open-darts

Release 0.1.5

Denis Voskov

Aug 24, 2023

ABOUT DARTS

CHAPTER

ONE

WHAT IS DARTS?

Delft: we belong to Civil Engineering and Geoscience (CEG) Department at Civil Engineering Faculty of TU Delft. The development team directly linked to the new GeoEnergy program which connects Geology, Geophysics and Petroleum Engineering sections of the department.

Advanced: the simulation framework is based on the recently proposed Operator-Based Linearization (OBL) approach, which helps to decouple the complex nonlinear physics and advanced unstructured discretization from the core simulation engine. The framework is targeting the solution of forward and inverse problems.

Research: the development team includes five PhD students from the CEG department and multiple MSc students working on their thesis project on DARTS platform. The simulation platform is developed within Delft Advanced Reservoir Simulation (DARSim) program and linked to multiple research in the area of reservoir simulation, inverse modeling and uncertainty quantification.

Terra: the developed framework is utilized for forward and inverse problems in petroleum engineering, low- and high-enthalpy geothermal applications, subsurface storage and subsurface integrity. The primary focus and developed capabilities are currently cover low-enthalpy geothermal operations which include multicomponent multiphase flow of mass and heat with complex chemical interactions. Another focus is thermal-compositional processes for Enhanced Oil Recovery.

Simulator: the main simulation kernel implemented on multi-core CPU and many-core GPU architectures. Advance multiscale nonlinear formulation improves the performance of forward and inverse models. Additional afford invested in representative proxy models for complex subsurface processes including multiphase multicomponent flow.

1.1 Introduction

DARTS is constructed within the Operator-based Linearization (OBL) framework for general-purpose reservoir simulations. The 'super-engine' in DARTS contains the governing equations characterizing the general thermal-compositional-reactive system should be expressed in form of operators, which are the functions of the thermodynamic state. These operators are then utilized to construct the residual and Jacobian for the solution of a nonlinear system using interpolation in thermodynamic parameter space. The values and derivatives of these operators at the supporting points will be evaluated adaptively during the simulation or pre-calculated in form of tables through the physics implemented in either C++ or Python.

The assembly of resulting Jacobian matrix in C++ is generalized and improved by the OBL approach, which greatly facilitates code development. The constructed linear system is passed to the dedicated linear solver for a solution. More details about the DARTS architecture are described in [1].

To enable the convenient usage of DARTS, the functionalities in C++ are exposed to users via a Python interface. At the same time, the Python interface provides the possibility to integrate different external modules (e.g., packages for physical property calculation) into DARTS.

1.1.1 Table of Contents

1.2 Conservation Equations

Mass and heat transfer involves a thermal multiphase flow system, which requires a set of equations to depict the flow dynamics. In this section, the governing equations and detailed spatial and temporal discretization and linearization procedures are introduced.

1.2.1 Governing Equations

For the investigated domain with volume Ω, bounded by surface Γ, the mass and energy conservation can be expressed in a uniformly integral way, as

$$\frac{\partial}{\partial t}\int_{\Omega} M^c d\Omega + \int_{\Gamma} \mathbf{F^c}\cdot\mathbf{n} d\Gamma = \int_{\Omega} Q^c d\Omega.$$

Here, M^c denotes the accumulation term for the c^{th} component ($c = 1, \ldots, n_c$, indexing for the mass components, [e.g., water, CO_2] and $c = n_c + 1$ for the energy quantity); $\mathbf{F_c}$ refers to the flux term of the c^{th} component; $\mathbf{n}$ refers to the unit normal pointing outward to the domain boundary; Q_c denotes the source/sink term of the c^{th} component.

The mass accumulation term collects each component distribution over n_p fluid phases in a summation form,

$$M^c = \phi \sum_{j=1}^{n_p} x_{cj}\rho_j s_j + (1-\phi), \quad c = 1, \ldots, n_c,$$

where ϕ is porosity, s_j is phase saturation, ρ_j is phase density $[\mathrm{kmol/m^3}]$ and x_{cj} is molar fraction of c component in j phase.

The energy accumulation term contains the internal energy of fluid and rock,

\begin{equation} \begin{aligned} M^{n_c+1} = \phi\sum\limits^{n_p}_{j=1}\rho_js_jU_j + (1 - \phi)U_r, \end{aligned} \end{equation}

where U_j is phase internal energy $[\mathrm{kJ}]$ and U_r is rock internal energy $[\mathrm{kJ}]$. The rock is assumed compressible and represented by the change of porosity through:

\begin{equation} \phi = \phi_0 \big(1 + c_r (p - p_\mathrm{ref}) \big), \end{equation}

where ϕ_0 is the initial porosity, c_r is the rock compressibility [1/bar] and p_{ref} is the reference pressure [bars].

The mass flux of each component is represented by the summation over n_p fluid phases,

\begin{equation} \begin{aligned} \bf{F^c} = \sum\limits_{j=1}^{n_p}x_{cj}\rho_j \bf{u_j} + s_{j}\rho_{j} \textbf{J}_{cj}, \quad c = 1, \ldots, n_c. \end{aligned} \end{equation}

Here the velocity $\mathbf{u_j}$ follows the extension of Darcy's law to multiphase flow,

$$\mathbf{u_j} = -\mathbf{K}\frac{\mathbf{k_{rj}}}{\mu_\mathbf{j}}(\nabla \mathbf{p_j} - \gamma_\mathbf{j}\nabla \mathbf{z}),$$

where $\mathbf{K}$ is the permeability tensor $[\mathrm{mD}]$, k_{rj} is the relative permeability of phase j, μ_j is the viscosity of phase j $[\mathrm{mPa \cdot s}]$, p_j is the pressure of phase j [bars], $\gamma_\mathbf{j} = \rho_\mathbf{j}\mathbf{g}$ is the specific weight $[\mathrm{N/m^3}]$ and z is the depth vector [m].

The $\mathbf{J}_{cj}$ is the diffusion flux of component c in phase j, which is described by Fick's law as

$$\mathbf{J}_{cj} = -\phi \mathbf{D}_{cj}\nabla x_{cj},$$

where $\mathbf{D}_{cj}$ is the diffusion coefficient [$\mathrm{m^2}$/day].

The energy flux includes the thermal convection and conduction terms,

\begin{equation} \begin{aligned} \bf{F}^{n_c+1} = \sum\limits^{n_p}_{j=1}h_j\rho_j \bf{u_j} + \kappa\nabla{T}, \end{aligned} \end{equation}

where h_j is phase enthalpy $[\mathrm{kJ/kg}]$ and κ is effective thermal conductivity $[\mathrm{kJ/m/day/K}]$.

Finally, the source term in mass conservation equations can be present in the following form

\begin{equation} \begin{aligned} {Q}^{c} = \sum\limits_{k=1}^{n_k}v_{ck}r_k, \quad c = 1, \ldots, n_c, \end{aligned} \end{equation}

where q_j is the phase source/sink term from the well, v_{ck} is the stoichiometric coefficient associated with chemical reaction k for the component c and r_k is the rate for the reaction. Similarly, the source term in the energy balance equation can be written as

\begin{equation} \begin{aligned} {Q}^{n_c+1} = \sum\limits_{k=1}^{n_k}v_{ek}r_{ek}. \end{aligned} \end{equation}

Here v_{ek} is the stoichiometric coefficient associated with kinetic reaction k for the energy and r_{ek} is the energy rate for kinetic reaction.

The nonlinear equations are discretized with the finite volume method using the multi-point flux approximation on general unstructured mesh in space and with the backward Euler approximation in time. For the i^{th} reservoir block, the governing equation in discretized residual form reads:

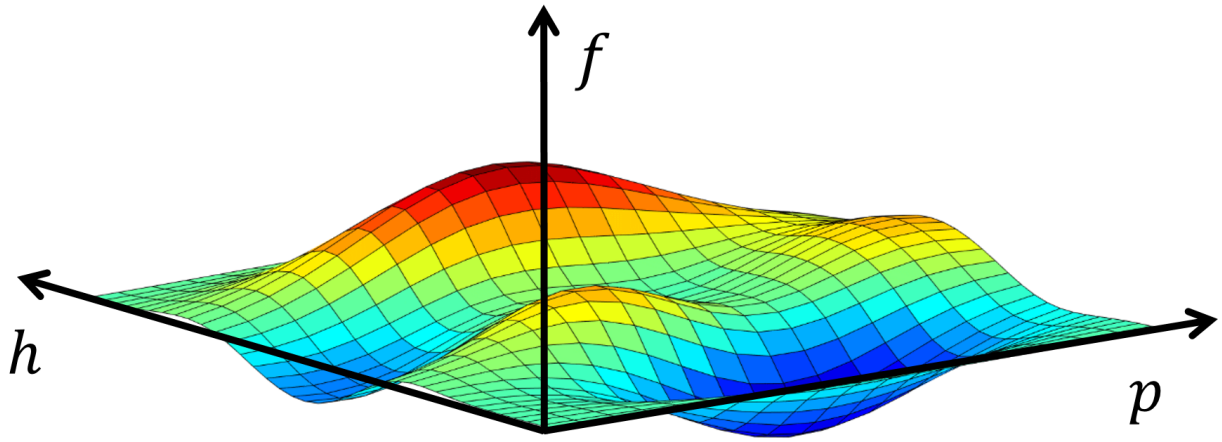
$$R_i^c = V_i\Big(M_i^c(\omega_i) - M_i^c(\omega_i^n)\Big) - \Delta t\Big(\sum_l A_l F_l^c(\omega) + V_i Q_i^c(\omega)\Big) = 0, \quad c = 1, \ldots, n_c + 1.$$

Here V_i is the volume of the i^{th} grid block, ω_i refers to state variables at the current time step, ω_i^n refers to state variables at previous time step, A_l is the contact area between neighboring grids.

1.3 Operator Form of Governing Equations

DARTS includes capabilities for the solution of forward and inverse problems for subsurface fluid and heat transport. The OBL approach is employed in DARTS for the solution of highly nonlinear problems. It was proposed recently for generalized complex multi-phase flow and transport applications and aims to improve the simulation performance [2, 3]. For spatial discretization, a finite volume fully implicit method in combination with two-point flux approximation on unstructured grids is implemented in DARTS. Besides conventional discretization in temporal and spatial space, DARTS also utilizes discretization in physical space using the OBL approach.

With the OBL approach, the governing equations are written in form of state-dependent operators. The state-dependent operators can be parameterized with respect to nonlinear unknowns in multi-dimension tables under different resolutions. The values and derivatives of the operators in the parameter space can be interpolated and evaluated based on supporting points. For the adaptive parameterization technique [4], the supporting points are calculated `on the fly' and stored for later re-usage, which can largely save time for parameterization in high-dimension parameter space (i.e. in multi-component compositional simulations). At the same time, the Jacobian assembly becomes flexible with the OBL, even for very complex physical problems.



Parameterization of the operators in 2D (pressure & enthalpy) space with a predefined OBL resolution [adapted from][5]. Here, the size of the quadrilateral represents the resolution for an operator interpolation.

1.3.1 Conservation of mass and energy

Pressure, temperature and overall composition are taken as the unified state variables in a given control volume in general-purpose thermal-compositional simulation. Upstream weighting of the physical state is used to determine the flux-related fluid properties determined at the interface l. The discretized mass conservation equation in operator form for girdblock (here we omit i) reads:

\begin{equation} V\phi_0[\alpha_c (\omega) -\alpha_c(\omega_n)]-\Delta t\sum_{l\in L(i)}\sum_{j=1}^{n_p}[\Gamma^l\beta_{cj}^l(\omega^u)\Delta\psi_j^l + \Gamma_d^l\gamma_{j}^l(\omega)\Delta \chi_{cj}]+\Delta t V \delta_c(\omega)=0 . \end{equation}

where V is the control volume, ω_n is the physical state of block i at the previous timestep, ω is the physical state of block i at the new timestep, ω^u is the physical state of upstream block, Γ^l and Γ_d^l are the fluid and diffusive transmissibilities respectively and $L(i)$ is a set of interfaces for gridblock i.

Here we defined the following state-dependent operators,

$$\alpha_{cf}(\omega) = \Big(1 + c_r(p - p_{ref})\Big) \sum_{j=1}^{n_p} x_{cj}\rho_j s_j,\ c = 1, \ldots, n_c; \tag{1.1}$$

$$\beta_{cj}(\omega) = x_{cj}\rho_j k_{rj}/\mu_j,\ c = 1, \ldots, n_c,\ j = 1, \ldots, n_p; \tag{1.2}$$

$$\gamma_j(\omega) = \Big(1 + c_r(p - p_{ref})\Big) s_j,\ j = 1, \ldots, n_p; \tag{1.3}$$

$$\chi_{cj}(\omega) = D_{cj}\rho_j x_{cj},\ c = 1, \ldots, n_c,\ j = 1, \ldots, n_p; \tag{1.4}$$

$$\delta_c(\omega) = \sum_{j=1}^{n_p} v_{cj} r_j(\omega),\ c = 1, \ldots, n_c; \tag{1.5}$$

The phase-potential-upwinding (PPU) strategy for OBL parametrization is applied in DARTS to model the gravity and capillary effect [4, 6]. The potential difference of phase j on the interface l between block 1 and 2 can be written as:

\begin{equation} \Delta \psi^l_{j} = p_1-p^c_{j}(\omega_1) - (p_2-p^c_{j}(\omega_2))-\frac{\rho_j(\omega_1)+\rho_j(\omega_2)}{2}g(z_2-z_1), \end{equation}

where p_j^c is the capillary pressure.

The discretized energy conservation equation in operator form can be written as:

$$V\phi_0[\alpha_{ef}(\omega) - \alpha_{ef}(\omega_n)] - \Delta t \sum_{l\in L(i)} \sum_{j=1}^{n_p} [\Gamma^l \beta^l_{ej}(\omega^u)\Delta\psi^l_j + \Gamma^l_d \gamma_j(\omega)\Delta\chi_{ej}] + \Delta t V \delta_e(\omega)$$
$$+ (1-\phi_0)VU_r[\alpha_{er}(\omega) - \alpha_{er}(\omega_n)] - \Delta t \sum_l (1-\phi_0)\Gamma^l_d \kappa_r \alpha_{er}(\omega)\Delta\chi_{er} = 0,$$

where:

$$\alpha_{ef}(\omega) = \Big(1 + c_r(p - p_{ref})\Big) \sum_{j=1}^{n_p} \rho_j s_j U_j; \tag{1.1}$$
$$\beta_{ej}(\omega) = h_j \rho_j k_{rj}/\mu_j,\ j = 1, \ldots (n_p); \tag{1.2}$$
$$\chi_{ej}(\omega) = \kappa_j T_j,\ j = 1, \ldots (n_p); \tag{1.3}$$
$$\delta_e(\omega) = \sum_{j=1}^{n_j} v_{ej} r_{ej}(\omega) \tag{1.4}$$

In addition, for accounting the energy of rock, three additional operators should be defined:

$$\alpha_{eri}(\omega) = \frac{U_r}{1 + c_r(p - p_{ref})}, \tag{1.1}$$
$$\alpha_{erc}(\omega) = \frac{1}{1 + c_r(p - p_{ref})} \tag{1.2}$$
$$\chi_{er}(\omega) = \kappa_r \tag{1.3}$$

α_{eri} and α_{erc} represent the rock internal energy and rock conduction, respectively. U_r is a state-dependent parameter, thus these two rock energy terms are treated separately.

This agglomeration of different physical terms into a single nonlinear operator simplifies the implementation of nonlinear formulations. Instead of performing complex evaluations of each property and its derivatives with respect to nonlinear unknowns, operators can be parameterized in physical space either at the pre-processing stage or adaptively with a limited number of supporting points. The evaluation of operators during the simulation is based on multi-linear interpolation, which improves the performance of the linearization stage. Besides, due to the piece-wise representation of operators, the nonlinearity of the system is reduced, which improves the nonlinear behavior [3, 4]. However, to delineate the nonlinear behavior in the system, especially strong nonlinearity (e.g., at high-enthalpy conditions), it is necessary to select a reasonable OBL resolution to characterize the physical space. Too coarse OBL resolution may lead to a large error in the solutions [2].

1.3.2 Well treatment

A connection-based multi-segment well is used to simulate the flow in the wellbore [7]. The communication between well blocks and reservoir blocks is treated in the same way as between reservoir blocks. Besides, the top well block is connected with a ghost control volume, which is selected as a placeholder for the well control equations. The bottom hole pressure (BHP), volumetric and mass rate controls are available in DARTS to model various well conditions.

As for the BHP well control, the injector and/or producer will operate under fixed bottom-hole pressure. A pressure constraint is defined at the ghost well block:

\begin{equation} p-p^{target} = 0. \end{equation}

The volumetric rate control in DARTS is implemented through the volumetric rate operator $\zeta_p^{vol}(\omega)$:

\begin{equation} \Gamma^l\zeta_j^{vol}(\omega)\Delta{p} - Q^{target} = 0, \end{equation}

where

$$\zeta_j^{vol} = \frac{\hat{s}_j(\omega)\sum_c \beta_{cj}(\omega)}{\hat{\rho}_t(\omega)},$$

where Q^{target} is the target volumetric flow rate at separator conditions $[\mathrm{m}^3/\mathrm{day}]$, $\beta_{cj}(\omega)$ is the mass flux operator as shown in (1.1), $\hat{s}_j$ and $\hat{\rho}_t(\omega)$ are the saturation and total fluid density respectively at separator conditions.

Any of the described well controls can be coupled with energy boundary conditions, defined by the temperature or enthalpy of the injected fluid at the injection well. Since temperature is the function of the thermodynamic state, it is expressed in operator form and the temperature well control reads:

\begin{equation} \chi(\omega) - T^{target} = 0, \end{equation}

where $\chi(\omega)$ is defined by (1.1) and T^{target} is the target temperature of injected fluid. Alternatively, the enthalpy of the injected fluid can be defined:

\begin{equation} h(\omega) - h^{target} = 0, \end{equation}

where h is the enthalpy of the well control block, h^{target} is the target enthalpy of injected fluid. For the production well control, enthalpy is taken equal to that of the upstream well block.

1.3.3 Treatment of porosity

The porosity ϕ depends on the concentrations of the minerals according to the relationship:

$$\phi = 1 - \sum_{m=1}^{M} \frac{\mathcal{M}_m c_{ms}}{\rho_m},$$

where M is the number of reactive minerals, $\mathcal{M}_m$ is the molar mass of mineral m, ρ_m is the mass density of mineral m and c_{ms} represents and the molar concentration of mineral m.

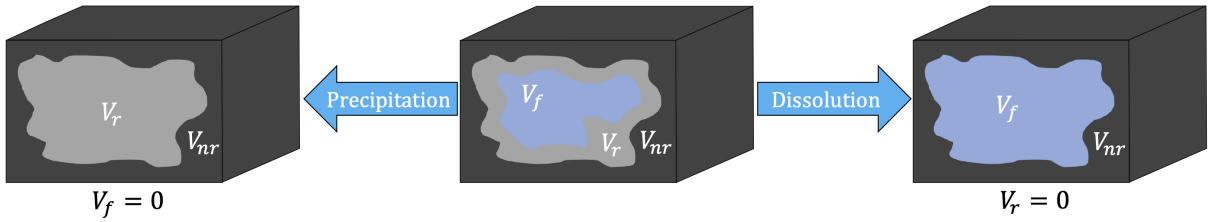
Let's represent the total bulk volume of control element in simulation

\begin{equation} V = V_f + V_r + V_{nr}, \end{equation}

and V_r denotes reactive volume and V_{nr} represents the non-reactive volume (not altered by any chemical reaction). Dividing this by the total (bulk) volume gives

\begin{equation} 1 = \phi + \phi_r + \phi_{nr} = \phi^T + \phi_{nr}, \end{equation}

where ϕ_r represents the reactive volume fraction, ϕ_{nr} is the non-reactive volume fraction, and ϕ^T is the total porosity defined as the sum of the fluid porosity and reactive volume fraction. Since only the reactive volume and fluid porosity can change due to chemical reactions, it follows directly that the total porosity remains constant throughout simulation (when neglecting compressibility). This and the changes in volume fractions due to precipitation and dissolution is illustrated in the following figure.



Schematic of the different volumes in the domain. The domain consists of three distinct regions, particularly the fluid

volume which is occupies by all the mobile phases (liquid and gaseous in the case of two phase flow), the reactive volume which consist of solid phases that can react or precipitate, and finally the nonreactive volume (the part of the control volume which doesn't participate in any chemical reaction)

Note that the fluid porosity can always be obtained with the following constitutive equation

$$\phi = \phi^T \Big(1 - \sum_{m=1}^{M} \hat{s}_m\Big),$$

where M is the number of solid phases (occupying the reactive volume fraction) and $\hat{s}_m$ is the saturation of solid phase. Please note that the s_α is the fluid saturation (defined over the pore volume) while $\hat{s}_m$ is the {solid saturation of mineral phase m} (defined over the pore and reactive rock volume).

1.4 References

CHAPTER

TWO

INSTALLATION OF OPEN-DARTS

2.1 Using pip

We recomend to use a python environment. Then installing open-darts is as simple as:

```
pip install open-darts
```

TO build open-darts from source go to build instructions

CHAPTER

THREE

TUTORIAL

3.1 Getting started with any model

In order to use DARTS you need to set up the model. It usually includes a few Python scripts called main.py, model.py, and reservoir.py. main.py usually manages the simulation run, the time stepping, and the output. model.py usually contains the description of the physics, and all the parameters required to define it. reservoir.py usually describes the simulation domain, computational grid and the static properties assigned to them. They are usually organized in a way that main.py is a startup scripts where the model is created, then it calls for the model constructor or some other model methods overloaded in model.py which then creates a reservoir and calls its methods defined in reservoir.py.

Apart of these three Python scripts, the model folder may contain extra scripts used for model description or post-processing, or benchmarking purposes. It also may include a mesh folder, or just *.geo, *.msh, *.grdecl files placed in the root folder. They define computational grids used in the model. After the simulation run the program may create a separate folder for the output files.

Plenty of different models are located in darts-models repository. As an example, we can run Unstructured_fine model.

1. Run main.py
2. In the command line you may see the program output that usually consist of two parts: first - pre-processing output, second - the output from time steps. Depending on the size of the computational grid the pre-processing part may take from a couple of seconds up to an hour. If you see the repeating output from time steps afterward, it means that the program is working fine.
3. In the end of the program you may see a picture plotted in python with some key calculated property or the result of the comparison of the calculated solution against other data. The standard output of the program may be found in a new folder called 'sol_*' which contains VTK snapshots of every times step. However, they are not written by default in every model.

3.2 Configure hardware usage

To set the number of threads (CPU cores) to be used, add:

from darts.engines import set_num_threads set_num_threads(NT)
All cores used by default.

Turn on GPU usage in calculation by adding the next lines at the start of the python script: Add palatform='gpu' at physics constructor, usually in model.py, for example
self.physics = DeadOil(..., platform='gpu')

If you would like to change the GPU device, add these lines to your model script:

from darts.engines import set_gpu_device set_gpu_device(N)

with N is your GPU device number. For example if you have 2 GPUs, you can call `set_gpu_device(0)` or `set_gpu_device(1)`.

CHAPTER

FOUR

HOW IS BUILT OPEN-DARTS?

The source code of open-darts is spread over 5 repositories hosted on gitlab.com and is built and tested automatically via continuous integration / continuous deployment (CI/CD) service. Open-darts is a python package with binary extensions, built from C++/OpenMP/CUDA code. The python part of open-darts is maintained in darts-package. The extension engines lives in darts-engines.

The build process reminds a chain. First, ecternal libraries (linear solvers and flash) are built. They are compiled as static libraries and are not aware of python. Then, binary extensions (engines) is built. They are linked to a specific python version. Finally, the python package along with extensions is packed into a python wheel. In order to make sure that the package works as expected, a test suite is executed for every compiled wheel in darts-models. Each of these steps is performed as a Gitlab pipeline. Once changes are made to the repository (someone pushes a commit), its pipeline is triggered. If it successfully finishes, then the pipeline of the next link in the chain is triggered, and so one until the whole package is built and tested. The pipeline status of each of the steps described above can be monitored via badges on the main page.

Open-darts wheel is compiled for Windows and Linux platforms, for Python 3.5, 3.6, 3.7 and 3.8. Therefore, starting from engines extensionsand for every subsequent step, the pipeline consists of 8 independent jobs. In this way, every version of open-darts package is verified to function as expected. Intermediate build results along with python wheels can be downloaded according to the instructions in the installation section of the getting started page.

4.1 More information

Visit the open-darts wiki.

CHAPTER

FIVE

GLOSSARY

In this section, we provide a short description of concepts and classes used in darts-package.

Concepts:

Cell - 3D polyhedral or 2D multilateral geometry, e.g. prism, tetrahedron, hexahedron, pyramid or quad, triangle.

Computational grid - the geometrical representation of a particular domain or a body by partitioning it into a set of smaller geometrical elements (cells). They are usually used to resolve spatial variability of material properties of the domain or physical processes happening there for the purpose of computer modeling.

Structured grid - the computational grid where it is possible to come up with such a cell numbering that will allow identifying the neighbors of every cell in a uniform way. Examples: rectangular grid in Cartesian, polar or spherical coordinates, curved quadrilateral grid

Unstructured grid - the computational grid where it is not possible to identify the neighbors of every cell from their ids in a uniform way. It is cell-specific.

Flux - a stream of some quantity over the particular interface, e.g. flux of fluid mass, heat flux, a flux of mechanical stress. It may be a scalar or a vector.

Flux Approximation - the representation of flux at a particular interface as a function of the quantities defined in the points close to the interface.

Two-Point Flux Approximation (TPFA) - the simple representation of flux as a function of quantity defined at two different points. It is usually used to approximate fluxes caused be the pressure or molecular diffusion, dispersion, or heat conduction processes. It is robust and simple but non-consistent for the general anisotropic cases.

Multi-Point Flux Approximation (MPFA) - the more advanced representation of flux as a function of quantity defined at a few different points. It has the same applications as TPFA but and it is capable to provide consistent approximation in general. However, its robustness faces numerical issues (monotonicity, coercivity, discrete maximum principle).

Multi-Point Stress Approximation (MPSA) - the multi-point approximation used to represent the traction vector at the interface. It utilizes the same concept as MPFA but is used for different applications (continuum mechanics).

Connection list - the dataset that contains the flux approximations over all relevant interfaces in the computational grid. The unified format of this dataset that supports TPFA, MPFA, and MPSA includes the following arrays: left and right cell ids that define the interface between two cells where the flux is approximated, offset that indicates the starting position of every flux approximation in the following arrays, stencil consists of cell ids contributing to the particular approximation, and trans includes the coefficients of the contribution from every cell in the stencil.

Classes:

StructDiscretizer provides two-point flux approximation (TPFA) for a structured grid either created rectangular or Corner-Point Geometry read from file. The approximation includes simple geometry processing, calculation of transmissibility list, filtering, and casting procedures.

UnstructDiscretizer provides a set of approximation methods for an unstructured polyhedral three-dimensional grig generated by Gmsh grid generator. The supported approximations are two-point flux approximation (TPFA) for both

fluid and heat fluxes, multi-point flux approximation (MPFA) and multi-point stress approximation (MPSA). Fractures are supported by TPFA only. This discretizer contains a number of arrays that store the geometry of either matrix or fracture cells used in all methods. Multi-point approximations also perform extended geometry processing (finding the neighbors, generation of faces) and it usually stores boundary faces and fracture shapes as long as matrix and fracture cells.

TransCalculations includes a set of methods that calculate particular TPFA between different kinds of cells (matrix, fracture).

ControlVolume is a parent class for the cells of different kinds of geometry: Hexahedron, Wedge, Pyramid, Tetrahedron, Quadrangle, Triangle, Line.

Face is a class used to store all required information about the interface. It is used in MPFA, MPSA.

nbHexahedron, nbWedge, nbPyramid, nbTetrahedron is an attempt to redefine cell classes with numba support.

StructReservoir represents a parent class for the domain covered with a structured grid.

DartsModel is a parent class for all models.

CHAPTER

SIX

RESERVOIR

Subsurface reservoirs are the main bodies for the modelling in DARTS. Reservoir is usually represented by its geometry covered by some computaional grid, properties like porosity, permeability or stiffness defined in the grid, boundary conditions that define fluxes over the reservoir boundary and some other things. Along with the model, reservoir defines the parameters required for any kind of modelling in DARTS.

The kind of computational grid spanning reservoir produces two types of reservoirs: structured and unstructured. The treatment of structured reservoir can be generalized that was done by StructReservoir class provided in DARTS. Many models use it directly without overloading and extension built-in methods. The models working with unstructured reservoir have to provide their own implementation that is usually represented by UnstructReservoir class defined in reservoir.py script.

6.1 Unstructured reservoir

Let us describe the basic parts which unstructured reservoir must and may include.

6.1.1 Constructor

In many (simple) models the constructor may get some model parameters like porosity and permeability, the type of physics and boundary conditions. In general case it becomes difficult to define reservoir parameters in model and send them to reservoir constructor. In this case all reservoir properties and associated parameters may be defined in reservoir.py.

Below we will discuss only the basic pieces of code, excluding many other lines of code.

Usually constructor starts includes the creating of conn_mesh class defined in C++ back-end:

```
self.mesh = conn_mesh()
```

followed by the calling the constructor of unstructured discretizer, e.g.

```
self.unstr_discr = UnstructDiscretizer(permx=permx, permy=permy, permz=permz, frac_
↪aper=frac_aper, mesh_file=mesh_file)
```

mesh loading

```
self.unstr_discr.load_mesh()
```

and some processing

```
self.unstr_discr.calc_cell_information()
```

This is basic scenario for this part of constructor, which can be limited for many models. Some reservoirs require different implementation of last two functions (unstructured, fluidflower, mpfa, mpsa) including some pre- and post-processing of reservoir parameters required for construction of unstructured discretizer, some reservoirs use different discretizer at all (mpfa-mpsa). However, almost all reservoir parameters are defined (or have to be defined) in this block of code in order to initialize and run discretizer. Therefore, in the models working with multi-point approximations this block of code was placed into separate method of unstructured reservoir to be called here in the constructor(mpfa, mpsa mpfa-mpsa).

Once the necessary properties prescribed and discretizer is initialized, the discretization can be run as follows

```
cell_m, cell_p, tran, tran_thermal = self.unstr_discr.calc_connections_all_
↪cells(cache=False)
```

in the case of two-point flux approximation,

```
self.cell_m, self.cell_p, self.stencil, self.offset, self.trans = self.unstr_discr.calc_
↪mpfa_connections_all_cells(True)
```

in the case of multi-point flux approximation,

```
self.pm.reconstruct_gradients_per_cell(dt)
self.pm.calc_all_fluxes_once(dt)
```

in the case of poromechanics discretizer `pm`.

The output obtained from discretizer then can be written in files and have to be provided to `self.mesh` in an initialization call

```
self.mesh.init(...)
```

where the call and number of arguments varies with different types of physics and discretization.

Next step is to expose all the specified arrays to c++ backend. Pybind11 allows to make a numpy wrap around c++ vectors in order to copy the data.

In many models the boundary conditions are specified in the end of constructor because they do not affect the results of discretization. Generally, it is not the case and we need to specify them before running discretization (mpfa, mpsa, mpfa-mpsa). Also in some models the well locations are found in the constructor. Although it is not needed for discretization, these data is used afterwards in the post-proccessing of discretization results.

The constructor represents the main functionality of `UnstructReservoir` class: defining the input data for discretizer, runnning discretization and passing data and results to c++ backend. Some models may introduce extra functions that called in the constructor in order to assign input for discretization, for pre- or post-processing or other purposes (fluidflower, mpfa, mpsa, mpfa-mpsa).

6.1.2 Well functions

The most of implementations of `UnstructReservoir` class contain a few functions that help to initialize wells. The calls

```
def add_well(self, name, depth)
def add_perforation(self, well, res_block, well_index)
def init_wells(self)
```

may have different implementations, but they are quite general for most of the models.

6.1.3 Writing output

One of the calls

```
def write_to_vtk(self, ...)
def export_vtk(self, ...)
```

is usually used in order to write output in VTK format. They can have different number of arguments and different implementations specific for a particular model or reservoir. However, it can be generalized.

CHAPTER

SEVEN

ENGINES

Delft Advanced Research Terra Simulator

class darts.engines.**conn_mesh**

Bases: pybind11_object

Class for connection-based mesh and it`s properties

add_conn(*self:* darts.engines.conn_mesh, *arg0: int*, *arg1: int*, *arg2: float*, *arg3: float*) → int

add_wells(*self:* darts.engines.conn_mesh, *arg0: darts.engines.ms_well_vector*) → int

add_wells_mpfa(*self:* darts.engines.conn_mesh, *arg0: darts.engines.ms_well_vector*, *arg1: int*) → int

connect_segments(*self:* darts.engines.conn_mesh, *arg0:* ms_well, *arg1:* ms_well, *arg2: int*, *arg3: int*, *arg4: int*) → int

get_res_tran(*self:* darts.engines.conn_mesh, *tran: darts.engines.value_vector*, *tranD: darts.engines.value_vector*) → int

Get reservoir transmissibilities

get_wells_tran(*self:* darts.engines.conn_mesh, *tran: darts.engines.value_vector*) → int

Get well indexes

init(**args*, ***kwargs*)

Overloaded function.

1. init(self: darts.engines.conn_mesh, arg0: str) -> int

Initialize from TPFACONNS (TPFACONNSN) keyword file

2. init(self: darts.engines.conn_mesh, block_m: darts.engines.index_vector, block_p: darts.engines.index_vector, tran: darts.engines.value_vector, tranD: darts.engines.value_vector = value_vector[]) -> int

Initialize by connection list defined by block_m, block_p, tran and tranD arrays

init_const_1d(*self:* darts.engines.conn_mesh, *arg0: float*, *arg1: int*) → int

init_grav_coef(*self:* darts.engines.conn_mesh, *grav_const: float = 9.80665e-05*) → int

Initialize gravity coefficients for every connection

init_mpfa(*self:* darts.engines.conn_mesh, *arg0: darts.engines.index_vector*, *arg1: darts.engines.index_vector*, *arg2: darts.engines.index_vector*, *arg3: darts.engines.index_vector*, *arg4: darts.engines.value_vector*, *arg5: darts.engines.value_vector*, *arg6: darts.engines.value_vector*, *arg7: darts.engines.value_vector*, *arg8: int*, *arg9: int*, *arg10: int*, *arg11: int*) → int

init_mpsa(**args*, ***kwargs*)

Overloaded function.

1. init_mpsa(self: darts.engines.conn_mesh, arg0: darts.engines.index_vector, arg1: darts.engines.index_vector, arg2: darts.engines.index_vector, arg3: darts.engines.index_vector, arg4: darts.engines.value_vector, arg5: int, arg6: int, arg7: int, arg8: int) -> int

2. init_mpsa(self: darts.engines.conn_mesh, arg0: darts.engines.index_vector, arg1: darts.engines.index_vector, arg2: darts.engines.index_vector, arg3: darts.engines.index_vector, arg4: darts.engines.value_vector, arg5: darts.engines.value_vector, arg6: int, arg7: int, arg8: int, arg9: int) -> int

init_pm(**args*, ***kwargs*)

Overloaded function.

1. init_pm(self: darts.engines.conn_mesh, arg0: darts.engines.index_vector, arg1: darts.engines.index_vector, arg2: darts.engines.index_vector, arg3: darts.engines.index_vector, arg4: darts.engines.value_vector, arg5: darts.engines.value_vector, arg6: int, arg7: int, arg8: int) -> int

2. init_pm(self: darts.engines.conn_mesh, arg0: darts.engines.index_vector, arg1: darts.engines.index_vector, arg2: darts.engines.index_vector, arg3: darts.engines.index_vector, arg4: darts.engines.value_vector, arg5: darts.engines.value_vector, arg6: darts.engines.value_vector, arg7: darts.engines.value_vector, arg8: int, arg9: int, arg10: int) -> int

3. init_pm(self: darts.engines.conn_mesh, arg0: darts.engines.index_vector, arg1: darts.engines.index_vector, arg2: darts.engines.index_vector, arg3: darts.engines.index_vector, arg4: darts.engines.value_vector, arg5: darts.engines.value_vector, arg6: darts.engines.value_vector, arg7: darts.engines.value_vector, arg8: darts.engines.value_vector, arg9: darts.engines.value_vector, arg10: int, arg11: int, arg12: int) -> int

init_pme(*self:* darts.engines.conn_mesh, *arg0: darts.engines.index_vector*, *arg1: darts.engines.index_vector*, *arg2: darts.engines.index_vector*, *arg3: darts.engines.index_vector*, *arg4: darts.engines.value_vector*, *arg5: darts.engines.value_vector*, *arg6: darts.engines.value_vector*, *arg7: darts.engines.value_vector*, *arg8: darts.engines.value_vector*, *arg9: darts.engines.value_vector*, *arg10: int*, *arg11: int*, *arg12: int*) → int

init_poro(*self:* darts.engines.conn_mesh, *arg0: str*) → int

reverse_and_sort(*self:* darts.engines.conn_mesh) → int

reverse_and_sort_dvel(*self:* darts.engines.conn_mesh) → int

reverse_and_sort_mpfa(*self:* darts.engines.conn_mesh) → int

reverse_and_sort_mpsa(*self:* darts.engines.conn_mesh) → int

reverse_and_sort_pm(*self:* darts.engines.conn_mesh) → int

reverse_and_sort_pme(*self:* darts.engines.conn_mesh) → int

save_enthalpy(*self:* darts.engines.conn_mesh, *arg0: str*) → int

save_poro(*self:* darts.engines.conn_mesh, *arg0: str*) → int

save_pressure(*self:* darts.engines.conn_mesh, *arg0: str*) → int

save_temperature(*self:* darts.engines.conn_mesh, *arg0: str*) → int

save_volume(*self:* darts.engines.conn_mesh, *arg0: str*) → int

save_zmf(*self:* darts.engines.conn_mesh, *arg0: str*) → int

set_res_tran(*self:* darts.engines.conn_mesh, *tran: darts.engines.value_vector*, *tranD: darts.engines.value_vector*) → int

Set reservoir transmissibilities

set_wells_tran(**args*, ***kwargs*)

Overloaded function.

1. set_wells_tran(self: darts.engines.conn_mesh, tran: darts.engines.value_vector) -> int

Set well indexes

2. set_wells_tran(self: darts.engines.conn_mesh, tran: darts.engines.value_vector) -> int

Set well indexes

class darts.engines.**engine_base**

Bases: pybind11_object

Base simulator engine class

add_value_to_Q(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

add_value_to_Q_inj_p(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

add_value_to_Q_p(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

add_value_to_cov_inj_p(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

add_value_to_cov_prod_p(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

add_value_to_inj_wei_p(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

add_value_to_prod_wei_p(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

apply_newton_update(*self:* darts.engines.engine_base, *arg0: float*) → int

calc_adjoint_gradient_dirac_all(*self:* darts.engines.engine_base) → int

calc_newton_residual(*self:* darts.engines.engine_base) → float

calc_well_residual(*self:* darts.engines.engine_base) → float

clear_Q(*self:* darts.engines.engine_base) → int

clear_Q_inj_p(*self:* darts.engines.engine_base) → int

clear_Q_p(*self:* darts.engines.engine_base) → int

clear_cov_inj_p(*self:* darts.engines.engine_base) → int

clear_cov_prod_p(*self:* darts.engines.engine_base) → int

clear_inj_wei_p(*self:* darts.engines.engine_base) → int

clear_previous_adjoint_assembly(*self:* darts.engines.engine_base) → int

clear_prod_wei_p(*self:* darts.engines.engine_base) → int

post_newtonloop(*self:* darts.engines.engine_base, *arg0: float*, *arg1: float*) → int

print_stat(*self:* darts.engines.engine_base) → int

push_back_to_BHP_all(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

push_back_to_BHP_wei_all(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

push_back_to_Q_all(*self:* darts.engines.engine_base) → int

push_back_to_Q_inj_all(*self:* darts.engines.engine_base) → int

push_back_to_binary_all(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

push_back_to_cov_BHP_all(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

push_back_to_cov_customized_op_all(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

push_back_to_cov_inj_all(*self:* darts.engines.engine_base) → int

push_back_to_cov_prod_all(*self:* darts.engines.engine_base) → int

push_back_to_cov_temperature_all(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

push_back_to_cov_well_tempr_all(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

push_back_to_customized_op_all(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

push_back_to_customized_op_wei_all(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

push_back_to_inj_wei_all(*self:* darts.engines.engine_base) → int

push_back_to_prod_wei_all(*self:* darts.engines.engine_base) → int

push_back_to_temperature_all(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

push_back_to_temperature_wei_all(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

push_back_to_well_tempr_all(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

push_back_to_well_tempr_wei_all(*self:* darts.engines.engine_base, *arg0: darts.engines.value_vector*) → int

report(*self:* darts.engines.engine_base) → int

run(*self:* darts.engines.engine_base, *arg0: float*) → int

run_single_newton_iteration(*self:* darts.engines.engine_base, *arg0: float*) → int

run_timestep(*self:* darts.engines.engine_base, *arg0: float*, *arg1: float*) → int

solve_linear_equation(*self:* darts.engines.engine_base) → int

test_assembly(*self:* darts.engines.engine_base, *arg0: int*, *arg1: int*, *arg2: int*) → int

test_spmv(*self:* darts.engines.engine_base, *arg0: int*, *arg1: int*, *arg2: int*) → int

class darts.engines.**engine_super_cpu1_1_t**

Bases: *engine_base*

Isothermal CPU simulator engine for 1 components and 1 phases with diffusion and kinetic reaction

init(*self:* darts.engines.engine_super_cpu1_1_t, *arg0:* darts.engines.conn_mesh, *arg1: darts.engines.ms_well_vector*, *arg2: darts.engines.op_vector*, *arg3:* darts.engines.sim_params, *arg4:* darts.engines.timer_node) → int

Initialize simulator by mesh, tables and wells

class darts.engines.**engine_super_cpu2_1**

Bases: *engine_base*

Non-isothermal CPU simulator engine for 2 components and 1 phases with diffusion and kinetic reaction

init(*self:* darts.engines.engine_super_cpu2_1, *arg0:* darts.engines.conn_mesh, *arg1: darts.engines.ms_well_vector*, *arg2: darts.engines.op_vector*, *arg3:* darts.engines.sim_params, *arg4:* darts.engines.timer_node) → int

Initialize simulator by mesh, tables and wells

class darts.engines.**engine_super_elastic_cpu1_2**

Bases: *engine_base*

Isothermal CPU simulator engine for 1 components and 2 phases with momentum balance, diffusion and kinetic reaction

apply_newton_update(*self:* darts.engines.engine_super_elastic_cpu1_2, *arg0: float*) → int

calc_newton_residual(*self:* darts.engines.engine_super_elastic_cpu1_2) → darts.engines.value_vector

init(*self:* darts.engines.engine_super_elastic_cpu1_2, *arg0:* darts.engines.conn_mesh, *arg1: darts.engines.ms_well_vector*, *arg2: darts.engines.op_vector*, *arg3:* darts.engines.sim_params, *arg4:* darts.engines.timer_node) → int

Initialize simulator by mesh, tables and wells

post_newtonloop(*self:* darts.engines.engine_super_elastic_cpu1_2, *arg0: float*, *arg1: float*) → int

class darts.engines.**engine_super_mp_cpu2_1**

Bases: *engine_base*

Non-isothermal CPU simulator engine for 2 components and 1 phases with diffusion and kinetic reaction

init(*self:* darts.engines.engine_super_mp_cpu2_1, *arg0:* darts.engines.conn_mesh, *arg1: darts.engines.ms_well_vector*, *arg2: darts.engines.op_vector*, *arg3:* darts.engines.sim_params, *arg4:* darts.engines.timer_node) → int

Initialize simulator by mesh, tables and wells

run_single_newton_iteration(*self:* darts.engines.engine_super_mp_cpu2_1, *arg0: float*) → int

class darts.engines.**ms_well**

Bases: pybind11_object

Multisegment well, modeled as an extension of the reservoir

init_mech_rate_parameters(*self:* darts.engines.ms_well, *N_VARS: int*, *P_VAR: int*, *n_vars: int*, *phase_names: List[str]*, *rate_ev: operator_set_gradient_evaluator_iface*, *thermal: int = 0*) → None

Init by NC and rate operators for poromechanics

init_rate_parameters(*self:* darts.engines.ms_well, *n_vars: int*, *phase_names: List[str]*, *rate_ev: operator_set_gradient_evaluator_iface*, *thermal: int = 0*) → None

Init by NC and rate operators

class darts.engines.**multilinear_adaptive_cpu_interpolator_i_d_1_1**

Bases: operator_set_gradient_evaluator_iface

Operator set interpolator with j index type and d value type for 1 operators in 1-dimensional parameter space

evaluate(*self:* darts.engines.multilinear_adaptive_cpu_interpolator_i_d_1_1, *state: darts.engines.value_vector*, *values: darts.engines.value_vector*) → int

Evaluate operators

evaluate_with_derivatives(*self:* darts.engines.multilinear_adaptive_cpu_interpolator_i_d_1_1, *state: darts.engines.value_vector*, *block_idx: darts.engines.index_vector*, *values: darts.engines.value_vector*, *derivatives: darts.engines.value_vector*) → int

Evaluate operators and derivatives (v)

init(*self:* darts.engines.multilinear_adaptive_cpu_interpolator_i_d_1_1) → int

Initialize interpolator

init_timer_node(*self:* darts.engines.multilinear_adaptive_cpu_interpolator_i_d_1_1, *timer_node:* darts.engines.timer_node) → None

Initialize timer

write_to_file(*self:* darts.engines.multilinear_adaptive_cpu_interpolator_i_d_1_1, *arg0: str*) → int

Write interpolator data to file

class darts.engines.**multilinear_adaptive_cpu_interpolator_l_d_1_1**

Bases: operator_set_gradient_evaluator_iface

Operator set interpolator with m index type and d value type for 1 operators in 1-dimensional parameter space

evaluate(*self:* darts.engines.multilinear_adaptive_cpu_interpolator_l_d_1_1, *state: darts.engines.value_vector*, *values: darts.engines.value_vector*) → int

Evaluate operators

evaluate_with_derivatives(*self:* darts.engines.multilinear_adaptive_cpu_interpolator_l_d_1_1, *state: darts.engines.value_vector*, *block_idx: darts.engines.index_vector*, *values: darts.engines.value_vector*, *derivatives: darts.engines.value_vector*) → int

Evaluate operators and derivatives (v)

init(*self:* darts.engines.multilinear_adaptive_cpu_interpolator_l_d_1_1) → int

Initialize interpolator

init_timer_node(*self:* darts.engines.multilinear_adaptive_cpu_interpolator_l_d_1_1, *timer_node:* darts.engines.timer_node) → None

Initialize timer

write_to_file(*self:* darts.engines.multilinear_adaptive_cpu_interpolator_l_d_1_1, *arg0: str*) → int

Write interpolator data to file

class darts.engines.**pm_discretizer**

Bases: pybind11_object

Multipoint discretizer for poromechanics

calc_all_fluxes_once(*self:* darts.engines.pm_discretizer, *arg0: float*) → None

get_gradient(*self:* darts.engines.pm_discretizer, *arg0: int*) → Tuple[darts.engines.index_vector, List[float]]

get_thermal_gradient(*self:* darts.engines.pm_discretizer, *arg0: int*) → Tuple[darts.engines.index_vector, List[float]]

init(*self:* darts.engines.pm_discretizer, *arg0: int*, *arg1: int*, *arg2: darts.engines.index_vector*) → None

reconstruct_gradients_per_cell(*self:* darts.engines.pm_discretizer, *arg0: float*) → None

reconstruct_gradients_thermal_per_cell(*self:* darts.engines.pm_discretizer, *arg0: float*) → None

class darts.engines.**sim_params**

Bases: pybind11_object

Class simulation parameters

property first_ts

Length of the first time step (days)

class linear_solver_t

Bases: pybind11_object

Available types of linear solvers

Members:

cpu_gmres_cpr_amg

cpu_gmres_ilu0

cpu_superlu

cpu_gmres_cpr_amg1r5

cpu_gmres_fs_cpr

cpu_samg

gpu_gmres_cpr_amg

gpu_gmres_ilu0

gpu_gmres_cpr_aips

gpu_gmres_cpr_amgx_ilu

gpu_gmres_cpr_amgx_ilu_sp

gpu_gmres_cpr_amgx_amgx

gpu_gmres_amgx

gpu_amgx

gpu_gmres_cpr_nf

gpu_bicgstab_cpr_amgx

gpu_cusolver

property name

class newton_solver_t

Bases: pybind11_object

Available types of newton solvers

Members:

newton_std

newton_global_chop

newton_local_chop

newton_inflection_point

property name

class nonlinear_norm_t

Bases: pybind11_object

Available types of nonlinear norm

Members:

L1

L2

LINF

property name

class darts.engines.timer_node

Bases: pybind11_object

Timers tree structure

get_timer(*self:* darts.engines.timer_node) → float

print(*self:* darts.engines.timer_node, *arg0: str*, *arg1: str*) → str

reset_recursive(*self:* darts.engines.timer_node) → None

start(*self:* darts.engines.timer_node) → None

stop(*self:* darts.engines.timer_node) → None

CHAPTER

EIGHT

PHYSICS_SUP

8.1 Properties

8.1.1 Properties Black Oil

8.1.2 Properties CCS Thermal

8.1.3 Basic Properties

8.2 Operators

CHAPTER

NINE

MODELS

class darts.models.darts_model.**DartsModel**

Bases: object

This is a base class for creating a model in DARTS. A model is composed of a darts.models.Reservoir object and a *darts.physics.Physics* object. Initialization and communication between these two objects takes place through the Model object

Variables

- **reservoir** – Reservoir object
- **physics** – Physics object

__init__()

" Initialize DartsModel class.

Variables

- **timer** – Timer object
- **params** – Object to set simulation parameters

export_vtk(*file_name: str = 'data'*, *local_cell_data: dict = {}*, *global_cell_data: dict = {}*, *vars_data_dtype: type = <class 'numpy.float32'>*, *export_grid_data: bool = True*)

Function to export results at timestamp t into *.vtk* format.

Parameters

- **file_name** (*str*) – Name to save .vtk file
- **local_cell_data** (*dict*) – Local cell data (active cells)
- **global_cell_data** (*dict*) – Global cell data (all cells including actnum)
- **vars_data_dtype** (*type*) –
- **export_grid_data** (*bool*) –

init()

Function to initialize the model, which includes: - initialize well (perforation) position - initialize well rate parameters - initialize reservoir initial conditions - initialize well control settings - define list of operator interpolators for accumulation-flux regions and wells - initialize engine

load_restart_data(*filename: str = 'restart.pkl'*)

Function to load data from previous simulation and uses them for following simulation. :param filename: restart_data filename :type filename: str

output_properties()

Function to return array of properties. Primary variables (vars) are obtained from engine, secondary variables (props) are interpolated by property_itor.

Returns
property_array

Return type
np.ndarray

print_stat()

Function to print the statistics information, including total timesteps, Newton iteration, linear iteration, etc..

print_timers()

Function to print the time information, including total time elapsed, time consumption at different stages of the simulation, etc..

reset()

Function to initialize the engine by calling 'physics.engine.init()' method.

run(*days: float | None = None*)

run_python(*days: float*, *restart_dt: float = 0*, *timestep_python: bool = False*)

run_timestep_python(*dt*, *t*)

save_restart_data(*filename: str = 'restart.pkl'*)

Function to save the simulation data for restart usage. :param filename: Name of the file where restart_data stores. :type filename: str

set_boundary_conditions()

Function to set boundary conditions. Passes boundary conditions to `Physics` object and wells.

This function is virtual in DartsModel, needs to be defined in child Model.

set_initial_conditions()

Function to set initial conditions. Passes initial conditions to `Physics` object.

This function is virtual in DartsModel, needs to be defined in child Model.

set_op_list()

Function to define list of operator interpolators for accumulation-flux regions and wells.

Operator list is in order [acc_flux_itor[0], ..., acc_flux_itor[n-1], acc_flux_w_itor]

set_physics()

Function to define properties and regions and initialize `Physics` object.

This function is virtual in DartsModel, needs to be defined in child Model.

set_sim_params(*first_ts: float | None = None*, *mult_ts: float | None = None*, *max_ts: float | None = None*, *runtime: float = 1000*, *tol_newton: float | None = None*, *tol_linear: float | None = None*, *it_newton: int | None = None*, *it_linear: int | None = None*, *newton_type=None*, *newton_params=None*)

Function to set simulation parameters.

Parameters

- **first_ts** (*float*) – First timestep
- **mult_ts** (*float*) – Timestep multiplier

- **max_ts** (*float*) – Maximum timestep
- **runtime** (*float*) – Total runtime in days, default is 1000
- **tol_newton** (*float*) – Tolerance for Newton iterations
- **tol_linear** (*float*) – Tolerance for linear iterations
- **it_newton** (*int*) – Maximum number of Newton iterations
- **it_linear** (*int*) – Maximum number of linear iterations
- **newton_type** –
- **newton_params** –

set_wells()

Function to define wells and initialize `Reservoir` object.

This function is virtual in DartsModel, needs to be defined in child Model.

CHAPTER

TEN

RESERVOIRS

class darts.models.reservoirs.struct_reservoir.**StructReservoir**(*timer*, *nx: int*, *ny: int*, *nz: int*, *dx*, *dy*, *dz*, *permx*, *permy*, *permz*, *poro*, *depth*, *actnum=1*, *global_to_local=0*, *op_num=0*, *coord=0*, *zcorn=0*, *is_cpg=False*)

Bases: object

__init__(*timer*, *nx: int*, *ny: int*, *nz: int*, *dx*, *dy*, *dz*, *permx*, *permy*, *permz*, *poro*, *depth*, *actnum=1*, *global_to_local=0*, *op_num=0*, *coord=0*, *zcorn=0*, *is_cpg=False*)

Class constructor method

Parameters

- **timer** – timer object to measure discretization time
- **nx** – number of reservoir blocks in the x-direction
- **ny** – number of reservoir blocks in the y-direction
- **nz** – number of reservoir blocks in the z-direction
- **dx** – size of the reservoir blocks in the x-direction (scalar or vector form) [m]
- **dy** – size of the reservoir blocks in the y-direction (scalar or vector form) [m]
- **dz** – size of the reservoir blocks in the z-direction (scalar or vector form) [m]
- **permx** – permeability of the reservoir blocks in the x-direction (scalar or vector form) [mD]
- **permy** – permeability of the reservoir blocks in the y-direction (scalar or vector form) [mD]
- **permz** – permeability of the reservoir blocks in the z-direction (scalar or vector form) [mD]
- **poro** – porosity of the reservoir blocks
- **actnum** – attribute of activity of the reservoir blocks (all are active by default)
- **global_to_local** – one can define arbitrary indexing (mapping from global to local) for local arrays. Default indexing is by X (fastest),then Y, and finally Z (slowest)
- **op_num** – index of required operator set of the reservoir blocks (the first by default). Use to introduce PVTNUM, SCALNUM, etc.
- **coord** – COORD keyword values for more accurate geometry during VTK export (no values by default)
- **zcron** – ZCORN keyword values for more accurate geometry during VTK export (no values by default)

add_perforation(*well*, *i*, *j*, *k*, *well_radius=0.1524*, *well_index=-1*, *well_indexD=-1*, *segment_direction='z_axis'*, *skin=0*, *multi_segment=True*, *verbose=False*)

well_indexD - thermal well index (for heat loss through the wellbore) if -1, use computed value based on cell geometry; if 0 - no heat losses

add_well(*name*, *wellbore_diameter=0.15*)

export_vtk(*file_name*, *t*, *local_cell_data*, *global_cell_data*, *export_constant_data=True*)

generate_cpg_vtk_grid()

generate_vtk_grid(*strict_vertical_layers=True*, *compute_depth_by_dz_sum=True*)

get_cell_cpg_widths()

get_cell_cpg_widths_new()

get_well(*well_name*)

find well by name :param well_name: :return: well object

init_wells()

set_boundary_volume(*xy_minus=-1*, *xy_plus=-1*, *yz_minus=-1*, *yz_plus=-1*, *xz_minus=-1*, *xz_plus=-1*)

class darts.models.reservoirs.unstruct_reservoir.**UnstructReservoir**

Bases: object

add_perforation(*well*, *res_block*, *well_index=-1*, *well_indexD=-1*, *well_radius=0.1524*, *skin=0.0*, *multi_segment=True*, *verbose=False*)

Class method which ads perforation to each (existing!) well

Parameters

- **well** – data object which contains data of the particular well
- **res_block** – reservoir block in which the well has a perforation
- **well_index** – well index (productivity index)

Returns

add_well(*name*, *depth*)

Class method which adds wells heads to the reservoir (Note: well head is not equal to a perforation!)

Parameters

- **name** –
- **depth** –

Returns

calc_boundary_cells(*boundary_data*)

Class method which calculates constant boundary values at a specif constant x,y,z-coordinate

Parameters
boundary_data – dictionary with the boundary location (X,Y,Z, and location)

Returns

CHAPTER

ELEVEN

DISCRETIZER

11.1 Structured Discretizer

class darts.mesh.struct_discretizer.**StructDiscretizer**(*nx*, *ny*, *nz*, *dx*, *dy*, *dz*, *permx*, *permy*, *permz*, *global_to_local=0*, *coord=0*, *zcorn=0*, *is_cpg=False*)

__init__(*nx*, *ny*, *nz*, *dx*, *dy*, *dz*, *permx*, *permy*, *permz*, *global_to_local=0*, *coord=0*, *zcorn=0*, *is_cpg=False*)

Class constructor method.

Parameters

- **nx** – number of reservoir blocks in the x-direction
- **ny** – number of reservoir blocks in the y-direction
- **nz** – number of reservoir blocks in the z-direction
- **dx** – size of the reservoir blocks in the x-direction (scalar or vector form) [m]
- **dy** – size of the reservoir blocks in the y-direction (scalar or vector form) [m]
- **dz** – size of the reservoir blocks in the z-direction (scalar or vector form) [m]
- **permx** – permeability of the reservoir blocks in the x-direction (scalar or vector form) [mD]
- **permy** – permeability of the reservoir blocks in the y-direction (scalar or vector form) [mD]
- **permz** – permeability of the reservoir blocks in the z-direction (scalar or vector form) [mD]
- **global_to_local** – one can define arbitrary indexing (mapping from global to local) for local arrays. Default indexing is by X (fastest),then Y, and finally Z (slowest)

calc_cpg_discr()

Class methods which performs the actual construction of the connection list

Return cell_m
minus-side of the connection

Return cell_p
plus-side of the connection

Return tran
transmissibility value of connection

Return tran_thermal
geometric coefficient of connection

`calc_structured_discr()`

Class methods which performs the actual construction of the connection list

Return cell_m
minus-side of the connection

Return cell_p
plus-side of the connection

Return tran
transmissibility value of connection

Return tran_thermal
geometric coefficient of connection

`calc_volumes()`

Class method which reshapes the volumes of all the cells to a flat array (Ntot x 1)

Returns
flat volume array (Ntot x 1)

`calc_well_index(i, j, k, well_radius=0.1524, segment_direction='z_axis', skin=0)`

Class method which construct the well index for each well segment/perforation

Parameters

- **i** – "human" counting of x-location coordinate of perforation
- **j** – "human" counting of y-location coordinate of perforation
- **k** – "human" counting of z-location coordinate of perforation
- **well_radius** – radius of the well-bore
- **segment_direction** – direction in which the segment perforates the reservoir block
- **skin** – skin factor for pressure loss around well-bore due to formation damage

Return well_index
well-index of particular perforation

`convert_to_3d_array(data, data_name: str)`

Class method which converts the data object (scalar or vector) to a true 3D array (Nx,Ny,Nz)

Parameters

- **data** – any type of data, e.g. permeability of the cells (scalar, vector, or array form)
- **data_name** – name of the data object (e.g. 'permx')

Return data
true data 3D data array

`convert_to_flat_array(data, data_name: str)`

Class methods which converts data object from any type to true flat array of size (Ntot x 1)

Parameters

- **data** – data object of any type (e.g. permeability of the cells) (scalar, vector, or 3D array form)
- **data_name** – name of the data object (e.g. 'permx')

Return data
true flat array (Ntot x 1)

static dump_connection_list(*cell_m*, *cell_p*, *conn*, *filename*)

Static method which dumps the connection list to a file

Parameters

- **cell_m** – minus-side of the connection
- **cell_p** – plus-side of the connection
- **conn** – transmisibility value of connection
- **filename** – name of the desired file

Returns

11.2 Unstructured Discretizer

class darts.mesh.unstruct_discretizer.**UnstructDiscretizer**(*permx*, *permy*, *permz*, *frac_aper*, *mesh_file: str*, *poro=0.2*, *num_matrix_cells=0*, *num_fracture_cells=0*, *num_well_cells=0*, *verbose=False*)

__init__(*permx*, *permy*, *permz*, *frac_aper*, *mesh_file: str*, *poro=0.2*, *num_matrix_cells=0*, *num_fracture_cells=0*, *num_well_cells=0*, *verbose=False*)

Class constructor method

Parameters

- **permx** – permeability data object (either in scalar or vector form) in x-direction
- **permy** – permeability data object (either in scalar or vector form) in y-direction
- **permz** – permeability data object (either in scalar or vector form) in z-direction
- **frac_aper** – fracture aperture data object (either in scalar or vector form)
- **mesh_file** – name of the mesh file (in string form)
- **num_matrix_cells** – number of matrix cells, if known before hand! (in scalar form)
- **num_fracture_cells** – number of fracture cells, if known before hand! (in scalar form)

calc_boundary_cells(*boundary_data*)

Class method which calculates constant boundary values at a specif constant x,y,z-coordinate

Parameters
boundary_data – dictionary with the boundary location (X,Y,Z, and location)

Returns

calc_boundary_cells_new(*boundary_data*)

Class method which calculates constant boundary values at a specif constant x,y,z-coordinate

Parameters
boundary_data – dictionary with the boundary direction (x,y, or z) and type (min or max)

Returns

calc_cell_information(*cache=0*)

Class method which calculates the geometrical properties of the grid

`calc_cell_information_with_wells`(*well_locations*, *well_radii*, *cache=0*)

Class method which calculates the geometrical properties of the grid

`calc_connections_all_cells`(*cache=0*)

Class methods which calculates the connection list for all cell types (matrix & fracture)

Return cell_m
minus-side of the connection

Return cell_p
plus-side of the connection

Return tran
transmissibility value of connection

Return tran_thermal
geometric coefficient of connection

`calc_equivalent_well_index`(*res_block: int*, *well_radius: float = 0.1524*, *skin: float = 0.0*) → List[float]

works only for wedge 2.5D extruded cells approximate calculation: triangle -> square with the same area -> Peaceman formula :param res_block: cell block index :param well_radius: well radius, m. :param skin: skin :return: well_index, well_index_thermal

`check_fracture_data_input`(*data*, *data_name: str*)

Class method which checks the input data for fracture cells

Parameters

- `data` – scalar or vector with data
- `data_name` – string which represents the data

Returns
correct data object size

`check_matrix_data_input`(*data*, *data_name: str*)

Class method which checks the input data for matrix cells

Parameters

- `data` – scalar or vector with data
- `data_name` – string which represents the data

Returns
correct data object size

`load_mesh`(*cache=0*)

Class method which loads the mesh data of a specified file, using the module meshio module (third party).

`load_mesh_with_wells`(*well_centers*, *well_radii*, *cache=0*)

Class method which reads msh file with meshio or reads mesh_data from cache

`store_centroid_all_cells`()

Class method which loops over all the cells and stores the volume in single array (first frac, then mat) :return:

`store_depth_all_cells`()

Class method which loops over all the cells and stores the depth in single array (first frac, then mat) :return:

store_volume_all_cells()

Class method which loops over all the cells and stores the volume in single array (first frac, then mat) :return:

static write_conn2p_therm_to_file(*cell_m*, *cell_p*, *tran*, *tranD*, *file_name*)

Static method which write a connection list to the specified file (for thermal application)

Parameters

- **cell_m** – negative residual contribution of cell block connections of interface
- **cell_p** – positive residual contribution of cell block connections of interface
- **tran** – transmissibility value of the interface
- **tranD** – geometric coefficient of interface
- **file_name** – file name where to write connection list

Returns

static write_conn2p_to_file(*cell_m*, *cell_p*, *tran*, *file_name*)

Static method which write a connection list to the specified file (for non-thermal application)

Parameters

- **cell_m** – negative residual contribution of cell block connections of interface
- **cell_p** – positive residual contribution of cell block connections of interface
- **tran** – transmissibility value of the interface
- **file_name** – file name where to write connection list

Returns

write_depth_to_file(*file_name*)

Class method which loops over all the cells and writes the volume into a file (first frac, then mat) :return:

static write_property_to_file(*data*, *key_word: str*, *file_name: str*, *num_cells: int*)

Static method which writes any specified property (in data) to any specified file

Parameters

- **data** – data object required to write to a file
- **key_word** – keyword (usually read by other simulator)
- **file_name** – name of the file where to write
- **num_cells** – number of reservoir blocks

Returns

write_to_vtk(*output_directory*, *property_array*, *cell_property*, *ith_step*)

Class method which writes output of unstructured grid to VTK format

Parameters

- **output_directory** – directory of output files
- **property_array** – np.array containing all cell properties (N_cells x N_prop)
- **cell_property** – list with property names (visible in ParaView (format strings)
- **ith_step** – integer containing the output step

Returns

write_volume_to_file(*file_name*)

Class method which loops over all the cells and writes the volume into a file (first frac, then mat) :return:

BIBLIOGRAPHY

[1] M. Khait. *Delft Advanced Research Terra Simulator: General Purpose Reservoir Simulator with Operator-Based Linearization*. PhD thesis, TU Delft, 2019. doi:10.4233/uuid:5f0f9b80-a7d6-488d-9bd2-d68b9d7b4b87.

[2] D. V. Voskov. Operator-based linearization approach for modeling of multiphase multi-component flow in porous media. *Journal of Computational Physics*, 337:275–288, 2017. doi:10.1016/j.jcp.2017.02.041.

[3] M. Khait and D. V. Voskov. Operator-based linearization for efficient modeling of geothermal processes. *Geothermics*, 74:7–18, 2018. doi:10.1016/j.geothermics.2018.01.012.

[4] M. Khait and D. V. Voskov. Adaptive parameterization for solving of thermal/compositional nonlinear flow and transport with buoyancy. *SPE Journal*, 23:522–534, 2018. doi:10.2118/182685-PA.

[5] M. Khait and D. V. Voskov. Operator-based linearization for general purpose reservoir simulation. *Journal of Petroleum Science and Engineering*, 157:990–998, 2017. doi:10.1016/j.petrol.2017.08.009.

[6] X. Lyu, M. Khait, and D. Voskov. Operator-based linearization approach for modelling of multiphase flow with buoyancy and capillarity. *SPE Journal*, pages 1–18, 2021. doi:https://doi.org/10.2118/205378-PA.

[7] M. Khait and D. V. Voskov. Integrated framework for modelling of thermal-compositional multiphase flow in porous media. In *SPE Reservoir Simulation Conference*. 2019.

PYTHON MODULE INDEX

d

INDEX

Symbols

A

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

W