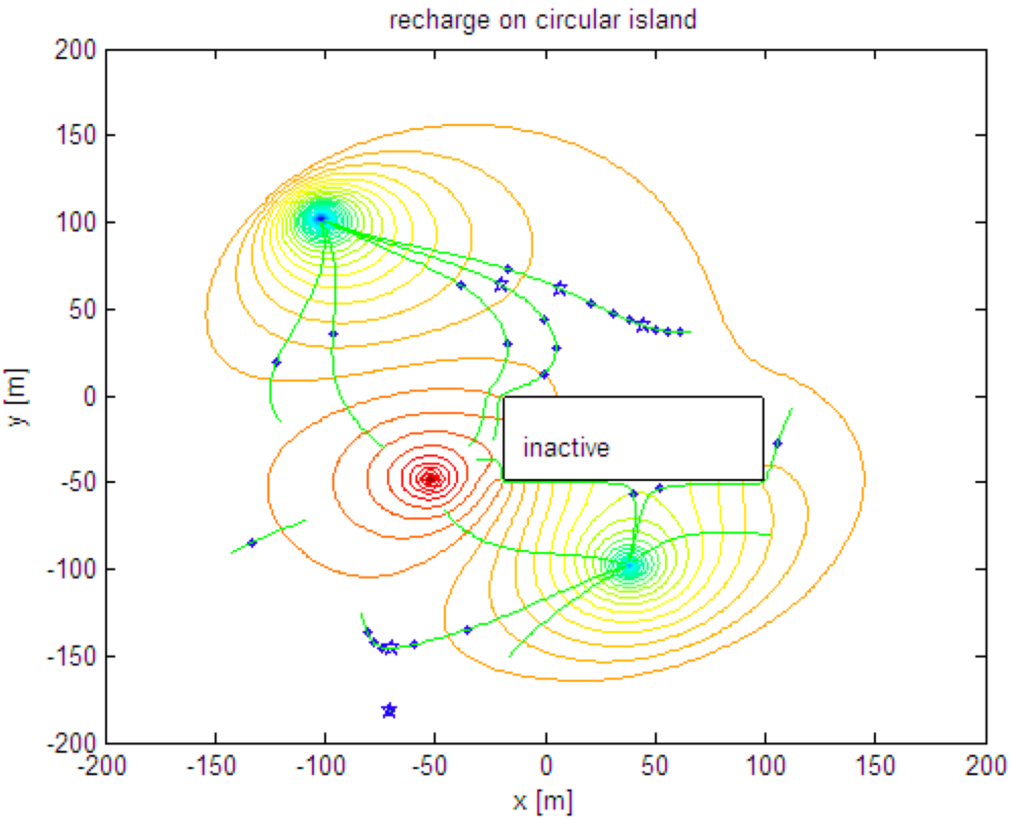
Syllabus CT 5440 (Geohydrology 2) Finite Difference Groundwater Modeling in Matlab

Prof. dr.ir. T.N.Olsthoorn

June 30, 2014



ABSTRACT

This syllabus explains the theory behind numerical groundwater modeling and how to make your own finite difference groundwater models in Matlab (The Mathworks ®). 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 Matlab. 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 Matlab 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.

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 Matlab 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 Matlab 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. Matlab 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, Matlab is also a powerful environment to visualize modeling results. Therefore, outside Matlab no additional packages are required. Some Matlab knowledge has, of course, to be acquired during the course. There exist very good Matlab manuals on the web; for instance Google for "learning and Matlab" to find such documents and keep one at hand during the course.

Contents

1 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.

1.1 General overview numerical models

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 1.1. 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 1.1. 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.

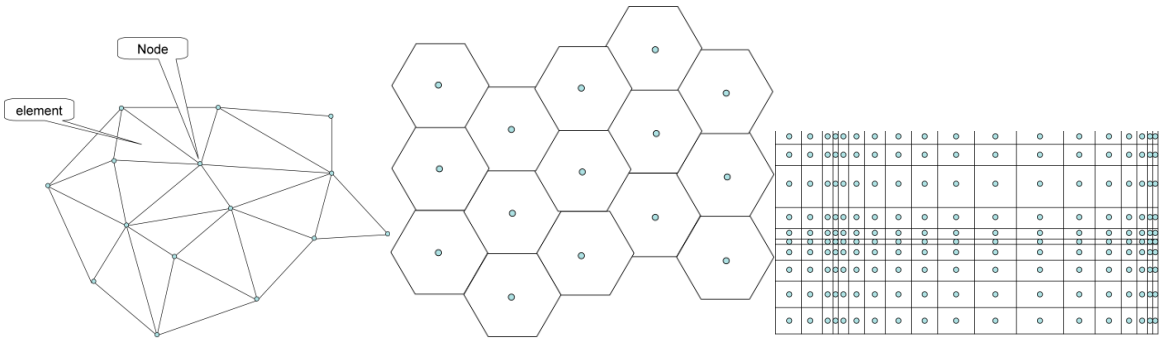


Figure 1.1: Different model meshes (grid). Left: a finite element triangular network with the nodes at the element corners. Middle: a hexagonal finite difference network with nodes in the center of hexagonal cells. Right: a rectangular finite difference network with nodes in the center of the cells. 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.

1.2 Deriving and assembling a numerical model

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 1.2 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.

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

$$Q_{ij} = C_{ij}\,(\phi_i - \phi_j) \tag{1.1}$$

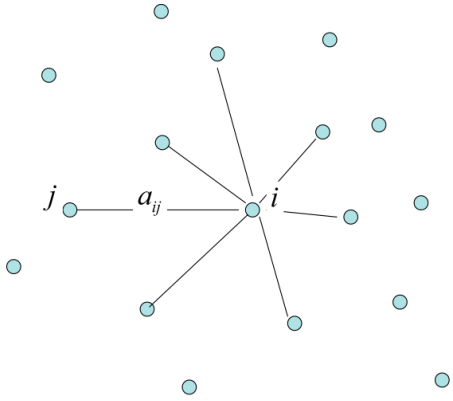


Figure 1.2: A model node with its surrounding connected neighbors

$$= \frac{1}{R_{ij}} (\phi_i - \phi_j) \tag{1.2}$$

C_{ij} $[(\mathrm{L}^3/\mathrm{T})/\mathrm{L}]$ or $[\mathrm{L}^2/\mathrm{T}]$ is called the "conductance" and its reciprocal is the "resistance" $[\mathrm{L}/(L^3/\mathrm{T})]$. 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 $[\mathrm{L}^3/\mathrm{T}]$ 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

$$\sum_{j=i,\, j\neq i}^{j=N} Q_{ij} = Q_i \tag{1.3}$$

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:

$$\sum_{j=1,\, j\neq i}^{N} C_{ij} (\phi_i - \phi_j) = Q_i \tag{1.4}$$

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. $7 \times 4 \times 0.81 \times 10^6 \approx 30\,\mathrm{Mb}$ 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

$$-C_{i1}\phi_1 - C_{i2}\phi_2 - \ldots + \left(\sum_{j=i,\, j\neq i}^{j=N} C_{ij} \right) \phi_{ii} \ldots - C_{i,\,N-1}\phi_{N-1} - C_{i,\,N}\phi_N = Q_i \tag{1.5}$$

or

$$-C_{i1}\phi_1 - C_{i2}\phi_2 - \ldots + C_{ii}\phi_{ii} \ldots - C_{i,\,N-1}\phi_{N-1} - C_{i,\,N}\phi_N = Q_i \tag{1.6}$$

where

$$C_{ii} \;=\; -\sum_{j=1, j\neq i}^{N} C_{ij} \tag{1.7}$$

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 1.6 can be written compactly as follows:

$$\sum_{j=1}^{N} C_{ij}\phi_j = Q_i \tag{1.8}$$

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 1.8 is equivalent to the matrix equation

$$\mathbf{C\Phi = Q} \tag{1.9}$$

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 1.7. 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, Matlab offers sparse matrices and sparse matrix functions. These sparse matrices work exactly like ordinary matrices but they store only the non-zero elements. Matlab also offers sparse matrix functions, which 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.

1.3 Boundary conditions

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 1.9; 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.

1.3.1 General head boundaries

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

$$Q_{ex,i} = \hat{C}_i \left(h_i - \phi_i\right) \tag{1.10}$$

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

$$\sum_{j=i,\, j\neq i}^{N} -C_{ij}\phi_j + C_{ii}\phi_i = Q_i + \hat{C}_i \left(h_i - \phi_i\right) \tag{1.11}$$

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

Equation 1.10 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

$$\sum_{j=i,\, j\neq i}^{N} -C_{ij}\phi_j + \left(C_{ii} + \hat{C}_i\right)\phi_i = Q_i + C_i h_i \tag{1.12}$$

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 Matlab, using the subscript **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 Matlab even when the model is non-linear and use Matlab's standard matrix solver (its famous backslash operation) repeatedly when needed, until convergence is achieved. This mostly works faster.

1.3.2 Drain boundaries (DRN)

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:

$$\mathbf{b_{drn}} = (\Phi > \mathbf{h_{drn}}) \tag{1.13}$$

Hence, the drains are implemented as follows:

$$\left(\mathbf{C} + diag\left(\hat{\mathbf{C}}_{\mathbf{drn}} \cdot \mathbf{b_{drn}}\right)\right) \Phi = \mathbf{Q} + \hat{\mathbf{C}}_{\mathbf{drn}} \cdot \mathbf{b_{drn}} \cdot \mathbf{h_{drn}} \tag{1.14}$$

Notice that in MATLAB 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.

1.3.3 River boundaries (RIV)

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:

$$\begin{aligned} Q_{riv} &= \hat{C}_R \left(h_{riv} - \phi\right), \quad \phi > h_{bot} \\ Q_{riv} &= \hat{C}_{riv} \left(h_{riv} - h_B\right), \quad \phi \leq h_{bot} \end{aligned}$$

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

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

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

$$\begin{aligned} Q_{riv} &= \hat{C}_{riv} \left(h_{riv} - \phi\right) b_{riv} + \hat{C}_{riv} \left(h_{riv} - h_{riv}\right) - \hat{C}_{riv} \left(h_{riv} - h_{bot}\right) b_{riv} \\ &= \hat{C}_{riv} \left(h_{riv} - h_{bot}\right) + \hat{C}_{riv} \left(h_{bot} - \phi\right) b_{riv} \end{aligned}$$

Therefore MODFLOW-type rivers can be implemented as follows

$$\left(\mathbf{C} + diag\left(\hat{\mathbf{C}}_{\mathbf{riv}} \cdot \mathbf{b_{riv}}\right)\right) \Phi = \mathbf{Q} + \hat{\mathbf{C}}_{\mathbf{riv}} \left(\mathbf{h_{riv}} - \left(1 - \mathbf{b_{riv}}\right) \mathbf{h_{bot}}\right) \tag{1.15}$$

1.4 All head-dependent boundaries, except the directly fixed-head boundaries

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

$$\left(\mathbf{C} + diag\left(\hat{\mathbf{C}}_{\mathbf{ghb}} + \hat{\mathbf{C}}_{\mathbf{drn}} \cdot \mathbf{b_{drn}} + \hat{\mathbf{C}_{\mathbf{riv}}} \cdot \mathbf{b_{riv}}\right)\right) \Phi = RHS \tag{1.16}$$

$$RHS = \mathbf{Q} + \hat{\mathbf{C}}_{\mathbf{ghb}} \cdot \mathbf{h_{ghb}} + \hat{\mathbf{C}}_{\mathbf{drn}} \cdot \mathbf{h_{drn}} \cdot \mathbf{b_{drn}} + \hat{\mathbf{C}}_{\mathbf{riv}} \cdot \left(\mathbf{h_{riv}} - \left(1 - \mathbf{b_{riv}}\right) \mathbf{h}_{bot}\right) \tag{1.17}$$

Equation 1.16 specifies the complete model from which the heads may be solved in Matlab simply by using its backslash operator.

1.5 Solving the model and checking the results

Combining for simplicity the contribution from the different head boundary conditions under $\hat{\mathbf{C}}$ and $\mathbf{h}$, different, the solution of equation 1.16 simplifies to:

$$\Phi = \left(\mathbf{C} + diag\left(\hat{\mathbf{C}}\right)\right) \setminus \left(\mathbf{Q} + \hat{\mathbf{C}} \cdot \mathbf{h}\right) \tag{1.18}$$

where the backslash (Matlab language means) left division or: solve this set of equations for the unknowns at the left.

The dot product of the vectors $\hat{\mathbf{C}}$ and $\mathbf{h}$, that is $\hat{\mathbf{C}} \cdot \mathbf{h}$, translates to $\mathbf{C}. * \mathbf{h}$ in Matlab, which stands for element-by-element multiplication instead of matrix multiplication and is indeed equivalent to the dot product of the two vectors. The column vector $\hat{\mathbf{C}} \cdot \mathbf{h}$ contains therefore the elements $c_i h_i$.

The latter system equation (1.18), which solves for the unknown heads Φ and includes the boundary conditions, represents the complete model .

Once the heads are computed by 1.18, we may calculate the net inflow of all the nodes or nodes by the matrix multiplication 1.9, 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

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

1.6 Dealing with fixed heads (Dirichlet boundaries)

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 Matlab. 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.

1.6.1 Inactive cells, active cells and cells with fixed heads

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 Matlab obtaining a Boolean array of active cells, inactive cells or fixed head cell can be done as follows:

```
Iact  = (IBOUND>0);
Iinact= (IBOUND==0);
Ifh   = (IBOUND<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(:)>0
Iinact = IBOUND(:)==0
Ifh    = IBOUND(:)<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   = find(IBOUND>0)
Iinact= find(IBOUND==0)
Ifh   = find(IBOUND<0)
```

Which shows how flexible Matlab is.

1.6.2 Including directly fixed head boundaries

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:

$$\mathbf{C}\Phi = \mathbf{Q} \tag{1.19}$$

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 $\neg$ to mean "not" we have:

$$\neg\mathbf{I_{inact}} = \mathbf{I_{fh}} \cap \mathbf{I_{act}} \tag{1.20}$$

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

$$\mathbf{C}\left(\neg\mathbf{I_{inact}}, \neg\mathbf{I_{inact}}\right)\Phi\left(\neg\mathbf{I_{inact}}\right) = Q\left(\neg\mathbf{I_{inact}}\right) \tag{1.21}$$

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 $\mathbf{C}(\neg\mathbf{I_{inact}}, \mathbf{I_{fh}})$ 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 $\Phi(\mathbf{I_{fh}})$. Hence, the matrix-vector multiplication $\mathbf{C}(\neg\mathbf{I_{inact}}, \mathbf{I_{fh}})\Phi(\mathbf{I_{fh}})$ 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 $\mathbf{C}(\neg\mathbf{I_{inact}}, \mathbf{I_{act}})$ untouched at the left hand side. The system equation then becomes:

$$\mathbf{C}(\neg\mathbf{I_{inact}}, \mathbf{I_{act}})\Phi(\mathbf{I_{act}}) = \mathbf{Q}(\mathbf{I_{act}}) - \mathbf{C}(\neg\mathbf{I_{inact}}, \mathbf{I_{fh}}) \cdot \Phi(\mathbf{I_{fh}}) \tag{1.22}$$

Because we only have to compute the heads at nodes indexed by $\mathbf{I_{act}}$, 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:

$$\mathbf{C}(\mathbf{I_{act}}, \mathbf{I_{act}})\Phi(\mathbf{I_{act}}) = \mathbf{Q}(\mathbf{I_{act}}) - \mathbf{C}(\mathbf{I_{act}}, \mathbf{I_{fh}}) \cdot \Phi(\mathbf{I_{fh}}) \tag{1.23}$$

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.

Matlab's backslash ($\backslash$) operator can be directly used to solve this set of linear equations:

$$\Phi(\mathbf{I_{act}}) = \mathbf{C}(\mathbf{I_{act}}, \mathbf{I_{act}}) \setminus (\mathbf{Q}(\mathbf{I_{act}}) - \mathbf{C}(\mathbf{I_{act}}, \mathbf{I_{fh}}) \cdot \Phi(\mathbf{I_{fh}})) \tag{1.24}$$

and where

$$\Phi(\mathbf{I_{fh}})$$

are known beforehand.

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

$$\mathbf{Q}(\neg\mathbf{I_{inact}}) = \mathbf{C}(\neg\mathbf{I_{inact}}, \neg\mathbf{I_{inact}})\Phi(\neg\mathbf{I_{inact}}) \tag{1.25}$$

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

In Matlab code this would look like the following snippet

```
Phi       = STRTHD(:);
Q         = zeros(size(IBOUND));
Iact      = IBOUND(:)>0;
Iinact    = IBOUnD==0;
Ifh       = IBOUND<0;
Phi(Iact) = C(Iact,Iact)
Q(Iact)-C(Iact,Ifh)*Phi(Ifh);
Q(~Iinact)= C(~Iinact,~Iinact)*Phi(~Iinact);
```

2 Finite difference modeling

Until now, everything said was true for all numerical models with nothing specific for finite differences. What remains to be done is to derive the conductances C_{ij}, which is specific for each method. We will do so now for the FDM based on a rectangular grid or mesh consisting of of rectangular cells. The mesh (model network or grid) divides the model area or model space in parts, i.e. cells. The model grid determines the number of connections between the individual nodes.

For convenience we will deal with a 2-dimensional grid with rectangular cells in which the heads heads are defined or computed at the cell centers (right-hand picture in figure 1.1). In such a model, the flow from one node, i, to its neighbor, j, first passes half a cell with the conductivity (or transmissivity), k_i, of the first cell and then half a cell with the conductivity (or transmissivity), k_j, of the receiving cell. This represents flow through two media placed in series, for which resistances add up (not conductances). Therefore, we compute the resistanceR_{ij} to the flow between the cell centers i and j and then take its inverse to get the conductance C_{ij}. Note that extension to 3D is straightforward.

Let D_x be a 2D array holding the N_x widths of all cell columns in the grid and let D_y be the 2D array with the $N_y$width of the rows of the grid. All these arrays are of size $N_y N_x$ (Note that in matlab the y-direction (rows) is the first dimension and the x-direction (columns) is the second dimension). Then the resistance between adjacent cells one column apart becomes:

```
Rx = 0.5 * ( Dx(:,1:end-1)./Dy(:,1:end-1)./Kx(:,1:end-1) + ...
             Dx(:,2:end  )./Dy(:,2:end  )./Kx(:,2:end  ) );

Ry = 0.5 * ( Dy(1:end-1,:)./Dx(1:end-1,:)./Ky(1:end-1,:) + ...
             Dy(2:end  ,:)./Dy(2:end  ,:)./Ky(2:end  ,:) );

Cx = 1./Rx;
Cy = 1./Ry;
```

The $\mathbf{R}_x$ and $\mathbf{C}_x$ arrays have one column less than the number of columns in the mesh, say than the array **IBOUND**. Likewise, the arrays $\mathbf{R}_y$ and $\mathbf{C}_y$ have one row less and the IBOUND array.

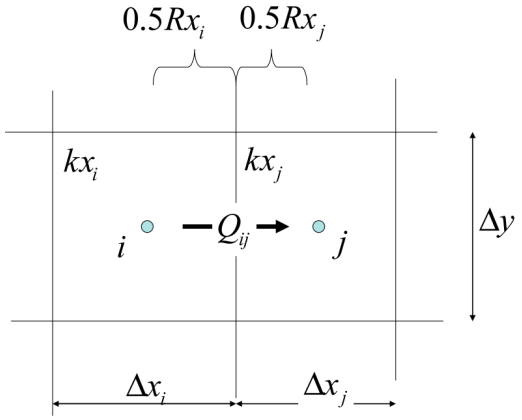


Figure 2.1: Resistance $R_x = 0.5R_{x,i} + 0.5R_{x,j}$ for flow from node i to j. Where $R_{x,i}$ and $R_{x,j}$ are the resistances for flow through an entire cell in x-direction

2.1 Setting up the system matrix

The easiest way to assemble the system matrix is probably by using the indices of adjacent cells like the i and the j in the previous formulas. For this to work we first have to number the cells of the model.

2.1.1 Numbering the cells of a rectangular grid

The easiest way to get a full array of cell number is probably by first assigning the values 1 to $NyNx$ to a vector and then reshape the vector to that of the IBOUND array

```
Nodes = reshape( 1:Ny*Nx, [Ny Nx] );
```

The unique number of cell k, m then is

```
i = Nodes(k,m);
```

and that of its right neighbor is

```
j = Nodes(k,m+1);
```

Notice that this way of numbering follows the natural way that Matlab uses to store elements of an array (y first, x second z third etc). Thus the position of the Node(k,m) in the array, which is i can also be computed as:

```
i = Ny*(m-1)+k
```

2.1.2 Assembling the system matrix

In equations (3) through (13) the indices refer to cells that are connected. Hence C_{ij} is the coefficient between the cells i and j, which, of course, are just adjacent cells.

The system matrix element C_{ij} is the negative value of the conductance between these cells i and j, where the first index refers to the equation number, i.e. the cell for which the water balance is computed, and the second index, j, refers to an adjacent cell, to which it is connected. Generally, $C_{ij} = C_{ji}$, which makes the system matrix C symmetric.

To put C_{ij} at the correct position in the system matrix, we need its value and its indices i and j, i.e. we need the the triple $[i, j, -C_{ij}]$.

This is straightforward with the cell numbering that we have generated. For all adjacent cells in the x-direction we get the cell numbers of the left neighbors **I** and those of the right neighbors **J** and the triple [I, **J**, $-$**Cx**] in Matlab by the flowing lines of code:

```
I = Nodes(:,1:end-1);
J = Nodes(:,2:end  );
[I(:) J(:) -Cx(:)]
```

Note that the colon operator (:) reshapes an array of any shape into a column vector. Therefore, this triple represents an an array consisting of three columns with a length equal to the number of existing neighbors along the x-axis in the whole grid..

If for convenience of remembering we use IW (western neighbor) instead of **I** and **IE** (eastern neighbor) instead of J this becomes:

```
IW = Nodes(:, 1:end-1);
IE = Nodes(:, 2:end  );
[IW(:), IE(:), -Cx(:)]
```

Likewise for adjacent cells along the **y**-axis. Calling them IN (northern neighbor) and IS (southern neighbor), we get

```
IN = Nodes(1:end-1,:);
IS = Nodes(2:end  ,:);
[IN(:), IS(:), -Cy(:)]
```

which is easily extended into the z-direction if required.

These two 3-column arrays can be combined by placing them on top of each other

```
[ [IW(:) IE(:) -Cx(:)] ;
  [IN(:) IS(:) -Cy(:)] ];
```

We can now fill the system matrix with the coefficients $-C_{ij}$. The function *sparse* is called to fill a sparse matrix, which remembers only the non-zero elements. This function requires the elements to be given in the form of three columns, with respectively the i index, the j index and the element value itself. The size of the full matrix also must be supplied (i.e. $NyNx$ by $NyNx$) as well as the maximum number of non-zero elements (i.e. $5NyNx$):

```
Nod = numel(IBOUND);
C=sparse([ IW(:);IN(:) ], [ IE(:);IS(:) ], [-Cx(:);-Cy(:) ] ,Nod,Nod,5*Nod);
```

We have now filled in the C_{ij} but not yet the C_{ji}. We could have done that in the previous code snippet. Because the system matrix is symmetric it can also be done using this symmetry by adding the obtained array with its transpose.

```
C = C + C'
```

The elements C_{ii}, are still missing, which form is the diagonal of the system matrix. It is easily obtained by summation of the elements along the rows of the system matrix:

```
diagC = -sum(C, 2)
```

This summation runs along the second dimension, i.e along the rows of the matrix.

It is possible to put this diagonal into the system matrix using the function *spdiags* as follows

```
C = spdiags(diagC,0,C);
```

Where the 0 means zero-offset from the diagonal of matrix C.

2.1.3 Solving the system and getting results

However, we prefer to keep it separate from the system array to prevent it from being altered when dealing with boundary conditions. We will put it into the system array immediately before we solve the system for its unknown heads in the following way, thus skipping the previous step.

```
Phi = spdiags(diagC,0,C)  Q;
```

Once we have computed the heads Phi, it is easy to compute the nodal inflows in the following way

```
Q = spdiags(diagC,0,C) * Phi;
```

as well as the flows in x-direction across the y-cell faces:

```
Qx = -Cx .* diff(Phi, 1, 2);  % diff once along the second (x or column) dimension
```

Likewise

```
Qy = +Cy .* diff(Phi, 1, 1);  % diff once along the first (y or row) dimension
```

Notice that the sign of the last equation is a + because the rows are numbered opposite to the y-coordinates. I.e. rows are numbered top first as printed on paper or scrolled on screen, while the first row has the highest y-coordinate to be consistent with the familiar Cartesian directions when inspecting a printed array of heads or flows.

It is now straightforward to construct a first finite difference model in Matlab, for the time being with only fixed and active heads. Other boundary conditions can be included layer.

2.2 Grid Object

2.2.1 Setting up the grid object in 2D

When modeling we will use the grid information over and over again. Not only will we use the coordinates of the grid lines, *xGr* and *yGr* but also those of the cell centers *xm* and *ym*, the cell widths *dx* and *dy*, cell volume, cell area etc. We can prevent a large amount of clutter by defining a grid object. This object stores the *xGr* and *yGr* and computes any derived values like the *xm*, *ym*, *dx*, *dy*, Area acts upon request. An object can do this because not only has it fields (like a struct) to store information, it also has methods that can work with this information. A gridObj can for instance plot itself.

Assume we have defined a class of 2D grids called *grid2DObj*. Then we can call it with arguments to obtain a 2D grid object instance, which we may conveniently call *gr*:

```
gr = grid2DObj( xGr, yGr );
```

If we define the grid2DObj class such that it can compute any of the required arrays and values, we may use them for instance as follows:

```
L = sum( gr.dx );      % lenght of entire network along x axis
H = sum( gr.dy );      % length of entire netwerk along y axis
A = sum( gr.Area(:) ); % Total area of network
A = gr.AREA;           % also implemented, it is equivalent to sum(gr.Area)
V = gr.Vol;            % array with volume of every cell
V = gr.VOL;            % also implemented, is equivalent to sum(gr.Vol(:));
gr.plot('c');          % Plot the network grid lines in cyan color
K = gr.const(10);      % Generate a full 2D array K (Ny, by Nx) based on scalar k
B = sum( gr.Area(Phi>0.5) ); % total area of grid cells in which head exceeds 0.5
```

where *gr.dx*, *gr.dy* and *gr.Area* are computed by the gridObj upon request and the call *gr.plot('c')* would plot the grid using cyan as line color and *gr.const(k)* where *k* is a scalar would generate an array *K* that has the size of the full grid. *gr.plot* calls the method *plot* which has been defined for the *grid2DObj* class and *gr.const* calls the method *const* which also has been defined for this *grid2DObj* class. The possibilities are virtually infinite. One of the big advantages of using *grid2DObj* in this way is the error checking it can do when called, and it can make sure that the *xGr* is oriented along the columns with all coordinates sorted increasing and unique, i.e. with double coordinates eliminated. For the *yGr* it can ensure that the *yGr* is oriented along the *y*-axis and with unique coordinates running downward, so that the first value is highest and the last is lowest. This way, the *xGr* and *yGr* coordinates given as input to the constructor *grid2DObj* can be in any order; the *grid2DObj* makes sure that *gr.xGr* and *gr.yGr* are consistent with the model network at all times.

The class of the objects, i.e. *grid2DObj*, has to be coded in Matlab by specifying its properties and method. Properties can be prescribed or dependent, i.e. computed when requested. We will make intense use of dependent properties. The *grid2DObj* is listed below. There is some error checking in the beginning as well as the use of function *getProp* and *getNext* to fetch valiables flexibly from te input line. Some housekeeping is also done. And there are numerous dependent variables defined. These are computed from the pertinent data stored in the object, only when the user requests them. For instance, the variable *dx* is simply computed from *xGr* upon user's request and not stored. Dependent properties are especially important for a large number of variables that generate a full array, i.e. with a value for every cell. Storing them permanently would be prohibative, while it would also be extremely difficult to keep them all up to date each time the some input is changed. All dependent properties must be implemented through so-called *get methods* as is seen in the last part of the *grid2DObj* listing below. This includes the area and volume of the cells. Once defined, the parameter can be indexed as a regular field of the object's instance; they can be accessed and indexed just like the fields in any *struct:*

```
V = gr.Vol(4,5);
A = gr.Area(:,3);
```

Notice that *grid2DObj* is already has features implemented that will be useful later on, such as noticing the object that the model is axially symmetric or not and by implementeing dependent variables like *Nr*, *rGr*, *dr*, *dR* etc, as aliases for *Nx*, *xGr*, *dx*, *dX* etc., to allow these alternative names to be used when the model is axially symmetric. Also, the dependent varialbles *Area* and *Vol* are computed differently if the model or the grid is axially symmetric instead of regular.

The varialbles are generally named such that a small letter indicates a scalar or a vector and its capital a full array with a value for every cell. For instance:

gr.xGr, *gr.dx*, *gr.xm* are vectors, *gr.Xgr*, *gr.dX*, *gr.Xm* are full-size arrays.

Also notice that *gr.xm* refers to the variable *xm* of the instantiation of the *grid2DObj,* which was created using the call *gr = grid2DObj(xGr,yGr)*. However, within the code of the *grid2DObj*, that is, in its definition, its classdef, it is always called just "*o*", hence *"o.xm*", where "*o*" is just the shortest abbreviation of the word Obj. To use this simple short "*o*" for the object inside its defining *classdef* file is just my habit. Users of *Python* generally use *self* for it. It just means "the object that will be generated" when the *grid2DObj* is called..

With this *grid2DObj* class we have an effective tool to deal with the finite difference grid. We will use it wherever we need information about the grid. Whenever desired we can add additonal properties and methods to the class definition.

2.2.2 Listing of the grid2DOBj

```
classdef grid2DObj
    %GRID2DOBj -- 2D gridObj to facilitate handling grids for 2D FDM
    % TO 140410
    properties
        % stored properties
        xGr,yGr,zGr,Z
        AXIAL = false;
    end
    properties (Dependent=true)
        % properties computed when used
                          rGr
        Nx,Ny,Nz,Nod,     Nr
        XGr,YGr,          RGr
        dx,dy,dz,         dr
        dX,dY,dZ,         dR
        xm,ym,zm,         rm
        Xm,Ym,Zm,         Rm
        xp,yp,            rp
        Xp,Yp,            Rp
        Area,AREA
        Vol,VOL
    end
    methods
        function o = grid2DObj(xGr_,yGr_,varargin)
            %GRID2DOBJ -- constructor
            % USAGE: gr = grid2DObj(xGr,yGr[,Z],'AXIAL'[,true|{false}])
            %  xGr = grid line coordinates of columns
            %  yGr = grid line coordinates of rows
            %  Z   = vector or 3D array of tops and bottoms of layer
            %        may be vector, [1 0] is assumed when omitted.
            % 'AXIAL',true|false indicates whether the grid is for an
```

```
    %        axially symmetric model or not, default = false
    % 'AXIAL' without true or false, then true is assumed

    [Z_,     varargin] = getNext(varargin,'double',[1 0]);
    [o.AXIAL,varargin] = getProp(varargin,'AXIAL',false);
    if ~isempty(varargin)
        o.AXIAL = true;
    end
    fprintf('Notice: grid has AXIAL = %d\n',o.AXIAL);

    o.xGr = unique(xGr_(:)');
    o.yGr = unique(yGr_(:) ); o.yGr = o.yGr(end:-1:1);

    if nargin<3 || o.AXIAL
        o.Z   = cat(3,o.const(1),o.const(0));
        o.zGr = mean(mean(o.Z,1),2);
    else
        if isvector(Z_)
            o.Z = YS(unique(Z_(:)));
            o.Z = o.Z(end:-1:1);
        end
        o.zGr = mean(mean(o.Z,1),2);
        if size(o.Z,1)~=o.Ny
            o.Z = bsxfun(@times,ones(o.Ny,1),o.Z);
        end
        if size(o.Z,2)~=o.Nx
            o.Z = bsxfun(@times,ones(1,o.Nx),o.Z);
        end
    end
    % verify
    if o.Ny<1, error('Need at least 2 distinct yGr coordinates.'); end
    if o.Nx<1, error('Need at least 2 distinct xGr coordinates.'); end
    if o.Nz<1, error('Need at least 2 distinct zGr coordinates.'); end
    if o.Nz>1, error('Use at most 2 z-elevations in 2D models.');  end
end
function sz = size(o,dim)
    %SIZE -- size of grid
    % USAGE: sz = gr.size([dim])
    %   dim is optional or 1 or 2
    if nargin<2,  sz = [o.Ny o.Nx]; return; end

    if  dim<2, sz = o.Ny;  else  sz = o.Nx;  end
end
function A = const(o,a)
    %CONST -- generate constant array of size Ny,Nx filled with a
    % USAGE A = gr.const(a)
    % a is a scalar or first entry of array or vector
    if isscalar(a)
        A = a + zeros(o.Ny,o.Nx);
    elseif size(a,2) == 1
        A = a * ones(1,o.Nx);
    elseif size(a,1) == 1
        A = ones(o.Ny,1) * a;
```

```
    else
        error('input must be scalar or a vector');
    end
end
function plot(o,varargin)
    %PLOT -- plots the grid/mesh using linespec
    % USAGE: gr.plot([lineSpec[,property,value[,property,value[,...]]]])
    %   lineSpec is legal Matlab line specification for plot (see
    %   plot function in Matlab, e.g 'r'  'b--'  'ko.-')
    %   property value pairs should be legal propery values pairs
    %   for plot function in Matlab (e.g, 'lineWidth',2, ...)
    set(gca,'nextPlot','add');
    for ix=1:numel(o.xGr)
        plot(o.xGr([ix ix]),o.yGr([1 end]),varargin{:});
    end
    for iy=1:numel(o.yGr)
        plot(o.xGr([1 end]),o.yGr([iy iy]),varargin{:});
    end
end
function [xr,Ix] = xr(o,x)
    %XR -- relative coordinates left oriented
    % USAGE: [xr,Ix] = gr.xr(x);
    Ix = interp1(o.xGr,1:o.Nx+1,x);
    xr = Ix - floor(Ix);
    Ix =      floor(Ix);
    xr(Ix==o.Nx+1) = 1;
    Ix(Ix==o.Nx+1) = o.Nx;
end
function [xrm,Ix] = xrm(o,x)
    %XRM -- relative coordinages, central oriented
    % USAGE: [xrm,ix] = gr.xrm(x);
    [xrm,Ix] = o.xr(x);
    xrm = xrm-0.5;
end
function [yr,Iy] = yr(o,y)
    %YR -- relative y coordinates, top oriented
    % USAGE: [yr,iy] = gr.yr(y);
    Iy = interp1(o.yGr,1:o.Ny+1,y);
    yr = Iy - floor(Iy);
    Iy =      floor(Iy);
    yr(Iy==o.Ny+1) = 1;
    Iy(Iy==o.Ny+1) = o.Ny;
end
function [yrm,Iy] = yrm(o,y)
    %YRM -- relative coordinates, center oriented
    % USAGE: [yrm,iy] = gr.yrm(y)
    [yrm,Iy] = o.yr(y);
    yrm = yrm - 0.5;
end
function [Idx,Ix,Iy,xr,yr,xrm,yrm] = Idx(o,x,y)
    %IDX -- global index of points x,y
    % USAGE: [Idx,Ix,Iy,xr,yr,xrm,yrm] = Idx(x,y)
    [xr,Ix] = o.xr(x);
```

```
    [yr,Iy] = o.yr(y);
    Idx = o.Ny*(Ix-1)+Iy;
    xrm = xr - 0.5;
    yrm = yr - 0.5;
end
function Ix = Ix(o,x), Ix = max(1,ceil(interp1(o.xGr,0:o.Nx,x))); end
function Iy = Iy(o,y), Iy = max(1,ceil(interp1(o.yGr,0:o.Ny,y))); end
function ux = ux(o,x), [~,ux] = o.xrm(x); end
function uy = uy(o,y), [~,uy] = o.yrm(y); end

% Dependent variables
function Ny = get.Ny(o), Ny = numel(o.yGr)-1; end
function Nx = get.Nx(o), Nx = numel(o.xGr)-1; end
function Nz = get.Nz(o), Nz = numel(o.zGr)-1; end
function Nod= get.Nod(o), Nod = o.Ny*o.Nx*o.Nz; end
function dx = get.dx(o), dx = abs(diff(o.xGr,1,2)); end
function dy = get.dy(o), dy = abs(diff(o.yGr,1,1)); end
function dz = get.dz(o), dz = abs(diff(o.zGr,1,3)); end
function xm = get.xm(o), xm = 0.5*(o.xGr(1:end-1) + o.xGr(2:end)); end
function ym = get.ym(o), ym = 0.5*(o.yGr(1:end-1) + o.yGr(2:end)); end
function zm = get.zm(o), zm = 0.5*(o.zGr(1:end-1) + o.zGr(2:end)); end
function xp = get.xp(o), xp = o.xGr(2:end-1); end
function yp = get.yp(o), yp = o.yGr;          end
function XGr = get.XGr(o), XGr = ones(size(o.yGr)) * o.xGr; end
function YGr = get.YGr(o), YGr = o.yGr * ones(size(o.xGr)); end
function dX  = get.dX(o), dX = ones(o.Ny,1) * o.dx; end
function dY  = get.dY(o), dY = o.dy * ones(1,o.Nx); end
function dZ  = get.dZ(o), dZ = abs(diff(o.Z  ,1,3)); end
function Xm  = get.Xm(o), Xm = ones(o.Ny,1) * o.xm; end
function Ym  = get.Ym(o), Ym = o.ym * ones(1,o.Nx); end
function Zm  = get.Zm(o), Zm = 0.5*(o.Z(:,:,1:end-1)+o.Z(:,:,2:end)); end
function Xp  = get.Xp(o), Xp = ones(size(o.yp)) * o.xp; end
function Yp  = get.Yp(o), Yp = o.yp * ones(size(o.xp)); end
function Area = get.Area(o)
    if ~o.AXIAL, Area = o.dy * o.dx;
    else         Area = pi.*(o.xGr(2:end).^2-o.xGr(1:end-1).^2);
    end
end
function AREA = get.AREA(o), AREA = sum(o.Area(:)); end
function Vol = get.Vol(o)
    if ~o.AXIAL, Vol = bsxfun(@times,o.Area,o.dZ);
    else         Vol = bsxfun(@times,o.Area,o.dy);
    end
end
function VOL  = get.VOL(o),  VOL = sum(o.Vol(:));  end

%aliases for x-variables in axially symmetric cases
function Nr  = get.Nr(o),  Nr  = o.Nx;  end
function rGr = get.rGr(o), rGr = o.xGr; end
function RGr = get.RGr(o), RGr = o.RGr; end
function rm  = get.rm(o),  rm  = o.xm;  end
function Rm  = get.Rm(o),  Rm  = o.Xm;  end
function dr  = get.dr(o),  dr  = o.dx;  end
```

```
        function dR  = get.dR(o),  dR  = o.dR;  end
        function rp  = get.rp(o),  rp  = o.xp;  end
        function Rp  = get.Rp(o),  Rp  = o.Xp;  end
    end
end
```

2.3 The actual model in Matlab

2.3.1 Implementation

The model will be a Matlab function that accepts the needed arguments and yields the heads and flows. This function resides in an "*m.file*" which is Matlab's text file storage of scripts and functions. *mfiles* have extension ".m". A second *m.file* will be a Matlab script, which is just set of commands in a file. This script will be used to set up the model, specify its boundary conditions, call the function (i.e. the actual model) and finally visualize the results by contouring the computed heads. These model scripts are described under the examples.

To make the model in Matlab, launch Matlab browse to the directory where you want to store the *mfiles* of this model.

Then start with opening a new file by pressing the correct icon of the Matlab editor and immediately save it with the desired model name, for instance *fdm*2. Then "*fdm2.m*" will be the file name given by Matlab and *fdm2* the name of the Matlab function it contains.

The first line in *fdm2.m* specifies the function name, and its outputs and inputs.

The first line should be the function heading:

```
function [Phi,Q,Qx,Qy,Psi]=fdm(gr,Tx,Ty,IBOUND,ID,FQ)
```

It defines a function *fdm2* with arguments to be passed to it, which will be local inside the function. It also defines its output, which may be multiple arrays as is the case here, where we will obtain the computed heads, the computed nodal flows, the computed horizontal and vertical flows across cell faces and the stream function, whic is called Psi.

Below the function definition line, insert a number of comment lines (i.e. all starting with "%") to provide the information that Matlab sends to the screen whenever you type

```
help fdm2
```

in the command window.

The actual model is listed below (in about 50 lines of Matlab).

The model must be set up and debugged line by line. This is done by selecting one or more lines, running them by pressing F9 (on Windows) or shift F7 (on Mac) checking if they are correct. Once all lines run smoothly and correctly, remove the comment "%" in the first line of the *modelScript* file. This turns the contents of the file into a function. Also change the call to the file *fdm2* into a function call:

```
fdm2; % [Phi,Q,Qx,Qy,Psi]=fdm2(gr,Tx,Ty,IBOUND,STRTHD,FQ); % call the model
```

becomes

```
[Phi,Q,Qx,Qy,Psi]=fdm2(gr,Tx,Ty,IBOUND,STRTHD,FQ); % call the model
```

Then the model can be run with any changed input.

Note: There are no error checks yet in the model. This is to keep the file short for he sake of this syllabus. You may add checks as desired.

2.3.2 Listing of simple model fdm2

```
function [Phi,Q,Qx,Qy,Psi]=fdm2(gr,Tx,Ty,IBOUND,Phi,Q)
%FDM2 a 2D block-centred steady-state finite difference model
% USAGE:
%    [Phi,Q,Qx,Qy,Psi]=fdm2(gr,Tx,Ty,IBOUND,FH,FQ)
% Inputs:
%    gr    = grid2DObj (see grid2DObj)
%    Tx,Ty = transmissivities, either scalar or full 2D arrays
%    IBOUND= boudary array as in MODFLOW (<0 fixed head, 0 inactive, >0 active)
%    IH    = initial heads (STRTHD in MODFLOW)
%    FQ    = fixed (prescribed) Q for each cell.
% Outputs:
%    Phi,Q computed heads and cell balances
%    Qx(Ny,Nx-1) horizontal cell face flow positive in positive xGr direction
%    Qy(Ny-1,Nx) vertial    cell face flow, postive in positive yGr direction
%    Psi(Ny+1,Nx-2)  stream function (only useful if flow is divergence free)
% TO 991017  TO 000530 001026 070414 090314 101130 140410

Nodes = reshape(1:gr.Nod,gr.size);               % Node numbering
IE=Nodes(:,2:end);   IW=Nodes(:,1:end-1);
IS=Nodes(2:end,:);   IN=Nodes(1:end-1,:);

Iact  = IBOUND(:) >0;
Inact = IBOUND(:)==0; Tx(Inact)=0; Ty(Inact)=0;
Ifh   = IBOUND(:) <0;

% resistances and conducctances
RX = 0.5*bsxfun(@rdivide,gr.dx,gr.dy)./Tx;
RY = 0.5*bsxfun(@rdivide,gr.dy,gr.dx)./Ty;

Cx = 1./(RX(:,1:end-1)+RX(:,2:end));
Cy = 1./(RY(1:end-1,:)+RY(2:end,:));

C     = sparse([IW(:); IN(:)], [IE(:); IS(:)], [-Cx(:); -Cy(:)], gr.Nod, gr.Nod, 5*gr.Nod);
C     = C + C';
diagC = -sum(C,2);  % Main diagonal

Phi = Phi(:); Phi(Inact)=NaN;
Q   = Q(  :); Q(  Inact)=NaN;

Phi(Iact) = spdiags(diagC(  Iact),0,C( Iact , Iact ))\(Q(Iact)-C(Iact,Ifh)*Phi(Ifh)); % solve
Q(~Inact) = spdiags(diagC(~Inact),0,C(~Inact,~Inact))* Phi(~Inact);        % reshape

Phi = reshape(Phi,gr.size);
Q   = reshape(Q  ,gr.size);

Qx = -Cx.*diff(Phi,1,2);   % Flow across vertical   cell faces
Qy = +Cy.*diff(Phi,1,1);   % Flow across horizontal cell faces

Psi= [flipud(cumsum(flipud(Qx)));zeros(size(Qx(1,:)))];
```

2.4 Exercises

These exercises not only demonstrate some of the possible uses of the obtained model. These also show how to prove that your model is correct. This is done by comparing with exact analytical solutions. Proving the correctness of numerical models is always essential. Not only has the code to be verified, but also the boundaries. It is generally possible to change the boundaries such that the model can be compared with analytical solutions.

2.4.1 Ecercise 1: Circular island with recharge

The first exercise is contained in the *modelScrip1t* listed below. It uses a mesh that runs from -1000 to 1000 with steps of d both in x direction and in y direction. This info suffices to generate a grid by calling the *grid2DObj* with the xGr and yGr coordinates. The recharge is constant and equal to 0.001 m/d. The recharge inflow into each cell is then equal to the cell area times the *rch*. The cell area is obtained from the *grid2DObj*, gr. This recharge provides the total actual inflow in the cells. Transmissivity Tx and Ty are uniform and equal to 600 m2/d. The starting heads are also uniform and equal to 0.

Although the area is rectangular, the model is actually a circular island. This is achieved by computing the distance for each cell center to the center of the island at $x = 0$,$y = 0$. Then all cells that are beyond radius $R0 = 800$ m are converted to fixed head cells and therefore, form a fixed-head boundary of the island.

With this information the model is called. It yields several outcomes as defined within it and in that order: The full array of heads, the full array of computed cell inflows from the outside world, the flow in x-direction over the cell faces (perpendicular to the y-axis) and the flow across the cell faces into the y-direction perpendicular to the x-axis. Notice that the model can also provide the stream function, but because the flow is not divergence free as a consequence of the recharge, this stream function is useless in this case.

After the model has run, its results are visualized by color contouring of heads (using *contourf(...)*) and verified by drawing contours in black and by plotting the heads along a line through the center of the island. As can be seen, there is a small difference between the analytic and the numerical results of the cell width, $d = 25$ m, but this difference vanishes the smaller d is, try $d = 5$, for instance.

Finally, the water balance is quantified using some *fprintf* statements. First for the area as a whole, then for the area with recharge and, finally, for the discharge through the cells with the fixed heads.

ModelScript1 sets up the model dimensions, specifies the cell properties, the fixed/initial head boundaries, and the flow boundary conditions for all nodes. After this it runs the model. Finally, the heads are contoured and some integrated flows are computed to check the model. Note that the heads are computed for the cell centers xm, ym.

The results is shown in the figure.

2.4.2 Listing of modelScript1

```
%%% modelScript -- sets up 2D FDM, runs it and shows results

FIXED = -1;   % indicatre fixed head node in IBOUND

%% Grid
d   = 25; % cell width, also try 5 m
xGr = [-1000:d:-d -d/2 d/2 d:d:1000];
yGr = [-1000:d:-d -d/2 d/2 d:d:1000];
gr  =  grid2DObj(xGr,yGr);

%% Island
R0     =  800; % +d/2;
R      = sqrt(gr.Xm.^2 + gr.Ym.^2);
```

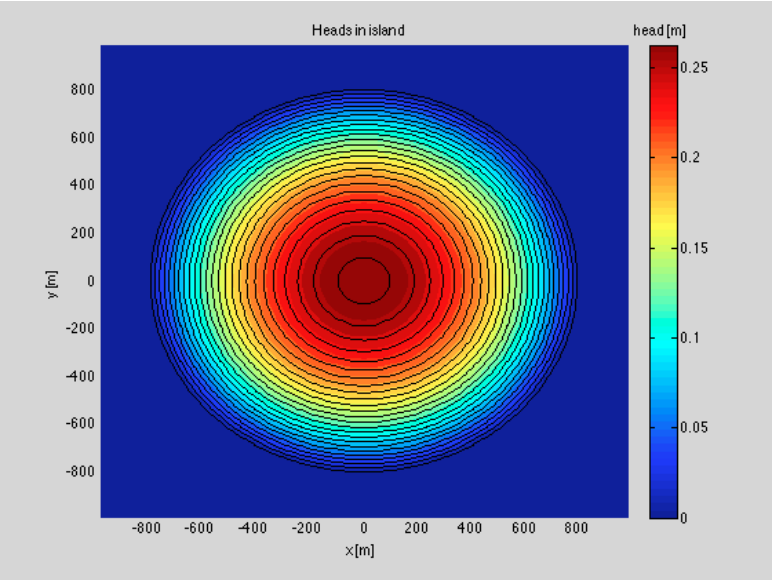


Figure 2.2: Heads in circular island

```
IBOUND = ones(gr.size);
IBOUND(R>=R0) = FIXED;

%% Transmissivities
Tx = 600; Ty = 600;

%% Start heads, initial heads
IH = gr.const(0);

%% Fixed inflows of cells
rch = 0.001; % net recharge rate
FQ  = gr.Area * rch;

%% Run model
[Phi,Q,Qx,Qy] = fdm2(gr,Tx,Ty,IBOUND,IH,FQ);

%% Visualize results
figure; hold on;
xlabel('x [m]'); ylabel('y [m]');
title('Heads in island');

phiMax = max(Phi(:));  hRange = phiMax*(0:25)/25;
contourf(gr.xm,gr.ym,Phi,hRange,'edgeColor','none');

hb=colorbar; set(get(hb,'title'),'string','head [m]');

%% Analytical comparison

% Analytical solution for head in island
PhiA = rch/4./Tx .* (R0.^2 - R.^2);

% black contours over the colored contours
contour(gr.xm,gr.ym,PhiA,hRange,'k')
```

```
%% Plot X-section to compare with analytical results
figure; hold on;
xlabel('x [m]'); ylabel('head [m]');
title('head through center of island');

Iy = find(gr.ym==0);

plot(gr.xm,Phi (Iy,:),'b');
plot(gr.xm,PhiA(Iy,:),'r.');

%% Check water balance
fprintf('Water balances:\n');
fprintf('Total water balance = %10g (should be zero)\n',sum(Q(:)));
fprintf('Total recharge      = %10.0f m3/d (should be pi*R0^2*rch = %6.0f m3/d)\n',sum(Q(Q>0)),p
fprintf('Total dicharge      = %10.0f m3/d (should match with total recharge.\n',sum(Q(Q<0)));
```

2.4.3 Exercise 2: Island with with arbitrary boundary, an internal lake and an inactive area with wells

2.4.3.1 Approach and result

To demonstrate the power of this simple model as well as of the Matlab environment, our next model will have an arbitrary boundary with an extra internal lake and a part of the area where the aquifer does not exist, i.e. where cells are inactive. We will further add some wells. Of course, one can also spatially vary Tx, Ty and rch in any way without any consequence for the code,

One can digitize the area of an arbitrary island, a lake and an inactive area on screen using the function *ginput* (see doc ginput) or, more conveniently using the function *rbline* (rubber band line) which was obtained from the central Matlab library (it was made by a user). To get the coordinates of for instance the island to be digitized, first plot the model and then apply the *rbline*.

```
figure; hold on;
gr.plot('y');
xy = rbline;
```

The digitizing is interrupted by pressing any key instead of a mouse button. The results are the coordinates just digitized and agreeing with the axes of the figure. To store it, select the coordinates x,y, open a new file and paste them. Then give a name and place brackets around them to make sure you get a regular Matlab assignment.

```
Island = [
      -919.35      -704.68
      -753.46      -640.35
      ...  % coordinates left out because of space
       -942.4      -769.01
      -919.35      -704.68
       ];
Lake = [
      -11.521       669.59
       34.562       576.02
  ...
      -39.171       722.22
      -11.521       669.59
      ];
Rock = [
```

```
    -324.88      -154.97
     43.779      -137.43
     57.604      -301.17
    -347.93      -342.11
    -324.88      -154.97
    ];
well = [
    -462.73        343.6  -300
     281.77       326.28  -300
    -12.925      -8.6623  -300
   ];
```

Save the file under the name *'example2Data.m'* so that is can be read in or executed from within our *modelScript2*, which is listed below.

ModelScript2 can be run and the results inspected. One can also issue the publish command:

```
publish('modelScript2');
```

to obtain a nicely formatted results with the outputs embedded.

The resulting figure is shown here

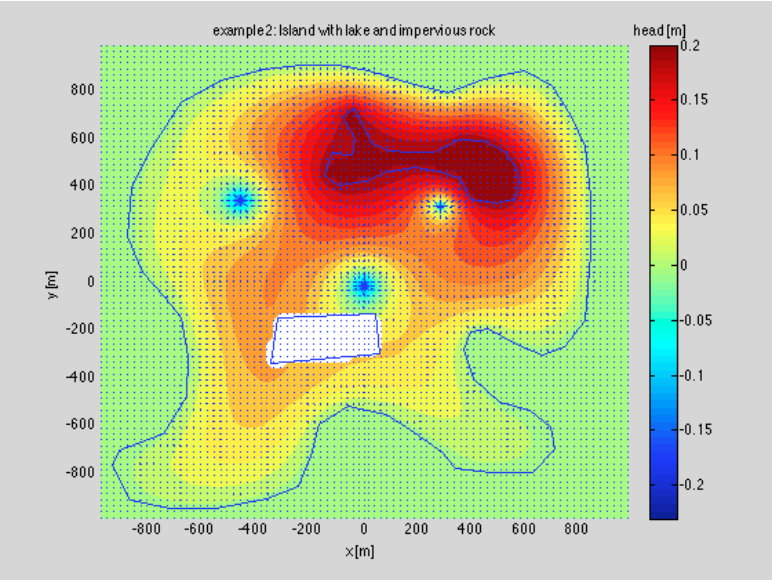


Figure 2.3: Results example2 showing head. Blue lines were drawn around the island, the lake with fixed head and the inactive area. Wells are also clearly visible. Small arrows indicate magnitude and direction of the discharge. Recharge is uniform, 0.001 m/d.

2.4.3.2 Listing of modelScript2

```
%% ModelScript -- Example 2 sets up 2D FDM, runs it and shows results

%% Arbitrary island, with an internal lake, an impervious area and wells

FXHD     = -1;
INACTIVE =  0;
ACTIVE   =  1;

%% Grid
d      = 25; % 5   % cell width
```

```
xGr   = [-1000:d:-d -d/2 d/2 d:d:1000];
yGr   = [-1000:d:-d -d/2 d/2 d:d:1000];
gr    =  grid2DObj(xGr,yGr);

%% Reads island, lake and rock contours and wells and termine cells within
example2Data;
inIsland = inpolygon(gr.Xm,gr.Ym,Island(:,1),Island(:,2));
inLake   = inpolygon(gr.Xm,gr.Ym,  Lake(:,1),  Lake(:,2));
inRock   = inpolygon(gr.Xm,gr.Ym,  Rock(:,1),  Rock(:,2));

%% Transmissivities
Tx = gr.const(600);
Ty = gr.const(600);

%% Recharge
rch   = 0.001; % net recharge rate
FQ    = gr.Area * rch;

%% Wells
for iw=1:size(well,1)
    ix = find(gr.xm>=well(iw,1),1,'first');
    iy = find(gr.ym<=well(iw,2),1,'first');
    FQ(iy,ix) = FQ(iy,iw) +well(iw,3);
end

%% IBOUND
IBOUND = zeros(gr.size);
IBOUND( inIsland) = ACTIVE;
IBOUND(~inIsland) = FXHD;
IBOUND( inLake)   = FXHD;
IBOUND( inRock)   = INACTIVE;

%% Initial and fixed heads
hLake = 0.2;
hSea  = 0;

IH    = gr.const(0);
IH(~inIsland) = hSea;
IH( inLake)   = hLake;

%% Run model
[Phi,Q,Qx,Qy] = fdm2(gr,Tx,Ty,IBOUND,IH,FQ);

%% Visualize results

figure; hold on;
xlabel('x [m]'); ylabel('y [m]');
title('example2: Island with lake and impervious rock');

% Contours of head
phiMax = max(Phi(:)); phiMin = min(Phi(:));  hRange = phiMin:(phiMax-phiMin)/25:phiMax;
contourf(gr.xm,gr.ym,Phi,hRange,'edgeColor','none');
```

```
% Draw line around island and lake
plot(Island(:,1),Island(:,2),'b');
plot(  Lake(:,1),  Lake(:,2),'b');
plot(  Rock(:,1),  Rock(:,2),'b');

% Show arrows of flow direction and magnitude
qx = [Qx(:,1), Qx, Qx(:,end)]; qx = 0.5*(qx(:,1:end-1) + qx(:,2:end));
qy = [Qy(1,:); Qy; Qy(end,:)]; qy = 0.5*(qy(1:end-1,:) + qy(2:end,:));
quiver(gr.Xm,gr.Ym,qx,qy);

hb = colorbar; set(get(hb,'title'),'string','head [m]')  % Colorbar

%% Check water balance
fprintf('Water balances:\n');
fprintf('Total water balance = %10g (should be zero)\n',sum(Q(IBOUND~=0)));
fprintf('Total recharge (active  cells) = %10.0f m3/d\n',sum(Q(IBOUND>0)));
fprintf('Total discharge(fixhd + wells) = %10.0f m3/d\n',sum(Q(IBOUND<0)));
```

2.4.4 Exercise3: Cross section with recharge

Compute the heads in a 1d model with recharge

We will check the head ϕ in a cross section through an aquifer of given width $2L$ and recharge N. The analytical solution can be found in Geohydrology I and is easily derived:

$$\phi = \frac{N}{2kD}\left(L^2 - x^2\right)$$

with x measured from the water divide to the fixed head boundary.

```
% modelScript to compute this case and compare with analytical solution
kD  =  10;              % transmissivity to be used
L   = 200;              % half width of model
xGr = -L:5:L;           % generate x-coordinates for mesh
yGr = [0 -25]';         % one row suffices because problem is 1D

IH     = gr.const(0);
IBOUND = gr.const(1);
IBOUND(:,[1,end])=FIXED;

N = 0.001;
FQ = gr.const(N);

[Phi,Q,Qx]=fdm2(gr,Tx,Ty,IH,FQ);           % Run model

fi = N/(2*kD)*(L.^2-xm.^2);                 % Analytical solution

figure; xlabel('x [m]'); ylabel('head [m]');
title('example3, head in 1D aquifer with recharge');
plot(gr.xm,Phi,'r+',gr.xm,fi,'b');                % Plot results
legend('model','analytic');
```

The results are given in figure 2.4. It is clear that the numerical and analytical solutions do not match completely. This is due to the fact that the boundary nodes of the numerical model are not exactly at $x = \pm L$, but at the cell centers, which is at $x = -L + 2.5 = -197.5\,\text{m}$ and at $x = L - 2.5 = +197.5\,\text{m}$.

To solve this you may add a very thin outer cell at both ends, or set the x-coordintes as follows

```
x=-L-2.5:5:L+2.5; % generate x-coordinates for mesh
```

This will yield a perfect match between model and analytical solution.

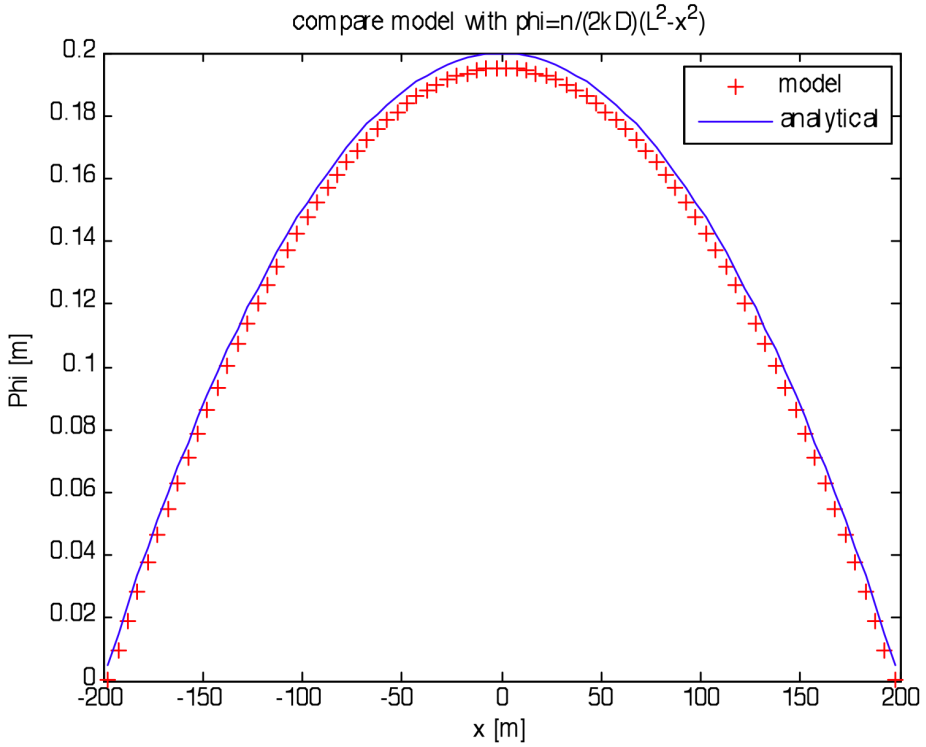


Figure 2.4: Comparison between model and analytical solution. The match is not perfect due to the fact that the exact locations of the outer cells in the model do not coincide with –L and L, the exact location of the outer nodes is in the cell center, see below to solve this.

2.4.5 Exercise: Cross section with leakage

Compute the heads in a cross section through an aquifer with transmissivty $kD = 1600\,\mathrm{m}^2/\mathrm{d}$ that is bounded to the left and right by surface water with fixed water level $H = -3\,\mathrm{m}$. The surface water is in direct contact with the aquifer, so there is no entry resistance. The aquifer is covered by a layer with known vertical resistance $c = 100\,\mathrm{d}$. On top of this aquifer the water level is maintained by numerous ditches with head maintained at $h = 0\,\mathrm{m}$. It is a typical Dutch situation. Take the start of the x-axis in the center of the cross section, which is 2L wide. The head at the top of

symmetrical with $x = 0$ in the center.

This problem is one-dimensional and involves the heads at two levels, that is in the aquifer and on top of the covering layer.. We may, therefore, solve this one-dimensional problem with a model consisting of two rows, one for the aquitard with resistance c and one for the aquifer with given transmissivity kD.

A possible Matlab script to solve this problem and visualize its results is given below. It uses the function $fdm2.m$ to compute the actual heads (and flows).

```
% Cross section through polder with fixed head H at both sides
L = 1000; kD=1600; c=100; lambda=sqrt(kD*c); % +/- Xsec of Bethune polder
xGr = [-L-5:10:L+5];
yGr = [0 -10 -40]';
gr = grid2DObj(xGr,yGr);

k = [gr.dy(1)/c kD/gr.dy(2)]';   % conductivities from c, kD and thickness
kx= gr.const(k);
```

```
H  = -2.75;                % head at left and right boundary
IH = NaN*ones(Ny,Nx);      % NaN matrix to store fixed heads
IH(1,:)=0;                 % head above aquitard
IH(2,[1,end])=H;           % head at left and right boundary

FQ = zeros(Ny,Nx);         % matrix to store fixed Qś

[Phi,Q,Qx]=fdm2(x,y,kx,ky,IH,FQ);  % run model

fi = H*cosh(xm/Lambda)./cosh(L/Lambda); % analytical

plot(xm,Phi,'+',xm,fi,'b');
title('compare model with phi=H*cosh(x/lambda)/cosh(L/lambda)');
xlabel('x [m]'); ylabel('Phi [m]');
legend('fixed head','model','analytical');
```

See text below to solve.

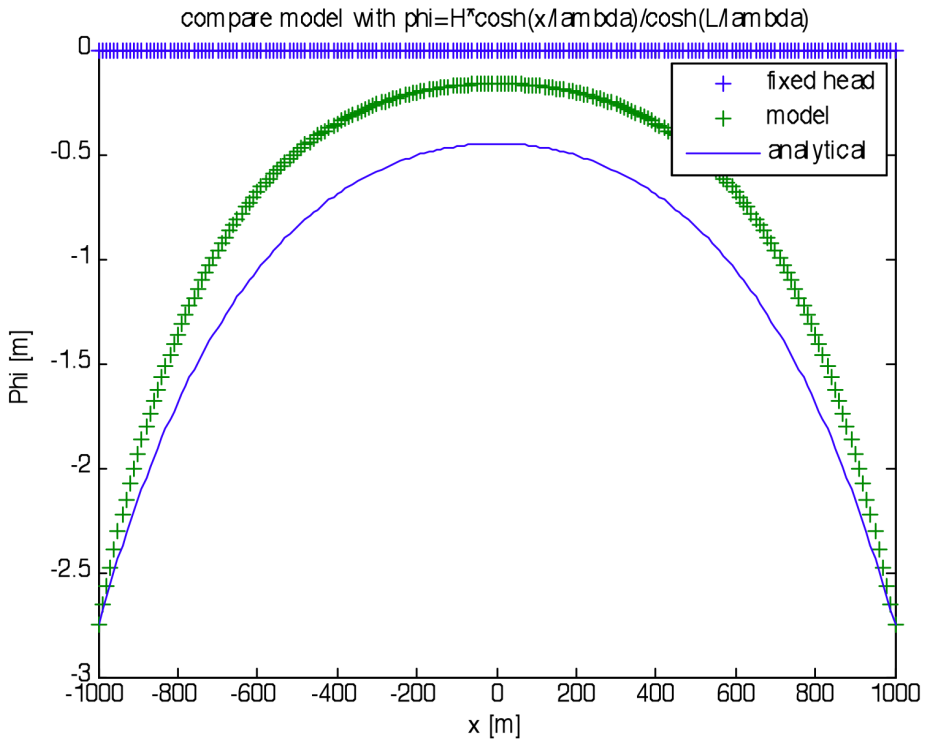


Figure 2.5: Comparison between model and analytical solution of cross section through polder. Obviously there is a large difference between the analytical and numerical solution. See text to resolve this problem.

Figure 2.5 shows the fixed heads and the numerically and analytically computed heads in the aquifer. Clearly, the model is way off compared with the analytical solution. This time we made sure that the center of the outer cells are exactly on the the boundaries. Therefore, there must be another and possibly more important reason.

We expect the leakage water to pass the top layer. However, it only passes half the top layer. This is because we compute the heads in the cell centers, which is midway between top and bottom. The leakage water therefore, only experiences half the resistance of the top aquitard.

To solve this, you may use a separate thin extra layer on top and specify the head in that one. Or you may double the thickness of the first layer so that the resistance between the node and the bottom of this layer equals the desired value. Or you just half the vertical conductivity of the first layer to get the same result. So, choosing the latter solution, we set

```
k=[0.5*dy(1)/c; kD/dy(2)]; % conductivities from c, kD and thickness
```

2.4.6 Exercise: Put a number of wells in the island and compare with the analytical solution

For this problem refer to the book Fitts (2002) Groundwater Science, section 6.2.9, page 187-190. It shows that image wells can be used to analytically compute the head within circular islands even though the wells are not at the center. If the distance from a well in the island to the center of the island is d_1, then the distance of the mirror well to the center of the island will be $d_2 = R^2/d_1$. In general with a well and its mirror well in a confined aquifer we have the solution

$$h - H = \frac{Q}{2\pi kD} \ln\left(\frac{r_1}{r_2}\right) + C$$

Where C follows from the requirement that h-H=0 for points at the boundary. As soon as the position of the well is given, we known d_1 and d_2, so

$$0 = \frac{Q}{2\pi kD} \ln\left(\frac{R - d_1}{d_2 - R}\right) + C$$

so that, using the index i to denote a specific well out of an arbitrary number of wells

$$C_i = \frac{Q_i}{2\pi kD} \ln\left(\frac{d_{2,i} - R}{R - d_{1,i}}\right)$$

and so

$$h - H = \sum_{i=1}^{N} \frac{Q_i}{2\pi kD} \ln\left(\frac{r_{1,i}}{r_{2,i}} \times \frac{d_{2,i} - R}{R - d_{1,i}}\right)$$

This drawdown due to the wells should be added to the head increase caused by the recharge.

Clearly, in the numerical model you may just put the well as an extraction in the cells that correspond to the well location.

Visualize you results by overlaying the contours of the numerical and the analytical solution.

Use three wells at locations (120,0), (30,100) and (-130,-20) with extractions 200, 120 and 300 m^3/d.

2.4.7 Exercise: Show the effect of anisotropy?

Anisotropic situations can be readily computed if the main conductivities align with the x and y axes of the model grid. However, it is not straightforward to apply anisotropy in arbitrary directions, unless the model grid can be rotated to align it with the main conductivity directions. In the finite element method, anisotropy in arbitrary direction in each cell is natural. In the finite difference model it is generally either limited to the main directions of the grid itself, or one must use a more advance technique, which involves more neighboring cells to copy with directional anisotropy in rectangular grids. This is beyond this course.

However, applying anisotropy aligned with the grid is straightforward. One only has to change the k_x conductivity with respect to the k_y conductivity.

Exercise, multiply dx by 3 and divide dy by 3 in the previous problem. Visualize the results as before. Make sure the x- and y-axes are at the same scale so that the anisotropy is clearly visible. To do this in Matlab use

```
axes equal; axes tight;
```

Compare the discharge that the model computes.

2.4.8 Exercise: Generate a random conductivity field and compute the heads given fixed head boundaries.

A random conductivity field may be generated using random functions, through geostatistical methods or for the sake of an exercise using the same method as in the previous example. Compare the results.

In conclusion, is may be quite difficult to separate the influence of varying recharge from that of heterogeneity. This is problem is manifest in model calibration, where the influence of spatial recharge variability and heterogeneity correlate in the resulting head distribution, which in turn is used as, at least in most calibration exercises, the main source of information in model calibration. If possible, recharge information should be obtained through independent means, so that it does not have to be calibrated. The result will be a much improved confidence in the obtained transmissivities.

2.4.9 Exercise: Generate a river through your model and compute the heads with recharge

A river is a set of lines with fixed heads or heads that are fixed through a resistance, which are called "general head boundaries" in Matlab. Normally assigning rivers to a grid is a GIS action. The river has to be intersected with the model cells. For each cell the intersecting surface area $A\,[\mathrm{m}^2]$ is computed and converted into a conductance $C\,[\mathrm{m}^2/\mathrm{d}]$ using the bottom resistance c [d] of the river . So we skip this GIS action for now.

3 Stream lines

3.1 Theory and implementation

Streamlines are lines of constant stream-function value. Flow lines are lines followed by particles. Therefore, flow lines have to be computed by tracing particles, but streamlines (if they exist) can be computed by contouring the stream function, without tracing. However, the stream function is only defined in 2D steady-sate flow without sources and sinks (and leakage or recharge for that matter). In practice, individual sources and sinks can be dealt with and will look similar to wells in the 2D image.

Because our model is 2D, streamlines will often be an efficient manner to show the flow in a quantitative way. A very powerful characteristic of streamlines is that the flow is known between any pair of points in the model. Further, the flow between any pair of streamlines is constant and equal to the difference of the stream function values.

With respect to the stream function, any streamline may be designated the zero line, after which the values of all other streamlines are fixed. The stream function can be computed by integrating the flow across an arbitrary line cutting streamlines. Assuming the bottom of the model is a streamline, we can compute the stream function easily by integrating the horizontal flow across cell faces from the bottom to the top of the model.

Mathematically

$$\Psi = \int_{y_{min}}^{y_{mzx}} q_x\left(y\right) dy$$

When we use the horizontal flow across the cell faces as an extra output of the model, we just cumulative these along the cell faces upward from the bottom of the model. This gives the stream function values at all cell corners (not the nodes). This stream function may subsequently be contoured, which yields stream lines.

To implement the stream function, we just add this line to the finite difference model that we already have and add the new parameters Psi to the list of outputs.

```
Psi= [flipud(cumsum(flipud(Qx)));zeros(size(Qx(1,:)))];
```

This line does the following. It receives the horizontal flows across the cell faces, which is the third output of *fdm2*. It adds a line of zeros through the bottom, because this will be the starting stream line with stream function value zero. Then we want to cumulate this matrix vertically from the bottom upward. Matlab's *cumsum(..)* accumulates matrices (try it), but starts at the top working downward. So we flip the matrix *Qx* up-down before calling *cumsum(..)*. When done, we *flipud(..)* again to put it right.

Next, add the following lines to your script file

```
contour(x(2:end-1),y,Psi(Qx));  % streamlines
```

The values of the stream function is the flow between any point and the bottom of the model. If you want that in specific steps of say $dPsi = 0.1\,\mathrm{m}^2/\mathrm{d}$, just specify the desired contours, for instance.

```
contour(x(2:end-1),y,Psi(Qx),[0:dPsi:max(Psi(Qx(:)))]); % streamlines
```

See *help contour* in Matlab for details.

Streamlines cause so-called branch cuts whenever an extraction or injection takes place inside the model. But because these branch cuts are vertical along the vertical cell faces, and run to the top of

the model, they look just like extraction wells, and do not disturb in a cross section. Notice that the jump in Psi across a branch cut is exactly equal to the extraction below the considered point in the branch cut.

Streamlines can also be computed by summation of the vertical flows across the horizontal cell faces starting from the left edge of the model. However, the branch cuts will then be horizontal looking like drains that discharge water across the left edge of the model. This makes for unpleasant graphics when wanting to show vertical well extractions in cross sections. This is whey it was chosen to compute the stream function from the bottom of the model upward.

3.2 Exercises with streamlines

3.2.1 Exercise: Building pit with wells inside a sheet piling

3.2.1.1 Background

Consider a symmetric cross section through a long building with sheet pilings 15 m deep at $x = 20\,\mathrm{m}$. Dewatering wells are placed inside this sheet piling between 6 and 11 m depth. Compute the necessary extraction to dewater the pit by 5 m. The aquifer is semi-confined. All elevations are relative to the fixed head at the top. The sheet piling is between $x = 19.9\,\mathrm{m}$ and x=20 m, and $z = 0\,\mathrm{m}$ and $z = -15\,\mathrm{m}$ with conductivity $k_W = 0.0001\,\mathrm{m/d}$. Wells are between $x = 19.8\,\mathrm{m}$ and $19.9\,\mathrm{m}$ and between $z = -6\,\mathrm{m}$ and $z = -11\,\mathrm{m}$ and are modeled as an extraction line in this cross section. The layers are given as in the Matlab script below.

The results are shown in the two figures. One shows the total cross section and the second one a detail. This detail demonstrates the streamlines and the head lines in this cross section in the neighborhood of the wells and the sheet piling. It clearly shows how the groundwater flows underneath the 15 m deep sheet piling towards the wells at the inside of this sheet piling between -6 and -11 m. The extraction is $8.67\,\mathrm{m}^2/\mathrm{d}$, which can be computed by summing the Q over the wells. Alternatively one may compute the Q entering the model through the first layer

```
sum(Q(1,:))
k=[0.5*dy(1)/c; kD/dy(2)]; % conductivities from c, kD and thickness
```

We set the fixed heads, array FH, of the wells at -5 m. The head in the center immediately below the building pit is then -4.6 m. By setting FH in the wells to -5.4 we make sure that the drawdown under the building pit is the required 5 m. The extraction will then be $9.38\,\mathrm{m}^2/\mathrm{d}$. This demonstrates the influence of partial penetration of the extraction wells.

Partial penetration means that the well screen only penetrates part of the aquifer thickness. This implies that the drawdown is larger than in the case of a fully penetrating screen. Partial penetration is the usual case, to save well money or to prevent upcoming of brackish water from below. If a building pit must be put dry, only the head at its bottom needs to be lowered, not at the bottom of the aquifer, which may be 100 m or more thick. Using short screens then reduces the amount of water that needs to be extracted.

3.2.1.2 Implementation and results

The example is implemented in the script listed below. The detailed head contours and stream lines near the building put are obtained by zooming first horizontally and then unrestrictedly before copying the figure. To carry out separate zooming of the horizontal axes select the zoom button in the figure heading and then click right mouse button for the options. To get you figure in word, use "edit copyfigure" in Matlab in the menu of the selected figure and then paste it in Word or PowerPoint as usual.

The script also demonstrates some nice tricks with logical indexing to defined the zones inWells and inSheet (within the pile sheeting) and to generate a list of the numbers of the geological layers for all model layers using the *interp1* function combined with the floor function. Furthermore, does it

demonstrate the use of coordinates, which can be given in any order, because the *grid2DObj* will make them unique and sorts them.

After we implemented axially symmetric flow, see next section, the same script will generate results for the axially symmetric case if the *grid2DObj* is called with the 'AXIAL' option like this:

```
gr = grid2DObj(xGr,yGr,'AXIAL',true);
```

There is nothing more to it.

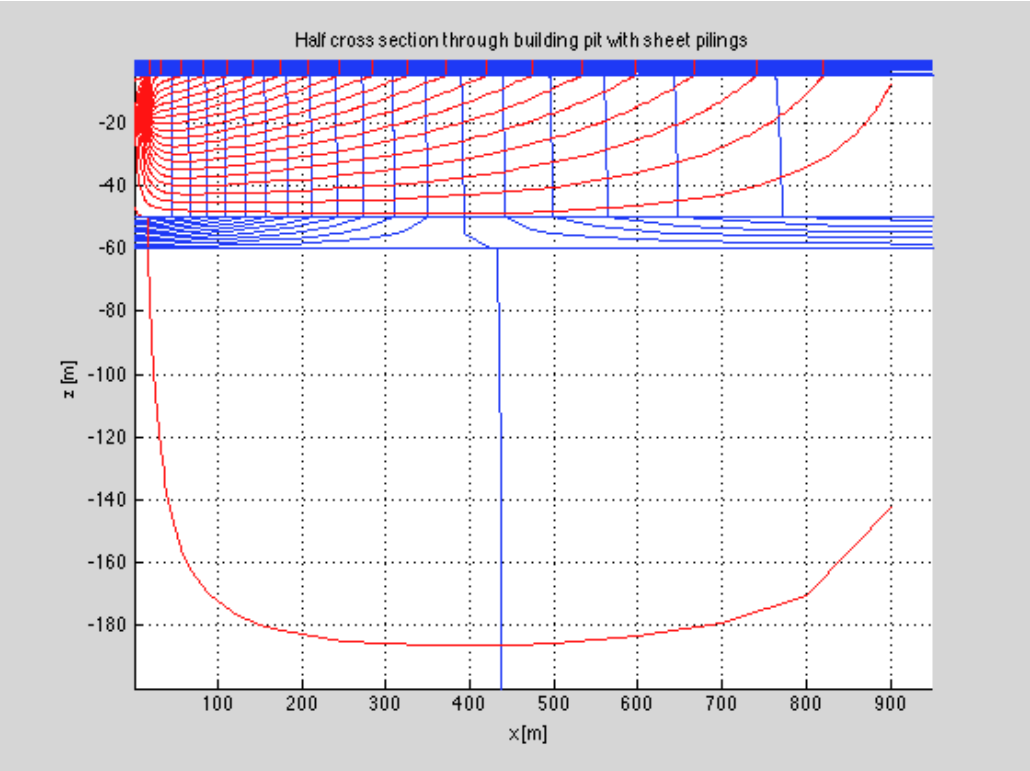


Figure 3.1: Building pit cross section with partially penetrating sheet piling and extraction wells at the inside

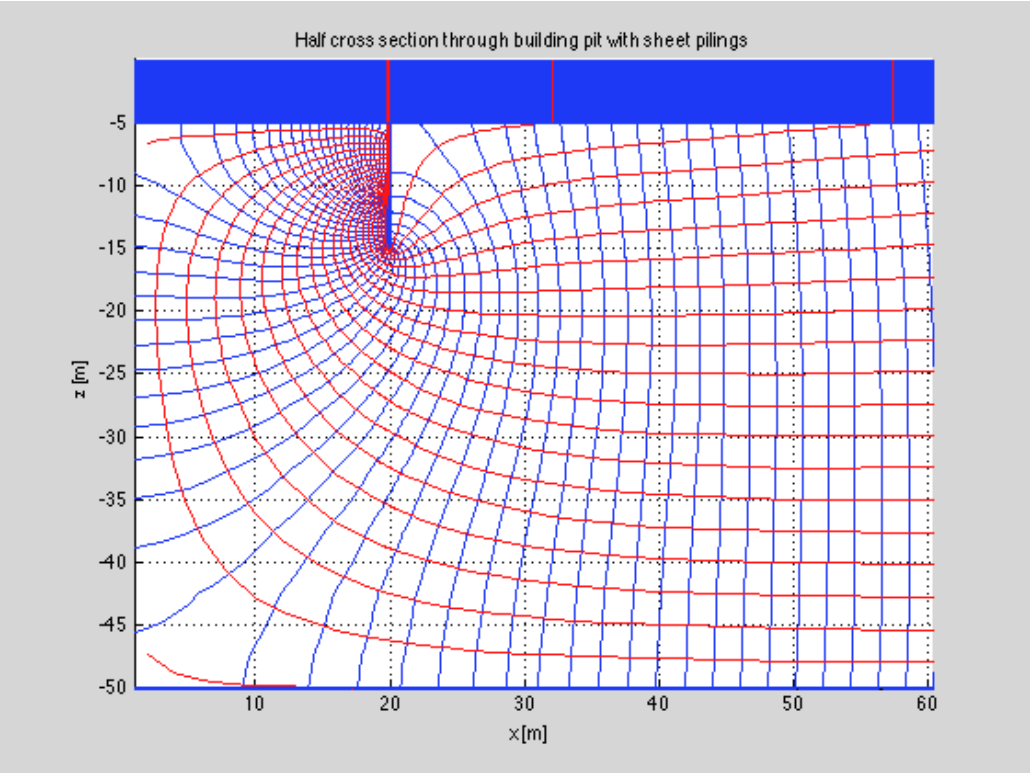


Figure 3.2: Detail showing the streamlines and head contours underneath the 15 m deep sheet piling towards the partially penetrating wells at its inside between 6 and 11 m depth

3.2.1.3 Listing of script XsecWithStreamLines

```
%% Example of a cross section with streamlines

%% Define a set of layers,
layers={
  % material, top, bottom     k
  'clay'        0       -5  0.02
  'sand'       -5      -50  20.0
  'clay'      -50      -60   0.01
  'sand'      -60      -200 30.0
};

% get k values from specified layers
top = [layers{:,2}];
bot = [layers{:,3}]; L = [top bot(end)];
kLay= [layers{:,4}];


%% Grid
% The column and row coordinates are refined where needed to have
% a very detailed result (top and bottom of wells and sheet piling
% just add coordinates then apply unique to sort out
xGr = [0:2:18, 18:0.2:22 19:0.1:21, 22:2:40, 40:10:100, ...
       100:25:250, 250:50:500, 500:100:1000];
yGr = [L L(1:end-1)-0.01 L(end)+0.01, -5:-0.1:-7, -7:-0.5:-14, -15:-0.1:-16, ...
      -16:-0.5:-19.5, -19.5:-0.1:-20.5, -20.5:-0.5:-25, -25:-5:-50];
```

```
gr = grid2DObj(xGr,yGr,'axial',0);

%% Special domains in section
xW     = [19.9 20  ]; yW    =[ 0 -15]; kW=0.0001;
xWells = [19.8 19.9]; yWells=[-6 -11]; FHWells=-5;

inWells = gr.Ym<yWells(1) & gr.Ym>yWells(2) & gr.Xm>xWells(1) & gr.Xm<xWells(2);
inSheet = gr.Ym<yW(1)     & gr.Ym>yW(2)     & gr.Xm>xW(1)     & gr.Xm<xW(2);

% Geological layer numbers for all model layers
iL= floor(interp1([top bot(end)],1:numel(kLay)+1,gr.ym));

%% Arrays
FIXHD = -1;
IBOUND = gr.const(1); IBOUND(1,:) = FIXHD;
IBOUND(inSheet) = FIXHD;
IBOUND(inWells) = FIXHD;

% Conductivities
k = gr.const(kLay(iL)');
k(inSheet)=kW;   % set k in sheet piling to kW

% Fixed heads in wells
FH = gr.const(0);
FHwells = -6;
FH(inWells)=FHwells;

% Fixed flows
FQ = gr.const(0);

%% Run model
[Phi,Q,Qx,Qy,Psi]=fdm2a(gr,k,k,IBOUND,FH,FQ);

%% Visualize

figure; axes('nextplot','add','xGrid','on','yGrid','on');
title('Half cross section through building pit with sheet pilings');
xlabel('x [m]'); ylabel('z [m]');

contour(gr.xm,gr.ym,Phi,-6:0.2:0,'b');

contour(gr.xp,gr.yp,Psi,20,'r');

for i=1:size(layers,1)
    plot(gr.xGr([1 end]),L([i i]));
end

por = 0.35;
t   = 365 * (0:10:100);
fdm2path(gr,Q,Qx,Qy,por,t,'...p...p...p');

%% Water balance and computed head below pit
sum(sum(Q(inWells)))
```

```
sum(sum(Q(1,:)))       % infiltration through top of model
sum(sum(Q))            % overall water balance
Phi(gr.ym<-5 & gr.ym>-6,1)   % head below building pit
```

3.2.2 Exercise: Add the stream lines to the 5-layer cross section of your pumping test

This gives a good view on the origin of the water and the paths it takes toward the well.

3.2.3 Exercise: Make a 5 layer vertical semi-confined cross section and show the heads in all layers if layer 4 is pumped

A multi-layer semi-confined model in a cross section is readily made with the Matlab model. The grid rows now represent layers. The conductivity of the layers determines if they represent (work as) aquifers or aquicludes. The head in the top layer is fixed. At other locations in the aquifers fixed heads or flows may be specified. This may also be done for the boundaries of the layers.

3.2.4 Exercise: Color your cross section according to the conductivities before contouring this will yield a publication-ready picture.

It is straightforward to color the conducivities using surface as shown in the box below. However, the black grid lines that it defaults to are disturbing. They can be left out by specifying 'edgeColor','none' as shown. However, the surface still masks the contours we had on the figure. To prevent that use a negative z-coordinates so that the surface will be actually below the figure (it is 3D with default Z equal to 0, our plotting plain).

```
surface(gr.xm,gr.ym,k)
surface(gr.xm,gr.ym,k,'edgeColor','none');
surface(gr,xm,gr.ym,gr.const(01),'edgeColor','none');
```

4 Axially symmetric finite difference models

4.1 Theory

An example of the results of a radial symmetric model has already been shown above. Very often we have to deal with radial symmetric flows, for instance to wells. Therefore, it comes in handy to have also a radial symmetric model that is extremely accurate, more accurate than computing radial symmetric flow with the previous model by multiplying the k_x with the distance to the left size of the model:

$$\begin{aligned} k_x &= 2\pi x_m k_x \\ k_y &= 2\pi x_m k_y \end{aligned}$$

Notice that in the radially symmetric flow case, the distance x represents the radial distance to the center at r=0 and $k_x = k_r$ while $k_y = k_z$. Normally the radially symmetric flow analysis would be done in terms of r and z rather than x and y. However, we will just adapt our finite difference model just a bit, to make it also simulate radially symmetric flow cases. Our function has all parameters expressed in x and y and so we stick to this convention.

Using a flat model this way to compute a radial symmetric flow is course a possibility and a good exercise to compare it with a truly a radial symmetric model developed hereafter by converting our flat model into a radial symmetric one, or rather one that can serve both flat and radial symmetric flow problems.

However, in order to convert our model into a radial symmetric one we have to alter its conductances. But in doing so we are not going to destroy the flat model that we already developed; instead the model is going to work for both radial symmetric and flat cases. Keeping both situations in a single Matlab function reduces maintenance in the future.

To make the model work for radial symmetric situations, the only thing to do is compute the resistance between adjacent nodes.

We know that for radial symmetric horizontal flow between two radii the logarithmic analytical solution is valid, from which the resistance against horizontal radial flow is readily derived. Given the formula for radial flow in a confined aquifer

$$\phi_1 - \phi_2 = \frac{Q}{2\pi kD} \ln\left(\frac{r_2}{r_1}\right)$$

we immediately obtain the resistance $R = \Delta\phi/Q$ across a cell with thickness $\Delta y = D$ and between r_1 and r_2 from the well:

$$R = \frac{\ln\left(\frac{r_2}{r_1}\right)}{2\pi k_x \Delta y}$$

The resistance of the right half of the cells left of the cell faces thus becomes

$$R_{x,R} = \frac{\ln\left(r_{,i+1}/r_{m,i}\right)}{2\pi k_x \Delta y}, \;\; i = 1..Nr-1$$

,

and that of the left half of the cells to the right of the cell faces becomes:

$$R_{x,L} = \frac{\ln\left(r_{m,,i}/r_{,i}\right)}{2\pi k_x \Delta y}, \;\; i = 2...Nr$$

The total resistance is then the sum of $R_{x,R}$ and $R_{x,L}$, which is an array of $Ny \times (Nr-1)$.
The vertical resistance for entire cells equals

$$R_y = \frac{\Delta y}{\pi \left(r_{i+1}^2 - r_i^2\right) k_{z,i}}$$

Or in Matlab:

```
RX =     bsxfun(@rdivide,log(gr.xGr(2:end-1)./gr.xm( 1:end-1)), ...
                         2*pi*Tx(:,1:end-1).*gr.dY(:,1:end-1))+ ...
         bsxfun(@rdivide,log(gr.xm( 2:end  )./gr.xGr(2:end-1)), ...
                         2*pi*Tx(:,2:end  ).*gr.dY(:,2:end  ));
RY = 0.5*bsxfun(@rdivide,gr.dY./Ty,
                         pi*(gr.xGr(2:end).^2 - gr.xGr(1:end-1).^2));
Cx = 1./RX;
Cy = 1./(RY(1:end-1,:)+RY(2:end,:));
```

To add this to our model without destroying what we already have, we place these lines together with the lines for the linear model in a sub-function where a switch chooses which one to use.

```
function [Cx,Cy] = conductances(gr,Tx,Ty,c)
    %CONDUCTANCE --- compute conductances
    % USAGE: [Cx,Cy] = contuctance(gr,Tx,Ty)

    % resistances and conducctances
    if ~gr.AXIAL
        RX = 0.5*bsxfun(@rdivide,gr.dx,gr.dy)./Tx;
        RY = 0.5*bsxfun(@rdivide,gr.dy,gr.dx)./Ty;
        Cx = 1./(RX(:,1:end-1)+RX(:,2:end));
    else
        RX = bsxfun(@rdivide,log(gr.xGr(2:end-1)./gr.xm( 1:end-1)), ...
                   2*pi*Tx(:,1:end-1).*gr.dY(:,1:end-1))+ ...
             bsxfun(@rdivide,log(gr.xm( 2:end  )./gr.xGr(2:end-1)), ...
                  2*pi*Tx(:,2:end  ).*gr.dY(:,2:end  ));
        RY = 0.5*bsxfun(@rdivide,gr.dY./Ty, pi*(gr.xGr(2:end).^2 - gr.xGr(1:end-1).^2));
        Cx = 1./RX;
    end
    Cy = 1./(RY(1:end-1,:)+RY(2:end,:));
```

and replace the line in which we computed the resistances and the conductances in the *fdm2* function by. See listing of *fdm2t* further down (transient version how this was done).

```
[Cx,Cy] = conductances(gr,Tx,Ty);
```

The switch 'AXIAL' is invoked by telling the grid object when it is created

```
[Phi,Q,Qx]=fdm2(gr,kx,ky,IBOUND,IH,FQ,'AXIAL',true)
```

4.2 Exercises axially symmetric model

4.2.1 Example changing the flat model for the building pit to an axially symmetric one

The result for axially s for the detail is shown below

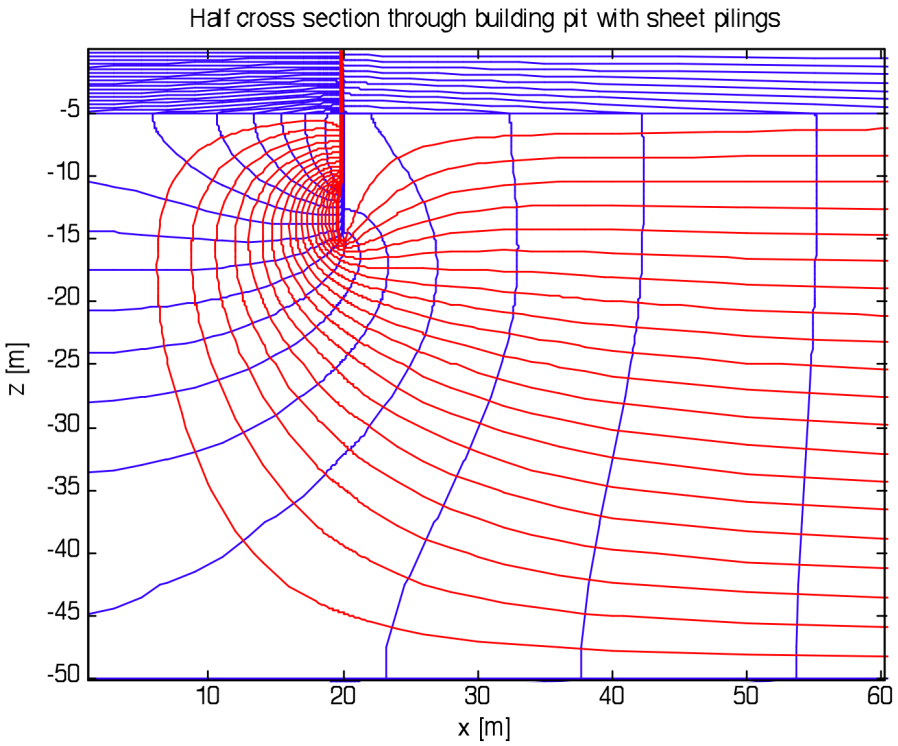


Figure 4.1: Same as above but for radial symmetric flow

The extraction is now 5250 m^3/d (not m^2/d !) with the fixed head in the wells -6.7 m to reach a drawdown of 5 m in the center of the pit.

Extractions in a cross section look like wells, because the streamlines to the extraction all connect with the top of the model. These are so-called branch cuts and are unavoidable, as the stream function is mulch-valued in the case of extractions or injections within the domain. This is, in fact, is nice for cross sections.

To make wells sharp, narrow the exaction column such that you only see a line of a column of the width of the borehole of the well.

4.2.2 Exercise: Show that the model is correct by comparing with analytical solutions like that of DeGlee

Show that the model is correct using analytical solutions. Plot the head and the flow contours (stream function)

The previous example with the cross section and its stream lines can immediately be computed in axially symmetric mode using the updated model. The only thing to change is adding 'AXIAL' to the call of the grid2DObj

```
gr = grid2DObj(xGr,yGr,'AXIAL',true)
```

The argument true may even be left out. Leaving 'AXIAL' out defaults to false. It is also possible to call with 'AXIAL',false.

4.2.3 Exercise: Compare model with confined well (Thiem)

Thiem is confined radial symmetric flow with fixed-head boundary at distance R. The analytical solution for the drawdown s is

$$s = \frac{Q}{2\pi kD} \ln\left(\frac{R}{r}\right)$$

4.2.4 Exercise: Compare model with semi-confined well (De Glee)

De Glee's radial symmetric stead-state flow to a fully penetrating well in a semi-confined aquifer has the analytical solution

$$\phi - h = \frac{Q}{2\pi kD} \mathrm{K}_0\left(\frac{r}{\lambda}\right); \quad \lambda = \sqrt{kDc}$$

with $\mathrm{K}_0(\ldots)$ well-known Bessel function of second kind and zero order.

We may compute this flow with the model in radial mode and compare with the analytical solution

In the semi-confined aquifer, the top is an aquitard with a fixed head above it. In the Matlab model, we may use the aquitard as the top layer. But then the fixed head is in the center of this layer. The resistance of the aquitard must than be generated by the half thickness of the top layer (between the node and the bottom of the cell). If the resistance of the aquitard is c, and the thickness of the top layer is H, then the vertical conductivity in this top layer must be set to .

We may also use an extra layer on top of the aquifer, make it very thin and specify the head in this thin top layer. In that case the conductivity of the top layer must be set to .

This is the only thing necessary to model a semi-confined aquifer with the radial symmetric model.

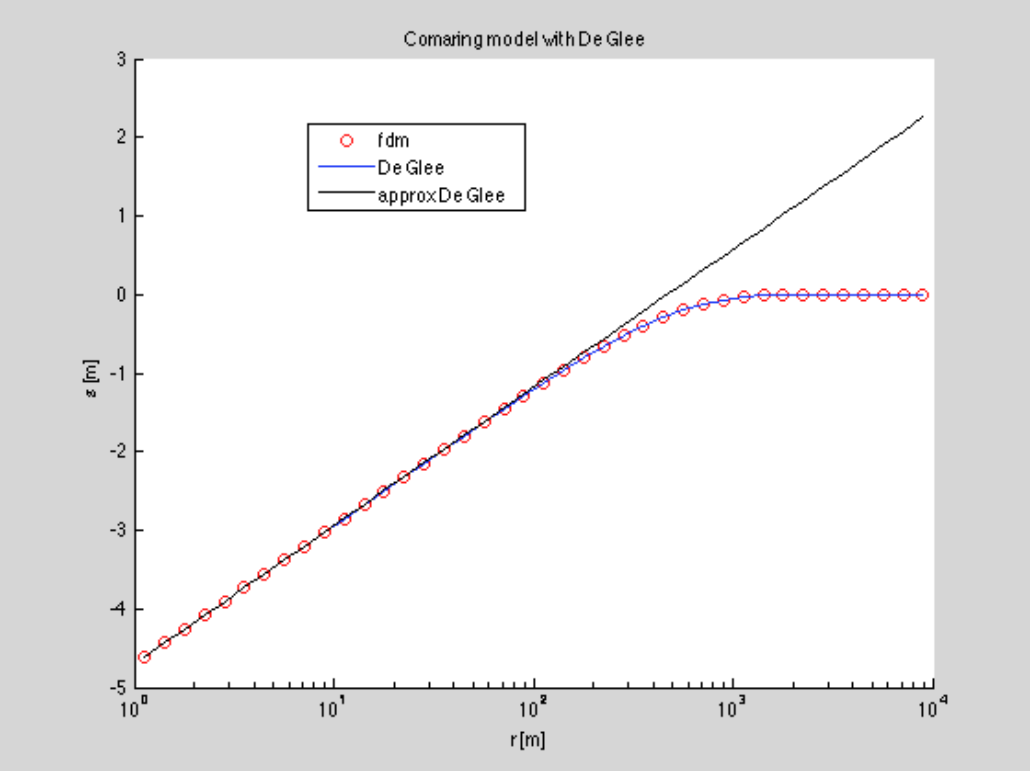


Figure 4.2: Comparing the model in radial mode with the semi-confined flow solution for steady state extraction from a well (De Glee).

4.2.4.1 Listing of script fdm_vs_DeGlee

```
%% Example comparing the the axially symmetric model with De Glee solution
close all

Qo    = -2400;
kD    =  500;
c     =  350;
lambda= sqrt(kD*c);

xGr = logspace(0,4,41);
```

```
yGr = [0,-1,-2];
gr  = grid2DObj(xGr,yGr,'AXIAL',true);

k       = gr.const([ 0.5*gr.dy(1)/c;   kD/gr.dy(2) ]);

IBOUND = gr.const(1); IBOUND(1,:)= -1;
IH     = gr.const(0);
FQ     = gr.const(0); FQ(end,1)=Qo;

[Phi,Q,Qx,Qy,Psi]=fdm2a(gr,k,k,IBOUND,IH,FQ);

fi1 = Qo/(2*pi*kD) * besselk(0,gr.xm/lambda);
fi2 = Qo/(2*pi*kD) * log(1.123*lambda./gr.xm);

figure; hold on; title('Comparing model with De Glee');
xlabel('r [m]'); ylabel('s [m]');
set(gca,'xScale','log');

plot(gr.xm,Phi(end,:),'ro');
plot(gr.xm,fi1       ,'b');
plot(gr.xm,fi2       ,'k');

legend('fdm','De Glee','approx De Glee');
```

4.2.5 Exercise: Compare vertical anisotropy

This should be straightforward. One can set up a model consisting of a number of layer and just use a vertical conductivity different from the horizontal one. It is also possible to generate complete k-fields and simulate strongly heterogeneous conditions.

4.2.6 Exercise: Compare with a circular island with recharge

The analytical solution has already been given using the model in the x-y plane. Here we should make an axially symmetric model, i.e. a cross section. One should be aware that our simple model is not aware of what part of the FQ is recharge and what part is from for instance direct injection. It is therefore that In the radial symmetric model the recharge in the top of the columns of the cross section has to be computed by the user as actual flow into each ring around the well, like this.

```
FQ(1,:)=pi*(r(2:end).^2-r(1:end-1).^2)*N;
```

with N the recharge

4.2.7 Exercise: Compute pumping test in layer 4 or 5 layer model

This is trivial with the model. A model with any number of layers is readily made by using model layers as layers, where the difference between aquifers and aquitards is merely the difference in their conductivities. The point is then that the model also calculates the heads in the resistant layers, which is not done in analytical formulas and where normally no piezometers can be satisfactorily set and used. If the heads in the resistant layers are not desired, one can simply choose to not plot them.

It is simple to implement separate resistant layers with no thickness and a given vertical resistance, but this does not add much value in a 2D model.

Having resistant layers may be convenient in a 3D model, for instance where many of them are only present over part of the region to be modeled and pinch out on many locations. Then the resistance, c, may be defined (in a 3D model) using a 3D array the size of which is $Ny \times N_x \times (N_z - 1)$ which thus fits between the regular model layers, i.e between the horizontal cell planes. The values in this

array which may be zero anywhere there is no resistance. Then the resistance is added to the vertical resistance of the overlaying and underlying layers within the FDM model.

4.2.8 Exercise: Compute effect of partial penetration

As stated before partial screen penetration of the aquifer is the rule rather than the exception when installing wells in the real world. Partially penetrating wells cause curvature of their stream lines near the well and an extra drawdown. Partially penetrating wells can be readily simulated with the axially symmetric model, simply by either extracting a given amount of water from a set of vertically spaced cells or by specifying a drawdown, to let the model compute the flow form such a well. There will be some difference between the the results of these different boundary conditions as should be the subject of this exercise to find out.

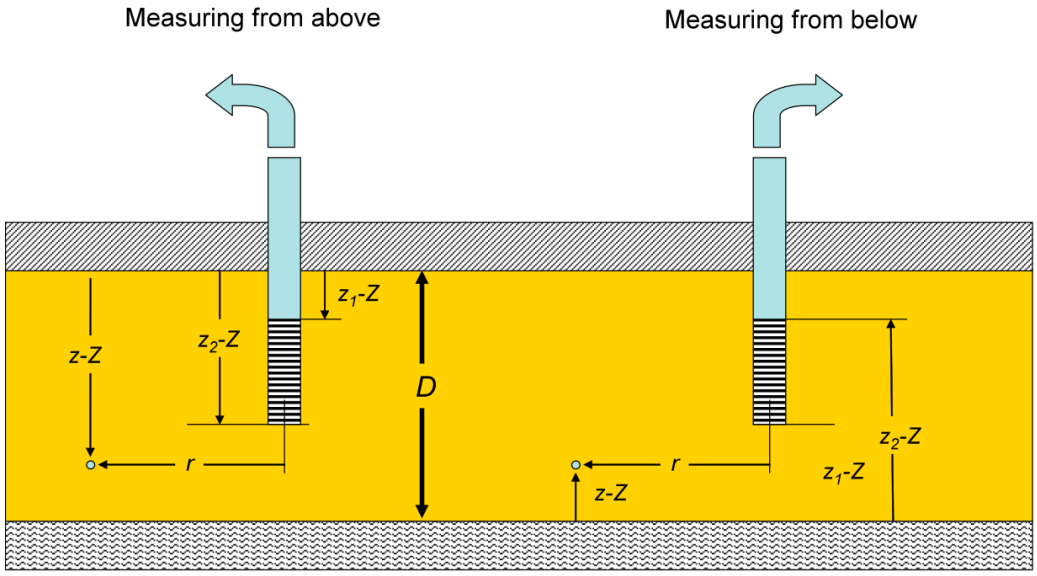


Figure 4.3: Partial penetrating wells with variables used in the formula

In Matlab one may compute both situations and subtract the two to get the extra drawdown and then compare it with the analytical formula.

The extra drawdown caused by partial penetration has been derived in the past and is given in several books on hydro-geology or pumping test analysis (e.g. Kruzeman & De Ridder, 1997):

$$\Delta s = \frac{Q}{2\pi kD}\frac{2D}{\pi d} \times \sum_{n=1}^{\infty}\left\{\frac{1}{n}\left[\sin\left(\frac{n\pi\,(z_1-Z)}{D}\right)-\sin\left(\frac{n\pi\,(z_2-Z)}{D}\right)\right]\cos\left(\frac{n\pi\,(z-Z)}{D}\right)\mathrm{K}_0\left(\frac{n\pi r}{D}\right)\right\}$$

Notice that Z is the reference elevation (i.e. the bottom or the top of the aquifer). This drawdown Δs is relative (has to be added to) the drawdown $s = \phi - \phi_0$ for fully penetrating wells. It was derived for a uniform extraction along the well screens in a homogeneous aquifer. (Notice also the sign of the Δs. The largest drawdown is in the center of the screen and the lowest drawdown at the top and the bottom of the aquifer)

.

This formula is valid for uniform extraction along the screen. This is readily implemented as the boundary condition for the well. In the real case, the boundary is rather a fixed head along the screen. This too is readily modeled with the Matlab model. The drawdown along the screen will than vary.

To check the model with respect to partial penetration, compute the drawdown with a fully and with partially penetrating well. Subtract the two drawdown matrices. This difference, which is also a matrix of the size of the model, can be compared with the analytical solution.

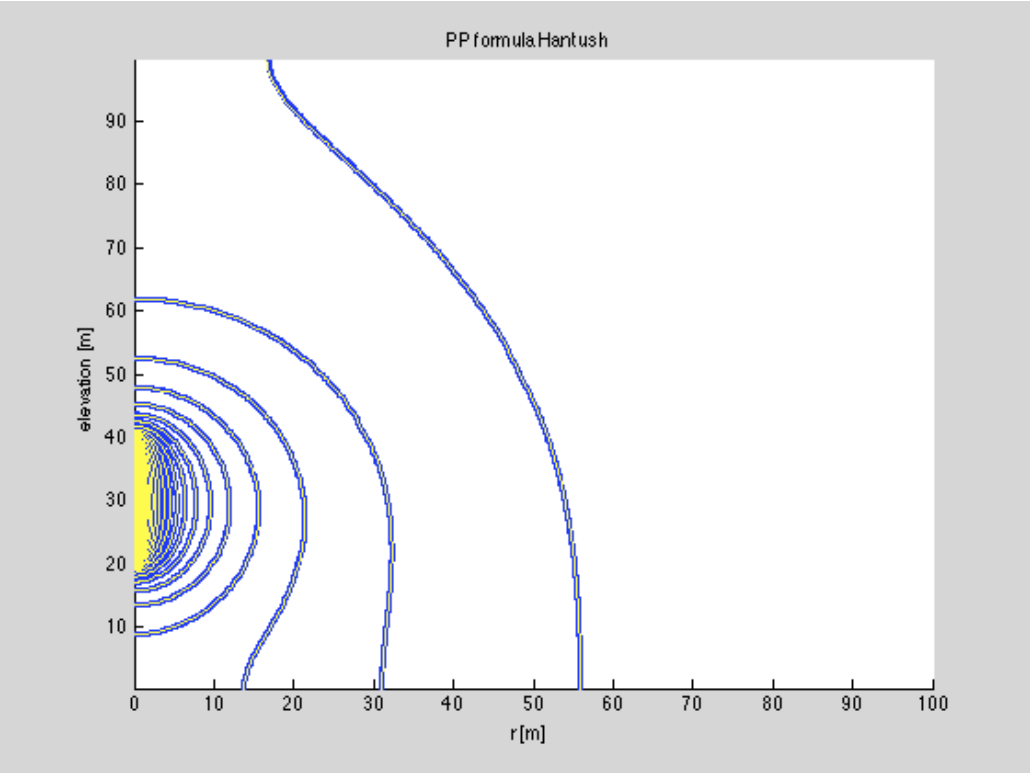


Figure 4.4: Result of partial penetration modeling.Thick blue lines are anlaytical soltuion PP+drawdown fully penetrating well, the thin yellow contours drawn over the blue lines are the contours computed with the numerical model. Both sets of lines virtually perfectly coincide.

```
% Show partial penetration analytically

ds=0; Z=0; D=sum(dy(2:end));  % ds is partial penetration

% 50 terms is more than sufficient
% analytical solution partial penetration
for i=1:50
   ds=ds+1/i*(sin(i*pi*(ZS(1)-Z)/D)-sin(i*pi*(ZS(2)-Z)/D))...
       .*cos(i*pi.*(ym*ones(size(xm))-Z)/D)...
       .*besselk(0,i*pi.*ones(size(dy))*xm/D);
end

ds=ds*Qo/(2*pi*kD)*(2*D/(pi*(ZS(1)-ZS(2))));

figure; hold on; xlabel('x [m]'); ylabel('y [m]');
title('dspp penetration contours\');
contour(xm,ym,ds)

set(gca,'xscale','log')
```

5 Transient modeling

5.1 Theory

With the previously developed models, all ingredients are already in place. The transient water balance during a given time step of length reads

$$\sum_{j=1,j\neq i}^{N} C_{ij}\phi_j + \left(C_{ii} + \hat{C}_i\right)\phi_i = Q_i + \hat{C}_i h_i - \hat{C}_{S,i}\frac{\phi^+ - \phi^-}{\Delta t}$$

This is the same as before. This equation represents a water balance of cell i in the model model, with general head boundary conditions implemented using the head h and their conductance $\hat{C}$. But we have now add storage as an additional term. The left part is the flow outward across the faces of cell i; the right-hand side expresses where this water comes from: injection into the node, a fixed head boundary outside the model and a release of storage over the considered time step. ϕ^+ is the head in the cell at the end of the time step and ϕ^- is the head in the cell at the beginning of the time step. This head represents the initial condition of the time step, necessary in all transient modeling. It is either the initial head at the start of the model or the head at the end of the previous time step. In any case, it is always known during the simulation.

All other heads and the flows have to be average values for the duration of the current time step and are still unknown.

$\hat{C}_s$ contains the storativity and the cell dimensions, and will be considered further down. The computation of this coefficient is specific to the numerical method, in our case finite differences.

Here we encounter two unknowns, ϕ_i and ϕ_i^+. We will only be able to resolve this situation if we assume some relation between them. For instance that the head change during the time step varies linearly from ϕ^- to ϕ^+ and that the average heads during the time step, ϕ are those at a time that is given by some value $t = t^- + \theta\Delta t$, where $0 \leq \theta \leq 1$.

With this in mind, we have

$$\phi - \phi^- = \theta\left(\phi^+ - \phi^-\right) \rightarrow \phi^+ - \phi^- = \frac{\phi - \phi^-}{\theta}$$

and therefore,

$$\sum_{j=1,j\neq i}^{N} C_{ij}\phi_j + + \left(C_{ii} + \hat{C}_i\right)\phi_i = Q_i + \hat{C}_i h_i - \frac{\hat{C}_{S,i}}{\theta\Delta t}\left(\phi - \phi^-\right)$$

Exactly like we did with the general head boundaries, we leave the fixed part at the right-hand side and put the variable part to the left hand side, yielding

$$\sum_{j=1,j\neq i}^{N} C_{ij}\phi_j + \left(C_{ii} + \hat{C}_i + \frac{\hat{C}_{S,i}}{\theta\Delta t}\right)\phi_i = Q_i + \hat{C}_i h_i + \frac{\hat{C}_{S,i}}{\theta\Delta t}\phi^-$$

The left-hand side is equivalent to adding $\hat{C}_i$ and $\hat{C}_{S,i}/\left(\theta\Delta t\right)$ to the diagonal matrix coefficient. The right-hand side is equivalent to a permanent inflow into the node during this time step. In Matlab/matrix formulation

$$\left(C + diag\left(\mathbf{C}_i + \hat{\mathbf{C}} + \frac{\hat{\mathbf{C}}_s}{\theta\Delta t}\right)\right)\cdot\Phi = \mathbf{Q} + \hat{\mathbf{C}}\cdot\mathbf{h} + \frac{\hat{\mathbf{C}}_s}{\theta\Delta t}\cdot\Phi^-$$

in which, consistent with the previous equations, $\mathbf{C}_i$ is the diagonal taken out of the system matrix and C is the system matrix without its diagonal. This is also the way we compute the model in Matlab. For, the values at the matrix diagonal are subject to change due to the boundary conditions and also due to transient simulations, where it depends on the time step Δt. Therefore, it is convenient and efficient to insert the full diagonal into the system matrix only just before the set of equations has to be solved.

This represents the complete transient model, including its initial and boundary conditions.

Hence, to solve this model in Matlab for the time step:

$$\Phi = \left(\mathbf{C} + diag\left(\mathbf{C}_i + \hat{\mathbf{C}} + \frac{\hat{\mathbf{C}}_S}{\theta \Delta t} \right) \right) \setminus \left(\mathbf{Q} + \hat{\mathbf{C}} \cdot \mathbf{h} + \frac{\hat{\mathbf{C}}_s}{\theta \Delta t} \cdot \Phi^- \right)$$

This yields average heads during the time step (based on the chosen value of θ). The head at the end of the time step requires a separate computation step:

$$\begin{aligned} \Phi^+ &= \Phi^- + \frac{1}{\theta}\left(\Phi - \Phi^-\right) \\ &= \frac{1}{\theta}\Phi + \left(1 + \frac{1}{\theta}\right)\Phi^- \end{aligned}$$

Then, by setting $\Phi^- = \Phi^+$ we enter into the next time step, with the heads at the end of the previous time step are the initial heads of the next one.

The value of θ is called the implicitness of the solution. A solution for $\theta = 0$ is called explicit and one with $\theta = 1$ is called fully implicit. Values of $\theta > 0.5$ yield stable solutions (without artificial oscillations). $\theta = 0$ requires small time steps in order to prevent oscillation. On the other hand computation steps are cheap because it does not require any solution of a system matrix. A value of $\theta = 1$ is called fully implicit. It may be less accurate in case of larger time steps, but it is rock-stable. Notice that MODFLOW just uses $\theta = 1$ without any choice for the user. An optimal value for finite element models seems $\theta = 2/3$. Anyway, all values above $\theta = 0.5$ yield unconditionally stable solutions. In practice, it may be most simple to use $\theta = 1$, which implies that the average flows and heads during the time step are well represented by those at the end of the time step. Given the success of MODFLOW there seems to be no real objection against $\theta = 1$. Notice that $\theta = 1$ makes the second step to update the heads at the end of the time step obsolete because it reduces to

$$\Phi^+ = \Phi$$

We only have to elaborate the values of $\hat{C}_S$. For the flat finite difference model these equal

$$\hat{C}_S = S_S \Delta x \Delta y \Delta z$$

Where S_y is specific yield (water table storage) and S_S is the specific elastic storage coefficient. The latter requires the thickness of the model cell to be given.

As can be seen, each cell is given both a specific yield (in case it has or gets a free water table) and an elastic storage for the saturated part. In our simple models we will not deal with variable aquifer or layer thickness during the simulation, although this is quite straightforward to implement.

For the radial symmetric model the storage coefficients equal

$$\hat{C}_S = \pi \left(r_{i+1}^2 - r_i^2 \right) \Delta z \, S_S$$

Where r_{i+1} and r_i are the radii of the cell.

In practice, S_y will be specified for the top cells with a free water table and S_s for all deeper cells.

What has to be changed to the model to make it transient?

The function call has to be extended with time, storage coefficients and initial heads, while θ may be specified or just set to a default value. We just keep θ as an internal parameter of the model. Here is the transient model.

5.1.1 Implementation fdm2t

```
function [Phi,Q]=fdm2t(gr,t,Tx,Ty,Ss,IBOUND,HI,QI)
%FDM2 a 2D block-centred transient finite difference model
% USAGE:
%    [Phi,Q]=fdm2(gr,t,Tx,Ty,Ss,IBOUND,HI,FQ)
% Inputs:
%    gr    = grid2DObj (see grid2DObj)
%    Tx,Ty = transmissivities, either scalar or full 2D arrays
%    IBOUND= boudary array as in MODFLOW (<0 fixed head, 0 inactive, >0 active)
%    HI    = initial heads (STRTHD in MODFLOW)
%    QI    = prescribed inflow for each cell.
%    t     = times, dt = diff(t) will be the time steps
% Outputs:
%    Phi(Ny  ,Nx  ,Nt ) [ L  ]  Nt=numel(diff(t)); Ndt=numel(dt);
%    Qin(Ny  ,Nx  ,Ndt) [L3/T]  flow into cells during time step
%    Qx( Ny  ,Nx-1,Ndt) [L3/T]  horizontal cell face flow positive in positive xGr direction
%    Qy( Ny-1,Nx  ,Ndt) [L3/T]  vertial    cell face flow, postive in positive yGr direction
%    Psi(Ny+1,Nx-2,Ndt) [L3/T]  stream function (only useful if flow is divergence free)
%    Qs( Ny  ,Nx  ,Ndt) [L3/T]  flow released from storage during the time step Dphi*S*V/Dt
% TO 991017  TO 000530 001026 070414 090314 101130 140410

theta = 0.67; % degree of implicitness

t     = permute(unique(t(:)),[3,2,1]); dt = diff(t,1,3); Ndt=numel(dt);
Nodes = reshape(1:gr.Nod,gr.size);               % Node numbering
IE=Nodes(:,2:end);   IW=Nodes(:,1:end-1);
IS=Nodes(2:end,:);   IN=Nodes(1:end-1,:);

Iact  = IBOUND(:) >0;
Inact = IBOUND(:)==0; Tx(Inact)=0; Ty(Inact)=0;
Ifh   = IBOUND(:) <0;

[Cx,Cy,Cs] = conductances(gr,Tx,Ty,Ss,theta);

C     = sparse([IW(:); IN(:)], [IE(:); IS(:)], [-Cx(:); -Cy(:)], gr.Nod, gr.Nod, 5*gr.Nod);
C     = C + C';     % add lower half of matrix
diagC = -sum(C,2);  % Main diagonal

HI(Inact)=NaN;   Phi = bsxfun(@times, t,HI);   HI = HI(:);   HT=HI;

QI(Inact)=  0;   QI = QI(:); % remains fixed
for idt = 1:Ndt
    HT(Iact) = spdiags(diagC(Iact)+Cs(Iact)/dt(idt),0,C( Iact , Iact )) ...
               \(QI(Iact) - C(Iact,Ifh)*HI(Ifh) + Cs(Iact)/dt(idt).*HI(Iact)); % solve

    Q(idt).NET= gr.const(0);
    Q(idt).STO= gr.const(0);
    Q(idt).FH = gr.const(0);

    % Water budget items for every cell
    % Q_NET is net inflow in cell to surrounding cells. In FH cells this
    % equals the flow Q_FH from fixed heads. See line 4 below Q_FH(Ifh)=Q_NET(Ifh).
    % in fact: Q_NET = Q_FH + Q_FQ = Q_STO
```

```
    Q(idt).NET(~Inact)  = reshape(spdiags(diagC(~Inact),0,C(~Inact,~Inact))* HT(~Inact),gr.size)
    Q(idt).STO(Iact)    = Cs(Iact)/dt(idt).*(HT(Iact)-HI(Iact));
    Q(idt).FH(Ifh)      = Q(idt).NET(Ifh);
    Q(idt).FQ           = QI;
    Q(idt).X            = -diff(reshape(HT,gr.size),1,2).*Cx;
    Q(idt).Y            =  diff(reshape(HT,gr.size),1,1).*Cy;

    % Stream function in XZ plane only useful in vertical cross sections
    Q(idt).Psi          = zeros(gr.Ny+1,gr.Nx-1);
    Q(idt).Psi(2:end,:) = cumsum(Q(idt).X,1);
    Q(idt).Psi(:,:)     = Q(idt).Psi(end:-1:1,:);

    % Update HI for next loop
    HI(Iact) = HT(Iact)/theta - (1-theta)/theta * HI(Iact);

    Phi(:,:,idt+1) = reshape(HI,gr.size);

end

function [Cx,Cy,Cs] = conductances(gr,Tx,Ty,Ss,theta)
    %CONDUCTANCE --- compute conductances
    % USAGE: [Cx,Cy] = contuctance(gr,Tx,Ty)

    if isscalar(Tx), Tx= gr.const(Tx); end
    if isscalar(Ty), Ty= gr.const(Ty); end
    if isscalar(Ss), Ss= gr.const(Ss); end

    % resistances and conducctances
    if ~gr.AXIAL
        RX = 0.5*bsxfun(@rdivide,gr.dx,gr.dy)./Tx;
        RY = 0.5*bsxfun(@rdivide,gr.dy,gr.dx)./Ty;
        Cx = 1./(RX(:,1:end-1)+RX(:,2:end));
        Cs = gr.Vol.*Ss/theta; Cs = Cs(:);
    else
        RX = bsxfun(@rdivide,log(gr.xGr(2:end-1)./gr.xm( 1:end-1)),2*pi*Tx(:,1:end-1).*gr.dY(:,1
             bsxfun(@rdivide,log(gr.xm( 2:end  )./gr.xGr(2:end-1)),2*pi*Tx(:,2:end  ).*gr.dY(:,2
        RY = 0.5*bsxfun(@rdivide,gr.dY./Ty, pi*(gr.xGr(2:end).^2 - gr.xGr(1:end-1).^2));
        Cs = gr.Vol.*Ss/theta; Cs = Cs(:);
        Cx = 1./RX;
    end
    Cy = 1./(RY(1:end-1,:)+RY(2:end,:));
```

The water balance can be checked using the flow outputs produced by the model. The computed $Q_{in,t}$ equals the net outflow across the cell faces during time step t

$$Q_{in,i,t} = \sum_{j=1,\, j\neq i}^{N} C_{ij}\left(\overline{\phi}_{i,t} - \overline{\phi}_{j,t}\right)$$

where $\overline{\phi}_t$ is the average head in the indexed cell during time step t.
In matrix form

$$Q_{in,t} = \mathbf{C}\Phi_t$$

where C includes the diagonal. Each row of Q_{in} is the net inflow in a cells from the outside world and, so it equals

The flow must equal the net inflow of the cell during the same time step + the release rate from storage, also as a flow (dropping the indices for convenience):

$$Q_{in,t} = Q_{FQ,t} + Q_{FH,t} - Q_{S,t}$$

Q_S the flow released from storage is a separate output of the model. It must equal

$$\begin{aligned} Q_{S,t} &= -\frac{C_S}{\theta \Delta t} \left(\phi^+ - \phi^-\right)_t \\ &= -\frac{\Delta V}{\Delta t} \left(\phi_{t+1} + \phi_t\right) \end{aligned}$$

where the prim + and - indicate heads at time $t + \theta \Delta t$ and t and ϕ_{t+1} and ϕ_t the heads at the end and start of the considered time step.

The water balance is checked in the script below.

The water budget terms are all packed in the budget struct, which is the second output of the function fdm2t. See the script *scriptTransientTheis* how the budget struct is used.

5.2 Exercises transient model

Prove that the model is correct

5.2.1 Exercise: Check the water balance

To check the water balance, the storage must be included. Check for your self which flows must add up to zero.

5.2.2 Exercise: Compare the model with Theis's solution

5.2.2.1 Theory

Theis's solution is for a fully penetrating well in a confined aquifer. The well-known solution for the drawdown is

$$s = \frac{Q}{4\pi kD} \mathrm{E}_0 \left(\frac{r^2 S}{4kDt} \right)$$

With $\mathrm{E}_o\left(\ldots\right)$ the exponential integral or Theis' well function. In Matlab, for time, $t(i)$,

```
s=Qo/(2*pi*kD).*expint(r.^2.*S./(4*kD*t(i)));
```

Here is a matlab script that will do the job.

5.2.2.2 Results

The actual script fixes the head at $r = 100\,\mathrm{m}$ just to show how the water budget works out. Hence the drawdowns become steady after some period of time, when the fixed-head boundary becomes active.

The picture shows the accuracy of the model. The model seems to be off for very small times only. This is because the model starts with zero initial heads at a very small but still non-zero time, while the analytical solution is only zero at zero time. This difference can be completely removed by using the analytical solution at the initial time as initial heads.

The top picture show that the analytical Theis drawdown continues to grow after about 10 days, whereas the drawdown in the model is then steady due to the fixed-head boundary at $r = 100\,\mathrm{m}$. Vice

versa, the drawdown as function of distance to the well becomes fixed after a certain time while that in the analytical solution continuous to draw, as evicenced by the black lines in the lower figure.

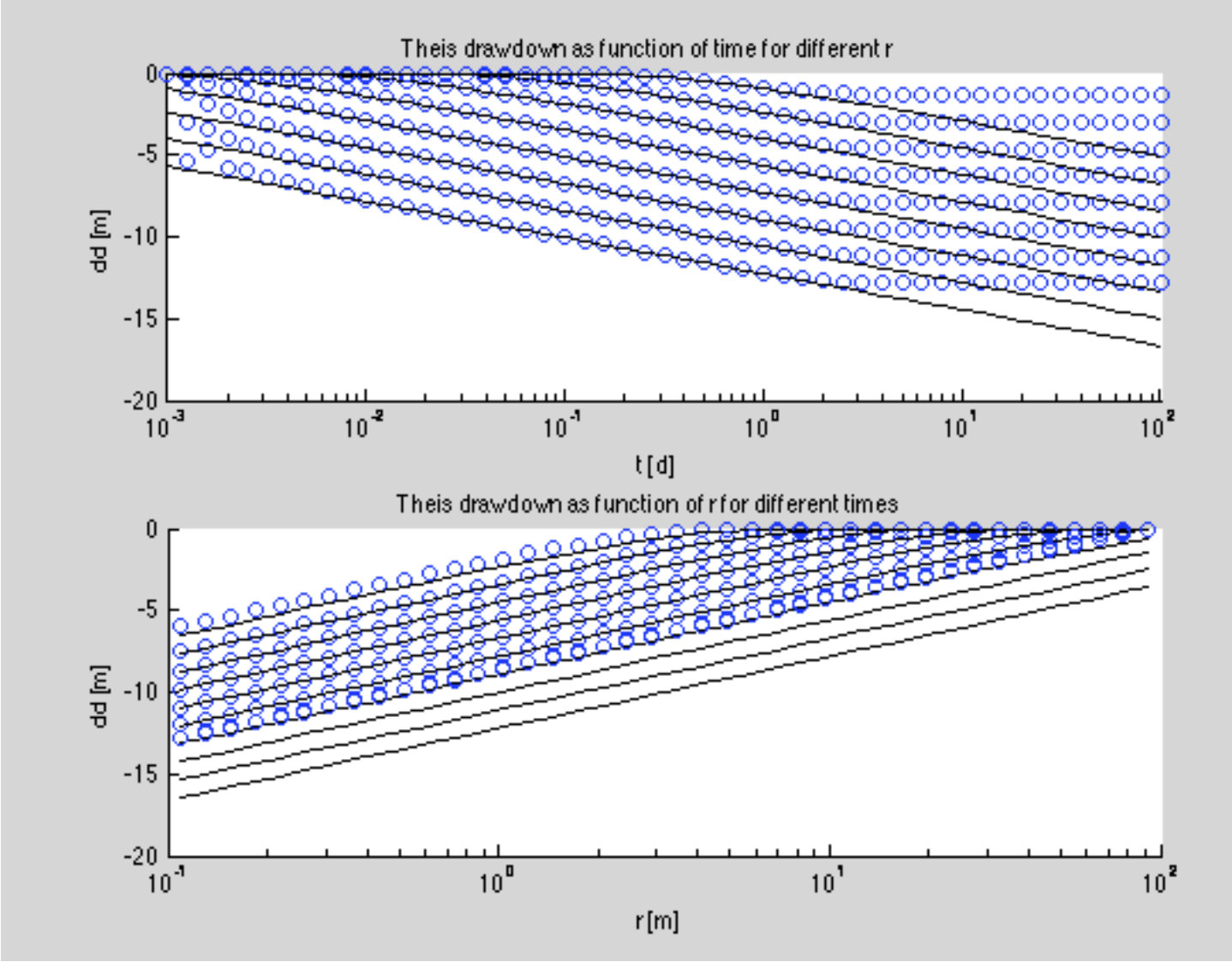


Figure 5.1: Drawdown as function of time for different distances from the well and as function of distance but for different times. The markers are the model and the lines are the analytical solution according to Theis

5.2.2.3 Listing of scriptTransientTheis

The lower part of the script shows the water budget of the entire model with time.

```
%% modelScript -- transient model for Theis well case
% Transient axially symmetric model simulating transient extraction
% from a confined aquifer according to Theis. Comparison with analytic
% solution.

close all;

%% Data for theis well and aquifer
Qw =-2400;
rW = 0.1;
kD = 200; D=20; k = kD/D;
```

```
S  = 0.1;       Ss = S/D;

t=logspace(-3,2,51);

%% grid
rGr = logspace(log10(rW),2,41);
yGr = [0  -20];
gr = grid2DObj(rGr,yGr,'AXIAL');

IBOUND = gr.const(1);
IBOUND(:,end)=-1;

HI = gr.const(0);
FQ = gr.const(0);  FQ(1,1) = Qw;

TheisR = @(t) Qw/(4*pi*kD) * expint(gr.rm.^2*S./(4*kD*t));
TheisT = @(r) Qw/(4*pi*kD) * expint(   r .^2*S./(4*kD*t));

[Phi,B]=fdm2t(gr,t,k,k,Ss,IBOUND,HI,FQ);

%% Check storage
for idt = numel(diff(t)):-1:1
    QIN(idt,1) = sum(B(idt).NET(:));
    QFH(idt,1) = sum(B(idt).FH(:));
    QFQ(idt,1) = sum(B(idt).FQ(:));
    QST(idt,1) = sum(B(idt).STO(:));
end

format shortg
fprintf('%12s %12s %12s %12s %12s\n','Qin','QFH','-FQ','QST','QFH+FQ-QST');
display([QIN -QFH -QFQ QST QFH+QFQ-QST]);

%% Drawdown as function of distance
close all
figure;

subplot(2,1,1,'nextplot','add','xScale','log');
title( 'Theis drawdown as function of time for different r');
xlabel('t [d]'); ylabel('dd [m]');

for ir=1:5:gr.Nr
   plot(t,squeeze(Phi(1,ir,:)),'bo'); % numerical
   plot(t,TheisT(gr.rm(ir)),'k'); % analytical
end

subplot(2,1,2,'nextplot','add','xScale','log');
title( 'Theis drawdown as function of r for different times');
xlabel('r [m]'); ylabel('dd [m]');

for it=5:5:length(t)
   plot(gr.rm,Phi (1,:,it)  ,'bo'); % numerical
   plot(gr.rm,TheisR(t(it)), 'k'); % analytical
end
```

5.2.3 Exercise: Compare the model with Hantush's solution

Hantush's solution concerns the drawdown due to a well in a semi-confined aquifer:

$$W_h\left(u,\,\frac{r}{\lambda}\right)=\int_y^\infty \frac{1}{y}\exp\left(-y-\frac{1}{4y}\left(\frac{r}{\lambda}\right)^2\right)dy$$

It may be readily implemented by writing a function that carries out the integration:

$$W_h=\int_{\ln u}^{\ln\infty}\exp\left(-\exp\left(\ln y\right)-\exp\ln\left(\left(\frac{r}{2\lambda}\right)^2\right)\right)d\left(\ln y\right)$$

where $\ln\infty\approx 20$ in practice and any required accuracy is obtained by sufficient reduction of the integration step size $\Delta\left(\ln y\right)$.

The drawdown is

$$s=\frac{Q}{4\pi kD}W_h\left(u,\,\frac{r}{\lambda}\right),\qquad u=\frac{r^2S}{4kDt}$$

5.2.4 Exercise: Compute delayed yield

Delayed yield may result from the drawdown above the aquitard that is caused by the leaking through the aquitard. 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 expands fast. Slowly the water table will determine the drawdown and will show up at a later time. The combined drawdown curve shows two theis-curves in series, the first one determined by the elastic storage, the second one by the water table storage. In Matlab is it readily modeled by giving all cells a small elastic storage coefficient and only the top layer cells a larger one, the specific yield. Compute the time-drawdown curve and compare it with the two Theis curves

5.2.5 Exercise: Compute well bore-storage (Boulton)

5.2.5.1 Background

The storage inside the well changes the drawdown shortly after the start of the pump. It may be implemented by modeling the well casing explicitly. 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 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)

Consider a large 5 m radius dug well that is also 5 m deep in a 20 m deep aquifer. The specific yield is 5\% and the extraction 130 m^3/d, i.e. 1000 people using 70 l/d plus 600 cattle using 100 L/d. What will be the drawdown in this well? Is it sustainable?

To analyze this situation, make an radial symmetric model, 20 m deep. Use a logarithmically increasing grid size with distance (in Matlab you ma use *logspace*(-1,4,41), such that the drawdown will not reach the outer boundary of the model) and say 20 layers of 1 m thickness vertically. Then refine around the diameter of the well and around its bottom, to accurately compute the concentrated flow in this region.

In the well use a very high conductivity, day $k=10000$ m/d, so that the well will obtain a uniform head like in the reality. The extraction may be put in an arbitrary model point inside this well. Then apply the storage coefficient to the model cells. All cells may be given the specific elastic storage

Figure 5.2: A large diameter well

coefficient by default. However, specific yield is different. It applies to the topmost cells only and we must use it there as a kind of elastic storage for the top row of cells. To do this, use S_y/dz for this row as storage coefficient. That is, do this for all top row cells and use 1/dz (i.e. $S_y = 1$) for the cells representing the inside of the well.

5.2.5.2 Results

Figure 5.3shows the results for large diameter well as drawdown versus distance for a large number of times. Figure 5.4. Figure 5.4shows the drawdown as function or time in the well and at the bottom of the aquifer. Theis\' solution without well bore storage is also shown for comparison.

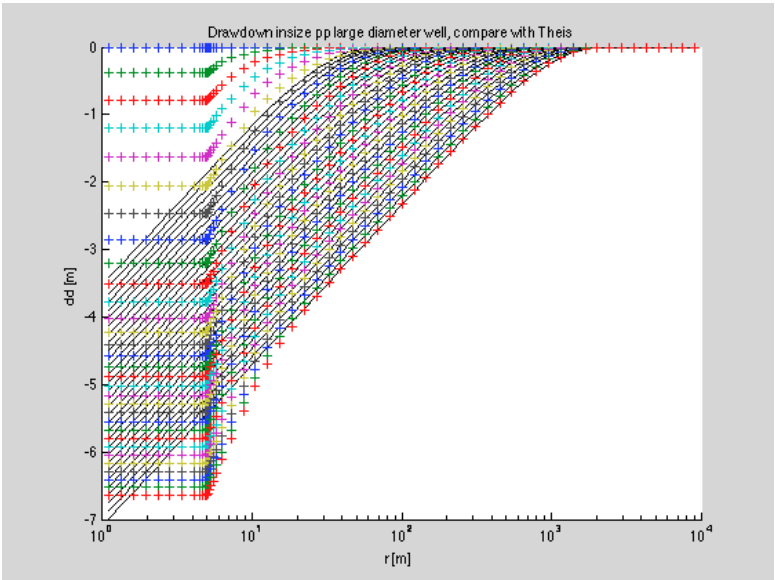


Figure 5.3: Drawdown (numeric +) along z=0 through well and at top of aquifer. Comparison with Theis solution (lines). The horizontal lines is the head inside the well (5 m radius)

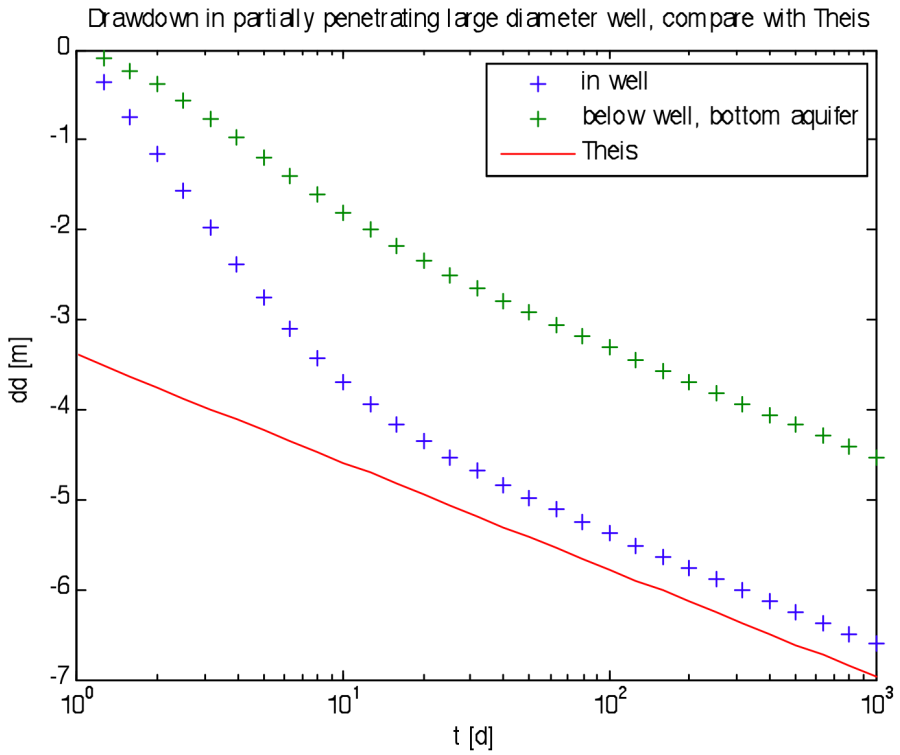


Figure 5.4: Drawdown inside large diameter well, below it at the bottom of the aquifer and comparison with Theis solution (this drawdown is quite substantial). The drawdown inside and below the well is less than Theis, because the large diameter compensates the partial penetration. The drawdown at the bottom of the aquifer is much less than inside the well due to partial penetration. Initially the drawdown in the well is less than Theis and declines more or less linearly due to the large storage inside it.

To check the water balance, see if the total extraction from the well over the entire period matches the water released from storage:

```
dt=diff(t);
St=zeros(length(dt));
for it=1:length(dt)
    St(it)=sum(sum(Qs(:,:,it)))*dt(it);
end
FromStorage=sum(St(:))
Injected   =Qw*sum(dt)
```

Matlab gives:

```
FromStorage = 1.2987e+005
Injected = -129870
```

These are indeed the same and equal the total extraction. Now check with the given well extraction ($-130\,\mathrm{m^3/d} \times 999\,\mathrm{days}$)

```
>> Qw*sum(dt)
ans = -129870
```

Which indeed matches the given infiltration (extraction = negative infiltration)
To visualize the flow, compute the steady-state model

```
FH(:,end)=0;   % now we must have some boundary fixed

[Phi,Qn,Qx,Psi]=fdm2(gr,K,K,IBOUND,FH,FQ); % steady state model

figure; axes('nextPlot','add','xscale','log');
title('large diameter well');
xlabel('r [m]'); ylabel('z [m]');

contour(rm,zm,Phi);          % head lines
contour(r(2:end-1),z,Psi);   % stream lines
```

Figure 5.5 shows head contours and streamlines in a cross section through the large-diameter well with well-bore storage. The streamlines inside the well-bore are due to the location of the fixed head, which are the actual cells that extract the water. Notice that the radial distance is on logarithmic scale.

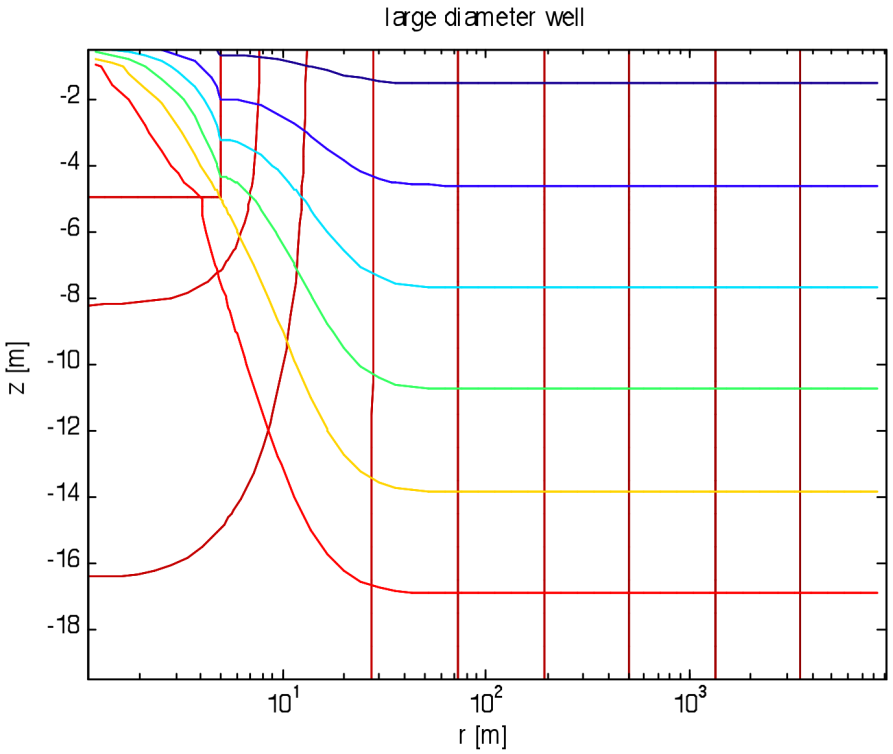


Figure 5.5: The contours of the steady-state computation with fixed head at R=10000. Inside the wells the stream lines continue to the point of extraction. Notice the logarithmic scale used to visualize the situation

5.2.5.3 Listing scriptLargeDiameterWell

```
% Large-diameter well drawdown with well storage (Bouton, 1963)

%% Specify well
rW  = 5;
zW  =-5;
k   = 1;
ss  = 0.000;
Sy  = 0.05;
Qw  =-130;
```

```
%% Grid
rGr = [logspace(0,4,50), rW+[-0.5 -0.25 -0.1 0 0.1 0.25 0.5 1]];
yGr = [0:-1:-20, zW+[-0.5 -0.25 -0.1 0 0.1 0.25 0.5]];
t   = logspace(0,3,31);

gr = grid2DObj(rGr,yGr,'AXIAL',true);

%% Define inWell domain
inWell = gr.Ym>zW & gr.Rm<rW;

%% Arrays
IBOUND = gr.const(1);
K  = gr.const(k); K(inWell)=10000;
Ss             = gr.const(ss);  % elastic storage
Ss(1,:)        = Sy/gr.dy(1);   % specific yield
Ss(1,gr.rm<rW)= 1/gr.dy(1);     % within well Ss=1 (free surface)

IH = gr.const(0);
FQ = gr.const(0); FQ(1,2)=Qw;

%% Run model
[Phi,Qt,Qr,Qz,Qs]=fdm2t(gr,t,K,K,Ss,IBOUND,IH,FQ);

%% Analytic solution Theis
kD = sum(K( :,end) .* gr.dy);
SY = sum(Ss(:,end) .* gr.dy);
fi = Qw/(4*pi*kD)*expint(bsxfun(@rdivide,gr.rm.^2*SY,4*kD*t(:)));

%% Visualize
close all
figure; axes('nextPlot','add','xScale','log');
title('Drawdown insize pp large diameter well, compare with Theis');
xlabel('r [m]'); ylabel('dd [m]');
plot(gr.rm(:),fi,'k');
plot(gr.rm,squeeze(Phi(1,:,:)),'+');
```

5.2.6 Exercise: Compute the effect of a shower of rain on a parcel of land compare with analytical solution

A shower of rain on a parcel of land cause the water to be raised instantaneously, but the head at the edges of the parcel remains equal to the ditch level. This comes down to an immediate drawdown at the edges of the parcel, which progresses into the parcel, initially fast becoming slower and slower over time.

To model this in Matlab, set the fixed head in the ditches equal to zero and use an initial head equal to n/Sy, where n is the shower in mm and Sy the specific yield. Then follow the drawdown over time. Used increasing time steps to track the fast initial drawdown well.

```
t=logspace(-3,3,61);
```

Compare the results with the analytical solution

$$s = \frac{n}{S_y}\left(1 - \frac{4}{\pi}\sum_{j=1}^{\infty}\left(\frac{(-1)^{j-1}}{2j-1}\cos\left((2j-1)\,\pi\frac{x}{L}\right)\exp\left(-(2j-1)^2\,\pi^2\frac{kD}{L^2 S}t\right)\right)\right)$$

Where L is the half-width of the cross section through the parcel ([Carslaw and Jaeger (1959)] p97, eq 8; Verruijt, 1999, p87).@@@

6 Particle tracking

6.1 Flow lines (following particles)

6.1.1 Background

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.

6.1.2 Theory

To show this, take the divergence for an arbitrary point within a 3D cell without sources and sinks. This divergence must be zero:

$$\frac{\partial q_x}{\partial x} + \frac{\partial q_y}{\partial y} + \frac{\partial q_z}{\partial z} = 0$$

Written out in the flows generated by the model for a cell with size $\Delta x \Delta y \Delta z$ gives

$$\frac{Q_{x2} - Q_{x1}}{\Delta x \Delta y \Delta z} + \frac{Q_{y2} - Q_{y1}}{\Delta x \Delta y \Delta z} + \frac{Q_{z2} - Q_{z1}}{\Delta x \Delta y \Delta z} = 0 \rightarrow Q_{x2} - Q_{x1} + Q_{y2} - Q_{y1} + Q_{z2} - Q_{z1} = 0$$

which must be true because it is the cell's water balance (without internal sources).

To analyze particle tracking within the realm of FDM further, consider the average velocity at cell faces, computed from the flows perpendicular to the cell faces (Q_x and Q_y) and the porosity of the cell. In 2D, the velocity in a cell with porosityϵ may be computed for the local flow in x and y direction. For generality we also have to consider the thickness of the cell, D, perpendicular to the x, y plane.

$$v_x = \frac{1}{\epsilon H}\frac{Q_x}{\Delta y}; \quad v_y = \frac{1}{\epsilon H}\frac{Q_y}{\Delta x}$$

Within a cell the flow is interpolated between that of the opposite cell faces

$$Q_x = Q_{x-} + (x - x_-)\frac{Q_{x+} - Q_{x-}}{x_+ - x_-}; \quad Q_y = Q_{y-} + (y - y_-)\frac{Q_{y+} - Q_{y-}}{y_+ - y_-}$$

$$
\begin{aligned}
v_x &= \frac{Q_x}{\epsilon H \Delta y} = \frac{Q_{x-}}{\epsilon H \Delta y} + (x - x_-)\frac{Q_{x+} - Q_{x-}}{\epsilon H \Delta x \Delta y} = \frac{\Delta x Q_{x-}}{\epsilon V} + (x - x_-)\frac{Q_{x+} - Q_{x-}}{\epsilon V} \\
v_y &= \frac{Q_y}{\epsilon H \Delta x} = \frac{Q}{\epsilon H \Delta x} + (y - y_-)\frac{Q_{y+} - Q_{y-}}{\epsilon H \Delta x \Delta y} = \frac{\Delta y Q_{y-}}{\epsilon V} + (y - y_-)\frac{Q_{y+} - Q_{y-}}{\epsilon V}
\end{aligned}
$$

In which the indices + and – refer to the sides of this cell.
Hence, $\Delta x = x_+ - x_-$ and $\Delta y = y_+ - y_-$.
By dividing by $\epsilon H \Delta y$ and $\epsilon H \Delta x$ within the cell we obtain the groundwater velocities

$$
\begin{aligned}
v_x &= \frac{Q_x}{\epsilon H \Delta y} = \frac{Q_{x-}}{\epsilon H \Delta y} + (x - x_-)\frac{Q_{x+} - Q_{x-}}{\epsilon H \Delta x \Delta y} \\
v_y &= \frac{Q_y}{\epsilon H \Delta x} = \frac{Q_{y-}}{\epsilon H \Delta x} + (y - y_-)\frac{Q_{y+} - Q_{y-}}{\epsilon H \Delta x \Delta y}
\end{aligned}
$$

Or,

$$
v_x = v_{x-} + \frac{v_{x+} - v_{x-}}{\Delta x}(x - x_-); \quad v_y = v_{y-} + \frac{v_{y+} - v_{y-}}{\Delta y}(y - y_-)
$$

With

$$
a_x = \frac{dv_x}{dx} = \frac{v_{x+} - v_{x-}}{\Delta x}; \quad a_y = \frac{dv_y}{dy} = \frac{v_{y+} - v_{y-}}{\Delta y}
$$

This simplifies to

$$
v_x = v_{x-} + a_x(x - x_-); \quad v_y = a_y(y - y_-)
$$

Working this out in terms of particle displacement yields

$$
\frac{dx}{dt} = v_{x-} + a_x(x - x_-); \quad \frac{dy}{dt} = v_{y-} + a_y(y - y_-)
$$

We leave out the y-direction for now to limit the length of this paper.
Integration yields

$$
\frac{d(v_{x-} + a_x(x - x_-))}{v_{x-} + a_x(x - x_-)} = a_x dt \rightarrow \ln(v_{x-} + a_x(x - x_-)) = a_x t + C
$$

With $x = x_0$ at $t = t_0$, the integration constant, C, may be computed, giving

$$
\ln\left(\frac{v_{x-} + a_x(x - x_-)}{v_{x-} + a_x(x_0 - x_-)}\right) = a_x(t - t_0)
$$

This equation is useful to compute at what time the particle hits a cell face given its initial position in the cell. Clearly, its velocity must not be zero (denominator), neither must the velocity at the target position be zero (numerator), or must the fraction be negative (which means opposite velocities at current and target positions, implying a water divide in between).

Reversing this formula yields the position of the particle at a given time

$$
x = x_- + \left(\frac{v_{x-}}{a_x} + (x_0 - x_-)\right)\exp(a_x(t - t_0)) - \frac{v_{x-}}{a_x}
$$

Which may be rewritten in relative coordinates

$$
u = \frac{x - x_-}{\Delta x} = \left(\frac{v_{x-}}{\Delta v} + u_0\right)\exp(a_x(t - t_0)) - \frac{v_{x-}}{\Delta v}; \quad with\ \Delta v = v_{x+} - v_{x-}
$$

This gives the relative position in the cell at given time starting at an arbitrary initial position x_0 in the cell at time t_0. For this to work, the only condition is that $a_x \neq 0$, in which case the velocity is constant and the new position becomes.

$$x = x_0 + v_x\,(t - t_0)\,; \quad or \quad t - t_0 = \frac{x - x_0}{v_x}$$

The model must capture this situation as it needs to use an alternative formula to compute the velocity at another location or the time to get to some other locations (i.e. the cell face).

If

$$\frac{v_{x-} + a_x\,(x - x_-)}{v_{x-} + a_x\,(x_0 - x_-)} \leq 0$$

the logarithm does not exist. The reason is that the velocity at the target location x is opposite to that at the current location x_0 so that the particle never reaches the target location. This happens if there is a water divide between the current and target locations. In this case no time can be computed. On the other hand the position of the particle may then be computed for any time up to infinity. In the model these situations must be captured.

6.1.3 Implementation

The logic of the particle tracking model is as follows:

First compute the velocities at the upstream and downstream sides of all cells, both in x and y direction, using the cell face flows at the cell interfaces and the porosities of this cells.

Then, given an arbitrary point x_p, y_p, find in which cell it is and start tracking during a given time step DT.

In case the velocity is zero the point remains at its position and the time moves on to $t + DT$. Particle position and cell index remain unchanged.

If the velocity is negative, compute movement of particle in the direction of the upstream cell face:

If the velocity is constant, use the appropriate formula. Compute the time dt to reach the cell face. If this time exceeds DT, use DT instead and compute the new particle position using DT. Do not update the cell index.

Else check the velocity direction at the target cell face and see if the particle will ever reach it. If so, compute the time dt until the hit. Provisionally update the particle position to this cell face and reduce the cell index for the x-direction by 1. Now check if $dt > DT$ to see if the target DT is reached before we hit the cell face. If this is the case, use DT instead and compute the particle position using DT. Don't update the cell index.

If the particle will never reach the upstream cell face, use DT and compute the particle position after DT. Don't update the cell index.

The same logic is used for the downstream cell face in the case the velocity is positive.

The same logic also applies for the y-direction.

Finally we have a provisional new position xpN, ypN with provisional change of cell index dic, djc (both -1, 0 or +1) in x and y direction respectively and two times dtx and dty that meet the criteria in both the x and y direction respectively.

The smallest of the two determines the final particle update. In case this is dtx, than the values $xp = xpN$, $ic = ic + dic$ and $dt = dtx$ will hold and the y-position of the particle has to be recomputed using the new dt. In case the smallest of the two is dty it is the other way around.

Clearly, dt may be smaller than the initial target time step DT, as a cell face is hit much sooner. Then DT is reduced by dt and the procedure is done all over again, causing the particle to move through the next cell. This is repeated until DT has become zero. This makes sure that the particle position at given time points will be stored together with the positions and times that a particle crosses cell faces.

The procedure is repeated with a new time step, until all have been worked through.

Because the cell face flows at the outer boundary faces of the model are always zero in the finite difference model, particles can never escape the model and need no special care in that respect. However,

particles will enter extraction cells, where they would simply slow down indefinitely as, because such cells behave as having a water divide inside (or a distributed extraction over the cell area). Therefore, it is better to capture particles entering cells that have an extraction which is beyond a given fraction of the through-flow of the cell. This is the same approach as MODPATH.

The extraction is provided by the output of *fdm2* and the through-flow is computed as the sum of the absolute values of the flows across all 4 cell faces. This threshold fraction may be set at 15\% or so. So the loop is broken off as soon as the particle enters a cell being a sink according to this criterion.

To allow dealing with the x and y (and possibly the z-axis) in the same manner, so that the program uses the same logic for the three axes, one must guarantee that the cell indices are aligned (increase) with the positive axis-direction. This is checked in the beginning and if necessary the concerned matrices are flipped accordingly left-right or upside-down.

To allow backward tracing, the matrices Q_x and Q_y are multiplied by -1. Backward tracing is signaled by using negative times in the input.

The implementation is such that the function *fdm2path* is called after the model has been run and the necessary nodal and cell-face flow matrices computed. The extra information that the *fdm2path* needs is the porosity of all cells and either the thickness of each cell or the sign that the model be computed in radial fashion

```
[XP YP TP]=fdmpath(x,y,DZ|radial,Q,Qx,Qy,por,T,[markers])
```

Use the x,y, Q, Q_x and Q_y matrices that are the output of *fdm2*.

Make sure the size of the porosity matrix equals the size of Q or use a scalar.

Make sure that the absolute values of the time series T are increasing. Use negative values for backward tracing. You don't need to start with a zero first time.

The third argument is either a matrix DZ of the cell thicknesses or a string such as 'R' or 'radial' to indicate the radial symmetric case. You may use a scalar for DZ. An empty matrix [] will be regarded the same as $DZ = 1$.

In the case of radial flow the horizontal and vertical velocities at the cell faces are computed as

$$v_x = \frac{Q_x}{\epsilon \Delta y 2\pi x}; \quad v_y = \frac{Q_y}{\epsilon\pi \left(x_+^2 - x_-^2\right)}$$

The optional markers is a string consisting of letters that are valid Matlab marker indicators. The default is (see doc marker).

```
'+o*.xsdph^v<>'
```

There will be a marker plotted in the paths for each given time (except zero) according to this list of markers, which is repeated of there are more times than markers given. For instance 'oooop' gives you four 'o-markers' and each fifth a 'p'=pentagon.

When the model is run, it picks up the current figure (assuming it is a contour plot generated after running the model) by letting you click at a point in it (left mouse button). The program will immediately compute and show the flow path with the markers (No markers will be visible if the particles leave the model before the first time is reached).

You can repeat this as long as you like.

To stop use the right-hand mouse button. Upon this the program stops and yields the X_P, Y_P and T_P coordinates of all the generated lines. These points and times include all points where particles crossed cell borders and all points at the given times. The individual lines in these matrices are separated by a NaN (Not a Number value). The coordinates at the given times can be picked out of these matrices:

```
for it=1:length(T)
    I= abs(TP-T(it))<1e-6;
plot(XP(I),YP(I),'b');
```

As with all Matlab functions, you can always get help by typing

```
help fdm2path
```

fdm2path has a self test built in. It will be run if you type

```
fdm2path
```

Finally, it is a good exercise to work this out for transient flow. In that case the flows are dependent of the time which requires some extra housekeeping. It is also a good exercise to work this out for 3D, possibly 3D and transient. This is not very difficult, but the convenience will be much less due to more difficult visualization and data handling. If you really need to do a complicated 3D transient modeling project, rather use standard software with an advanced user interface with easy entry and management of the input, output and visualization. Having said all this, the current modeling provided in this syllabus comprise a practical, powerful and efficient modeling toolbox with many uses for practical real-life groundwater modeling.

6.1.4 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

```
%% Axially symmetric flow to a well, check fdmpath:
clear all; close all;
Qo   = 2400;
por  = 0.35;

xGr=logspace(-1,4,51);
yGr=[0 -1 -2];
gr   = grid2DObj(xGr,yGr);

IBOUND = gr.const(1); IBOUND(:,end)=-1;
k      = gr.const(10);
FH     = gr.const( 0);
FQ     = gr.const( 0);
FQ(:,1)= Qo.*gr.dy./sum(gr.dy); % divide Q over the 2 layers

% run flow model fdm2
[Phi,Q,Qx,Qy,Psi] = fdm2(gr,k,k,IBOUND,FH,FQ);
```

```
% contour heads
contour(gr.xm,gr.ym,Phi,30); set(gca,'xlim',[0 3000]);

% path lines
T=3650;                                  % time series (one point only)
[XP YP TP]=fdmpath(gr,Qx,Qy,por,T,'o');  % run fdmpath (verify the call)

% verify travel time
R=sqrt(Qo*T(end)/(pi*por*sum(gr.dy)))    % check this yields 1996 m
```

6.1.5 Example

The previous example showing the cross section with stream lines can be used to demonstrate the particle tracking. The only thing that need to be done, once the figure of the cross section has been plotted is running the lines

```
por = 0.35;
t=365 * (0:10:100);
[XP,YP,TP]=fdm2path(gr,Q,Qx,Qy,por,t,'...p...p...p');
```

or putting them at the end of the script. Then cross hair lines appear allowing to click on any point in the figure to track on particle for the specified time and using the specified makers that the intermediate times given. The result figure is shown below. Particle tracking without using the mouse is possible by adding starting locations as extra inputs to *fdm2path* (see its documentation by typing

```
help fdm2path
```

The only thing necessary to change the particle tracking to axially symmetric flow is using the 'AXIAL',true argument in the call to *grid2DObj* in the script:

```
gr = grid2DObj(xGr,yGr,'AXIAL',true);
```

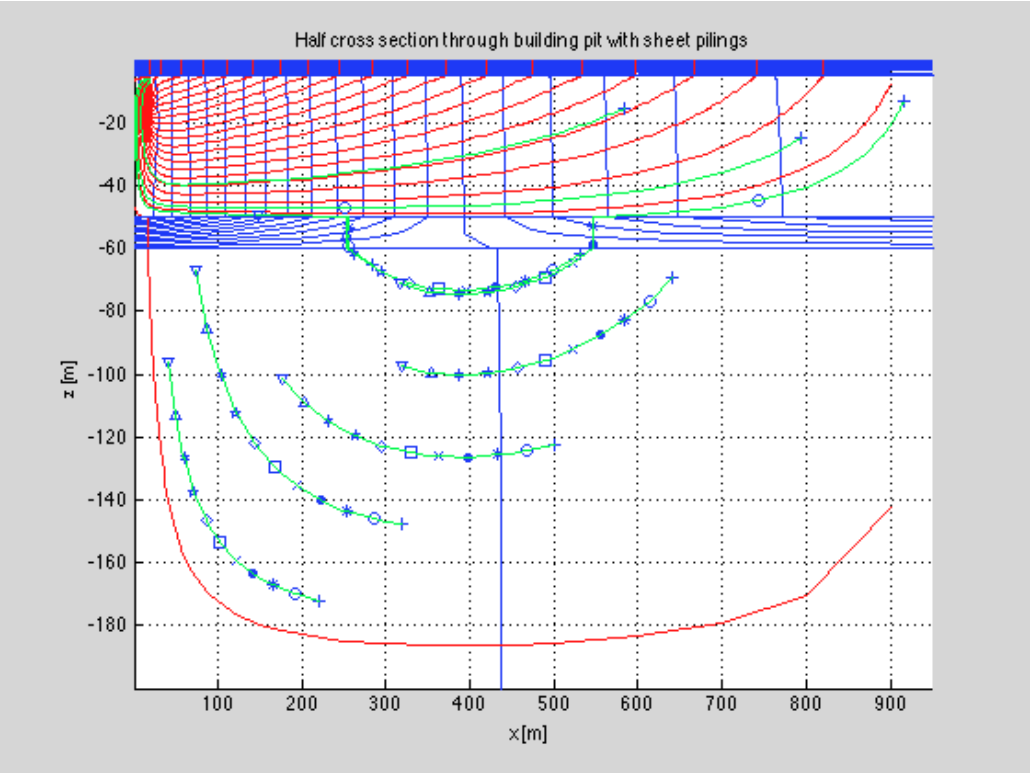


Figure 6.1: Cross section (flat) with heads, streamlines and some particle tracks, obtained by clicking on the figure when *fdm2path* is running (backward traces as times were negative, see input above). There is great detail near the sheet piling where all the streamlines converge, which can only be seen by zooming in.

6.1.6 Listing of fdm2path

```
function [XP YP TP]=fdm2path(gr,Q,Qx,Qy,por,T,markers,XStart,YStart)
%FDM2PATH 2D particle tracking.
%
% Example:
%    [XP YP TP]=fdm2path(gr,Q,Qx,Qy,por,T,markers [,XStart,YStart])
%
% To use this function:
%    generate a 2D steady-state model, launch this function using its produced
%    matrixes Q,Qx,Qy and other necessary parameters DX and por
%    if no starting values are used you must click on an existing picture and
%    the flow path will be immedidately drawn with markers at the given time points in T
%    Repeat this for more lines. Click te right hand button to stop
%    Type fdm2path for selftest and demo
%
% INPUT:
%    gr is a grid2DObj with the information about the mesh
%    So use 'radial' mode, specify gr = grid2DObj(xGr,yGr,'AXIAL'true);
%    Q Qx Qy [L3/T] output of fdm2 (steady state only)
%    por     [ -  ] matrix of porosities (scalar is enough)
%    T       [ T  ] time points where markers are desired, A marker at t=0 will always be placed
%                   Use negative times vor backward tracing
%    markers [  - ] is a series of markers for the consecutive times
%           e.g.    '>+o*.xsdph^v<'. The series will be repeated if necessary.
%    XP YP TP [ L ] coordintates of flow paths, there is a [NaN NaN NaN] between
```

```
%    consecutive tracks if muliple starting points or clicks are used.
%
% See also: fmd2t fdm2c fdm2ct fdm3 fdm3t
%
% TO 070424 070501 140420

% Copyright 2009-2014 Theo Olsthoorn, TU-Delft and Waternet, without any warranty
% under free software foundation GNU license version 3 or later

if nargin==0; selftest; return; end

if nargin<9
    markers='+o*.xsdph^v<>';
end
Lm=numel(markers);

if isscalar(por), por=gr.const(por);   end

x = gr.xGr;
y = gr.yGr;
DZ = gr.dZ;

% first make sure the positive direction of the grid is
% aligned with the positive gr.xGr and y directions
if sign(x(end)-x(1))<0
    x=fliplr(x); Q=fliplr(Q); Qx=fliplr(Qx); Qy=fliplr(Qy); DZ=fliplr(DZ); por=fliplr(por);
end
if sign(y(end)-y(1))<0,
    y=flipud(y); Q=flipud(Q); Qx=flipud(Qx); Qy=flipud(Qy); DZ=flipud(DZ); por=flipud(por);
end

dx= diff(x);
dy= diff(y);

% then check which cell are sinks

if T(end)<T(1)  % if times negative then track particles backward in tme
    Qx=-Qx;
    Qy=-Qy;
    T=-T;
end

sinkfrac=0.25;

Qy(isnan(Qy))=0;
Qx(isnan(Qx))=0;
Q(isnan( Q ))=0;

Qthrough=zeros(size(Q));
if ~isempty(Qx)
    Qthrough=Qthrough+[zeros(size(Qx(:,1))),abs(Qx)]+[abs(Qx),zeros(size(Qx(:,1)))];
end
if ~isempty(Qy)
```

```
    Qthrough=Qthrough+[zeros(size(Qy(1,:)));abs(Qy)]+[abs(Qy);zeros(size(Qy(1,:)))];
end
sink= Q < -sinkfrac*Qthrough;
%figure; spy(sink)
if gr.AXIAL % then the flow is axially symmetric
    fprintf('Fdmpath in radial mode.\n')
    if ~isempty(Qx)
        A=dy*2*pi*x;
        vx2=[Qx, zeros(size(Qx(:,1)))]./(A(:,2:end)  .*por);
        vx1=[zeros(size(Qx(:,1))), Qx]./(A(:,1:end-1).*por);
        ax=(vx2-vx1)./(ones(size(Qx(:,1)))*diff(x));
    end
    if ~isempty(Qy)
        A=ones(size(dy))*(pi*(x(2:end).^2-x(1:end-1).^2));
        vy2=[Qy; zeros(size(Qy(1,:)))]./(A.*por);
        vy1=[zeros(size(Qy(1,:))); Qy]./(A.*por);
        ay=(vy2-vy1)./(diff(y)*ones(size(Qy(1,:))));
    end
else
    fprintf('Fdmpath in flat mode.\n')
    if ~isempty(Qx)
        vx2=[Qx, zeros(size(Qx(:,1)))]./((dy*ones(size(dx))).*por.*DZ);
        vx1=[zeros(size(Qx(:,1))), Qx]./((dy*ones(size(dx))).*por.*DZ);
        ax=(vx2-vx1)./(ones(size(Qx(:,1)))*diff(x));
    end
    if ~isempty(Qy)
        vy2=[Qy; zeros(size(Qy(1,:)))]./((ones(size(dy))*dx).*por.*DZ);
        vy1=[zeros(size(Qy(1,:))); Qy]./((ones(size(dy))*dx).*por.*DZ);
        ay=(vy2-vy1)./(diff(y)*ones(size(Qy(1,:))));
    end
end

XP=([]); YP=([]); TP=([]); j=1;

% startpoints must be inside model
%Iout=find( XStart<min(x) | XStart>max(x) | YStart<min(y) | YStart>max(y) );
%XStart(Iout)=[];
%YStart(Iout)=[];

while 1
    if exist('XStart','var') && exist('YStart','var')
        if j>length(XStart), break; end
        Xp=XStart(j); Yp=YStart(j);  % get starting points for stream lines
        j=j+1;
    else
       % get starting points for stream lines
       [Xp, Yp, button]=ginput(1);  if button~=1; break; end

    end
    DT=diff(T(:)); if T(1)~=0, DT=[T(1);DT]; end %#ok

    for ip=1:length(Xp);
        xp=Xp(ip); yp=Yp(ip); t=T(1);
```

```
XP=[XP;NaN;xp]; %#ok
YP=[YP;NaN;yp]; %#ok
TP=[TP;NaN; t]; %#ok
iLast=length(TP); % to later plot only this  line

ic=find(x<xp,1,'last'); if isempty(ic) || ic==length(x), break; end
jc=find(y<yp,1,'last'); if isempty(jc) || jc==length(y), break; end

line(xp,yp,'marker',markers(1)); hold on;  % initial marker

for idt=1:length(DT);
    dt=DT(idt);
    while dt>0
        if isempty(Qx)
            dic=0; dtx=dt;
        else
            [xpN,dic,dtx]=postime(...
                 xp,x(ic),x(ic+1),vx1(jc,ic),vx2(jc,ic),ax(jc,ic),dt);
        end
        if isempty(Qy)
            djc=0; dty=dt;
        else
            [ypN,djc,dty]=postime(...
                 yp,y(jc),y(jc+1),vy1(jc,ic),vy2(jc,ic),ay(jc,ic),dt);
        end

        [ddt,i]=min([dtx,dty]);

        switch i
            case 1
                if ~isempty(Qy)
                    xp=xpN;
                    yp=pos(yp,y(jc),vy1(jc,ic),ay(jc,ic),ddt);
                end
                ic=ic+dic;
            case 2
                if ~isempty(Qx)
                    xp=pos(xp,x(ic),vx1(jc,ic),ax(jc,ic),ddt);
                    yp=ypN;
                end
                jc=jc+djc;
        end

        dt=dt-ddt; t=t+ddt;
        XP=[XP;xp]; YP=[YP;yp]; TP=[TP;t]; %#ok
        if length(XP)>20000; break; end

      if dt==0
          m=mod(idt+1,Lm); if m==0, m=Lm; end   % the +1 because the first marker is the
          line(xp,yp,'marker',markers(m)); hold on;
      end

      if sink(jc,ic);
```

```
                break;  % from while
              end

            end

            if sink(jc,ic);
              break; % from for
            end
        end
        line(XP(iLast:end),YP(iLast:end),'color','g');
    end
end
XP=[XP;NaN]; YP=[YP;NaN]; TP=[TP;NaN];
end

function [xp,dic,dt]=postime(xp,x1,x2,v1,v2,ax,Dt)
EPS=1e-6;

v=v1+ax*(xp-x1);
if abs(v)<EPS
    % ic=ic
    dt=Dt;  % immediately jumpt to end of time step
    dic=0;
    return; % x remains same location
end

if v<0  % point moves to face at left side
    if abs(ax)<EPS  % v will be constant
        dt=(x1-xp)/v;
        if dt>Dt
            dt=Dt;
            xp=xp+v*dt;
            dic=0;
        else
            xp=x1;
            dic=-1;
        end
    elseif v1>=0           % point will never reach left face
        dt=Dt;         % immediately jump to end of time step
        xp=pos(xp,x1,v1,ax,dt); % compute position at Dt
        dic=0; % ic=ic
    else
        dt=tim(xp,x1,x1,v1,ax);
        if dt>Dt
            dt=Dt;
            xp=pos(xp,x1,v1,ax,dt);
            dic=0;
        else
            xp=x1;
            dic=-1;
        end
    end
end
```

```
if v>0
    if abs(ax)<EPS
        dt=(x2-xp)/v;
        if dt>Dt
            dt=Dt;
            dic=0;
            xp=xp+dt*v;
        else
            xp=x2;
            dic=+1;
        end
    elseif v2<=0
        dt=Dt;
        xp=pos(xp,x1,v1,ax,dt);
        dic=0;
    else
        dt=tim(xp,x2,x1,v1,ax);  % CHECK
        if dt>Dt
            dt=Dt;
            xp=pos(xp,x1,v1,ax,dt);
            dic=0;
        else
            xp=x2;
            dic=+1;
        end
    end
end
end

function xp=pos(xstart,x1,v1,ax,dt)
EPS=1e-6;
if abs(ax)<EPS
    vx=v1+ax*(xstart-x1);
    xp=xstart+vx*dt;
else
    xp=x1+(v1/ax+(xstart-x1))*exp(ax*dt)-v1/ax;
end
end

function dt=tim(xstart,xtarget,x1,v1,ax)
    dt=1/ax*log((v1+ax*(xtarget-x1))/(v1+ax*(xstart-x1)));
end

function selftest
    eval('help fdm2path')
    clear all; close all

    xGr = linspace(-2500,2500,22);
    yGr = linspace(-2500,2500,22);
    gr  = grid2DObj(xGr,yGr,50);

    IBOUND = gr.const(1); IBOUND(:,[1 end])=-1; IBOUND([1 end],:)=-1;
```

```
    k  = gr.const(10);
    FH = gr.const(0);
    FQ = gr.const(0.001*gr.Area);

    [Phi,Q,Qx,Qy]=fdm2(gr,kx,ky,IBOUND,FH,FQ);

    contour(gr.xm,gr.ym,Phi); hold on

    %Track particles
    por = 0.35;
    t=[60 365 3650 25*365 100*365];
    fdm2path(gr,Q,Qx,Qy,por,t,'...p...p...p');
end
```

7 Mass transport and random walk

7.1 Particle tracking for mass transport

7.1.1 Background

This chapter is concerned with tracking of large numbers of particles simultaneously. The developed routine may be used to simulate mass transport and random walk methods within the block finite-difference grid. The groundwater transport code MT3DMS (Zheng, 1999) implements the Method Of Characteristics (MOC) among others. This MOC method requires tracking of a large number of concentration particles through the model, each of which carries its concentration along with it. This massive particle tracking must be done efficiently to keep acceptable computation times. The implementation in MT3DMS is described by [Zheng (1993)]. His method applies the linear variation of the discharge within each cell in each direction, which is inherent to the finite difference method. To speed-up the computation time, Euler is used for all particles within the mesh except for the particles in cells with sources and sinks, for which this method would be too inaccurate. For those cells Zheng (1993) uses a 4-step Runge Kutta method. To maintain sufficient accuracy when applying the simple one-step Euler method, the time step is limited such that no particle passes more than one cell in any direction throughout the model (Courant number). This implies that the number of transport steps may be very large in case the velocity is high and residence time in one or more cells is very short. Further efficiency steps described by Zheng (1993) are reduction of the number of particles automatically where the concentration gradients are low. Further, particles are automatically removed from sink-cells and particles are automatically added to sources cells. In MT3DMS the maximum number of particles in the model is chosen by the user, which means that the array to store them is preallocated and fixed. The place of removed particles is reused by new particles as required.

In this chapter we take a different approach in which the particles are tracked in accordance with the velocity in the rectangular grid cells. The particle movement is computed exactly. Particles are tracked to the cell face and then they may move into an adjacent cell and so on. To maintain computation speed we have to vectorized the computation as much as possible.

7.1.2 Theory

For ease of vectorization of large-scale particle tracking in a finite difference model, the cell coordinates may be scaled such that the width of the cells becomes unity in both the x and y directions. Doing this, we keep the travel the same and the mapping form the original grid to the new grid is unique. Because the discharges at the cell faces, Q, are known from the flow model, we have for the particle velocity:

$$v_{x,i} = \frac{1}{R} \frac{\overline{Q}_{x,i} + \frac{x - x_m}{\Delta x,i} \Delta Q_{x,i}}{\epsilon_i \Delta y_i \Delta z_i}$$

where the index x denotes coordinate direction and index i indicates the cell. R is the retardation, which is larger than 1 in case sorption takes place. Δz_i it the the local of the 2D aquifer that is modeled, which may be different between cells. This formulation is equal in 2 and in 3 dimensions. The velocity in the y direction is obtained by replacing x by y.

Dropping the cell index for convenience, the velocity in x direction for a particle in any cell becomes

$$v_x = \overline{v}_x + \Delta v_x \ (x - x_m)$$

in which $\overline{v}_x$ is the velocity in x direction at the center of the cell

$$\overline{v}_{x,i} = 0.5\frac{Q_{x,i+1} + Q_{x,i}}{\epsilon_i \Delta y_i \Delta z_i}$$

and Δv_x the velocity difference between the downstream and upstream face of the cell:

$$\Delta v_{x,,i} = \frac{Q_{x,i+1} - Q_{x,i}}{\epsilon_i \Delta y_i \Delta z_i}$$

Using relative coordinates $u = \frac{x - x_m}{\Delta x}$ so, where that $-0.5 \leq u \leq 0.5$ with each cell, the velocity v_u in relative coordinates is then obtained by differentiation using the chain rule:

$$v_u = \frac{du}{dt} = \frac{du}{dx}\frac{dx}{dt} = \frac{1}{\Delta x} v_x$$

so that

$$v_u = \frac{\overline{v}_x}{\Delta x} + u\frac{\Delta v_x}{\Delta x}$$

This expression is equivalent to

$$v_u = \overline{v}_u + ua$$

with

$$\begin{aligned} \overline{v}_{u,i} &= \frac{1}{2R_i}\frac{Q_{x,i} + Q_{x,i+1}}{\epsilon_i V_i} \\ a_{x,i} &= \frac{1}{R_i}\frac{Q_{x,i+1} - Q_{x,i}}{\epsilon_i V_i} \end{aligned}$$

In which V the cell volume of the cell. Notice that R may also be considered as variable from cell to cell. As can be seen, retardation if constant in time is numerically trivial.

$\overline{v}_u$ is the velocity in the center of the cell in the a grid where relative coordinates $-0.5 \leq u \leq 0.5$ are used in each cell, as explained above. The quantity a can be seen as the spatial acceleration.

This allows immediate and efficient computation of the velocities in the scaled grid. For any cell i and equivalently for the x and the y coordinate direction parallel to the grid, we have the following relative velocity

$$v = \overline{v} + au$$

$$\frac{du}{dt} = \overline{v} + au$$

so that

$$\frac{d\left(\overline{v} + au\right)}{\overline{v} + au} = adt$$

and, with as boundary condition that $u = u_0$ at $t = t_0$ can be integrated to
So that:

$$\begin{aligned} \ln\frac{\overline{v} + au}{\overline{v} + au_0} &= a\left(t - t_0\right) \\ \frac{\overline{v} + au}{\overline{v} + au_0} &= \exp\left(a\left(t - t_0\right)\right) \end{aligned}$$

For the cases when $a \to 0$, this equation can be more conveniently written in the following form

$$\frac{\overline{v}+au}{\overline{v}+au_0} = (\exp(a(t-t_0))-1)+1$$

and with u explicitly:

$$u = (\overline{v}+au_0)\frac{\exp(a(t-t_0))-1}{a} + u_0$$

In the case when $a \to 0$ the factor with:

$$\lim_{\xi\to 0}(\exp\xi - 1) = \xi$$

the expression with the exponent reduces to

$$\lim_{a\to 0}\frac{\exp(a(t-t_0))-1}{a} = \frac{a(t-t_0)}{a} = t-t_0$$

so that

$$\lim_{a\to 0}\overline{v}\left((\overline{v}+au_0)\frac{\exp(a(t-t_0))-1}{a}+u_0\right)+u_0 \quad = \quad \overline{v}(t-t_0)+u_0$$

or, when $a \to 0$ equation reduces to

$$u = \overline{v}(t-t_0)+u_0$$

which is exactly the expression for cells with a spatially constant velocity.

Hence, in the numerical implementation all cells may be treated equivalently, even in the case where $|a| = 0$ by setting a to some suitable lower limit of for instance $EPS = 10^{-8}$. If desired, this limit can be used to switch between the linear and the exponential expression for the velocity.

The difficulty with particle tracking in the case of a numerical model is to correctly handle particles that pass a cell face during the time step. Such ubiquitous situations should be dealt with because the velocity may jump between particles. The the code for each particle the time is computed till it hits any of the faces of the cell in this is is. If this time is larger than the time until the end of the time step, then the particle is moved entirely within the current cell. If not, than the particle is moved over the time after which it hits the first cell face. This time is subtracted from the time to go during this time step for this particle. The particles is moved into the adjacent cell, thereby updating its cell number and its relative coordinate. The the procedure is repeated until the particle arrives at the end of the current model time step. This is done for up to hundreds of thousands of particles at the same time.

The time to hitting a cell face can be computed for each coordinate direction separately

$$t-t_0 \quad = \tfrac{1}{a} \quad \ln\left(\frac{\overline{v}+au}{\overline{v}+au_0}\right)$$

noting that u_0 is the current starting position of the particle within the considered cell, so that $-0.5 \le u_0 \le 0.5$. For each direction 2 such times can be computed, the time to the downstream face and the time to the upstream face. What the downstream and the upstream faces are can be seem from the velocity direction at u_0. However if both times are quantified the time to the cell face is expected to be positive for the downstream face and negative for the upstream face. The negative time means the particle never gets their, so this can be replaced by $t-t_0 = \infty$, to that the lowest of the two times is the sought one. Using that infinite time should bring the particle exactly at the downstream face. In case the velocity at the downstream case is opposite to the that at the current position, we have a water divide somewhere between the current point and the checked cell face. In that case, the

computed time to reach that face will be a complex number, i.e. a negative logarithm, which also implies that the particle never arrives at the downstream face; instead, it will gradually approach the water divide, while never completely getting there

$$\ln \frac{\overline{v}+u}{\overline{v}+u_0} = \ln \frac{v}{v_0}$$

which is the log of the ratio of the velocities at both points. In the case $\frac{v}{v_0} < 0$ or $\frac{v}{v_0} = -\left|\frac{v}{v_0}\right|$, we may write the log expression in complex form

$$\begin{aligned} \ln\left(-\left|\frac{v}{v_0}\right|\right) &= \ln\left(\left|\frac{v}{v_0}\right| e^{i\pi}\right) \\ &= \ln\left|\frac{v}{v_0}\right| + i\pi \end{aligned}$$

which implies that the result of the negative log is always a complex number that always has the same imaginary part, equal to π. Hence, always when $v/v_0 <= 0$ the particle will never arrive that the downstream cell face and its time to get there can be set to ∞ beforehand. This implies that the particle may be moved using the available time as it will always remain in the same cell.

With $v \approx v_0$ the log will approach zero while $|a| \to 0$. This is the case when the velocity in the cell is constant. There is no really satisfactory approximation for this case to compute the time using some default small value for a because also the sign matters which is indefinite in the log when $v = v_0$. When $|a| < EPS$ we should switch to an equation for constant velocity:

$$t - t_0 = \frac{u - u_0}{\overline{v}}$$

where, like before, $t - t_0 = \infty$ whenever the computed value $t - t_0 < 0$ or when $|\overline{v}| < EPS$.

We analyze this for application as a criterion in the code to decide when to apply the log formula and when to apply the linear formula. To compute the time till the particle hits a cell-face, we write

$$\begin{aligned} t &= \frac{1}{a}\ln\left(\frac{v}{v_0}\right) \\ &= \frac{1}{a}\ln\left(\frac{v_1 - v_0}{v_0} + 1\right) \\ \lim_{v_1 \to v_0} t &= \frac{1}{a}\frac{v_1 - v_0}{v_1} \\ &= \frac{1}{a}\frac{\overline{v} + u_1 a - \overline{v} - u_0 a}{\overline{v} + u_0 a} \\ \lim_{v_1 \to v_0} t &= \frac{u_1 - u_0}{\overline{v} + u_0 a} \end{aligned}$$

However, $v_1 \to v_0$ also implies that $a \to 0$ so that this limit becomes the time computed for constant-velocity cells. Therefore, we have a positive time when the above limit is larger than zero, and we should use a constant-velocity equation when when a is sufficiently small. We could compute T_a and T_v as

$$T_a = \frac{u_1 - u_0}{\overline{v} + u_0 a}, \quad T_v = \frac{u_1 - u_0}{\overline{v}}$$

and decide to use T_a (i.e. the log function) when $abs\,(T_a - T_v) > EPS_T$ and use T_v (i.e. the linear velocity function) otherwise. Of course, the time to use has to be larger than zero. A somewhat simpler approach would be the following. $1/a$ is a time and a measure of the maximum time a particle would need to pass a cell. So to decide when to use the log formula, we can use the criterion $a > \frac{1}{T_{max}}$ and then use the previous time as a second criterion to see if the time to be computed is larger than some minimum time criterion *Tmin.*.

In the model, we have to deal with different coordinate directions at the same time. Therefore, the time to the next cell face is computed for all coordinate directions. The smallest time is the time where the particle hits the cell face. Its cell number is determined by the direction that has the smallest time, where there are 4 directions in a cell of a 2D- finite difference model.

By this approach, there is no need to limit the step size to the Courant number. Particles are correctly tracked through all time steps and through as many cells as necessary during each time step. The time steps can have any value for advection, but should be small enough to obtain a proper computation of dispersion and diffusion.

The outlined procedure is vectorizable and robust. It may result in a method that is faster than that used in MT3DMS because there is no need to limit the transport to Courant numbers. It is definitely more accurate because no integration approximations had to be made.

In the MOC procedure within MT3MS the particles are not indexed, so that it is not feasible to track individual particles over time. In the procedure developed here we start with a fixed number of particles, either distributed in every cell or are just just swarm of particles with initial starting positions, hat are subsequently traced through time. This allows following each individual particle through time, because its index is fixed during the simulation. Particles will get trapped within cells that are sinks, such particles will be made inactive, thus speeding op the tracking the fewer particles are still active. Because the traced particles also remember the cell in which they are at the end of each time step, it is easy to investigate where they are trapped and when.

7.1.3 Random walk: dispersion and diffusion

Dispersion and diffusion are considered the result of random movement of particles on top of the advection described in the previous subsection. The transport flux of dispersion and diffusion combined is given by

$$J = \epsilon \left(a \left| \frac{v}{R} \right| + \frac{D_{diff}}{R} \right) \nabla c$$

where the mass transport vector J is 3D in principle and a is a 9×9 tensor with values depending on the direction of the mass flux with respect to the Cartesian coordinate system. D_{diff} is the diffusion coefficient and ∇c the concentration gradient. When we orient the local coordinate system such that J is parallel to the main axes, then the tensor a will be diagonal with one longitudinal dispersivity a_L in the direction of flow and 2 transverse dispersivities perpendicular to the flow. In 2D in which we work in, there would only be one transverse dispersivity a_T. Hence

$$R \begin{bmatrix} J_\xi \\ J_\psi \end{bmatrix} = \epsilon \left(\begin{bmatrix} a_L & 0 \\ 0 & a_T \end{bmatrix} v_\xi + D_{diff} \right) \begin{bmatrix} \partial c/d\xi \\ \partial c/d\psi \end{bmatrix}$$

where ξ and ψ are the local axes along the flow and perpendicular to it.

Realizing that the spread of any set of particles is due to their random movement, which yields a normal distribution with standard deviation equal to

$$\sigma = \sqrt{2 \frac{D}{R} t}$$

we may immediately compute this standard deviation in both the direction of flow and perpendicular to it

$$\begin{aligned} \sigma_\xi &= \sqrt{2 \left(a_L \left| \frac{v_\xi}{R} \right| + \frac{D_{diff}}{R} \right) (t - t_0)} \\ \sigma_\psi &= \sqrt{2 \left(a_T \left| \frac{v_\xi}{R} \right| + \frac{D_{diff}}{R} \right) (t - t_0)} \end{aligned}$$

If the concentration consists of a large number of particles their displacement can be computed by sampling form the normal distribution with zero mean and the above given standard deviations. This

displacement is for each particle different because of the sampling and depends of the length of the time step $t - t_0$. Of course, this random walk should be applied many times along the path during a time step, but the overall outcome is essentially the same, be it that for large time steps the velocity may change substantially during the time step, which would require such multiple steps.

We may add a physical limit to dispersion, i.e. that it cannot be against the flow. Hence the upstream physical displacement due to dispersion should be

$$\Delta\xi < v_\xi \Delta t$$

where

$$\Delta\xi = N\left(0\,,\,\sqrt{2a_L\left|\frac{v_\xi}{R}\right|\Delta t}\right)$$

where $N(\mu, \sigma)$ is sample form the normal distribution with mean μ and standard deviation σ.

The implementation is straightforward to this point. However particles may now jump from into a different cell without the advection causing it. This implies that after each application of this dispersion, diffusion procedure, we have to look up the particles and figure out what their new cell numbers and relative coordinates are.

Furthermore, particles may jump across model boundaries, out of the grid. To prevent this, the grid outer boundary is considered impervious of particles. Particles that jump across it are mirrored to make sure they stay in.

7.1.4 Decay

Particles can be considered to each carry a mass or a concentration. Conceptually the simplest is considering them to carry a mass of a given species. Due to decay this mass is reduced according to

$$m_{t-t_0} = m_{t_0} e^{-\lambda(t-t_0)}$$

where λ [1/T] is the decay parameter or decay constant and $t - t_0$ is the length of the time step.

This decay is implemented by a property mass of the particles which as an initial value of 1. This mass property is then changed according to the decay formula after each time step. Clearly in this simple approach the mass of each particle will be the same at all times. But this may be changed in the future, for instance when particles with different mass are introduced during the simulation. Of course the λ decay may be different for different species, different particles even and also depending on the location (cell index). This implies that more advances analyses are indeed possible with this approach.

7.1.5 Implementation and use

The implementation is in the form of a Matlab function, Moc. A flow model should be run first to obtain the the flows across the cell faces, after which Moc is called to track the particles

```
[Phi,Q,Qx,Qy] = fdm2(gr,Tx,Ty,IBOUND,STRTHD,FQ);

% Then

P = Moc(gr,Qx,Qy,Peff,R,t,n    ,  'aL',aL,'aT',aT,'Diff',Diff,'lambda',lambda);
P = Moc(gr,Qx,Qy,Peff,R,t,xP,yP,  'aL',aL,'aT',aT,'Diff',Diff,'lambda',lambda);
```

As before, *gr* is a grid2DObj with the grid information, Q_x, Q_y are the flows across the cell faces in x and y direction respectively. *Peff* is the effective porosity, R is the retardation, it may be a scalar or a full matrix with a value for each cell. t a time vector, which does not need to start at zero. n indicates the particles to be uniformly distributed in every model cell; $n \times n$ particles are

placed. Alternatively, x_p, y_p are the coordinates of a swarm of particles to be tracked. There may be several hundred thousands of them. The rest is optional. They are property name property pairs as is usual with many Matlab functions. The property name indicates how to interpret the following input. '*aL*' stands for longitudinal dispersivity [L], a_T for transverse dispersivity, '*Diff*' [L2/T] is dispersion coefficient and '*lambda*' [1/T] is the decay parameter. The values of these parameters may be scalars or full size arrays.

The output P is a *struct array* of size equal to the time vector. This *struct* has telling where the particles were at each point of time, in which cell they were and what their mass was.

7.1.6 Examples

There are two scripts, 'modelScript3' and 'modelScript4' with an example. They set up the flow model, run it, set up the Moc model, run it and visualize the results. modelScript4 has several options, showing uniformly distributed particles, a full swarm of particles and a set of swarms of particles.

Results of the modelScript4 can be found in the *htm* directory with the files belonging to this course., it is also listed below.

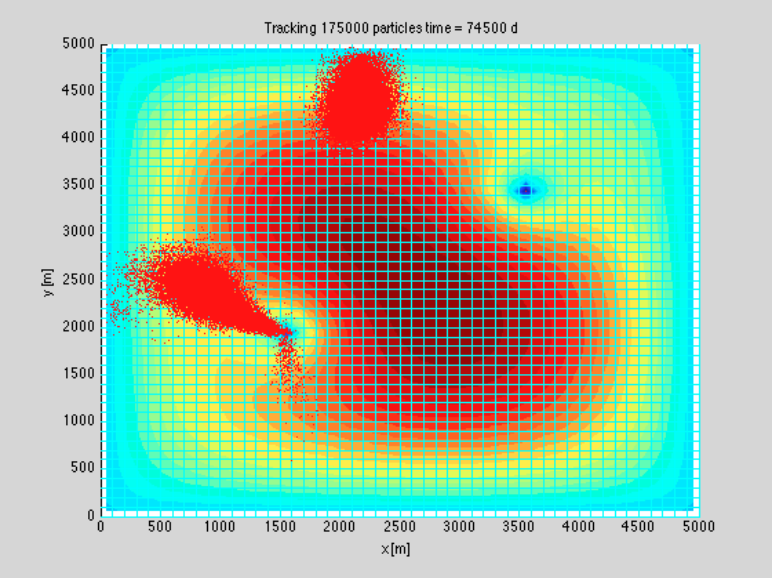


Figure 7.1: Tracking 175000 particles with random walk dispersion and diffusion starting at several positions. Snapshot at 74500 days when most particles were already captured

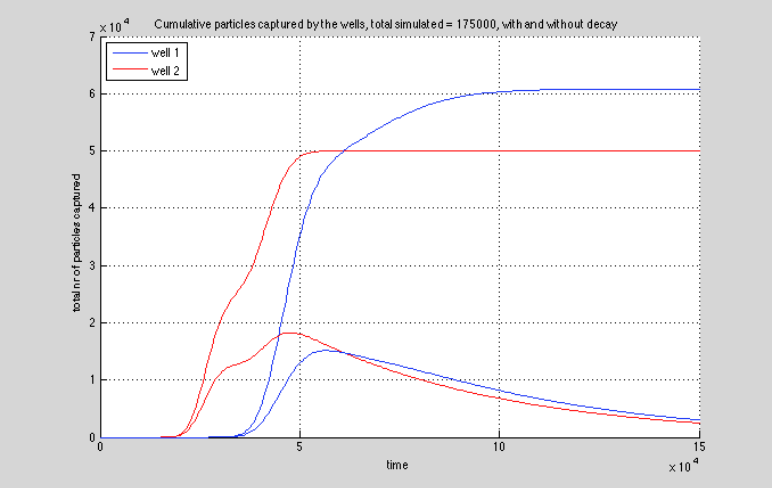


Figure 7.2: Cumulative number of captured particles and their mass under decay

7.1.7 Listing of modelScript4, which uses Moc for simultaneous tracking of massive numbers of particles

```
%%% ModelScript -- Example 4 sets up 2D FDM, runs it and then runs
%   Runs a particle tracking model (Moc method of characteristics)
%   Shows the results
%   TO 140417

%% Cleanup
close all;

%% Constants used in IBOUND
FXHD     = -1;
INACTIVE =  0;
ACTIVE   =  1;

clr = 'brgkmcy';  % list of colors

%% Generate the Grid for th FDM flow modoel
xGr   = 0:100:5000;
yGr   = 0:100:5000;
zGr   = [0 -100];
gr    =  grid2DObj(xGr,yGr,zGr); % generates 2D gridObj with depth

% Show the grid
figure; hold on;
xlabel('x [m]'); ylabel('y [m]');
title('example2: MOC, flow from left to right');
gr.plot('c');  % lines in cyan color

%% Transmissivities
Tx = gr.const(600);
Ty = gr.const(600);

%% Recharge
rch   = 0.001; % net recharge rate
FQ    = gr.Area * rch;

%% Wells
well = [1500,2000,-2400
        3500,3500,-2400];

Idwell     = gr.Idx(well(:,1),well(:,2));
FQ(Idwell) = well(:,3);

IH  = gr.const(0);

%% IBOUND (specifies where heads are fixed)
IBOUND = ones(gr.size);
IBOUND(:,[1 end]) = FXHD;
IBOUND([1 end],:) = FXHD;

%% Run the flow model
[Phi,Q,Qx,Qy] = fdm2(gr,Tx,Ty,IBOUND,IH,FQ);
```

```
%% Setup the particle tracking model
t    = 0:500:150000; % times
Peff = gr.const(0.35); % effective porosity

aL   = 100;   % [ m ] longitudinal dispersivity
aT   = aL/10; % [ m ] transversal dispersivity
Diff = 1e-4;  % [m2/d] diffusion coefficient
R    = 2;     % [ - ] retardation
lambda = 2e-5;% [1/d] decay

%% Generate starting particles and run MOC
swarm      = true;   % a swarm of particles
pointSwarm = true;   % several point swamrs of particles

if swarm
    Np =25000;
    if ~pointSwarm
        x = (rand(Np,1)-0.5)*1000 + gr.xm(round(gr.Nx/2)) ;
        y = (rand(Np,1)-0.5)*1000 + gr.ym(round(gr.Ny/3)) ;

        P = Moc(gr,Qx,Qy,Peff,R,t,x,y ,'aL',aL,'aT',aT,'Diff',Diff,'lambda',lambda);

    else % if pointSwarm
        xc = [1895   2160   2252   2656   3255   2586   1238];
        yc = [3019   3151   2770   2741   2595   1791   2770];
        for i=numel(xc):-1:1
            x((i-1)*Np+1:i*Np) = xc(i);
            y((i-1)*Np+1:i*Np) = yc(i);
        end

        P = Moc(gr,Qx,Qy,Peff,R,t,x,y,'aL',aL,'aT',aT,'Diff',Diff,'lambda',lambda);

    end
else  % uniformly distributed particles (nxn in each cells)
    n = 3;
%    [P,Icells] = Moc(gr,Qx,Qy,Peff,R,t,n); % no dispersion, diffusion, decay

    P = Moc(gr,Qx,Qy,Peff,R,t,n,'aL',aL,'aT',aT,'Diff',Diff,'lambda',lambda);

end

%% Visualize results
close all;

set(gca,'nextplot','add','xlim',gr.xGr([1 end]),'ylim',gr.yGr([end 1]));
xlabel('x [m]'); ylabel('y [m]');
title('example2: MOC, flow from left to right');

% Contour heads
phiMax = max(Phi(:)); phiMin = min(Phi(:));  hRange = phiMin:(phiMax-phiMin)/25:phiMax;
contourf(gr.xm,gr.ym,Phi,hRange,'edgeColor','none');
```

```
gr.plot('c'); % Plot grid on top

%% Show moving particles (stored in struct P)
time = [P.time];
Np   = numel(P,x);
for it=1:numel(time)/2
    % Title will change according to passing time
    ttl = sprintf('Tracking %d particles time = %.0f d',Np,time(it));
    if it==1
        % First loop, title and plot
        ht = title(ttl);
        h = plot(P(it).x,P(it).y,'r.','markerSize',3);
    else
        % Subsequent loops, reset title and points
        set(ht,'string',ttl);
        set(h,'xData',P(it).x,'yData',P(it).y);
        drawnow();  % necessary to update plot
        pause(0.1); % smooth movie
    end
end

%% Plot cumulative number of particles captured by wells
figure; hold on; grid on;
ht = title(sprintf('Cumulative particles captured by the wells, total simulated = %d',Np));
xlabel('time'); ylabel('total nr of particles captured');

% Make array of cell nrs in which particles are after each time step
PIcell = [P.Icells];

leg = [];
for iw = numel(Idwell):-1:1
    h(iw) = plot(t,sum(PIcell==Idwell(iw),1),clr(iw));
    leg{iw} = sprintf('well %d',iw);
end
legend(h,leg{:},2);

%% Plot total mass caputured by well where 1 particle represents 1 mass unit

% Update title of previous graph using handle ht.
set(ht,'string',sprintf('%s, with and without decay',get(ht,'string')));

% This mass is subject to decay
if isfield(P,'mass')
    clr = 'brgkmcy';

    mass   = [P.mass];
    for iw = numel(Idwell):-1:1
        % Add to previous plot for comparison
        h(iw) = plot(t,sum(mass.*(PIcell==Idwell(iw)),1),clr(iw));
    end
    % Also refer to legend of previous plot
end
```

```
%% Plot vectors indicating flow direction and strength

figure; hold on;
xlabel('x [m]'); ylabel('y [m]');
title('Flow model with Quiver');

% Contour heads
phiMax = max(Phi(:)); phiMin = min(Phi(:));  hRange = phiMin:(phiMax-phiMin)/25:phiMax;
contourf(gr.xm,gr.ym,Phi,hRange,'edgeColor','none');

% Show arrows of flow direction and magnitude
qx = [Qx(:,1), Qx, Qx(:,end)]; qx = 0.5*(qx(:,1:end-1) + qx(:,2:end));
qy = [Qy(1,:); Qy; Qy(end,:)]; qy = 0.5*(qy(1:end-1,:) + qy(2:end,:));
quiver(gr.Xm,gr.Ym,qx,qy);

hb = colorbar; set(get(hb,'title'),'string','head [m]')  % Colorbar

%% Check water balance
fprintf('Water balances:\n');
fprintf('Total water balance = %10g (should be zero)\n',sum(Q(IBOUND~=0)));
fprintf('Total recharge (active  cells) = %10.0f m3/d\n',sum(Q(IBOUND>0)));
fprintf('Total discharge(fixhd + wells) = %10.0f m3/d\n',sum(Q(IBOUND<0)));
```

7.1.8 Listing of Moc

```
function P = Moc(gr,Qx,Qy,Peff,R,t,varargin)
% MOC (massive particle tracking, steady state) with random walk and decay
%
% USAGE P = Moc(gr,Qx,Qy,Peff,R,t,n     ,  'aL',aL,'aT',aT,'Diff',Diff,'lambda',lambda);
%       P = Moc(gr,Qx,Qy,Peff,R,t,xP,yP,  'aL',aL,'aT',aT,'Diff',Diff,'lambda',lambda);
%
%  gr = grid2DObj
%  Qx,Qy from [Phi,Q,Qx,Qy] = Fdm2(...)
%  Peff = effective porosity
%  R    = [ - ] retardation, default 1
%  t    = [ T ] times to compute particle position at
%  n    = [   ] number of equally distributed particles per cell
%  xP,yP= [ L ]swarm of particles (real world coordinates)
%  rest of the input parameters is optional
%  aL   = [ L ] longitudinal dispersivity, default []
%  aT   = [ L ] transversal  dispersivity, default []
%  Diff = [L2/T] diffusion coefficient,     default []
%  aL, aT and Diff may be scalar or full cell arrays
%  lambda = decay parameter [1/T], default [];
%  randWalk is only initialize diff all aL, aT and Diff are given.
%  P struct contains P.time, P.x and P.y for all times
%
% TO 140418

% Values necessary to update cell when particle moves into adjacent cell
RWALL= [  -0.5,  +0.5, -0.5,+0.5]; % cell boundaries in relative coordinates
DIDX = [-gr.Ny, gr.Ny, -1  , 1  ]; % Idx to change if moving to neighbor cell
XRNW  = [  0.5,  -0.5,  0  , 0  ]; % Add to create new relative x coordinate
```

```
YRNW  = [  0   ,   0   ,  0.5,-0.5]; % Add to create new relative y coordinate

%% Get data for dispersion and diffusion
[aT  ,varargin]   = getProp(varargin,'aT',[]);
[aL  ,varargin]   = getProp(varargin,'aL',[]);
[Diff,varargin]   = getProp(varargin,'Diff',[]);
[lambda,varargin] = getProp(varargin,'lambda',[]);

if isscalar(R), R = gr.const(R); end

decay        = ~isempty(lambda);

if decay
    if isscalar(lambda), lambda = gr.const(lambda); end
end

mustRandWalk = ~isempty(aT) || ~isempty(Diff);
if mustRandWalk
    if isempty(aL),    aL   = 0;     end
    if isempty(aT),    aT   = aL/10; end
    if isscalar(aL),   aL   = gr.const(abs(aL)  ); end
    if isscalar(aT),   aT   = gr.const(abs(aT)  ); end
    if isscalar(Diff), Diff = gr.const(abs(Diff)); end
end

%% Initials, constant when steady state
dt     = diff(unique([0; t(:)]));
epsVR  = gr.Vol.*Peff.*R;

% Zero flow at extends of model (always) [L3/T]
Qx = [zeros(gr.Ny,1), Qx, zeros(gr.Ny,1)];
Qy = [zeros(1,gr.Nx); Qy; zeros(1,gr.Nx)];

activeCells = ~reshape(...
    Qx(:,1:end-1)>=0 & Qx(:,2:end)<=0 & Qy(1:end-1,:)<=0 & Qy(2:end,:)>=0, [gr.Nod,1]);

% Accelleration [1/T]
ax  =  diff(Qx,1,2)./epsVR;
ay  = -diff(Qy,1,1)./epsVR;

% Velocity in cell centers [1/T]
vx =  0.5*(Qx(:,1:end-1)+Qx(:,2:end))./epsVR;
vy = -0.5*(Qy(1:end-1,:)+Qy(2:end,:))./epsVR;


%% Initialize particles
if numel(varargin)<2
    n = varargin{1};
    [xr0,yr0,Icells,Np] = distrParticles(gr,n);
else
    [xr0,yr0,Icells,Np] = prepSwarm(gr,varargin{1},varargin{2});
end
```

```
xr = xr0;
yr = yr0;

x  = reshape(gr.Xm(Icells),size(Icells)) + reshape(gr.dX(Icells),size(Icells)).*xr0;
y  = reshape(gr.Ym(Icells),size(Icells)) - reshape(gr.dY(Icells),size(Icells)).*yr0;

active = activeCells(Icells);

%% Initialyze time steps
DtRemaining = zeros(Np,4); % DtRemaining computed for each of the 4 directions
DtRest      = zeros(Np,1); % Remaining time in current time step dt(it-1)
tStep       = zeros(Np,1); % Intermediate time step (within step)

%% Initialize output struct
for it=numel(t):-1:1
    P(it).x      = zeros(size(x));
    P(it).y      = zeros(size(x));
    P(it).active = true( size(x));
    P(it).Icells = zeros(size(x));
    if decay
        P(it).mass = zeros(size(x));
    end
end
P(1).time   = t(1);
P(1).x(:)   = x;
P(1).y(:)   = y;
if decay
    P(1).mass(:) = 1;
end
P(1).active(:) = active;
P(1).Icells(:) = Icells;

fprintf('Simulating tracking of %d particles over %d time steps\n.',Np,numel(dt));

%% Loop over times, starting at time 2
for it=2:numel(t)

    fprintf('.');  if rem(it,50)==0, fprintf('%d\n',it); end

    % initialize next time step
    DtRest(:) = dt(it-1);

    Ip = find(DtRest>0 & active);  % Indices of remaining particles

    while ~isempty(Ip) % loop until DtRest == 0 for all particles

        K  = zeros(size(Ip));
        Ic = Icells(    Ip);

        % Forward computation of particles position
        xr(Ip) = estimate(Ip,Ic,ax(:),vx(:),DtRest,xr0);
        yr(Ip) = estimate(Ip,Ic,ay(:),vy(:),DtRest,yr0);
```

```
% particles that passed one or more cell faces
L{1}  = xr(Ip)<=RWALL(1);
L{2}  = xr(Ip)>=RWALL(2);
L{3}  = yr(Ip)<=RWALL(3);
L{4}  = yr(Ip)>=RWALL(4);

% particles that are still within their original cell
L0    = ~(L{1} | L{2} | L{3} | L{4});

% all particles that not hitting a cell face are done in this loop
xr0(   Ip(L0)) = xr(Ip(L0));
yr0(   Ip(L0)) = yr(Ip(L0));
DtRest(Ip(L0)) = 0;

% ===== All particles that hit any cell face (~L0) ====================

% Compute time till particle hits a cell face
DtRemaining(Ip,:) = 0;

for i=1:2  % West and East
    if any(L{i})
        Ipp = Ip(L{i});
        Icc = Icells(Ipp);
        DtRemaining(Ipp,i) = ...
           estimateTrest(Ipp,Icc,ax(:),vx(:),DtRest,xr0,RWALL(i));
    end
end

for i=3:4 % North and South
    if any(L{i})
        Ipp = Ip(L{i});
        Icc = Icells(Ipp);
        DtRemaining(Ipp,i) = ...
            estimateTrest(Ipp,Icc,ay(:),vy(:),DtRest,yr0,RWALL(i));
    end
end

[dt_remaining,kk] = max(DtRemaining(Ip(~L0),:),[],2);

kk(dt_remaining==0)=0;

K(~L0) = kk;

if ~isempty(Ip(~L0))
     tStep(Ip(~L0)) = DtRest(Ip(~L0)) - dt_remaining;
    DtRest(Ip(~L0)) = dt_remaining;
end

% if K==0, i.e. point at cell face,but dt_remaining = 0
% then move point to adjacent cell

for i=4:-1:1
    Ipp = Ip(L{i} & (K==0));
```

```
        if any(Ipp)
            Icells(Ipp) = Icells(Ipp) + DIDX(i);
            if i <=2,  xr(Ipp)= XRNW(i); xr0(Ipp) = XRNW(i); end
            if i >=3,  yr(Ipp)= YRNW(i); yr0(Ipp) = YRNW(i); end
        end
        Ipp = Ip(L{i} & K==i);
        if any(Ipp)
            Icc= Icells(Ipp);
            xr(Ipp) = estimate(Ipp,Icc,ax(:),vx(:),tStep,xr0);
            yr(Ipp) = estimate(Ipp,Icc,ay(:),vy(:),tStep,yr0);

            Icells(Ipp) = Icells(Ipp) + DIDX(i);
            if i <=2,  xr(Ipp)= XRNW(i); xr0(Ipp) = XRNW(i); end
            if i >=3,  yr(Ipp)= YRNW(i); yr0(Ipp) = YRNW(i); end
        end
    end

    Ip = Ip(DtRest(Ip)>0);
end

active = activeCells(Icells);

xr(~active) = 0;       % put particles in sinks in
yr(~active) = 0;       % center of sink

x  = reshape(gr.Xm(Icells),size(Icells)) + xr.*reshape(gr.dX(Icells),size(Icells));
y  = reshape(gr.Ym(Icells),size(Icells)) - yr.*reshape(gr.dY(Icells),size(Icells));

Ip = active;
Ic = Icells(active);

if mustRandWalk && ~isempty(Ic)
    % dx and dy are based solely on the velocity at the end points
    [dxRW,dyRW] = ...
        randWalk(gr,Ip,Ic,xr,yr, ...
             ax(:),ay(:),vx(:),vy(:),aL(:),aT(:),Diff(:),R(:),dt(it-1));
      x(Ip) = x(Ip) + dxRW;
      y(Ip) = y(Ip) - dyRW;

      % mirror particles that jumped over the model boundary
      I = x<gr.xGr(  1); x(I) = 2*gr.xGr(  1)-x(I);
      I = x>gr.xGr(end); x(I) = 2*gr.xGr(end)-x(I);
      I = y>gr.yGr(  1); y(I) = 2*gr.yGr(  1)-y(I);
      I = y<gr.yGr(end); y(I) = 2*gr.yGr(end)-y(I);

      if any(isnan(Icells)), error('stop'); end
     [Icells(Ip),~,~,~,~,xr(Ip),yr(Ip)] = gr.Idx(x(Ip),y(Ip));
     active(Ip) = activeCells(Icells(Ip));
end

% store output in struct
P(it).time   = t(it);     %#ok
P(it).x(:)     = x;       %#ok
```

```
    P(it).y(:)      = y;      %#ok
    P(it).active(:) = active; %#ok
    P(it).Icells(:) = Icells; %#ok

    if decay
        P(it).mass      = P(it-1).mass .* ...
            exp(-reshape(lambda(Icells),size(Icells)).*dt(it-1)); %#ok
    end

    % update which particles are active
    xr0(active) = xr(active);
    yr0(active) = yr(active);

end
fprintf('%d done\n',it);

%% Functions used above

function [xr0,yr0,Icells,Np] = distrParticles(gr,n)
    %distrParticles -- generate n by n particles in all cells
    % USAGE: [xr0,yr0,Icells,Np] = distrParticles(gr,n)
    %  if n<0, particles are randomized by adding a random number between
    %  0 and 1 ot their relative cell coordinate and mirroring it at their
    %  cell face to keep the particles in their origial cells.
    %  TO 140418
    randomize = n<0;
    n = abs(n);
    u = (1/(2*n):1/n:1-1/(2*n))-0.5;
    for i=n:-1:1
        uu = u(i);
        for j=n:-1:1
            IPnt =  (gr.Nod*(n*(i-1)+j-1) +1):gr.Nod*(n*(i-1)+j);
            vv = u(j);
            xr0(IPnt,1)  = uu;
            yr0(IPnt,1)  = vv;
            Icells(IPnt,1)  = 1:gr.Nod;  % ix of these points
           % active(IPnt,1)  = activeCells;
        end
    end
    Np = gr.Nod * n*n;     % Total number of particles

    if randomize
        xr0 = xr0 + rand(size(xr0))-0.5;
        yr0 = yr0 + rand(size(yr0))-0.5;

        %mirrors at cell faces to keep particles in their original cells
        xr0(xr0<-0.5) = -1 - xr0(xr0<-0.5);
        xr0(xr0> 0.5) =  1 - xr0(xr0> 0.5);
        yr0(yr0<-0.5) = -1 - yr0(yr0<-0.5);
        yr0(yr0> 0.5) =  1 - yr0(yr0> 0.5);
    end

function [xr0,yr0,Icells,Np] = prepSwarm(gr,xP,yP)
```

```
    %prepSwarm -- prepares a swarm of particles
    % USAGE: [xr0,yr0,Icells,Np] = prepareSwarm(gr,xP,yP)
   [Icells,~,~,~,~,xr0,yr0] = gr.Idx(xP(:),yP(:));
    L      = ~isnan(Icells);
    xr0    = xr0(L);
    yr0    = yr0(L);
    Icells = Icells(L);
    Np     = numel(Icells);

function r = estimate(Ip,Ic,a,v,Dt,r0)
    %ESTIMATE -- compute new relative coordinate (x or y direction)
    %USAGE: xr = estimate(Ip,Ic,ax,vx,Dt,xr0)
    % r, r0 = relative coordinate

    T_MAX = 1e+6;

    r = zeros(size(Ip));

    L = abs(a(Ic))>1/T_MAX;

    if any(~L)
        r(~L) =  v(Ic(~L)).*Dt(Ip(~L)) +r0(Ip(~L));
    end
    r( L) = (v(Ic( L)) + a(Ic( L)).*r0(Ip( L))).* ...
            (exp(a(Ic(L)).*Dt(Ip(L))) - 1)./a(Ic(L)) + r0(Ip(L));

function tRemaining = estimateTrest(Ip,Ic,a,v,DtRest,r0,RWALL)
    %ESTIMATETREST -- compute remaining time given RWALL and DtRest
    %USAGE: tRemaining = estimateTrest(Ip,Ic,a,v,DtRest,r0,RWALL)
    T_MIN = 1e-3; % about a minute;
    T_MAX = 1e6;  % about 1000 years
    tRemaining = zeros(size(Ip));

    % Note that dt = log(v1/v0)/a) and that v1/v0 >0 is required to yield a
    % positive dt. In the limit this log becomes
    % dt = log(dV/V0-1)/a = dV/(aV0)
    % which equals dt =(r-r0)/(V0+r0a)
    % so this is the correct time check to judge whether the log gives a
    % postive time and when r0a matters, it does when T_MAX>1/a, i.e.
    % then a>1/T_MAX

    La   = a(Ic)>1/T_MAX & (RWALL-r0(Ip))./(v(Ic)+r0(Ip).*a(Ic)) > T_MIN;
    Lv   = ~La           & (RWALL-r0(Ip))./ v(Ic)                > T_MIN;

    % Handle points in cells with abs(a)>EPS first, i.e. use log
    arg = (v(Ic(La)) + a(Ic(La)) .* RWALL)./ ...
          (v(Ic(La)) + a(Ic(La)) .* r0(Ip(La)));
    tRemaining(La) = DtRest(Ip(La)) - log(arg)./a(Ic(La));

    % Handle point in cells with abs(a)<T_MIN second, i.e. constant velocity
    tRemaining(Lv) = DtRest(Ip(Lv)) - (RWALL - r0(Ip(Lv)))./v(Ic(Lv));

    % assembling
```

```
    tRemaining(tRemaining <             T_MIN) = 0;
    tRemaining(tRemaining > DtRest(Ip)- T_MIN) = 0;

function [dx,dy] = randWalk(gr,Ip,Ic,xr,yr,ax,ay,vx,vy,aL,aT,Diff,R,dt)
    %randWalk -- take a random walk stap
    % USAGE [xr,yr] = randomWalk(xr,yr,ax,ay,vx,vy,aL,aT,Diff,dt)
    VX     = (vx(Ic) + ax(Ic) .* xr(Ip)).*reshape(gr.dX(Ic),size(Ic)); % true VX
    VY     = (vy(Ic) + ay(Ic) .* yr(Ip)).*reshape(gr.dY(Ic),size(Ic)); % true VY

    V      = sqrt(VX.^2 + VY.^2); % absolute velocity

    sigmaL = sqrt(2 * (aL(Ic).*V+Diff(Ic))./R(Ic) * dt); % logitudinal
    sigmaT = sqrt(2 * (aT(Ic).*V+Diff(Ic))./R(Ic) * dt); % transverse

    dsLong = randn(numel(xr(Ip)),1).*sigmaL; % random longitudinal displacement

    % upstream dispersion is physically 0
    L          = dsLong < -V*dt./R(Ic);
    dsLong (L) = -V(L)*dt./R(Ic(L));

    dsTrans= randn(numel(xr(Ip)),1).*sigmaT; % transverse displacement
    dx     = (VX.*dsLong -VY.*dsTrans)./V;   % total in x-direction
    dy     = (VY.*dsLong +VX.*dsTrans)./V;   % total in y-direction
```

8 Calibration

8.1 Introduction and background

Calibration is often applied to optimize model parameters such that the model results match the measurements in the best possible way. The model can be any model that can be computed and can produce the measured quantities, to that the model computed values can be compared with the actual measurements. Calibration is as old as modeling and even hand-computing. In the past it was done analytically or, when that was not possible by trial and error. Trial and error has been the most used method even up to the early 1990s when automated optimization of parameters became increasingly popular. Not only that, automated optimization of model parameters is a must when more than a handful of parameter needs to be calibrated. Often a large number of parameters is involved in model calibration, but that is not always warranted by enough data of sufficient quality. The more parameters, or rather degrees of freedom, have to be calibrated by confronting the model results with actual data, the better the fit but the lower the reliability of the thus obtained parameter values. It is the general problem of equifinality or over-fitting a model with a large number of degrees of freedom with too little independent data or information to allow this.

A large model may have millions of parameters, such as the conductivity of all the cells of a finite difference model. This enormous number of degrees of freedom has to be drastically reduced, often to a hand full or say less than 10 or 20. This can be done by assigning zones comprising large numbers of cells that will have the same, yet unknown, parameter values. Another way, but similar is to specify a set of parameters that will generate a conductivity field in a kind of random ways, using methods from geostatistics. Although no two cells may have the same parameter value, the overall number of degrees of freedom is reduced to the parameters that were used in generating the conductivity field for example. Both methods are forms of regularization. Other forms of regularization may be for instance linking conductivities to geological formations, while those formations are mapped and their thickness and position assumed given. One should notice however, that whatever form of regularization is applied, the zoning itself and the position and extent of the layers are assumed fixed, that is, the conceptual model or the model structure is assumed fixed. It almost goes without saying that the largest uncertainty of a certain model may be in just these zonings and layer positions. If these are not challenged in the calibration, and when they are actually wrong, the result of the calibration may be such that the parameters are adjusted to enforce the best possible match between data and the wrong model. Hence, the parameter values would make no sense in that case. To some extent parameter values always try to compensate errors in the conceptual model and even errors in the measurements themselves. It than becomes clear what has been said about the reliability of the model when calibrated with too many parameters. The match between the model output and the parameters may seem satisfactory, however this was only due to the large flexibility of the wrong model to adjust to the given data. Better fit, but less reliable. The consequence is, that this model with the so-fixed parameters values is very unlikely to match new independent data that will come available in the future. This is the same a saying that the predictions with such a model will probably be wrong when the use of the model is outside the narrow scope of the available measurements. The lesson to take home is that one should calibrate any model with the least possible parameters and still achieve a good match.

Another issue is dependence or correlation between calibrated parameters in a model. This is often found and can be contributed to insensitivity of the underlying model with either one of the correlated parameters but rather with their product or ratio. Such situations can readily be investigate with analytical groundwater flow formulas for example, by inspection of the parameters. For instance the Theis solution for the transient drawdown near a well in an aquifer is

$$s = \frac{Q}{4\pi kD} W\left(\frac{r^2 S}{4kDt}\right)$$

It shows that this drawdown is determined by only two independent parameters instead of 8, namely Q/kD and $r^2S/(kDt)$. With Q and t considered independent, the parameters would be kD and r^2S/kD. In fact we generally write kD for transmissivity, which already illustrates the point that in aquifer flow only the product of k and D matters. When working out a transient pumping test of which we expect the drawdown in observation wells to obey Theis' formula, we plot the drawdown s/Q versus t/r^2, which illustrates that not t or r are independent but only t/r^2.

Another example is the head in an aquifer between ditches with recharge

$$h = \frac{N}{8kD}\left(L^2 - x^2\right)$$

We see that the head does not depend on the transmissivity and the recharge, but on their ratio. This is generally also true for the real world in which a water-table aquifer is fed by recharge and bounded by some open water. The correlation between N and kD is 100%. It is therefore useless and pointless to try and calibrate the recharge together with the transmissivity from groundwater heads alone. Only the ratio N/kD matters. Therefore, recharge has to be determined independently for a useful calibration of aquifer parameters. In fact, any parameter that can be determined independently from the model should be so determined and not be made subject to the calibration. Independent information should always be valued and preferred over calibrated values.

In all such situations, it does not make sense to calibrate the parameters in the formula independently, they are heavily correlated, and hence they are difficult to determine independently from a single pumping test alone. The more parameters we choose to calibrate, the larger will the number and extent of correlations between them. The statistical results, such as correlations and confidence levels are part of the result of a sound model calibration.

Automatic calibration can be embedded in a model code, such as is the case with MODFLOW 2000. But it can also be done using an independent calibration code, like it is the case with the famous codes PEST, UCODE and ModelMuse. Although these codes have first been made for the calibration of groundwater models, they are independent of them and are now used for any other models worldwide. The popularity and versatility of these codes has probably made the USGS decide to remove the embedded calibration code form MODFLOW 2005 and any later versions. The difference between an embedded and an external code for calibration is as follows:

Embedded code is fast because internal and programmed in the underlying computer language with no external communication through files is necessary. It is also fast because it can compute the sensitivities internally, often based on efficient implementation of their analytically derived solutions. Embedded calibration code is also easy to use as it is always present and the step from forward modeling to calibration is natural without complex administration and extra work. Visualization is also generally include in such code. Next to MODFLOW 2000, the *Microfem* finite element model has parameter optimization embedded. Pumping test analysis programs are classic examples of codes that both simulate groundwater drawdowns forwardly and enable calibration of subsurface parameters based on the measurements. A good general extremely versatile pumping test analysis program is MLU (www.Microfem.com). A somewhat limited version can be downloaded while a full course is also present on the site.

External codes like PEST and UCODE are independent of the model code. They generally take charge of the model and run it as many times as is needed to obtain the optimal parameters. In this way at least $N + 1$ model runs are required to obtain the full sensitivity matrix, the Jacobian, which allows one optimization step. In generally requires 10-50 of such full sensitivity computations to obtain the optimized parameters and their calibration statistics. For 20 parameters this would come down to 200-1000 forward model runs. The number of runs required depends on the convergence of the calibration, which is monitored by the calibration code. The more parameters, the higher the correlations between them and the smaller their sensitivities, the longer it takes. If the independent

information based on what the model is calibrated is insufficient we enter the realm of equifinality and the calibration will not converge. This should be regarded as a positive sign to rethink the model and its parameters and act accordingly to embed it in more sound data with fewer assumptions.

The power of such independent calibration codes is their versatility; they can be used with any model, also models that have never been made with calibration in mind. Moreover, calibration can be more integrated, for instance, a groundwater management may be optimized by running a groundwater code and an economic code next to each-other, while exchanging the data between them. Also, model codes and calibration codes can be further developed independently. Also, these parallel developments require completely different skills. Generally those general purpose calibration codes are statistically versatile; they include an extended set of of statistical analysis methods not found in optimization code embedded in for instance groundwater models.

Calibration generally is interpreted as the optimization of a user chosen set of parameters, which are generally regularized model parameters as explained above. Automatic calibration is generally the wrong word, because the only thing automatically done by the code is determining the values of the set op the user chosen parameters by which the computed model values match the measured values as closely as possible, if the calibration succeeds. Most work is, however, on the side of the user, namely choosing and preparing the parameters to be optimized and judging the calibration results and acting upon them. Action is almost always necessary as the results of the calibration run will probably tell that the model is unable to fit the data satisfactorily. This is the case if the errors between final model and measurements are far from randomly distributed. Such a situation points at errors in the conceptual model of one sort or another. And the user has then the expensive and important obligation to investigate the likely cause of it. He must do this by exploring independent information about the structure of the model. More measurement will generally not help much, what is needed is more information about the structure of the subsurface, which requires new drilllings of a reevaluation of or additional campaigns of geophysics for instance, if not pumping tests or independent determination of the recharge. And so on. There is no guarantee of success, however.

A model may be deemed successful if it not only does a good job in history matching but also in predicting future behavior. Predicting the behavior of for instance the groundwater system with a model beyond what was explicitly included in the calibration of the past, i.e. its history matching, may very much depend on parameters not include or important in the previous calibration. The extent of an aquifer may not have been important during the pumping tests, that were calibrated and based upon the new pumping station was designed and realized. If the extent of the aquifer were limited, then after some initial successful period, drawdowns may increase unexpectedly up to the point where the required performance of the station can no longer be attained. There are examples of such situations (e.g. Brunn, on Botswana). Therefore one should be warned and not be too optimistic on the results of computed assisted calibration of groundwater models. Although it cannot be a panacea for everything and one must remain critical, computer assisted calibration is used worldwide and is extremely useful to better come to appreciate the peculiarities of the groundwater system and our uncertainty about them.

8.2 Calibration in action

The idea is that the model, in our case a groundwater model input depends on a set of parameters. With a concrete values for these parameters, the model can be run and the quantities computed for which there also exist measurements. Measurements can be of any type as long as they can be produced directly or indirectly from the model output. These measurements are compared with the computed quantities by some concrete norm. This can be the squared sum of differences between the values produced by the model and the measurements, which has the disadvantage that quantities of incompatible dimensions are being compared. It is generally better to compared weighted differences between model and measurement, where the weighting is done by the standard deviation of over the quantities of similar type, like heads separately from flows. Weighting has the advantage that the compared quantities are dimensionless and scaled such that their expected values are the same, namely unity. The norm of cost function to measure the deviation between model and measurements may then

be written as follows

$$I = \frac{\mu_h}{N_h} \sum_{i=1}^{N_h} \frac{\left(\hat{h}_i - h_i\right)^2}{\sigma_h^2} + \frac{\mu_Q}{N_Q} \sum_{j=1}^{N_Q} \frac{\left(\hat{Q}_j - Q_j\right)^2}{\sigma_Q^2} + \frac{\mu_P}{N_P} \sum_{k=1}^{N_P} \frac{\left(\hat{P}_k - P_k\right)^2}{\sigma_P^2} + ...$$

where h stands for heads, Q for flows and p for other a priori information about the values of the calibrated properties. Each group of measurements or information has its own standard deviation, or standard error if computed based on the model outcomes proper. The coefficients μ allow for weighting the the different groups mutually. To weight them equally, all μ should be unity. The choice of the μ values will be by the user and kind of arbitrary depending on how important the different properties are considered relative to the other groups.

The a priori group with parameter values is special. It allows to weight information on the calibrated parameters directly. This prevents the calibration process to prevent drifting of the parameter values too much form their expected values. Parameters that are not sensitive to the data will automatically stay around their initial, i.e. expected value. Furthermore, adding a priori information helps overcome equifinality because now every parameter is given some sensitivity with respect to the actual parameters used in the model. Although a priori info may be useful in certain and even many circumstances it should be prevented that the calibration process becomes artificial, i.e. dominated by fake a priori data. Therefore the parameter group should be considered with care. On the other hand truly available a priori information can be embedded in this way.

Another way to embed a priori information such as transmissivities obtained from pumping tests data, and therefore, locally know transmissivities is to embed them in the generation of the transmissivity field that i used in the model. Such a field may be generated by a few parameters to be calibrated but this is then done in such a way that the transmissivities at locations with known values are fixed, while these values are fixed to a lesser extent the larger the distance from those points. Several geostatistical or other interpolation method allow for such conditioning of a generated field.

8.3 Use linear or log transformed parameters?

We may chose to calibrate parameters as they are or as their log-transformed values. There are many reasons to prefer log-transformed parameters in the calibration. Firstly the log-transformed parameters are dimensionless. Secondly most physical parameters in the model, such as hydraulic conductivity, storage coefficients, hydraulic resistances etc, are positive with their natural lowest value equal to 0 but with a virtually unlimited upper value. Such values may vary many orders of magnitude within a single model. Their statistical distribution will naturally be skewed upward, but will be normal if transferred to logarithmic scale. On log-scale, the value can vary between $-\infty$ and $+\infty$. Adding or subtracting to or from a log-transformed parameters is equivalent to multiplying or dividing the parameter by some value. If we start with a given expected parameter of for instance the hydraulic conductivity, then we may calibrate of optimize a multiplication factor to that conductivity, which is natural. This multiplier will then be calibrated on log-scale:

$$k = k_0 \exp(p)$$

This formula translates the optimized parameter value p to the actual value used in the model, k, where k_0 is the initial parameter value.

Not all parameters can be calibrated this way. For instance for a head it would not work because a head has no natural zero. Mixing linear and log-transformed parameters is possible, but has a great disadvantage of comparing apples and pears in the same box. This makes no sense in general. We can, however, still use log transformed parameters also in this case by writing

$$h = h_0 + \exp(p)\,\Delta h$$

This formula translates parameter p to a non-log parameters h, where h_0 is a fixed constant and Δh functions as a weight or scale factor.Both h_0 and Δh have to be chosen by the user in a logical and sensible way. As before p is the parameter calibrated and h is the value used in the model.

This way, we can treat all parameters in the same way by linearly. The calibration is only concerned with the vector of p values, where p_i is parameter i. Notice that parameters p and properties or a priori information P do not have to be the same thing as certain properties may be computed using parameters. In fact the calibration system is extremely flexible, which is a plus for the use of external optimization software. The disadvantage is the burden on specifying parameters from the side of the user.

8.4 Calibration in Matlab, with example

While a groundwater model is relatively in forward mode, i.e. head changes and flows tend to change fairly linearly with parameters, this is definitely not the case the other way round: i.e. the dependence of optimal parameters on heads and flows is strictly non-linear. Optimization, i.e. the search for the parameter values that given the lowest cost function must, therefore be done iteratively, whereby the sensitivities are updated at every calibration step. Matlab has functions that allow performing such a non-linear search or optimization:

```
p = lsqnonlin(@FUN,p0,LB,UB,options,varargin);
```

This function takes a number of parameters among which the parameter vector p_0 and updates it to yield the optimal parameter vector p.

@FUN is a function, i.e. a pointer to a function, *FUN*, that accepts the parameter vector and yields the differences between the model computed and measured quantities. *lsqnonlin* will run this function to obtain the sensitivities of the differences between model and measurements and then change the parameter set. This is done repeatedly until the optimal set is obtained or the maximum number of iterations is exceeded. Different options can be specified in the input struct *options*. LB and UB are optional lower and upper parameters boundaries, both vectors with the same size as p and p_0. The input parameter *varargin* allows for more inputs as described in the documentation. Next to just p many more output parameters are possible, which are also described in the documentation.

The essence is the function *FUN*, which is a user defined function of the following form

```
e = FUN(p)
```

where p is the parameter vector and e is a vector of differences between model and measurements. The name of the function is free, of course.

It is clear that FUN is a wrapper around the entire groundwater model. It accepts the parameter values, translates these to model parameter values, prepares the input of the model and runs the model, after which it extracts the model computed measurements and subtracts the actual measurements assembling them into a difference vector, e, will be the the output of *FUN*. It is also clear that such a wrapper function will be dedicated to a particular calibration problem.

One yet unsolved issue is how to pass additional data and parameters values to FUN in order to use them to set up the model input and work with its output? One option, obviously is reading the measurements from the hard-disk every time FUN is run. Another method is defining global parameters both in Matlab's workspace and in the function FUN, so that FUN as access to parameters present in the workspace. Although generally not preferred, the latter method has the advantage of zero computational overload. No reading data no transfer of data in memory.

The documentation in Matlab writes the optimization problem as follows

$$\min_p \left(|f(p)|_2^2 \right) = \min_p \left(f_1(p)^2 + f_2(p)^2 + f_3(p)^2 + ... f_n(p)^2 \right)$$

where the left hand says that the 2-norm of the vector f is minimized and the right hand side writes this as a sum of individual values. This is equivalent to

$$\min_p \left(|e|_2^2 \right) = \min_p \left(e_1^2 + e_2^2 + e_3^2 + ... e_n^2 \right)$$

where each e can be a an arbitrary function of the parameter set p including their weighting.

If we have, for instance built the vector e of differences between model and measurements and we desire to minimize the weighted function

$$\min \sum \mu_i e_i^2$$

we would make sure that the output vector of the function FUN is the vector $\sqrt{\mu} \cdot \mathbf{e}$.

To translate the cost function I, that was given above, into the current framework we would generate the following output vector by taking the individual terms like $\hat{h}_i - h_i$, $\hat{Q}_j - Q_j$ and $\hat{P}_k - P_k$ that are in the sum and divide them by respectively $\sigma_h \sqrt{\frac{N_h}{\mu_h}}$, $\sigma_Q \sqrt{\frac{N_Q}{\mu_Q}}$ and $\sigma_P \sqrt{\frac{N_P}{\mu_P}}$.

For instance, assume we have to analyze a pumping test using Theis' formula and and the measurements. The function FUN would then look like this

The function *FUN* can be simple as follows

```
function e = FUN(p)
% used with simple calibration (modelScript5), called by lsqnonlin
global eMeas Q T0 S0 r

fprintf('.'); % shows each call by printing a dot

t = eMeas(:,1);

% Parameter update
T = exp(p(1)) * T0;
S = exp(p(2)) * S0;

% compute drawdown (would normally be hte model)
s = Q/(4*pi*T) * expint( r^2 *S./(4*T*t));

% output difference with measurements
e = s(:) - eMeas(:,2);
```

8.5 Example

The above method can be extended to other analytical formulas used with pumping tests or with arbitrary numerical models as outlined in the text. The example is implemented in the modelScript5, which is listed below. It can be easily extended to calibrate arbitrary simple and complex models of any type.

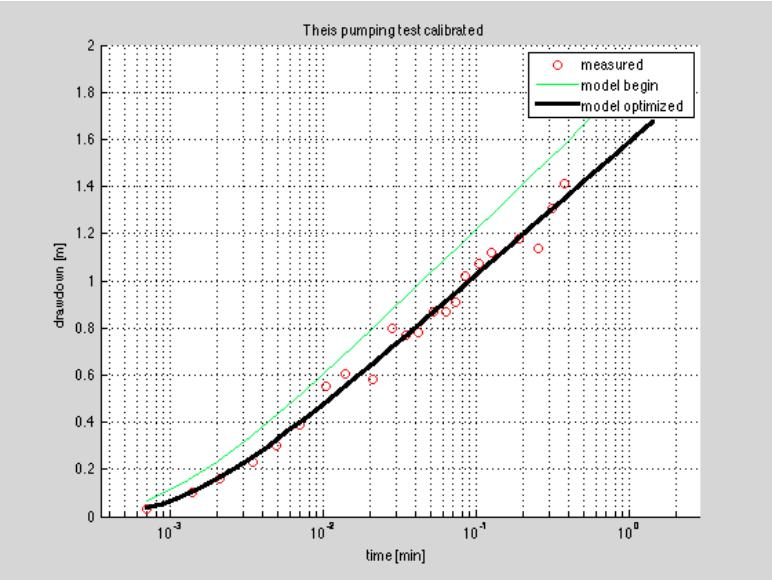


Figure 8.1: Calibration result showing the measurements, the initial model and the final model after calibration

8.6 Statistical analysis of the calibration results with example

8.6.1 Basics

The results of every calibration should also be statistically analyzed. The final differences between the model and the measurements provide the essential data for this in the vector e

The parameter uncertainties are then given by

$$cov\left(p\right) = \sigma^2 \left(JJ^T\right)^{-1}$$

where

$$\sigma^2 = \frac{e^T e}{N - M}$$

where $e^T e$ is the sum of the squared differences between model and measurements and $N - M$ the number of degrees of freedom, i.e. the number measurements minus the number of parameters.

The diagonal of the covariance matrix are a linear approximation of the variances of the optimized parameters, i.e. of the p-values.

The correlation matrix of the parameters is can be obtained by dividing each element of the covariance matrix with the corresponding values of dd^T where d is the diagonal of the covariance matrix:

$$corr\left(p\right) = cov\left(p\right) ./ \left(dd^T\right)$$

Both the variances and the correlation need to be looked at to judge the result of a calibration.

A very important point for the user is to judge whether the calibration is satisfactory. For a groundwater model in which heads are to be matched it is expected that the heads errors are distributed at random in space (and time). This can be judged by the eye when the errors are plotted on the map in the form of bubbles whose size matches the error and whose color indicates whether the error is positive or negative. If the distribution is not random, then the data still contain information that can not be mimicked by the model and the model needs to be revised, probably including the parameter choices and based on independent data. Only after a number of such cyclic model improvements the model is judged satisfactory it makes sense to look at the actual optimized parameter values. The calibration is to be considered successful when the errors are randomly distributed, because than all information in them has been optimally used and extracted by the model.

The confidence band for the parameters can be estimated from the root of the values of the diagonal of the covariance matrix. However, these values only approximate the true standard deviation of the parameters to the extent in which the calibration is linear (which it is not) and to the extent in which the off-diagonal values are close to zero compared to the diagonal values.

A more sophisticated analysis is done by computing the eigen values and eigen vectors of the covariance matrix or alternatively computing the singular values and the accompanying matrices obtained from a singular value decomposition of the covariance matrix. Each eigen vector is a linear combination of the parameters such that the eigen values are orthogonal, which means mutually independent. Each eigen vector can thus be regarded a new parameter, i.e. a combination from the user chosen parameters, but which is independent of the other eigen values or new parameters. The accompanying eigen values are inversely proportional to the variance of the eigen vectors and this proportional to the importance of the eigen vectors. This allows judging how many eigen vectors or how many independent parameters actually drive this model calibration, and, for that matter, how many parameters may be reasonably calibrated with the given data in the given model. The eigen vectors allows verifying to what extent parameters are truly correlated and which parameter if any dominates the so-manieth eigen vector.

A similar analysis is possible with the singular value decomposition. The latter seems to be more robust under certain circumstances. The type of analyses described briefly here, are standard output of the mentioned dedicated calibration tools and software.

8.6.2 Making use of the output of lsqnonlin

In this sub section we show the statistical analysis that was presented theoretically above. We do this for the output of the simple example that was described above. To make sure we readily obtain the information necessary for the analysis, we called lsqnonlin function with all its possible output parameters.

The third output pa meter was called e; it is the difference vector, i.e. model output minus measurements. The second output, i.e. the *resnorm* is simply $e^T e$. Notice that the error variance equals

$$var = \frac{e^T e}{N - M}$$

where N is the number of measurements and M the number of parameters. We compute that ourselves in the Matlab script.

The outputs *exitflag*, *OUTPUT* and *LAMBDA* give information about the calibration process and the error bounds if applicable.

The last output , J, is the *Jacobian*, i.e. the sensitivities of the errors with respect to the parameters. In this case we have 22 measurements (*length eMeas*) and 2 parameters. Therefore, the *Jacobian* is a 22x2 matrix. It is output as a *sparse matrix*, which can be turned into a normal *full matrix* by the command *J=full(J)*.

The covariance matrix of the parameters, the standard deviation of the parameters, the correlation matrix of the parameters and the eigen vectors and eigen values are subsequently computed in Matlab using these data as shown in the script below. Some of the lines in the script have no ending semicolon so that the outputs are immediately printed.

```
% Show statistical results
varE   = resnorm/(size(eMeas,1)-size(p,1)) % error variance
stdE   = sqrt(varE)                 % error std deviation
covP   = varE * [J'*J]^(-1)         % parameter covariance matrix
sigmaP = sqrt(diag(covP));          % parameter stddev
corP   = covP ./ (sigmaP * sigmaP') % parameter correlation matrix
[EVec,EVal]  = eig(covP)            % eigen vectors, eigen values
```

Sample output follows in the next block. Notice that it (will be a little different after each run due to random errors that are generated when calling *eMeasNew*.

```
varE =
    0.0020342
stdE =
     0.045103
covP =
    0.0010911   -0.0028038
   -0.0028038    0.0090719
corP =
            1     -0.89119
     -0.89119            1
EVec =
     -0.95347     -0.30148
     -0.30148      0.95347
EVal =
   0.00020454            0
            0    0.0099584
```

What is evident from this statistical output is:

- standard deviation of the errors is about 4.5 cm.
- The correlation between the optimized parameters (T and S) is high, 89%.
- Eigen vector 1 is dominated by parameter 1 (transmissivity).
- Eigen vector 2 is dominated by parameter 2 (storativity).
- The second eigen value is about 50 times larger than the first, and therefore far less important for the calibration. This is caused by the fact that the sensitivity of the the error for the storativity is much less than that for the sensitivity of the transmissivity.

8.7 Listing of modelScript5 (simple calibration)

```
%% modelScript 5 Simple calibration
%  Demo of a simple calibration using lsqnonlin in Matlab
%
% We simulate an artificial pumping test to be analyzes using the Theis
% drawdown formula. Drawdowns are generated using some unknown values
% transmissivity T and storativity S in the eMeasNew function and adding
% some random measurment errors.
% Then the calibration starts with iniital values T0 and S0. The
% calibration optimizes the multiplyers p. When done the data are plotted
% with markers and the theis formulat is plotted using the original values
% for the transmissivity (green line) and the final ones (thick black).
%
% TO 140419

global eMeas Q T0 S0 r

%% Fixed data
Q  = 1200;  % [m3/d]
r = 30;     % [ m  ]

%% Initial trial parameters
T0 = 350;   % [m2/d]
```

```
S0 = 0.001; % [ -  ]

%% New data for this example
eMeasNew(Q,r,T0,S0);

%% Load data
load eMeas; % eMeas(:,1) is time and eMeas(:,2) are drawdowns

%% Initial parameter mutiplyers, first for T, second for S
p = ones(2,1);

%% Final parameter multiplyers after calibration
[p,resnorm,e,exitFlag,OUTPUT,LAMBDA,J] = lsqnonlin(@FUN,p);

%% Final parameters:
T = exp(p(1))*T0;
S = exp(p(2))*S0;

%% Suitable time vector to simulate model
t =logspace(log10(eMeas(1)),log10(eMeas(end)),30);

%% Model in shape of anonymous function forTheis drawdown
Theis = @(t,T,S) Q/(4*pi*T) * expint(r^2*S./(4*T*t));

%% Visualize output

%% Default axis parameters
defaults={'xScale','log','yScale','lin', ...
    'xGrid','on','yGrid','on','xLim',[t(1)/2 2*t(end)]};

%% Figure setup
figure;  axes('nextplot','add',defaults{:});
xlabel('time [min]'); ylabel('drawdown [m]');
title('Theis pumping test calibrated');

% Plot
plot(eMeas(:,1),eMeas(:,2),'ro');  % measurements

plot(t, Theis(t,T0,S0),'g','lineWidth',1); % model with initial paramters
plot(t, Theis(t,T, S ),'k','lineWidth',2); % model with final parameters

legend('measured','model begin','model optimized');

%% Statistical analysis

% Analysis using the ouputs of lsqnonlin
% the outputs are
% p (pameters), var (error variance), e (errors) ... J (jacobian, parameter
% senitivities).

J = full(J); % Jacobian is sparse, make it full

% Show statistical results
```

```
varE   = resnorm/(size(eMeas,1)-size(p,1)) % error variance (from lsqnonlin) = e'*e
stdE   = sqrt(varE)                 % error std deviation
covP   = varE * [J'*J]^(-1)         % parameter covariance matrix
sigmaP = sqrt(diag(covP));          % parameter stddev (approximate, linear)
corP   = covP ./ (sigmaP * sigmaP') % parameter correlation matrix
[EVec,EVal]  = eig(covP)            % EVec (eigen vectors) Eval (eigen values)
```

Bibliography

[Bear (1972)] Bear, J. (1972) Dynamics of Fluids in Porous Media. Dover ISBN 0-486-65675-6, 762p

[Bruggeman (1999)] Bruggeman, G.A. (1999) Analytical solutions of geohydrological problems. Elsevier. ISBN 0-444-81829-4, 999p

[Harbough (2005)] Harbaugh, A.W. (2005) MODFLOW-2005, The US Geological Survey Modular Ground Water Model - the Ground Water Flow Process. USGS, Techniques and Method 6-A16. ca 200p.

[McDonald and Harbaugh (1988)] McDonald, M.G. & A.W. Harbaugh (1988) A Modular three-dimensional finite-difference ground-water flow model. Series: Techniques of Water-Resources Investigations on the United States Geological Survey. Book 6, Chapter A1.

[Carslaw and Jaeger (1959)] Carslaw HS and Jaeger JC (1959) Conduction of heat in solids. Second Edition. Oxford Science Publications. Reprint 1990. 510p. ISBN 0-19-8533368-3.

[Carslaw and Jaeger (1959)] Konikow, L.F. (2010) Applying dispersive changes to Lagrangian particles in Groundwater Transport Models. Transport in Porous Media Vol. 85, pp437-449.

[Fitts (2002)] Fitts, C. (2002) Groundwater Science. Academic Press, ISBN 0-12-257855-4.

[The Mathworks (2008)] The Mathworks (2006) Matlab reference.

[Kruseman and de Ridder (1994)] Kruseman G.P. and N.A. De Ridder (1994) Analysis and Evaluarion of Pumping Test Data. Second Edition (Completely Revised) ILRI Publication 47, Wageningen, The Netherlands. 370pp. ISBN 90-70754-207

[Olsthoorn (2014)] Olsthoorn (2008-2014) mfLab, www.code.Google.com/p/mfLab

[Olsthoorn (1998)] Olsthoorn (1998) Groundwater modeling: Calibration and the use of Spreadsheets. PhD thesis, TU Delft, ISBN 90-407-1702-8.

[Pollock (2012)] Pollock, D.W. (2012) User Guide for MODPATH Version 6 - A Particle Tracking Model for Modflow. USGS Techniques and Methods 6-A1.

[Strack (1989)] Strack, O.D.L. (1989) Groundwater Mechanics. Prentice Hall. 0-13-365412-5, 731pp (new edition, 2013)

[Verruijt (1982)] Veeruijt, A. (1982) Groundwater Flow, Second Edition. ISBN 0-333-32958 9, 143pp.

[Zheng (1993)] Zheng, C. (1993) Extension of the Method of Characteristics for Simulation of Solute Transport in Three Dimensions. Groundwater, Vol 23, Nr. 3, May-June 1993, p46-465