ERDC Submission

1 Introduction

This repository is a collection of functions built over the course of the GEO 425P class on Modeling Flow and Transport in Fluid Media. Additional scripts utlizing these functions are included. The README will start by examining two scripts and the reasoning behind some of the choices made while skimming over the functions called. The remaining documentation will briefly describe each function, its importance, and then go into further detail for specific lines of code. As each function comes with a header, the variable definitions amongst other details will be skipped, and only some implementation and underlying motivations will be discussed. Note that these problems have been done with the assumption of saturated pores with no compaction or reactions.

2 Walkthrough

2.1 bv4987_TopoCell.m

2.2 bv4987_GWage1D.m

2.3 build_grid.m

For a majority of this course, we used finite difference methods and implemented several functions to solve specific 1D and 2D flow problems. The first function was build_grid.m. It constructs our computational domain using a staggered grid approach with our hydraulic head specified at the cell centers, and our volumetric flux at the cell faces.

Lines 51 - 85:

These lines are set aside for 1D grids, in our class when dealing with one dimension it was assumed along the x-direction.

Lines 88 - 91:

These fields determined how a later function, comp_streamfun.m, would compute a given streamfunction.

Lines 100 - 106:

Below is an instance of a 2D Grid:

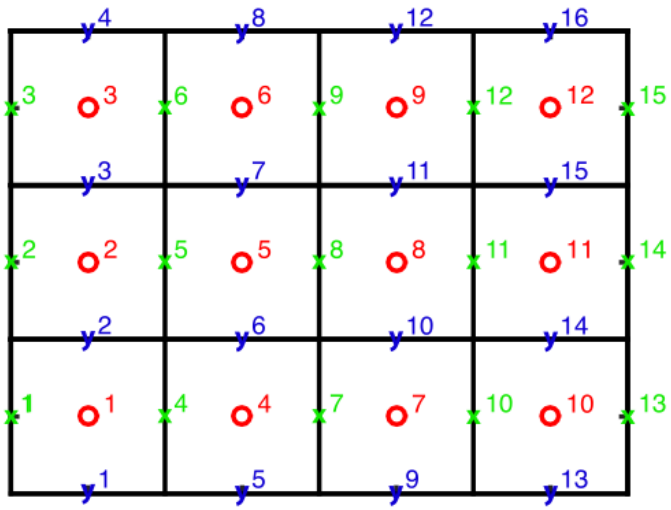

Figure 1: Example 2D Grid Visualization

It can be seen from Figure 1, that the number of x faces is equal to the cell rows times the face columns, and the y faces equal the face rows times the cell columns.

Lines 116 - 118:

Each respective cell and face's degrees of freedom were assembled into a grid matching their physical representation, to allow for straightforward implementation of the boundaries.

Lines 129 - 130:

Many of the 2D implementations of the operators and fluxes and heads have the y values "stacked" above the x values, or "x-first". To simulate this the y face degrees of freedom must have the number of x faces added to it to serve as a new baseline.

2.4 build_ops.m

build_ops.m discretizes our divergence and gradient operators whose size depends on the grid specifications. Directly discretizing our div-grad operators helps reduce errors by ensuring we conserve energy. The purpose of G is to take us from hydraulic head on the cell centers to fluxes on the cell faces, hence why it has dimensions' N by Nf. D, on the other hand, takes us from the fluxes on the cell faces to the sources back on the cell centers, and is why it is Nf by N.

Line 25:

Making use of the adjoint relation between div and grad, we can construct G much easier and faster.

Line 26:

We implement natural boundary conditions onto G by default. If a nonhomogeneous Neumann condition arises, it is implemented using residuals. See comp_flux.m for more information.

Lines 31 - 40:

From inspection of the divergence matrices for a simple 2D grid, it can be shown that the columns don't "communicate with

each other" when dealing with y fluxes. This is due to the linear numbering as seen in Figure 1. The result for D_y is a block diagonal matrix of the 1D divergence matrix case repeated across the main diagonal. For the x fluxes, it is a bit more complicated. From Figure 1 when looking at any row of cells the x faces jump in values equal to the number of rows. Regular grids like Figure 1 with simple and consistent jumps in the dof result in a D_x where the two diagonals in the difference matrix are offset by the number of rows in the grid. This results in a relatively "nice" overall block diagonal matrix. MATLAB's kron function allows us to quickly assemble the x and y component from their corresponding 1D D and I matrices.

2.5 build_bnd.m

build_bnd.m uses the Param and Grid structures to create a constraint and projection matrix, B and N respectively, using I as the standard basis for the full solution space of the hydraulic head, h. It additionally makes an effective source term from the Neumann boundary conditions.

Lines 40 - 41:

Taking advantage of the fact that our computational grid can't tell the difference between what is a flux or a source term. We treat the influx of fluid through any boundary specified with Neumann conditions as a flux term. Here the source is specified by Param.qb.

Lines 44 - 47:

Dirichlet Boundary Conditions are expressed below by:

$$Bh = g \tag{1}$$

Where B is the constraint matrix, h is the hydraulic head, and g is the set of Dirichlet conditions. Since I serves as the standard basis for the full solution space of h, we can construct B by pulling the rows of I that correspond to grid boundaries with prescribed heads. To make N a proper projection matrix, (see solve_lbvp), we take it to be the nullspace of B. The quickest way to do so is to again pull from I, this time removing columns associated with constrained boundaries. This effectively makes N the standard basis for the reduced solution space of h.

2.6 solve_lbvp.m

Despite the output name, solve_lbvp.m is a somewhat general linear boudary value problem solver. As long as the proper inputs are taken, it solves a differential equation in its reduced space.

Line 44 - 46:

2.7 comp_flux.m

comp_flux.m computes the flux in the interior points through Darcy's Law, reconstructs the boundary fluxes through residuals and determines their direction based on information of the boundaries' degrees of freedom.

Line 37:

From a simplified Darcy's Law given by:

$$q = -K\nabla h \tag{2}$$

From eqn (2), can compute the internal fluxes.

Line 45:

Lines 48 - 52:

2.8 comp_mean.m

2.9 comp_streamfun.m

2.10 plot_flownet.m

2.11 flux_upwind.m

2.12 age_ode.m