The Journal of Open Source Software

Frackit: a framework for stochastic fracture network generation and analysis

Dennis Gläser[1], **Bernd Flemisch**[1], **Holger Class**[1], and **Rainer Helmig**[1]

1 Department of Hydromechanics and Modelling of Hydrosystems, University of Stuttgart, Pfaffenwaldring 61, 70569 Stuttgart, Germany

DOI: 10.21105/joss.02291

Software

- Review
- Repository
- Archive

Editor: Patrick Diehl

Reviewers:

- @ilyasst
- @ctdegroot

Submitted: 27 May 2020
Published: 11 December 2020

Summary

`Frackit` is a framework for the stochastic generation of fracture networks composed of two-dimensional geometries, for instance, polygons and/or ellipses, which can be embedded in arbitrary three-dimensional domain shapes. Great flexibility with respect to the geometries that can be used is achieved by extensive use of the open-source Computer-Aided-Design (CAD) library OpenCascade (opencascade.com), which furthermore provides the possibility to export the generated networks into a variety of CAD file formats for subsequent post-processing or meshing with other software. Besides that, output routines to the geometry file format of Gmsh (gmsh.info/; Geuzaine & Remacle (2009)) are provided, which is an open-source mesh generator that is widely used in academic research on numerical methods (see e.g. Keilegavlen et al. (2020); Berge et al. (2020)). The code is written in C++, but Python bindings are provided that allow users to access all functionality from Python.

Introduction

The numerical simulation of flow and transport phenomena in fractured porous media is an active field of research, given the importance of fractures in many geotechnical engineering applications, as for example groundwater management (Qian et al., 2014), enhanced oil recovery techniques (Torabi et al., 2012), geothermal energy (McFarland & Murphy, 1976; Shaik et al., 2011) or unconventional natural gas production (Sovacool, 2014). A number of mathematical models and numerical schemes, aiming at an accurate description of flow through fractured rock, have been presented recently (see e.g. R. Ahmed et al. (2015); Raheel Ahmed et al. (2017); Brenner et al. (2018); Köppel et al. (2019); Schädle et al. (2019); Nordbotten et al. (2019)). Many of these describe the fractures as lower-dimensional geometries, that is, as curves or planes embedded in two- or three-dimensional space, respectively. On those, integrated balance equations are solved together with transmission conditions describing the interaction with the surrounding medium. Moreover, many models require that the computational meshes used for the different domains are conforming in the sense that the faces of the discretization used for the bulk medium coincide with the discretization of the fractures (see Figure 1).

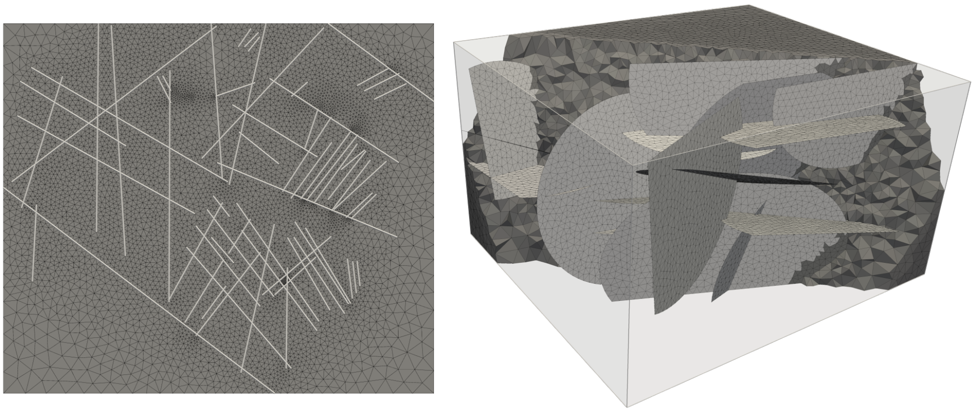

Figure 1: Exemplary grids used with numerical schemes that require conformity of the bulk discretization with the fracture planes. The network shown on the left is taken from Bernd Flemisch et al. (2018).

Information on the in-situ locations of fractures is typically sparse and difficult to determine. As a response to this, a common approach is to study the hydraulic properties of rock in function of the fracture network topology by means of numerical simulations performed on stochastically generated fracture networks. Such investigations have been presented, among many others, in Ito & Yongkoo (2003); Assteerawatt (2008); I.-H. Lee & Ni (2015); Zhang (2015); I. Lee et al. (2019). An overview over several other works in which stochastically generated fracture networks have been used can be found in Lei et al. (2017).

In many such studies, the researchers have used self-developed codes for the stochastic generation of the fracture networks, but unfortunately, in most cases the code is not publicly available. Two examples for open-source software packages are Hyman et al. (2015) and Alghalandis (2017), both of which contain functionality related to network generation and analysis, while the former additionally provides meshing and simulation capabilities to simulate flow and transport on the generated networks. An example for a commercial software package is FracMan (golder.com/fracman), while DFN.LAB (fractorylab.org) is a non-open-source research code, to which access is only granted to research collaborators of the development team.

Statement of need

In the context of simulation-based research on fractured media, stochastically generated fracture networks are useful in applications where field data is sparse or unavailable, where the impact of different network patterns is studied, or when the performance of a numerical scheme is to be investigated on networks that exhibit certain properties. However, when the simulations are carried out with grid-based numerical schemes, the mesh quality is of particular importance and should be taken into account already during the network generation. In particular, small length scales and angles at intersecting geometries should be avoided, as well as small distances due to fractures being very close to each other or to the domain boundary. Within the above-mentioned non-commercial packages, such a feature (without the check for small angles) is only available in Hyman et al. (2015). They describe fractures by polygons, which are also used to approximate elliptical shapes, and the domains are restricted to hexahedra. In `Frackit`, such checks can be performed between arbitrary, possibly non-linear geometries, for instance, between elliptical disks and cylinder surfaces, and there is no need to represent geometries by linear approximations. This enables users to mesh the resulting geometries with the desired resolution without artifacts from the discretization (approximation) of the geometries themselves.

Generators often work with input files for setting a large number of parameters of the network generation, or provide a single generator class with parameter setter functions, in which the entire network generation process occurs. While this enables new users to obtain results quickly and with less effort, this might also limit the possibilities for customization of the generation algorithm. In contrast to that, `Frackit` is structured in a modular way, allowing users to pick the functionalities they need to assemble custom network generation algorithms. The modular design also facilitates the incorporation of new features without having to modify existing classes or functions. Moreover, the generated networks can be exported into a variety of output file formats for subsequent post-processing, meshing and simulation with other software. If desired, the geometric data produced by `Frackit` contains the complete fragmentation of all geometric entities involved, i.e. the intersection geometries between all entities are computed. Thus, this information can be directly used in the context of discrete fracture-matrix (dfm) simulations in a conforming way as described before. For instance, the open-source simulator DuMuX [B. Flemisch et al. (2011); Koch et al. (2020); dumux.org] contains a module for conforming dfm simulations of single- and multi-phase flow through fractured porous media, which has been used in several works (Andrianov & Nick, 2019; Gläser et al., 2017, 2019). It supports the Gmsh file format, and thus, `Frackit` can be used in a fully open-source toolchain with Gmsh and DuMuX to generate random fracture networks, construct computational meshes, and perform analyses on them by means of numerical simulations.

Concept

The design of `Frackit` is such that there is no predefined program flow, but instead, users should implement their own applications using the provided classes and functions. This allows for full customization of each step of the network generation. Furthermore, in the case of available measurement data, one could skip the network generation process and use `Frackit` to compute the fragmentation of the measured data and to generate CAD files for subsequent meshing. The code is structured around three fundamental tasks involved in the generation of random networks:

- Random generation of a new fracture entity candidate based on statistical parameters (and possibly information on previously generated entities)
- Evaluation of geometric constraints for a new entity candidate against previously generated entities or the domain boundary
- Fragmentation of the generated raw entities and the embedding domain

This paper focusses on the first two of the above-mentioned steps, and we refer to the code documentation for details on further available functionalities. The presented code snippets focus on the implementation in C++, and for details on how to use `Frackit` from Python we refer to the examples provided in the repository (git.iws.uni-stuttgart.de/tools/frackit).

Random generation of fracture entities

In the network generation procedure, a domain is populated with fracture entities that are generated following user-defined statistical properties regarding their size, orientation and spatial distribution. In `Frackit`, this process is termed *geometry sampling* and is realized in the code in *sampler* classes. In the current implementation, there are three such sampler classes available, which sample quadrilaterals, polygons and elliptical disks in three-dimensional space. A sampler class of `Frackit` receives an instance of a `PointSampler` implementation and a number of probability distributions that define the size and orientation of the raw entities. `PointSampler` classes are used to sample the spatial distribution of the geometries inside a domain geometry. For example, a point sampler that samples points uniformly within the unit cube (defined in the variable `domain`) could be constructed like this:

```cpp
// the type used for coordinates values
using ctype = double;

// define axis-aligned box in which to sample the centers points
using Domain = Frackit::Box<ctype>;
Domain domain(0.0, 0.0, 0.0,  // xmin, ymin, zmin
              1.0, 1.0, 1.0); // xmax, ymax, zmax

// let us uniformly sample points within this box
auto pointSampler = Frackit::makeUniformPointSampler(domain);
```

The convenience function `makeUniformPointSampler()` can be used for uniform sampling over the provided domain geometry. For nun-uniform samplers, one can write

```cpp
auto pointSampler = Frackit::makePointSampler<Traits>(domain);
```

where in the `Traits` class users define the type of distribution to be used for each coordinate direction. Inside a geometry sampler class, a geometry is created by sampling a point from the point sampler, and then constructing an instance of the desired geometry around this point using the provided distributions for its size and orientation. For example, the `QuadrilateralSampler` class expects distributions for the strike and dip angles, and for the lengths in strike and dip direction. The following piece of code shows how an instance of the `QuadrilateralSampler` class can be created, using normal distributions for all parameters regarding orientation and uniform distributions for the lengths (we reuse the `pointSampler` variable defined in the previous code snippet):

```cpp
// let us use uniform distributions for the quadrilateral parameters
using NormalDistro = std::normal_distribution<ctype>;
using UniformDistro = std::uniform_real_distribution<ctype>;

// Distributions for strike/dip angles & lengths
NormalDistro strikeAngleDistro(toRadians(45.0), // mean value
                               toRadians(5.0)); // standard deviation
NormalDistro dipAngleDistro(toRadians(45.0),    // mean value
                            toRadians(5.0));    // standard deviation
UniformDistro strikeLengthDistro(0.4, 0.6);     // min & max length
UniformDistro dipLengthDistro(0.4, 0.6);        // min & max length

// instance of the quadrilateral sampler class
using QuadSampler = Frackit::QuadrilateralSampler</*spaceDimension*/3>;
QuadSampler quadSampler(pointSampler,
                        strikeAngleDistro,
                        dipAngleDistro,
                        strikeLengthDistro,
                        dipLengthDistro);
```

As for point samplers, one can use different distributions by implementing a `Traits` class which is then passed to the `QuadrilateralSampler` as template argument. The definitions of the strike and dip angles as used within the `QuadrilateralSampler` class are illustrated in Figure 2. Consider a quadrilateral whose center is the origin and which lies in the plane defined by the two basis vectors $\mathbf{b}_1$ and $\mathbf{b}_2$. The latter lies in the x-y-plane and the strike angle is the angle between the y-axis and $\mathbf{b}_2$. The dip angle describes the angle between $\mathbf{b}_1$ and the x-y-plane.

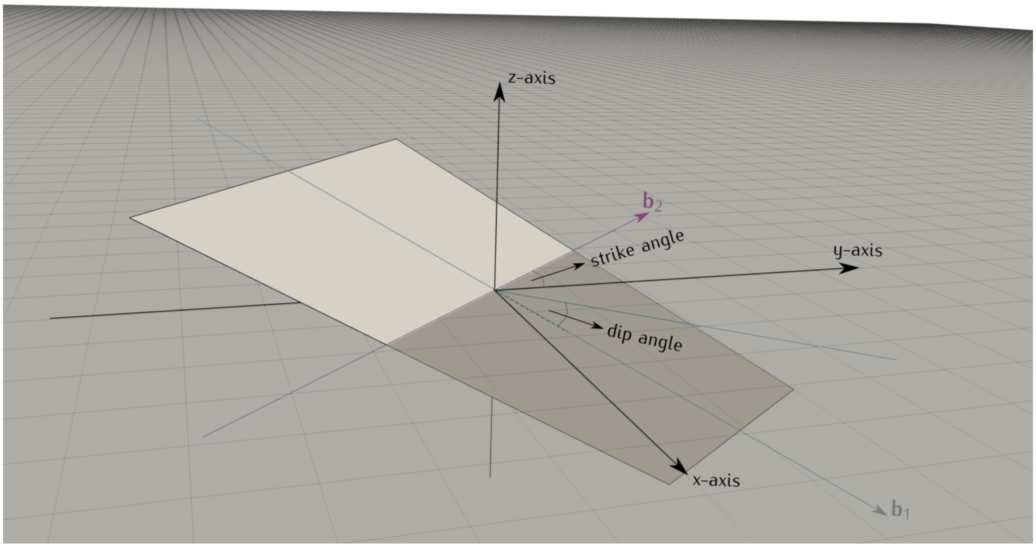

Figure 2: Illustration of the strike and dip angles involved in the random generation of quadrilaterals. The grey plane with the structured mesh illustrates the x-y-plane.

In the code, random generation of geometries from sampler classes occurs by using the `()` operator. For example, from the `quadSampler` variable defined in the previous code snippet, we obtain a random quadrilateral by writing:

```
// generate random quadrilateral
const auto quad = quadSampler();
```

Evaluation of geometric constraints

While the domain is populated with the raw fracture entities, users have the possibility to enforce geometric constraints between them in order for the network to exhibit the desired topological characteristics such as, for instance, fracture spacing. Furthermore, constraints can be used to avoid very small length scales that could cause problems during mesh generation or could lead to ill-shaped elements. In the code, constraints can be defined and evaluated using the `EntityNetworkConstraints` class. These have to be fulfilled by a new fracture entity candidate against previously accepted entities. If any of the defined constraints is violated, the candidate may be rejected and a new one is sampled. The current implementation of the `EntityNetworkConstraints` class allows users to define a minimum distance between two entities that do not intersect. If two entities intersect, one can choose to enforce a minimum length of the intersection curve, a minimum intersection angle and a minimum distance between the intersection curve and the boundaries of the intersecting entities. An illustration of such situations is shown in Figure 3.

The Journal of Open Source Software

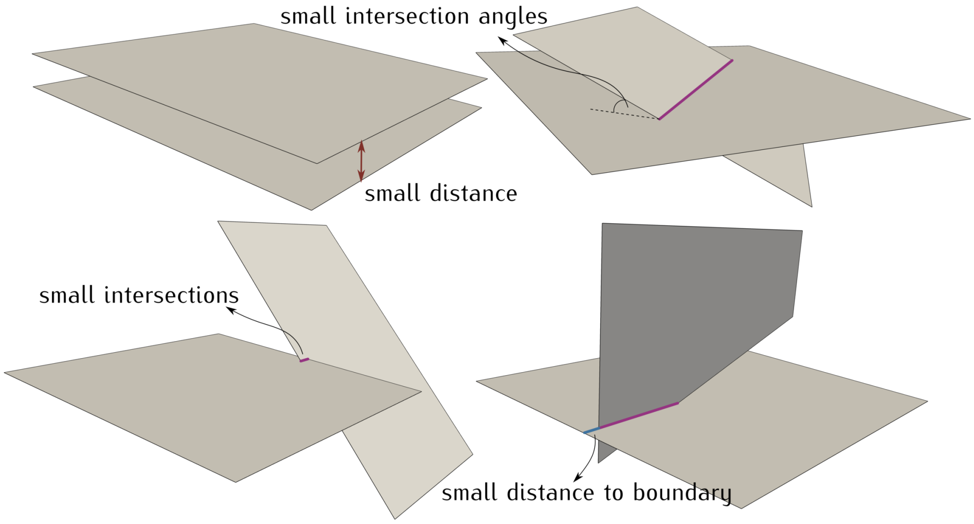

Figure 3: Illustration of the geometric settings that can be avoided using geometric constraints.

Please note that internally, the `EntityNetworkConstraints` class uses geometry algorithms that are part of the public interface of `Frackit`. Thus, one can easily develop custom constraints and enforce additional geometric constraints. The following code snippet illustrates how to set up an instance of the `EntityNetworkConstraints` class:

```
// Instantiate constraints class. This leaves all constraints deactivated.
Frackit::EntityNetworkConstraints<ctype> constraints;

// Set values to activate constraints
constraints.setMinDistance(0.1);                  // in meter
constraints.setMinIntersectingAngle(M_PI/4.0); // in radians
constraints.setMinIntersectionMagnitude(0.05); // in meter
constraints.setMinIntersectionDistance(0.05);  // in meter

// define geometric tolerance to be used for intersections
constraints.setIntersectionEpsilon(1e-6);
```

When using the default constructor of `EntityNetworkConstraints` all constraints are inactive, and when defining values for the different constraint types, these get activated internally. The last line in the above code snippet shows how to define the geometric tolerance that should be used in the intersection algorithms between entities. If no tolerance is set, a default value is computed based on the size of the entities for which the intersection is to be determined. For two quadrilaterals `quad1` and `quad2`, one can then evaluate the defined constraints by writing:

```
bool fulfilled = constraints.evaluate(quad1, quad2);
```

The function `evaluate()` returns true if all constraints are fulfilled. One can also check the fulfillment of the constraints of a new candidate against an entire set of entities. Let `quad` be a new candidate for a quadrilateral, and `quadSet` be a vector of quadrilaterals (`std::vector< Quadrilateral<ctype> >`), then one can write

```cpp
bool fulfilled = constraints.evaluate(quadSet, quad);
```

to evaluate the constraints between quad and all entities stored in quadSet.

Example application

In the following we want to illustrate an exemplary workflow using Frackit together with Gmsh and DuMuX. The images are taken from the Frackit documentation (git.iws.uni-stuttgart.de/tools/frackit) and the configurations of the geometry samplers are, apart from small modifications, very similar to the ones used in example 3 provided in the Frackit repository. For further details on how to set up such configurations we refer to the source code and the documentation of that example in the repository. Note that this example is not meant to represent a realistic fracture network, but should rather highlight the flexibility of the code with respect to the geometries that can be used.

Let us consider a domain consisting of three solid layers of which we want to generate a fracture network only in the center volume. The following piece of code shows how to read in the domain geometry from a CAD file, extract its three volumes and select the middle one as the subdomain in which to place the fracture network (this assumes knowledge of the ordering of the volumes).

```cpp
////////////////////////////////////////////////////
// 1. Read in the domain geometry from .brep file. //
//    The file name is defined in CMakeLists.txt   //
////////////////////////////////////////////////////
const auto domainShape = Frackit::OCCUtilities::readShape(BREPFILE);

// obtain the three solids contained in the file
const auto solids = Frackit::OCCUtilities::getSolids(domainShape);

// The sub-domain we want to create a network in is the center one.
const auto& networkDomain = solids[1];

// get the bounding box of the domain
const auto bBox = Frackit::OCCUtilities::getBoundingBox(networkDomain);
```

The last command constructs the bounding box of the center volume of our domain, which we can then use to instantiate point sampler classes that define the spatial distribution of the fracture entities. With these, we can construct geometry samplers as outlined above. In this example, we define three geometry sampler instances to sample from three different orientations of fractures, and we use quadrilaterals for two of the orientations and elliptical disks for the third orientation. Moreover, we define different constraints that should be fulfilled between the entities of different orientations. As mentioned above, details on how to implement such settings can be found in example 3 in the Frackit repository.

A number of fractures is then generated for each orientation. Subsequently, the raw entities and the three volumes of the domain are cast into an instance of the ContainedEntityNetwork class. This can be used to define arbitrarily many (sub-)domains, and to insert entities to be embedded in a specific sub-domain. The ContainedEntityNetwork computes and stores the fragments of all entities and sub-domains resulting from mutual intersection. Output routines for instances of this class are implemented, which generate geometry files that are ready to be meshed using designated tools such as Gmsh.

The image below illustrates the workflow chosen in this example, using Frackit to generate a random fracture network, Gmsh to mesh the resulting geometry, and DuMuX to perform a single-phase flow simulation on the resulting mesh. The bottom picture shows the pressure distribution on the fractures and the velocities in the domain as computed with DuMuX, using the illustrated boundary conditions.

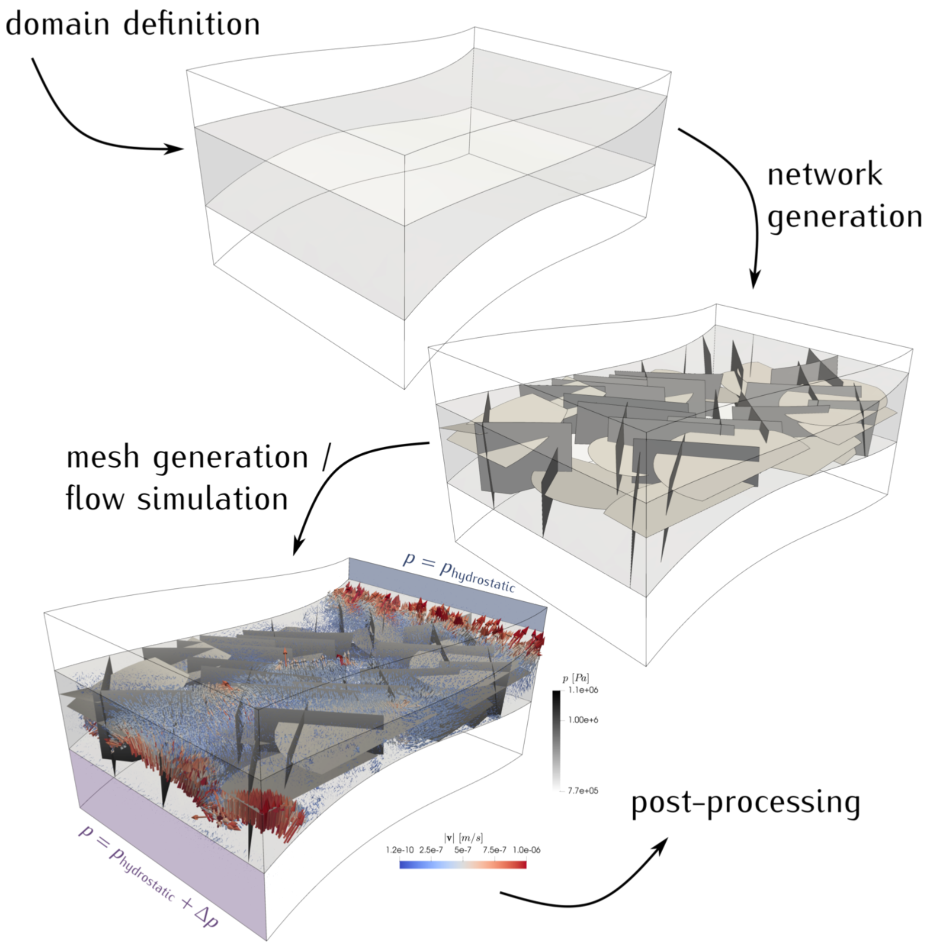

Figure 4: Illustration of the workflow using Frackit, Gmsh and DuMuX in the exemplary application.

The source code of this example, including installation instructions, can be found at https://git.iws.uni-stuttgart.de/dumux-pub/glaeser2020a.

Future developments

We are planning to add fracture network characterization capabilities, such as the detection of isolated clusters of fractures or the determination of connectivity measures. In order to do this efficiently, we want to integrate data structures and algorithms for graphs, together with functionalities to translate the generated fracture networks into graph representations.

Acknowledgements

We thank the German Research Foundation (Deutsche Forschungsgemeinschaft) for supporting this work by funding SFB 1313, Project Number 327154368.

References

Ahmed, R., Edwards, M. G., Lamine, S., Huisman, B. A. H., & Pal, M. (2015). Control-volume distributed multi-point flux approximation coupled with a lower-dimensional fracture model. *Journal of Computational Physics*, *284*, 462–489. https://doi.org/10.1016/j.jcp.2014.12.047

Ahmed, Raheel, Edwards, M. G., Lamine, S., Huisman, B. A. H., & Pal, M. (2017). CVD-MPFA full pressure support, coupled unstructured discrete fracture–matrix darcy-flux approximations. *Journal of Computational Physics*, *349*, 265–299. https://doi.org/10.1016/j.jcp.2017.07.041

Alghalandis, Y. F. (2017). ADFNE: Open source software for discrete fracture network engineering, two and three dimensional applications. *Computers & Geosciences*, *102*, 1–11. https://doi.org/10.1016/j.cageo.2017.02.002

Andrianov, N., & Nick, H. M. (2019). Modeling of waterflood efficiency using outcrop-based fractured models. *Journal of Petroleum Science and Engineering*, *183*, 106350. https://doi.org/10.1016/j.petrol.2019.106350

Assteerawatt, A. (2008). *Flow and transport modelling of fractured aquifers based on a geostatistical approach* [PhD thesis, Universitätsbibliothek der Universität Stuttgart]. https://doi.org/10.18419/opus-289

Berge, R. L., Berre, I., Keilegavlen, E., Nordbotten, J. M., & Wohlmuth, B. (2020). Finite volume discretization for poroelastic media with fractures modeled by contact mechanics. *International Journal for Numerical Methods in Engineering*, *121*(4), 644–663. https://doi.org/10.1002/nme.6238

Brenner, K., Hennicker, J., Masson, R., & Samier, P. (2018). Hybrid-dimensional modelling of two-phase flow through fractured porous media with enhanced matrix fracture transmission conditions. *Journal of Computational Physics*. https://doi.org/10.1016/j.jcp.2017.12.003

Flemisch, Bernd, Berre, I., Boon, W., Fumagalli, A., Schwenck, N., Scotti, A., Stefansson, I., & Tatomir, A. (2018). Benchmarks for single-phase flow in fractured porous media. *Advances in Water Resources*, *111*, 239–258. https://doi.org/10.1016/j.advwatres.2017.10.036

Flemisch, B., Darcis, M., Erbertseder, K., Faigle, B., Lauser, A., Mosthaf, K., Müthing, S., Nuske, P., Tatomir, A., Wolff, M., & Helmig, R. (2011). DuMux: DUNE for multi-Phase, Component, Scale, Physics, … Flow and transport in porous media. *Advances in Water Resources*, *34*, 1102–1112. https://doi.org/10.1016/j.advwatres.2011.03.007

Geuzaine, C., & Remacle, J.-F. (2009). Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, *79*(11), 1309–1331. https://doi.org/10.1002/nme.2579

Gläser, D., Flemisch, B., Helmig, R., & Class, H. (2019). A hybrid-dimensional discrete fracture model for non-isothermal two-phase flow in fractured porous media. *GEM-International Journal on Geomathematics*, *10*(1), 5. https://doi.org/10.1007/s13137-019-0116-8

Gläser, D., Helmig, R., Flemisch, B., & Class, H. (2017). A discrete fracture model for two-phase flow in fractured porous media. *Advances in Water Resources*, *110*, 335–348. https://doi.org/10.1016/j.advwatres.2017.10.031

Hyman, J. D., Karra, S., Makedonska, N., Gable, C. W., Painter, S. L., & Viswanathan, H. S. (2015). dfnWorks: A discrete fracture network framework for modeling subsurface flow and transport. *Computers & Geosciences*, *84*, 10–19. https://doi.org/10.1016/j.cageo.2015.08.001

Ito, K., & Yongkoo, S. (2003). *A 3-dimensional discrete fracture network generator to examine fracture-matrix interaction using TOUGH2*. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States).

Keilegavlen, E., Berge, R., Fumagalli, A., Starnoni, M., Stefansson, I., Varela, J., & Berre, I. (2020). PorePy: An open-source software for simulation of multiphysics processes in fractured porous media. *Computational Geosciences*, 1–23. https://doi.org/10.1007/s10596-020-10002-5

Koch, T., Gläser, D., Weishaupt, K., Ackermann, S., Beck, M., Becker, B., Burbulla, S., Class, H., Coltman, E., Emmert, S., Fetzer, T., Grüninger, C., Heck, K., Hommel, J., Kurz, T., Lipp, M., Mohammadi, F., Scherrer, S., Schneider, M., … Flemisch, B. (2020). DuMu^x 3 - an open-source simulator for solving flow and transport problems in porous media with a focus on model coupling. *Computers & Mathematics with Applications*. https://doi.org/10.1016/j.camwa.2020.02.012

Köppel, M., Martin, V., & Roberts, J. E. (2019). A stabilized lagrange multiplier finite-element method for flow in porous media with fractures. *GEM-International Journal on Geomathematics*, *10*(1), 7. https://doi.org/10.1007/s13137-019-0117-7

Lee, I.-H., & Ni, C.-F. (2015). Fracture-based modeling of complex flow and CO2 migration in three-dimensional fractured rocks. *Computers & Geosciences*, *81*, 64–77. https://doi.org/10.1016/j.cageo.2015.04.012

Lee, I., Ni, C.-F., Lin, F.-P., Lin, C.-P., Ke, C.-C., & others. (2019). Stochastic modeling of flow and conservative transport in three-dimensional discrete fracture networks. *Hydrology and Earth System Sciences*, *23*(1), 19–34. https://doi.org/10.5194/hess-23-19-2019

Lei, Q., Latham, J.-P., & Tsang, C.-F. (2017). The use of discrete fracture networks for modelling coupled geomechanical and hydrological behaviour of fractured rocks. *Computers and Geotechnics*, *85*, 151–176. https://doi.org/10.1016/j.compgeo.2016.12.024

McFarland, R. D., & Murphy, H. (1976). *Extracting energy from hydraulically-fractured geothermal reservoirs*. Los Alamos Scientific Lab., N. Mex.(USA).

Nordbotten, J. M., Boon, W. M., Fumagalli, A., & Keilegavlen, E. (2019). Unified approach to discretization of flow in fractured porous media. *Computational Geosciences*, *23*(2), 225–237. https://doi.org/10.1007/s10596-018-9778-9

Qian, J., Zhou, X., Zhan, H., Dong, H., & Ma, L. (2014). Numerical simulation and evaluation of groundwater resources in a fractured chalk aquifer: A case study in zinder well field, niger. *Environmental Earth Sciences*, *72*(8), 3053–3065. https://doi.org/10.1007/s12665-014-3211-z

Schädle, P., Zulian, P., Vogler, D., Bhopalam, S. R., Nestola, M. G. C., Ebigbo, A., Krause, R., & Saar, M. O. (2019). 3D non-conforming mesh model for flow in fractured porous media using lagrange multipliers. *Computers & Geosciences*, *132*, 42–55. https://doi.org/10.1016/j.cageo.2019.06.014

Shaik, A. R., Rahman, S. S., Tran, N. H., & Tran, T. (2011). Numerical simulation of fluid-rock coupling heat transfer in naturally fractured geothermal system. *Applied Thermal Engineering*, *31*(10), 1600–1606. https://doi.org/10.1016/j.applthermaleng.2011.01.038

Sovacool, B. K. (2014). Cornucopia or curse? Reviewing the costs and benefits of shale gas hydraulic fracturing (fracking). *Renewable and Sustainable Energy Reviews*, *37*, 249–264. https://doi.org/10.1016/j.rser.2014.04.068

Torabi, F., Firouz, A. Q., Kavousi, A., & Asghari, K. (2012). Comparative evaluation of immiscible, near miscible and miscible CO2 huff-n-puff to enhance oil recovery from a single matrix–fracture system (experimental and simulation studies). *Fuel*, *93*, 443–453. https://doi.org/10.1016/j.fuel.2011.08.037

Zhang, Q.-H. (2015). Finite element generation of arbitrary 3-d fracture networks for flow analysis in complicated discrete fracture networks. *Journal of Hydrology*, *529*, 890–908. https://doi.org/10.1016/j.jhydrol.2015.08.065