THERMAID

User Quick Start Guide v.1.01

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

THERMAID is based on Maflot, developed by Rouven Künze and Ivan Lunati at the University of Lausanne, 2012. The initial code basis was provided through their code available at www.maflot.com.

I thank Prof. Stephen. A. Miller for his support and for sharing his pearls of wisdom with me during the creation of the Thermaid code. I would also like to show my gratitude to Reza Sohrabi without whom the code and documentation would not exist in this form.

This project was supported by the Swiss National Fond (SNF) through grant 'NFP70:Energy Turnaround (153971)'.

COPYRIGHT

LICENSE

INTRODUCTION

A large fraction of the world's water and energy resources are located in naturally fractured reservoirs within the earth's crust. Understanding the dynamics of such reservoirs in terms of flow, heat transport and fracture stability is crucial to successful application of engineered geothermal systems (also known as enhanced geothermal systems, EGS) for geothermal energy production in the future.

Fractured reservoirs can be considered to consist of two distinct separate media, namely the fracture and matrix space respectively. Depending on the properties of matrix and fracture, different types of reservoirs can be defined. In case of enhanced geothermal systems, typically two cases may prevail: 1) reservoirs with low porosity matrix for which both the permeability and the storage capacity of the rock mass are controlled by the fractures and 2) reservoir with sufficient matrix porosity such that fluid storage is dominated by the matrix while the fractures contain only a small fraction of the fluid but control the permeability.

Simulation of flow and transport through fractured porous media is challenging due to the high permeability contrast between the fractures and the surrounding rock matrix. However, accurate and efficient simulation of flow through a fracture network is crucial in order to understand, optimize and engineer reservoirs. It has been a research topic for several decades and is still under active research. Additionally accurate estimations of the fracture stability are necessary in order to predict permeability evolution and forecast induced seismicity. Discrete fracture models (DFM) have been developed to address the computational problem of scales for fluid flow and heat transport. Yet traditional conforming DFM where the fractures are explicitly resolved by the numerical grid, suffer from computationally expensive pre-processing in the numerical grid generation and can encounter severe time step restrictions during the simulation if explicit time-stepping and small cells around the fractures are used.

An alternative approach is used by the embedded discrete fracture models (EDFM), which treat fracture and matrix in two separate computational domains. The embedded fracture model was first introduced by Lee et al. for single phase problems and later extended to twophase flow. The embedded discrete fracture model is a promising technique in modelling the behavior of enhanced geothermal systems.

Slip tendency analysis is used in order to estimate fault reactivation potential in earthquake prone areas as well as fracture stability in geothermal reservoirs. Slip tendency is the ratio of shear stress to effective normal stress on a surface. Fracture or fault reactivation is likely to occur if the shear stress to effective normal stress ratio equals or exceeds the frictional sliding resistance. In general, the stress field is heterogeneous due to the contrast of mechanical properties and past rock mass rupture. Moreover, the stability of the fractures is controlled by the local stress field. However, assessing this local stress field is not trivial. On the other hand it may not be necessary to determine the local stress field and resolve the complex deformation process during fault slip in order to obtain an indicator for the likelihood of slip. Using slip tendency, predictions on fracture instabilities during the hydraulic stimulation of a fractured reservoir are feasible without solving for the typically non-linear evolution of the stress equilibrium equation.

Thermaid is an open source implementation of an embedded discrete fracture model for single phase flow and heat transport with additional capabilities to determine fracture stability in fractured reservoirs. Thermaid, is a fractured reservoir modelling framework implemented in MATLAB which can be used as a standalone simulation package for TH(m) cases in geothermal reservoirs or as a blue print for the re-implementation of the method e.g. in a high performance computing (HPC) framework. Of course, additional model capabilities can be implemented by new users directly in the Thermaid package.

THEORY OF THE EMBEDDED DISCRETE FRACTURE MODEL

The conceptual idea of the EDFM is the distinct separation of a fractured reservoir into a fracture and a matrix domain. We introduce a transfer function to account for coupling effects between the two domains (cf. Figure 1), so the fracture and matrix domains are computationally independent except for the transfer function. As the fractures are generally very thin and highly permeable compared to the surrounding matrix rock, the gradient of fracture pressure normal to the fracture is negligible. This allows for a lower dimensional representation of fractures (i.e. 1D objects within a 2D reservoir).

For more detail on the underlying theory implemented in Thermaid please refer to Thermaid's scientific publication.

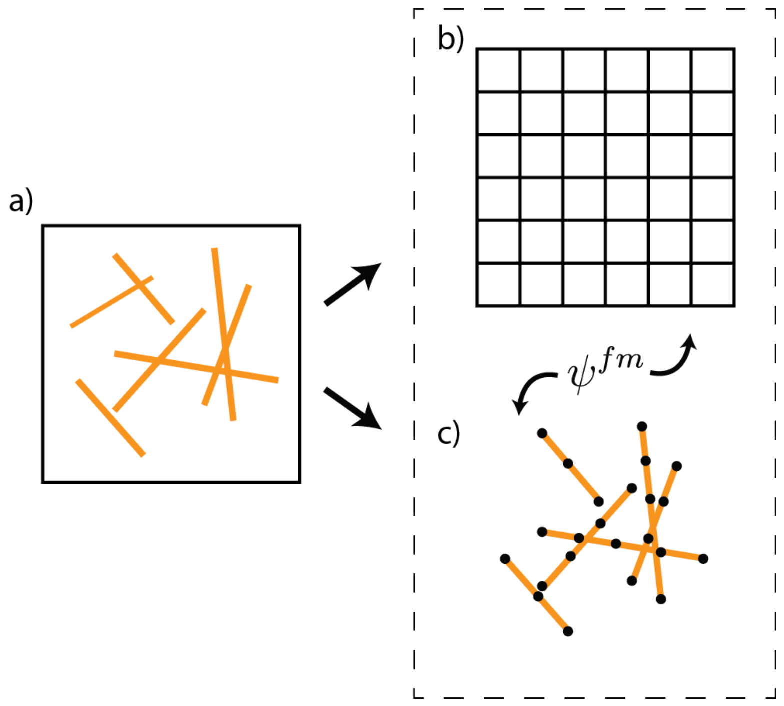


Figure 1: A fractured domain a) is separated in a uniform grid b) and a fracture grid c). The two resulting domains are coupled using the transfer functionΨ_{fm}.

QUICK START

You can get started running the Thermaid code right away. Open MATLAB, change into the Thermaid directory and run the startup script:

```
>> cd PATH_TO_/THERMAID
>> startup
```

The startup script will add the necessary directories to the search path and execute a small sanity test. If your MATLAB version gives you an error message at this point, please note that Thermaid has been developed and tested with MATLAB R2015b. Please contact the authors for support.

After the startup script has successfully finished, you can run any of the provided examples by typing the following into your command window:

```
>> ex1
```

This will run the first of the examples and show the results in a plot. Please see the detailed examples description section for more information on what problems the examples solve and how to visualize your results.

Beware, that this code is scientific software that should not be used for mindless simulations. Always look at your simulations results critically and evaluate if the computed results are physically making sense.

We encourage you to take the time to carefully read through this manual after you played around a little with the examples, so that you can understand more in depth, how the code and its underlying physical model works.

With this being said, let’s get started. In the following sections the basic elements of the code are outlined. Then, the usage of the input file, the conventions of the boundary conditions are explained. Finally we provide some tips on visualizing your results and provide more details on the examples.

RUNNING THE CODE

Thermaid can be run from its directory by entering the command

```
>> THERMAID('InputFile')
```

Alternatively, you can access Thermaid form any directory by adding the Thermaid directory to your current path:

```
>> addpath(genpath('/path/to/THERMAID')) TODO??
```

When calling Thermaid an input file must be given. You can edit the existing input file, or (preferably) copy and edit it according to your needs. Afterwards you can execute it by calling

```
>> THERMAID('My_InputFile')
```

The input file is more or less self-explanatory and can be modified according to your needs. Nevertheless, the contents of the input file are explained throughout this guide.

As an additional option to the Thermaid call, the default graphical output of the results can be suppressed. In order to do this the call has to be modified to

```
>> THERMAID('My_InputFile',0)
```

This might be useful, if timed simulations are desired as the graphical output takes a big part of the simulation time. You can still plot the final results afterwards as the data is written to the workspace.

ELEMENTS OF THE CODE

Thermaid consists of multiple functions. `THERMAID.m` itself is the main function that controls the simulation and calls most of the sub functions. The sub functions deal with specific parts of the program, such as initialization, discretization and solution of the flow and transport equations. The individual functions are introduced briefly here while the underlying theory is further explained throughout the scientific publication.

THERMAID.m

The main function. All sub functions are called out of this main function. It contains the time loop and is responsible for general functionalities of the code. More insight to the main functions structure is given in the scientific publication.

InputFile.m

This is the user’s control file for the simulation. It describes to discrete fracture network as well as the problems parameters and boundary conditions. The input file is self-explanatory but a short explanation is given in section *Input File.*

startup.m

The startup routine of Thermaid. During the execution the Thermaid directories are added to the MATLAB environment path and a simple example to verify the build.

initialize.m

Initializes the interface based values such as permeability and porosity as well the gravity contributions to the pressure system.

calc_density.m

The density is temperature and pressure dependent. An equation of state is implemented in this function to calculate the correct value.

intersectionsGrid.m

Finds the intersections of the discrete fracture network with the matrix grid and calculates the connectivity index.

calc_d_mean.m

Calculates the mean distance of the fracture to the cell. See the scientific publication for more detail.

intersectionsSegments.m

Finds intersections between individual fractures and computes the corresponding transmissivities.

calc_frac_flux_mat.m

Calculate the matrix operator for flux extraction on the fractures. This enhances the computational performance of Thermaid.

calc_frac_grad_expand_mat.m

Calculate the matrix operator for expanded gradient operations on the fractures. This is used in the velocity calculations for instance.

calc_frac_grad_mat.m

Calculate the matrix operator for expanded gradient operations on the fractures.

calc_frac_mean_mat.m

Calculate the matrix operator for the mean on the fractures.

calc_interface_values.m

Calculate interface values for the matrix interfaces.

calc_interface_values_fracture.m

Calculate harmonic mean interface values for the fracture interfaces.

pressureSystem.m

Constructs the mass balance matrix and right hand side vector for the pressure system. The linear algebraic system is solver outside in the main function.

calcVelocity.m

Calculates the velocity field from the pressure using Darcy's law.

transport_heat_System.m

Solves the transport problem based on the mass continuity equation of the solvent. The system is solved within the function.

transport_heat_Advection.m

Constructs the advection matrix using a first order upwind approximation.

transport_heat_Diffusion.m

Constructs the diffusion operator for the transport equation.

calc_frac_stability.m

Calculate fracture stability based on slip tendency. Refer to the scientific publication for more detailed background information.

attach_pre_timestep.m

This function allows user-defined code to be executed before each time step. This can be used to load external data into the simulation or to interface with other software.

attach_post_timestep.m

This function allows user-defined code to be execute after each time step. This can be used for visualization for instance.

INPUT FILE

The input file is more or less self-explanatory and can be modified according to your needs. It is recommended that you create a renamed copy of the input file before you modify it.

All user input data is saved in a structure called `udata` which is used to facilitate data transfer within Thermaid to all the relevant positions. Units are always given in brackets. The form of the Input file respectively all variables shown here must be preserved to avoid errors.

The first section of the input file defines the global dimensions of the domain and its discretization:

```
%% GRID PARAMETERS ------------------------------------------------------------%
udata.len = [500 500]; % physical length of the domain in x and y direction [m]
udata.Nf = [51 51]; % number of cells in x and y direction
udata.dx = udata.len./udata.Nf; % cell length [m]
```

Next, the simulation parameters corresponding to the time of the simulation are set.

```
%% SIMULATION PARAMETER FOR TRANSIENT SIMULATIONS------------------------------%
udata.timeSim = 86400; % total simulation time [s]
udata.dt = 3600; % time step length [s]
udata.tol = 1.e-4; % tolerance in pressure-heat loop [-]
udata.maxit = 100; % maximum number of pressure-heat loops to converge
```

The fracture network is set by calling one of the existing DFN or load a DFN from file. Please look at examples 4 and 5 to learn more about reading DFNs from file.

```
%% FRACTURE NETWORK -----------------------------------------------------------%
% Thirteen 'random' fractures
frac_complex_n13;
if (udata.dxf < min(udata.dx))
      error('dxf < dx')
end
```

Note that an error check is included in the input file to make sure that the discretization length of the fracture is bigger than the minimum cell spacing. Now, that the fracture network and the most general parameters for the simulation have been defined, we define the initial conditions:

```
%% INITIAL CONDITIONS-----------------------------------------------------------%
udata.T0 = zeros(udata.Nf(1),udata.Nf(2)); % Initial matrix temperature [°C]
udata.T0f = zeros(udata.Nf_f,1); % Initial fracture temperature [°C]
udata.tmax = 0; % maximum temperature [°C] for plotting
udata.p0 = zeros(udata.Nf(1),udata.Nf(2)); % Initial matrix pressure [Pa]
udata.p0f = zeros(udata.Nf_f,1); % Initial fracture pressure [Pa]
```

Setting boundary conditions is fairly simple in Thermaid, however it might take some time to get used to it. Following this section there is a section to clarify the conventions used in boundary conditions. The general structure looks like this:

```
%% BC FLUID ------------------------------------------------------------------%
udata.ibcs = zeros(2*sum(udata.Nf),1); % type 0:Neumann(N); 1:Dirichlet(D)
udata.Fix = zeros(2*sum(udata.Nf),1); % value N [m2/s] (inflow>0); D [Pa]

udata.ibcs(1:udata.Nf(2)) = 1;
udata.ibcs(udata.Nf(2)+1:2*udata.Nf(2))=1;
udata.Fix(1:udata.Nf(2)) = 5e6;
udata.Fix(udata.Nf(2)+1:2*udata.Nf(2))=0;

%% BC TRANSPORT --------------------------------------------------------------%
udata.flagHeatTransport = 0;
udata.FixT = zeros(2*sum(udata.Nf),1);%normalized concentration of boundary flow
[-]
```

Note that heat transport can be turned on and off using the user data flag `udata.flagHeatTransport`. If the value is set to 0, then a pressure-only simulation is executed.

You can add internal sources by using the source terms:

```
%% SOURCE TERMS --------------------------------------------------------------%
Q = zeros(udata.Nf); % source term [m2/s]; inflow positive
QT = zeros(udata.Nf); % normalized concentration for source term [-]
```

Next, the hydraulic parameters of the matrix and fractures have to be defined. Note that we define also whether or not to account for gravity here:

```
%% GRAVITY---------------------------------------------------------------------%
udata.gravity = 0; % gravity acceleration in y [m/s2]

%% PERMEABILITY -----------------------------------------------------------------%
udata.K = ones(udata.Nf(1),udata.Nf(2))*1e-9; % permeability field [m2]
udata.K_f = ones(udata.Nf_f,1)*1e-5; % fracture permeability field [m2]

%% Fracture aperture ------------------------------------------------------------%
udata.b0 = sqrt(12.*udata.K_f).*ones(udata.Nf_f,1); % fracture aperture field
[m]

%% Porosity ---------------------------------------------------------------------%
udata.phi = ones(udata.Nf(1),udata.Nf(2))*0.3; % porosity field
udata.phi_f = ones(udata.Nf_f,1)*0.3; % fracture porosity field
```

Fluid density and viscosity can either be calculated using an equation of state, or held constant at a value specified in the input file:

```
%% Fluid density -------------------------------------------------------------%
udata.const_density = 1;
if(udata.const_density)
       udata.density_l = 1000*ones(udata.Nf); % density of the rock [kg/m3]
       udata.density_lf = 1000*ones(udata.Nf_f,1); % density of the rock [kg/m3]
end

%% Fluid viscosity -----------------------------------------------------------%
udata.const_viscosity = 1;
if(udata.const_viscosity)
       udata.viscosity = 1e-3*ones(udata.Nf); % density of the rock [kg/m3]
       udata.viscosity_f = 1e-3*ones(udata.Nf_f,1); % density of the rock
[kg/m3]
end
```

The following parameters are related to mechanic capabilities of Thermaid:

```
%% MECHANIC PROPERTIES---------------------------------------------------------%
udata.flagIncompressible = 0; % If 1 - incompressible fluids are used
udata.compressibility_l = 5e-10; % Bulk Modulus of the fluid [Pa]
udata.compressibility_s = 5e-10; % Bulk Modulus of the rock [Pa]

udata.shear_modulus = 29e9; % Shear modulus of the rock [Pa]
udata.poisson_ratio = 0.25; % Poisson ratio of the rock [-]

udata.friction_coeff = 0.6; % Friction coefficient (shear failure) [-]
udata.K_enh = 1e2; % Permeability enhancement factor [-]

udata.therm_exp_coeff = 7.9e-6;%Thermal expansion coefficient of the rock matrix
[-]

udata.flagFracStability = 0; % If 0 the remaining parameters in this section are
not used

udata.sigma_1 = 20e6*ones(udata.Nf_f,1); % Maximum principal stress [Pa]
udata.sigma_1 = 20e6*ones(udata.Nf_f,1); % Maximum principal stress [Pa]
udata.sigma_1 = 20e6*ones(udata.Nf_f,1); % Maximum principal stress [Pa]

udata.stress_trend = [0 90]; % [TR(S1) TR(S3)]
udata.stress_plunge = 90; % PL(S1)

udata.frac_az = 90*ones(size(udata.frac_angle)); % Fracture dip [°]
udata.frac_dip = min(abs(udata.frac_angle),180-abs(udata.frac_angle)); %
Fracture azimuth (from N) [°]
udata.use_thermal_stress = 0; % Boolean for thermal stress in calculations
```

Finally we have to define the rock density and thermal properties:

```
%% ROCK DENSITY --------------------------------------------------------------%
udata.density_s = 2500*ones(udata.Nf); % density of the rock [kg/m3]
udata.density_sf = 2500*ones(udata.Nf_f,1); % density of the rock [kg/m3]

%% THERMAL DIFFUSION ---------------------------------------------------------%
udata.lambda_l = 0.5; % Thermal conductivity of the fluid [W/(m*K)]
udata.lambda_s = 2.0; % Thermal conductivity of the rock [W/(m*K)]
udata.cp_l = 4000; % Specific heat capacity of the fluid [J/(kg*K)]
udata.cp_s = 1000; % Specific heat capacity of the rock [J/(kg*K)]
udata.ibcD = zeros(2*sum(udata.Nf),1); % 1 -> Diffusion on boundary cells
```

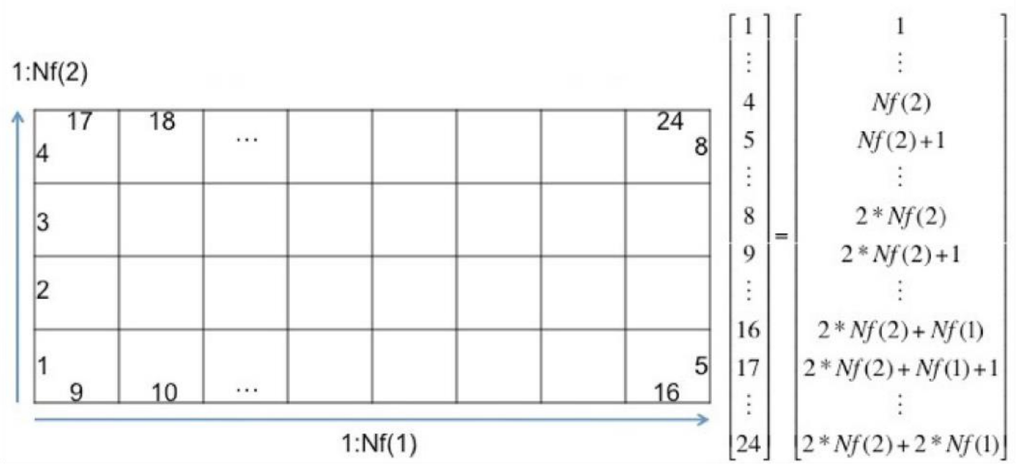
With this, all necessary input parameters are defined and can be used in a Thermaid simulation.

BOUNDARY CONDITIONS

The boundary conditions follow a convention that is explained here in detail. Note, that for the pressure equation two types of boundary conditions (Dirichlet and Neumann) can be used. The transport equation can only be constrained by Dirichlet boundary conditions as the flux is prescribed by the pressure equation.

> **Dirichlet**: The value at the boundary should fixed. I.e. on the left side boundary 1MPa pressure is applied. Another example would be the tracer amount at the top is 1kg/m^3.
>
> **Neumann**: The flux at the boundary is fixed. I.e a constant flux of 10m^3/s enters at the bottom of the domain.

The input file uses a vector formulation to assign the boundary condition type and value. Thus, two vectors $ibcs$ and Fix with dimensions $2 \cdot [Nf(2) + Nf(1)]$ where $Nf(1)$ and $Nf(2)$ are the cells in x and y directions respectively, are used.



$$\begin{bmatrix} 1 \\ \vdots \\ 4 \\ 5 \\ \vdots \\ 8 \\ 9 \\ \vdots \\ 16 \\ 17 \\ \vdots \\ 24 \end{bmatrix} = \begin{bmatrix} 1 \\ \vdots \\ Nf(2) \\ Nf(2)+1 \\ \vdots \\ 2*Nf(2) \\ 2*Nf(2)+1 \\ \vdots \\ 2*Nf(2)+Nf(1) \\ 2*Nf(2)+Nf(1)+1 \\ \vdots \\ 2*Nf(2)+2*Nf(1) \end{bmatrix}$$

Figure 2: Visualization of the boundary condition conventions used in Thermaid. This image is directly taken from the Maflot manual.

The ordering of the vector entries is shown in the example in Figure 2. For convenience the index sets for the four boundaries are:

```
Left:          1 : Nf(2)
Right:         Nf(2)+1 : 2*Nf(2)
Bottom:        2*Nf(2)+1 : 2*Nf(2)+Nf(1)
Top:           2*Nf(2)+Nf(1)+1 : 2*Nf(2)+2*Nf(1)
```

The vector $ibcs$ defines the type of the boundary:

```
Neumann:      0
Dirichlet:    1
```

The vector Fix (or $FixC$ for the transport solution) defines the value to the defined type.

EXAMPLES

Thermaid is distributed with six examples that can directly be used to understand the method and used as a basis for new simulations. The scripts to run all examples are provided with the Thermaid source code. They can be found in the `/examples` subdirectory and can be used directly from within MATLAB by the following commands

```
>> ex1
```

The above command runs the first of five examples. The other examples can be called in the same fashion. The following examples are included:

- *Example 1 : Crossed fractures*
  A simple test case of two perpendicular fractures interacting in an external pressure field
- *Example 2: Crossed fractures with heat transport*
  A simple test case of two perpendicular fractures interacting to an external pressure field including heat transport
- *Example 3 : Complex fracture network with 13 fractures*
  A complex fracture network test case on a larger scale
- *Example 4 : Load a FracSim3D fracture network*
  Load and use a DFN loaded from a FracSim3D file
- *Example 5 : Load a FracMan fracture network*
  Load and use a DFN loaded from a FracMan file
- *Example 6 : Single inclined fracture*
  Steady-state pressure field surrounding an inclined fracture

EXAMPLE 1: CROSSED FRACTURES

In order to validate the implementation of flow equations of the model, we use a benchmark experiment first presented by (Hajibeygi et al., 2011) is used. The same test case is also used by (Pluimers, 2015) to study the sensitivity of the EDFM with respect to numerical resolution and fracture position within the intersecting grid cells. Figure 3 shows the benchmark geometry that consists of two perpendicular 5m-long fractures intersecting in the middle of a square domain. The aperture of the fractures is fixed at $b = 4mm$. The domain is 9m by 9m square domain with Dirichlet boundary conditions on the left and right sides. On the left, a constant pressure of 1Pa is applied, whereas the right side is fixed to 0Pa. On the top and bottom a no-flow Neumann boundary is applied. The matrix domain is discretized by 301x301 cells while the fractures are modelled by 304 fracture segments (152 each). As fracture permeability and permeability contrast between fracture and matrix are some of the most important variables to be determined in EGS reservoirs to accurately predict flow in the reservoir, we evaluate our method with two different permeability contrast values, namely $\frac{k^f}{k^m} = 10^3$ and $\frac{k^f}{k^m} = 10^5$. We compare the solution at steady state obtained by THERMAID to a reference solution computed by *COMSOL Multiphysics*, which is a widely used finite element package with subsurface flow and transport capabilities. The reference solution is computed on a conforming discrete fracture network were the matrix elements are aligned exactly on the grid with a total of 1'750'693 degrees of freedom (dof).

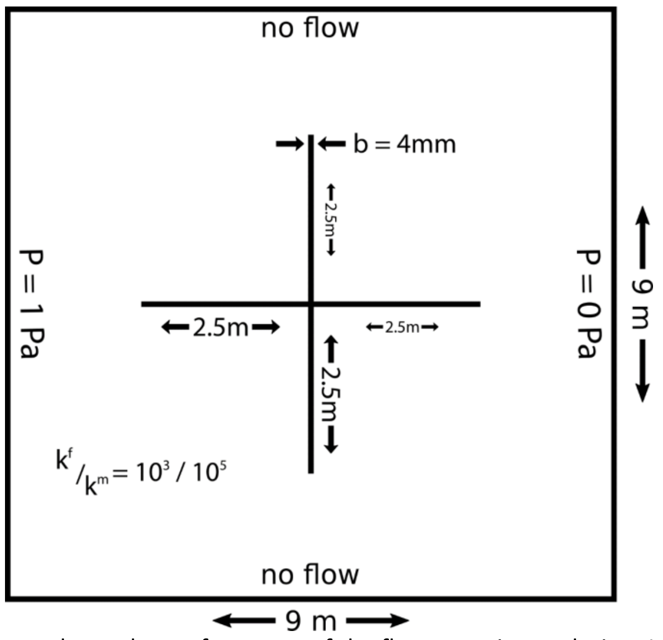


Figure 3 : Numerical setup to evaluate the performance of the flow equations solution. Incompressible fluids are assumed in this benchmark experiment. On the left boundary a constant pressure of 1Pa is assumed. The right boundary is set to 0Pa. On the other boundaries a no-flow boundary condition is applied.

The fracture pressure shown in Figure 4 shows a very good match between the reference and the implemented model. However, if the results are compared in detail, a slight offset between the reference solution and the current implementation becomes visible. We quantify the difference between our model with the reference by the 'normalized root mean squared error (NRSME)' as well as the 'normalized mean absolute error (NMAE)'.

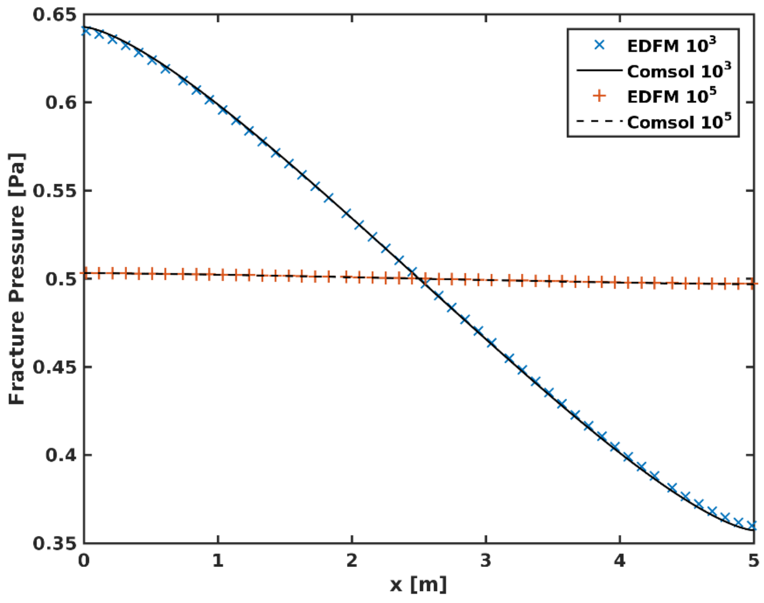


Figure 4 : Steady-state fracture pressure through the horizontal fracture. For both permeability contrasts we see very good agreement between the implemented method and the reference solution.

We observe errors of well below 2% compared to the pressure range in the benchmark setup. Both NRSME and NMAE show slightly bigger fracture pressure errors for a permeability contrast of $\frac{k^f}{k^m} = 10^5$. Especially the fracture pressure at this permeability contrast shows, with a deviation of ~1.5%, the largest observed error in this benchmark. Figure 5 shows the matrix pressure at y = 4.5m, which corresponds to the matrix cells intersected by the horizontal fracture. Again, the reference case and the THERMAID solutions are in good agreement. Slight deviations from the reference solution ~0.5% are visible.

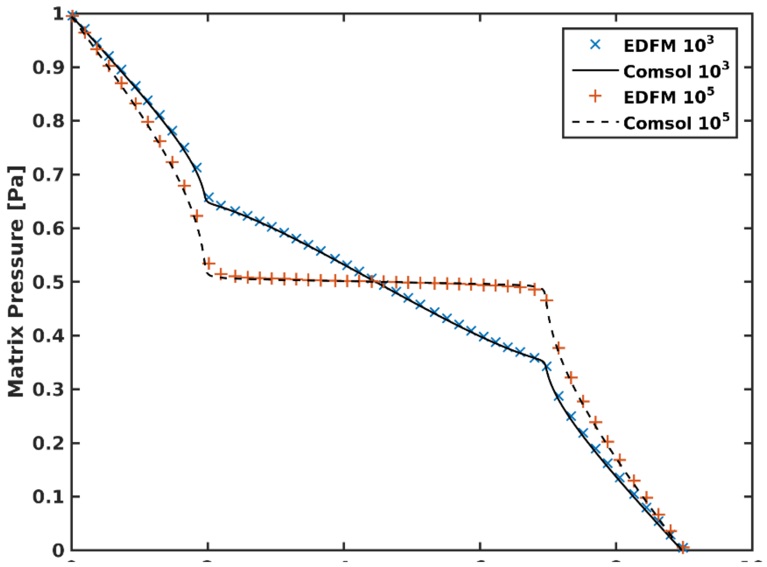


Figure 5: Steady state matrix pressure at y=4.5m - the position of the horizontal fracture. We evaluated two permeability contrast between matrix and fracture. In both cases we see very good consistency between the implemented method and the reference solution.

EXAMPLE 2: CROSSED FRACTURES WITH HEAT TRANSPORT

This example is used in the scientific publication. More information will follow. In the meantime please refer to the scientific publication.

EXAMPLE 3: COMPLEX FRACTURE NETWORK WITH 13 FRACTURES

This example is used in the scientific publication. More information will follow. In the meantime please refer to the scientific publication.

EXAMPLE 4: LOAD A FRACSIM3D FRACTURE NETWORK

More information will follow.

EXAMPLE 5: LOAD A FRACMAN FRACTURE NETWORK

More information will follow.

EXAMPLE 6: SINGLE INCLINED FRACTURE

This example is used in the scientific publication. More information will follow. In the meantime please refer to the scientific publication.

IMPLEMENTATION DETAILS

THERMAID is described in detail in its scientific publication. Due to the limited extend of most research publications, some information were left out and will be provided in this manual instead. Here we describe the discretization of the governing equations by the finite volume method and implementation of the two-dimensional embedded discrete fracture method in MATLAB. As briefly discussed in the introduction, the matrix and fracture domains are discretized by regular Cartesian grids in 2D for the matrix and 1D for the fractures respectively.

NUMERICAL DISCRETIZATION IN SPACE

Using a finite volume approach, we discretize the domain Ω as the integration over finite control volumes Ω_{ij} with $\Omega = \sum_{ij=1}^{N} \Omega_{ij}$. Using the Gauss theorem, the divergence integral over the volume can be rewritten as the surface integral normal to the boundary of the volume. Applied to a matrix grid cell on the right hand side (RHS) of the pressure equation this yields

$$-\int_{\Omega_{ij}} \nabla \cdot \left(\frac{k}{\mu} \cdot \nabla p\right)^{m} + \psi^{mf} + Q^{m}\, dV \Rightarrow -\int_{\partial\Omega_{ij}} (\left(\frac{k}{\mu} \cdot \nabla p\right)^{m} + \psi^{mf}) \cdot n\ \, ds + \int_{\Omega_{ij}} \mathrm{Q}^{\mathrm{m}}\, dV$$

Note that gravity is neglected here and in the remains of this section to better facilitate comprehension of the implementation. The pressure gradient over the cell boundary $\partial\Omega_{ij}$ is approximated by a two-point flux approximation that is similar to the central difference scheme of the finite difference method and is second-order accurate in space. As the domain is generally heterogeneous in terms of rock properties, a harmonic averaging technique is used to calculate the appropriate values at the cell boundaries. This leads to the discretized expression for the RHS of the matrix pressure

$$\begin{aligned}
&\frac{\Delta y \left(\frac{k}{\mu}\right)^{m}_{i-\frac{1}{2},j}}{\Delta x}(p^{m}_{i,j} - p^{m}_{i-1,j}) + \frac{\Delta y \left(\frac{k}{\mu}\right)^{m}_{i+\frac{1}{2},j}}{\Delta x}(p^{m}_{i,j} - p^{m}_{i+1,j}) \\
&+\frac{\Delta x \left(\frac{k}{\mu}\right)^{m}_{i,j-\frac{1}{2}}}{\Delta y}(p^{m}_{i,j} - p^{m}_{i,j-1}) + \frac{\Delta x \left(\frac{k}{\mu}\right)^{m}_{i,j+\frac{1}{2}}}{\Delta y}(p^{m}_{i,j} - p^{m}_{i,j+1}) \\
&= Q^{m}_{i,j} + \sum_{\Omega_{ij}\cap\Omega_k} CI_k \left(\frac{k}{\mu}\right)_{ij,k} (p^{f}_{k} - p^{m}_{i,j})
\end{aligned}$$

where Δx and Δy represent the lengths of the grid cells in x and y direction, respectively. All other notations are adopted from the elemental literature of the Finite Volume Method.

The RHS for the fracture pressure can be expressed using the same approach as

$$\frac{b\left(\frac{k}{\mu}\right)^{f}_{k-\frac{1}{2}}}{\Delta x^{f}}(p^{f}_{k}-p^{f}_{k-1})+\frac{b\left(\frac{k}{\mu}\right)^{f}_{k+\frac{1}{2}}}{\Delta x^{f}}(p^{f}_{k}-p^{f}_{k+1})$$
$$=Q^{f}_{k}-\sum_{\Omega_{ij}\cap\Omega_{k}} CI_{k}\left(\frac{k}{\mu}\right)_{ij,k}(p^{f}_{k}-p^{m}_{i,j})$$

Here, Δx^{f} is the discretization lengths of the fracture segments and b the aperture of the fracture.

The discretization of the RHS of the temperature equation is analogous to the pressure equations and omitted here for brevity. It is worth noting, however, that the advection term must be treated with special care. This is not a limitation of the EDFM but a consistent issue in a wide range of numerical methods. In this EDFM implementation, we use an upwind method in the fractures in combination with a *minmod*-flux limited QUICK scheme in the matrix.

CONNECTIVITY INDEX

The connectivity index CI between matrix and fracture is discretization-dependent, and defined based on the linear pressure distribution assumed within a grid cell intersected by a fracture (Hajibeygi et al., 2011). It is defined as the length fraction $A_{ij,k}$ of fracture segment k inside matrix cell ij divided by the average distance $\langle d_{ij,k}\rangle$ between matrix cell ij and fracture segment k.

$$CI_{ij,k}=\frac{A_{ij,k}}{\langle d\rangle_{ij,k}}$$

The average distance $\langle d_{ij,k}\rangle$ can be calculated as

$$\langle d\rangle_{ij,k}=\frac{\int x_{k}(x')dx'}{V_{ij}}$$

where x_k is the distance from the fracture within the matrix cell and V_{ij} the volume of the matrix cell. This allows a proper accounting for the reduced influence of a fracture segment on a matrix cell if the fracture segment does not cross the matrix cell through its centre. Figure 6 shows the numerical discretization of a single fracture within a regular matrix and helps to understand the concept of the connectivity index. In many cases the equation above has to be evaluated by numerical integration. For rectangular grids however, there exists an analytical solution for fracture intersections horizontally, vertically or on the diagonal to a grid cell (Hajibeygi et al., 2011). This has later been extended to arbitrary intersections with rectangular grid cells by (Pluimers, 2015). For enhanced efficiency, these analytical expressions are used in the present implementation.

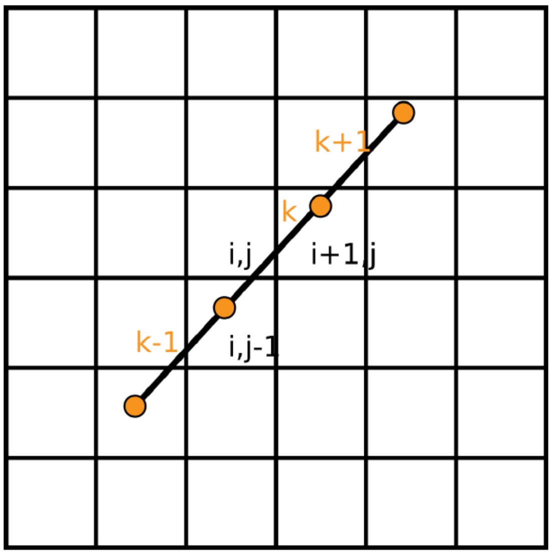


Figure 6: Discretization of a single fracture within a regular matrix. Black indices denote matrix cells whereas coloured indices belong to the fracture discretization. The fracture segment k is intersecting cells i, j-1; i, j and i+1, j. It is obvious that the fracture's influence on the matrix cells is not equal. This is captured by the connectivity index CI, which depends on the lengths and position of the fracture intersection with the matrix cell.

FRACTURE INTERSECTIONS

Fractures often intersect other fractures in naturally fractured reservoirs, which potentially significantly impacts flow dynamics in the reservoir. Therefore, we must also consider fracture-fracture coupling in the model. The additional transmissivity at a fracture intersection can be obtained using an approach similar to that in electrical and known as the star-delta transformation in circuits. The additional fracture-fracture transmissivity can be calculated as

$$T_{i,j} = \frac{\alpha_i \cdot \alpha_j}{\alpha_i + \alpha_j} \text{ with } \alpha_i = \frac{b_i \Xi_\mathrm{i}}{0.5 \cdot \Delta x_f}$$

where b_i denotes the fracture aperture, Ξ_i the total mobility and Δx_f the numerical discretization spacing in the fracture. This approach can be generalized to more than two fractures intersecting in a single point on a fracture segment, but is omitted here due to its rare occurrence.

TIME-DISCRETIZATION

The time derivatives in the pressure and heat equations are treated using the backward Euler method, which is an implicit time-discretization of order one. This means that the local truncation error (defined as the error made in one step) is $O(h^2)$. The backward Euler method is an unconditionally stable implicit time-discretization scheme that theoretically allows arbitrarily large time steps. In practice, when encountering non-linear behavior, such as the temperature- and pressure-dependent evolution of fluid density, issues with non-convergence might appear and place an indirect restriction on the time-step. Nonetheless, much larger time steps are allowed in the implemented method when compared to explicit schemes.

SYSTEM OF EQUATIONS

The discretized equations for pressure and transport can be assembled separately as linear systems of equations of the form $A \cdot x = b$. Since the matrix and fracture domains are strongly coupled, the solution for both domains must be found either in iteratively where the fracture and matrix systems are solved independently until reaching convergence, or implicitly at the same time. The implicit solution can be achieved through the assembly of a block matrix of the pressure or the transport systems respectively, i.e.

$$A = \begin{pmatrix} A^{mm} & A^{mf} \\ A^{fm} & A^{ff} \end{pmatrix}$$

Assuming that A has been assembled for the pressure equation, then the diagonal submatrix block A^{mm} contains the matrix transmissivities. A^{ff} contains the fracture transmissivities, including the additional transmissivities due to fracture-fracture intersections, and the off-diagonal submatrix blocks contain the fracture-matrix and matrix-fracture transmissivities A^{fm}/A^{mf}. The block matrix structure for the heat transport equation remains unchanged. Of course the transmissivities are replaced by the appropriate coefficients of the heat transport equation. Due to the numerical treatment of the advection term a different sparse pattern arises. In any case, the solution of the assembled block matrices is performed by the backslash operator of MATLAB in the current implementation.

SOLUTION STRATEGY

We adapt a serial iterative scheme in order to accurately account for the coupling between the pressure and transport equations. Instead of assembling and solving a very large system for pressure and transport in a single step, the problem is divided in two parts. In a first step, the pressure system is assembled and solved. Using Darcy's law, the fluid velocities can be calculated in an intermediate step. Once the fluid velocities are found, the transport system can be assembled and subsequently solved. At this point the fracture stability is evaluated. As this is an analytical analysis no additional system of equations is involved. In strongly coupled problems, multiple iterations must be used to capture any arising nonlinearities. In most cases, the flow and transport exhibit rather loose coupling in which only a few iterations are needed to converge to the solution. If on the other hand, fracture stability ceases and permeability enhancement in unstable fracture parts is used, the number of iterations might increase significantly and even place a limit on the allowable timestep.

FLOWCHART

THERMAID consists of multiple parts that are involved in the solution of a coupled flow, heat transport and fracture stability problem. The general structure of the main function that controls the simulation and calls most of the sub functions is presented in a simplified flow chart. Throughout the flowchart the following terms are used: *time* denotes the current point in time in the simulation, *timeEnd* is the specified duration of the simulation, *err(p,T)* is the residual error of the pressure and temperature solutions between iterations and *tol* a user specified tolerance of the acceptable residual error. The sub functions that deal with specific parts of the program, such as initialization, discretization and solution of the flow and transport equations, are further explained in the beginning of the user manual and in the inline documentation.

INTERFACE FOR USER-DEFINED CALCULATIONS

As shown in the flowchart, there are two points in the procedure where users can introduce additional calculations easily without having to modify the general implementation. This allows users to add their own visualization procedures, model calculations such as e.g. the implementation of time-dependent boundary conditions and even interfaces to third-party programs. This interface is presented as two additional MATLAB scripts that can be specified during the simulation startup that are executed at the specified positions in each time-step. Using conditionals, it is possible to restrict the additional computations only at certain times during the simulations. Examples of how to use the interfaces are provided with the package and in the user manual.

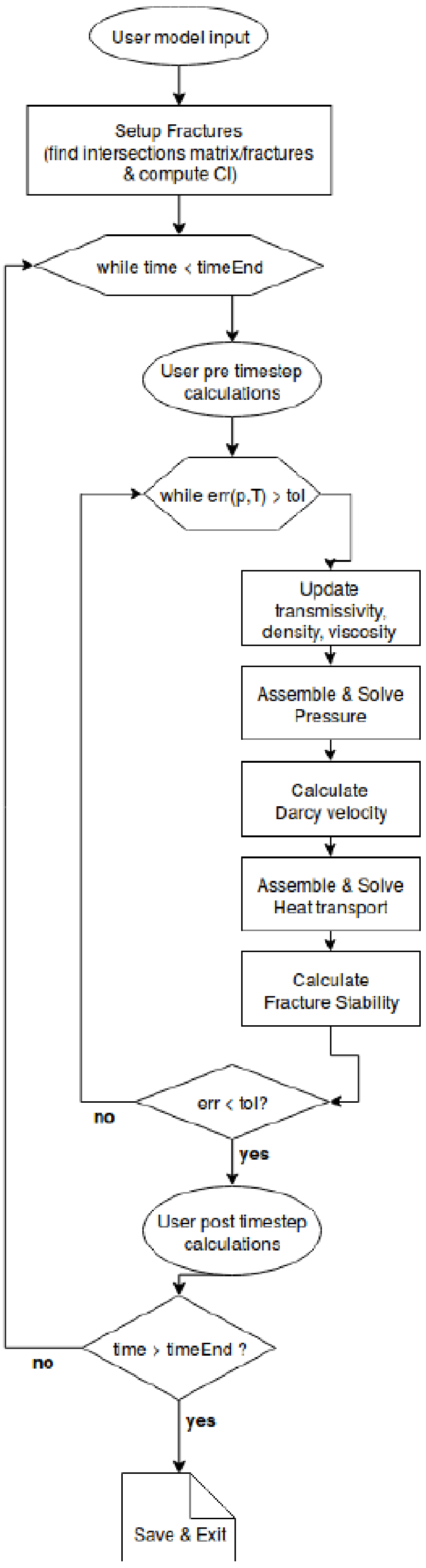

User model input
Setup Fractures (find intersections matrix/fractures & compute CI)
while time < timeEnd
User pre timestep calculations
while err(p,T) > tol
Update transmissivity, density, viscosity
Assemble & Solve Pressure
Calculate Darcy velocity
Assemble & Solve Heat transport
Calculate Fracture Stability
err < tol?
no
yes
User post timestep calculations
time > timeEnd ?
no
yes
Save & Exit

Figure 7: The simplified flowchart of the implemented solution strategy for coupled flow and transport problems including fracture stability analysis. Round objects denote parts of the code where the user has (optional) abilities to perform additional calculations and add runtime visualization.

VISUALIZATION WITH THERMAID

Visualization with Thermaid is simple due to MATLAB's visualization features. After Thermaid simulations have finished, the results and all other relevant Thermaid variables are written into the MATLAB workspace. From here you can continue to use any visualization routine that you are familiar with in MATLAB. We have found the plot and pcolor functions to be especially useful. Examples of this can be found in the `ex1.m` and `ex2.m` scripts.

Thermaid can also produce visualization at runtime. This is enabled through the `attach_post_timestep.m` function. The standard `attach_post_timestep.m` function includes optional visualization that are turned on by the showPlot flag that is an input parameter to the Thermaid function as discussed in section Running the Code. By default a pressure solution plot is generated. If the `udata.flagHeatTransport` is set in the input file, an additional plot of the temperature solution is produced.

UNDERSTANDING EFFICIENT RUNTIME VISUALIZATION

More information will follow.

BIBLIOGRAPHY

Hajibeygi, H., Karvounis, D., & Jenny, P. (2011). A hierarchical fracture model for the iterative multiscale finite volume method. *Journal of Computational Physics*, *230*(24), 8729–8743. http://doi.org/10.1016/j.jcp.2011.08.021