("shape of kx {0} differs from that of model {1}".format(kx.sh
if ky.shape != gr.shape:
    raise AssertionError("shape of ky {0} differs from that of model {1}".format(ky.sh
if kz.shape != gr.shape:
    raise AssertionError("shape of kz {0} differs from that of model {1}".format(kz.sh

active = (IBOUND > 0).reshape(gr.Nod,)  # boolean vector denoting the active cells
inact  = (IBOUND ==0).reshape(gr.Nod,) # boolean vector denoting inacive cells
fxhd   = (IBOUND < 0).reshape(gr.Nod,)  # boolean vector denoting fixed-head cells

# reshaping shorthands
rx = lambda a : np.reshape(a, (1, 1, gr.Nx))
ry = lambda a : np.reshape(a, (1, gr.Ny, 1))
rz = lambda a : np.reshape(a, (gr.Nz, 1, 1))

# half cell flow resistances
Rx = 0.5 * rx(gr.dx) / (ry(gr.dy) * rz(gr.dz)) / kx
Ry = 0.5 * ry(gr.dy) / (rz(gr.dz) * rx(gr.dx)) / ky
Rz = 0.5 * rz(gr.dz) / (rx(gr.dx) * ry(gr.dy)) / kz

# set flow resistance in inactive cells to infinite
```

```
    Rx = Rx.reshape(gr.Nod,); Rx[inact] = np.Inf; Rx=Rx.reshape(gr.shape)
    Ry = Ry.reshape(gr.Nod,); Ry[inact] = np.Inf; Ry=Ry.reshape(gr.shape)
    Rz = Rz.reshape(gr.Nod,); Rz[inact] = np.Inf; Rz=Rz.reshape(gr.shape)

    # conductances between adjacent cells
    Cx = 1 / (Rx[:, :, :-1] + Rx[:, :, 1:])
    Cy = 1 / (Ry[:, :-1, :] + Ry[:, 1:, :])
    Cz = 1 / (Rz[:-1, :, :] + Rz[1:, :, :])

    IE = gr.NOD[:, :, 1:]  # east neighbor cell numbers
    IW = gr.NOD[:, :, :-1] # west neighbor cell numbers
    IN = gr.NOD[:, :-1, :] # north neighbor cell numbers
    IS = gr.NOD[:, 1:, :]  # south neighbor cell numbers
    IT = gr.NOD[:-1, :, :] # top neighbor cell numbers
    IB = gr.NOD[1:, :, :]  # bottom neighbor cell numbers

    R = lambda x : x.ravel()  # generate anonymous function R(x) as shorthand for x.ravel

    # notice the call  csc_matrix( (data, (rowind, coind) ), (M,N))  tuple within tupple
    # also notice that Cij = negative but that Cii will be postive, namely -sum(Cij)
    A = sp.csc_matrix(( -np.concatenate(( R(Cx), R(Cx), R(Cy), R(Cy), R(Cz), R(Cz)) ),\
                        (np.concatenate(( R(IE), R(IW), R(IN), R(IS), R(IB), R(IT)) ),\
                         np.concatenate(( R(IW), R(IE), R(IS), R(IN), R(IT), R(IB)) ),\
                      )),(gr.Nod,gr.Nod))

    # to use the vector of diagonal values in a call of sp.diags() we need to have it aa a
    # standard nondimensional numpy vector.
    # To get this:
    # - first turn the matrix obtained by A.sum(axis=1) into a np.array by np.array( .. )
    # - then take the whole column to loose the array orientation (to get a dimensionless
    adiag = np.array(-A.sum(axis=1))[:,0]

    Adiag = sp.diags(adiag)  # diagonal matrix with a[i,i]

    RHS = FQ.reshape((gr.Nod,1)) - A[:,fxhd].dot(HI.reshape((gr.Nod,1))[fxhd]) # Right-han

    Out.Phi = HI.flatten() # allocate space to store heads

    Out.Phi[active] = spsolve( (A+Adiag)[active][:,active] ,RHS[active] ) # solve heads at

    # net cell inflow
    Out.Q   = (A+Adiag).dot(Out.Phi).reshape(gr.shape)

    # set inactive cells to NaN
    Out.Phi[inact] = np.NaN # put NaN at inactive locations

    # reshape Phi to shape of grid
    Out.Phi = Out.Phi.reshape(gr.shape)

    #Flows across cell faces
    Out.Qx =  -np.diff( Out.Phi, axis=2) * Cx
    Out.Qy =  +np.diff( Out.Phi, axis=1) * Cy
    Out.Qz =  +np.diff( Out.Phi, axis=0) * Cz

    return Out # all outputs in a named tuple for easy access
Overwriting fdm_c.py
```

In [2]:
```
import fdm_c
from importlib import reload
reload(fdm_c)
```

Out[2]:
```
<module 'fdm_c' from '/Users/Theo/GRWMODELS/Python_projects/FDM_course/fdm_c.py'>
```

Example

We use the same 3D example as before but now apply the grid object.

```
In [9]: # Make sure that your modules like grid are in the sys.path
        import sys

        path2modules = './modules/'

        if not path2modules in sys.path:
            sys.path.append(path2modules)


In [11]: reload(mfgrid)

Out[11]: <module 'mfgrid' from '/Users/Theo/GRWMODELS/Python_projects/FDM_course/mfgrid.py'>

In [12]: gr.z

Out[12]: array([  20.,    0.,  -10., -100.])

In [13]: %matplotlib notebook
         import matplotlib.pyplot as plt # combines namespace of numpy and pyplot

         import numpy as np
         import mfgrid
         Grid = mfgrid.Grid

         # specify a rectangular grid
         x = np.arange(-1000.,  1000.,  25.)
         y = np.arange( 1000., -1000., -25.)
         z = np.array([20, 0 ,-10, -100.])

         gr = Grid(x, y, z) # generating a grid object for this model

         k = 10.0 # m/d uniform conductivity
         kx = k * gr.const(k) # using gr.const(value) to generate a full 3D array of the correct
         ky = k * gr.const(k)
         kz = k * gr.const(k)

         IBOUND = gr.const(1)
         IBOUND[:, -1, :] = -1  # last row of model heads are prescribed
         IBOUND[:, 40:45, 20:70]=0 # inactive

         FQ = gr.const(0.)     # all flows zero. Note SHP is the shape of the model grid
         FQ[2, 30, 25] = -1200  # extraction in this cell

         HI = gr.const(0.)

         Out = fdm_c.fdm3(gr, kx, ky, kz, FQ, HI, IBOUND)

         layer = 2 # contours for this layer
         nc = 50   # number of contours in total

         plt.xlabel('x [m]')
         plt.ylabel('y [m]')
         plt.title("Contours (%d in total) of the head in layer %d with inactive section" % (nc,
         plt.contour(gr.xm, gr.ym, Out.Phi[:,:,layer], nc) # using gr here also

         #plt.quiver(X, Y, U, V) # show velocity vectors
         #X, Y, U, V = fdm_c.quivdata(Out, gr, iz=0) # use function in fdm_c
         X, Y, U, V = gr.quivdata(Out, iz=0) # use method in Grid
         plt.quiver(X, Y, U, V)

<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Out[13]: <matplotlib.quiver.Quiver at 0x10eb88048>
```

Conclusion

Managing and providing spatial information with respect to the grid of our finite difference models can be effectively delegated to a Grid object, which is a instance of the Grid class.

We have adapted our finite differnce model `fmd3` and the function `quivdata` to make use of the grid. We then used the grid to more effectively setup a the same model as before and handle its spatial informaion.

We finally applied the adapte function quivdata to show the quiver. We can alternatively use this function or the method quivdata implemented in the Grid class.

```
In [ ]:
```

CHAPTER 7

Axially symmetric modeling

Prof. dr.ir. T.N.Olsthoorn

Heemstede, Okt. 2016, 24 May 2017

Theory

```
In [ ]:
```

```
In [2]: myModules = '/Users/Theo/GRWMODELS/python/modules/fdm/'
```

```
        import sys
        if not myModules in sys.path:
            sys.path.append(myModules)

        import mfgrid
```

```
def inpoly(...)
```

Often it is useful to simulate groundwater flow in an axial symmetric using only the coordinates x and z, where x stands for the radius r and y is not used. The thickness in y direction is no longer constant but varies with x as $y = 2\pi x^2$. Note that here $x > 0$. Columns in the grid are now, in fact, cylinders with thickness equal to that of the corresponding column.

To alow this, we have to adapt the computation of the cell resistances. Because in axial flow we have

$:raw-latex:*Delta* ‘*:raw-latex:*phi = *:raw-latex:*frac {Q} {2 pi k_x Delta z}‘ **:raw-latex:‘\ln ‘:raw-latex:‘**frac {r_1} {r_2}‘ $

The resistance $R_x = **:raw-latex:‘\frac {\Delta \phi} {Q}‘** = **:raw-latex:‘\frac ‘1 {2 :raw-latex:‘**pi *k_x :raw-latex:*Delta *z} :raw-latex:*ln *:raw-latex:*frac {r_2} {r_1}‘ $, with $r_2 > r_1$

We can thus write out the resistance against horitonzal axial flow between the innermost cell face at $x = r_1$ and the cell center at $x = r_m$ as well as the resistance between the cell center and the outer most cell face at $x = r_2$.

```
Rx1 = 1 / (2 * pi * kx * dz) * log(xm / x[:-1])
Rx2 = 1 / (2 * pi * kx * dz) * log(x[1:] / xm)
```

In vertical direction we have the resistance due to vertical flow through the rings

$R_z = \frac{\Delta z}{k_z \pi (r_2^2 - r_1^2)}$

These resistances thus become:

```
Rz  = dz / (kz * pi * (x[1:]**2 - x[:-1]**2))
```

And the cell conductances now become

```
Cx = 1 / (Rx1[1:] + Rx2[:-1])
Cy = np.zeros(sz)
Cz = 0.5 / (Rz[:,:,1:] + Rz[:,:,:-1])
```

Notice that the conductance in the y direction is left in and set to zeros. This implies that there will be no flow between adjacent model rows. When we now use a model with more than one row, we can look at it as a set of simultaneously computed independent axisymmetric models, because these models all share the same distance from zero by their x-coordinate and have no mutual interaction in the y-direction. Different axisymmetric models can thus be simulated simultaneously, which may be useful for comparisons, sensitivity compuatations and for calibration.

These are the only changes we have to make to convert our 3D groundwater model to an axisymmetric model.

We can build this into our existing model and use a switch to tell the model to run the rectangular or the axisymmetric case.

In Python this is easily done using a named input parameter `axial=false`.

The call would then look as follows

```
Phi = fdm3(x, y, z, kx, ky, kz, FQ, HI, IBOUND, axial=true)
```

The y coordinate vector is irrelevant here. It can be left empty or set to any value, e.g. `none`. Whatever it is, it will be ignored in the axisymmetrical case. Notice that the size of the model is obtained from the IBOUND array.

Implementation; the adepted module to include axial symmetry

```
In [9]: %%writefile fdm_d.py

        import numpy as np
        import pdb
        import scipy.sparse as sp
        from scipy.sparse.linalg import spsolve # to use its short name
        from collections import namedtuple

        class InputError(Exception):
            pass

        def quivdata(Out, gr, iz=0):
            """Returns coordinates and velocity components to show velocity with quiver

            Compute arrays to display velocity vectors using matplotlib's quiver.
            The quiver is always drawn in the xy-plane for a specific layer. Default iz=0

            Parameters
            ----------
            `Out` : namedtuple holding arrays `Qx`, `Qy`, `Qz` as defined in `fdm3`
                `Qx` : ndarray, shape: (Nz, Ny, Nx-1), [L3/T]
                    Interfacial flows in finite difference model in x-direction from `fdm3'
                `Qy` : ndarray, shape: (Nz, Ny-1, Nx), [L3/T]
                    Interfacial flows in finite difference model in y-direction from `fdm3`
                `Qz` : ndarray, shape: (Nz-1, Ny, Nx), [L3/T]
                    Interfacial flows in finite difference model in z-direction from `fdm3`
            `gr` : `grid_object` generated by Grid
            `iz` : int [-]
                    iz is the number of the layer for which the data are requested,
```

```
            and all output arrays will be 2D for that layer.
            if iz==None, then all outputs will be full 3D arrays and cover all layers
            simultaneously

    Returns
    -------
    `Xm` : ndarray, shape: (Nz, Ny, Nx), [L]
        x-coordinates of cell centers
    `Ym` : ndarray, shape: (Nz, Ny, Nx), [L]
        y-coodinates of cell centers
    `ZM` : ndarray, shape: (Nz, Ny, Nx), [L]
        `z`-coordinates at cell centers
    `U` : ndarray, shape: (Nz, Ny, Nx), [L3/d]
        Flow in `x`-direction at cell centers
    `V` : ndarray, shape: (Nz, Ny, Nx), [L3/T]
        Flow in `y`-direction at cell centers
    `W` : ndarray, shape: (Nz, Ny, Nx), [L3/T]
        Flow in `z`-direction at cell centers.

    """

    X, Y = np.meshgrid(gr.xm, gr.ym) # coordinates of cell centers

    shp = (gr.Ny, gr.Nx) # 2D tuple to select a single layer

    # Flows at cell centers
    U = np.concatenate((Out.Qx[iz, :, 0].reshape((1, gr.Ny, 1)), \
                        0.5 * (Out.Qx[iz, :, :-1].reshape((1, gr.Ny, gr.Nx-2)) +\
                               Out.Qx[iz, :, 1: ].reshape((1, gr.Ny, gr.Nx-2))), \
                        Out.Qx[iz, :, -1].reshape((1, gr.Ny, 1))), axis=2).reshape(shp)
    V = np.concatenate((Out.Qy[iz, 0, :].reshape((1, 1, gr.Nx)), \
                        0.5 * (Out.Qy[iz, :-1, :].reshape((1, gr.Ny-2, gr.Nx)) +\
                               Out.Qy[iz, 1:,  :].reshape((1, gr.Ny-2, gr.Nx))), \
                        Out.Qy[iz, -1, :].reshape((1, 1, gr.Nx))), axis=1).reshape(shp)
    return X, Y, U, V


def unique(x, tol=0.0001):
    """return sorted unique values of x, keeping ascending or descending direction"""
    if x[0]>x[-1]:  # vector is reversed
        x = np.sort(x)[::-1]  # sort and reverse
        return x[np.hstack((np.diff(x) < -tol, True))]
    else:
        x = np.sort(x)
        return x[np.hstack((np.diff(x) > +tol, True))]


def fdm3(gr, kx, ky, kz, FQ, HI, IBOUND):
    '''Steady state 3D Finite Difference Model returning computed heads and flows.

    Heads and flows are returned as 3D arrays as specified under output parmeters.

    Parameters
    ----------
    'gr' : `grid_object`, generated by gr = Grid(x, y, z, ..)
        if `gr.axial`==True, then the model is run in axially symmetric model
    `kx`, `ky`, `kz` : ndarray, shape: (Nz, Ny, Nx), [L/T]
        hydraulic conductivities along the three axes, 3D arrays.
    `FQ` : ndarray, shape: (Nz, Ny, Nx), [L3/T]
        prescrived cell flows (injection positive, zero of no inflow/outflow)
    `IH` : ndarray, shape: (Nz, Ny, Nx), [L]
        initial heads. `IH` has the prescribed heads for the cells with prescribed head.
    `IBOUND` : ndarray, shape: (Nz, Ny, Nx) of int
```

```
    boundary array like in MODFLOW with values denoting
    * IBOUND>0  the head in the corresponding cells will be computed
    * IBOUND=0  cells are inactive, will be given value NaN
    * IBOUND<0  coresponding cells have prescribed head

outputs
-------
`Out` : namedtuple containing heads and flows:
    `Out.Phi` : ndarray, shape: (Nz, Ny, Nx), [L3/T]
        computed heads. Inactive cells will have NaNs
    `Out.Q`   : ndarray, shape: (Nz, Ny, Nx), [L3/T]
        net inflow in all cells, inactive cells have 0
    `Out.Qx   : ndarray, shape: (Nz, Ny, Nx-1), [L3/T]
        intercell flows in x-direction (parallel to the rows)
    `Out.Qy`  : ndarray, shape: (Nz, Ny-1, Nx), [L3/T]
        intercell flows in y-direction (parallel to the columns)
    `Out.Qz`  : ndarray, shape: (Nz-1, Ny, Nx), [L3/T]
        intercell flows in z-direction (vertially upward postitive)
    the 3D array with the final heads with `NaN` at inactive cells.

TO 160905
'''

import pdb

# define the named tuple to hold all the output of the model fdm3
Out = namedtuple('Out',['Phi', 'Q', 'Qx', 'Qy', 'Qz'])
Out.__doc__ = """fdm3 output, <namedtuple>, containing fields Phi, Qx, Qy and Qz\n \
               Use Out.Phi, Out.Q, Out.Qx, Out.Qy and Out.Qz"""

if gr.axial:
    print('Running in axial mode, y-values are ignored.')

if kx.shape != gr.shape:
    raise AssertionError("shape of kx {0} differs from that of model {1}".format(kx.sh
if ky.shape != gr.shape:
    raise AssertionError("shape of ky {0} differs from that of model {1}".format(ky.sh
if kz.shape != gr.shape:
    raise AssertionError("shape of kz {0} differs from that of model {1}".format(kz.sh

active = (IBOUND>0).reshape(gr.Nod,)  # boolean vector denoting the active cells
inact  = (IBOUND==0).reshape(gr.Nod,) # boolean vector denoting inacive cells
fxhd   = (IBOUND<0).reshape(gr.Nod,)  # boolean vector denoting fixed-head cells

# reshaping shorthands
dx = np.reshape(gr.dx, (1, 1, gr.Nx))
dy = np.reshape(gr.dy, (1, gr.Ny, 1))

# half cell flow resistances
if not gr.axial:
    Rx1 = 0.5 *    dx / (   dy * gr.DZ) / kx
    Rx2 = Rx1
    Ry1 = 0.5 *    dy / (gr.DZ *    dx) / ky
    Rz1 = 0.5 * gr.DZ / (   dx *    dy) / kz
else:
    min_dx = 0.000001
    x = gr.x; x[0] = max(0.1 * x[1], x[0]) # preventd division by zero x[0]
    Rx1 = 1 / (2 * np.pi * kx * gr.DZ) * np.log(x[1:] /  gr.xm).reshape((1, 1, gr.Nx))
    Rx2 = 1 / (2 * np.pi * kx * gr.DZ) * np.log(gr.xm / x[:-1]).reshape((1, 1, gr.Nx))
    Ry1 = np.inf * np.ones(gr.shape)
    Rz1 = 0.5 * gr.DZ / (np.pi * (gr.x[1:]**2 - gr.x[:-1]**2).reshape((1, 1, gr.Nx)))

# set flow resistance in inactive cells to infinite
```

```
Rx1[inact.reshape(gr.shape)] = np.inf
Rx2[inact.reshape(gr.shape)] = np.inf
Ry1[inact.reshape(gr.shape)] = np.inf
Ry2 = Ry1
Rz1[inact.reshape(gr.shape)] = np.inf
Rz2 = Rz1

# conductances between adjacent cells
Cx = 1 / (Rx1[ :, :, 1:] + Rx2[:, :, :-1])
Cy = 1 / (Ry1[ :, :-1,:] + Ry2[:,  1:, :])
Cz = 1 / (Rz1[:-1, :, :] + Rz2[1:, :,  :])

#pdb.set_trace()

IE = gr.NOD[:, :, 1:]  # east neighbor cell numbers
IW = gr.NOD[:, :, :-1] # west neighbor cell numbers
IN = gr.NOD[:, :-1, :] # north neighbor cell numbers
IS = gr.NOD[:, 1:, :]  # south neighbor cell numbers
IT = gr.NOD[:-1, :, :] # top neighbor cell numbers
IB = gr.NOD[1:, :, :]  # bottom neighbor cell numbers

R = lambda x : x.ravel()  # generate anonymous function R(x) as shorthand for x.ravel

# notice the call  csc_matrix( (data, (rowind, coind) ), (M,N))  tuple within tupple
# also notice that Cij = negative but that Cii will be postive, namely -sum(Cij)
A = sp.csc_matrix(( -np.concatenate(( R(Cx), R(Cx), R(Cy), R(Cy), R(Cz), R(Cz)) ),\
                     (np.concatenate(( R(IE), R(IW), R(IN), R(IS), R(IB), R(IT)) ),\
                      np.concatenate(( R(IW), R(IE), R(IS), R(IN), R(IT), R(IB)) ),\
                   )),(gr.Nod,gr.Nod))

# to use the vector of diagonal values in a call of sp.diags() we need to have it aa a
# standard nondimensional numpy vector.
# To get this:
# - first turn the matrix obtained by A.sum(axis=1) into a np.array by np.array( .. )
# - then take the whole column to loose the array orientation (to get a dimensionless
adiag = np.array(-A.sum(axis=1))[:,0]

Adiag = sp.diags(adiag)  # diagonal matrix with a[i,i]

#pdb.set_trace()

RHS = FQ.reshape((gr.Nod,1)) - A[:,fxhd].dot(HI.reshape((gr.Nod,1))[fxhd]) # Right-han

Out.Phi = HI.flatten() # allocate space to store heads

Out.Phi[active] = spsolve( (A+Adiag)[active][:,active] ,RHS[active] ) # solve heads at

# net cell inflow
Out.Q  = (A+Adiag).dot(Out.Phi).reshape(gr.shape)

# reshape Phi to shape of grid
Out.Phi = Out.Phi.reshape(gr.shape)

#Flows across cell faces
Out.Qx =  -np.diff( Out.Phi, axis=2) * Cx
Out.Qy =  +np.diff( Out.Phi, axis=1) * Cy
Out.Qz =  +np.diff( Out.Phi, axis=0) * Cz

    # set inactive cells to NaN
Out.Phi[inact.reshape(gr.shape)] = np.NaN # put NaN at inactive locations


return Out # all outputs in a named tuple for easy access
```

```
Overwriting fdm_d.py
```

Examples

Here we'll work out a few axially symmetric examples to verify this model using analytical solutions.

We will also compute the darwdown in a multi-layer aquifer system.

We'll keep truly axially symmetric 3D flow for the next chapter, after we introduced the stream function.

```
In [10]: %matplotlib notebook

         import numpy as np
         import matplotlib.pyplot as plt
         from importlib import reload
         import fdm_d # from current directory
         import mfgrid  # path has been added to sys.path above

         # when we have been editing files, make sure to reload
         reload(fdm_d)
         reload(mfgrid)
```

```
def inpoly(...)
```

```
Out[10]: <module 'mfgrid' from '/Users/Theo/GRWMODELS/python/modules/fdm/mfgrid.py'>
```

Circular island

The first example is flow in a circular island with recharge, like we did in a previous chapter using a large-scale 2D or 3D model.

We use a single layer, which, in axially symmetric cases becomes a single row of cells, in which `x` should be read as `r`, the distance to the center of the island.

```
In [12]: # easily switch between linear and axisymmetrical flow
         axial = True

         # m/d, recharge rate
         rch = 0.01

         # The aquifer
         z0 = 0.  # m, ground surface elevaton
         D = 100. # m, aquifer thickness

         k0 = 10. # m/d, conductivity
         kD = k0 * D # m2/d, transmissivity, only used in the analytical solution

         # coordinates
         R = 750.0 # m, radius of the island

         # grid coordinates
         # R-0.1 and R+0.1 added to x-coordinates to allow head boundary at almost exactly x==R
         x = np.hstack([0., R-0.1, R+0.1, np.logspace(0 , np.log10(3*R), 100)])
         y = np.array([-0.5, 0.5])  # is ignored in axially symmetric flow
         z = z0 -  np.array([0, D])

         # grid
         gr = mfgrid.Grid(x, y, z, axial=axial) # generate a grid object and tell it's axially sy

         # required model arrays
         k  = gr.const(k0) # m/d, uniform conductivity array of correct size
         FQ = rch * gr.Area.reshape(gr.shape) # m3/d, cell inflows (gr.Area knows about axial)
         IH = gr.const(0.) # m, initial heads
```

```
        IBOUND = np.ones(gr.shape);   IBOUND[gr.XM>R] = -1 # boundary array, all r>R fixed heads

        # run the model, return Out containing heads and flows
        Out = fdm_d.fdm3(gr, k, k, k, FQ, IH, IBOUND)

        # tells which cells are x<=R (in the island), used in analytical solutions
        Island = np.logical_and(gr.xm>=-R, gr.xm<=R)

        # plot, set up plot
        plt.figure()

        # plot numerical results
        plt.plot(gr.xm, Out.Phi[0,0, :], 'bo', label='Numeric')

        # works for both axially symmetric and linear flow
        # sets correct title and plots analytical solution
        if axial:
            plt.xlabel('r [m]')
            plt.title('Circular island with radius R={0} m and recharge N = {1} m/d'.format(R,rc
            plt.plot(gr.xm[Island], rch/(4 * kD) * (R**2 - gr.xm[Island] ** 2),'r-', label='Analy
        else:
            plt.xlabel('x [m]')
            plt.title('Cross section with fixed head beyond x={0} m and recharge N = {1}'.format
            plt.plot(gr.xm[Island], rch/(2 * kD) * (R**2 - gr.xm[Island] ** 2),'r-', label='Analy
        plt.legend()
```

```
Running in axial mode, y-values are ignored.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Out[12]: <matplotlib.legend.Legend at 0x111493710>
```

As can be seen, the numerical and analyticl solutions agree. By changing the variable axial to False or True one can choose between a flat cross sectional model and an axisymmetrical model. The correct axis and analytical solutions will then be shown in the figure. The burden of changing variable for both cases is completely carried by the grid object and the way in which the conductances are computed in fdm3.

To show that the water balance matches we compute below the total flow into the model.

```
In [13]: print('The total net Q into the model should be (well, almost) zero: Qtotal = {} m3/d'.fo
```

```
The total net Q into the model should be (well, almost) zero: Qtotal = -2.9103830456733704e-11 m3
```

```
In [14]: if axial:
             dim='m3/d'
         else:
             dim='m2/d'

         print('The total net outflow should match the hand computed total recharge:')
         print('By hand: rch * pi * R**2  =   {1:8.5g} {0}'.format(dim, rch * np.pi * R**2))
         print('Total model computed inflow:  {1:8.5g} {0}'.format(dim, sum(Out.Q.ravel()[Out.Q.ra
         print('Total model computed outflow: {1:8.5g} {0}'.format(dim, sum(Out.Q.ravel()[Out.Q.ra
```

```
The total net outflow should match the hand computed total recharge:
By hand: rch * pi * R**2  =      17671 m3/d
Total model computed inflow:     17676 m3/d
Total model computed outflow:   -17676 m3/d
```

A well in a semi-confined aquifer

We can model semi-confined flow in a single aquifer with one layer for the confining unit on top and one or more for the aquifer below. Because the analytical solutions for flow to wells do not consider vertical resistance within the aquifer, it suffices to use a single layer for the aquifer to compare our model with the analytical solution.

The steady-state flow to a well in a semi-confined aquifer is governed by

$ s = :raw-latex:‘\frac ‘Q {2 :raw-latex:‘pi *kD} :raw-latex:*frac { K_0 left(frac r lambda right) } { frac {r_0} lambda K_1 left(frac {r_0} lambda right) }‘ $

where $:raw-latex:‘\frac {r_0}‘ :raw-latex:‘\lambda ‘K_1 :raw-latex:‘left‘(:raw-latex:‘\frac {r_0}‘ :raw-latex:‘\lambda ‘:raw-latex:‘right‘) :raw-latex:‘\rightarrow ‘1 $ for $ r_0 << :raw-latex:‘lambda ‘$.

With $ r_0 $ the radius of the well, $:raw-latex:‘\lambda ‘= :raw-latex:‘sqrt {left(kDc right)}‘ $, `c` [T] is the vertical hydraulic resistance of the confining unit and `kD` [L2/T] the transmissivity of the underlying aquifer.

```
In [15]: #aquifer
         c = 250. # d, vertical hydraulic resistance of the confining unit
         k1 =  20. # m/d, horizontal hydraulic conductivity of the aquifer
         d =  10. # m, thickness of the confining unit
         D =  50. # m, thickness of teh aquifer
         k0 = 0.5 * d/c # m/d, vertical hydraulic conductivity of confining unit
                        # 0.5 is used because water in the model enters at center of layer, not a
         kD = k1 * D # m2/d, transmissivity of aquifer
         lam = np.sqrt(kD * c) # characteristic or spreading length of the semi-confind aquifer s

         Q = 1200 # m3/d, extraction by well (we use -Q for extraction in model)

         r0 = 5.0 # m, radius of the well
         z0 = 0 # m, ground elevation, top of confining unit
         z = z0 - np.array([d+D, 0, d])
         y = np.array([-0.5, 0.5])  # m, a one m thick model (ignored when axially symmetric)
         x = np.hstack((0, 1.01*r0, np.logspace(np.log10(r0), np.log10(5*lam), 51)))
         #x = np.hstack((0.99*r0, np.logspace(np.log10(r0), np.log10(5*lam), 51))) # inject exact

         gr = mfgrid.Grid(x, y, z, axial=True)

         k = gr.const(k1); k[0, : ,:] = k0
         FQ = gr.const(0.);  FQ[-1, :, 0] = -Q
         FH = gr.const(0.)
         IBOUND = gr.const(1); IBOUND[0, :, :] = -1

         Out = fdm_d.fdm3(gr, k, k, k, FQ, FH, IBOUND)

         # analytical stuff
         import scipy
         K0 = scipy.special.k0 # bessel function
         K1 = scipy.special.k1 # bessel function
         # analytical solution
         fi = -Q/(2 * np.pi * kD ) * K0(gr.xm / lam) / ((r0/lam) * K1(r0/lam))

         plt.figure()
         plt.xlabel('r [m]')
         plt.ylabel('head change [m]')
         plt.title('Drawdown by well in semi-confined aquifer, $ r $ on linear scale')
         plt.plot(gr.xm, fi, 'r', label='analytic')
         plt.plot(gr.xm, Out.Phi[-1, 0,:], 'bo', label='numeric')
         plt.legend()

         plt.figure()
         plt.xlabel('r [m]')
         plt.ylabel('head change [m]')
         plt.title('Drawdown by well in semi-confined aquifer, $ r $ on log scale')
         plt.setp(plt.gca(), 'xscale','log')
         plt.plot(gr.xm, fi, 'r', label='analytic')
         plt.plot(gr.xm, Out.Phi[-1, 0, :], 'bo', label='numeric')
         plt.legend()
```

```
Running in axial mode, y-values are ignored.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Out[15]: <matplotlib.legend.Legend at 0x111dc2cc0>
```

The numerical and analytical solutions agree. Only the first point differs. This is because the flow was specified in the center of the first cell (between r=0 and r=r0), instead of at the border of that cell, i.e. at exactly r=r0. We can solve this in various ways. The best way is to make the first cell coordinates r0-delta and r0+delta, with delta some small number, so that the water is injected almost exactly at r=r0. I leave that as an exercise.

Conclusion

We now have a flexible 3D steady-state finite difference model which can be used to solve fully 3D problems as well as 2D problems and 1D problemns. It also can solve axisymmetrical problems, i.e. axisymmetical cross sections. With this model one can also readily solve steady-state pumping tests in multilayer aquifers. But to show the results of such simulations it is useful to not only show the heads but also the stream function, i.e. the stream lines. This is the subject of the next chapter.

CHAPTER 8

Stream lines

Prof. dr.ir. T.N.Olsthoorn

Heemstede, 21 Oct 2016, 24 May 2017

Stream lines

Streamlines are lines of constant stream-function value. This differs from flow paths, which we will tackle later on. Flow lines of flow paths are lines that are followed by particles. Because in a dynamic model the flow conditions will normally vary with time, such flow lines are not unique, i.e. particles injected at the same spot but at a different time will follow different tracks and must be traced. Dit is not the case with streamlines. Streamlines (if they exist) can be computed by contouring the `stream function`, without any tracing. However, the stream function is only defined in 2D steady-sate flow without sources and sinks (and without leakage or recharge for that matter). It is said that the flow must be 2D and divergence free, which can be mathematically expressed as

$ **:raw-latex:‘\frac {\partial q_x} {\partial x}‘** + **:raw-latex:‘\frac {\partial q_y} {\partial y}‘** = 0 $

Concrete sinks and source cause so-called branch-cuts in the stream function. In practice, these can be dealt with in way that they will resamble vertical wells in the 2D image, which is often what they are.

Steamlines are especially useful in cross sections, either flat or axisymmetrical. This is because in such cross sections flow tends to be essentially 2D and divergence of flow is virtually absent. Even in transient flow situations, the divergence of fow due to elastic storage in a cross sectionis so small relative to the local specific discharge, that it can be practically ingored, meaning that also in transient flow in cross sections the stream function may be used, yielding a continuously changing streamline pattern along with the development of the flow, which, in turn lends itself of animation.

Where the conditions are fulfilled (divergence free 2D flow), showing streamlines as contours of the stream function is efficient manner to show the flow in a quantitative way. The beauty and the power of the stream function is that it provides a complete spacial picture of the actual flow. Given the function, the specific discharge at any point in the system is known as well as the total discharge between any pair of arbitrarily chosen point. As a plus, the stream function can readily be computed from the results of the finite difference model.

The stream function

The steam function is total discharge between a streamline taken as reference and any point in the model. And a sream line is a line on which the stream function has the same value. Hence stream lines are contours of the

stream function and the have a value, that of the stream function at the line.

One can draw an arbitrary line between the reference or zero streamline and such a point, and the total discharge across this line is given by the value of the stream function. There is, of course, also a stream line through this arbitrarily chosen point. And it follows that the discharge between this streamline and the reference stream line is the same everywhere in the model. This is true for any pair of stream lines.

While the reference stream line, the one that we will give the stream function value zero, can be chosen arbitrarily, we should taken one that we know beforehand. In practical modeling that is generally so for the bottom of our model, when it is closed. And if it is not closed, but we know how much discharge enters the bottom we can select the point to the left end of the bottom as zero and compute the stream function along the bottom before hand. Hence chosing the bottom of our model to have a stream function value zero is most often a good choice.

Remember we're dealing with cross sections. That is in practice we deal with the zx plane of our model.

Mathematically we can obtain the stream function, indicated by $, :raw-latex:'Psi '$, by integrating the horizontal specific discharge from the bottom of our model upward to any elevation:

$$\Psi = \int_{z_{min}}^{z_{max}} (y)\ dy$$

The finite differnce model yields among others the flows across the cell faces. So to obtain the total fow between the bottom of the model and the top of the lowest layer, we just have the lowest layer of `Qx`. When we accumulate `Qx` from the bottom of the model upwards we obtain the stream function values in all cell corners, and not in the cell centers, of the cross section. The stream function for a cross section of a model can, therefore, be easily computed from the infacial flows

$$\Psi = \sum_{0}^{n} Q_{x_i}$$

Where $ Q_{x_i} $ is the interfacial flow in `x`-direction at some given `x`-coordinate, 0 the index for the bottom plane of the model and `n` the index of plane `n` of the model.

Notice that the dimension of $ **:raw-latex:'\Psi '$ is [L2/T] for a flat cross section or layer and [L3/T] for an axially symmetric cross section. When contouring the stream function $:raw-latex:'**Psi '$, we should always denote the difference between two adjacent contours, i.e. the stream lines, in the title together with its dimension. With this, the flow in such a picture is fully determined by the stream line contours.

The implemenation follows next.

The stream function implemented

```
In [1]: ## This function is merged with the file fdm_d.py into fdm.py
        # So import fdm.py to use it with fdm3

        import numpy as np
        def psi(Qx, row=0):
            """Returns stream function values in z-x plane for a given grid row.

            The values are at the cell corners in an array of shape [Nz+1, Nx-1].
            The stream function can be vertically contoured using gr.Zp and gr.Xp as
            coordinates, where gr is an instance of the Grid class.

            Arguments:
            Qx --- the flow along the x-axis at the cell faces, excluding the outer
                   two plains. Qx is one of the fields in the named tuple returned
                   by fdm3.
            row --- The row of the cross section (default 0).

            It is assumed:
               1) that there is no flow perpendicular to that row
```

```
    2) that there is no storage within the cross section
    3) and no flow enters the model from below.
The stream function is computed by integrating the facial flows
from bottom to the top of the model.
The outer grid lines, i.e. x[0] and x[-1] are excluded, as they are not in Qx
The stream function will be zero along the bottom of the cross section.

"""
Psi = Qx[:, row, :] # Copy the section for which the stream line is to be computed.
                    # and transpose to get the [z,x] orientation in 2D
Psi = Psi[::-1,:].cumsum(axis=0)[::-1,:]          # cumsum from the bottom
Psi = np.vstack((Psi, np.zeros(Psi[0,:].shape))) # add a row of zeros at the bottom
return Psi
```

The function expects a 3D array as Qx in from the finite difference model and the row number that will be considered the cross sectional plane for which the $:raw-latex:‘Psi ‘$ is to be computed. It is the user’s responsibility to make sure that the requiremenst for the stream function to make sense are fulfilled.

The function can also take a 2D array. But this option is seldom used, because in our finite difference framwork, the model is always 3D, even when it only has one row. We always keep the z-direction vertical, thus preventing a lot of confusion.

The function can be easily adapted to compute the stream function for planes along columns. But this option is also seldom used, so it is left out.

Examples

Two examples will be worked out: a building pit which shows how detailed vertical flows are computed in a practical case to optimized pumping. It also demonstrates how one can readily switch between flat and axially symmetric cross section and it demonstrates the stream lines.

The second example shows the heads and stream lines in a confined aquifer with a partially penetrating screen that extracts water. This model is also axisymmetric. It also compares the heads from the numeric models with those from an analytical solution.

Smart pumping below a building pit: a flat and an axially symmetric model

The first example will consider a building pit which can both be computed in a flat cross section and an axially symmetric cross section. The subsurface consists of a 5 m thick semi-confining unit atop a 20 m thick aquifer atop a 10 m thick semi-confining layer atop a 30 m thick second aquifer. The building pit, which has a width of 20 m, a radius of 10 m, and is surrounded by impervious sheet piling to a depth of 12 m. Wells with 5 m long screens are installed inside the sheet piling immediately below the top confining layer. The objective is to design the necessary extraction to make sure that the head below the building pit is lowered by 5 m.

The properties are as follows

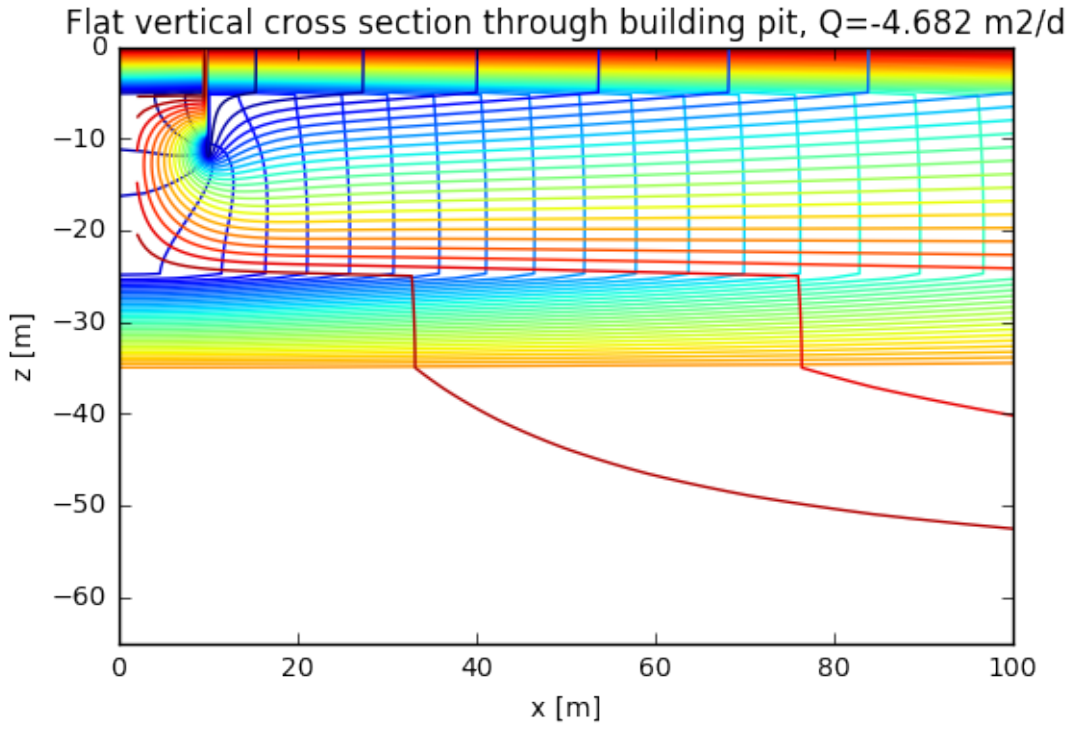
```
In [2]: import sys

        myModules = './modules/'

        if not myModules in sys.path:
            sys.path.append(myModules)

        #%matplotlib notebook
        import matplotlib.pylab as plt
        import numpy as np
        import fdm_d
        import mfgrid
        import mfetc
        #import mfexceptions as err
```

```
def inpoly(...)
```

```
In [3]: axial = False

        # aquifer
        d1 = 5.   # m, thickness of top confining layer
        c1 = 250. # d, vertical hydraulic resistance of top confining unit
        D1 = 20.  # m, thickness of first aquifer
        k1 = 10. # m/d, hydralic conductivity of first aquifer
        d2 = 10. # m, thickness of second confining layer
        c2 = 500. # d, vertical hydraulic resistance of the second confining layer
        D2 = 30. # m, thickness of second aquifer
        k2 = 25. # m/d, hydraulic conductivity of the second aquifer

        # coordinates
        R = 10.0 # m, half width or radius of building pit
        RR = 2500 # m, extent of model

        z0 = 0 # m, ground surface elevation
        z = z0 - np.cumsum(np.array([0, d1, D1, d2, D2])) # one model layer per system layer
        z = np.arange(z[0], z[-1]-0.5, -0.5) # refined vertical grid
        y = np.array([-0.5, 0.5])
        x = np.hstack([0, R - np.logspace(-1, np.log10(R), 21), R-0.1,  R + np.logspace(-1, np.lo

        gr = mfgrid.Grid(x, y, z, axial)

        # specifying k for all layers (We do this sequentially)
        k = gr.const(d1/c1)
        k[0:1, :, :] = 0.5*d1/c1  # fixed head in center of top layer
        k[gr.ZM < z0-d1      ] = k1
        k[gr.ZM < z0-d1-D1   ] = d2/c2
        k[gr.ZM < z0-d1-D1-d2] = k2

        # sheet piling
        kp  = 1e-7 # m/d, k sheet piling
        xpl =  9.9 # m, left of sheet piling
        xpr = 10.1 # m, right of sheet piling
        zpt =   0. # m, top of sheet piling
        zpb = -12. # m, bottom of sheet piling
        Ipiling= gr.inblock((xpl, xpr), None, (zpt, zpb))

        # well screens
        hWells = -5.0 # m, head in the wells
        xwl =  9.6 # m, left of well screens
        xwr = xpl  # m, right of well screens
        zwt = -5.  # m, top of well screens
        zwb = -10. # m, bottom of well screens
        Iwells = gr.inblock((xwl, xwr), None, (zwb, zwt))

        # required system arrays
        FQ = gr.const(0) # prescribed inflows
        FH = gr.const(0) # prescribed heads
        IBOUND = gr.const(1) # boundary array
        IBOUND[0, :, :] = -1 # top of model has prescibed heads

        # adaptation to piling and wells
        k[ Ipiling] = kp  # set k sheet piling to its conductivity
        FH[Iwells] = hWells # head in wells to hWells
        IBOUND[Iwells] = -1 # mark Wells as fixed heads

        # run the fdm model
        Out = fdm_d.fdm3(gr, k, k, k, FQ, FH, IBOUND)

        # compute total extraction
```

```
        Qout = np.sum(Out.Q[Out.Q<0]) # total outflow
        Qin  = np.sum(Out.Q[Out.Q>0]) # must match total inflow

        print('Total outflow = {0:.5g} {1}'.format(Qout,'m3/d' if axial else 'm2/d'))
        print('Total inflow  = {0:.5g} {1}'.format(Qin, 'm3/d' if axial else 'm2/d'))

        # show the results
        plt.figure()
        plt.setp(plt.gca(), 'xlim',[0, 100])
        plt.ylabel('z [m]')
        if axial:
            dim = 'm3/d'
            plt.xlabel('r [m]')
            plt.title('Axisymmetric cross section throug building pit, Q={0:.4g} {1}'.format(Qout,
        else:
            dim = 'm2/d'
            plt.xlabel('x [m]')
            plt.title('Flat vertical cross section through building pit, Q={0:.4g} {1}'.format(Qou

        # show head contours (50 contours)
        plt.contour(gr.xc, gr.zc, Out.Phi[:, 0, :], 50)

        # compute stream lines
        Psi = psi(Out.Qx)

        # show stream lines (30 stream lines)
        plt.contour(gr.Xp, gr.Zp, Psi, 30)
        plt.show()
```

```
Total outflow = -4.6819 m2/d
Total inflow  = 4.6819 m2/d
```



The figure shows only the results near the building pit (see xlim setting).

The figure shows both the head contours and the stream lines, which, if the horizontal and vertical scales of the figure are the same, are perpendicular to each other because the aquifer material is isotropic, `kx=kz`.

One can now immediately change from axially symmetric flow to flow in a flat cross section by changing the

variabl `axial` above. This also leads to the correct total extraction and dimension in the title of the figure.

The stream lines clearly show how the attracted water flows underneath the practically impervious sheet piling towards teh wells that are just inside it below the building pit. One can now readily verify the effect of the length and the permeability of the sheet piling on the discharge from the building pit required to maintain the target head below it.

There is an unlimited number of variations possible.

Notice the way that the total discharge was computed. Also notice how the locaton of the sheet piling and the wells were specified using the method `gr.inblock`, which allows to obtain a logical array with True where cell centers are inside the specified block.

A partially penetrating well, analytic verification

The next example is a well with a screen that partially penetrates the aquifer. For this situation there exists an analytical solution, which allows us to verify out numerical code. The solution describes the deviation of the head caused by the paratial penetration of the screen relative to the head loss caused by a fully penetrating screen. The solution was published by Hantush and can be found in the book

```
Kruseman, GP and De Ridder, NA (1994) Analysis and Evaluation of Pumping Test Data.
↪ Pudoc, Wageningen. Also available on the internet
```

$$\Delta s_{pp} = \frac{Q}{2\pi kD} \times \frac{2D}{\pi d} \sum_{n=1}^{\infty} \frac{1}{n} \left[\sin\left(\frac{n\pi b}{D}\right) - \sin\left(\frac{n\pi a}{D}\right) \right] \cos\left(\frac{n\pi z}{D}\right) K_0\left(\frac{n\pi r}{D}\right)$$

Where: * `D` : [L], aquifer thickness * `d` : [L], screen length * `a` : [L], distance between bottom of screen and bottom of aquifer * `b` : [L], distance between top of screen and top of aquifer * `z` : [L], distance/elevattion above bottom of aquifer * `r` : [m], distance to center of well

The figure below shows the layout, but notice that z_1 and z_2 in the figure are a and b respectively.

Fig. 8.1: Partial Penetration

The distances may also be taken relative to the top of the aquifer

We just have to add this extra drawdown (both positive and negative) to the drawdown due to a fully penetrating screen. Because the influence by partial penetration does not reach beyond about 1.5 times the aquifer thickness (ingnoring strong vertical anisotropies), it is valid to do so not only for confind aquifers for which the solution mathematically hold,but also for both semi-confined aquifers and transient situations. For semi-confined aquifers, the leakage through confining units with the area influenced by partial penetration is usually negligible relative to the total. For this to be true, the aquifer thickness should be much smaller than the spreading length $ **:raw-latex:‘\lambda ‘= :raw-latex:‘**sqrt { kDc }‘ >> D $. For transient situations, the change of elastic storage within the zone influenced by partial penetration is also generally negligible realtive to that from larger distances. This is true in unconfined aquifers, where elastic storage is orders of magnitude smaller than elastic storage. It is also true byond a very short time after the start of pumping, when ever more water is released from storage at larger distances than the zone influenced by partial penetration. The actual differences, may, of course, be studied by a detailed transient numerical model.

Below we set-up an axially symmetric model with a patially penetrating screen in a confined aquifer and compare the influence of partial penetration with the analytical solution.

```
In [4]: # The analytical solution
```

```
        def pp(gr, a, b, D, tol=1e-2):
            """Returns pp, the extra drawdown factor due to partial penetration

            To compute the head difference between a fully and a partially penetrating
            well use dpp = Q/(2pikD) pp
            pp is computed according to Hantush (see e.g. Kruseman and De Ridder in their
            book on the evaluation of pumping test data, available for free online)

            Parameters:
            -----------
            `gr` : mfgrid.Grid object,
                See mfgrid module, gr holds FDM grid coordinates
            `a` : float
                elevation of screen bottom above bottom of aquifer
            `b` : float
                elevation of screen top above bottom of aquifer
            `D` : float
                thickness of the aquifer

            Returns:
            --------
            pp : ndarray, float
                the extra drawdown factor due to partial penetration according to
                Hantush (see Kruseman and De Ridder)
                pp only depends on the geometry of screen relative to the aquifer

            TO 181022
            """
            from scipy.special import k0 # bessel function
            import mfexceptions as err

            d = b-a # screen length
            if d<=0:
                raise err.InputError("","b (sreeen top) must be larger than a (screen bottom)")

            Dpp = np.zeros(gr.shape)

            Nmax = 400 # max cycles in for loop
            crit = np.zeros(Nmax) # store criterion, as it fluctuates heavily between successive

            for n in range(1, Nmax):
                dpp = (np.sin(n * np.pi * b/D) - np.sin(n * np.pi * a/D)) *\
                        np.cos(n * np.pi *gr.ZM/D) * k0(n * np.pi * gr.XM/D) / n
                Dpp += dpp
                n1 = max(n-10,0)
                crit[n] = np.max(np.abs(dpp).ravel())
                #print('{0:4} : {1:10g}'.format(n, np.max(np.abs(dpp).ravel())))
                if n>10 and np.sum(crit[n1:n])<tol:
                    break
            print("Partial penetration computed in {0} iteration".format(n))
            return (2 * D / (np.pi * d)) * Dpp
```

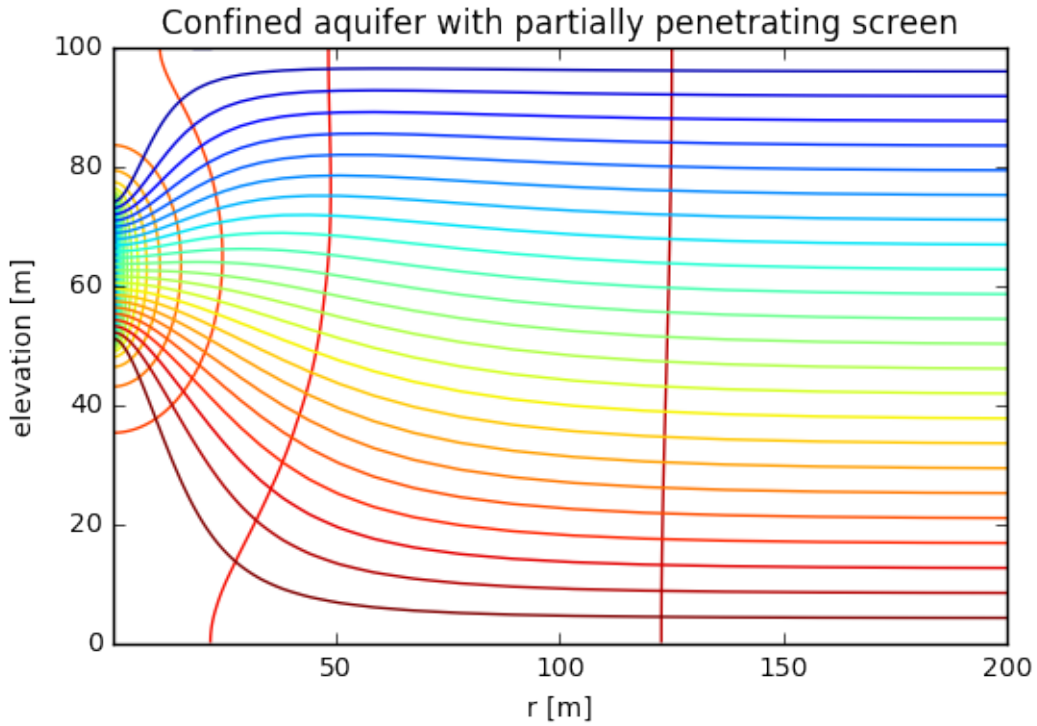
In [5]:
```
        """ For convenience of comparing with the analytical solution,
        all elevations are taken with respect to the bottom of the aquifer.
        """
        import matplotlib.pylab as plt
        import numpy as np
        import mfgrid
        import fdm_d
        import pdb

        Q  = -1200. # m3/d, extraction by screen

        D  = 100.  # m, thickness of the aquifer
```

```
        d  =  25.  # m, screen length

        z0 = D     # m, elevation of top of model
        b  = 75.   # m, elevation of screen top (relative to above bottom of aquifer)
        a  = b - d # m, elevation of screen bottom

        dz = 0.5   # m, thickenss of model layers

        r0 = 0.2   # m, well radius
        R  = 1000. # m, outer extent of model
        z = np.arange(D, 0-dz, -dz)
        x = np.hstack( ( 0.99 * r0, np.logspace(np.log10(r0), np.log10(R), 61), R-0.2 ) )
        y = np.array([-0.5, 0.5]) # ignored

        gr = mfgrid.Grid(x, y, z, axial=True)

        k = 10.0 # m/d, conductivity
        kD = k*D # m2/d, transmissivity

        K = gr.const(k)  # conductivity array
        FQ = gr.const(0) # fixed flows array
        FH = gr.const(0) # fixed heads array
        IBOUND = gr.const(1);   IBOUND[:, :, -1] = -1  # fix head at right hand side of model

        # layers with screen
        Iscr = np.logical_and(gr.zm > a, gr.zm < b)

        # introduce the well by dividing the total flow uniformely across its screen
        FQ[Iscr, 0,0] = Q/d * gr.dz[Iscr]
```

In [6]:
```
        # running the model, computing the stream function
        Out = fdm_d.fdm3(gr, K, K, K, FQ, FH, IBOUND)
        Psi = psi(Out.Qx)
```

```
Running in axial mode, y-values are ignored.
```

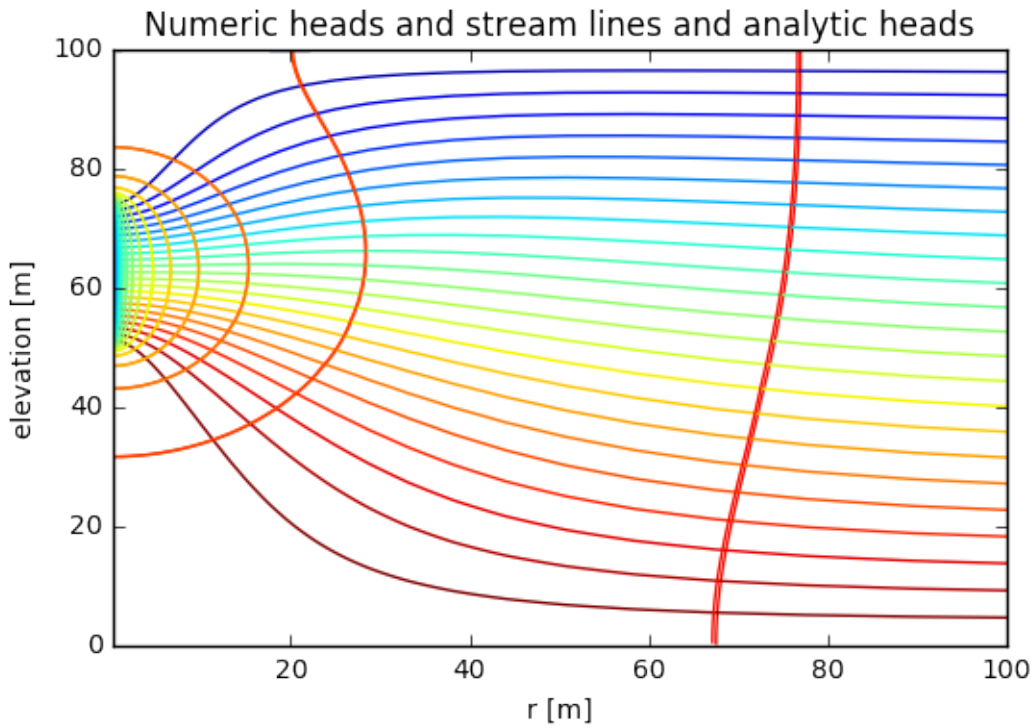
In [7]:
```
        plt.figure()
        plt.title('Confined aquifer with partially penetrating screen')
        plt.setp(plt.gca(), 'xlabel','r [m]', 'ylabel', 'elevation [m]', 'xlim', (gr.x[0], 200.))
        plt.contour(gr.xc, gr.zc, Out.Phi[:, 0, :], 25)
        plt.contour(gr.xp, gr.zp, Psi, 25)
        plt.show()
```



Below the same numerically computed contours are shown together with the analytically computed ones. As can be seen the difference is very small and can be attributed to small differences between the two methods, especially around the edges of the screen. The boundary conditions are the same,namely a fixed extraction per unit of screen length.

```
In [12]: # Compute the drawdown analytically as the superposition of
         # that of a fully penetrating well and the extra drawdown due to
         # partial penetration.

         dpp = pp(gr, a, b, D, tol=1e-2)

         s = Q/(2 * np.pi * kD) * (np.log(R/ gr.XM) + dpp)
         #s =  dpp

         # suitable set of levels for head contouring of this case
         cont = np.arange(0,-5,-0.25)[::-1]

         plt.figure()
         plt.title('Numeric heads and stream lines and analytic heads');
         plt.setp(plt.gca(), 'xlabel','r [m]', 'ylabel', 'elevation [m]', 'xlim', (gr.x[0], 100.),

         plt.contour(gr.xc, gr.zc, Out.Phi[:, 0, :], cont) # plot the numeric contours
         plt.contour(gr.xp, gr.zp, Psi, 25) # plot the stream lines
         plt.contour(gr.xm, gr.zm, s[:, 0, :], cont) # plot the analytic contours
         plt.show()
```

```
Partial penetration computed in 201 iteration
```



Conclusion

We have implemented the stream function and used it to obtain the stream lines as its contours. This works both for flat models and for axially symmetric ones. Together with the head contours, the stream lines give a complete picture of the flow.

As can be seen, the stream function jumps at and above the wells. The branchcut at this jump runs from the wells vertically to the top of the model and, therefore, the look like wells. The number of vertial stream lines in the branchcut is exactly equal to their extraction. The flow can be computed by counting stream lines, as between each pair of stream lines the total discharge is the same. The specific dischage at any point can be computed by dividing the discharge between two stream lines by the distance between them.

The second example compares the heads or rather the drawdowns in a confined aquifer with a partially penetrating well screen. It shows that the analytical and numerical dawdowns are virtually the same, as the respective lines almost exactly lie upon onanother.

The next chapter will deal with transient flow. Particle tracking is delayed to later.

CHAPTER 9

Transient flow

Prof. dr.ir. T.N.Olsthoorn

Heemstede, Oct. 2016, 24 May 2017

Theory

Transient flow, in fact, requires a relatively straightforward extension of our steady-state model fdm3. The water balance equation for an arbitrary single cell in our model becomes:

$$\int_t^{t+\Delta t} \left(\sum Q_{in} + Q_{ext} \right) dt = W_{t+\Delta t} - W_t$$

The the left side of the equation describes the total net inflow for this cell, divided in a sum that originates from the surrounding cells and the $ Q_{ext} $ denotes the total net inflow from the outside world (Injection, Recharge, Leakage etc.). It's clear that extractions are counted as negative injections. This total net inflow shoul be integrated over the considered time spand, i.e. $ t **:raw-latex:‘\rightarrow ‘t + :raw-latex:‘**Delta **‘**t $, during which the flows may vary in arbitrary ways. To the right we have the volume of water in the cell at the end of this period minus that at the start of this period. There is no approximation in this equation.

When it comes to our model, we will write the right hand side in terms of volume, storage coefficient $ S_S $, and head $ h $. The integral to the left is dropped by writing for the flow their average values during the considered period:

$$\left(\sum \overline{Q}_{in} + \overline{Q}_{ext} \right) \Delta t = S_S V \left(h_{t+\Delta t} - h_t \right)$$

where

$$V = \Delta x \Delta y \Delta z$$

the volume of the cell.

As we saw in an earlier chapter where we derived the finite differenc method, the flow from each neighbor indexed $ j $ into the considered cell with index $ i $ is formulated as

$$Q_{ji} = C_{ij}(h_j - h_i)$$

Thus, $ Q $ varies during the time step according to how the heads vary. The question then is, which $ h $ is the true average during the time step, such that, then it is used, we get the average flows during the time step and hence we can integrate by multiplying with $:raw-latex:‘Delta ‘t $.

The answer is, we don’t know. We only know for sure, that there exists some value of $ 0< :raw-latex:‘epsilon ‘< 1$ such that

$$Q_{t+\epsilon\Delta t} = \overline{Q}$$

But then, if we solve the system of equation, we wil end up with the head for this time, $ h_{t + **:raw-latex:‘\epsilon ‘:raw-latex:‘**Delta ‘t} $, which definitely differs from the head at the end of the time step, which we need to compute the storage during the time step as expressed in the equation above.

To solve this dilemma, we have to make an assumption, which is, that we assume that the time steps of our simulation are small enough to safely approximate the change of head during them as being linear. With this approximation, we have

$$h_{t+\epsilon\Delta t} = h_t + \epsilon(h_{t+\Delta t} - h_t)$$

or

$$h_{t+\Delta t} = h_t + \frac{h_{t+\epsilon\Delta t} - h_t}{\epsilon}$$

This implies that we can now replace the unknown head at the end of the time step by the one we actually compute during the time step, by writing

$$\left(\sum \overline{Q}_{in} + \overline{Q}_{ext}\right)_{t=t+\epsilon\Delta t} = S_S \frac{V}{\epsilon\Delta t}\left(h_{t+\epsilon\Delta t} - h_t\right)$$

If we just realize that all flows and heads will be compute that $ t + **:raw-latex:‘\epsilon ‘:raw-latex:‘**Delta ‘t $, we can omit this index and write and reorder keeping the unknown $ Q_{in} $ to the left and putting all known $ Q_{ext} $ to the right, positive when entering the cell, we obtain:

$$-\sum Q_{in} = Q_{ext} - S_S \frac{V}{\epsilon\Delta t}\left(h - h_t\right)$$

We see that $ h_t $ is known, because it is the head in the considered cell at the beginning of the considered time step, i.e. at the end of the last time step. Therefore, we can rearrange to have all known values to the right of the equal sign and all unknowns at the left. This gives

$$-\sum Q_{in} + S_S \frac{V}{\epsilon\Delta t} h = Q_{ext} + S_S \frac{V}{\epsilon\Delta t} h_t$$

Just remember that the first term, $ -:raw-latex:‘sum ‘Q_{in} $ can be written out as a vector product, this becomes

$$\left[\begin{array}{cccc} -C_E, & \ldots & -C_B, & +\left(C_E + \ldots + C_B\right) \end{array}\right] \left[\begin{array}{c} h_E \\ h_W \\ h_N \\ h_S \\ h_T \\ h_B \\ h \end{array}\right] + S_S \frac{V}{\epsilon\Delta t} h = Q_{ext} + S_S \frac{V}{\epsilon\Delta t} h_t$$

To bring the right-most term of the left-hand side of this equation under the vector product, we only have to put the the factor in front of $ h $ to the sum of coefficients like so:

$$\left[\begin{array}{cccc} -C_E, & \ldots & -C_B, & +\left(C_E + \ldots + C_B + S_S \frac{V}{\epsilon\Delta t}\right) \end{array}\right] \left[\begin{array}{c} h_E \\ h_W \\ h_N \\ h_S \\ h_T \\ h_B \\ h \end{array}\right] = Q_{ext} + S_S \frac{V}{\epsilon\Delta t} h_t$$

We see that we have now a water balance equation for the transient cell that has exactly the same structure as the one we developed for the steady-state case, with unknowns at the left and the known flows on the right. Notice that the second term to the right has the same dimension as $Q_{ext} $, namely [L3/T]. The include the storage due to the unknown head, it suffices to put the coefficient $ S_S V / (**:raw-latex:'\epsilon ':raw-latex:'**Delta **'**t) $ in the coefficient of the requested node to the left. It doesn't change any of the other coefficients.

If we compare this with the first chapter, in which it was shown how exchange between model and a fixed head in the outside world through resistance was implemented, we see that it's the same with implementation of transient flow. The conducntace is added to the matrix coefficient on the left, while the known part C_{ext}h_{ext} is kept on the right side, and it too has dimension [L3/T].

Consedering the entire model, we have such an equation for each and every cell in it. These equations are combined into a system of equations

$$\mathbf{A} \times \mathbf{h}_{t+\epsilon\Delta t} = \overline{\mathbf{Q}}_{ext} + \mathbf{S}_S \frac{\mathbf{V}}{\epsilon\Delta t}\mathbf{h}_t$$

Where **:raw-latex:'\matbh{A}'** is the system matrix, in which the coefficient vector $ **:raw-latex:'\mathbf{S}'**_S **:raw-latex:'\mathbf{V}'**/(**:raw-latex:'\epsilon ':raw-latex:'**Delta **'**t) $ was added to its diagonal.

It is clear that these coefficents change with the lenght of the time step and, therefore the diagonal and the right-hand side of the equation have to be adapted with each new time step, if its length changes.

The dealing with fixed heads does not change and was explained earlier in the chaper "finite differnce modeling".

The answer that we obtain after having solved the system equation, is the head in all cells at time $ t + **:raw-latex:'\epsilon ':raw-latex:'**Delta *t $. Therefore we a small extra step to obtain the head $ h_{t + :raw-latex:*Delta **'**t} $ at the end of the time step

$$\mathbf{h}_{t+\Delta t} = \mathbf{h}_t + \frac{\mathbf{h}_{t+\epsilon\Delta t} - \mathbf{h}_t}{\epsilon}$$

The only point that we have still circumvented is the choice of $ 0:raw-latex:*epsilon'<1 $, the so called implicitness. It is the point in time expressed as the fraction of the current time-step length at which the flows and heads represent the average values during the entire time step. In fact, we don't know. If the head change is linear, then $:raw-latex:*epsilon'=0.5 $ would be exact. But when the head exponentially appoaches a new equilibrium a value $ **:raw-latex:'\epsilon '> 0.5 $ is better. Not only because heads will always thrive to a new equilibrium with time, but also because values of $:raw-latex:'**epsilon'<0.5 $ yield unstable solutions, we always choose a value larger than 0.5. The most drastic choice is $\epsilon = 1$, a choice which is said to make the model fully implicit. It conceptually assumes that the heads at the end of the current timestep well represent their average values during the time step. It may not be the most accurate value, especially with larger time steps, but it makes the model rock stable. In fact this is the choice that the world's most used finite difference model, MODFLOW, implicitly makes.

Is it a good or a bad choice? Well it's not bad as errors due to this time discretization well damp out during subsequent time steps. Nevertheless, we will keep the value of $:raw-latex:'epsilon '$ explicit, so as to allow investigating the sensitivity of the outcomes of the model for it.

Procedure

We compute the required coefficients for each time step as follows:

Compute

for non-axially symmetric models:

$$\mathbf{C}_S = \frac{\mathbf{S}_S \mathbf{\Delta x \Delta y \Delta z}}{\epsilon\Delta t}$$

for axially symmetric models:

$$\mathbf{C}_S = \frac{\mathbf{S}_S \pi \left(\mathbf{x}[1:]^2 - \mathbf{x}[:-1]^2\right) \Delta z}{\epsilon\Delta t}$$

Add this vector $ **:raw-latex:'\mathbf{C}'**_S $ explicitly to the diagonal of the system matrix at each time step.

Compute

$$\mathbf{C}_S \mathbf{h}_t$$

at the RHS of the system equation for each time step.

We solve for $:raw-latex:‘\mathbf{h}‘_{t + :raw-latex:‘\epsilon ‘:raw-latex:‘Delta ‘t} $

$$\mathbf{h}_{t+\epsilon\Delta t} = \mathbf{A}^{-1} \times (\mathbf{Q}_{ext} + \mathbf{C}_S\mathbf{h}_t)$$

Finally compute

$$\mathbf{h}_{t+\Delta t} = \mathbf{h}_t + \frac{\mathbf{h}_{t+\epsilon\Delta t} - \mathbf{h}_t}{\epsilon}$$

And just before computing the new time step, update the know $\mathbf{h}_t$ with the value just computed

$$\mathbf{h}_t = \mathbf{h}_{t+\Delta_t}$$

Implementation

We have to adapt our fdm3 model at a few places. First is its signature. We need two extra inputs, namely t and $:raw-latex:‘\mathbf{S}‘_S and a possibility to adapt the implicitness $:raw-latex:‘\epsilon ‘$. Calling this model ‘‘fdm3t‘‘ we have

Out = fdm3t(gr, t, kx, ky, kz, Ss, FQ, FH, IBOUND, epsilon=0.67)

Where the output Out contains the computed arrays, see doc string of implementation below.

The heads are at the end of the time steps not including the intial heads. The flows [L3/T] are average flows during each time step.

All these array are, therefore, 4 dimensional, 3 spacial dimensions and and the fourth being time. The shape of the heads, $ Q $ and $ Qs $ arrays are, therore, $ (Ny, Nx, Nz, Nt) $

The new $ Qs $ is the flow [m3/T] during the time step that enters the cell from storage (because all flows are positive when they enter a cell). Thus Qs is positive when the head declines, yielding water from storage to the cell that then flows towards surrounding cells (or to the external world).

We only use the specific storage of each cell. This can be easily computed from the total storage and even from the specific yield when required.

What we have not (yet) implemented are non-lineairities like change of transmissivity of the model and, therefore, its cells under transient unconfined conditions. To do this is not difficult. It requires updating the transmissivities of the cells after a given number of so-called inner iterations. Each such adaptation is called an outer iterations. Once the head and or flow changes after an out iterations have become negligible, the model is said to have converged and the outcomes are used. Notice, however, that convergence is not always guaranteed for non-linear systems that are solved in terms of linear equations that are updated like it is done here as well as in MODFLOW. Special solvers can do a better job by adding non-linear Newton-Raphson schemes to the solver. MODFLOW.NWT and the new MODFLOW.USG have such a solver, which may be necessary for strongly non-linear models. These issues are beyond this course.

Implementation; the adepted module to include axial symmetry

In [8]:
```
%%writefile fdm_t.py

import numpy as np
import pdb
import scipy.sparse as sp
from scipy.sparse.linalg import spsolve # to use its short name
from collections import namedtuple

class InputError(Exception):
    pass
```

```
def unique(x, tol=0.0001):
    """return sorted unique values of x, keeping ascending or descending direction"""
    if x[0]>x[-1]:  # vector is reversed
        x = np.sort(x)[::-1]  # sort and reverse
        return x[np.hstack((np.diff(x) < -tol, True))]
    else:
        x = np.sort(x)
        return x[np.hstack((np.diff(x) > +tol, True))]


def fdm3t(gr, t, kx, ky, kz, Ss, FQ, HI, IBOUND, epsilon=0.67):
    '''Transient 3D Finite Difference Model returning computed heads and flows.

    Heads and flows are returned as 3D arrays as specified under output parmeters.

    Parameters
    ----------
    'gr' : `grid_object`, generated by gr = Grid(x, y, z, ..)
        if `gr.axial`==True, then the model is run in axially symmetric model
    t : ndarray, shape: [Nt+1]
        times at which the heads and flows are desired including the start time,
        which is usually zero, but can have any value.
    `kx`, `ky`, `kz` : ndarray, shape: (Nz, Ny, Nx), [L/T]
        hydraulic conductivities along the three axes, 3D arrays.
    `Ss` : ndarray, shape: (Nz, Ny, Nx), [L-1]
        specific elastic storage
    `FQ` : ndarray, shape: (Nz, Ny, Nx), [L3/T]
        prescrived cell flows (injection positive, zero of no inflow/outflow)
    `IH` : ndarray, shape: (Nz, Ny, Nx), [L]
        initial heads. `IH` has the prescribed heads for the cells with prescribed head.
    `IBOUND` : ndarray, shape: (Nz, Ny, Nx) of int
        boundary array like in MODFLOW with values denoting
        * IBOUND > 0  the head in the corresponding cells will be computed
        * IBOUND = 0  cells are inactive, will be given value NaN
        * IBOUND < 0  coresponding cells have prescribed head
    `epsilon` : float, dimension [-]
        degree of implicitness, choose value between 0.5 and 1.0

    outputs
    -------
    `Out` : namedtuple containing heads and flows:
        `Out.Phi` : ndarray, shape: (Nt+1, Nz, Ny, Nx), [L3/T]
            computed heads. Inactive cells will have NaNs
            To get heads at time t[i], use Out.Phi[i]
            Out.Phi[0] = initial heads
        `Out.Q`   : ndarray, shape: (Nt, Nz, Ny, Nx), [L3/T]
            net inflow in all cells during time step, inactive cells have 0
            Q during time step i, use Out.Q[i]
        `Out.Qs`  : ndarray, shape: (Nt, Nz, Ny, Nx), [L3/T]
            release from storage during time step.
        `Out.Qx   : ndarray, shape: (Nt, Nz, Ny, Nx-1), [L3/T]
            intercell flows in x-direction (parallel to the rows)
        `Out.Qy`  : ndarray, shape: (Nt, Nz, Ny-1, Nx), [L3/T]
            intercell flows in y-direction (parallel to the columns)
        `Out.Qz`  : ndarray, shape: (Nt, Nz-1, Ny, Nx), [L3/T]
            intercell flows in z-direction (vertially upward postitive)

    TO 161024
    '''

    import pdb

    # define the named tuple to hold all the output of the model fdm3
```

```
Out = namedtuple('Out',['t', 'Phi', 'Q', 'Qs', 'Qx', 'Qy', 'Qz'])
Out.__doc__ = """fdm3 output, <namedtuple>, containing fields `t`, `Phi`, `Q`, `Qs`,
                Use Out.Phi, Out.Q, Out.Qx, Out.Qy and Out.Qz
                or
                Out.Phi[i] for the 3D heads of time `i`
                Out.Q[i] for the 3D flows of time step `i`
                Notice the difference between time and time step
                The shape of Phi is (Nt + 1,Nz, Ny, Nx)
                The shape of Q, Qs is (Nt, Nz, Ny, Nx)
                For the other shapes see docstring of fdm_t
                """

if gr.axial:
    print('Running in axial mode, y-values are ignored.')

if kx.shape != gr.shape:
    raise AssertionError("shape of kx {0} differs from that of model {1}".format(kx.sh
if ky.shape != gr.shape:
    raise AssertionError("shape of ky {0} differs from that of model {1}".format(ky.sh
if kz.shape != gr.shape:
    raise AssertionError("shape of kz {0} differs from that of model {1}".format(kz.sh
if Ss.shape != gr.shape:
    raise AssertionError("shape of Ss {0} differs from that of model {1}".format(Ss.sh

active = (IBOUND>0).reshape(gr.Nod,)  # boolean vector denoting the active cells
inact  = (IBOUND==0).reshape(gr.Nod,) # boolean vector denoting inacive cells
fxhd   = (IBOUND<0).reshape(gr.Nod,)  # boolean vector denoting fixed-head cells

# reshaping shorthands
dx = np.reshape(gr.dx, (1, 1, gr.Nx))
dy = np.reshape(gr.dy, (1, gr.Ny, 1))

# half cell flow resistances
if not gr.axial:
    Rx1 = 0.5 *    dx / (   dy * gr.DZ) / kx
    Rx2 = Rx1
    Ry1 = 0.5 *    dy / (gr.DZ *    dx) / ky
    Rz1 = 0.5 * gr.DZ / (   dx *    dy) / kz
else:
    # prevent div by zero warning in next line; has not effect because x[0] is not use
    x = gr.x.copy();  x[0] = x[0] if x[0]>0 else 0.1* x[1]

    Rx1 = 1 / (2 * np.pi * kx * gr.DZ) * np.log(x[1:] /  gr.xm).reshape((1, 1, gr.Nx))
    Rx2 = 1 / (2 * np.pi * kx * gr.DZ) * np.log(gr.xm / x[:-1]).reshape((1, 1, gr.Nx))
    Ry1 = np.inf * np.ones(gr.shape)
    Rz1 = 0.5 * gr.DZ / (np.pi * (gr.x[1:]**2 - gr.x[:-1]**2).reshape((1, 1, gr.Nx)) +

# set flow resistance in inactive cells to infinite
Rx1[inact.reshape(gr.shape)] = np.inf
Rx2[inact.reshape(gr.shape)] = np.inf
Ry1[inact.reshape(gr.shape)] = np.inf
Ry2 = Ry1
Rz1[inact.reshape(gr.shape)] = np.inf
Rz2 = Rz1

# conductances between adjacent cells
Cx = 1 / (Rx1[  :, :,  :-1] + Rx2[:, : , 1:])
Cy = 1 / (Ry1[  :, :-1,:  ] + Ry2[:, 1:, : ])
Cz = 1 / (Rz1[:-1, :,  :  ] + Rz2[1:, :, : ])

# storage term, variable dt not included
Cs = (Ss * gr.Volume / epsilon).ravel()
```

```
# cell number of neighboring cells
IE = gr.NOD[  :, :,   1:] # east neighbor cell numbers
IW = gr.NOD[  :, :,  :-1] # west neighbor cell numbers
IN = gr.NOD[  :, :-1,  :] # north neighbor cell numbers
IS = gr.NOD[  :, 1:,   :] # south neighbor cell numbers
IT = gr.NOD[:-1, :,    :] # top neighbor cell numbers
IB = gr.NOD[ 1:, :,    :] # bottom neighbor cell numbers

R = lambda x : x.ravel()  # generate anonymous function R(x) as shorthand for x.ravel

# notice the call  csc_matrix( (data, (rowind, coind) ), (M,N))  tuple within tupple
# also notice that Cij = negative but that Cii will be postive, namely -sum(Cij)
A = sp.csc_matrix(( np.concatenate(( R(Cx), R(Cx), R(Cy), R(Cy), R(Cz), R(Cz)) ),\
                    (np.concatenate(( R(IE), R(IW), R(IN), R(IS), R(IB), R(IT)) ),\
                     np.concatenate(( R(IW), R(IE), R(IS), R(IN), R(IT), R(IB)) ),\
                  )), (gr.Nod,gr.Nod))

A = -A + sp.diags(np.array(A.sum(axis=1))[:,0]) # Change sign and add diagonal

#Initialize output arrays (= memory allocation)
Nt = len(t)-1
Out.Phi = np.zeros((Nt+1, gr.Nod)) # Nt+1 times
Out.Q   = np.zeros((Nt  , gr.Nod)) # Nt time steps
Out.Qs  = np.zeros((Nt  , gr.Nod))
Out.Qx  = np.zeros((Nt, gr.Nz, gr.Ny, gr.Nx-1))
Out.Qy  = np.zeros((Nt, gr.Nz, gr.Ny-1, gr.Nx))
Out.Qz  = np.zeros((Nt, gr.Nz-1, gr.Ny, gr.Nx))

# reshape input arrays to vectors for use in system equation
FQ = R(FQ);  HI = R(HI);  Cs = R(Cs)

# initialize heads
Out.Phi[0] = HI

# solve heads at active locations at t_i+eps*dt_i

Nt=len(t)  # for heads, at all times Phi at t[0] = initial head
Ndt=len(np.diff(t)) # for flows, average within time step

for idt, dt in enumerate(np.diff(t)):

    it = idt + 1

    # this A is not complete !!
    RHS = FQ - (A + sp.diags(Cs / dt))[:,fxhd].dot(Out.Phi[it-1][fxhd]) # Right-hand s

    Out.Phi[it][active] = spsolve( (A + sp.diags(Cs / dt))[active][:,active],
                                RHS[active] + Cs[active] / dt * Out.Phi[it-1][active])

    # net cell inflow
    Out.Q[idt]  = A.dot(Out.Phi[it])

    Out.Qs[idt] = -Cs/dt * (Out.Phi[it]-Out.Phi[it-1])


    #Flows across cell faces
    Out.Qx[idt] =  -np.diff( Out.Phi[it].reshape(gr.shape), axis=2) * Cx
    Out.Qy[idt] =  +np.diff( Out.Phi[it].reshape(gr.shape), axis=1) * Cy
    Out.Qz[idt] =  +np.diff( Out.Phi[it].reshape(gr.shape), axis=0) * Cz

    # update head to end of time step
    Out.Phi[it] = Out.Phi[it-1] + (Out.Phi[it] - Out.Phi[it-1]) / epsilon
```

```
            # reshape Phi to shape of grid
            Out.Phi = Out.Phi.reshape((Nt,) + gr.shape)
            Out.Q   = Out.Q.reshape( (Ndt,) + gr.shape)
            Out.Qs  = Out.Qs.reshape((Ndt,) + gr.shape)

            return Out # all outputs in a named tuple for easy access
```

```
Overwriting fdm_t.py
```

Examples

Here we'll work out a few axially symmetric examples to verify this model using analytical solutions.

We will also compute the darwdown in a multi-layer aquifer system.

We'll keep truly axially symmetric 3D flow for the next chapter, after we introduced the stream function.

Preparatory work

As always we set the path to our own modules and import the required modules and general packages.

Each time after we edited one or more of our modules, we have to `reload` them. This is why `reload` is imported.

```
In [2]: myModules = './modules/'

        #Adding the path to our modules to the pythonpath
        import sys
        if not myModules in sys.path:
            sys.path.append(myModules)
```

```
In [3]: # import the required general packages
        import numpy as np
        import matplotlib.pyplot as plt
        from importlib import reload
        import fdm_t    # first time import
        reload(fdm_t)   # in case we edited fdm_t above we need to reload

        # allow inline plotting
        %matplotlib notebook
```

```
In [4]: # import our own modules and packages, when they exist
        import fdm_t # from current directory
        import mfgrid  # path has been added to sys.path above
        import mfetc
        import mfexceptions as err
```

```
def inpoly(...)
```

```
In [9]: # when we have been editing files, make sure to reload
        reload(fdm_t)
        reload(mfgrid)
        reload(mfetc)
        reload(err)
```

```
def inpoly(...)
```

```
Out[9]: <module 'mfexceptions' from './modules/mfexceptions.py'>
```

A well in a confined (or unconfined) infinite aquifer (Theis)

The most famous analytic transient groundwater solution is that of the Theis well, a fully penetrating well in a homogeneous confined aquifer, that starts pumping at a constant rate at t=0. We will now use our model in axially

symmetric mode to compute this drawdown numerically and compare it with the analytical solution.

The analytical solution of the drawdown s Theis well reads

$$s = \frac{Q}{4\pi kD} W(u), \quad u = \frac{r^2 S}{4kDt}$$

$W(-)$ is called the Theis well function. It is a regular mathematical function known at the `exponential integral`

$$W(u) = \int_u^\infty \frac{e^{-y}}{y} dy$$

This exponential integral is available in Python as

from scipy.special import expi, defined as

$$expi(w) = \int_{-\infty}^{w} \frac{e^{\nu}}{\nu} d\nu$$

To convert the `Well` function `w` to the `expi` function as defined in Python.

$$W(u) = \int_{y=u}^{y=\infty} \frac{e^{-y}}{y} dy = \int_{\nu=-\infty}^{\nu=-u} \frac{e^{\nu}}{\nu} d\nu = \int_{y=\infty}^{y=w} \frac{e^{y}}{y} dy = expi(w) = expi(-u)$$

Which allows us to just use `expi(-u)` for the analytical solution. It may practially be implemented by defining an anonymous function (lambda function or marco) as follows:

W = lambda u: scipy.linalg.expi(-u)

```
In [6]: from scipy.special import expi
        def W(u): return -expi(-u)
        W(0.01) # check if it works
```

```
Out[6]: 4.0379295765381134
```

```
In [10]: #aquifer
         c = 250. # d, vertical hydraulic resistance of the confining unit
         k =  20. # m/d, horizontal hydraulic conductivity of the aquifer
         D =  50. # m, thickness of teh aquifer
         S = 0.001 # -, elastic storage coefficient of aquifer
         ss = S/D # 1/m, speciic elastic storage coefficient
         kD = k*D # m2/d, transmissivity
         Q = 1200 # m3/d, extraction by well (we use -Q for extraction in model)
         t = np.logspace(-3.,1.,51)
         Nt = len(t)
         Ndt = len(np.diff(t))

         r0 = 0.2 # m, radius of the well
         R  = 1e4 # m, extent of model, analytic solution had not external boundary

         z = np.array([0, -D])
         y = np.array([-0.5, 0.5])  # m, a one m thick model (ignored when axially symmetric)
         x = np.hstack((0.999 * r0, np.logspace(np.log10(r0), np.log10(R), 51)))

         gr = mfgrid.Grid(x, y, z, axial=True)

         K  = gr.const(k)
         Ss = gr.const(ss)
         FQ = gr.const(0.);  FQ[:, 0, 0] = -Q
         FH = gr.const(0.)
         IBOUND = gr.const(1) # no fixed heads
```

```
Out = fdm_t.fdm3t(gr, t, K, K, K, Ss, FQ, FH, IBOUND, epsilon=1.0)

# analytical stuff
import scipy
W = lambda u: -scipy.special.expi(-u)

u = (gr.xm**2 * S).reshape((1,gr.Nx)) / (4 * kD *t.reshape((Nt,1)))
fi = -Q / (4 * np.pi * kD ) * W(u)

# show results
plt.figure()
plt.setp(plt.gca(),'xscale','log')
plt.xlabel('r [m]')
plt.ylabel('head change [m]')
plt.title('Theis drawdown')

for it in range(1,Nt):
    plt.plot(gr.xm, fi[it], 'r')
    plt.plot(gr.xm, Out.Phi[it][-1, 0, :], 'b.', label='t={0:7.3g} d'.format(t[it]))
plt.legend()
```

```
Running in axial mode, y-values are ignored.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Out[10]: <matplotlib.legend.Legend at 0x10e174438>
```

The results show that the numerical model is accurate, except for the first two or three time steps and for the last few steps.

The error with the last few steps is obvious: out model is too small, causing the drawdown to bounce back from the closed outer boundary which as as effect an increased drawdown. As an exercise increas the outer boundary (by increasing the variable R) to see this.

One can compute the are of influence, that is the distance where the straigt drawdown line on log scale (as in the figure) hits zero. It can be derived from the analytical simplified solution of the logarithmic approximation of the Theis drawdown, which is

$$s \approx \frac{Q}{4\pi kD} \ln\left(\frac{2.25kDt}{r^2 S}\right)$$

which, of course, is zero for when the argument of the logarithm equals 1. This yields an expression for the radius of influence $ r_{inf} $

$$r_{inf} = \sqrt{\frac{2.25kDt}{S}}$$

which in our case is $ r_{inf} (t=1000 d) = 15000 m = 15 km $, while our model radius is only 10 km. The problem is solved by setting $R = 1e5$ m (100 km).

The deviations shortly after the start of the pomp are due to an our inaccurate initial head. We used zero, indeed, but hile this sounds correct, it is not optimal for this case. The best choice is to use the analytical solution for the first time (which then has to be > 0), and start the model from there. When we do this, the drawdown will match from the first to the last step. If we, however, obstinentely start with all heads equal to zero, then the model needs about 3 steps to get into line with the analytical solution, no matter how short the initial time step that we start with. But with this in mind, the analytical and numerical results coincide pratically perfectly.

In real cases, the start of the full extraction exactly at t=0 is physically impossible anyway, due to fact that both the pump and the water have to be accellerate initially, which takes perhaps a minute.

Well in the center of a circular island

We will now compute the same extraction using full fledged 2D spacial flat model to compare with the analytical solution.

```
In [11]: # anonymous function to extract index for given x-value
         ix = lambda x: int(np.floor(np.interp(x, gr.xm, np.arange(len(gr.xm)))))
         iy = lambda y: int(np.ceil (np.interp(y, gr.ym[::-1], np.arange(len(gr.ym))[::-1])))

         # aquifer
         S = 0.001 # [-], storage coefficient (either specific yield or elastic)
         k = 10.  # m/d
         D = 100. # m, thickness of the aquifer
         kD = k*D # m2/d, transmissivity of aquifer
         Q = -1200 # m3/d, extraction

         r0 = 0.1 # m, well radius
         R  =1e6 # m, exent of model

         # grid
         Npoints = 51
         x = np.logspace(np.log10(r0), np.log10(R), Npoints);  x = np.hstack((-x[::-1], x))
         y = np.logspace(np.log10(r0), np.log10(R), Npoints);  y = np.hstack((-y[::-1], y))
         z = np.array([0, -D])

         gr = mfgrid.Grid(x, y, z, axial=False)

         K = gr.const(k);       K[-1, iy(0.), ix(0.)] = 1000 * k # remove resistance from central
         Ss = gr.const(S/D)
         FQ = gr.const(0.); FQ[-1, iy(0.), ix(0.)] = Q
         FH = gr.const(0.)
         IBOUND = gr.const(1) # no fixed heads

         Out = fdm_t.fdm3t(gr, t, K, K, K, Ss, FQ, FH, IBOUND, epsilon=1.0)

         # analytical stuff
         u = ((np.abs(gr.xm)**2) * S).reshape((1,gr.Nx)) / (4 * kD *t.reshape((len(t),1)))
         fi =  Q/(4 * np.pi * kD) * W(u)


In [12]: # Extract the heads
         plt.figure()
         plt.setp(plt.gca(), 'xlabel','x [m]', 'ylabel', 'head [m]' , 'xscale', 'log')
         for it in range(len(t)):
             plt.plot(gr.xm[gr.xm>0], Out.Phi[it][-1, iy(0), gr.xm>0], 'b.')
             plt.plot(gr.xm[gr.xm>0], fi[it][gr.xm>0], 'r')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

As can be seen, the numerical and analyticl solutions agree. By changing the variable axial to False or True one can choose between a flat cross sectional model and an axisymmetrical model. The correct axis and analytical solutions will then be shown in the figure. The burden of changing variable for both cases is completely carried by the grid object and the way in which the conductances are computed in fdm3.

To show that the water balance matches we compute below the total flow into the model.

```
In [13]: # plotting head versus time for a set of distances

         plt.figure()
         plt.setp(plt.gca(), 'xlabel', 't [d]', 'ylabel', 'head [m]', 'xscale','log')
         plt.title('head in the cell with the well')

         for iix in range(ix(0.0),ix(0.0)+20):
             plt.plot(t, Out.Phi[:, -1, iy(0.0), iix], 'b.')
             plt.plot(t, fi[:, iix], 'r')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Firstly, we observe that our large-scale model agrees well with the axisymmetric analytic solution.

Secondly, we see that it take the model a few steps to reach the analytic solution. We have discussed this in the previous example.

Thridly, it can be seen that the drawdown in the center of the model, that is the cell [50, 50] is too big relative to that of the analytial well. This is the case from the beginning of the simulation. It, therefore, has no relation with the size of the model. This extra drawdown is,obviously due to the the fact that we have specified the head in the center of the center cell so that the water first has to flow against the resistance within that cell, i.e. within the well, before it enters the aquifer. This, of course, is nog what we mean by a well. To get the head at the boundary of the well, in this case at the boundary of the center cell, we should give that cell a high conductivity.

We'll leave this as an exercise for the user.

Even after we have turned the center cell into a "real" well, there remains a minor difference between the analytical solution and the model. A small distance near the well is natural because the cell that assumes the task of the well is square while the analytical solution assumes a circular well. This shape issue is very local and should not influence deviations further from the well, because the extraction is the same in both the numerical and analytical solutions. Remember there are no fixed heads, only initial heads. We simply have to attribute these remaining minor differences to the discritization. Indeed if we refine our model by increasing Npoints above the difference between the model and the analytical sollution gets even smaller. So, indeed, these last differences are due to the chosen discritization.

One may also experiment with the parmeter `\epsilon`. one will see that the value 1.0 gives the most stable results, initially, with no differece after a few ininitial time steps.

Water balance

The water balance should match at all times. There are two ways to check

The net inflow of the central cell should equal the extraction at all time steps. The total flow relased form storage must also match the extraction at all time steps.

```
In [14]: digits = 1

         # average inflow well cell during each time step:
         repr( [ round( Out.Q[idt, -1, iy(0.0), ix(0.0)] , digits) for idt in range(len(np.diff(t
```

```
Out[14]: '[-1199.8, -1200.0, -1200.0, -1200.0, -1200.0, -1200.0, -1200.0, -1200.0, -1200.0, -1200
```

```
In [15]: # Show the flow from storage during each time step:
         repr( [round( np.sum(Out.Qs[idt]), digits ) for idt in range(len(np.diff(t)))] )
```

```
Out[15]: '[1200.0, 1200.0, 1200.0, 1200.0, 1200.0, 1200.0, 1200.0, 1200.0, 1200.0, 1200.0, 1200.0,
```

Hence the injection in the well-cell is indeed equal to $ Q=-1200 $ m3/d, and the same is true for the relase from storage, which adds 1200 m3/d to the model during every time step. The difference in the first time step of the iflow is due to the storage in the cell itself, which can be set to zero by setting Ss for that cell to zero, if desired.

A well in a semi-confined aquifer (Hantush)

A second famous example for transient flow is that to a well in a semi-confined aquifer according to Hantush. The setup is similar to that of Theis with the only difference that water is also enetering the aquifer from an overlying confining bed. This leakage is proportional to the difference of the head maintained above the semi-confining unit and that in de pumped aquifer.

The sollution according to Hantush can be mathematically written as

$$s = \frac{Q}{4\pi kD} W_h(u, \frac{r}{\lambda}), \quad u = \frac{r^2 S}{4kDt}, \quad \lambda = \sqrt{kDc}$$

Where $W_u(-,-)$ is Hantush's well function, kD [L2/T]transmissivity of the homogeneous aquifer and c [T] the vertical hydraulic resistance of the homogeneous confining unit, S the storage coefficient of the aquifer, t [T] time, r [m] distance to the center of the well, and Q [L3/T] the constant extraction that starts at $t = 0$.

The well fucntion can mathematically be written as

$$W_h(u, \frac{r}{\lambda}) = \int_u^\infty \frac{e^{-y-\frac{1}{y}\left(\frac{r}{2\lambda}\right)^2}}{y} dt$$

For $ t :raw-latex:`\rightarrow `:raw-latex:`infty $, we have $ u :raw-latex:`rightarrow `0$, the steady state solution, to that because

$$s(r, \infty) = \frac{Q}{2\pi kD} K_0\left(\frac{r}{\lambda}\right) = \frac{Q}{4\pi kD} W_h\left(0, \frac{r}{\lambda}\right) = \frac{Q}{4\pi kD} \int_0^\infty \frac{e^{-y-\frac{1}{y}\left(\frac{r}{2\lambda}\right)^2}}{y} dt$$

So that

$$\int_0^\infty \frac{e^{-y-\frac{1}{y}\left(\frac{r}{2\lambda}\right)^2}}{y} dt = 2K_0\left(\frac{r}{\lambda}\right)$$

Bruggeman (1999, p877) provides a somewhat different parameterization of the function, one that completely separates time from space, which are mixed in the Hantush form, because both u and r/λ contain the distance r to the well. The form given by Bruggeman uses $:raw-latex:`\tau `= t/(cS)$ and $:raw-latex:`rho = r/:raw-latex:`lambda $ instead. Both parameters are dimensionless. :raw-latex:`tau is time relative to :math:`cS$, a characteristic time for the aquifer system and ρ is a distance relative to a characteristic distance λ of the aquifer system.

$$Wh(u, \rho) = Wb(\tau, \rho) = \int_0^\tau \frac{e^{-y-\frac{\rho^2}{4y}}}{y} dy$$

We may now numerically integrate to zero and add half the bessel function.

```
In [16]: def Wh(U, rho, npl=20, inf=1e2):
             """returns Hantush's well function

             The computation is done by integration from u to exp(umax)

             Parameters:
             -----------
             U : ndarray
                 (r^2 S)/(4 kDt)
             rho: float
                 r / lambda = r / sqrt(kD c)
             npl : int
                 number of integration poins per ln cycle
             inf : float
                 practical value for infinity as upper limit for integration

             Returns:
             --------
             Hantush's well function value

             TO 161024
             """
             import numpy as np

             linf = np.log10(inf)
             if isinstance(U, float):
                 U = np.array([U])
             else:
                 U = U.ravel()
             wh = np.zeros((len(U),1))
             for it,u in enumerate(U):
                 y = np.logspace(np.log10(u), linf,
                                 int( (linf - np.log10(u))*npl+1) )
```

```
            arg = np.exp(-y - (0.25 * rho**2) / y) / y
            wh[it] = np.sum(0.5 * (arg[:-1] + arg[1:]) * np.diff(y))
        return wh
```

In [17]:
```
# Compute and show the Hantush type curves together with the Theis type curve
plt.figure()
plt.setp(plt.gca(), 'xscale','log', 'yscale','log',
         'xlabel', '1/u','ylabel','Wh(u)',
         'title', 'Hantush type curves',
         'ylim',(1e-5, 1e2), 'xlim', (0.1, 1e6))

# Values of r/L
Rho = [0.01, 0.05, 0.1, 0.5, 1., 2., 3.]

# values for U
U = 1/np.logspace(-1, 6, 71)

# Theis
plt.plot(1/U, -scipy.special.expi(-U), label='Theis')
# Hantush for each value of r/L
for ir, rho in enumerate(Rho):
    plt.plot(1/U, Wh(U,rho), label='r/L={0}'.format(rho))
plt.legend(fontsize='small')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Out[17]:
```
<matplotlib.legend.Legend at 0x111620908>
```

We will now try to reproduce these Hantush type curves with our 3D transient finite difference model and compare them with the analytical solution.

To produce the curves, we make sure that $\frac{Q}{4\pi kD} = 1$. We also make sure that $1/u$ will vary from 0.1 to 10^6. And we make sure that we produce curves for the same set of values for r/λ.

In [18]:
```
#aquifer
d =  10. # m, thickness of the confining unit
D =  10. # m, thickness of teh aquifer

c  = 250. # d, vertical hydraulic resistance of the confining unit
k0 = (0.5 * d)/c # m/d, water flows only through 05.*d of confining unit
k1 =  100. # m/d, horizontal hydraulic conductivity of the aquifer
kD = k1 * D # m2/d, transmissivity of aquifer
L  = np.sqrt(kD * c) # characteristic or spreading length of the semi-confind aquifer sy

S = 0.001 # [-], storage coefficient of aquifer

Q = 4 * np.pi * kD # m3/d, extraction by well making sure Q/(4 pi kD) = 1

t = np.logspace(-12, 12, 241)  # 7 log cycles, should be enough

# coordinates
Rho = np.array([0.01, 0.05, 0.1, 0.5, 1., 2., 3.]) # desired r/L curves
rm  = Rho * L  # distances to extract drawdowns from to get dedired r/L curves

r0 = 0.2 # m, radius of the well
z0 = 0 # m, ground elevation, top of confining unit
z = z0 - np.array([0, d, d+D])
y = np.array([-0.5, 0.5])  # m, a one m thick model (ignored when axially symmetric)
x = np.hstack((rm-0.1, rm+0.1 ,1.01*r0, np.logspace(np.log10(r0), np.log10(5 * L), 151))
# added rm-0.1 and rm+0.1 to have gr.xm points exactly on rm for extraction of heads

gr = mfgrid.Grid(x, y, z, axial=True)
```

```
        # System arrays
        K  = gr.const(np.array([k0, k1]))
        Kz = gr.const(np.array([k0, 1e3 * k1])) # no vertical resistance in the aquifer
        Ss = gr.const(S/D)
        FQ = gr.const(0.);  FQ[-1, 0, 0] = Q
        FH = gr.const(0.)
        IBOUND = gr.const(1)
        IBOUND[0, :, :] = -1  # heads in confining unit are fixed for Hantush

        # run model
        Out = fdm_t.fdm3t(gr, t, K, K, Kz, Ss, FQ, FH, IBOUND)

        # indices for gr.xm == rm
        Ix = np.array( np.interp(rm,gr.x,np.arange(gr.Nx + 1)), dtype=int)

        # visualize
        plt.figure()
        plt.setp(plt.gca(),'xscale','log','yscale','log',
                 'xlabel','1/u','ylabel','Wh(u)',
                 'ylim',(1e-5, 1e2),'xlim', (1e-1, 1e6),
                 'title', 'Hantush type curves numerically and analytically')

        # Theis
        plt.plot(1/U, W(U), label='Theis')

        # Hantush
        for ir, rho in enumerate(Rho):
            # Hantush analytic
            plt.plot(1/U, Wh(U,rho), label='r/L={0}'.format(rho))

        for ir, rho in enumerate(Rho):
            # Hantush numeric
            u = (rm[ir]**2 * S) / (4 * kD * t)
            plt.plot(1/u, Out.Phi[:, -1, 0, Ix[ir]], '+') #, label='r/L={0}'.format(rho))

        plt.legend(fontsize='small')
```

```
Running in axial mode, y-values are ignored.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Out[18]: <matplotlib.legend.Legend at 0x1114b8ba8>
```

The numerical and analytical solutions agree except for the last two lines, the ones for $ r:raw-latex:‘lambda ‘$ is 5 and 6. There is no obvious reason for that. It needs some detailed study to find out. The analytically computed type curves are correct as can be verified on the graph of these type curves in Kruseman and De Ridder.

Excercises

Compute / show delayed yield

Delayed yield may result from the drawdown occuring above the aquitard, caused by downward leakage through the aquitard as a consequence of pumping in the underlying aquifer. It also results from the combination of elastic storage and water-table storage in the same unconfined aquifer. Initially the drawdown is due to elastic storage, which spreads fast. Slowly then, the water table dradown will take over and finally determine the drawdown. As a consequence the effect of this water-table drawdown becomes visible only after that of elastic storage.

The combined drawdown curve shows two theis-curves in series, the first one is determined by the elastic storage, the second one by the water table storage. In Python this can readily be modeled by giving all cells a small elastic

specific storage as was done in the previous examples, and only give the top layer cells a larger one that matches the specific yield. Then compute the time-drawdown curve and compare it with the two Theis curves.

Compute well-bore storage (Boulton)

The storage inside the well reduces the drawdown shortly after the start of the pump. This effect was analytically analyzed by Boulton (1963) and later by Neuman (1971). It may be implemented numerically in axially symmetric mode by modeling the water inside the well casing explicitly. To to this, a thin column may be given a zero horizontal conductivity to represent the impervious well casing. Then the top cell inside the casing is given a storage coefficient equal to 1. To represent the free water level inside the screen and the casing, use a large vertical conductivity. The extraction may then be from any of the cells inside the screen or the casing. The large vertical conductivity inside the well makes sure the head is the same throughout the well screen and casing. The result should be compared with the analytical solution given by Boulton. A practical manner is comparing it with curves for Boulton in Pumping Test Books (e.g. Kruzeman & De Ridder, 1970, 1995)

The figure gives an example of a large open well in India.

Fig. 9.1: Large open well

Conclusion

We now have a flexible 3D transient finite difference model which can be used to solve fully 3D transient problems, whereby one can easily switch between regular mode and axially symmetric model.

The model was verified using the analytical solutions of Theis and Hantush. The Theis solution was verified both by an axially symmetric model as by a large flat model.

Last part to consider is particle tracking.

CHAPTER 10

Particle tracking (under construction)

Prof. dr.ir. T.N.Olsthoorn

*Heemstede, 26 Oct 2016, 24 May 2017

Flow lines as opposed to stream lines

Particle tracking is one of the functions most used in a groundwater model. Contrary to stream lines that require steady-state 2D flow without sources and sinks, particles may always be tracked to create flow lines. Clearly, particles starting at the same location may not follow the same path if released at different times in a transient model. In the random walk technique particles are even given a random displacement at each time step to simulate dispersion, which alters the path of individual particles in an unforeseen manner, thus simulating dispersion.

Particle tracking in finite difference models is quite straightforward. The flows perpendicular to the cell faces are known and, therefore, the specific discharge at theses faces may be approximated by dividing by their surface area. Average. As the porosity in the cells at either side of a cell face may differ, so may the groundwater velocity perpendicular to the cell face, even though the specific discharge does not.

In finite difference modeling, the flow in `x`, `y` and `z`- direction, which is parallel to the axes of the model, is linearly interpolated between that at opposite cell faces. This implies that the flow in `x`-direction (and velocity for that matter) is only a function of `x`, the velocity in `y`-direction only a function of `y` and the one in `z`-direction only depends on `z`. This is consistent with the model assumptions and largely simplifies the analysis. However, for large cells it may not be accurate. So it may be necessary to use smaller cells where large variations in velocity occur in value and direction. On the other hand the elegance of this approach is that the divergence remains zero in a cell. This means no water is lost, so that the flow paths by themselves are consistent.

Theory

It's easier to write equations in Lyx than it is to write them in markdown or pure LaTeX.

The idea is to use relative coordinates.

Because the elevation of cells in a 3D finite difference grid may vary from cell to cell in the same layer, it may jump when particles cross vertical cell faces. Exactly at such cell faces, the elevation is undetermined in such a grid. This makes it much more convenient to use relative coordinates, in which each cell of the grid is a cube of sides of length 1. To keep the travel time within each cell in each direction the same we divide the velocity in direction x by Δx and likewise in the other directions.

Because we only have the flows at the cell faces, we have to assume that the velocity varies linearly between two opposite cell faces. For the x-direction we thus obtain for an arbitrary cell $v_x = \frac{Q_{xL}}{\epsilon R \Delta y \Delta z} + \left(\frac{Q_{xr} - Q_{xl}}{\epsilon R \Delta y \Delta z}\right)\left(\frac{x - x_L}{\Delta x}\right)$

where ϵ is the effective porosity of the considered cell and R is the retardation due to sorption ($R = 1$ in the absence of sorption).

To change to the velocity in the grid that consists of unit cubes, we have

$$v_u = \frac{v_x}{\Delta x} = \frac{Q_{x0}}{\epsilon R V} + \frac{Q_{x0} - Q_{x1}}{\epsilon R V}\left(\frac{x - x_0}{\Delta x}\right) = v_{u0} + (v_{u1} - v_{u0})(u - u_0) = v_{u0} + a_u (u - u_0)$$

with

$$a_u = v_{u1} - v_{u0}$$

The index 0 denotes the left side of the cell and the index 1 denotes the right side. This corresponds with the relative local coordinates in the unit cube cell.

All values v_{u0}, v_{u1} and a_u can be computed a priori for all cells in the relative grid. The same is true for the other two axes, y, and z, that become directions v and w in the relative grid.

As long as we are within a single cell we can set

$$U = u - u_0$$

in which $0 \leq U \leq 1$ and U is the local coordinate.

The travel time in this cell from intial position U_s to an arbitrary position U then follows from

$$\frac{dU}{dt} = v_{u0} + a_u U$$

$$dt = \frac{dU}{v_{u0} + a_u U}$$

$$dt = \frac{1}{a_u}\frac{d(v_{u0} + a_u U)}{v_{u0} + a_u U}$$

$$a_u (t - t_s) = \ln\left(\frac{v_{u0} + a_u U}{v_{u0} + a_u U_s}\right)$$

in which the index s denotes the position at the start, $t = t_s$, $U = U_s$.

If $v_{u_0} > 0$, then we find the time that the particle hits the cell face by setting $U = 1$. If $v_{u_0} < 0$ we find it by setting $U = 0$.

Of course, this only makes sense if the argument of the $\ln(-)$ is greater than 0. That is, if the velocities at the two opposite cell faces have the same sign. If not, the velocity is zero somewhere within the cell and the particle can never reach the opposite face. The arrival time may then immediately be set equal to ∞.

Another problem occurs when $a_u = 0$, i.e. when the velocities at opposite cell faces are the same. In that case the velocity is constant so that the arrival time can be obtained from

$$t - t_s = \frac{U - U_s}{v_{u0}}$$

Therefore we have to select the linear or the logarithmic equation to compute the time that the cell-face is it of set it to ∞ when it will ever be hit, which is also true of both v_{u0} and a_u are zero.

We do the same for the v and the w directions that correspond to y and z in the original grid.

The result is three hitting times, of which the smallest one determines when the particle firs hits one of the 6 cell faces of the cube. If this time is smaller than our end-time, we move the particle to that face and subtract the time from the original time, to get the time that it still has to travel. If the time is larger than the end-time, we set the time to the end-time and move the particle accordingly, after which the simulation for this particle has finished.

$$a_u U = (v_{u0} + a_u U_s)\, e^{a_u (t - t_s)} - v_{u0}$$

$$U = U_s e^{a_u (t - t_s)} + \frac{v_{u0}}{a_u} \left(e^{a_u (t - t_s)} - 1 \right)$$

and for the linear case

$$U = (t - t_s)\, v_{u0} + U_s$$

For these formulas to work, it is necessary that the the sign of the velocities is in line with the direction of the axis. In the relative grid we let u, v and w run in the direction of increasing cell indices. This is the case for the x direction where increasing x values coincide with increasing column indices. This is not true for the y and the z directions, where the coordinates run opposite to the cell indices in those directions. Therefore, we have to invert the sing of the Q_y and Q_z arrays front up.

When the end-time has not been reached, the particle crosses over to the next cell. We update its indices to that of the next cell and also update its relative coordinates U, V and W to reflect the starting position of the particle within the new cell.

Particles may end-up in sinks, i.e. cell from which water leaves the model. We will assume that particles have left the model when the enter a cell that is a large-enough sink, that is, a cell for which the total extraction is larger than $sinkFrac \times Q_{intot}$, where is chosen by the user as $0 \leq sinkFrac \leq 1$, usually 0.25 and Q_{intot} is the total inflow of the cell through its cell faces.

Relative coordinates can be readily computed by interpolation using cell grid indices as known values like so:

$$u = \text{interp}\left(xp,\, xGr,\, \text{arange}\left(\text{len}\left(xGr\right)\right)\right)$$

$$U = u - \text{floor}\left(u\right)$$

where xp is a grid coordinate, xGr are the grid coordinates of the grid lines between the columns and $\text{arange}()$ is the Python function that generates numbers between 0 and the specified number ($\text{len}(xGr)$).

Computing grid coordinates from relative coordinates works the other way around

$$x = \text{interp}\left(U + i_u,\, \text{arange}\left(\text{len}\left(xGr\right)\right),\, xGr\right)$$

in which i_u is the cell index of the particle along the x direction, $u = U + i_u$, and $i_u = \text{floor}\left(u\right)$.

In general, we should not have to worry about particles leaving the model, because the velocities perpendicular to all outer faces of the model are zero in the finite difference concept.

Implementation

The implementation can be found in the module `./modules/mfpath.py`. It is about 900 lines, too long to include it in this notebook.

The logic of the particle tracking model is as follows:

- Switch to normalized coordinates u, v, and w
- Compute the velocities at all cell faces in all directions (6 values per cell)
- To track particles compute in local coordinates when they hit the walls of their local cell, which is a uit cube in th normalize grid. The local coordinats are all between 0 and 1 in each cell. To get the normalized coordinates, add the indices of the cell, $u = iu + U$ where iu is the cell index along the u axis and U the local coordinate.
- Doing this and knowning by the sight of the velocity vu wich cell face will be hit, we obtain three hitting times per particle. The smallest time is chosen, together with the corresponding cell face.
- The particle is moved over this smallest time interval, updating its three ocal coordinates.
- The index of the cell is updated for the direction into which the particle hits the cell facce.
- The local coordinate of the particle in this direction is reset: if the particle left into the direction of inceasing grid index, it is reset to zero and 1 is added to its cell index is, if it left in opposite direction, it is set to 1.0 and its cell index is reduced by one.

- The coordinates in the other two directions where the particle did not hit the cell face, are left unchanged.

The time in the previous cell is subtracted from the remaining time, after which the procedure is repreated, as long as the remaining time is still larger than zero. After the remaining time has been used up by all particles that are still moving, the next time in the series for which we want particle coordinates has been reached. The particle coordinates are then saved after having them back-transformed to those of the original grid. This play is repeated until all particles have reached the final simulation time, or until no more flowing particles are presented in the model, because the last ones have been swallowed by sinks or stagnated near water divides.

Afterwards, the trackes can be displayed simultaneously or be simulated or animated. The stored partciles correspond to the times that were given to the model. The detail is therefore completely defined by the user.

Verification

To check the particle tracking use some convenient analytical solutions

A cross section, thickness `H`, porosity and recharge `n`, with a water divide at `x=0` center obeys the following relations

$$v_x = \frac{dx}{dt} = \frac{nx}{\epsilon H} \rightarrow \frac{dx}{x} = \frac{n}{\epsilon H} dt \rightarrow \ln(x) = \frac{n}{\epsilon H} t + C \quad (\text{with } t = t_0,\ x = x_0)$$

$$\ln(x_0) = \frac{n}{\epsilon H} t_0 + C \rightarrow C = \ln(x_0) - \frac{n}{\epsilon H} t_0$$

$$\ln\left(\frac{x}{x_0}\right) = \frac{n}{\epsilon H}(t - t_0) \rightarrow x = x_0 \exp\left(\frac{n}{\epsilon H}(t - t_0)\right)$$

This can be used to check the travel time in the model in two directions.

Another simple check is a well in a confined aquifer. Here we have

$$Qt = \epsilon H \pi R^2 \rightarrow R = \sqrt{\frac{Qt}{\pi \epsilon H}}$$

So set up a model, run it, contour the results, run fdm2path, and check its results by clicking a point near the well

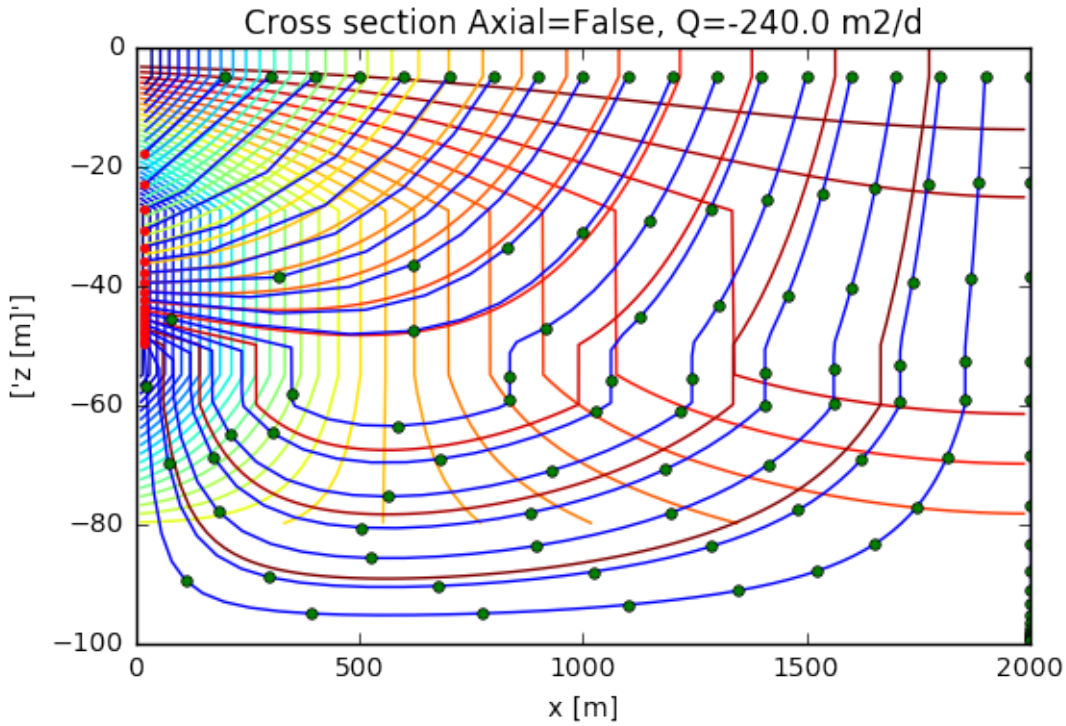
Example

The example used when we developed the stream lines is reused here to show both the stream lines and the tracked particles. They should match in the steady-state situaion, when particles are released on stream lines.

Cross section (flat) with heads, streamlines and some particle tracks, obtained by clicking on the figure when fdm2path is running (backward traces as times were negative, see input above). There is great detail near the sheet piling where all the streamlines converge, which can only be seen by zooming in.

```
In [3]: myModules = './modules/'
        import numpy as np
        import matplotlib.pyplot as plt
        import sys

        if myModules not in sys.path:
            sys.path.insert(1, myModules)

        from importlib import reload

In [4]: import mfgrid
        import fdm
        import mfpath
        import pdb

        reload(fdm)
```

```
        axial = False
        Q    = -2400 if axial else -240. # m3/d if axial else m2/
        por  = 0.35 # [-], effective porosity

        xGr = np.logspace(-1, 4, 51)
        xGr = np.linspace(0, 2000, 101)
        yGr = np.array([-0.5, 0.5])
        zGr = np.array([0., -5, -50, -60, -100])
        gr   = mfgrid.Grid(xGr, yGr, zGr, axial)

        IBOUND = gr.const(1); IBOUND[0, :, :] = -1 # head in top confining unit fixed
        k      = gr.const(np.array([0.01, 10., 0.01, 20.]))
        FH     = gr.const( 0.);
        FQ     = gr.const( 0.)
        FQ[1, 0, 0] =  Q # insecond layer

        # run flow model
        Out = fdm.fdm3(gr, (k, k, k), FQ, FH, IBOUND)

        Psi = fdm.psi(Out.Qx)

        #pdb.set_trace()

        # visualize
        title = 'Cross section Axial={0}, Q={1} {2}'.format(axial, Q, 'm3/d' if axial else 'm2/d')
        ax = plt.figure().add_subplot(111)
        xlim = gr.x[[0, -1]]
        ax.set(xlabel='x [m]', ylabel=['z [m]'], title=title, xlim=xlim)

        ax.contour(gr.xm, gr.zm, Out.Phi[:, 0, :], 30)
        ax.contour(gr.xp, gr.zp, Psi, 30)
        #plt.show()

        # path lines
        T=np.linspace(0, 3650, 100) #time series
        if True:
            Xp = np.linspace(200, 2000., 19)
            Yp = np.zeros(Xp.shape)
            Zp = np.ones(Xp.shape) * -5.
        else:
            Zp = np.linspace(-5., -95., 19)
            Yp = np.zeros(Zp.shape)
            Xp = np.ones(Zp.shape) * 1000.

        #Pcl = mfpath.particle_tracker(gr, Out, por, T, Xp, Yp, Zp)
        Pcl = mfpath.particle_tracker(gr, Out, gr.const(por), T=T, particles=(Xp, Yp, Zp), sinkfra

        mfpath.plot_particles(Pcl, axes=ax, first_axis='y', ls='none',
                              markers='o    ', mfc='green', markersize=4)

        plt.show()
        #R = np.sqrt(Q * T[-1] / (np.pi * por * np.sum(gr.dy)))
Forward tracking, because T is ascending
Job done, 19 particles tracked for time  from t=0.0 to t=3650.0 in 99time steps.
    The results are in variable Pcl (a 'named_tuple 'Pcl'.
    whose importnat fields are Status, X, Y, Z, T, up, vp, wp.
    At the and there were:
    1 particles still active
    18 particles captured by sinks
    0 particles stagnant
    The average arrival time of the captured particles is 0.05263157894736842
```



TODO: More examples will follow. A number of them in 3D are already in the testsuite

Conclusion

We have implemented the particle tracking and used it both interactively as in batch mode. The method was verified by comparison with the stream lines drived from the stream function. Tracking particles allows computing travel times, both forward and backward tin time.

Flow lines can be used where stream lines cannot, as flow lines don't require 2D divergence-free flow. Flow lines can also be used in transient situations. Flow lines with travel times are amongst the most used results of groundwater modeling, wherever pollution is involved.

To even better track particles while including dispersion and diffusion, we could track massive amounts of particles. This also allows computation of arrival concentrations. This is the subject of the last chapter.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search