Complex Systems Platform

CSP3D3.0

User’s Guide

copyright (c) S.K. Matthäi, S. Geiger, and S.G. Roberts (2000)
# Table of Contents

## CSP3D3.0 User’s Guide (Matthäi et al.)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. WHAT IS CSP ?</td>
<td>1</td>
</tr>
<tr>
<td>Design Philosophy</td>
<td>5</td>
</tr>
<tr>
<td>Design History</td>
<td>6</td>
</tr>
<tr>
<td>Getting Started</td>
<td>7</td>
</tr>
<tr>
<td>2. A MODEL OF A REAL WORLD SYSTEM EXPRESSED IN CSP</td>
<td>9</td>
</tr>
<tr>
<td>1. CSP_MeshManager: The Finite-Element Mesh</td>
<td>12</td>
</tr>
<tr>
<td>2. PropertyDatabase for Physical Variables</td>
<td>15</td>
</tr>
<tr>
<td>3. Scalar-, Vector-, and TensorVariable: CSP Basic Data Types</td>
<td>16</td>
</tr>
<tr>
<td>4. CSP_MemoryManager: Storage and Access of Physical Variables</td>
<td>17</td>
</tr>
<tr>
<td>3. SIMULATION OF A STEADY-STATE FLUID PRESSURE DISTRIBUTION</td>
<td>19</td>
</tr>
<tr>
<td>SuperGroup</td>
<td>26</td>
</tr>
<tr>
<td>Group</td>
<td>27</td>
</tr>
<tr>
<td>SmnErrorHandler &amp; Global Variables</td>
<td>28</td>
</tr>
<tr>
<td>CSP_Operands</td>
<td>30</td>
</tr>
<tr>
<td>Operands</td>
<td>30</td>
</tr>
<tr>
<td>Interrelations</td>
<td>33</td>
</tr>
<tr>
<td>Visitors</td>
<td>36</td>
</tr>
<tr>
<td>(2D and 3D) Algorithms</td>
<td>37</td>
</tr>
<tr>
<td>CSP_Algorithms for Linear Triangular Elements</td>
<td>39</td>
</tr>
<tr>
<td>4. CSP INPUT &amp; OUTPUT</td>
<td>42</td>
</tr>
<tr>
<td>Regular Meshes from Pixel Data</td>
<td>42</td>
</tr>
<tr>
<td>1. Drawing the Geologic Cross Section</td>
<td>42</td>
</tr>
<tr>
<td>2. Importing the Section into NIH Image</td>
<td>43</td>
</tr>
<tr>
<td>3. Color Coding the Image</td>
<td>43</td>
</tr>
<tr>
<td>4. Output of the section as color-coded text file</td>
<td>44</td>
</tr>
<tr>
<td>Spatially-Varially Refined Meshes from PSLG’s</td>
<td>45</td>
</tr>
<tr>
<td>3D Meshes generated by GoCud</td>
<td>48</td>
</tr>
<tr>
<td>From Rhino</td>
<td>48</td>
</tr>
<tr>
<td>CSP Geometry Preprocessor</td>
<td>49</td>
</tr>
<tr>
<td>In Gocad</td>
<td>50</td>
</tr>
<tr>
<td>Caveats</td>
<td>51</td>
</tr>
<tr>
<td>3D Mixed Tetrahedral &amp; Triangular Element Meshes From GoCud</td>
<td>51</td>
</tr>
<tr>
<td>IMPLEMENTATION (SEE EXAMPLE 14.CPP)</td>
<td>52</td>
</tr>
<tr>
<td>Output via the VTK Interface</td>
<td>53</td>
</tr>
<tr>
<td>Output via the JPEG Interface</td>
<td>53</td>
</tr>
<tr>
<td>5. INITIAL AND ESSENTIAL CONDITIONS</td>
<td>54</td>
</tr>
<tr>
<td>6. DEFINING &amp; SOLVING FINITE-ELEMENT EQUATIONS</td>
<td>59</td>
</tr>
<tr>
<td>Example: Steady-state fluid pressure</td>
<td>61</td>
</tr>
<tr>
<td>Transient fluid pressure</td>
<td>65</td>
</tr>
<tr>
<td>Buoyancy-Driven Flow</td>
<td>66</td>
</tr>
<tr>
<td>Lumping</td>
<td>66</td>
</tr>
<tr>
<td>References Cited</td>
<td>67</td>
</tr>
<tr>
<td>7. THE CSP LIBRARIES</td>
<td>68</td>
</tr>
<tr>
<td>Interrelations “Seed” Library</td>
<td>68</td>
</tr>
<tr>
<td>CSP Algorithms Library</td>
<td>72</td>
</tr>
<tr>
<td>Velocity &amp; InterstitialVelocity</td>
<td>72</td>
</tr>
<tr>
<td>Rock Type</td>
<td>75</td>
</tr>
<tr>
<td>Buoyancy-Driven Fluid Flow</td>
<td>79</td>
</tr>
<tr>
<td>LinearElasticity</td>
<td>81</td>
</tr>
<tr>
<td>Visitors “Seed” Library</td>
<td>85</td>
</tr>
<tr>
<td>BoundaryTransferVisitor (2D only)</td>
<td>85</td>
</tr>
<tr>
<td>InterfaceVisitor (2D only)</td>
<td>85</td>
</tr>
<tr>
<td>CSP_Operands</td>
<td>90</td>
</tr>
<tr>
<td>SmnErrorHandler &amp; Global Variables</td>
<td>90</td>
</tr>
<tr>
<td>CSP_Operands</td>
<td>90</td>
</tr>
<tr>
<td>CSP_Operands</td>
<td>90</td>
</tr>
<tr>
<td>CSP_Operands</td>
<td>90</td>
</tr>
<tr>
<td>CSP_Operands</td>
<td>90</td>
</tr>
</tbody>
</table>
# Table of Contents

- **CSP_ID_Visitor** ................................................................. 86
- **AlterationVisitor** ............................................................ 86
- **HeatTransportVisitor (2D only)** ........................................ 87
- **H20PropertiesVisitor** ....................................................... 88
- **HydroFractureVisitor (2D only)** ........................................ 88
- **TransportVisitor (2D only)** ............................................... 90
- **VelocityAndVolumeFluxVisitor** ........................................ 92
- **FiniteVolumeAdvectionVisitor (2D only)** ......................... 92

**Finite Element Library** ......................................................... 92
- **The Linear Triangular Element: LinearTriangle** .................. 93
- **The Linear Tetrahedral Element: LinearTetrahedron** ........ 95
- **The QuadraticTriangle with 6 Nodes** .................................. 98
- **The BarycentricQuadraticTriangle with 7 Nodes** .............. 103
- **The QuadraticTetrahedron with 10 Nodes** ......................... 107

**PDE Operators Library (Algorithms)** ...................................... 109
- **PDE Operators for Numerical Integration** ......................... 114
- **PDE Operators for Analytical Integration** .......................... 111

8. **A COMPREHENSIVE SIMULATION EXAMPLE: CARBONATE DISSOLUTION** ........................................ 117
   - **1. Physical Variables** .................................................... 117
   - **2. Initial values and Boundary Conditions** ...................... 117
   - **3. The Interrelations** .................................................... 118
   - **4. The calculation** ........................................................ 119
   - **5. Solving the Global Equation** ..................................... 119
   - **6. The Groups** ............................................................ 120
   - **7. The Visitors** ............................................................. 120
   - **8. FiniteDifferenceGrids** ............................................. 120
   - **9. Speciation Calculations** ........................................... 121
   - **10. Initial Conditions** ................................................... 122
   - **11. The Overall Model** .................................................. 122

9. **Mesh Pre-Processors** ....................................................... 125
   - **Geometry Utilities** ....................................................... 125

**APPENDIX** ........................................................................ I

**A: The Distribution** .............................................................. I

**B: Installation** ....................................................................... 8
   - **CSP Projects in CodeWarrior** ........................................ 9
   - **Mac / OS ≥8.6** ........................................................... 11
   - **PC / Windows NT4.0** .................................................. 12
   - **Linux** .......................................................................... 12
   - **SGI / Irix 6.5 or Later** ................................................. 12

**Deprecated Capabilities available on Request** ....................... 14

**HDF Interface** ..................................................................... 14

**CSP GLOSSARY** ................................................................... 15

copyright (c) 2000, by Stephan K. Matthäi, Sebastian Geiger, and Stephen G. Roberts

Original object-oriented C++ library (complex systems platform = CSP, current version 3.0) designed and developed in collaboration with Dr. S. G. Roberts (ANU), and Sebastian Geiger (ETH) runs on PC, Mac, and unix workstations and supercomputers.

1. What is CSP?

The Complex Systems Platform (CSP) is a collection of C++ objects (software modules) designed for the simulation of geological processes and their interactions over time and in two or three spatial dimensions. CSP was developed as a cross-platform toolkit for the PC, Mac, or unix-linux hardware environments. The CSP code reflects the ANSI / ISO C++ standard (July 1998) and compiles with the latest GNU, CodeWarrior, KAI and other compilers capable of compiling the compiler test program suite of the Blitz++ computational library by T. Verhuizen. Thus, an operational C++ compiler of this group forms the prerequisite for using CSP.

CSP is essentially a finite-element based computational environment, but finite volume, finite-difference and sporadic particle-dynamics capabilities are implemented as well. As is necessitated by the complexity of typical geological systems, abstract data types (ADT’s), tree-type search capilities and boolean operators are given the same importance as the classical numerical methods in CSP. This gives CSP what one could regard as GIS capabilities. In terms of C++ intricacies, CSP makes extensive use of the C++ Standard Template Library (STL) and the Meschach++ library for matrix computations. The inversion of large (>1000 x 1000) (sparse) solution matrices can be carried out, among other available methods, with an algebraic multigrid method (AMG Ruge & Stueben 1987). In this case, matrix-inversion time relates to matrix size in a \( n \log n \) fashion \( h = \text{rows} = \text{columns} \) as opposed to \( n^2 \) for more typically used methods.

Input-output parameter-range checking for all modules simulating process interactions ascertains the correctness of complex calculations and, in combination with a C++ exception handling scheme, permitts the precise location of range violations, as well as sometimes even runtime error recovery. A partitioning of the
geological models into subdomains which can be treated individually in computations, also facilitates convenient and highly selective data input and output from very large meshes. A graphical user-interface for CSP is in preparation.

We have designed CSP with geological applications in mind. Given high-quality input meshes, CSP performs fast and reliably in simulations in which material properties vary over many orders of magnitude both in time and spatially. Models with a high spatial resolution (for example, highly permeable cm-wide fractures in km-thick low-permeability rocks) and highly variable material properties have been computed successfully also when non-linear equations of state and pressure and temperature-dependent physical variables were incorporated. In CSP, partial differential equations are typically embedded into Algorithm objects combining finite-element forms of differential operators and associating them with physical variables as if one writes a partial differential equation.

Geological applications necessitate the solution of weakly to fully coupled equations, as well as post-processing computations of physical variables that depend on the dependent variables of the global system of equations. These partially dependent physical variables track component processes in the geological system such as compaction or dehydration. CSP encapsulates such interactions in process modules called Interrelations. Interrelations can be tested and verified in isolation. In complex runs, exceptions are raised when Interrelation modules receive out-of-range input or when they produce erratic output. Exceptions carry the Interrelation method and Interrelation object names as well as specific error information. Interrelations are managed via an InterrelationManager which checks mutual dependencies of Interrelations, determining the best possible execution queue for approval by the user. This capability greatly reduces the development time for new complex simulations.

Interrelations only compute single variables, whereas Visitors objects in CSP can compute a range of variables at once such as the PT-dependent properties of water.

To directly modify, combine, or calculate distributed physical variables, one can associate variables with CSP_Operand objects. They have overloaded interfaces which permit to add, subtract, and multiply Operands, to create temporary variables at runtime, and to address Group regions selectively.

Component processes may be restricted to specific regions of a model. Certain processes affect rock units only above a certain temperature / pressure etc., relative permeabilities only need to be calculated where multiple phases are present, and thermal volume strain must only be calculated where the rock cools or is heated. For these reasons, CSP allows restrict computations to groups of finite elements. In CSP, a Group is a mesh area (2D) or a mesh volume (3D) of finite elements which are group members on the basis of their spatial coordinates, material properties, property ranges or combinations thereof. With Group-restricted computations, any computation that can be applied to the entire model is also possible for a region thereof. An Observer module tells adjacent regions to update themselves when dependent variables at group boundaries change by more than a specified tolerance. This domain decomposition capability allows for spatially variable time-stepping (certain regions can be solved more often than others), for distributed computations (parallel or vector processing), or for efficient memory swapping. Solution matrices
and right-hand vectors may also be output at runtime for message passing to and from other computers. For example, matrices can be assembled on the front end- for inversion on a high-performance-computer back end.

Any computed group variable can also be output for visualization. In 3D, this allows to greatly reduce the number of tetrahedra that must be rendered and helps to selectively visualize features of interest. In 2D models, computed variables can also be output onto regular grids for faster rendering and animation.

CSP implements two and three-dimensional finite-element models. For the simple generation of 2D model, an integrated meshing module that creates regular triangle meshes from color-coded pixel images is available. Planar, straight line graphs (PSLG) containing polygons, lines, and regional property information can also be meshed using constrained, conforming Delauney triangulations. Triangle and tetrahedral meshes with attached material properties can be supplied directly to CSP as HDF Vsets, ICEMTetra (NASTRAN) meshes, or in GoCad TSurf and TSolid files. In any of these formats, triangle / tetrahedra can be tagged to specify which geological objects they belong to (volumes in 3D). CSP will also automatically identify element neighbors and outer model or group boundaries.

CSP also comprises a PropertyDatabase containing the physical variables for a specific simulation. This database can be initialized from a text file or MS Excel spreadsheet, but variables can also be added dynamically at runtime. Importantly, for fast property access, storage for each variable is allocated as contiguous heap memory. This storage can shrink or grow when the user decides to remesh the model at runtime.

The current version of CSP (CSP3D3.0) represents an API (application programmer interface) and class library for the assembly of finite-element computations. This means that users must write their own main programs to implement customised CSP computations. To learn how to use CSP, the user will start by customizing existing CSP models according to his needs. A wide range of such example simulation programs is provided with CSP3D. Thus far these examples include:

- Steady-state and transient topography-, pressure- and-or buoyancy driven fluid flows including fluid sources and sinks and spatially highly variable material properties (2D and 3D).
- Advection of heat and solutes or reactants with the flow (2D).
- Thermohaline convection (dependent on the choice of equation of state) (2D).
- Chemical diffusion (2D and 3D).
- Gradient reactions along the fluid flow path, e.g. quartz precipitation (2D).
- Relatively simple rate-controlled reactive transport processes, like calcite dissolution etc. (2D).
- Two-phase fluid flow of chemically inert and only slightly compressible phases (2D).
• Hydrofracture (2D).

• Linear-elastic deformation of a heterogeneous rock mass (2D and 3D).

• Combinations of the above processes plus a variety of interdependencies among physical variables (2D, some 3D).

CSP also provides a large number of utility classes for geometric manipulations of models and data (for instance polygons, triangles, line segments and points, DXF file exchange etc.), for the pre-processing of input data and for the communication with other computational tools. There are a number of interfaces which connect CSP to the Visualization Toolkit (VTK, kitware.com), GoCad, Tekplot etc.
DESIGN PHILOSOPHY

CSP adheres to an object-oriented design with the goal to disassemble the complexity of geological and other processes into components that are small enough that we can understand and express them mathematically, verify their accurate numerical behaviour, and ascertain that the interaction of such modular components occurs within parameter value ranges for which the behavior of the interacting processes has been tested. While we can understand component processes in isolation, we cannot predict how these will interact or what system behavior they may induce, especially when material properties have values which vary over many orders of magnitude. To facilitate the simulation of such complex interactions and the resulting overall system behaviour has been the main design objective of the CSP code. CSP therefore is a tool for simulation research on complex earth-, engineering-, and other systems and lends itself to:

- Investigate how processes that are well understood in isolation will interact,
- Investigate processes that cannot be observed directly because they occur over very long spans of time or at a great depth,
- Interpret the meaning of geological observations and constraints,
- Evaluate proposed cause-and-effect chains in complex geological systems,
- Carry out sensitivity analyses of how geological-system behavior changes in response to critical variables,
- Determine which geological data should be acquired to constrain specific processes,
- Forecast how man-made changes to natural systems are likely to affect their future behavior.
DESIGN HISTORY

A first draft of CSP was implemented by Stephan K. Matthäi (1995), while studying sedimentary basin-evolution with the Fortran code Akcess basin as a postdoc of Larry Cathles at the Department of Geological Sciences, Cornell University, Ithaca, New York. In June 1995, Stephan joined the Rock Fracture Project at Stanford University to work on fracture-flow processes in sandstones under the leadership of Atilla Aydin and Dave D. Pollard. In September 1995, Stephen G. Roberts (PhD, Berkeley 1986) from the School of Mathematical Sciences at the Australian National University, Canberra, dedicated his sabbatical to the joint development of CSP. From the foundations laid in this period of close collaboration at Stanford University, a first 2D version of CSP emerged. This implementation was subsequently revised and extended following suggestions of Bruce Eckel (e.g., “Thinking in C++”, Prentice Hall), and has since been applied on a variety of research problems (see attached publication list).

In October 1996, S.K. Matthäi joined Christoph A. Heinrich’s Mineral Resource research group at the Swiss Federal Institute of Technology (ETH). He extended CSP to incorporate linear-elasticity, buoyancy-driven fluid flow, two-phase flow of slightly compressible liquids, a first order transport scheme for reactants with minimal-numerical-diffusion, gradient reactions and reactive transport processes. In December 1996, a collaboration with ICEMCFD (a finite-element meshing provider) commenced and triangular-element meshes generated with their program Tetra were employed in 2D reservoir simulations (Matthäi et al. 1998, SPEJ Journal). In spring 1998, Stephan visited Stephen Roberts at the Australian National University to port CSP to three dimensions and to implement a second order chemical transport scheme. This collaboration facilitated the current 3D version of CSP.

In the beginning of 1999, Sebastian Geiger and Thomas Driesner have joined the CSP User and later the Developer’s team. Sebastian has aided in the implementation of an alteration visitor to simulate reactive transport in complex geochemical systems and is currently improving and implementing numerical methods for solving two-phase flow equations. Thomas is currently implementing robust equations of state for water-NaCl mixtures in the single and two-phase region.

In the year 2000, Sebastian Geiger came to the ETH, and has since contributed a first order finite volume advection scheme to CSP. He has also helped to greatly extend the finite – element library.

In collaboration with David Campagna (William Mary College, Virginia), particle dynamics methods for the simulation of the formation of discrete fractures are being developed.
GETTING STARTED

Chapter Overview: The following chapters cover:

Chapter 2: A physical simulation problem expressed as a CSP object model.

Chapter 3: Illustrative example of how a CSP simulation can be structured in a `main()` program, using an example from fluid flow.

Chapter 4: Alternatives building finite-element models from own data and export of results to various visualization tools.

Chapter 5: Assignment of initial and boundary (essential) conditions to CSP models.

Chapter 6: Finite-element forms of partial differential operators and related C++ classes.

Chapter 7: CSP3D3.0 libraries containing subclasses derived from the base classes of the main library. This chapter comprises an introduction to specific Interrelations, Algorithms, Visitors, FiniteElement, and PDE Operator subclass objects which can be directly applied to the physical problems of interest. The description of the Algorithm library is intertwined with an introduction to modeling fluid flow and linear elastic deformation of porous media.

Chapter 8: FiniteVolume library for conservative transport simulations.

Chapter 9: Example of a reactive-transport simulation for the dissolution of carbonate from a fractured limestone.

Chapter 10: Auxiliary programs for mesh geometry pre-processing and visualization of model output.

Appendix. Classes contained in the CSP3D distribution are tabulated in Appendix A and installation advice and internet ftp sites for the support libraries are described in Appendix B. Finally, a Glossary of terms aids the user in learning the object-oriented- and the CSP terminology.

The overall documentation of the CSP3D class library i.e., application programmer interface (API) is structured into three segments, the User’s Guide (this document), the Developer’s Guide, and the Reference Manual (in HTML). The CSP User’s Guide (this document) describes the object modules and interdependencies in the CSP class library. The Developer’s Guide treats the CSP library and finite elements in depth and makes recommendations how specific applications could be built using it. Importantly, many modern C++ programming tools now integrate class browsers which allow a user to quickly examine class hierarchies and the public class interfaces. We recommend such a browser to ease the transition to CSP.

The objects themselves are documented in the CSP Reference Manual which should always be open in a Browser window while code is being developed.
The following second chapter describes the basic object hierarchy of CSP and explains the conventions adhered to in writing this documentation. The better a documentation is, the less one has to know about conventions, since intuition will go a long way in interpreting correctly what is missing. While this was an objective which I tried to achieve, there remain essentials which a CSP user needs to know before starting:

1. The usage of the C++ language imposes its terminology on CSP. One should be familiar with object-oriented terminology including expressions like class and subclass, interface, method, inheritance, polymorphism, object instantiation, operator overloading etc. to make use of this documentation. Consult a C++ book and the UML Reference Manual (Rumbaugh et al. 1999, Addison & Wesley) to learn about OOP basics. Use the CSP Glossary after the Appendix for the most commonly used expressions in CSP.

2. For people used to procedural programming, perhaps the most important principle to appreciate in C++ is that objects have a lifecycle. It begins with the object construction also called instantiation, since a named “instance” of the object is created. The construction is performed by a constructor function which initializes the object’s internal data and storage. The lifecycle of an object ends with its destruction, when the memory required by the object is returned back to the operating system.

3. The diagrams used in this documentation are all drawn using the “unified modeling language” (UML), including “class”, “sequence”, “interaction”, and “collaboration” diagrams. The overview chapter of the aforementioned UML manual is therefore highly recommended reading.

4. In class listings, combined but capitalized nouns illustrate the convention used for class names in CSP: In class or method names, all words are spelled out but chained together. Where two capital letters would meet, they are separated by an underscore. Classes which will be mentioned like SpMat, Mat, and Vec do not adhere to this naming convention. Usually this indicates that such classes do not belong to CSP, but to support libraries like the Meschach++ library.

5. In the text, class names are written in boldface when they are first introduced und subsequently typed in courier font.
2. A Model of a Real World System expressed in CSP

This section uses a cross-sectional geologic model (Fig. 1) to explain basic object terminology and object relationships in CSP. Terminology is introduced via the description of the unifying modeling process.

A typical CSP simulation aims at understanding the consequences of a well characterized or hypothetical physical process, or process interactions in nature. In the simplest case, this involves seeking the value of a dependent variable, \( v \) in a domain \( D \), for a set of boundary, essential and initial conditions. This requires (1) a range of simplifications relative to the real-world scenario and process, (2) a bit of housekeeping for the variables and results, and (3) an algorithm that carries out the computation. (4) It also necessitates a scheme which tells the user if something went wrong and permits error recovery. Finite-element computations further require:

1. A discretization of the continuous real-world domain which one wants to model, (mesh in Fig. 1) meaning that the real-world system is only represented at a few material points. For example, a triangular finite-element mesh or a regular grid, can represent key geometrical features of the real-world system.

2. A set of physical variables of which one hopes that it suffices to describe the behavior of interest in the real-world system. This also requires a mapping of (independent) material properties (=physical variables) onto the discretization, for example, a grid point temperature or a mesh cell-elastic modulus.

3. Spatial and temporal coupling mechanisms which relate dependent variables with one another. These are typically basic laws of physics, which apply over the entire problem domain and are expressed as finite-element forms of partial differential equations (PDEs). Integration of these PDEs over the model domain using the finite-element method gives the results of interest.

4. Interrelations among physical variables that apply outside the description of the global coupling mechanism(s) and can be expressed in the smallest units of the discretization, like finite elements. For example, the effective permeability of a mesh cell could be computed as a function of pressure and temperature at the corner points.

5. Essential conditions and initial values of the dependent physical variables for the boundaries of the domain of interest and at the starting time of the computation, respectively.

6. Input and output methods which transform the results of a computation into a human-readable format such as graphics, charts etc.

The CSP code is built in modules organized into hierarchical layers implementing these features and capabilities. The remainder of this chapter introduces the top-level objects of this hierarchy.
Fig. 1: Two-dimensional, cross-sectional model of a geological object illustrating how CSP object names relate to model entities: The road cut is represented by a SuperGroup object with the name “foot hills”. This object is a mesh of triangular elements to which “permeability” and “porosity” are assigned as material (element) properties. Selected sub-regions of the “foot hills” model are identified as Group objects identified by the unique strings “red rock”, “road”, and “joints”. The Group object “joints” is not contiguous since it comprises all joints (black lines) in the jointed unit. It could be used, for instance, to monitor the fluid flux through these joints. The enumeration flags “LEFT, BOTTOM, RIGHT” and “TOP”, identify the model boundaries. The upper boundary (TOP) is flagged as IRREGULAR.

The CSP base-library structure (Fig. 2) is described below with a focus changing from high-level to low level functionality, introducing corresponding objects. On the top level, the user will interact with:

1. The model is represented by a SuperGroup object which contains and manages internally:
   - (If finite elements are used), a geometrical mesh object (CSP_MeshManager) with spatial sub-domains to which different material properties are assigned. The finite-element mesh connectivity imposes a hierarchy of SuperGroup > Group > Element > Node, ConstraintPoint, see Figure 2.
   - A property database generating access keys to physical variables (classes PropertyDatabase, Parameter, CSP_Index, and handles to distributed variables (Operand, CSP_Operand).
   - Property storage (CSP_MemoryManager) tailored according to the specifications stored in the PropertyDatabase and accessed via keys, using the connectivity information of the mesh.
2. Global and element-restricted computations performed with three types of objects listed in an order of increasing complexity:

- **Interrelation** objects for element-restricted computations of single physical variables. *Example:* Computation of the interstitial flow velocity from the Darcy velocity and the porosity at each element.

- **Visitor** objects for more complex but still element-by-element computations such as chemical reactions in the rock volume represented by each element. *Example:* Calculate H2O properties at ambient pressure and temperature.

- **Algorithm** objects for the assembly of finite-element equations into a global sparse matrix and a right-hand vector, and the inversion of this matrix using different solution schemes in collaboration with the classes **Solver**, **FiniteElement**, **SpMat**, **Mat**, **Vec** etc. *Example:* Solve Laplace equation in the domain of interest.

3. Interfaces for the import and export of data to and from computations (classes **MeshInterface**, **FileHandler**, **CSP_VTK_Interface**). *Example:* Output results directly as JPEG images.

4. The user may also resort to CSP internal tools to analyze the results from computations (**CSP_StatisticalAnalyzer**).

![Diagram](image.jpg)

Fig. 2: Basic object relationships in CSP from a viewpoint of the **SuperGroup** object which represents the finite-element mesh and the associated physical variables managed by the **PropertyDatabase**. The SuperGroup contains objects which store the mesh and the associated variables and permits Visitors, Interrelations, and Algorithms to act on these data obeying rules imposed by the SuperGroup. All computations occur in the frame of a `main()` program written by the application programmer or through a future graphical user interface (GUI) of CSP. Results can be visualized with the **CSP_OpenGLViewer** as a complementary graphics tool.
In special cases, one may want to manipulate or compute specific data on regular grids or Voronoi meshes. This can be done using `FiniteDifferenceGrids` or the `FiniteVolumeManager`, respectively.

To add specific functionality to existing objects in the presented object hierarchy, one can always create new subclasses inherited from these objects. Figure 2 shows the SuperGroup object representing the model (cf., Fig. 1) managing the mesh, the physical variables, and the property storage of the simulation.

In the following subsections, the aforementioned CSP objects (modules) are described in more detail.

### 1. CSP.MeshManager: The Finite-Element Mesh

Since CSP is primarily designed for finite-element computations, the SuperGroup contains a `CSP.MeshManager` object which stores the connectivity of the mesh. This section establishes the names of the components of this mesh and describes the corresponding class objects and definitions.

Existing CSP applications work primarily with triangular and tetrahedral element meshes. Mixed triangle-tetra meshes were implemented to represent fractures or narrow fault zones. In the future, prism elements will be supported to better represent thin layers or faults using a minimum number of finite elements.

**Variable placement** on mesh topology (Fig. 3) is defined by the enumeration `PLACEMENT`:

![Fig. 3: Triangular element discretization of folded layers illustrating node- and element numbering and the placement of node and element variables. Grey-shading indicates domains with elevated Young’s modulus.](fig3-mesh-explanation)
enum PLACEMENT { NODE,
    CONSTRAINT_POINT,
    ELEMENT,
    GROUP,
    SUPERGROUP,
    NEIGHBORS,
    UNDEFINED };

Figure 4 illustrates the meaning of this enumeration for different finite-element types.

In finite-element meshes, each Node, Element, and ConstraintPoint has a unique number discriminating it from other objects of its kind. Such IDs can be set and retrieved via the ID() method interface from node, element, and constraint point objects and they are used to determine where an element contribution is placed in global matrices for computations.
For finite-element computations, distributed physical variables need constraint flags such that the solver knows at which mesh locations it can modify values as opposed to locations where fixed variable values must be honored as constraints on a computation. In CSP, these flags are 1-byte DATA_STYLE enumerations. Supported flags are:

```c
enum DATA_STYLE { PLAIN,      // modifyable, dependent or indep. var.
  INIT_GUESS, // not physically meaningful
  INIT_COND,  // physically meaningful
  DIRICH,     // Dirichlet boundary condition
  NEUMANN,    // Neumann boundary condition
  EOS,        // Value from equation of state
  FIELD_DATA, // (Geological) field data
  BOUNDARY,   // internal region boundaries
  ALL, CSP_FIXED, MULTIPLE }
```

At the same node, fluid pressure may be flagged as DIRICH while temperature is flagged PLAIN. Typically, CSP commands modifying variable values, only affect those spatially distributed variables which have a given target flag. This also applies to Algorithms which do not overwrite variables with a flag other than PLAIN (default). The only exceptions to this rule are (1) when one explicitly assigns boundary conditions using the overloaded SuperGroup methods `AssignBoundaryConditions()`, or (2) when transport visitors advect properties on the mesh. These operations will overwrite variables irrespective of their flags.

As shown in Figure 1, the discretized area or volume of the real world system which is represented by the SuperGroup object has boundaries (Fig. 5) identified by capitalized names. These C++ enumeration variables are declared as follows:

```c
enum SG_BOUNDARY { NOT, IRREGULAR = IRREGULAR_OUTSIDE,
  TOP = TOP_OUTSIDE,             
  BOTTOM = BOTTOM_OUTSIDE,       
  LEFT = LEFT_OUTSIDE,           
  RIGHT = RIGHT_OUTSIDE,         
  FRONT = FRONT_OUTSIDE,         
  BACK = BACK_OUTSIDE,           
  CNR1 = CNR_MIN,               
  CNR2 = CNR_MIN_MAXX,           
  CNR3 = CNR_MAX_MAXX,           
  CNR4 = CNR_MAX_MINXZ,          
  CNR5 = CNR_MIN_MAXZ,           
  CNR6 = CNR_MIN_MAXXZ,          
  CNR7 = CNR_MAX,                
  CNR8 = CNR_MAX_MAXZ,           
  EDGE1 = BACK_BOTTOM,           
  EDGE2 = BACK_RIGHT,            
  EDGE3 = BACK_TOP,              
  EDGE4 = BACK_LEFT,             
  EDGE5 = BOTTOM_RIGHT,          
  EDGE6 = TOP_RIGHT,             
  EDGE7 = TOP_LEFT,              
  EDGE8 = BOTTOM_LEFT,           
  EDGE9 = FRONT_BOTTOM,          
  EDGE10 = FRONT_RIGHT,          
  EDGE11 = FRONT_TOP,            
  EDGE12 = FRONT_LEFT }
```

Where the values were defined previously using `#define` directives. What these enumeration strings correspond to is shown in the following figure:
The **boundary flags** comprise labels for 4 or 8 corner elements and nodes in 2 and in 3D, respectively (Fig. 5), and for 12 boundary edges (in 3D). Boundary flags are used to automatically identify and assign boundary conditions to box-shaped models. For irregularly shaped models, boundaries must be defined outside of CSP in the CAD system creating the geometry. The numbering of the corners is counterclockwise following the right-hand coordinate rule and starting from the lower left \((xyz=0)\). If one tries to generate a model with irregularly shaped boundaries using default (box) constructors, boundary assignments attempted by CSP will prompt warnings.

### 2. PropertyDatabase for Physical Variables

Material property management in CSP begins with the generation of a text file. This file may be a spreadsheet output as tab-delimited text. The name of the text file must be communicated to CSP by setting it at the onset or inside the main program. The file’s default name is `CSP_variables.txt`. The user can set an alternative file name by assigning a string to the global variable `global_physvar_textfile`:

```cpp
// put before main()
CSP_String global_physvar_textfile("my_variables.txt");

// put inside main
global_physvar_textfile ="my_variables.txt";
```

The variables text file defines the name of each physical variable, an abbreviation thereof (notation), units which should be SI units (unit conversion utilities are available through the interface of the PropertyDatabase), and a type identifier (either a scalar (=1), vector (=2), or tensor (=3) as represented by the classes ScalarVariable, VectorVariable, or TensorVariable, see next subsection).

<table>
<thead>
<tr>
<th>var.name</th>
<th>notation</th>
<th>unit</th>
<th>type-index</th>
<th>min.</th>
<th>max.</th>
<th>place</th>
<th>usage</th>
</tr>
</thead>
</table>

---

Fig. 5: Boundary flags of a CSP model.
Definitions of physical variables occupy a single line each. This line starts with the variable name with or without white-space, the notation, and the SI unit of the variable. Next, \textit{min. max.} indicate a physically meaningful range for each property. This range is very important, because it gives CSP the possibility to check whether a calculation went wrong, ascertaining that calculation errors are caught where they arise instead of being propagated. CSP always performs a range check, before a computed property value is stored on the mesh. However, it is the responsibility of the user, when defining new calculations, to check input values in order to ascertain that these calculations remains in a tested range.

In the variables text file, the legitimate value range is followed by the desired placement of the variable. Options are the finite elements (element), the constraint points (point) or nodes (node). The other remaining parameters in the text file are ignored by CSP, but are intended as a documentation template for the user.

Physical-variable specifications are stored as Parameter records in the PropertyDatabase inside of the SuperGroup object. The PropertyDatabase can be queried for these records or for variable keys (struct \texttt{CSP\_INDEX} or equivalent \texttt{CSP\_Index} class).

\begin{verbatim}
const PropertyDatabase& p_ref = supergroup.ReferencePropertyDatabase();
p_ref .FlushToScreen( "fluid pressure" );
\end{verbatim}

Keys are later used by the \texttt{CSP\_MemoryManager} to efficiently write and retrieve variable values (Fig. 5). When the PropertyDatabase is first initialized, it flushes all variable names to screen in alphabetical order. It also writes a binary variable file "CSP\_variables.dat". This file is subsequently used for queries during the run instead of the text file.

3. \textbf{SCALAR-, VECTOR-, AND TENSOR-VARIABLE: CSP BASIC DATA TYPES}

Each physical variable in the PropertyDatabase is represented in the model by an STL vector of a suitable data type. Implemented types are ScalarVariable, VectorVariable, and TensorVariable. Corresponding enumerations are contained in the C++ header file "CSP\_definitions.h". To avoid the use of costly RTTI (runtime type identification) at this basic level of computation, the enumeration \texttt{VARIABLE\_TYPE} is defined as additional type discriminator for the three basic classes of variables.\footnote{In C++ one would normally use polymorphism to express variable types using a single base class. However, the need to put the classes into containers and efficiency, preempts the use of inheritance for these low level classes.}

\begin{verbatim}
VARIABLE\_TYPE { SCALAR, VECTOR, TENSOR };
\end{verbatim}

To insure portability of the CSP code between platforms with a different numerical precision, there is a test program 'main\_CSP\_number\_types.cpp' with which the
precision of C++ inbuilt-variable types on the target platform can be established once it has been compiled into an executable. This program writes the header file ‘CSP_number_types.h’ which is automatically included into CSP programs via the header ‘CSP_definitions.h’.

ScalarVariable, VectorVariable, and TensorVariable are classes that associate DATA_TYPE variables with DATA_STYLE enums (described further below). They have overloaded operators that allow the user to write mathematical expressions which are documented in detail in the CSP3D3.0 HTML Reference. Examples of their usage may be the assignment of 3D displacements to a model as vectors or the use tensors to store a stress state. The dimensions of vector and tensor variables are always equivalent to the dimensionality of your model. Thus, a 2D tensor variable contains a $2 \times 2$ matrix of scalar variables.

![Diagram](image.png)

Fig. 6: Steps involved in accessing physical variables stored in the CSP_MemoryManager, using string identifiers such as "permeability", is queried from SuperGroup method, Algorithm, Visitor or Interrelation or SuperGroup->variable name.

4. **CSP_MemoryManager: Storage and Access of Physical Variables**

Depending on their type (scalar, vector or tensor), physical variables are stored in one of three arrays of C++ standard-template-library (STL) vectors differing in size as do the numbers of elements, nodes, and constraint points in the finite-element mesh. This storage structure can grow or shrink at runtime and is contained in the CSP_MemoryManager. It provides a property-access service to element, node, and constraint point objects stored inside the CSP_MeshManager (Fig. 5). For efficient
property access, CSP_INDEX keys are issued by the PropertyDatabase. Once the PropertyDatabase has been initialized there are three ways to access and modify variables:

1. By the name, e.g., “fluid pressure”
2. By the CSP_INDEX, a structure with the members placement, type, and property index (Fig. 5).
3. Via anOperand or CSP_Operand class object which one can associate with an existing variable, or use to define or remove a variable at runtime.

The first way is used in all the high-level interfaces of the SuperGroup, Algorithm and other class objects with which a user will interact. Behind the scenes, the property database issues a CSP_INDEX for each queried variable (CSP_INDEX StorageKeys(const char* s ) const) to rapidly carry out ‘Read’ and ‘Store’ operations (Fig. 5). The third variable access strategy is described in the section on CSP_Operands.
3. Simulation of a Steady-State Fluid Pressure Distribution

How the SuperGroup interacts with other CSP objects can be illustrated using an example main() program which uses the functionality of the CSP library (API = application programmer interface) to calculate a steady-state fluid pressure distribution in an aquifer model with Dirichlet boundary conditions for fluid pressure. The steady-state fluid-pressure distribution in a permeable region subjected to fixed boundary pressures (Dirichlet boundary conditions) is a function of spatial permeability variations and is described by the partial differential equation

\[ Q = \frac{k}{\mu} \nabla^2 p \] ..............................................................(1)

where \( p, k, \mu \), and \( Q \) are the fluid pressure, the permeability, the dynamic viscosity of the fluid, and a fluid source term, respectively. The solution of this equation allows the calculation of the fluid flux in each element, from the spatially varying fluid-pressure gradients and the hydraulic conductivity, \( K (K = k/\mu) \):

\[ f^{(e)} = \frac{k^{(e)}}{\mu} \nabla p^{(e)} \] ..............................................................(2)

Via piece-wise integration over the finite elements, the finite-element form of equation 1, yields the incremental change in fluid pressure over the domain of interest. Resulting volume integral terms for each finite element are assembled into a matrix equation of the form:

\[ \{K\} \{p\} = \{q\} \] ..............................................................(3)

The left-hand side of this equation contains element contributions to the global conductance matrix \( K \); \( p \) is the solution vector which will hold the nodal fluid-pressure values, and the right-hand vector \( q \) stores the fluid source terms \( Q \). This global system of linear equations has a size determined by its degrees of freedom (DOF). The DOF is equivalent to the number of nodes at which fluid pressure must be computed. This number is equal to the total number of nodes in the model, minus the Dirichlet nodes at which the fluid pressure is prescribed. Dirichlet nodes impose fixed coefficients on the other algebraic equations constituting the solution matrix.

The element contributions to the global system of linear algebraic equations consist of element conductance matrices \( K^{(e)} \) which are added to the zeroed global matrix \( K \) (equation 3) on the left-hand side, and of vector contributions \( Q^{(e)} \) which are added to the global right-hand vector \( q \). For (two-dimensional) linear triangular finite-elements, element conductance matrices have the form:

\[ K^{(e)} = \iiint_A \begin{bmatrix} \frac{\partial N_1}{\partial x} & \frac{\partial N_1}{\partial y} \\ \frac{\partial N_2}{\partial x} & \frac{\partial N_2}{\partial y} \\ \frac{\partial N_3}{\partial x} & \frac{\partial N_3}{\partial y} \end{bmatrix} \begin{bmatrix} \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial x} & \frac{\partial N_3}{\partial x} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_2}{\partial y} & \frac{\partial N_3}{\partial y} \end{bmatrix} dx dy \] ..............................................................(4)
where $N_{1,3}$ are three element interpolation functions (one for each node), used to describe the fluid-pressure variation over the element. $A$ is the area of the element, and $K_{x,y}$ is the directional hydraulic conductivity. $K_{x,y}$ is constant within each element but is allowed to vary from element to element. In the example, $K_{x,y}$ is isotropic such that $K_x = K_y$.

The global conductance matrix $K$ is accumulated during the execution of the Velocity algorithm. Equation (4) is represented by the generic differential operator \texttt{OPERAND\_GRADIENT\_N\_GRADIENT\_N}. In this case, \texttt{OPERAND} refers to the hydraulic conductivity and \texttt{N} refers to the fluid pressure modelled by the element interpolation functions. \texttt{N} appears twice, first as a placeholder for the element weighting functions and, second, for the element interpolation functions. In the Galerkin finite-element formulation used in the example, however, weighting and interpolation functions are the same.

Element contributions to the right-hand vector have the form:

\[
Q^{(e)} = q^{(e)} \int_{\text{integration}} N dA \rightarrow \frac{q^{(e)} A^{(e)}}{3} \left\{ \begin{array}{c} N_1 \\ N_2 \\ N_3 \end{array} \right\}
\]

$q^{(e)}$ is the fluid source term per unit area ($\text{m}^3 \text{m}^{-2}$), which could be envisaged as the rainfall on the landsurface represented by the element or as the amount of fluid produced by dehydration reactions from a triangular rock slab of unit thickness. The factor 3 appears since the source term is distributed over the 3 nodes of each triangular element. $Q^{(e)}$ terms are accumulated by the \texttt{OPERAND\_N} differential operator in the Velocity algorithm.

For the CSP simulation example, an adaptively refined finite-element mesh created by the program Triangle is used to represent a rectangular domain (see Chapter 4 to learn about mesh creation). The Velocity subclass of the CSP\_Algorithm is applied to the SuperGroup which also calculates the Darcy flow velocity as a vector property from the computed fluid pressure gradients and the hydraulic conductivities of the elements (see equation 2). In the example, the results of the computation are output as JPEG images (Fig. 6), text files, VTK files, and GoCad formatted Tsurface files.

```csharp
#include "CSP_definitions.h"

#include "SkmErrorHandler.h"
#include "SuperGroup.h"
#include "LinearTriangle.h"
#include "FileHandler.h"
#include "CSP_VTK_Interface.h"
#include "MeshInterface.h"
#include "CSP_Operand.h"

#include "Velocity.h"
#include "ConductivityFromPerm.h"
#include "ScalarVolumeFlux.h"
```
// global objects / variables
SkmErrorHandler skm_err;
DATA_TYPE global_time;
bool ADAPTIVELY_REFINED;
bool Y_POINTS_UPWARD_IN_2D;
FiniteElementPtr global_FEtypes[1];
CSP_String global_physvar_textfile;

// main program starts here

int main() // main returns 0 if everything went fine
{
    // declaration of global variables inside the main() function
    extern SkmErrorHandler skm_err;
    extern DATA_TYPE global_time;
    extern bool ADAPTIVELY_REFINED;
    extern bool Y_POINTS_UPWARD_IN_2D;
    extern FiniteElementPtr global_FEtypes[1];
    extern CSP_String global_physvar_textfile;

    // -------------------------------------------------------------
    // 0. read mesh from file and build model (SuperGroup object)
    // -------------------------------------------------------------
    // create a container for the mesh
    MeshInterface mesh_interface;
    CSP_VSet mesh_container;
    // choose a finite-element type compatible with the input mesh
    LinearTriangle default_element;
    global_FEtypes[0] = &default_element;
    char file_name[200];
    cout << "main: Enter name of 'Triangle' input file set: ";
    cin >> file_name;
    // read mesh and stor it in a mesh_container object
    mesh_interface.ReadTriangle2DMesh( file_name, mesh_container );
    // construct model from the data in the mesh_container
    SuperGroup example_model( mesh_container );
    const PropertyDatabase& p_ref = example_model.ReferencePropertyDatabase();
    // ---------------------------------------------------------------
    // 1. assign initial values & Dirichlet boundary conditions
    // ---------------------------------------------------------------
    example_model.InputUniformValueWhere( PLAIN, "fluid density", 1000.0 );
    example_model.InputUniformValueWhere( PLAIN, "fluid volume source", 0.0 );
    example_model.InputUniformValueWhere( PLAIN, "fluid pressure", 0.0 );
    example_model.AssignBoundaryValues( RIGHT, "fluid pressure", DIRICH, 1.0,
     1.0 );
    example_model.AssignBoundaryValues( LEFT, "fluid pressure", DIRICH, 100.0,
    100.0 );
    // ---------------------------------------------------------------
    // 2. use interrelation to calculate hydraulic conductivity from
    // permeability
    // ---------------------------------------------------------------
    double fluid_viscosity = 1.6e-3; // Pa s-1
    ConductivityFromPerm interrelation1(p_ref, fluid_viscosity );
    example_model.Pass( interrelation1 );
    // ---------------------------------------------------------------
    // 3. use CSP_Algorithm to compute fluid-pressure distribution and
    // Darcy-flow-velocity in a post-processing step
    // ---------------------------------------------------------------
    Velocity steady_state_pressure( p_ref );
    // lefthand side of equation
    steady_state_pressure.AddOperation( OPERAND__GRADIENT_N__GRADIENT_N,
      "conductivity",}
"fluid pressure","fluid pressure");

    steady_state_pressure.AddOperation( OPERAND__N,
              "fluid volume source", "fluid pressure");

    steady_state_pressure.Reset();

    ScalarVolumeFlux  flux_interrelation( p_ref );
    example_model.Pass( flux_interrelation );

    example_model.OutputDataToJPG( "fluid-pressure", "fluid pressure", 1 );
    example_model.OutputDataAsTextColumns("velocity", "velocity", 1 );
    example_model.OutputVariablesToTSurface( "model-results", 1 );
    vtk_output.OutputDataToVTK( example_model,"nvelo","nodal velocity", 1 );
    vtk_output.OutputDataToVTK( example_model,"pfluid ","fluid pressure", 1 );

    Vec  xy;
    example_model.Dimensions( xyz );
    double dx = 0.2 * (xy(1) - xy(0)),
             dy = 0.2 * (xy(3) - xy(2));

    example_model.FormAndAddRectangularGroup("granite",
               xy(0)+dx, xy(1)-dx,
               xy(2)+dy, xy(3)-dy );

    example_model.ChangePropertyInGroupToWhere("granite",
              "fluid volume source", 1.0e-10, PLAIN );
    example_model.ChangePropertyInGroupToWhere("granite",
              "fluid volume source", 1.0e-10, BOUNDARY );

    steady_state_pressure.RestrictApplicationTo("granite");
    steady_state_pressure.AddOperation( OPERAND__GRADIENT_N__GRADIENT_N,
              "conductivity", "fluid pressure", "fluid pressure");
    steady_state_pressure.AddOperation( OPERAND__N,
              "fluid volume source", "fluid pressure");
// assign Dirichlet boundary conditions to group boundary
steady_state_pressure.SetOperandEssentialStatus(
    "fluid pressure", BOUNDARY );

// compute \([K]\{p\} = \{q\} \& v = K \text{ grad } P\) in model subregion “granite”
example_model.Pass( steady_state_pressure );
example_model.Pass( flux_interrelation );
example_model.OutputDataToJPG( "fluid-pressure", "fluid pressure", 2 );
example_model.OutputDataToJPG( "volume-flux", "scalar volume flux", 2 );
return 0;
}

The example source code displays several key features which are germane to any
C++ object-oriented program:

- Comment lines start with a double slash (//).
- Functions typically have a return value which is output when the body of the
  function enclosed in “{}” has been executed.
- For all objects which are used in the main program, corresponding header files
  must be included into the source code file, before the function which uses
  these object begins. These header files contain necessary object declarations.
  The actual functionality of the objects is defined (implemented) inside the
  binary library files.
- Each object must be constructed by a constructor call before it can be used
  (for example the MeshInterface object in program segment (0)).
- Constructor (functions) which take no arguments are called default constructors,
  others use data to initialize objects.
- Object construction implies that an instance of a C++ class is created. This
  instance has a unique name independent of the class name. This process is
  also called instantiation.
- Before global objects or variables can be used in a function, C++ requires
  these to be declared using an extern statement. This statement just tells the
  compiler that there is global instance of a class x with the instance name y.
  Again, the class must be declared in a corresponding header file.
- Once an object has been instantiated, it can be used by calling its interfaces.
  Interfaces are functions which only work in conjunction with that object. In
  C++, this is signified by the scope operator “.”. Hence, the statement
  dog.bark(loud), tells the instance dog of a class representing dogs to invoke
  its member function bark(). The function argument loud could be double
  giving a value in decibels.
- There are no destructor calls in main(). Unless an object has been created
dynamically (using the new operator) its destructor is called automatically by
C++ at the end of the main program. Dynamic objects are destructed by
calling the delete() function.
In the example program,

1. a (SuperGroup) model is constructed from a series of input files created before by the Triangle mesher \textit{\textbf{use d.1 model as example}}. The user just enters a common file name “file.1” (one stands for the first refinement level). The model name suffices as argument for the member function \texttt{ReadTriangle2DMesh()} which reads the input files “.node”, “.ele”, “.neigh”, and “.edge” as is described in more detail in Chapter 4.

2. The input data describe the geometry of the model and the material property “permeability”. Thus, initial values for the fluid source term etc. must still be assigned to the mesh. By default, all variables specified in the “...variables.txt” file are initialized to a status of \texttt{PLAIN} and a value of \texttt{NAN} (not a number). One can immediately tell if one forgot to initialize a variable, because the result of calculations involving such a variable will also be \texttt{NAN}.

3. Fixed fluid-pressure values are assigned as Dirichlet boundary conditions to the right and left model boundaries.

4. An interrelation subclass named \texttt{ConductivityFromPerm} is instantiated and applied to the model (calling the \texttt{Pass()} member function of the SuperGroup instance representing the model), in order to calculate the hydraulic conductivity from the permeability and the fluid viscosity of water at room temperature.

5. The PDE (equation 1) is solved by the Velocity CSP\_Algorithm subclass which also post-processes flow velocities.

6. The length of the computed flow velocity vectors is computed by another Interrelation subclass and stored as the scalar variable “volume flux” for later display.

7. The computed “fluid pressure”, “velocity”, and “volume flux” are output to shaded JPG images which associate colors with variable values and to other file formats which store the actual values.

* NAN can be tested for using the standard C++ function \texttt{isnan()}.
After the pressure distribution and the scalar volume flux in the overall model have been computed and output in program step 5, a rectangular region is defined as a group object called “granite”. Now a fluid source term is applied to this region, boundary conditions are set, and the Velocity algorithm is applied. The resulting balloon shaped fluid-pressure distribution and the corresponding variation in the scalar volume flux in “granite” is again output to JPEG images (Fig. 7a).
Fig. 8: Example 2 input and output for a finite-element model of steady-state fluid flow in a two-dimensional domain with Dirichlet boundary conditions. a) Input permeability distribution of two highly permeable fractures in a homogeneous matrix; b) computed fluid pressure distribution; c) calculated volume flux. The lighter color always represents the higher property value. In a grayscale image which may apply to this copy of the User’s guide, red and blue are indistinguishable but the example is readily reproduced with CSP as an exercise.

In the following sections the CSP object classes which were used in this example are introduced. For documentation of individual member functions of these classes, the CSP HTML Reference should be consulted.

**SUPERGROUP**

The SuperGroup and sub-regions thereof available through Group class objects represent the geometry and material properties of a CSP model.

The class SuperGroup (Fig. 2) has the largest number of interfaces of all CSP classes and it is the class with which every CSP user will interact. As discussed above, the SuperGroup essentially consists of the finite-element mesh (CSP_MeshManager). a
property database (PropertyDatabase) for the physical variables, and a storage scheme for the distributed physical variable values (CSP_MemoryManager) assigned to the nodes, constraint points, and the elements. Groups of elements that make up geological entities in the model are contained as Group objects.

Typically, public interfaces of the SuperGroup invoke a data-access-and-modify or I/O process as is shown in Figure 6. The key steps in this process are that the database is queried for the specifications of a variable identified by a string (for instance "permeability"). The database returns a CSP_INDEX that uniquely identifies the variable for efficient access in the computations.

Once a target object in the mesh hierarchy has been found, element, constraint point or node ID numbers are used to retrieve values or status information for the variable from the CSP_MemoryManager via methods like Read() or Store(). These methods are overloaded to retrieve Scalar-, Vector-, and Tensor Variables.

Within the CSP_MemoryManager, the variables live in STL-vector templates and are instantiated at runtime, dependent on the specific variable type. The length of such vectors varies with the placement of the variables. Originally, the CSP_MemoryManager object allocates storage for the variables that were specified to the variable database in the variables text input file. If one creates a new variable at runtime via a CSP_Operand, a new vector is inserted into the CSP_MemoryManager for this variable. Thus, one can efficiently create new variables at runtime. However, only variables that one wants to use again outside the context of algorithms should be created at runtime. Any variable in the database requires extra storage even if it is not used.

One can build a SuperGroup representation for any model that is based on a finite-element discretization of a real-world system. Computations are applied to the SuperGroup through its interface Pass(). One can carry out finite-element computations specified as Algorithms or derived types of the base classes Algorithm and CSP_Algorithm. Pass() will also execute Interrelations among variables (including the dependent variables that are computed at each timestep), and “Accept()” will give access to visitor objects as is described later. An example of an Interrelation-based calculation would be to calculate a permeability that is dependent on fluid pressure.

Any derived algorithm or interrelation can also be restricted to Group objects. This enables a geological domain-decomposition. In this design, Observer objects tell adjacent groups to update each other, facilitating parallel computations with CSP.

**GROUP**

The aforementioned Group class (Figs. 1, 2) implements selective access to user-named subregions of a model represented by a SuperGroup instance. A Group is essentially a list of finite elements that belong to it (they do not have to represent contiguous space, see Fig. 1). Elements become group members on the basis of selection criteria. The smallest contained unit in a group is the finite element. A group cannot contain single nodes or constraint points. Group operations affect SuperGroup
property values within the groups. The Group class associates additional DATA_STYLE flags with its nodes, constraint points, or elements for the purpose of group-restricted computations.

One can define groups in different ways: By property values, as rectangular regions identified by a bounding box, from lists of finite elements, and on the basis of existing groups. For the latter, boolean algebraic operations can be applied to existing groups in order to identify new groups. This capability includes group intersections, differences, symmetric differences and unions. Using this capability, one could, for instance, define a Group as a subregion of a shaly layer in which the temperature is above 170°C and the confining pressure is between 60 and 80 MPa.

Group-restricted operations include computations of interrelations and algorithms, visitations (via Visitor objects), and the modification of distributed variables and their SuperGroup flags.

**SKMErrorHandler & Global Variables**

Yes, there are some global objects and variables inside of CSP! — And worse yet, for convenience, they are declared in the file “CSP_globals.cpp”, but must be defined before any main() program begins. Their values can be changed anywhere in the code. If one does not want to include the CSP_globals.cpp file one can alternatively declare the variables as extern in each of the code segments which one writes. The global objects and variables are:

**SkmErrorHandler.** The object instance “skm_err” handles how errors are reported and processed by CSP in a platform specific way. It must therefore be known to all objects which want to use its error handling capabilities.

The enum SKM_ERR is an “ontogenetic” rudiment of CSP from the time before C++ compilers properly implemented standard exceptions (some still do not!). CSP now contains some exception handling which will gradually complement the SkmErrorHandler scheme.

```cpp
enum SKM_ERR { INFO, EXCEPTION, WARNING, ERROR, FATAL_ERROR };
```

The SKM_ERR enumeration is used internally by the error handler to record if something of interest to the user happened or went wrong and to define how this is reported. Important here is that CSP issues INFO and WARNING messages which do not influence program execution. A fixed maximum number of ERRORS may occur before CSP terminates a simulation and a FATAL_ERROR immediately terminates a CSP run.

The SkmErrorHandler logs information strings to ‘cerr’ and to the log file 'ErrorHandler.log'. To be able to trace a message to its source, a string that consists of the class and method name from which the message originated is supplied as first message argument. In case of an overloaded method additional information in braces is included into the output to identify the called method uniquely, for example:

```cpp
SkmErrorHandler::Notice(string):
```
Messages which are logged to file, are automatically preceded by a time string which tells the user when the incident prompting the message occurred. Since messages may also contain numerical data in the description of the error message the SkmErrorHandler interface is overloaded to accept CSP_String arguments (CSP_String has overloaded operators that allow the method writer to convert numbers to strings). To handle top level (C++) exceptions the SkmErrorHandler employs the standard exception base classes exception, logic_error, and runtime_error. For CSP-specific error types instances of the class CSP_Exception can be thrown. A goal achieved with this exception handling was to store the transient state of a run before abortion due to a fatal error. This now permits that a simulation can be continued at a later time, following the diagnosis and remedy of error conditions. To save a computation to a binary file, the model state is first exported into a CSP_VSet and then made permanent as a file.

The SkmErrorHandler is only used inside high-level CSP objects like the SuperGroup. These objects carry enough information such that it makes sense to collect it into user-readable diagnostics on a CSP run. Small single-purpose classes report errors directly to the 'cerr' stream to prevent a loss in their generality by making them depend on the error handler. All example programs in this User’s Guide instantiate a global instance of the SkmErrorHandler in the file which contains the main() program via the default constructor:

```cpp
SkmErrorHandler   skm_err;
```

This instance is re-declared inside the file 'CSP_globals.cpp' or one can declare it in any other source code file using the error handler as:

```cpp
extern SkmErrorHandler   skm_err;
```

To call the error handler inside an original method one writes something like:

```cpp
skm_err.notice( WARNING, "Supergroup::AssignBoundaryValues3D",
                "With this method only scalars can be handled" );
```

In this example, the message type is a warning issued by the method AssignBoundaryValues3D() of the SuperGroup, followed by the message string(s).

**Global runtime.** There is only one global time simulated by a model and this is kept track of by the floating-point variable 'global_time'.

```cpp
DATA_TYPE   global_time = 0.0;
```

The global_FEtypes is a fixed size array of pointers to finite-element subclass objects representing single or multiple element types constituting the mesh:

```cpp
FiniteElementPtr   global_FEtypes[CSP_N_FINITE_ELEMENTS];
```

This array of memory locations is initialized as the very first thing in a simulation main program by instantiating one or several finite-element objects and by assigning the global_FEtypes array pointers to their memory addresses, e.g.
LinearTetrahedron tetrahedron;
global_FEtypes[0] = &tetrahedron;

The last line of the example code above makes the first array pointer point at the heap address of the tetrahedron object.

Finally, the previously discussed text file with the physical variable specifications is named by the global variable global_physvar_textfile. This is a CSP_String.

**CSP_OPERANDS**

CSP_Operands associate themselves with existing physical variables defined in the PropertyDatabase or prompt the latter to create new physical variables at runtime. If a new variable is created at runtime, it will be destroyed again when the destructor of the CSP_Operand is called. CSP_Operands also give interfaces to the standard mathematical functions in the C++ library file “math.h”. In the following example, the natural logarithm of the variable “porosity” times a factor is output to a text file. The CSP_Operand is created dynamically such that it can be removed again at runtime:

```cpp
// existing variable
CSP_Operand phi( current_supergroup, "porosity", SCALAR, ELEMENT );

// new variable only needed for logarithmic output
CSP_Operand* log_phi = new CSP_Operand( current_supergroup,
    "log porosity", SCALAR, ELEMENT );

// assign the porosity to the new variable without direct assignment
double factor(23.4);
*log_phi = 0.0;
*log_phi += phi * factor;

// logarithmitize the values
log_phi->Log();
log_phi->Out( logarithmitized_porosity );

// delete the temporary variable
delete log_phi;
```

In summary, CSP_Operands provide direct access to and mathematical manipulation options for distributed physical variables.

**OPERANDS**

It is a design objective of CSP, that expressions involving physical variables whose values are stored in the basic variables ScalarVariable, VectorVariable, and TensorVariable should read like mathematical equations in which variables are represented by a notation for local or distributed data. At the same time range violations should be reported if a calculation result is not physically meaningful. Normally, these capabilities are made available through Operand objects which work behind the scenes, but CSP users can also profit from them directly, if they are
prepared to write their own Interrelation, Algorithm, PDE Operator or Visitor subclasses. Apart from facilitating to write mathematical expressions which are applied to the whole SuperGroup as can be done using CSP_Operand objects, Operand objects meet the need to use distributed instances of properties in calculations inside objects, partially ignoring variable IDs and the variable placement in the mesh (Node, ConstraintPoint or Element). This requires (1) the specification of a mathematical notation to represent variables in mathematical expressions, and (2) the association of the CSP_INDEX of the physical variable with this notation. Now the PropertyDatabase no longer needs to be queried every time the expression is evaluated. Finally (3), temporary variable storage is needed to hold a value for range-checking before it is written back into the SuperGroup storage. Point 3 is essential when it comes to tracking down where in a chain of expressions encapsulated into interrelations, a physically meaningless value was produced.

These capabilities are implemented in the Operand class which is used in Interrelations, CSP_Algorithms, Algorithms, PDE Operator and Visitor objects. The Operand is a simple class holding a CSP_String with its name, a CSP_INDEX describing the represented variable, any one of the basic CSP variables, a physically meaningful range for them, and an output condition flag which specifies the DATA_STYLE of the distributed property which the variable is permitted to overwrite if the calculation yields a result which is within the permitted range.

The Operand operators +, -, *, /, +=, -=, *=, /= are overloaded such that Operands can interact with each other, double-type data, and CSP basic variables. One may, however, not write expressions like A = C. A = B means following the logic of an assignment that the Operand A now refers to exactly the same variable as B and it will also contain the same values. If Operand A represents a property stored in the SuperGroup, the variable A will no longer exist following such an assignment.

Importantly, scalar-, vector-, and tensor variables are more than their mathematical counterparts, because they also contain STATUS flags which specify how they are treated in the assembly of the finite-element equations. This requires some rules about the flag assignment using Operands. The scalar variable expression

\[ A = C + B; \]

must assign a flag to the temporary variable that is created. The rule is that the temporary Operand which is created by the addition will be flagged like the Operand immediately to the right of the assignment. Thus, if C=PLAIN and B=DIRICH, A will be flagged as PLAIN. If combined C++ operators like += is used, A will retain its flag.

If one compares an Operand with a DATA_STYLE enumeration, the comparison will be carried out using the flag(s) of the underlying variable. Thus, if a vector variable in a 2D calculation has two different flags (for instance, a DIRICH flag for the \( x \) displacement and a PLAIN flag for the \( y \) displacement), any comparison will evaluate as false. In this case, one should use the method \texttt{VectorVariable::Flag( int i )} to carry out a more specific comparison. Any comparisons with not initialized Operands or Operands that store NAN (not a number) values in the variables will evaluate to false.
By default, Operands provide storage for a single variable value of type ScalarVariable, VectorVariable, or TensorVariable. However, Operands can only be created if this variable exists in the PropertyDatabase by contrast to CSP_Operands which can spawn the generation of a new variable in the PropertyDatabase.

The construction of temporary 'Operands' in expressions like:

\[ C = (A + C) \times (B - D) \times 34.2; \]

should be circumvented, because the construction of temporary objects costs runtime and can only be avoided by the compiler in some cases through construction of the Operands in the return statements of the overloaded operators (return-value optimization). To avoid the creation of temporaries in extensive calculations one should rather write statements like:

\[ C = 34.2; \]
\[ C += A; \]

Operands also store an integer variable called 'calculation offset'. This is used by Algorithms to modify the placement of dependent variables in global solution matrices, when coupled systems of equations are solved.
INTERRELATIONS

As seen in the steady-state fluid-pressure example, CSP calculations which can be
carried out on an element by element basis, because they do not involve the inversion
of a global solution matrix, are done by subclasses inherited from the Interrelation
base class. When writing the source code for such subclasses, one associates required
global physical variable(s) with Operand class objects and writes intended
mathematical expressions inside the virtual Calculate() method of the new
subclass. A decisive rule for interrelation objects is that a single interrelation can
only alter the value of a single physical variable. This restriction allows to execute
interrelations in a sequence that takes mutual interdependencies into account.

Fig. 9: Sequence diagram illustrating the use of an interrelation module (e.g., Conductivity) in a CSP
simulation.

As an example of a calculation which could be carried out using an Interrelation
object, the calculation of the element property “hydraulic conductivity” by dividing
permeability by fluid viscosity is presented in Figure 9. The code for this interrelation
is made public in the header file of the class 'ConductivityFromPerm' and is also
listed below. To apply this interrelation to a model, one simple creates an instance of
ConductivityFromPerm and supplies it as argument to the Pass() interface of the
SuperGroup.

```c
#ifndef __ConductivityFromPerm_h__
#define __ConductivityFromPerm_h__

#include "CSP_definitions.h"
#include "Interrelation.h"
```

```c
#include "CSP_definitions.h"
#include "Interrelation.h"
```
The source code listing shows that a new Interrelation subclass is created in a few simple steps:

1. Test the formula that describes the interdependencies of the physical variables of interest. This may involve graphing and designing this Interrelation as a formula in Matlab or Mathematica or doing a curve fit through some experimental data. One must also define a value range for each of the involved input variables as the range in which the mathematical formulation produces meaningful results. Ideally, this range should be at least as wide as the value range specified in the property-database text file from which the CSP program initializes the property storage. Each input range must be tested before variable values are used in the equation: Using the global instance skm_err of the SkmErrorHandler, range violations can be reported as was described earlier. This approach is the only way of ascertaining that a program produces correct results once a model becomes complex and contains several interrelations.

2. Define a notation and make each symbol a reference to an Operand. These references will be private data members of the subclass declared by public derivation from the Interrelation class. Importantly, if one does not use references to Operands but Operands instead, the calculation will either not read any of the variable values stored in the SuperGroup object, or the result will not be transmitted back into the SuperGroup.
3. Register the variables in the constructor of the derived Interrelation subclass. In C++ all references must be initialized in the constructor. First, one constructs the base class, then the base class method GlobalProperty() is used to register the involved variables and finally one defines which of these variables will store the result of the calculation. This is done with the base class method DefineResultProperty(). One must also define the output condition flag (OutputCondition()) for the result property. The use of this flag is to prevent Interrelations from overwriting variable values that have been fixed (e.g., DIRICH conditions).

4. Override the pure virtual member function Calculate() of the Interrelation base class. The overriding new method Calculate() will contain the mathematical formulation of the interrelation written as equation, using the Operand reference notation and involving * / + - < > <= >= and other mathematical symbols which are interpreted as overloaded operators of the Operand class. If many temporary variables are needed they should be defined as private data of the Interrelation subclass to prevent their destruction when the Calculate() method goes out of scope. Also, if one writes something like

   K += b + c - d * 24.5;

one will automatically create four temporary Operands (why? - This is a little C++ exercise).

5. Decide where to apply the new Interrelation by passing it to the SuperGroup. This will invoke the Interrelation method Apply(), which computes the formula and performs a range check on the result values.

6. One must also define the application domain, if the new interrelation shall apply only to a group of elements rather than to the whole SuperGroup. Use the method RestrictApplicationTo() to achieve this.

Calculations one can perform with Interrelation subclasses are limited to result variables that are on the same or a higher level in the SuperGroup mesh hierarchy than the independent variables. For instance, one can calculate an element property from node- or constraint point properties, but not the other way around. If one calculates an element property from node property values, the interrelation algorithm will use averages over the nodes rather than individual values in the calculation. If this would be too limiting one can redefine the virtual Interrelation base class method Apply() as an original method. Before doing this, however, one may consider to do the calculation as a postprocessing operation of an Algorithm or via a Visitor object.
VISITORS

The visitor scheme, as described in Gamma et al. (Reusable object-oriented software patterns), implements access to the public interfaces of classes inside of class hierarchies (Fig. 10). It thus permits the addition of new functionality without having to modify the existing classes. In the visitor design pattern, each target class has an interface `Accept(visitor&)`. This interface calls the visitor’s member function `Visit(target_class*)` such that the visitor gains access to the public interface of the target class object. With this simple technique a wide variety of operations are instrumentalized in CSP. Visitors derived from the `CSP_FEM_Visitor` base class can differentiate the treatment of sub-targets in a hierarchy (here the SuperGroup object hierarchy) by overloading the member function `Visit()` for groups, elements, constraint points and nodes, i.e. `Visit(Node*)`, `Visit(ConstraintPoint*)` etc. Any kind of distributed data can be retrieved from the finite-element mesh via visitor objects. Corresponding `Accept(CSP_FEM_Visitor&)` methods have been added to all of the elements in the SuperGroup hierarchy. As an open-source example of an inherited visitor, the `ID_Visitor` collects ID numbers of target objects. The declaration for this visitor is in “CSP_ID_Visitor.h” (including inline functions that show how the visitor accumulates desired data).

![Diagram](CSP_visitor.CV5)

Fig. 10: Application of visitor object to SuperGroup or sub-regions thereof (Figure CSP_visitor.CV5).

The application sequence for a CSP Visitor is as follows (see Fig. 10): Firstly, the visitor is submitted to the SuperGroup `(super_group.Accept(vtor))`. The SuperGroup determines whether the visitor wants to visit the whole model or a named sub-region thereof which must be represented by a Group. Subsequently, either the SuperGroup or the Group, allows the visitor element by element access to its
members. The visitor may also choose to travel directly from an element to its neighbor, tracking a fluid flow path or conducting a mesh sweep such as a "flood fill".

(2D AND 3D) ALGORITHMS

Algorithms, in CSP terminology, were designed for calculations that require the inversion of a global solution matrix $G$. One or several second-order partial differential equations are transformed in a set of linear algebraic equations which are solved in a problem of the form $Gx = rh$, where $x$ is the solution vector and $rh$ is the right-hand vector. $G$, $x$, and $rh$ are implemented as Meschach++ sparse matrix (SpMat) and double precision floating point vector (Vec) objects.

CSP3D3.0 supports two design patterns for computing global and spatially restricted variations of dependent physical variables. The more flexible way is implemented through an Algorithm base class which collaborates with MathOperatorLHS and MathOperatorRHS base classes, facilitating the composition of differential equations without yet specifying the type of finite element which shall be used to compute corresponding contributions to the global solution matrices.

Fig. 11: Sequence diagram for the application of an Algorithm to the SuperGroup. Key steps are (1) construction of MathOperator objects added to the Algorithm to compose the PDE which shall be solved (2) application of Algorithm to SuperGroup, and (3) inversion of the global solution matrix (Meschach++ SpMat class type) by Solver object.
The simpler way, albeit restricted to 2D linear triangular finite elements, is based on a design pattern where a CSP\_Algorithm base class collaborates with a Triangle class which provides the finite-element related interpolation functions and exact analytic integration methods.

Both, the Algorithm and the CSP\_Algorithm class serve the purpose of the assembly of a global square sparse matrix (Meschach++ SpMat class) for inversion, given a right-hand side and a solution vector into which the result is returned. The matrix inversion is carried out by the Solver class object which offers the user different solution methods. The enum COMPUTATION\_METHOD is used by the Solver class to specify the solution method. One can choose between algebraic multigrid (AMG), conjugate gradient (CG) and Gaussian elimination (GE) schemes to invert sparse solution matrices.

```cpp
enum COMPUTATION\_METHOD { AMG, CG, GE };
```

Dependent on the type of computation, solver parameters have to be set manually through member functions of the Algorithm or CSP\_Algorithm classes such as:

```cpp
algo.IncreaseMultiGridVectorStorage( 5 );
```

This method increases storage limits for the algebraic multigrid solver (AMG) by Ruge and Stueben (1987) and Stueben (2000). Details on solver parameters are given in the HTML Reference manual in the documentation of the Solver class. The following source code computes the evolution of fluid pressure with time using a lumped capacitance matrix and a fully implicit finite-difference time-stepping scheme:

```cpp
// Algorithm computes: ([C] + dt[K])\{p\}t+dt = [C]\{p\}t + dt \{Q\}t+dt
// ---------------------------------------------------------------
Algorithm transient_pressure;
transient_pressure.IncreaseMultiGridVectorStorage( 5 );

// [K] conductance matrix
Integral\_dNT\_op\_dN\_dV conductance1( p\_ref, "permeability", "fluid pressure", "fluid pressure" );
conductance1.MultiplyWithTimeIncrement();

// [C] capacitance matrix lefthandside
Integral\_NT\_lhsop\_N\_dV capacitance\_lhs( p\_ref, "storativity", "fluid pressure", "fluid pressure" );
capacitance\_lhs.LumpedFormulation(true);

// [C] capacitance vector righthandside
Integral\_NT\_op\_N\_dV capacitance\_rhs( p\_ref, "storativity", "fluid pressure" );
capacitance\_rhs.LumpedFormulation(true);

// [Q] fluid volume source
Integral\_NT\_op\_N\_dV source1( p\_ref, "fluid volume source", "fluid pressure" );
source1.MultiplyWithTimeIncrement(true);
source1.AddAccumulateLater();
source1.lumpedFormulation(true);

transient\_pressure.Add( &conductance1 );
transient\_pressure.Add( &capacitance\_lhs );
transient\_pressure.Add( &capacitance\_rhs );
transient\_pressure.Add( &source1 );
```
transient_pressure.TimeIncrement( time_increment );  

...  

while ( time < max_time && timestep <=100 )  
{
    time += time_increment;
    example_model.Pass( transient_pressure );
    example_model.MinMaxOf("fluid pressure", pmin, pmax );
    cout <<\nmain: Time: << time <<" Fluid pressure range (Pa): ";
    cout << pmin <<" to " << pmax << end;
    example_model.OutputDataToJPG( "fluid-pressure", "fluid pressure",
         timestep, 10.0, 1000.0 );
    example_model.OutputDataAsTextColumns( "fluid-pressure", "fluid
         pressure", timestep );
    timestep++;
}  

CSP_ALGORITHMS FOR LINEAR TRIANGULAR ELEMENTS  

The old-style CSP Algorithm class uses PDE operators defined as enums in the file “finite_element.h”. Enumerations such as OPERAND__GRADIENT_N__GRADIENT_N define the general form of the differential equation components. They are replaced by MathOperator subclasses for each basic type of PDE in the new Algorithm design pattern. In the enumerations OPERAND and N are placeholders for variables defined using the method AddOperation() when building an instance of a CSP_Algorithm. Thus, the enumeration syntax defines the position of physical variables in PDEs: The 'OPERAND' typically is a variable needed to setup a material property matrix and N stands for test functions (=element interpolation functions) and weighting functions. An Operand will never be an output variable, but since element-interpolation functions are used to approximate the value of the dependent variable, the physical variable that one puts as a placeholder of N will be the result and output property of the finite-element computation. In the code library the following differential operators are available for triangular finite elements:

<table>
<thead>
<tr>
<th>Enum</th>
<th>Placement in Gx=rh</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPERAND__N</td>
<td>right-hand vector operator</td>
<td>volume integrals</td>
</tr>
<tr>
<td>OPERAND__N2</td>
<td>right-hand vector operator</td>
<td>volume integrals</td>
</tr>
<tr>
<td>LUMPED__OPERAND__N</td>
<td>right-hand vector operator</td>
<td>volume integrals in lumped elements</td>
</tr>
<tr>
<td>LUMPED__OPERAND__N2</td>
<td>right-hand vector operator</td>
<td>volume integrals</td>
</tr>
<tr>
<td>NEUMANN_X__OPERAND__N</td>
<td>right-hand vector operator</td>
<td>surface integrals, horizontal component</td>
</tr>
<tr>
<td>NEUMANN_Y__OPERAND__N</td>
<td>right-hand vector operator</td>
<td>surface integrals, vertical component</td>
</tr>
<tr>
<td>NEUMANN__OPERAND__N__N</td>
<td>right-hand vector operator</td>
<td>surface integrals</td>
</tr>
<tr>
<td>OPERAND__N__N</td>
<td>right-hand vector operator</td>
<td>volume integrals</td>
</tr>
<tr>
<td>LUMPED__OPERAND__N__N</td>
<td>left-hand vector operator</td>
<td>volume integrals, lumped element formulation</td>
</tr>
<tr>
<td>OPERAND__GRADIENT_N__GRADIENT_N</td>
<td>left-hand grad2 (Laplacian), volume integrals</td>
<td></td>
</tr>
</tbody>
</table>

LUMPED refers to operators which diagonalize element matrices. These operators are designed for finite-difference timestep calculations where a time-derivative is calculated. The lumping procedure is explained in the documentation of the finite-element library (Chapter 7). The N2 operators are intended for transient calculations, prompting the addition of the matrices or vectors after all other accumulations or multiplications into the right-hand side have occurred.
All CSP\_Algorithm differential operators are implemented as methods of the Triangle class with a corresponding name, e.g., Operand\_N() etc. Basic equations are defined and stored in an STL map of Operation objects. Each operation represents an equation to be solved and contains information on the place that the Operands have in the equation. Operands are shared among equations. The CSP\_Algorithm object encapsulates the whole process of (cf., Fig. 12):

1. Definition of a finite-element equation
2. Collection of distributed variables values from the SuperGroup object
3. Calculation of the finite-element matrices (contributions)
4. Assembly of the solution matrix
5. Inversion of the solution matrix
6. Transfer of the results back to the SuperGroup after range checking them

Fig. 12: Steps involved in initializing an “old-style’’ CSP\_Algorithm object with a lumped left-hand side operator in which “storativity” is the Operand and “fluid pressure” is both the test-function and dependent variable (Galerkin finite-element method).

In a main program, finite-element equations (Algorithms) are defined as was illustrated by the example of the pressure diffusion equation (Chapter 2).

Algorithm’s are executed by passing them to the SuperGroup:

```cpp
my_supergroup.Pass( evolve_pressure );
evolve_pressure.Reset();
```
Inside the SuperGroup, the public methods of the CSP_Algorithm class are called in an execution sequence similar to that shown in Figure 10. By default, CSP_Algorithms are steady state. For large problems or for transient calculations it is a good practice to re-use CSP_Algorithm objects to minimize memory allocation for global matrices. If one uses a steady-state CSP_Algorithm again, for instance in a Newton-Raphson iteration, global matrices will be reset automatically.

**Time Stepping.** To use a CSP_Algorithm in a timestepping calculation one has to configure it for a transient calculation with SetTransient() or by directly by setting the time increment as shown in this code example (example3.cpp):

```cpp
extern double    global_time = 0.0,
    max_time  = 315360000.0,  // one year
    time_increment  = 5.0;   // 5 seconds
long   timestep;
CSP_Algorithm   evolve_pressure( variable_database_reference );

while ( global_time < max_time )
{
    // ------------ specify production rate from a well during timestep --------
    supergroup.ChangePropertyInGroupToWhere( "well1", 
        "fluid volume source", pumping_sink, BOUNDARY );
    supergroup.ChangePropertyInGroupToWhere( "well1", 
        "fluid volume source", pumping_sink, PLAIN );

    // ------------ CSP_Algorithm for transient fluid flow to the well ---------
    evolve_pressure.AddOperation( OPERAND__GRADIENT_N__GRADIENT_N, 
        "conductivity", "fluid pressure", "fluid pressure");
    evolve_pressure.AddOperation( LUMPED__OPERAND__N__N, 
        "storativity", "fluid pressure", "fluid pressure");
    evolve_pressure.AddOperation( LUMPED__OPERAND__N, 
        "storativity", "fluid pressure");
    evolve_pressure.AddOperation( LUMPED__OPERAND__N2, 
        "fluid volume source", "fluid pressure");
    evolve_pressure.SetTimeIncrement( time_increment );

    // ------------ compute solution for each timestep -------------------------
    supergroup.Pass( evolve_pressure );
    evolve_pressure.Reset();
    global_time += time_increment;
    timestep++;
}
```

Only in time-stepping calculations, initial conditions are assembled into the right-hand vector when the Algorithm is passed to the SuperGroup. A call to Reset() empties the global solution matrix. *Calling Reset() is necessary* since the global matrix is a sparse matrix and the number of entries for it can be different in the next computation. If one forgets to call Reset() the matrix cannot re-allocate memory and one will get a corresponding error message from Meschach++.
4. CSP Input & Output

This chapter describes three different ways of creating geometry input files for CSP. (1) Directly from color-coded images created with a drawing program and turned into meshes by CSP itself, (2) from line drawings which were output as DXF files from the drawing program of choice and are converted into input files for the 2D meshing tool Triangle with the aid of the program SKM_GeometryPreprocessor.exe, and (3) using the commercial program GoCad. A fourth way using the commercial program ICEM_TETRA will be made available in the first half of 2001.

I strongly recommend to test newly created CSP simulation programs, especially those designed for transient simulations with simplified and coarsely meshed models first, before building and using memory-intensive elaborate spatially variably refined models.

REGULAR MESHES FROM PIXEL DATA

The fastest and simplest way to create 2D models in CSP is to use color-coded pixel images of your input geology. These representations are converted by a triangular-mesh generator inside of CSP (Triangulator object) into regular meshes for computations. Input geometries are drawn either directly in “pixel paint” programs or with drawing programs like Canvas (Aldus Freehand, Adobe Illustrator, MacDraw etc.). In these drawings, structures and horizons are color-coded and the regions of interest are clipped (and modified) using the public domain program NIH Image (Mac) or Scion Image (PC). NIH Image permits the output of selected image regions as 256-color pixel dumps in text file format. These files can be read by CSP. On unix / linux systems other drawing programs are available in the public domain. Below, the regular-mesh-input generation procedure is described in detail for NIH Image (Scion Image) and tips are given for drawing and manipulating geometries.

The following steps are required for the production of a color-coded text input file:

1. Drawing the Geologic Cross Section
2. Importing the Section into NIH Image
3. Color-coding the section
4. Output of the section as color-coded text file

These steps are described in detail in the following subsections. Steps 1, 2 can also be done in NIH Image.

1. Drawing the Geologic Cross Section

As the first step towards a CSP model, a geologic cross-section needs to be prepared in which each geological unit or structure has a unique color of the rainbow color scheme (In NIH Image: Options -> Color Tables -> Rainbow). The horizons must be
filled with a unique color and must not be framed with black lines as these would be interpreted as separate low-permeability units (see below). Care has to be taken to avoid blank areas between the horizons. The most convenient way of drawing a horizon with polygon tools is to use the same line color as the fill color of the polygon. Smoothing of the lines is recommended. However one needs to check afterwards with the magnification tool whether the horizons still overlap.

The generated drawing can be labeled further as a useful documentation of the model (publishing, presentation etc.). Since one may want to modify the section in the future or even build an adaptively refined version of it using the procedure (2) from above, it is advisable to keep a copy in the format of the original drawing program before proceeding with the next steps.

2. Importing the Section into NIH Image

NIH image can import both PICT- and TIFF files. Save a copy of your model file (with a new name) as TIFF file. Now open this file with NIH image and choose the Rainbow color palette. Once the file appears on the screen, use the selection tool (stippled quadrangle in the upper right of the tools menu) to extract the region of the geological section which you intend to input into your CSP model (or use the whole section). If the section is too large to fit on the screen use "Scale to fit window" from the Options menu before using the selection tool. One can also scale drawings with NIH Image. Since the model is build from pixel data, the screen size of the geological cross-section controls model resolution. For test runs, the model should not be larger than 100 by 100 pixels. This will lead to a computation size of about 30-50 MB RAM in CSP depending on how many physical variables the model has. A 1000 x 1000 image will translate into a 2000 x 2000 triangle mesh (4 millions), requiring about 1.5-2.3 GB RAM (Fig. 13).

3. Color Coding the Image

The constructor for regular-gridded CSP models (SuperGroup::SuperGroup(Triangulator& )) prompts the user for the name of the input text file and expects the input of property values of choice in the form of color-coded pixels. Use the paint bucket and pen tools to change colors in the image according to the 256-rainbow color vs. property value scheme. With this scheme, permeability values are assigned ranging from $10^{-21}$ m$^2$ to 1.0 Darcy ($10^{-12}$ m$^2$):

<table>
<thead>
<tr>
<th>Permeability (m$^2$)</th>
<th>Color Value</th>
<th>Color Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-21}$</td>
<td>255</td>
<td>BLACK</td>
</tr>
<tr>
<td>$10^{-20}$</td>
<td>0</td>
<td>WHITE</td>
</tr>
<tr>
<td>$10^{-19}$</td>
<td>236-254</td>
<td>RED</td>
</tr>
<tr>
<td>$10^{-18}$</td>
<td>216-236</td>
<td>ORANGE</td>
</tr>
<tr>
<td>$10^{-17}$</td>
<td>206-215</td>
<td>PEACH</td>
</tr>
<tr>
<td>$10^{-16}$</td>
<td>183-205</td>
<td>YELLOW</td>
</tr>
<tr>
<td>$10^{-15}$</td>
<td>126-182</td>
<td>GREEN</td>
</tr>
<tr>
<td>$10^{-14}$</td>
<td>75-125</td>
<td>AQUAMARINE</td>
</tr>
<tr>
<td>$10^{-13}$</td>
<td>45-74</td>
<td>BLUE</td>
</tr>
<tr>
<td>$10^{-12}$</td>
<td>1-44</td>
<td>AZURE</td>
</tr>
</tbody>
</table>

The desired integer color values (second column) are set by calibrating the pipette on the color bar in NIH image. The color value chosen by the pipette appears in the
“Info” window in the lower left hand corner underneath the color bar. Values with decimal places which may be output when NIH Image tracks color saturation (which is optional) are not permitted. One can vary values later in CSP once a mesh has been created and Group objects can be build for each uniformly colored region.

Ideally the extracted region should now be complete for output as text-formatted CSP input file. This is achieved using "File -> Export -> text" from the file menu. Before you output the file, test whether the color values are unique. The cross hair tool (+ symbol) will indicate the 256-color value when pointed at a horizon (Again, the tool should not indicate decimal color values, otherwise the text file cannot be interpreted by CSP). With the magnifying tool individual pixels of the drawing can be checked which are poorly visible in the default view. This is necessary when converting pictures from drawing-program formats to TIFF since minor changes may have occurred in the image. Units must not have black outlines (color value 255) ! - If one finds any errors, the color picker (pipette tool) and pencil tools can be used to replace erroneous colors. In many cases, one can also changes whole areas with the paint-bucket tool (the color for this tool is also chosen with the pipette; choose it either from the image or from the palette=color bar). Like in Photoshop or GIMP, the paint bucket will color a mono-chromatic region bounded by other colors. Such regions will be filled-in completely with the new color.

One should give individual structures or units enough thickness to achieve contiguity in the finite-element models (Fig. 13). In a steady-state model, oblique fractures must be interconnected pixel chains to be represented as continuous high-permeability pathways. In models with fluid sources and sinks (e.g., wells), these units must be at least 3 elements across to get meaningful finite-element results for linear element types.

![Fig. 13: Correct (a) and incorrect (b) representation of a fracture in a 2D pixel input image for the Triangulator mesher inside of CSP.](image)

4. Output of the section as color-coded text file

If the image satisfies the listed guidelines it can be exported as pixel dump in text format: Choose "Export" from the File menu and then click the text box. Keep a TIFF version of the geometry because it may need to be modified later. I recommend against using selected region output ("Export Selection as...text") as input files for
CSP since region size almost always differs between manual selections or they are slightly offset relative to one-another. This can render work useless if one later wants to compare simulation results. Instead, one extracts only one selection from the original geometry file and make copies with alternate color coding. This assures that all output files have the same size and the same reference frame.

Importantly, on older Macs, one should give the NIH Image application enough memory to work with larger drawings. This is done by (1) quitting NIH image; (2) selecting this application file in the respective folder; (3) choosing “Get Info” from the Apple file menu; (4) increasing the “current size” to 4000 k or larger dependent on the size of the image.

Finally, when the SuperGroup constructor reads the input file, it asks the user to specify the actual dimensions of the model. This is necessary because the pixel-data lack this information. In order to get optimal finite-element shapes, one should keep the aspect ratio, when specifying model dimensions.

**Spatially-Varially Refined Meshes from PSLG’s**

Constrained-conforming 2D Delaunay triangulations can be created from “parallel straight line graphs” with the very efficient and flexible program Triangle (J. Shewchuk, http://www.cs.cmu.edu/~quake/triangle.html). Please refer to the documentation of this program for details on how to operate it (on the command line type “triangleMesher –h” in the directory where the triangleMesh.exe lives). The utility program GeometryPreprocessor.exe by SKM is provided with CSP to create “.poly” input files for Triangle using “.dxf” files as input (DXF is a graphics interchange format created by AutoCAD. One must make sure that the DXF file format is compatible with the original DXF standard specification (new AutoCAD files are not, those up to version 12 are). A number of other rules must be adhered to when creating a DXF file for the GeometryPreprocessor program:

1. The first polygon in the drawing must be the bounding rectangle/graph (drawn as a closed polygon, not as a rectangle and insuring that the origin is in 0.0, 0.0).
2. The absolute dimensions of the drawing will be those of the final model. Change the drawing scales accordingly.
3. Draw polygons and lines, but do not smooth any of these. Also duplicate polygon segments must be avoided. Thus, at least one of two polygons which share a boundary, must be an open polygon.
4. “Triangle” expects minimum angle bounds for the triangular finite-elements it generates. These should not be violated by any intersections of lines in the drawing or by angles inside the polygons themselves (Triangle cannot discriminate between wanted and unwanted angle bounds and will therefore continue to refine the model until it runs out of numerical precision). The angle bound (option –qXX.) can be used to influence element size gradients in the mesh. The larger the bound the more smoothly transitions will be.
5. Triangle will need “material points” to assign properties to polygon- or line-delimited regions in the model. One can pick such points manually and create a regional property list inside of the “.poly” file (see instructions in Triangle
documentation for the format of this list). Alternatively, (Option 2 in G.P. menu) one can let the GeometryPreprocessor assign these points on the basis of specific fill colors applied to closed polygons in the drawing. The GeometryPreprocessor program will prompt the user for property values which shall be assigned for each color. However, preparing the material points list by hand (Option 1) is highly recommended over the more advanced option 2 since the conversion algorithm is much more robust.

6. One can also use drawing layers in DXF files to discriminate among objects which should or should not become part of the mesh. The GeometryPreprocessor expects at least one drawing layer specification. If the drawing program does not indicate the name of the default layer, the DXF file can be examined directly since it is in text format. Many drawing programs call the initial layer DEFAULT layer. Use input layer names in capital letters since DXF file content is not case sensitive.

7. In the “.poly” files one can specify a maximum triangle size for each PSLG region. Otherwise, Triangle will make the mesh as coarse as possible given the amount of geometric detail in the model. The GeometryPreprocessor’s Option 2 will assign default size bounds which can be edited before the meshing.

8. The PSLG must be exact! - Duplicate region boundaries must be avoided and filled polygons which overlap each other cannot be used to mimic cross-cutting relationships. Triangle removes duplicate points, but the more of such glitches occur the more likely it is that one will get a corrupted mesh. The error reporting of Triangle (options –cv), is very good.

9. When running Triangle (on a PC or Unix/Irix machine) one must give angle bounds for the triangulation and also request a “*.neigh” file needed by CSP to establish neighbors of each finite element. The suggested set of command line arguments for Triangle is (run triangle –h for what the options mean):

\[\text{triangle } -\text{a } -\text{A } -\text{e } -\text{p } -q\text{30. } -\text{n } -\text{O } -\text{V } \text{myfile.poly}\]

The GeometryPreprocessor (Option 2) program helps to identify faulty input geometries by writing a separate DXF file (*.faulty.dxf") containing polygons that violate the angle bounds for Triangle. This, however, is useful only for input geometries which consist entirely of closed polygons. Furthermore, Option 2 reports if it cannot find a unique material point for a polygon by outputting the coordinates of the problematic polygon as well as those of the polygon which contains the ambiguous material point. In most cases the reason for this type of error are duplicate or overlapping polygons. The GeometryPreprocessor (Option 2) will fail to detect small angles that arise where polygons intersect. Such locations (in model coordinates) will however be reported by ‘Triangle’ which is useful for fixing these errors.

Triangle creates “.node, .ele” and “.neigh” files with a number corresponding to the refinement level (one can refine the same mesh several times to test for mesh convergence and mesh sensitivity of a computation). With the verbose meshing option, some useful mesh diagnostics will be reported in the meshing process. One can also examine the quality of meshes visually, with the program “ShowMe”. This
program was also written by J. Shewchuk but runs only on platforms with X-windows capabilities.

While a Triangle output mesh may be perfectly fine in terms of angle bounds etc., it may still contain knots where clusters of extremely small elements were created. This is most certainly the case when the reported mesh statistics indicate the existence of elements with an area below $1.0e^{-3}$. In this case two very closely spaced points in a polygon, or a failed point snap leaving a minute gap in the model, are the most likely culprits.

“Triangle”-generated meshes are imported into CSP using its mesh interface which is encapsulated into the class `MeshInterface`. The code for this is simple:

```cpp
MeshInterface mesh_interface;
CSP_VSet vset;
char fname[200];

cout <<"\nmain: Enter name of 'Triangle' input file set: ";
cin >> fname;
mesh_interface.ReadTriangle2DMesh( fname, vset );

SuperGroup my_new_model( vset );
```

The method `ReadTriangle2DMesh()` reads the set of input text files (.node, .ele, .neigh) created by Triangle and puts the results into the CSP_Vset with the name ‘vset’. Property values will be stored and interpreted as the element variable “permeability”. A fast way of arriving from this data at desired values of other properties is to first create element Group objects for each of the permeability values and then use the Group class interface to modify the corresponding regional properties:

```cpp
set<CSP_String,less<CSP_String> > group_names;
my_new_model.GroupsFromPropertyValues( "permeability", group_names );
```

In this example, the names of the new groups are output to an STL set of group names.

**Trouble Shooting.** There are many checks that CSP performs on mesh topology and additional checks can be invoked while running Triangle by using the –C option. If one still gets a message from the Solver, e.g., `DIAGONAL NOT POSITIVE`, the most likely reason for this is that one of the region points failed to lie inside the target region and the Triangle flag –O (no holes) was specified. In this case, Triangle assigns a zero material property value to the Elements in that region. Clearly this value will be propagated into the global solution matrix. One can see whether this is the case by examining the “*.1.ele” file with the material properties of the elements.
3D MESHES GENERATED BY GOCAD

An interface between CSP and GoCad is provided by the MeshInterface and the SuperGroup class objects. CSP can write GoCad ‘TSolid’ and ‘TSurf’ files and it can read GoCad output in the same format. Clearly, the GoCad program has a very extensive documentation by itself. File formats are briefly explained in the publically available part of the GoCad documentation at (http://www.ensg.u-nancy.fr/GOCAD/Welcome.html).

Geometry building in GoCad is difficult, not in terms of creating individual surfaces but in terms of achieving a volumetric decomposition with correct surface intersections as is needed before a useful 3D mesh can be generated. Hence, the program Rhino (R. McNeel & Associates) is recommended as a more flexible tool to build initial three-dimensional object descriptions for input into GoCad where these geometries can be meshed.

From Rhino

If closed surfaces have been prepared for 3D volumetric meshing in GoCad or other meshing tools one must first check whether these surfaces have any holes i.e. naked edges. If Rhino is used for the geometry generation, choose: Analysis -> Edge Tools -> Show Naked Edges.

- Naked edges can be capped by surfaces which are joined to repair the closed surface.
- While blending in Rhino gives very aesthetic caps and transitions, care should be taken since once such surfaces are meshed an extremely large number of triangles arises in the regions with tiny radii. GoCad has difficulties with triangulated surfaces over which triangle size varies a lot.

When one outputs Rhino geometry as raw triangles (File -> Export Selected -> filename and dropdown menu option “triangulated surfaces (*.raw)”), as is necessary to bring these objects into GoCad, Rhino must first create triangle meshes from the analytical NURBS curves (non-uniform regular B-splines) which it uses internally to describe geometry. This process is controlled by the output menu. Rhino will first break the surface down into quadrilaterals, the properties of which the user can control. These can be viewed with the preview command. The actual triangles will only be displayed in part.
The options and their desirable settings depend on the goals of the modeling. A highly resolved surface will lead to 3D meshes with a very large number of tetrahedra. The options in the Rhino dialog box have the following meaning:

- **Max angle**: Angle between NURBS and actual triangle plane. The smaller, the more triangles are generated and the better the surface is matched.
- **Max aspect ratio**: Aspect-ratio (width over height) of quadrilateral which is broken into triangles.
- **Min edge length**: Lower bound on quadrilateral edges such that the meshes do not get too fine.
- **Max edge length**: Upper bound which can be used to force a certain degree of refinement, especially for flat surfaces where “Max angle” has no effect.
- **Max distance**: Gives alternative way to “Max angle” in terms of specifying how well the mesh will actually match the triangulated surfaces.
- **Min initial grid quads**: Sets the minimum size of the quadrilaterals on the basis of which the matching iterations will be performed.

### CSP Geometry Preprocessor

Again the GeometryPreprocessor module (GeometryPreprocessor.exe) is needed to transform the Rhino output into input files for GoCad. Option 3 or 4 are currently used for the processing of triangulated surfaces from Rhino. Use the possible coordinate transformation only for models relating to the Sigma mine project. The preprocessor will list the objects by their Rhino name prompting the user for a specification of the GoCad object type (see GoCad documentation for an explanation of the deeper significance of the available options for geological objects). The preprocessor will write a GoCad TSurf file for each surface object. These files can be read directly in GoCad using the: File -> Load Object option.
In Gocad

Rule 1: Save project after each processing step since GoCad is prone to crash without warning. Once a triangulated surface from Rhino has been imported into the current GoCad project, it must be examined again for its suitability for meshing. If a closed surface shall be meshed using tetrahedra, it first must be tested for holes which may have accidentally been created when Rhino carried out the triangulation of the NURBS surfaces. The steps in GoCaD are:

1. Import Tsurf objects from the CSP Geometry Preprocessor module into GoCad.

2. Test surfaces for holes by choosing Surface Mode -> Graphics -> Show Borders in the property visualization panel on the left. To get the properties for the surface of interest, this surface’s name must be selected in the box above the property panel.

3. Complicated closed surfaces should be tetrahedralized individually, before creating a model which is meshed as a whole. This way, potential errors are located and detected before time is invested into complex model building.

4. If there are problems meshing individual volumes, - under the Edit menu - options can be found to improve surface meshes (Since GoCad cannot move nodes on surface meshes when building the volumetric meshes, this may help). Try the options:

   Edit   -> Beautify    -> Beautify triangles
   -> Switch Triangle
   -> Remove Mercedes.

   to improve surface meshes while examining these in the display window.

5. Create a geological model from closed surfaces, open surfaces, and the bounding box (this will not copy properties which must therefore be assigned to the model at a later stage). The model is created by:

   Model -> From Surfaces (in the Surface mode).

6. Change from General to Solid mode and mesh the model by calling:

   New (Solid) -> From Model.

Various options exist: Choose all three ! - “to densify”, “get constraint faces”, and “copy properties”. If the constraint faces are omitted, GoCad will only output a single TVOLUME object such that CSP will not be able to discern different model regions. Worse ! – CSP will not be able to read the file correctly, because GoCad will present the property information in a different way.

With option “To densify” selected, GoCad may crash, especially when trying to allocate memory for its expensive octrees. However, this option must be selected, because, otherwise, GoCad will not insert node points inside the surface-delimited
volumes which define the model. If “to densify” is omitted pointy tetrahedra will reach from one side of a volume to the other!

7. Assign material properties to the model. Only one property named “permeability” is needed by CSP. From the space subdivision given by this property other material regions can be derived later after defining CSP groups. Choosing Property -> Create

brings up a dialog which allows to define the property. Select each of the surface assigning properties by entering property name (“permeability”), unit, and NODATA Value. The latter value permits to assign property values which are unique to the tetrahedralized volumes delimited by the closed surfaces to which the property was assigned. However, only one NODATA value is currently recognized per Tsolid input file.

8. Output model as GoCad TSolid object (Save Object As...) with extension *.so. Open this file in a text editor to see whether the material properties were assigned correctly and whether it contains the right number of TVOLUME descriptors corresponding to the volumina delimited by the input surfaces. Warning: If the property “permeability” is missing or other properties are included, the standard CSP MeshInterface object will not be able to read the “Tsolid” file.

When the GoCad Tsolid file is generated on a UNIX system and CSP is used on a PC, care must be taken to ftp this textfile correctly to the PC file system. The “ascii” option must be used in ftp in order to turn unix line breaks into PC-suitable ones. If the file is opened in CodeWarrior, the line breaks can be changed after the transfer using the respective editor bar, drop-down menu.

Caveats

The calculate range dialog in the GoCad property menu does not work. If fractures are to be included as planes or other surfaces shall be output explicitly for computations, those must be used which GoCad creates in conjunction with the generation of the Solid. Only these are compatible with the solids in that their triangles represent the faces of the adjacent tetrahedra. If this approach is used, however, there appears to be no way of correctly transferring assigned property values to the TSolid and TSurf output files.

3D MIXED TETRAHEDRAL & TRIANGULAR ELEMENT MESHES FROM GOCAD

In three-dimensional fluid-flow models, it is possible to represent planar zones which are more permeable than the surrounding rock by triangular surface meshes. In such models, each triangular element shares the nodes of, and lies in between two adjacent tetrahedral elements which still have eachother as neighbors. The neighbors of the triangular elements forming the surface mesh are assigned as they would be in 2-D models. Thus, only triangles feature in neighbor lists of triangles and only tetrahedra feature in the neighbor lists of tetrahedra.
Since the triangles are oriented arbitrarily in space, a special mapping from 3D to the triangles local (2D) coordinate system must be applied. Thus far, this is only implemented for the QuadraticTriangle element. The mapping is switched on by constructing this element with the boolean value “true” as constructor argument (see example code 14).

Mixed triangular-tetrahedral meshes will contain far less elements than equivalent meshes where planar objects have a volumetric representation. As an alternative application of the mixed surface-volume mesh capability, one may use it to compute and assign boundary conditions on irregular 3D boundaries or interpolate displacements on fault planes. Importantly, CSP also supports the input of material property values from surface meshes such that these can be used as input data or for comparison with modeling results. This capability facilitates a direct comparison of modeling results with surface-based information from mining or seismics. Through the CSP interface to GoCad one can also use its smooth interpolation capabilities to the benefit of CSP (in GoCad, scattered material properties data can be directly interpolated on surfaces).

With surface elements, one should also be able to model surface-normal growth processes or moving interfaces in a model: Erosion, mineral precipitation, friction etc.

**Implementation (see example14.cpp)**

1. **MeshInterface** class object reads GoCad surfaces with associated property information and stores this information in a CSP_VSet.
2. **CSP_VSet** object: If necessary, node property data (such as vein thickness) is converted to element property data as the latter cannot be assigned in GoCad.
3. **CSP_VSetConverter**: Converts CSP_VSet for element type which is being used.
4. **CSP_MeshManager**: Renumbers nodes in CSP_VSet, identifying to which nodes inside the already existing SuperGroup the new nodes correspond to. This comparison is carried out for the node coordinates up to a user-defined precision. If corresponding coordinates can not be found in the SuperGroup object, a fatal error is reported.
5. **CSP_MemoryManager**: Assigns the property data to the SuperGroup model. This requires the creation of new storage for element variables assigned to the new planar elements.
6. **SuperGroup**: Special Group interface `AddGroupFrom( CSP_VSet, element type )` builds group from supplied element and node lists. This method uses the new element ID’s to form a Group object which corresponds to the added surface mesh. It entails the steps from above, using the corresponding Managers.

Example code 14, illustrates how mixed meshes from GoCad are used in a 3D steady-state fluid-flow simulation. Very importantly, when exporting the surfaces from GoCad, only surfaces created together with the solids can be used, because GoCad modifies original surface meshes when it creates the solids.
OUTPUT VIA THE VTK INTERFACE

A cost-effective way to visualize CSP simulation results is instrumentalized by the CSP_VTK_Interface object for the visualization toolkit (VTK) supported by kitware.com. VTK is a very powerful C++ library of visualization algorithms and tools for structured and unstructured data, albeit with somewhat limited interactivity. In stead of running a GUI based viewer, one writes visualization scripts in Tcl, Python, Java or C++ and executes these to get results. Apart from an impressive sample suite which comes with VTK, a suite of SKM-generated Tcl scripts demonstrating many features relevant to CSP-related visualization issues, is provided with the CSP Distributed Example set.

As the perhaps most severe limitation as seen from a CSP perspective, VTK does not directly support the display of scalar element properties – oddly – as this is easily done in OpenGL by using flat shading. However, VTK’s data model does not support this. Due to this limitation, I have chosen to output CSP element properties as point data to VTK. These can therein be connected by means of a Delauney triangulation if continuous scalar fields are to be displayed. However, material boundaries will be blurred using this approach.

The use of the interface to VTK is simple. The interface object is instantiated and its interfaces are called as is shown in the code example below for an instance of a SuperGroup named “box” and for a Group identified by the string “fault”:

```cpp
CSP_VTK_Interface  vtk_output;
//                          S.G.  filename        variable name   timestep
vtk_output.OutputDataToVTK( box, "nodal_test",      "nodal_test", 0 );
vtk_output.OutputDataToVTK( box, "fluid-pressure",  "fluid pressure", 0 );
//                     S.G.  groupname  filename   variable name  timestep
vtk_output.OutputDataToVTK( box,"fault","nodal_test","nodal_test",tstep );
```

OUTPUT VIA THE JPEG INTERFACE

For convinience and to save space when monitoring runs, scalar variables from two-dimensional models can be output directly to JPEG image files, by calling corresponding methods of the SuperGroup (.OutputDataToJPG( char*, char*, long )). In this operation, CSP maps the variable values to a regular grid corresponding to the raster image and it associates a greyscale- or rainbow color scheme with these data. By default, the actual value range determines the color-mapping. Red (black) generally marks the highest values, and blue (white) marks the lowest. Alternatively, a fixed value range can be set by the user, if images are to be strung together as animations.

A caveat of the JPG interface is the number of pixels of the auxiliary grid. Pixel size is automatically set to 1.6 of the inner radius of the smallest element in the finite element mesh. If a model locally contains very small elements, the memory required for the grid will be excessive causing CSP to terminate with an error message.
5. Initial and Essential Conditions

A boundary of a CSP SuperGroup model or on the outside of a Group sub-region thereof, is defined by those finite-elements which have at least one face (segment) at this boundary. To such elements or their faces a range of boundary conditions can be applied.

The simplest way of constraining a model is to fix values of dependent variables at boundary nodes or some nodes inside. These “first order” constraints which must be satisfied by the solution are called “Dirichlet” conditions and are incorporated into the resulting system of linear algebraic equations by simple substitution, to the effect that the number of degrees of freedom is reduced. Since nodal values can be fixed inside a model one speaks of “essential” rather than boundary conditions. Dirichlet conditions inside of CSP can be assigned with the SuperGroup methods (for details see HTML documentation of the SuperGroup class):

```cpp
void AssignBoundaryFlags( SG_BOUNDARY, const char*, DATA_STYLE );
void AssignBoundaryValues( SG_BOUNDARY, const char*, DATA_STYLE, double, double );
void AssignBoundaryValues( SG_BOUNDARY, const char*,
const VectorVariable&,
const VectorVariable& );
void AssignConstraint( const char*, long, const ScalarVariable& );
void AssignConstraint( const char*, long, const VectorVariable& );
void AssignConstraint( const char*, long, const TensorVariable& );
void AddBoundaryNormalFluxes( SG_BOUNDARY, const char*,
double, double ); // influxes are positive
bool AssignBoundaryValues3D( SG_BOUNDARY, const char*, DATA_STYLE,
double, double, double, double );
bool AssignBoundaryValues3D( SG_BOUNDARY, const char*,
const VectorVariable&, const VectorVariable&,
const VectorVariable&, const VectorVariable& );
void ChangePropertyStatusInGroupToWhere( const char*, const char*,
DATA_STYLE, DATA_STYLE );
void ChangePropertyStatusInGroupToWhere( const char*, const char*,
DATA_STYLE, double, double );
```

or, when carrying out group-restricted computations with methods of the involved Algorithm or CSP_Algorithm class objects such as:

```cpp
domain_pressure.SetOperandEssentialStatus( "fluid pressure", BOUNDARY );
```

which is a statement taken from the code example “example1.cpp”. *One must set a Dirichlet value at least for one Node*, otherwise there will be an infinite number of correct solutions to the integration problem and the solver will produce an arbitrary result. This is the reason for why one speaks of essential conditions.

In order to assign more complex constraints like fixed flow rates across interfaces or fixed chemical source rates etc. another approach is needed, since these are constraints to be imposed on the first or higher-order derivatives of the dependent
variable. Thus, one also refers to them as ‘second order’ or ‘higher order’ essential conditions. A little digression is warranted in order to explain the handling of such conditions:

Think for a moment about the difference between specific properties such as the density of inhabitants per km\(^2\), and overall properties, such as the number of inhabitants of a village. A finite element represents a certain volume of rock with specific properties defined per unit volume such as the heat capacity or a fluid volume source term (m\(^3\) m\(^{-3}\) s\(^{-1}\)). Such properties must be integrated over the element volume like the permeability, thermal conductivity etc. Likewise, one calculates how much quartz is precipitated per unit volume of rock (assuming a quartz-saturated fluid) with an Interrelation which multiplies the fluid mass flux with time and the solubility-gradient in the direction of flow. However, if one is specifically interested in the total amount of quartz precipitated within an element, one must multiply the calculated amount with the element’s volume.

**Neumann “second order” essential conditions.** The area of the face of an element (Fig. 15) comes into play when essential conditions are distributed as sources or sinks to node or element variables. To apply such Neumann boundary conditions (also called non-essential conditions), one has two alternatives:

1. One can distribute the specific face-normal quantities as point sources on the boundary nodes of the element. This necessitates to adjust these quantities since the boundary face area is rarely unity, and because they are added to several nodes at once. Thus, for a boundary flux, \(q\) per unit area, the nodal contribution from a linear triangular element to the righthand side of the global equation has the form

![Diagram of boundary elements in CSP model](image)
where $t$ is the element thickness, $L$ is the boundary segment length (facial area), $A$ is the element area (volume in 3D), and $N_b$ is the number of nodes at the boundary. Neumann boundary conditions are also assigned with `AssignBoundaryValues()` and require a separate variable to hold the prescribed boundary in- or outfluxes specified as boundary-normal components. These are added in the global matrix accumulation procedure using the differential operators as follows:

\[
\{ F \} = \{ F \} + \frac{tL}{N_b} \begin{cases} q \\
0 \end{cases}
\]

Initial conditions such as a fluid pressure distribution or initial stresses at the onset of a transient computation must also be specified by the user. CSP initializes any physical variable specified in the variables "*.txt" file with a default value of `NAN` (not a number) and a `DATA_STYLE` flag of `PLAIN`. With these settings, one cannot compute anything, because a `NAN` value will be the result, highlighting any not initialized variables as was the design intention.

To initialize variables one must write a statement like

\[
\text{supergroup\_instance.InputUniformValueWhere} \ (\text{PLAIN}, \ "\text{fluid pressure}\", \ 0.0 \ ) ;
\]

into the program. The `PLAIN` flag specifies that the supplied 0.0 value of the variable "fluid pressure" (third method argument) is assigned only where the target variable is flagged as such. Thus, where the variable is flagged as `DIRICH` or `INIT_COND` it is not overwritten. As discussed earlier, this capability to differentiate variable treatment on the basis of flags is needed to apply essential conditions. With respect to initial values, it permits changing flags within overlapping Groups. To change conditional flags in the entire SuperGroup, use the method

\[
\text{supergroup\_instance.ChangeConditionFromTo} (\"fluid pressure\", \text{DIRICH}, \text{PLAIN} ) ;
\]

This example will remove Dirichlet boundary conditions assigned to the distributed variable "fluid pressure".
**Dirichlet essential conditions.** CSP allows to assign essential conditions (= constraint values) anywhere within a SuperGroup model. In CSP one assigns boundary conditions using the overloaded AssignBoundaryValue() and AssignBoundaryValue3D() methods of the SuperGroup.

```c
moab.AssignBoundaryValues( LEFT,  "fluid pressure", DIRICH,  1.0e+7, 0.0 );
moab.AssignBoundaryValues( RIGHT, "fluid pressure", DIRICH,  1.0e+7, 0.0 );
```

In this example, scalar Dirichlet boundary conditions are assigned to the variable “fluid pressure”. The two input values allow to specify linear fluid-pressure gradients along the left and right vertical model boundaries. The floating-point values supplied as third and fourth method arguments are the pressure values which shall be fixed. In the example code, values of the Node variable “fluid pressure” will be varied linearly from \(10^7\) to 0 between the first and the last Node situated at the left and right model boundaries, respectively. A few important additional CSP conventions apply with regard to the boundary labeling and boundary condition assignment:

1. A righthand-rule coordinate system is used always. In 2D, the TOP boundary is the one with the highest y-coordinate values since Y points upward reflecting altitude.
2. In 3D, X points to the right, Y to the top, and Z to the front, adhering to a right-hand coordinate system.
3. In 3D, all faces of the model are numbered counterclockwise, looking down the Z-axis from the front into the model.
4. These faces are labeled BOTTOM, LEFT, FRONT, RIGHT, BACK and TOP.
5. Face BOTTOM has the corners CNR1, CNR5, CNR6, CNR2.
6. Face LEFT has the corners CNR1, CNR4, CNR8, CNR5.
7. Face FRONT has the corners CNR1, CNR2, CNR3, CNR4.
8. Face RIGHT has the corners CNR2, CNR6, CNR7, CNR3.
9. Face BACK has the corners CNR5, CNR8, CNR7, CNR6.
10. Face TOP has the corners CNR3, CNR7, CNR8, CNR4.

Corner locations and boundary gradients are only meaningful if the model is rectangular. If the boundary faces are tilted more than 45° relative to an orthogonal-normal orientation, CSP may wrongly recognize boundaries. However, in this case the boundary may be identified and flagged accordingly in the input data. This way the user can specify named boundary entities already in the graphics program in which the geometry is build. Please contact SKM if you are in need of this capability (s.matthai@ic.ac.uk).

*With a little bit of original source code, one is able to compute essential conditions for 3D models using surface-based computations on each model boundary. Ask Stephan or Sebastian (geiger@erdw.ethz.ch) about this.*

**Periodic boundary conditions** can be applied in CSP3D but this capability has not been rigorously tested. The SuperGroup method:
bool PeriodicBoundary( const char* phys_variable, SG_BOUNDARY A, SG_BOUNDARY B );

will try to make a model periodic across 2 opposite boundaries that must have the identical number of nodes and coordinate positions on the coordinate axis parallel to the target boundaries.

Periodic boundary conditions are used to simulate infinite but periodic models. This is done by connecting the mesh at two or more identical and diametrically opposite model boundaries such that the model “wraps” around itself. This can be implemented in the following steps.

1. When the element matrices are accumulated into the global solution matrix, the nodal entries from boundary \( B \) (opposite of boundary \( A \)) are accumulated to the corresponding locations at boundary \( A \).

2. (The degrees of freedom in the global solution matrix are reduced by the number of nodes on boundary \( B \)).

3. An \( x_i = u_i - u_j = G \) type replacement is applied on boundary \( B \) for any essential condition which is specified on boundary \( A \).

4. A single, additional Dirichlet (nodal) essential condition must be specified, because, otherwise, there is no unique solution to the partial differential equation.

Periodic boundary conditions are only meaningful when there is a representative elementary volume (REV) for the analyzed material. This applies only if the model is scale-invariant at any scale larger than the representative elementary volume.

The **determination of a permeability tensor** for a given cross-sectional model of rock can also be achieved elegantly without periodic boundary condition:

1. One uses a drawing program to draw an exact representation of the flow geometry in a 2D circular region. The region needs to be circular, if one would like to determine the directional permeability. Individual fractures are represented as closed polygons with correct aperture. The circular region is placed in the center of a quadratic bounding box the diagonal of which is twice the diameter of the circle. A number of representations of this geometry are created by rotating the circular region relative to the bounding box by up to 90 degrees. These models will later be used to measure the directional permeability.

2. The geometries are output from the drawing program as DXF files and converted into parallel-straight-line-graph text files (.poly) by a utility program. In these PSLG files, regional material properties are assigned using material points which must be specified for each polygon. This procedure is discussed in detail in Chapter 4.

**Developer Note:** PeriodicBoundary() tells the CSP_MeshManager that a pair of model boundaries have to be treated as periodic. When NodeNumberVectors are created in the accumulation process this is taken into account such that the finite-element matrices are accumulated into the correct locations.
6. Defining & Solving Finite-Element Equations

Various operators facilitating the composition of partial differential equations (PDEs) in their finite-element forms are supplied with CSP3D3.0. The new Algorithm class largely succeeds in separating the definition of such operators from their element dependent accumulation. Using this new approach, PDE operators derived from the base classes MathOperatorLHS and MathOperatorRHS, respectively, are provided as a “seed” library to be extended by users and developers. This library is described in more detail in Chapter 7. This chapter just illustrates the general approach of solving PDEs with the finite element method.

When the finite-element method (FEM, e.g., Huyakorn and Pinder, 1983; Istock 1989; Zienkiewicz and Taylor, 1994; Bathe, 1996) is used to solve a partial differential equation (PDE), which describes the distribution of a field variable (e.g., fluid pressure, solute concentration) across a domain of interest, this domain is discretized into a finite number of sub-regions, each of which is represented by a single finite element. All finite elements together form the finite element mesh. Since PDEs describe how a dependent variable changes as a function of independent variables, one needs to integrate them, to compute the desired values of the field variable. In the FEM, field variables are integrated element by element, such that the incremental change across the mesh yields the variation of the field variable in the model. However, in stead of integrating an equation which represents an exact solution to the PDE of interest, one integrates some piece-wise continuous functions of which one hopes that they are suited to model the PDE. For this approach to be successful, the functions must have a range of properties which are discussed in detail in the Developer’s Guide. The general user of the FEM, needs to know that the functions must be differentiable to at least one order higher than the order of that derivative of the solution which is wanted for this solution to be exact. This requirement reflects a side effect of the finite-element discretization: Depending on the order of the functions, across element boundaries discontinuities may exist in the derivatives of a solution and most certainly in going across corner nodes into adjacent elements. For this reason it may be impossible to find a germane derivative of a solution at a node (the derivatives computed for each of the elements which share the node will differ from one another). Implications of this for calculating fluxes, stresses and strains are discussed in the Developer’s Guide and in the description of the Finite Volume library (FV, Chapter 7). The FV method represents a complementary approach to finite-elements which capitalizes on the continuity of solution derivatives within finite elements.

This section explains the basics of the Galerkin finite-element method for a linear triangular element as is implemented as the LinearTriangle class in CSP, comprising common matrices, vectors and integration procedures. The finite-element forms of second order partial differential equations are discussed approximating the physics of fluid flow through porous media and structural mechanics.

A small set of PDE operators representing volume, surface and time integrals over changes in the dependent or independent variables suffices to compose finite-element simulations of fluid flow, heat flow, and linear elasticity. The finite-element forms of
such common PDE operators have a standard notation in the finite-element literature as is introduced below. In the CSP_Algorithm-related way, they were associated with the Triangle class, identified by enumerations, and used in conjunction with the CSP_Algorithm. New subclasses of the MathOperatorLHS and MathOperatorRHS objects have the same purpose but are much more versatile. They constitute the PDE operator library to be used together with the new Algorithm class.

<table>
<thead>
<tr>
<th>Notation</th>
<th>PDE Operator</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic operators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>area of 2D element, volume in 3D</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>material property matrix</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>elasticity matrix</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>Jacobian matrix</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>interpolation functions (i.f.)</td>
<td></td>
</tr>
<tr>
<td>DN</td>
<td>i.f. derivative matrix (N×N_i)</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>i.f. deriv.m. 2dof per node (dim x n)</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>scalar property value</td>
<td></td>
</tr>
<tr>
<td>composite operators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[K]</td>
<td>A B \mathrm{^T} DB</td>
<td>stiffness matrix</td>
</tr>
<tr>
<td>[C]</td>
<td>A DN^{\mathrm{T}} KDN</td>
<td>conductance matrix</td>
</tr>
<tr>
<td>[S]</td>
<td>A/3 N r N</td>
<td>lumped capacitance matrix</td>
</tr>
<tr>
<td>{Q}</td>
<td>source-sink terms</td>
<td></td>
</tr>
<tr>
<td>[V]</td>
<td>advection-dispersion matrix</td>
<td></td>
</tr>
<tr>
<td>{u}</td>
<td>flux, displacement</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Notation used in the description of finite-element method.

Below, these operators are explained mainly for two-dimensional problems. \( N_i \) are the element interpolation functions which are used to model variations of the dependent variable, \( f \), inside of each element, such that

\[
f(x, y) = \sum_{i=1}^{n} N_i(x, y) \phi_i
\]

\( f(x, y) \) is the approximated value of at the point \( x, y \) inside of the element, calculated as the sum of the products of \( f \) values at the nodes with the corresponding interpolation functions for these nodes. \( n \) refers to the number of nodes, and basal indices denote node numbers local to the element. Since the discussion focuses on individual finite-elements, the \( (e) \) superscript is omitted which usually denotes element-, as opposed to global matrices.

\( A \), the area of a two-dimensional element is equivalent to the spatial integral over the interpolation functions:

\[
A = \int \int N \, dx \, dy
\]

\( DN \) is the matrix of the first spatial derivatives of the element interpolation functions and has the general form (in 2 dimensions):

\[
[B] = \begin{bmatrix}
\frac{\partial N_1}{\partial x} & \ldots & \frac{\partial N_n}{\partial x} \\
\frac{\partial N_1}{\partial y} & \ldots & \frac{\partial N_n}{\partial y}
\end{bmatrix}, \quad \begin{bmatrix}
\frac{\partial N_1}{\partial x} & \frac{\partial N_1}{\partial y} \\
\vdots & \vdots \\
\frac{\partial N_n}{\partial x} & \frac{\partial N_n}{\partial y}
\end{bmatrix}^T
\]
**MTRL** is a symmetrical material property matrix with a size of space dimensions squared. It can be used to represent scalar, vector, or tensor properties. For example, an anisotropic hydraulic conductivity, $K$, in a two-dimensional element is expressed as:

$$
[MTRL] = \begin{bmatrix}
K_x & 0 \\
0 & K_y \\
\end{bmatrix}
$$

$J$ is the Jacobian matrix for the translation of the local coordinate system $(\varepsilon, \eta, \zeta)$ into global coordinates $(x, y, z)$.

$$
\begin{bmatrix}
\frac{\partial N_1}{\partial \varepsilon} & \cdots & \frac{\partial N_n}{\partial \varepsilon} \\
\frac{\partial N_1}{\partial \eta} & \cdots & \frac{\partial N_n}{\partial \eta} \\
\frac{\partial N_1}{\partial \zeta} & \cdots & \frac{\partial N_n}{\partial \zeta}
\end{bmatrix}
= \begin{bmatrix}
x_1 & y_1 & z_1 \\
:\ & \vdots & \vdots \\
: & \vdots & \vdots \\
x_n & y_n & z_n
\end{bmatrix}
$$

Matrix contributions calculated for a straight-sided reference element are translated to the global coordinate system using the inverse of the Jacobian, the transposed inverse of $J$, and $J$’s determinant. In CSP, this procedure is applied in all higher-order elements. It is further discussed below and in the Developer’s Guide.

$N$ is a 1 by $n$ matrix (vector) with element interpolation function values at point $x, y$:

$$
[N] = \begin{bmatrix}
N_1 & \cdots & N_n
\end{bmatrix}
$$

Test function products of the form $N^T N$ are often required when properties are integrated over an element.

**EXAMPLE: STEADY-STATE FLUID PRESSURE**

The variation of fluid pressure in a domain is the sum of local fluid pressure variations in the $x, y, z$ directions and their perturbations by local fluid sources and sinks:

$$
\frac{\partial}{\partial x} \left( K_x \frac{\partial p}{\partial x} \right) + \frac{\partial}{\partial y} \left( K_y \frac{\partial p}{\partial y} \right) + \frac{\partial}{\partial z} \left( K_z \frac{\partial p}{\partial z} \right) = Q
$$

If one works with hydraulic conductivities, $K$, which are constant over the finite elements, the steady-state pressure equation simplifies to the elliptic PDE

$$
K^{(e)}_x \frac{\partial^2 p}{\partial x^2} + K^{(e)}_y \frac{\partial^2 p}{\partial y^2} + K^{(e)}_z \frac{\partial^2 p}{\partial z^2} = Q^{(e)}
$$
also known as Laplace equation and abbreviated using the \textit{nabla} operator

\[ K \nabla^2 p = Q \]

A global matrix expression arises from the element by element contributions:

\[ [K][p] = [Q] \]

where \( K \) is a global matrix of the combined element conductances, \( p \) is the solution vector which will hold the nodal values of the dependent variable “fluid pressure”, and \( Q \) is an elemental fluid source or sink. The exact form of the conductance matrix is dependent on the dimensionality of the model and the type of finite element which is used in the computation. When global element coordinates are used a two-dimensional conductance matrix has the form

\[
K^{(e)} = \int_A \left[ \begin{array}{c c c}
\frac{\partial N_1}{\partial x} & \frac{\partial N_1}{\partial y} & \ldots & \frac{\partial N_n}{\partial x} \\
\vdots & \ddots & \ddots & \vdots \\
\frac{\partial N_n}{\partial x} & \frac{\partial N_n}{\partial y} & \ldots & \frac{\partial N_n}{\partial x}
\end{array} \right] \left[ \begin{array}{cc}
K_x & 0 \\
0 & K_y
\end{array} \right] \left[ \begin{array}{c c c}
\frac{\partial N_1}{\partial x} & \frac{\partial N_1}{\partial y} & \ldots & \frac{\partial N_n}{\partial x} \\
\vdots & \ddots & \ddots & \vdots \\
\frac{\partial N_n}{\partial x} & \frac{\partial N_n}{\partial y} & \ldots & \frac{\partial N_n}{\partial x}
\end{array} \right] dxdy
\]

This was already presented in Chapter 2. This equation is solved by (associative but not commutative) matrix multiplication and the integral is just equal to the area, \( A \), of the finite element. Thus,

\[
K^{(e)} = \frac{1}{2A} \int_A \left[ \begin{array}{c c c}
\frac{\partial N_1}{\partial x} & \frac{\partial N_1}{\partial y} & \ldots & \frac{\partial N_n}{\partial x} \\
\vdots & \ddots & \ddots & \vdots \\
\frac{\partial N_n}{\partial x} & \frac{\partial N_n}{\partial y} & \ldots & \frac{\partial N_n}{\partial x}
\end{array} \right] \left[ \begin{array}{cc}
K_x & 0 \\
0 & K_y
\end{array} \right] \left[ \begin{array}{c c c}
\frac{\partial N_1}{\partial x} & \frac{\partial N_1}{\partial y} & \ldots & \frac{\partial N_n}{\partial x} \\
\vdots & \ddots & \ddots & \vdots \\
\frac{\partial N_n}{\partial x} & \frac{\partial N_n}{\partial y} & \ldots & \frac{\partial N_n}{\partial x}
\end{array} \right] dxdy
\]

For straight-sided triangular element there is a simple integration formula which allows to integrate element matrices on the basis of the order of the interpolation functions:

\[
\int_A \left( N_1 \right)^a \left( N_2 \right)^b \left( N_3 \right)^c dA = \frac{a!b!c!}{(a+b+c+2)!} 2A
\]

\( a,b,c \) are the exponents of the interpolation functions, and the factor 2 refers to the dimension of the finite element.
If local coordinates are used allowing for curved boundaries of higher-order elements, local coordinates on a reference element are transformed into global coordinates by multiplication with the inverse of the Jacobian matrix and its determinant. The conductance matrix for a two-dimensional then element becomes:

\[
\overline{K}^{(e)}_{nn} = \int_{-1}^{1} \int_{-1}^{1} \begin{bmatrix}
\frac{\partial N_1}{\partial \xi} & \frac{\partial N_1}{\partial \eta} \\
\vdots & \vdots \\
\frac{\partial N_n}{\partial \xi} & \frac{\partial N_n}{\partial \eta}
\end{bmatrix} \begin{bmatrix}
K_x & 0 \\
0 & K_y
\end{bmatrix} \begin{bmatrix}
\frac{\partial N_1}{\partial \xi} & \frac{\partial N_1}{\partial \eta} \\
\vdots & \vdots \\
\frac{\partial N_n}{\partial \xi} & \frac{\partial N_n}{\partial \eta}
\end{bmatrix} \left| J \right| d\xi d\eta
\]

This integral is normally solved by numerical integration:

\[
\overline{K}^{(e)}_{nn} = \sum_{i=1}^{n_{\text{int}}} \sum_{j=1}^{n_{\text{int}}} W_{ij} \frac{\partial N_1}{\partial \xi} \frac{\partial N_1}{\partial \eta} \ldots \frac{\partial N_n}{\partial \xi} \frac{\partial N_n}{\partial \eta} \left| J \right| d\xi d\eta
\]

This equation amounts to a simple summation of the matrix terms which arise at each integration point. However, at each integration point the interpolation function derivatives and \( J, J^{-1} \) and \( \det J \) must be newly evaluated. \( W_{ij} \) represents weighting function values for each of the \( n \) integration points inside each element. They ascertain that the \( n \) contributions add up to the area of the reference element or another target quantity. The location of integration points inside the reference element is chosen according to theoretical considerations, normally Gaussian quadrature rules. In summary, interpolation-function partial derivatives at each integration point and with respect to the local coordinate system, are multiplied with the inverse of the Jacobian matrix at this point, and then summed to perform the numerical integration. All this happens behind the scenes, when higher-order elements are used in a CSP model.

Each of the integrals from above corresponds to a PDE operator. To compose the described partial differential equation using an Algorithm object, references to the math operators are submitted to the Algorithm instance before it is handed over to the SuperGroup object (see also example13.cpp source):

```cpp
// 2. Steady-state fluid pressure computation \( [K] \{p\} = \{Q\} \)
// -------------------------------------------------------
Algorithm steady_state_pressure;
steady_state_pressure.IncreaseMultiGridVectorStorage( 5 );
```
Integral_dNT_op_dN_dV conductance( p_ref, "conductivity", "fluid pressure", "fluid pressure" );
Integral_NT_op_N_dV source( p_ref, "fluid volume source", "fluid pressure" );

VelocityAndVolumeFlux postpro( example_model, "conductivity", "porosity", "fluid pressure", true );
steady_state_pressure.Add( &conductance );
steady_state_pressure.Add( &source );
steady_state_pressure.AddPostProcess( &postpro );

example_model.Pass( steady_state_pressure );

Note that this code and the math operator objects are now the same for two and three dimensional models and a variety of finite-element types. The MathOperatorLHS subclass Integral_dNT_op_dN_dV represents the exact volume integral over the derivatives of the element interpolation functions modeling the fluid-pressure distribution described by the PDE based on Darcy’s Law: The MathOperatorRHS subclass Integral_NT_op_N_dV represents a fluid source term per specific volume such that it needs to be integrated over each element. It enters the right-hand side of the matrix equation.

Fig. 16: Collaboration diagram describing the accumulation process of additive and product terms in the partial differential solved by the Algorithm.

When an Algorithm is passed to a SuperGroup or Group object it hands the control of the interactions with each Element it loops over, to each one of the math-operator subclasses (Fig. 16, see also Fig. 11). These request the information which they need from each Element object. This information may either be provided directly by each element such as matrices (Mat) or vectors (Vec, Ivec) with physical-variable integrals, node coordinates and node numbers. Alternatively, if the information is
specific to the type of finite-element which the element represents, the corresponding subclass of the FiniteElement is accessed by the Element via a reference. This capability is instrumentalized through a Bridge design pattern which is further described in the CSP3D Developer’s Guide.

**TRANSIENT FLUID PRESSURE**

The transient fluid pressure distribution in a fluid-saturated domain is described by:

\[
\frac{\partial}{\partial t} \left( K \frac{\partial p}{\partial x} \right) + \frac{\partial}{\partial y} \left( K \frac{\partial p}{\partial y} \right) + \frac{\partial}{\partial z} \left( K \frac{\partial p}{\partial z} \right) - \frac{\partial Q}{\partial t} = S \frac{\partial p}{\partial t}
\]

This equation is solved iteratively breaking the total time into steps. Time-stepping is usually implemented by a fully-implicit (Backward Euler) finite-difference scheme:

\[
([S] + \Delta t[K])\{p\}_{t+1} = [S]\{p\}_t + \Delta t\{Q\}_{t+1}
\]

As already discussed above, \( S \) and \( K \) are the storage capacity and conductance matrices, and \( p \) and \( Q \) are the fluid pressure (dependent variable) and fluid production/consumption vectors, respectively. The fluid-pressure distribution at the next timestep \( t+1 \) is computed from the present fluid-pressure distribution at \( t \), potential pressure or flux changes at Dirichlet or Neumann nodes, respectively, and by taking into account fluid sink/source rates.

For a 2D finite-element mesh, the storage capacity matrix has the form:

\[
S^{(e)}_{non} = \int_{x,y} \left[ \begin{array}{c} N_1 \\ \vdots \\ N_n \end{array} \right] S^{(e)} \left[ \begin{array}{c} N_1 \\ \vdots \\ N_n \end{array} \right] dA
\]

If this integral is solved by **numerical integration**, the storage capacity matrix is computed as:

\[
S^{(e)}_{non} = \sum_{i=1}^{n_e} \sum_{j=1}^{n_f} W_i(\varepsilon_i, \eta_j) W_j(\varepsilon_j, \eta_j) \left[ \begin{array}{c} N_i(\varepsilon_i, \eta_j) \\ \vdots \\ N_n(\varepsilon_i, \eta_j) \end{array} \right] S^{(e)} \left[ \begin{array}{c} N_i(\varepsilon_i, \eta_j) \\ \vdots \\ N_n(\varepsilon_i, \eta_j) \end{array} \right] d\varepsilon_i d\eta_j
\]
**BUOYANCY-DRIVEN FLOW**

The buoyancy force $F_b$ which acts on a volume of fluid, $V$ with a density $\rho$, diverging from a reference density $\rho_0$, is

$$F_b = V(\rho - \rho_0)g$$

$$\Delta p^{(e)} = A\int_{z_i}^{z_f} \frac{\partial \rho}{\partial z} dz$$

where $g$ is the acceleration of gravity ($9.81 \text{ m s}^{-2}$). The corresponding vertical “fluid pressure” contribution of the element is equivalent to the integral over the changing fluid density. The buoyancy-related volumetric flow in the element, $q$, due to this gradient is calculated from Darcy’s law:

$$q^{(e)} = \frac{(\rho / \rho_0)k\Delta P}{\mu}$$

Since the hydraulic conductivity refers to a volume flux not a mass flux it must be reduced because a greater flow volume must pass through the rock if fluid density is less than the reference density.

The effect of buoyancy forces on the flow, requires the accumulation of an \texttt{OPERAND N GRADIENT N} (for \texttt{CSP Algorithm}) or an \texttt{Integral N op dN dV} operator to represent the vertical pressure gradient term in the equation. This term is non-zero only where fluid density deviates from the reference density and is accumulated into the right-hand side of the matrix equation.

$$G^{(e)}_{lumped} = \int A \begin{bmatrix} \frac{\partial N_1}{\partial z} \\ \vdots \\ \frac{\partial N_n}{\partial z} \end{bmatrix} K_y (\rho - \rho_0) g [N_1 \cdots N_n] dA$$

$$= \frac{Kg}{3} \begin{bmatrix} \rho_1 - \rho_0 \\ \vdots \\ \rho_n - \rho_0 \end{bmatrix} A$$

**LUMPING**

The diagonalization of element matrices is called “lumping”. In this process, the “off-diagonal” elements of a matrix are set to zero by multiplying these terms into the diagonal. This is equivalent to defining the interpolation functions differently such that
\[ N_i N_j = \begin{cases} \frac{1}{n} & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \]

where \( n \) is the number of nodes in the element. In a ‘consistent’ matrix for a two-dimensional triangular element (left matrix below), the area factor \( A \) is divided by 12. This is required to obtain an exact integral over the interpolation functions according to the integration formula presented earlier.

\[
\begin{bmatrix}
2 & 1 & 1 \\
1 & 2 & 1 \\
1 & 1 & 2 \\
\end{bmatrix}
\rightarrow
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\]

In the “lumped” matrix on the right, the same is achieved by multiplying the identity matrix with a third of the area. This denominator is equivalent to the number of nodes = interpolation functions used. The determinant remains the same (4A/12 vs. A/3).

Lumping is used most commonly in transient calculations where different, simpler, interpolation functions are desirable for time derivatives in finite-difference time stepping schemes. Relative to the consistent formulation, lumping can introduce errors since off-diagonal matrix terms are effectively ignored. Lumping must therefore be used with caution, especially for linear elasticity problems where off-diagonal terms describe shear stresses etc.

**REFERENCES CITED**


Liu, Y. and Vinokur, M, 1997, Exact Integrations of Polynomials and Symmetric Quadrature Formulas over Arbitrary Polyhedral Grids, NASA Technical Memorandum 112202


7. The CSP Libraries

The CSP libraries are written in layers which are increasingly high-level and specific in terms of the functionality which they provide. Thus, while the standard CSP3D3.0 library contains the base classes for most objects, as are listed in the CSP Distribution Section of Appendix A, additional libraries contain subclasses derived from base classes in the main library, such as the CSP_Algorithm, CSP_FEM_Visitor, Interrelation, MathOperatorLHS, MathOperatorRHS, FiniteElement, etc. classes. CSP3D3.0 ships with the following libraries:

- CSP3D3.0 Library
- CSP3D3.0 Interrelations
- CSP3D3.0 Finite Elements
- CSP3D3.0 Finite Volumes
- CSP3D3.0 Algorithms
- CSP3D3.0 Visitors

These libraries are continuously supplemented by additional objects from CSP developers. These publicly generated extensions are made available to other CSP developers and users such that they can tap this growing reservoir to build increasingly realistic simulations.

All CSP libraries depend on the basic CSP3D3.0 Library. Clearly, as the CSP examples code is meant to demonstrate the capabilities of CSP, it must be linked with all of the CSP libraries in order to function. The examples code combines a choice of functions in a main program. These functions perform a wide variety of computations, including the examples referred to earlier in this User’s Guide.

INTERRELATIONS “Seed” Library

Local interdependencies among physical variables in a model are expressed by subclasses inherited from the base class Interrelation. Their declarations are included in the main program file in standard C/C++ manner

#include “MyInterrelation.h”

Interrelation subclasses are executed when passed to the SuperGroup by calling its Pass() method:

my_supergroup.Pass( inherited_interrelation );

Interrelation “Seed” library means that only a few interrelation subclasses are supplied with the CSP3D3.0 distribution to illustrate what interrelations can do and how to write them. These interrelations are briefly described below in an alphabetical order.
All interrelations expect specific names and placements of physical variables (in italics) which they interrelate. These properties must be defined in the variable database input file. The provided interrelations either completely replace or add to or subtract from the value of the target variable. This important behavioral difference is indicated in brackets after the subclass name.

**Conductivity.** (replace) Calculates the hydraulic conductivity “conductivity” [m s\(^{-1}\)] of a permeable medium from its “permeability” [m\(^2\)] and the physical variable “fluid viscosity” [Pa s\(^{-1}\)]. These variables are expected to be element variables. If the fluid viscosity is a node variable, the average value over the nodes of the element will be used in the calculation.

**ConductivityFromPerm.** (replace) Simplified version of previous interrelation. Calculates the hydraulic conductivity “conductivity” [m s\(^{-1}\)] of a permeable medium from its “permeability” [m\(^2\)] using a fixed value for the viscosity of the fluid [Pa s\(^{-1}\)].

**Dilatancy.** (replace) This 2D interrelation calculates the physical element variable “dilatancy” [m\(^2\) m\(^{-2}\)] not from the variable “strain”, but as the difference of the element “area” before and after displacements have been applied (“new area” [m\(^2\)]). Dilatancy can be applied to any 2D element type to obtain an independent measure how well a strain calculation performed (as is discussed in the description of the finite element library, strains tend to be most accurate at the integration points inside of elements and often erratic if they are calculated directly at the nodes.

**ExtractTensorVariableComponent.** (replace) Permits to extract specific components of tensor variables as identified by their index pair \((i,j)\), and place them into a scalar variable of choice. Thus, diagonal and off-diagonal components can be extracted if needed.

**ExtractVectorVariableLength.** (replace) Computes the length of vector variables assuming that the vectors originate in the coordinate origin. The computed length can be assigned to a scalar variable of choice.

**ExtractVectorVariableComponent.** (replace) As for tensor variables (see above), vector variable components identified by an index integer, can be mapped to user-specified scalar variables.

**FluidExpansionSource.** (add/subtract) Calculates a “fluid volume source” [m\(^3\) m\(^{-3}\)] source or sink term which arises due to the expansion or contraction of the pore fluid in the rock from one to the next timestep. The fluid-volume change is corrected for the thermal expansion or contraction of the country rock. If this value is not set by the user, a value for concrete at 20°C is applied (12.5e-6 K\(^{-1}\)). A volume change usually accompanies a change in fluid pressure or temperature and is monitored by taking the difference between the element variables “previous fluid density” and “fluid density” [kg m\(^{-3}\)] multiplied by the “porosity” [X]. The temperature change is monitored by the variables “previous temperature” and “temperature” [°C] and is used in conjunction with a user-specified thermal expansivity of the rock [m\(^3\) m\(^{-3}\) K\(^{-1}\)], to calculate the thermal expansion of the pore space in the rock (“porosity”, [X]). Since a source term in a transient calculation is always expressed as a rate [m\(^3\) m\(^{-3}\) s\(^{-1}\)], the calculated “fluid volume source” is divided by the time increment over which the change occurred.
FluidHeatCapacity_HILL. (replace) Calculates the physical variable “heat capacity fluid” [J kg⁻¹ K⁻¹] from the nodal variables “fluid density” [kg m⁻³] and “temperature” [°C]. This calculation is carried out using the Hill (1986) equation of state for pure water which covers a temperature range from 0 to 1000°C and a pressure from atmospheric to 1 GPa. The heat capacity calculation is encapsulated into the class HILL_H2O_Density, which, in turn, draws upon the C functions defined in Hill.c.

FluidViscosity_HILL. (replace) Calculates the physical variable “fluid viscosity” [Pa s⁻¹] from the nodal variables “fluid density” [kg m⁻³] and “temperature” [°C]. This calculation is carried out using the Hill (1986) equation of state for pure water which covers a temperature range from 0 to 1000°C and a pressure from atmospheric to 1 GPa. The viscosity calculation is encapsulated into the class IAPS_H2O_Viscosity, which, in turn, draws upon the C functions defined in Hill.c.

HeatTransfer. (replace) The variable “heat transfer velocity” [m s⁻¹], vh, is calculated from the Darcy flow velocity “velocity”, v, [m s⁻¹] the heat capacities of the fluid and the country rock, hf, hc (“heat capacity fluid”, “heat capacity”) [J kg⁻¹ K⁻¹], and their densities, rf, and r (“fluid density” and “density”) [kg m⁻³], respectively, according to the formula:

\[ vh = \frac{(v \cdot hf \cdot rf)}{(hr \cdot r)} \]

HydraulicHead. (replace) Calculates the nodal variable “hydraulic head” from the “fluid pressure” and the “elevation”, which refers to the height above the lowest topographical point in a model. The head, h, is calculated according to the formula

\[ h = \frac{pf}{(rho0 \cdot g)} + \text{elevation} \]

\( g \) is the acceleration of gravity (9.80665 m N) and \( rho0 \) is the fluid reference density which is assumed to be 1000 kg m⁻³.

ModifyPropertyWhere. (replace) For a user-specified physical variable, this generic interrelation searches for specific scalar values and replaces them with a fixed scalar value.

NodeToElementProperty. (replace) Takes the average value of a user-specified nodal physical variable of all the nodes of an element. The result is returned into a user-specified element variable.

QuartzSolubility. (replace) Calculates the solubility of quartz in pure water “quartz solubility” [kg m⁻³] in an element from the nodal variables “temperature” [°C] and “nodal fluid density” [kg m⁻³]. The calculation is based on Craig Manning's fit of quartz-solubility data. The result is returned in grams per kg H₂O.

QuartzSolubilityNaCl. (replace) As above, but calculation of aqueous solubility according to the empirical formula of Fournier & Potter, GCA 46, p. 1969-73, with a correction for the presence sodium chloride according to Fournier, 1983, GCA 47, p. 579-586. The fluid salinity is specified by the nodal variable “mass fraction NaCl”.

QuartzPrecipitation. (add/subtract) Integrates over a user-specified time-increment, how much quartz is precipitated or dissolved per unit volume of each finite element through which fluid flows at a Darcy velocity “velocity” [m s⁻¹]. This is achieved by projecting the “quartz solubility gradient” vector onto the flow
velocity vector. In this calculation, the physical variable "fluid density" [kg m\(^3\)] is used as element average density from which the time-integrated fluid mass flux [kg m\(^2\) s\(^{-1}\)] is derived. The result is added to or subtracted from the variable “quartz” [kg m\(^3\)].

RelativeFluidDensity_HILL. (replace) Calculates a “relative fluid density” (\(\rho - \rho_0\)) which represents the difference between the actual “fluid density”, \(\rho\), and a reference fluid density, \(\rho_0\), of 997.044829 kg m\(^3\) at 25°C and atmospheric pressure (101325 Pa). The relative fluid density which is increasingly negative with increasing temperature, is used for the calculation of buoyancy forces on the flow in convection simulations (see also the GravityCorrectedVelocity algorithm in the Algorithms section below).

ScalarVolumeFlux. (replace) Calculates the fluid volume flux in the direction of flow as the length of the flow-velocity vector and returns the result into the physical variable “scalar volume flux” [m\(^3\) m\(^2\) s\(^{-1}\)]. This calculation uses the “velocity” [m s\(^{-1}\)] as input variable and must be informed, by setting a boolean variable when constructing the interrelation, whether the velocity refers to the Darcy velocity or to the interstitial flow velocity. In the second case the velocity is multiplied by the scalar element variable “porosity” [X] before taking its length as the volume flux.

ScalarMassFlux. (replace) Calculates the length of the Darcy flow velocity vector [m s\(^{-1}\)] and multiplies it by the element variable “fluid density” [kg m\(^3\)] in order to compute the physical variable “scalar mass flux” [kg m\(^2\) s\(^{-1}\)] in each finite element. If the flow model is three-dimensional, the scalar mass flux is calculated directly from the element variables “fluid pressure gradient” [Pa m\(^{-1}\)], “conductivity” [m s\(^{-1}\)], and “fluid density” [kg m\(^3\)]. In this case, the variable fluid pressure gradient is computed conveniently, from the variable fluid pressure [Pa] via the SuperGroup method “CopyGradientOfProperty_A_To_B()”.

TransportStepSize. (no variable is changed) Determines the maximum permitted advection-time-increment size for property advection on FiniteDifferenceGrid objects by the TransportVisitor or HeatTransportVisitor objects. This time increment is calculated on the basis of two vector variables that define nodal and elemental transport velocities, respectively, and a third variable which stores the radius of the inner circle of each triangular element. The stepsize is chosen such that the product of the velocity and the time increment is smaller than the diameter of the inner circle for all triangles. The physical variables are submitted to TransportStepSize as constructor arguments, and are named, in most cases “nodal velocity” [m s\(^{-1}\)], “velocity” [m s\(^{-1}\)], and “inner radius” [m]. When the transport-step-size interrelation is passed to the SuperGroup, it calculates the time-increment which can subsequently be retrieved using the method AdvectionTimeIncrement().
CSP Algorithms Library

The CSP3D.0 Algorithms library contains several subclasses of the “old-style” CSP_Algorithm for two-dimensional computations using linear triangular finite elements. Since CSP_Algorithm’s are intended for modeling “beginners”, background information is given on modeling fluid flow and the importance and physical meaning of the variables used.

Velocity & Interstitial Velocity

Darcy's law describes the proportionality between steady-state volumetric flow, \( q \), through a cross section of homogeneous isotropic rock and the applied fluid-pressure gradient (grad \( p \)). Homogeneous isotropic means that the permeability, \( k \), of the rock is uniform and non-directional.

\[
q = \frac{k \nabla p}{\mu}
\]

The nabla operator (triangular symbol) refers to the gradient of fluid pressure in the direction of flow. Darcy's law only applies to laminar flow in which inertia does not play a role. The validity of Darcy's law is discussed by Gustensen and Rothman (JGR 98:B4, 6431-6441, 1987) who bracket the conditions under which it holds.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Property</th>
<th>SI Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k )</td>
<td>permeability</td>
<td>( [m^2] )</td>
</tr>
<tr>
<td>( K )</td>
<td>hydraulic conductivity</td>
<td>( [m s^{-1}] )</td>
</tr>
<tr>
<td>( p )</td>
<td>fluid pressure</td>
<td>( [Pa] )</td>
</tr>
<tr>
<td>( \mu )</td>
<td>dynamic viscosity of the pore fluid</td>
<td>( [Pa s^{-1}] )</td>
</tr>
<tr>
<td>( q )</td>
<td>specific discharge =Darcy velocity</td>
<td>( [m s^{-1}] )</td>
</tr>
<tr>
<td>( S )</td>
<td>storativity</td>
<td>( [m^3 m^3 Pa^{-1}] )</td>
</tr>
<tr>
<td>( \phi )</td>
<td>porosity</td>
<td>( [X] )</td>
</tr>
<tr>
<td>( v )</td>
<td>Interstitial velocity</td>
<td>( [m s^{-1}] )</td>
</tr>
<tr>
<td>( \rho )</td>
<td>fluid density</td>
<td>( [kg m^3] )</td>
</tr>
</tbody>
</table>

Table 2: Notation used in this description.

Specific discharge. Darcy's law gives the volumetric discharge, \( q \), from a rock sample subjected to a unit fluid-pressure gradient. If the sample is a little cylinder of rock, the specific discharge refers to the velocity of the piston in a syringe with the same cross-sectional area as the sample. In such an experiment the fluid-pressure gradient in the sample must be held constant and the fluid must be essentially incompressible. While \( q \) is a vector property it describes a volume flux across a specific cross-sectional area. For a density of the fluid of 1000 kg m\(^{-3} \), \( q \) is equivalent to a mass flux of 1000 kg s\(^{-1} \) (\( q \times 1000 \)).

\[ \text{mass flux [kg s}^{-1} \text{]} = \text{volume flux [m}^3 \text{s}^{-1} \text{]} \times \text{fluid density [kg m}^{-3} \text{]} \]

Otherwise, a density correction must be applied to calculate the mass flux from \( q \). The specific discharge \( q \), is called the Darcy velocity or transport velocity. From the
discussion of the syringe it is clear that this velocity is not equivalent to the flow
velocity in the pore or fracture space. This interstitial flow velocity, \( v \), is calculated from

\[
v = q \Phi
\]

Since \( F \) must represent an average over all the pores in the volume of interest, \( v \) is a
meaningful measure, only if the rock is homogeneous. In a porous sandstone with
fractures which are not explicitly represented in the model, the velocity spectrum in
the rock may have two maxima. This dichotomy would be poorly represented by a
mean velocity. When including flow velocities in calculations one uses the Darcy
velocity if one wants to calculate fluid-volume dependent processes such as advection
of heat, for example. If one is interested in the spreading of pressure perturbations or
the transport of chemical species in transient flows, one must use the interstitial
velocity to model, for instance, the breakthrough of a tracer in a water well.

**Permeability**

The permeability is calculated from the flow that occurs through a rock sample when
it is subjected to a fixed pressure gradient. This assumes an equilibrium state of fluid
pressure in the sample i.e. a linear decrease in fluid pressure along the flow path and a
volumetrically unimportant expansion of the fluid as it flows down this pressure
gradient. To ascertain that the pressure gradient in the sample has equilibrated, the
permeability is normally calculated from the flux after a multiple of the fluid volume
in the pore space of the sample has passed through it.

In contrast with fluid pressure, permeability is an average volumetric property and a
permeability value is meaningful, only, if it is representative of at least 30 or more
pores. If a single \( k \) value applies to a rock it can be referred to as homogeneous. The
rock is homogeneous and isotropic, if its permeability does not vary with the
direction of the flow. Otherwise the rock is anisotropic. An assembly of rocks, each
of which is homogeneous and isotropic, should be referred to as inhomogeneous rock
mass. A heterogeneous rock mass is then distinguished by both spatial and directional
variations in its permeability.

In thinsection, microfractures and other inhomogeneities can be seen in the pore
space of many rocks. This means that even on the cm-scale the rock is not
homogeneous and isotropic and a measured permeability just represents a somewhat
coincidental average over its composite flow properties. In this case, the permeability
should be referred to as effective permeability, meaning that the measured value is
used to represent the volume in spite of the fact that the rock is not homogeneous.
Practically, the representativeness of such an effective measurement depends greatly
on the type of calculation that it is employed in. Generally, without additional
provisions made, effective permeabilities should only be used for modeling steady
state flow.
Hydraulic conductivity

The hydraulic conductivity, $K$, has the same dimensions and is essentially equivalent to the specific discharge $q$. $K$ is a direct and widely used measure which is specific to the type of fluid which is considered, as it depends on its viscosity. The hydraulic conductivity is important in surface hydrology and, because it contains fluid viscosity, it can also be used to describe the flow properties of a fracture. Since a fracture is an open flow channel its permeability would be infinite.

In a CSP calculation a **steady-state pressure distribution** and a resulting two-dimensional Darcy flow-velocity field is computed by the following piece of code:

```cpp
Velocity steady_state_pressure( p_ref );
steady_state_pressure.AddOperation( OPERAND__GRADIENT_N__GRADIENT_N,
                                 "permeability", "fluid pressure","fluid pressure");
steady_state_pressure.AddOperation( OPERAND__N, "fluid volume source",
                                 "fluid pressure");
model.Pass( steady_state_pressure );
```

To compute the interstitial flow velocity, one replaces the `Velocity` algorithm with the `InterstitialVelocity` algorithm.

Storage Capacity

The fluid which resides in the pore and fracture space of a rock expands and contracts in response to changes in fluid pressure. This implies that a small amount of fluid is added or withdrawn from the flow along the fluid-flow path. This contribution is parameterized by the storage capacity, $S$, of the rock per unit volume. $S$ is defined as the additional volume of fluid that will be contained in a unit volume of rock after a unit increase in fluid pressure. $S$ therefore has the dimensions $\text{m}^3 \text{m}^{-3} \text{Pa}^{-1}$ and its calculation involves the compressibilities of the fluid, $\beta_f$, and that of the interconnected pore space, $\beta_b$ (Fischer & Paterson 1992).

$$S = \Phi(\beta_f - \beta_m) + (\beta_b - \beta_m)$$

$\beta_m$ is the mean compressibility of the rock-forming minerals. Only the interconnected pore space is relevant for fluid flow. All compressibilities are a function of pressure and temperature. For example, $S$, of Solenhofen limestone at 400°C and 300 MPa confining pressure, is about $10^9 \text{ m}^3 \text{ Pa}^{-1}$ (Table 2). In the petroleum-engineering literature, an alternative approach is used to calculate the storage behavior of a reservoir rock. Firstly, for single-phase fluid flow, the rock compressibility (equivalent to the bulk modulus, $M$, [Pa$^{-1}$]), $c_r$, and the fluid compressibility, $c_f$, are measured.

$$c = \frac{\partial V}{\partial p}$$
Then the total-system compressibility, $c_t$, is calculated as the sum of these compressibilities

$$c_t = (1 - \Phi) c_r + \Phi c_f$$

In contrast to the definition of storativity (which includes the porosity), $c_t$ includes the compressibility of the matrix. Due to the change of the compressibility of geologic fluids with pressure and temperature, the storativity of a rock is also a function of the absolute temperature and fluid pressure. $S$ increases near the critical region of water and then decays with increasing fluid pressure, reflecting a decline in the compressibility of water. Argon is commonly used as a pore fluid in measurements of $S$ in experiments. Experimentally determined values of $S$ must therefore be re-calculated for an aqueous pore fluid via an equation of state for water. Some experimental data are given in Table 2. Fischer and Paterson (1992) constrained that, at fixed $pT$-conditions, $S$ varies about 1-2 orders of magnitude among different rock types, largely reflecting differences in pore geometries. Their experiments do not show a simple relationship between the porosity $F$ and $S$ as one would assume from the storativity equation above. A possible explanation of this observation is that the formation of microcracks increases $\beta_m$ while the porosity remains close to constant.

Storativities for pure water should represent minimum values for metamorphic fluids since these usually contain appreciable mole-fractions of CO$_2$ or CH$_4$. At the moderate pressures that typify many hydrothermal systems (<200 MPa) the compressibilities of these gases are significantly larger than that of water.

Since the porosity of a rock may increase during deformation, so can the storage capacity, as demonstrated by Fischer and Paterson (1992) and Zhang et al. (1995a, b). At a low effective stress, deformation produces interconnected high-aspect-ratio cracks, increasing the storativity of the deformed rock. The fluid in networks of interconnected fractures may be released spontaneously during a pressure decrease if these fractures are held open by fluid pressure. The significance of this contribution to $S$ depends on the general rock properties. Calculations show that in a porous, compressible sandstone, at 5 km subsurface depth, open fractures with a diameter greater than one hundred meters are required to change the storativity appreciably. On the other hand, the experiments of Fischer and Paterson (1992) show that even microcracks can change the storativity of a low-porosity rock at metamorphic pressure-temperature conditions.

<table>
<thead>
<tr>
<th>Rock Type</th>
<th>log $S$ Argon</th>
<th>log $S$ Water</th>
</tr>
</thead>
<tbody>
<tr>
<td>(200°C, 300 MPa confining- and 150 MPa fluid-pressure)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Solenhofen limestone</td>
<td>-9.33</td>
<td></td>
</tr>
<tr>
<td>Carrara marble</td>
<td>-9.93</td>
<td></td>
</tr>
<tr>
<td>Gosford sandstone (600°C)</td>
<td>-9.30</td>
<td></td>
</tr>
<tr>
<td>(400°C, 300 MPa confining- and 100, 150, 200 MPa fluid-pressure)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Solenhofen limestone</td>
<td>-9.32</td>
<td>-9.90</td>
</tr>
<tr>
<td></td>
<td>-9.31</td>
<td>-9.78</td>
</tr>
<tr>
<td></td>
<td>-9.53</td>
<td>-9.95</td>
</tr>
</tbody>
</table>

Table 3: Storage capacity of different rock types at metamorphic PT-conditions (after Fischer and Paterson, 1992).
The hydraulic diffusivity, \( \kappa \), [m\(^2\) s\(^{-1}\)] is a convenient measure to calculate over what distance, as a function of time, a pressure perturbation travels through in the rock. Due to the difference in the definitions of storativity and total-system compressibility two alternate ways of calculating the hydraulic diffusivity result:

\[
\kappa_j = \frac{k}{S\mu} \quad \text{versus} \quad \kappa_j = \frac{k}{\Phi\mu c_i}
\]

Pressure diffusion is very slow if the hydraulic conductivity is as low as that of shales or metamorphic rocks (e.g., Neuzil, 1994, Brace 1980). The diffusion of fluid pressure in a rock is analogous to the diffusion of heat in a solid which obeys the basic differential equation for heat conduction:

\[
\frac{\partial T}{\partial t} = \kappa_T \nabla^2 T
\]

where \( \kappa_T \) is the thermal diffusivity. The thermal diffusivity is given by

\[
\kappa_T = \frac{\vartheta}{\rho C}
\]

\( \vartheta, \rho, C \) are the thermal conductivity, the density of the material and the specific heat (joules), defined as the amount of heat that must be supplied to raise the temperature of a unit mass by one Kelvin.

A fluctuation in fluid pressure might also be propagated by fluid flow. Assuming that the porosity of the rock is constant, a pressure change due to fluid transport adds to the transient fluid-pressure equation.

\[
S \frac{\partial p}{\partial t} = \frac{k}{\mu} \nabla^2 p - \frac{1}{S} \mu \cdot \nabla p + Q
\]

where \( u \) is the fluid-volume flux through the system and \( Q \) is a fluid source term. Phillips (1991) shows that this advection term can be dropped in most cases for geologic environments. In analogy to analytical solutions to the heat flow equation in a homogeneous isotropic medium (e.g., Courant and Hilbert), a pressure disturbance should diffuse over a distance, \( d \), given by

\[
d = \sqrt{\frac{2\kappa_T}{\tau}}
\]

\[
\tau = \frac{d^2}{2\kappa_p}
\]

in terms of the time, \( t \), which it takes for a pressure disturbance to diffuse over \( d \) in a rock with a uniform hydraulic diffusivity. \( t \) can be considered as the characteristic response time of the rock at the conditions of interest. According to this formula it should take about 50,000 years before a pressure disturbance applied at one side is propagated to the other side of a 100 m-column of close to impermeable Solenhofen limestone \((10^{17.5} \text{ m s}^{-1}, \phi = 0.028, S (400^\circ \text{C}, 150 \text{ MPa}) = 10^9 \text{ m}^3 \text{ Pa}^{-1})\). This is very slow and highlights that the consideration of the response time is crucial for the
description of the dynamic behavior of a hydrothermal system at depth. In a system where long fractures account for the bulk permeability the pressure disturbance may be able to propagate significantly faster.

Fluid Pressure Perturbations, Poroelasticity

As a pressure perturbation propagates into a rock pile only once an appreciable pressure gradient is established there will be a significant flow response. Before this is the case, pore-fluid (and matrix) will respond by dampening the perturbation via contraction or expansion. In this "delay period", the flow system is governed by poroelastic behavior. This is another name for a behavior that is controlled by the storativity. Later, when a steady state is approached, predicting system behavior is no longer dependent on predicting its storage capacity.

In summary, there is a decisive difference between the pressure response and the flow response of a fluid-rock system. This difference can be visualized by a liver as opposed to a bladder analogy for rocks. Porous rocks behave more like a liver from which fluid oozes out, if cut with a knife and squeezed.

A transient fluid pressure distribution is evolved in the following code example (see also example3.cpp):

```
Velocity velo(ref_to_PropertyDatabase);

while ( global_time < max_time )
{
  ...
  velo.AddOperation( OPERAND__GRADIENT_N__GRADIENT_N,
                     "conductivity","fluid pressure","fluid pressure");
  velo.AddOperation( LUMPED__OPERAND__N__N,
                     "storativity","fluid pressure","fluid pressure");
  velo.AddOperation( LUMPED__OPERAND__N,
                     "storativity","fluid pressure");
  velo.AddOperation( LUMPED__OPERAND__N2,
                     "fluid volume source","fluid pressure");
  velo.SetTimeIncrement( time_increment );
  supergroup.Pass( init_pressure2 );
  velo.Reset();
  ...
  global_time += time_increment;
  timestep++;
}
```

At each timestep fluid pressure diffuses to a new state which is determined by the boundary conditions and the fluid source/sink terms in the flow model.

Regimes of Fluid Flow

Experiments as well as solutions to the fluid-pressure equation, show that spatial variations in the permeability of a flow domain will be reflected in spatial variations in the transport velocity. In the following descriptions of flow patterns, upstream will refer to a region with a higher fluid pressure and downstream to a region with a lower fluid pressure than the location in question. Clearly, this applies only to pressure-
driven fluid flow. For a discussion of the influence of spatial permeability variations, it is convenient to consider only the deviations from a uniform flow pattern due to externally imposed fluid-pressure boundary conditions. This treatment introduces the useful concept of a far-field (=unperturbed) fluid pressure gradient.

**Convergent Flow / Fluid-Flow Focusing.** Upstream from narrow, relatively more permeable zones parallel to the far-field fluid-pressure gradient, fluid motion is directed toward the more permeable zones. This implies an increase of the fluid flux as one approaches the more permeable zone. This flow regime is called convergent and is reflected in a convergence of streamlines.

**Divergent Flow.** At the downstream termination of a more permeable flow zone the fluid has nowhere to go except for into the less permeable country rock. This implies that the flow has to spread out over a wider area in order to accommodate the fluid volume which is currently traveling in the more permeable zone. This spreading process is also referred to as hydro-dynamic dispersion. In the absence of local fluid sources, a divergence of the flow implies a decrease in the fluid flux in the flow direction.

**Fluid Sources and Sinks.** If fluid is generated or consumed along the fluid-flow path this will affect the flow at a given location. This can be accounted for by a fluid source term $Q$ on the right-hand side of the diffusion advection equation discussed earlier, introducing the convention that a fluid source shall be positive. Fluid sources in hydrothermal systems can arise where metamorphic devolatilization occurs or where a water-saturated melt crystallizes. Fluid sinks occur where water is discharged from springs at the earth's surface or where anhydrous rocks are hydrated during hydrothermal alteration. By default, the fluid flow in a fluid source region is divergent and the flow in a fluid sink region must be convergent. A fluid source implies

$$\frac{\partial p}{\partial x^2} + \frac{\partial p}{\partial y^2} = \nabla^2 p > 0$$

and a fluid sink is expressed by the second equation.

**Topography-Driven Fluid Flow** (see also `example7.cpp`). In groundwater hydrology, gradients of the *hydraulic head* instead of fluid pressure, are used to define flow lines, following the rationale that near-surface flow tends to be driven purely by topography and fluid density is constant. This is a special case of pressure-driven fluid flow. Topography influences fluid pressure in the rock, because gravity acts as a body force on the fluid. The vertical variation in fluid pressure in the earth's crust subjected to gravity is

$$\frac{\partial p}{\partial z} = \rho g z$$

At the base of a fluid-saturated mountain, we have a fluid pressure that corresponds to the weight of a water column with the height of the mountain. At the same time, the
fluid pressure in a creek in the adjacent valley at the same elevation is equal to the atmospheric pressure. We may find springs out of which relatively overpressured water from the inside of the mountain flows into the valley. This kind of flow is described by gradients in the hydraulic head, \( h \), the sum of the pressure head (piezometric head = atmospheric pressure + weight of watercolumn) and the elevation head:

\[
 h = \frac{P}{\rho g} + z, \quad p = \rho g z + p_{\text{atm}}
\]

Where \( z = 0 \) at the base of the system. To appreciate this formula, note that water in a swimming pool does not flow upward in spite of the fact that the fluid pressure in the pool increases with depth. To inject water through a jet at the bottom of the pool, we have to overcome the gravity-induced fluid-pressure at the location of the jet. This pressure is called hydrostatic pressure or piezometric head.

In the earth's crust, upward fluid flow occurs only where the vertical fluid-pressure gradient is above hydrostatic, but the hydrostatic pressure gradient depends on fluid temperature as a function of depth as is shown in the cartoon (Fig. 18) which depicts that fluid viscosity is also a function of temperature and pressure.

In CSP, one can use the `H2OPropertiesVisitor` or various Interrelations to compute the properties of pure H\(_2\)O as a function of \( P, T \) (see `example8.cpp`).

**Buoyancy-Driven Fluid Flow**

In order to incorporate fluid-density variations into a Darcy’s law-based flow equation like that for transient fluid pressure, an additional “buoyancy”-pressure term has to be incorporated such that

\[
 S \frac{\partial p}{\partial t} = \nabla \cdot \left[ \frac{k}{\mu} \left( \nabla p + \rho g \nabla z \right) \right] + Q_{\text{f(ef)}}
\]
Again, \( S \), is the (now \( P,T \)-dependent) specific storage, \( p_r \) denotes “relative fluid pressure”, which shall be defined as the absolute fluid pressure minus the hydrostatic overburden for a reference fluid density at 25°C and atmospheric pressure. \( Q_{fr,c} \) are fluid expansion- or contraction-related volume source and sink terms which arise because fluid is heated or cools along the flow path, respectively. Mass is therefore conserved indirectly rather than solving the conservation law

\[
\frac{\partial \phi p_f}{\partial t} = \nabla \left[ \frac{k_p}{\mu} (\nabla p + \rho_f g \nabla z) \right] + Q_m \tag{2}
\]

explicitly, treating sources as mass- rather than volume sources. The resulting Darcy velocity is calculated by the GravityCorrectedVelocity CSP_Algorithm as a post-processing calculation:

\[
q_x = -\frac{k_z}{\mu} \nabla p_{rx} \quad q_z = -\frac{k_z}{\mu} \left( \frac{\partial p_r}{\partial z} + \rho_f g \nabla z \right)
\]

In order to better satisfy the continuity equation,

\[
\frac{\partial \phi p_f}{\partial t} + \frac{\partial \rho_f q_x}{\partial x} + \frac{\partial \rho_f q_z}{\partial z} = 0 \tag{4}
\]

(linear triangular element-based velocities are not continuous across element boundaries) velocities are extrapolated to the nodes using an extrapolation scheme. The reduced pressure equation (1) is solved alternatingly with a temperature diffusion – advection equation

\[
\frac{\partial T}{\partial t} = \frac{K_T}{C_f \rho_f} \nabla^2 T - \frac{1}{C_f \rho_f} \left( \frac{C_f \rho_f}{C_r \rho_r} q \cdot \nabla T \right) + Q_h
\]

with a heat source and sink term, \( Q_h \), accommodating fluid enthalpy changes during flow. \( C \) denotes the heat capacity and the subscripts, \( f \), and, \( r \), refer to the fluid and the rock, respectively.

To avoid excess numerical diffusion (from which many models in the literature suffer) and to rule out temperature oscillations in regions with very high heat-transport velocity gradients, a first-order characteristics-based heat advection algorithm is used with a constant, but much higher resolution, than given by the finite-element mesh. In code example example8.cpp, this happens inside the HeatTransportVisitor object and is reported as advection steps. The finite-difference time increment is chosen such that the heat advection distance is less than a given value. This makes the method robust even if the flow velocities vary greatly during a simulation.

To compute absolute fluid pressure values needed by the H2OPropertiesVisitor for the re-evaluation of the fluid properties after each transport and thermal equilibration
step, the pressure equation is solved always first with the actual fluid density rather than relative fluid density values. The fluid density at any one point in the model is calculated with the Hill equation of state (EOS) for water (Hill 1986). Only once exact fluid-densities have been determined this way, the reduced fluid pressure is computed and the Darcy velocities are found.

Clearly, this semi-coupled approach is just adequate for the single-phase region where, in practice, fluid enthalpy changes introduce only a weak non-linearity into the temperature equation. However, since it is fast, large spans of geological time can be modeled with timesteps small enough to capture transient flow patterns. This is very important for highly-permeable sections of the crust since convective plumes often only have livespans of up to a few years. Where steam separation occurs, however, an iterative solution scheme must be used like the one suggested by Huyakorn and Pinder (1983, chapter 5).

With regard to crustal regimes of fluid flow, one distinguishes forced convection from Rayleigh-Taylor convection systems. Lateral variations in temperature expressed in sloped isotherms will produce fluid-density variations in the plane perpendicular to the gravity vector. Resulting lateral fluid-pressure variations will drive fluid flow. Where the fluid is hotter, it will rise because it is more buoyant. Cold fluid from the sides will be drawn into those regions. This scenario is referred to as forced convection.

The transient form of the fluid-pressure advection-diffusion equation from above, for the reduced pressure, is assembled using the GravityCorrectedVelocity CSP_Algorithm in the following piece of CSP code:

```
GravityCorrectedVelocity Pr( p_ref );
...
Pr.AddOperation( OPERAND__GRADIENT_N__GRADIENT_N, 
    "conductivity","fluid pressure","fluid pressure");
Pr.AddOperation( LUMPED__OPERAND__N__N, 
    "storativity","fluid pressure","fluid pressure");
Pr.AddOperation( LUMPED__OPERAND__N, 
    "storativity","fluid pressure","fluid pressure");
Pr.AddOperation( OPERAND__N_GRADIENT__N, 
    "relative fluid density", "fluid pressure");
Pr.SetTimeIncrement( time_increment );
model.Pass( Pr );
Pr.Reset();
```

This code is executed in the time-stepping loop in which each pressure solve is followed by a heat advection step and a thermal conduction step.

**LinearElasticity**

The elastic deformation of a discretized rock volume in response to (1) boundary displacements, (2) point forces applied to the nodes, (3) stresses applied at the boundaries, (4) body forces, and (5) volume changes (thermal expansion, contraction, or compaction), can be simulated crudely with the LinearElasticity subclass. Since the LinearElasticity algorithm is derived from the CSP_Algorithm which
depends on the linear Triangle finite-element, derivatives of linear interpolation functions are used to compute element contributions to the global stiffness matrix. Experience with this approach shows that mesh convergence is not always obtained, mesh locking occurs, and residuals are large. The method is only first-order accurate. This means that while the displacements in the model are computed fairly realistically (however only if variations in the material properties are small), the strains and stresses computed in post-processing operations, are noisy and useless. To improve this situation, use the QuadraticTriangle or the BarycentricQuadraticTriangle in conjunction with the Algorithm class.

The LinearElasticity algorithm solves the matrix equation:

\[ [K][u] = [F] \]

where \( K \) is the global stiffness matrix, \( u \) is the nodal displacement vector and \( F \) represents boundary conditions in the form of displacements, stresses, body forces, and volumetric source or sink terms. Element contributions to the stiffness matrix have the form:

\[ [tA][B]^T[D][B][u] = [F] \]

\( t \), is a model thickness multiplier, \( A \) is the area of the element, and \( B \) is the interpolation-function derivative matrix (since a Galerkin method is used, the test functions are also used as weighting functions). \( D \) is the material property matrix for a plane stress computation (stresses act only in the model plane):

\[
[D] = \begin{pmatrix}
E & \mu & 0 \\
\mu & 1 & 0 \\
0 & 0 & 1 - \mu \\
\end{pmatrix}
\]

In \( D \), \( E \) refers to Young’s modulus and \( \mu \) to Poisson’s ratio. After the displacements have been computed, the strain, which is constant in each element is calculated as:

\[ \{ e \} = \{ B \} \{ U \} \]

From the strain which is constant in the elements, the stress is calculated as:

\[ \{ s \} = [D]\{ e \} \]

The right-hand side vector \( \{ F \} \) of the global matrix equation can include either one or a combination of fixed displacements, body forces, stresses, or volume strains as essential conditions.

**Volume strain** can be incorporated into \( \{ F \} \), by defining fictitious point forces at the nodes of elements undergoing the volume change. The resulting element contribution to \( \{ F \} \) is:

\[ \{ F^{(e')} \} = tA \{ B \}^T[D]\{ \varepsilon \} \]
The actual volume change vector $e_0$ has no shear components, and its form, in the case of thermal expansion is:

$$\{e_0\} = \alpha \Delta T \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

where $\alpha$ is the thermal expansivity and $T$ is the temperature. The resulting vector has six components since each node has two degrees of freedom, determining its displacement in two dimensions.

The directional components of **point forces** are assigned individually to desired nodes of the finite element. In the equation below, $f_x$ and $f_y$ are point forces which act on node 1 of the element in horizontal and in vertical direction, respectively.

$$\{F\} = \begin{bmatrix} f_x \\ f_y \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

**Fixed displacements** are added to the matrix equation after all other right-hand terms have been applied to $\{F\}$. They are inserted as Dirichlet boundary conditions without reducing the size of the global solution matrix. This procedure is documented in the CSP3D Reference guide for the method `AssignEssentialConditions()` of the CSP3D Algorithm.

**Stresses**. Stresses acting on element faces at model boundaries are applied by distributing these as nodal point forces, but only on the boundary nodes of each triangular element. This requires a knowledge of the length of the boundary segment (area of a face), $L$, and the $x$, $y$ components of the stresses as forces per unit area. Since, the stress is distributed to two nodes, $a$ and $b$, the element face-area term ($tL$) is divided by 2.

$$\{F\} = \{F\} + \frac{tL}{2} \begin{bmatrix} \sigma_{x_a} \\ \sigma_{y_a} \\ \sigma_{x_b} \\ \sigma_{y_b} \\ 0 \\ 0 \end{bmatrix}$$

**Body forces** are accumulated into $\{F\}$, by distributing the total force among the nodes of an element:

$$\{F\} = \{F\} + (tA/3) \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ x_3 \\ y_3 \end{bmatrix}$$
where \( x \) and \( y \) are the directional components of the body force per unit volume and the indices denote nodes 1 to 3 of the triangular element (again \( t \) refers to the element thickness and \( A \) to its area).

The following code example assigns a uniform horizontal displacement to the upper boundary of a CSP model and gravity as a body force. It further prescribes a local model expansion due to deviatoric stresses exerted by the pore fluid. The algorithm \texttt{LinearElasticity} calculates the resulting displacements, strains, and stresses.

```cpp
Vectorvariable vmin;
vmin.Flag(0) = DIRICH;    // horizontal displacement is fixed
vmin.Flag(1) = PLAIN;     // no constraint is imposed in vertical direction
vmin(0) = displacement;  // horizontal displacement
vmin(1) = 0.0;           // vertical displacement
vmax = vmin;
supergroup.AssignBoundaryValues(TOP, "displacement", vmin, vmax);

LinearElasticity stress_strain( p_ref );

// Young's modulus is read automatically by LinearElasticity
stress_strain.AddOperation( OPERAND__GRADIENT_N__GRADIENT_N, 
                           "Poisson's ratio","displacement","displacement");

// collects boundary forces into the righthand vector
stress_strain.AddOperation( OPERAND__N, "force", "displacement");

// internal displacement term is (Pf=Plith-Pf) * fluid compressibility
stress_strain.AddOperation( OPERAND__N2, "fluid pressure", "displacement");

// body force gravity
stress_strain.AddOperation( BODY_OPERAND__N, "density", "displacement");
supergroup.Pass( stress_strain );
supergroup.MoveNodeCoordinatesBy("displacement");
```

The \texttt{LinearElasticity} algorithm is largely provided as a simple example of source code for mechanical calculations using \texttt{C++}. One should therefore examine the source files to get a better feel on how the finite-element approach works.
VISITORS “SEED” LIBRARY

A number of subclasses of the CSP_FEM_Visitor base class are supplied with the CSP3D3.0 distribution to facilitate the simulation of heat and solute advection, the mapping of properties to and from FiniteDifferenceGrid objects, to simulate hydraulic fracture in models where overpressure is generated, and to monitor fluxes across model- and sub-region boundaries. Example code is provided to illustrate the application of these visitor objects. To use one of the visitor subclasses, first include its declaration into the program file:

```cpp
#include "MyVisitor.h"
```

Now instantiate the visitor object and submit it to the SuperGroup or Group:

```cpp
MyVisitor guest(...);
supergroup.Accept( guest );
supergroup.GroupAccept( "the garden", guest );
```

// or alternatively

```cpp
guest.RestrictApplicationTo( "the garden" );
supergroup.Accept( guest );
```

That's it. Source code is supplied for the CSP_ID_Visitor illustrating how to traverse the mesh and collect ID numbers from finite elements, constraint points or nodes. This code example should persuade the user how easy it is to create original visitors. Feel free to contact CSP authors for advice in writing new visitors to complement the library.

BoundaryTransferVisitor (2D only)

The BoundaryTransferVisitor visits Groups determining fluxes across their boundaries. These fluxes are stored in STL maps separating the influx from the outflux and using the element ID numbers as keys. The maps are preserved after the visitation such that if the visitor is accepted a second time, only the flux values have to be changed. A specific call can be made to purge the maps before renewed application of the visitor. Boundary fluxes are always computed for the elements inside the target regions. The fluxes can also be integrated over time, if a time interval is supplied. A balance of the flux total (influx versus outflux) may also be obtained. Finally, the BoundaryTransferVisitor object can be used to assign NEUMANN (flux-type) boundary conditions.

InterfaceVisitor (2D only)

The InterfaceVisitor visits the SuperGroup or Groups determining fluxes across model boundaries or interfaces between groups where groups terminate against
The group interface can be output to VTK for visual verification. An `UpdateGeometry()` function permits to track changing group interfaces. The `InterfaceVisitor` can also monitor fluxes across a SuperGroup boundary where it coincides with a group boundary. The model boundaries are identified by values of the enumeration `SG_BOUNDARY`.

Like for the `BoundaryTransferVisitor`, fluxes are stored in STL maps separating the influx from the outflux and using element ID numbers as keys. The maps are preserved after the visitation such that if the visitor is accepted a second time, only the flux values have to be changed. A specific call can be made to purge the maps before renewed application of the visitor. Fluxes are averaged between the elements inside and outside the interface. They can also be integrated over time, if a time interval is supplied. A balance of the flux total (influx versus outflux) may also be obtained.

**CSP_ID_Visitor**

The ID visitor is provided as an example of the visitor design pattern. Thus, it is only intended to illustrate how visitors work and not for any practical application. The visitor has a private `Meschach++` `Ivec` storage vector into which it collects the ID numbers of visited objects. These may either be `Node`, `ConstraintPoint` or `Element` instances in the model. A key feature of the visitor pattern is an incomplete declaration in the base class `CSP_FEM_Visitor`, of the objects which shall be visited. This declaration allows to define access pointers to the visited objects as arguments of the pure virtual method `Visit(T* object)`. The interface of the objects pointed to by “T”, can later be accessed by the concrete `Visit(T* object)` methods in the derived visitor classes. These classes provide definitions of the visited objects in corresponding header files.

**AlterationVisitor**

The AlterationVisitor computes the reactive exchange between the pore fluid and the rock in sequential node visitations. At the end of each visitation the new speciation is written back to the node property storage. The visitor can be used for reactive transport calculations where operator splitting is used to separate transport from reaction steps, see Steefel & MacQuarie (1996) for a discussion of this and other approaches. In a typical application advection/dispersion steps alternate with the invocation of the visitor, but the user must ascertain that no more than the entire pore-volue is exchanged before the next reaction step occurs. From this restriction, it is evident that countless reaction steps are required to change the mineralogy of a low-porosity rock appreciably. To improve its performance, the AlterationVisitor monitors reaction rates and concentrations at each timestep and carries out speciation calculations only when the pore-fluid is out of equilibrium or equilibrium was not attained in the previous visitation of a node. Nodes where the compulsory property ‘H2O_aq’ is flagged DIRICH are ignored. The visitor incorporates the ODE_StiffSolver object, the functionality of which is demonstrated in the distributed ‘example6.cpp’ program.
The AlterationVisitor reads the chemical reactions it is supposed to solve from a text input file. Another text file with the same name but with the extra extension "-prop.txt" provides the charges and atomic weights of the involved aqueous and gaseous species as well as the solids:

<table>
<thead>
<tr>
<th>Species</th>
<th>Type</th>
<th>Charge</th>
<th>Atomic Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>OH_1m</td>
<td>AQUEOUS</td>
<td>-1</td>
<td>17.0073</td>
</tr>
<tr>
<td>CO2_aq</td>
<td>AQUEOUS</td>
<td>0</td>
<td>44.01</td>
</tr>
<tr>
<td>H_1p</td>
<td>AQUEOUS</td>
<td>1</td>
<td>1.0079</td>
</tr>
<tr>
<td>H2O_aq</td>
<td>AQUEOUS</td>
<td>0</td>
<td>18.02</td>
</tr>
<tr>
<td>HCO3_1m</td>
<td>AQUEOUS</td>
<td>-1</td>
<td>161.1043</td>
</tr>
<tr>
<td>Ca_2p</td>
<td>AQUEOUS</td>
<td>2</td>
<td>40.078</td>
</tr>
<tr>
<td>HCl_aq</td>
<td>AQUEOUS</td>
<td>0</td>
<td>36.46</td>
</tr>
<tr>
<td>Cl_1m</td>
<td>AQUEOUS</td>
<td>-1</td>
<td>35.4530</td>
</tr>
<tr>
<td>CC</td>
<td>SOLID</td>
<td>2710.0</td>
<td>100.09</td>
</tr>
</tbody>
</table>

This is discussed again in the carbonate-dissolution example code (Chapter 8). The AlterationVisitor is documented in more detail in the CSP3D3.0 Reference Guide.

**HeatTransportVisitor (2D only)**

A visitor is provided to efficiently advect heat in buoyancy-driven fluid-flow calculations. It maps the target property which is usually the temperature onto an internal FiniteDifferenceGrid object. After a series of advection steps the size of which is determined automatically, the temperature is interpolated back onto the finite-element mesh. The HeatTransportVisitor differs from the TransportVisitor in that it includes and automatically builds its own advection grid. The grid point spacing is equivalent to the inner radius of the smallest finite element in the mesh. Since the visitor calculates the inner radius at runtime, the physical element variable "inner radius" of scalar type must be defined in the property database file. The visitor differs from the TransportVisitor also, because it uses a simpler advection scheme. The velocity field must be defined by a nodal vector variable and, thus far, the visitor can only be used in two-dimensional models. The transported property must be a scalar variable. Due to a different implementation, the HeatTransportVisitor affords less time than the TransportVisitor to identify the grid-points that belong to each element.

The following code example advects the variable "temperature" using the transport variable "nodal velocity":

```cpp
HeatTransportVisitor advector( sgroup, "temperature", "nodal velocity" );
advector.AdvectUntil( sgroup, advection_time_interval );
moab.OutputDataToJPG("adv-temperature", "temperature", 1 );
```

After the execution of this code segment, the temperature field has been displacement by a spatially variable amount depending on the "nodal velocity" vector field and on the timespan over which the displacement occurred. In such a transient heat-advection calculation the heat-transport visitor has to be applied after each thermal diffusion step via the AdvectUntil() interface. When the transported (nodal) property is mapped back onto the mesh, all previous values are overwritten. This means that boundary conditions must be re-applied to the advected property after each of such visitations.

**H2OPropertiesVisitor**

Calculates the PVT properties of pure H$_2$O at the Nodes of a mesh. This includes the physical variables "nodal fluid density" [kg m$^{-3}$], "nodal relative fluid density" [kg m$^{-3}$] for a reference pressure, "nodal heat capacity liquid" [J kg K$^{-1}$], "nodal enthalpy liquid" [J kg] and, optionally, also the same properties of the gas phase. These have the names "nodal heat capacity steam", "nodal enthalpy steam", as well as the "nodal saturation". The nodal variables are calculated from the "absolute fluid pressure" [Pa] and the "temperature" [°C]. The calculation is carried out using the Hill (1986) equation of state for pure water which covers a temperature range from 0 to 1000°C and a pressure from atmospheric to 1 GPa. The density calculation is encapsulated into the class HILL_H2O_Density, which, in turn, draws upon the C functions defined in Hill.c.

The conditions for the reference density are set by a call to

```c
void ReferenceDensity( double Tc, double Pa );
```

And the calculations of the properties of steam can be suppressed by a call to

```c
void LiquidPhaseProperties();
```

disabling the computation of both water and steam properties (tsteam() option=4 vs. 6(=default)) in Hill.c. Since this eliminates an iteration loop, it greatly speeds up the visitations. However, it should be applied only after verifying that the PT conditions in the model are always outside of the critical region. With the

```c
void Verbose();
```

option water properties can be output to screen directly after their computation. This enables a first visual examination if a new simulation is setup.

**HydroFractureVisitor (2D only)**

The HydroFractureVisitor enhances the hydraulic “conductivity” [m s$^{-1}$] within and beyond the boundary of regions in which the "absolute fluid pressure" [Pa] exceeds a fracture-threshold pressure given by the lithostatic load [Pa] augmented by the tensile strength [Pa] of the rock. Through this capability, the visitor permits just enough fluid to dissipate from the overpressured regions that fluid pressure in these is kept at the fracture threshold. This is achieved within the framework of a Newton-Raphson iteration loop which will terminate when the absolute fluid pressure in all the elements of the model is no higher than the confining pressure plus the tensile strength of the rock.
Conceptually, this treatment of hydraulic fracture is equivalent to creating an interconnected network of small fractures in the rock, which increases the effective hydraulic conductivity sufficiently, to bleed excess fluid out of the regions where fluid pressure would otherwise have exceeded the lithostatic pressure plus the tensile strength of the rock. When the hydro-fracture visitor is accepted by the SuperGroup it will perform six different operations for each element it visits:

1. Test whether at least one element node is overpressured. Only where an overpressure exists, conductivity will be enhanced and diffused.

3. In overpressured elements, using the hydraulic conductivity, $K$, which shall be adjusted, the visitor calculates a “trial” fluid pressure gradient [Pa m$^{-1}$]

$$\nabla P = \sqrt{\left(\frac{\partial P}{\partial x}\right)^2 + \left(\frac{\partial P}{\partial y}\right)^2}$$

backward from the transport (Darcy) “velocity” [m s$^{-1}$].

4. The hydraulic conductivity, $K$, of elements with at least one overpressured node is enhanced proportional to the quotient $\text{grad} P$ over the “storativity”, $\beta$, [m$^3$ m$^{-3}$ Pa$^{-1}$] following the rationale that $\beta$ is the proportionality constant, which relates the change of pressure in an element to the amount of fluid leaving or entering it. $K$ is adjusted to the value which will just valve off enough fluid to bring the pressure down to the confining pressure plus the tensile strength of the rock.

5. Only in elements with an appreciable internal fluid-pressure gradient, an increased hydraulic conductivity fosters the dissipation of fluid. If $\text{grad} P$ is negligible the conductivity is not enhanced.

6. The new hydraulic conductivity is compared with the previous hydraulic conductivity. Only if the new conductivity is greater than the old one it is assigned to the element.

In the code segment where the HydroFractureVisitor gets applied, a decision must also be made whether regions that are no longer overpressured shall be kept at an enhanced conductivity. Pseudo code of such a hydrofracture visitor application loop might look like this:

1. Calculate the hydrostatic pressure.

2. Calculate the hydro-fracture threshold pressure as the “lithostatic pressure” plus the tensile strength of the rock, minus the hydrostatic pressure.

3. Start a Newton-Raphson iteration computing the variable “effective pressure” ($P_f - P_{\text{hydrostatic}}$) and modifying the hydraulic “conductivity” with the hydro-fracture visitor until the overpressure is dissipated.

4. Recompute fluid properties and calculate fluid transport velocity.

5. Compute new “absolute fluid pressure” = “effective pressure” + “hydrostatic pressure”.
The HydroFractureVisitor gets applied in the iteration loop (3) as follows:

```cpp
while ( dP > 0.0 && step <= MAX_PF_ITERATIONS )
{
    // 0. starting with fluid pressure before the erratic timestep
    // -----------------------------------------------------------
    sg.CopyReplace( "previous effective pressure", "effective pressure" );

    // 1. diffusing new conductivity across border of hydrofracture
    // regions (letting fracs grow)
    // -------------------------------
    sg.Accept( hydrofrac_visitor );

    // 2. re-compute fluid-pressure distribution (when hydrofractures
    // are propagated one must not consider convection)
    // -------------------------------------------------------------
    ...

    // 3. calculate new overpressures with an interration
    // -----------------------------------------------------
    sg.CopyReplace( "excess pressure", "previous excess pressure" );
    sg.Pass( excess_pressure );

    // 4. test for loop exit condition
    // --------------------------------
    sg.MinMaxOf( "excess pressure", dP_min, dP_max );
    if ( dP_max <= 0.0 ) break;
    step++;
}
```

Importantly, when a transient fluid pressure distribution is computed, a backup copy of the “effective pressure” before the step must be kept and re-used as initial condition in each of the “tries” in the Newton Raphson loop.

**TransportVisitor (2D only)**

The TransportVisitor advects properties distributed on an array of regular grids using a two-dimensional velocity vector field defined by nodal and elemental velocities on the triangular element mesh. *The velocity field must not contain discontinuities across element boundaries*, which arise when the interstitial velocity is considered and porosity changes occur from one element to the other. In this case, use a finite-volume approach to model transport (see code `example12.cpp`). To calculate incremental field displacements, the visitor uses a seven-point quadratic interpolation scheme (first order since no spatial derivatives of the advected property are considered). In contrast with the HeatTransportVisitor, the grids are not managed by the visitor itself but have to be defined and initialized beforehand. This can be done by code similar to the following:

```cpp
double  RESOLUTION;
int     n_species = 5;
Vec     xy;
supergroup.Dimensions( xy );
frame_x = RESOLUTION * 10.0;
frame_y = RESOLUTION * 10.0;
vector<FiniteDifferenceGrid>  aqueous_system(n_species);
```
for ( i=0; i<n_species; i++ )
    aqueous_system[i].Initialize( -frame_x, xy(1)+frame_x,
                                -frame_y, xy(3)+frame_y,
                                RESOLUTION, RESOLUTION);

Once the grids are initialized, values must be assigned to them in one of three ways:
(1) by setting selected rows, columns, or the entire grid to specific values (see
    interface of FiniteDifferenceGrid class); (2) by mapping properties from the finite-
    element mesh onto the grids (SuperGroup::WritePropertyToGrid()); (3) by assigning
    property distributions specified as regular-gridded data onto the grids (methods like
    In() of FiniteDifferenceGrid).

When the TransportVisitor is constructed one must specify the node and element
variables which define the velocity field:

TransportVisitor advector( supergroup_instance,
                        "nodal velocity", "velocity",
                        aqueous_system, time_increment );

TransportStepSize step_size("nodal velocity", "velocity", "inner radius");
supergroup.Pass( step_size );
advector.AdjustTimeIncrement( step_size.AdvectionTimeIncrement() );

while ( time <= total_time )
{
    ...
    advector.AdvectUntil( supergroup, time_interval );
    ...
}

In this code example, the advection time_increment is adjusted. This variable cannot
assume an arbitrary value but geometrical arguments dictate that it must be less than
the diameter of the inner circle of each triangular finite-element. The maximum
increment which satisfies this conditions is calculated on an element-by-element basis
from the velocity-vector field and the finite-element element size characteristics. This
is done by the Interrelation TransportStepSize which, in this example, uses the
variables “inner radius”, “nodal velocity” and “velocity” to identify the maximum
permissible advection time-increment for the current mesh and velocity distribution.
The call to the method AdvectUntil() prompts the visitor to advect the property
distributions on the grids in as many steps as it takes to span the time interval
specified as method argument.

Since the TransportVisitor uses the velocity-vector field specified at the
element centers and the nodes to displace upstream solute concentrations on the
transport grids, it must incorporate a scheme which interpolates concentrations
advected into the model from outside the mesh. In order to achieve this, the
TransportVisitor uses a five-point interpolation scheme which is fitted to solute
concentrations along the flow path immediately adjacent to the grid boundary. This fit
is used to extrapolate solute concentration outside of the grid. While this scheme is
capable of advecting linear gradients across boundaries, in some cases, it fails to carry
out such an extrapolation and the user receives warning messages. This may occur
when (1) the transport grid resolution is very high, (2) the flow velocities are very
high while the advection timespan is long, and (3) the concentration increases steeply
behind an inflow boundary. Putting a frame around the transport grids with fixed
concentrations may help out of such a situation.
VelocityAndVolumeFluxVisitor

The VelocityAndVolumeFluxVisitor computes the flow velocities and the scalar volume flux from the fluid pressure distribution, the porosity, and the hydraulic conductivity of visited elements, using the elements interpolation functions. The variable names "conductivity", "porosity", "fluid pressure", "velocity", "pore velocity", and "volume flux" are hardwired into the VelocityAndVolumeFluxVisitor. The visitor is intended as a post-processor which is applied after a fluid pressure computation, but only if nodal velocities are desired for restricted model regions output to visualization tools. This then has the advantage that the visitor will only use elements inside the region(s) to compute nodal properties. Otherwise non-selective post-processing would imply nodal averaging among interior and exterior finite-element results.

FiniteVolumeAdvectionVisitor (2D only)

In CSP, solute transport can either be calculated using the TransportVisitor or the FiniteVolumeAdvectionVisitor. The advantages of the second visitor are, that the finite volume mesh on which solutes are advected, is directly related to the finite element mesh, such that they are complementary: The finite-element interpolation functions describe property variations continuously inside each finite element and the finite-volume cells describe property variations continuously across element boundaries but not across FV cell boundaries which is, in turn, achieved by the elements. Due to the close relation of the 2 meshes, refinement of the finite volume mesh occurs only in those areas where small finite elements (and therefore finite volumes) are employed to capture rapid change in the field variables. This property allows to compute solute transport much more time efficient than with the TransportVisitor, which uses a uniform advection grid, where the grid spacing is determined by the size of the smallest finite element, meaning that an unnecessary high number of grid points are used in areas with large finite elements.

The application of the FiniteVolumeAdvectionVisitor is demonstrated in the code example example12.cpp. Currently, the FiniteVolumeAdvectionVisitor only works for DIRCHLET boundary conditions. There is also significant numerical dispersion due to the first-order only approach which is currently used.

The FiniteVolumeAdvectionVisitor is part of the FiniteVolume Library and can be used to simulate the advective transport of multiple solutes or of multiple fluid phases.

Finite Element Library

The current distribution of CSP ships with implementations of five finite-element types, the LinearTriangle (3-nodes), the QuadraticTriangle (6-nodes), the QuadraticBarycentricTriangle (7-nodes), the LinearTetrahedron (4-nodes), and the QuadraticTetrahedron (10-nodes). When this Guide will be distributed, the Finite Element library will probably also contain a CubicLineElement (4-nodes). Linear triangular and tetrahedral input meshes can be transformed for these element
types with the CSP_VSetConverter and hybrid meshes with both triangular and tetrahedral elements have been successfully used in CSP to model fluid flow in highly permeable but narrow fault zones.

**The Linear Triangular Element: LinearTriangle**

Linear interpolation functions, \( L \), for the triangular element are equivalent to the *area*- or so-called *natural coordinates* of a straight-sided triangle

\[
L_1 = \frac{(a_1 + b_1 x_1 + c_1 y_1)}{2A} \\
L_2 = \frac{(a_2 + b_2 x_2 + c_2 y_2)}{2A} \\
L_3 = \frac{(a_3 + b_3 x_3 + c_3 y_3)}{2A}
\]

where \( a, b, c \) are linear coefficients and \( x, y \) are the Cartesian coordinates of the nodes of the triangle. The sums in \( L \) are divided by two times the area, \( A \) of the triangle such that \( L_1 \) to \( L_3 \) sum up to 1. The area coordinates of a point \( x,y \) in a triangle are linearly related to its Cartesian coordinates such that

\[
\begin{bmatrix}
1 \\
x \\
y
\end{bmatrix}
= \begin{bmatrix}
1 & 1 & 1 \\
x_1 & x_2 & x_3 \\
y_1 & y_2 & y_3
\end{bmatrix}
\begin{bmatrix}
L_1 \\
L_2 \\
L_3
\end{bmatrix}
\]

Here, the indexed Cartesian coordinates in the matrix are the node coordinates of the triangle (Fig. 1).

![Fig. 18: Counterclockwise node numbering in linear triangular element.](image)

The coefficients of the area-coordinate functions are calculated from the Cartesian coordinates of the nodes \( n_1 \) to \( n_3 \).
\[ a_1 = x_2y_3 - x_3y_2 \]
\[ a_2 = x_3y_1 - x_1y_3 \]
\[ a_3 = x_1y_2 - x_2y_1 \]
\[ b_1 = y_2 - y_3 \]
\[ b_2 = y_3 - y_1 \]
\[ b_3 = y_1 - y_2 \]
\[ c_1 = x_3 - x_2 \]
\[ c_2 = x_1 - x_3 \]
\[ c_3 = x_2 - x_1 \]

The area of the triangle is
\[ A = \frac{1}{2} \left( x_2y_3 + x_1y_2 + x_3y_1 - x_2y_1 - x_3y_2 - x_1y_3 \right) \]

The spatial derivatives of the interpolation functions \((N=L)\) in generic matrix form are
\[
[B] = \begin{bmatrix}
\frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial x} & \frac{\partial N_3}{\partial x} \\
\frac{\partial N_1}{\partial y} & \frac{\partial N_2}{\partial y} & \frac{\partial N_3}{\partial y}
\end{bmatrix} = \begin{bmatrix}
b_1 & b_2 & b_3 \\
c_1 & c_2 & c_3
\end{bmatrix}
\]

This matrix is expanded for linear elasticity problems, where each node has 2 degrees of freedom (x,y displacements). This expanded form is also known as the '\(B\)' matrix:
\[
[B] = 
\begin{bmatrix}
b_1 & 0 & b_2 & 0 & b_3 & 0 \\
0 & c_1 & 0 & c_2 & 0 & c_3 \\
c_1 & b_1 & c_2 & b_2 & c_3 & b_3
\end{bmatrix}
\]

For the linear triangle, the interpolation functions can be integrated analytically over the area of the element, using the general formula
\[
\int_{A} L_1 L_2 L_3 dA = 2A \frac{l!j!k!}{(2+l+j+k)!}
\]

where \(l, j, k\) represent exponents of the area-coordinate functions. These exponents are equivalent to the degree of the interpolation functions and their products as they arise during matrix multiplication. To illustrate the use of the integration formula for a single entry into the finite-element matrix integral consider
\[
\int_{A} N_i \frac{\partial N_i}{\partial x} dA = \frac{b_i}{2A} \int_{A} N_i dA = \frac{b_i}{2A} 2A \frac{l!j!k!}{(2+l+j+k)!} = \frac{b_i}{6}
\]

Here, the partial derivative of the interpolation function with respect to \(x\) is a constant. Therefore, it can be pulled out of the integral, and, since the interpolation functions \(N_2\) and \(N_3\) do not appear in the equation, their exponents are zero.

For the conductance matrix \(K\) integration in this manner for all terms gives
The Linear Tetrahedral Element: LinearTetrahedron

The interpolation functions for the LinearTetrahedron are implemented similar to those of the linear triangle and the barycentric triangular element, with the difference that a third spatial coordinate is incorporated. The interpolation functions are therefore expressed as volume coordinates instead of area coordinates

\[
L_i = \frac{a_i + b_i x + c_i y + d_i z}{6V}, \quad L_i(x_j, y_j, z_j) = \delta_{ij}
\]

Where \( \delta_{ij} \) (Kronecker-Delta) simply means that the interpolation functions are normal to their weights. At any coordinate, the four orthonormal interpolation functions \( L \) sum up to 1

\[
\sum_{i=1}^{4} L_i(x, y, z) = 1
\]

Cartesian \( (x,y,z) \) and volume \( (L) \) coordinates have the relation

\[
\begin{bmatrix}
1 \\
x \\
y \\
z
\end{bmatrix} = \begin{bmatrix}
1 & 1 & 1 & 1 \\
x_1 & x_2 & x_3 & x_4 \\
y_1 & y_2 & y_3 & y_4 \\
z_1 & z_2 & z_3 & z_4
\end{bmatrix} \begin{bmatrix}
L_1 \\
L_2 \\
L_3 \\
L_4
\end{bmatrix}
\]

where the Cartesian coordinate indices correspond to the four nodes of the tetrahedron. The determinant of the square \((4\times4)\) matrix is equivalent to \( 6V \), if the nodes are numbered so that the sequence 1-2-3 runs counterclockwise when viewed from node 4 (Fig. 19).
The coefficients for the basis functions are computed from the global node coordinates as

\[
a_i = -x_i(y_j z_k - z_j y_k) - x_j(y_i z_k - z_i y_k) - x_k(y_i z_j - z_i y_j)
\]

\[
b_i = (y_j z_k - z_j y_k) + (y_i z_k - z_i y_k) + (y_i z_j - z_i y_j)
\]

\[
c_i = (z_i x_k - x_i z_k) + (z_i x_j - x_i z_j) + (z_i x_j - x_i z_j)
\]

\[
d_i = (x_i y_j - y_i x_j) + (x_i y_k - y_i x_k) + (x_i y_k - y_i x_k)
\]

The remaining coefficients are obtained by cyclic permutations of the indices with a sign change for each permutation.

The volume of the tetrahedron is

\[
V = \frac{(a_1 + a_2 + a_3 + a_4)}{6}
\]

Differentiation of the interpolation functions gives

\[
\frac{\partial L_i}{\partial x} = \frac{b_i}{6V}, \quad \frac{\partial L_i}{\partial y} = \frac{c_i}{6V}, \quad \frac{\partial L_i}{\partial z} = \frac{d_i}{6V}
\]

while the general analytical formula for the integration is

\[
J(k, l, m, n) = \frac{1}{V} \int L_1^k L_2^l L_3^m L_4^n dV = 6V \frac{k!l!m!n!}{(3+k+l+m+n)!}
\]

\(k, l, m, n\) are equivalent to the exponents of the interpolation functions \(L_1\) to \(L_4\). Each exponent is unity for linear, 2 for quadratic, and 3 for cubic interpolation function terms. The factor 3 in the denominator, is equal to the space dimension of the element. When an interpolation function integral only contains first order terms, the integral reduces to

\[
I(1,1,1,1) = 6V \frac{1}{5040} = 6V1.1904762 \times 10^{-3}
\]

\((0! = 1)\). Evaluating, the volume integral \(N^T r N dV\), one arrives at

\[
[I] = \int \begin{bmatrix} N_1 \\ N_2 \\ N_3 \\ N_4 \end{bmatrix} r \begin{bmatrix} N_1 & N_2 & N_3 & N_4 \end{bmatrix} dV = \int \begin{bmatrix} N_1 \\ N_2 \\ N_3 \\ N_4 \end{bmatrix} \begin{bmatrix} N_1^2 & N_1 N_2 & N_1 N_3 & N_1 N_4 \\ N_2 N_1 & N_2^2 & N_2 N_3 & N_2 N_4 \\ N_3 N_1 & N_3 N_2 & N_3^2 & N_3 N_4 \\ N_4 N_1 & N_4 N_2 & N_4 N_3 & N_4^2 \end{bmatrix} dV
\]

Individual product terms are integrated as follows...
\[ \int_V N_1 N_1 dV = \int_V N_1^2 dV = 6V \frac{20!}{(3+2+0+0+0)!V} = \frac{1}{10} V \]

\[ \int_V N_1 N_2 dV = 6V \frac{110!}{(3+1+1+0+0)!V} = \frac{1}{20} V \]

and so forth, until one arrives at the usual integrals like that over the interpolation function products

\[
[I] = \begin{vmatrix}
\frac{1}{10} & \frac{1}{20} & \frac{1}{20} & \frac{1}{20} \\
\frac{1}{10} & \frac{1}{20} & \frac{1}{20} & \frac{1}{20} \\
\frac{1}{20} & \frac{1}{20} & \frac{1}{20} & \frac{1}{20} \\
\frac{1}{20} & \frac{1}{20} & \frac{1}{20} & \frac{1}{20}
\end{vmatrix} V
\]

Integration exponents for interpolation function derivatives are 0 since these functions are linear for the tetrahedral element.
The QuadraticTriangle with 6 Nodes

Quadratic triangular elements (Fig. 19) are used very often for finite-element computations of elasticity problems. The quadratic order of the interpolation functions of this element makes displacement derivatives continuous across element boundaries (not across corner nodes) and special ways of numerical integration can rule out zero-energy modes of deformation which lead to unrealistic results. The element can also be constructed for three-dimensional computations in which case 3D coordinates are transformed into the plane of the 2D local reference element. In linear-elasticity applications, mesh convergence can usually be obtained in contrast with linear elements which give appaling results. The quadratic interpolation implies that displacement derivatives are represented as bilinear fields inside each element.

In the following, the steps are discussed of how one solves a linear elasticity problem using quadratic triangular finite elements, including the post-processing calculation of strains and stresses at the elements integration points.

Given the reference triangle shown in Figure 20, interpolation functions \( N_i \) (= shape functions since the isoparametric approach is used) in the local coordinate system, can be defined as:

\[
\begin{align*}
N(3) &= 4. \times r \times (1. - r - s) \\
N(4) &= 4. \times r \times s \\
N(5) &= 4. \times s \times (1. - r - s) \\
N(0) &= 1. - r - s - N(3)/2. - N(5)/2. \\
N(1) &= r - N(3)/2. - N(4)/2. \\
N(2) &= s - N(4)/2. - N(5)/2. \\
\end{align*}
\]

and their spatial derivatives \( dN/dr, dN/ds \), with regard to the local reference frame:

\[
\begin{align*}
DNS(0) &= -3. + 4.*r + 4.*s
\end{align*}
\]
DNS(1) = 0.
DNS(2) = -1. + 4.*s
DNS(3) = -4.*r
DNS(4) = 4.*r
DNS(5) = 4. - 4.*r - 8.*s

DNR(0) = -3. + 4.*r + 4.*s
DNR(1) = -1. + 4.*r
DNR(2) = 0.
DNR(3) = 4. - 8.*r - 4.*s
DNR(4) = 4.*s
DNR(5) = -4.*s

**Jacobian Transformation.** The use of a local coordinate system makes it necessary to transform element contributions, which are calculated on the reference element, into the global coordinate system. This is achieved via a Jacobian coordinate transformation. For this purpose one needs to form a Jacobian matrix, \( J \), with the size of spatial dimensions x spatial dimensions (2 x 2 in 2D):

\[
J(r,s) = \begin{bmatrix}
\frac{\partial N_1}{\partial r}(r,s) & \cdots & \frac{\partial N_6}{\partial r}(r,s) \\
\frac{\partial N_1}{\partial s}(r,s) & \cdots & \frac{\partial N_6}{\partial s}(r,s)
\end{bmatrix}
\begin{bmatrix}
x_1 & y_1 \\
\vdots & \vdots \\
x_6 & y_6
\end{bmatrix}
\]

\( J \) can be calculated as the product of the shape-function derivative matrix, \( DN \), calculated at the point \((r,s)\) in the local coordinate system, and the global node-coordinate matrix, \( XY \) (cf. Zienkiewicz, Vol I, (4th ed.) p. 162). With the inverse of \( J \) one can carry out the desired transformation of the local to the global shape-function derivative matrix, \( DN(x,y) \), but \( J^{-1} \) is only valid at the transformation point \((x,y)\) of the local point \((r,s)\) where the local shape-function derivatives were calculated. Therefore, \( J^{-1} \) must be calculated again at each point where derivatives shall be transformed into the global coordinate frame.

In summary, to calculate the global shape-function derivative matrix \( DN(x,y) \), at a point \((x,y)\) representing the mapping of point \((r,s)\), the following steps must be performed:

1. Form \( DN(r,s) \) and \( XY \),
2. Form \( J = DN(r,s) \times XY \)
3. Invert \( J \)
4. Premultiply \( J^{-1} \) with \( DN(r,s) \): \( DN(x,y) = J^{-1} * DN(r,s) \)

As a next step, depending on the degrees of freedom per node of the problem which shall be solved, \( DN(x,y) \) must be expanded into a correspondingly larger matrix, \( B \). Taking a linear elasticity problem in 2D, the nodal degree of freedom is 2, and the the often so-called \( B \) matrix is:
This matrix is built by substitution of the elements of \(DN(x,y)\) into the corresponding locations. Since there is no analytical integration formula for the curved (deformed) isoparametric triangular element, numerical integration has to be used. The numerical integration formula presented below, however, is only exact for an undeformed (straight-sided) triangle.

**Element Stiffness Matrix.** The element’s contribution to the global stiffness matrix, \(K\), given by the spatial integral

\[
K^{(e)} = \int\int B^T DB \, dxy
\]

over the matrix product of \(B\)-transposed \(D\) \(B\), where \(D\) is the 3 x 3 material property matrix, is approximated numerically as the sum

\[
K^{(e)\text{num}} = \sum_{i=1}^{nip} |J| w_i B_i^T DB_i
\]

of such products computed at \(nip\) integration points in the triangle. In this example, the midside nodes are chosen as integration points (dark nodes in Fig. 1, \(nip=3\)). \(|J|\) refers to the determinant of the Jacobian matrix which must be non-singular and \(w_i\), is a weight that is applied to the matrix contribution from each integration point. This weighting depends on the area of the reference element and the number of integration points chosen. Thus, with the reference element’s area being \(\frac{1}{2}\) and \(nip=3\), it follows \(w_i = \frac{1}{6}\) such that the integral terms span the area of 0.5. Due to the location of the integration points, all weights can be treated equally.

For the case of **plane strain** (no strain normal to the plane of the two-dimensional model) which is considered here, the \(D\) matrix incorporates the properties Young’s modulus, \(E\), and Poisson’s ratio, \(\nu\):

\[
D = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & 0 \\ \nu & 1-\nu & 0 \\ 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix}
\]

**Matrix Inversion.** A common linear-elasticity problem where the dependent variable is the nodal displacement, is defined by the global stiffness matrix, \(K\) on the lefthand side and a prescribed displacement boundary-condition vector \(uv\) on the righthand side

\[
Kd = uv
\]
The solution is returned into the nodal displacement vector $\mathbf{d}$, with dimensions nodes x degrees of freedom per node. Initial stresses and strains, boundary stresses and body forces are ignored.

**Strains & Stresses.** Among others Barlow (1976, Int. J. Num. Meth. Eng. 10, 243-251) has shown that the best locations for strain (change of length and shape) and stress computations in the quadratic (and cubic) triangular element are the midside nodes. He recommends to linearly interpolate or extrapolate these to the element center or to the nodes, respectively, for the later usual visualization (linear interpolation is recommended, because derivatives of quadratic- are of course linear functions). If strains or stresses are interpolated to the nodes, an averaging of nodal values computed by the elements sharing the node is often applied since the stresses are not continuous from element to element when quadratic interpolation functions are used.

From the nodal displacements in the global coordinate system, $dx_i$, $dy_i$ where $i$ refers to node $i$, the strains at each integration point are calculated as

$$
\begin{bmatrix}
\varepsilon_{x_j} \\
\varepsilon_{y_j} \\
\varepsilon_{xy_j}
\end{bmatrix} =
\begin{bmatrix}
\frac{\partial N_1}{\partial x} & 0 & \ldots & \frac{\partial N_6}{\partial x} & 0 \\
0 & \frac{\partial N_1}{\partial y} & \ldots & 0 & \frac{\partial N_6}{\partial y} \\
\frac{\partial N_1}{\partial y} & \frac{\partial N_1}{\partial x} & \ldots & \frac{\partial N_6}{\partial y} & \frac{\partial N_6}{\partial x}
\end{bmatrix}
\begin{bmatrix}
dx_i \\
dy_i \\
dx_6 \\
dy_6
\end{bmatrix}
$$

Where $j$ refers to the integration point $j$. This multiplication is equivalent to multiplying $\mathbf{B}_i$, the expanded shape function $x,y$-derivative matrix (see above) at the integration point $i$ (a midside node) with the nodal displacement vector, $\mathbf{d}^{(e)}$, for the element:

$$\varepsilon_i = \mathbf{B}_i \mathbf{d}^{(e)}$$

The $\mathbf{B}_i$ are computed before the nodal displacements are applied to change the elements shape in global coordinates. The stresses at the integration points are finally computed from the strains as

$$\sigma_j = D\varepsilon_i$$

**Principal Strain- & Stresses Axes.** The principal strain / stress axis are found as the Eigenvectors of the calculated strain and stress tensors

$$\Sigma^{2D} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} \\ \sigma_{xy} & \sigma_{yy} \end{bmatrix}, \quad \Sigma^{3D} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{xy} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{xz} & \sigma_{yz} & \sigma_{zz} \end{bmatrix}$$

in which the off-diagonal elements are the shear strains / stresses. The Eigenvalues represent the principal strain /stress magnitudes (in the directions defined by the Eigenvectors). The Eigenvector matrix EIG contains the principal axis 1, 2, 3 in its corresponding rows.

**Dilatation.** The sum of the Eigenvalues of the strain tensor.
Mean Stress. The average of the principal stresses.

Testing. A series of tests can be performed to detect errors in the implementation of newly implemented elements like the discussed one. Thus, for any point inside the implemented element (including its boundary) and with reference to the global coordinate system:

1. The interpolation functions must sum up to 1 (down to numerical precision).
2. Their derivatives must sum up to zero.
3. The sum of the products of interpolation function derivatives with nodal scalar variables must represent the gradient of a scalar variable prescribed on the nodes anywhere inside the triangle.
4. The element volume as calculated by integrating the shape functions over all integration points in the element must be correct.
5. In an even-sided symmetric transformation of an element's shape, for instance an equal-sided triangle, the length of the vectors defined by the interpolation functions derivatives at the nodes and integration points, must be the same at opposite nodes. Thus, a transformation into the highest possible symmetry makes the testing easier.

Global to Local. Interpolation of property values at a point specified in global coordinates for mapped elements.

Property Value Gradients. If the gradient of a property value is desired at a global point \((x,y)\), the global interpolation-function derivatives at the element’s nodes can be interpolated to this point using the element’s interpolation functions. For this purpose, the global \(x,y\)-derivatives of shape function 1 at node 1, shape function 2 at node 2 and so forth are assembled into a derivative matrix \(DN\) (size spatial dimensions x nodes). These data are now interpolated with the element’s interpolation functions to the point of interest, leading to \(DN'(x,y)\), just as one would interpolate any nodal property value. The desired property-gradient is found by summing the products of the interpolated derivatives with the corresponding nodal property values:

\[
p_{i,x}(x,y) = \sum_{i=1}^{\text{nodes}} \frac{\partial N_i}{\partial x}(x,y) p_i
\]

where \(p_{i,x}(x,y)\) is the property gradient in the \(x\)-coordinate direction at the global point \((x,y)\).

Inter-Element Continuity. Once an element has been verified in isolation, further tests can be devised for element patches in order to see whether the \(n\)th order derivatives of the shape functions are continuous across the sides of the elements and their corner nodes. For the quadratic triangle they are not continuous across the corner nodes.
The Barycentric Quadratic Triangle with 7 Nodes

The barycentric triangular finite element has quadratic interpolation functions involving an extra node in the element’s center. This particularity makes the element well suited for buoyancy-driven fluid flow and elastic-viscous deformation simulations. Since this element can be dramatically over integrated using a 7 or 13-point numerical integration scheme it is capable of very accurately representing integrated field variables like varying fluid density or body forces. The 13-point integration scheme is only recommended for calculations where buoyancy forces vary dramatically over short distances but the overall scale of the model dictates that the mesh has to be relatively coarse. An example of such a calculation would be a model of dike swarm intrusion. The node-numbering in a right-handed coordinate system and the splitting of the elemental area in areal coordinates is shown in Figure 21.

![Figure 21: The triangular barycentric element. (Left): Node-numbering scheme; (center): Natural (area) coordinate breakdown of the reference element; (right): Coordinate convention for the local coordinate system.](image)

The origin of the local coordinate system is located in node 1 of the barycentric triangle and the limbs are stretched such that node 2 and 3 have the local two-dimensional coordinates (1,0) and (0,1), respectively.

Element Interpolation Functions in Global Coordinates

The Barycentric Quadratic Triangle is implemented as an isoparametric element using a local coordinate system and Jacobian transformations. However, element interpolation functions in global coordinates are still needed to interpolate property values at points whose coordinates are known only in the global coordinate system. Interpolation using this approach is correct only as long as the triangle remains straight sided.

As for the Linear Triangle element, interpolation functions in global coordinates are composed using area coordinates, \( L \). The seven interpolation functions \( N_1 \) to \( N_{123} \) are now expressed in terms of the area coordinate functions:
\[ N_1 = L_1(2L_1 - 1) + 3L_1L_2L_3 \]
\[ N_2 = L_2(2L_2 - 1) + 3L_1L_2L_3 \]
\[ N_3 = L_3(2L_3 - 1) + 3L_1L_2L_3 \]
\[ N_{12} = 4L_1L_2 - 12L_1L_3L_3 \]
\[ N_{13} = 4L_1L_3 - 12L_2L_1L_3 \]
\[ N_{23} = 4L_2L_3 - 12L_2L_2L_3 \]
\[ N_{123} = 27L_1L_2L_3 \]

\( N_{123} \) is sometimes called the bubble function and is unique in that the corresponding node is not shared among elements. The first partial derivatives of these functions with regard to the node coordinates \( x \) are:

\[
\frac{\partial N_1}{\partial x} = \left(4L_1 - 1\right) \frac{\beta_1}{2A} + 3L_1L_2L_3
\]
\[
\frac{\partial N_2}{\partial x} = \left(4L_2 - 1\right) \frac{\beta_2}{2A} + 3L_1L_2L_3
\]
\[
\frac{\partial N_3}{\partial x} = \left(4L_3 - 1\right) \frac{\beta_3}{2A} + 3L_1L_2L_3
\]
\[
\frac{\partial N_{12}}{\partial x} = \left(L_2\beta_1 - L_1\beta_2\right) \frac{2}{A} - 12L_1L_2L_3
\]
\[
\frac{\partial N_{23}}{\partial x} = \left(L_3\beta_2 - L_2\beta_3\right) \frac{2}{A} - 12L_1L_2L_3
\]
\[
\frac{\partial N_{31}}{\partial x} = \left(L_1\beta_3 - L_3\beta_1\right) \frac{2}{A} - 12L_1L_2L_3
\]
\[
\frac{\partial N_{123}}{\partial x} = 27L_1L_2L_3
\]

Again the analytical integration formula already presented for the triangular element could be used but this would become a fairly complicated expression:

\[
\int_A N_1N_2N_3N_{12}N_{13}N_{23}N_{123}dA = A \frac{j!k!l!m!n!o!p!}{(3+j+k+l+m+n+o+p)!}
\]

**Local Coordinates**

Figure 17 illustrates a local coordinate reference frame for the barycentric element. Since local coordinates are used, local shape function derivatives required at the integration points are constants for the reference element. At the corresponding nodes the derivatives are:

<table>
<thead>
<tr>
<th>node</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>12</th>
<th>23</th>
<th>31</th>
<th>123</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varepsilon_i )</td>
<td>-3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>-5</td>
<td>0</td>
</tr>
<tr>
<td>( \eta_i )</td>
<td>-3</td>
<td>0</td>
<td>3</td>
<td>-5</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
The use of a local reference element has the advantage that the sides of the triangle can deform into curvilinear shapes without compromising on the accuracy of the numerically evaluated integrals. The Jacobian matrix for each integration point is formed from the shape (=interpolation) function derivatives at this point and the elements global coordinates:

\[
\begin{bmatrix}
\frac{\partial N_1}{\partial \xi} & \cdots & \frac{\partial N_n}{\partial \xi} \\
\frac{\partial N_1}{\partial \eta} & \cdots & \frac{\partial N_n}{\partial \eta} \\
\frac{\partial N_1}{\partial \zeta} & \cdots & \frac{\partial N_n}{\partial \zeta}
\end{bmatrix}
\begin{bmatrix}
x_1 & y_1 & z_1 \\
\vdots & \vdots & \vdots \\
x_n & y_n & z_n
\end{bmatrix}
\]

**Numerical Integration**

The integration points and their weights are defined in terms of the local coordinates of the reference element. The location of the points \((IP7\text{ or } IP13, \text{ in } L \text{ coordinates})\) and their weights \((W7\text{ or } W13)\) are computed as follows:

**Matrix IP7(7,3), IP13(13,3);**

```plaintext
double g1, g2, g3, g4;
g1 = (6.0+sqrt(15.0))/21.0;
g2 = (6.0-sqrt(15.0))/21.0;
g3 = (9.0-2.0*sqrt(15.0))/21.0;
g4 = (9.0+2.0*sqrt(15.0))/21.0;

for ( i=0; i<7; i++ ) IP7(i,0)=1./3.;  IP7(i,1)=1./3.;
IP7(0,0)=g1; IP7(1,0)=g1;
IP7(2,0)=g3;  IP7(2,1)=g1;
IP7(3,0)=g1; IP7(3,1)=g3;
IP7(4,0)=g2; IP7(4,1)=g2;
IP7(5,0)=g4; IP7(5,1)=g4;
IP7(6,0)=g2; IP7(6,1)=g2;
for ( i=0; i<7; i++ ) IP7(i,2) = 1.0 - IP7(i,1) - IP7(i,0);

for ( i=0; i<13; i++ ) IP13(i,0) = 0.3333333;
IP13(0,0) = 0.3333333; IP13(1,0) = 0.3333333; IP13(2,0) = 0.3333333;
IP13(3,0) = 0.3333333; IP13(4,0) = 0.3333333; IP13(5,0) = 0.3333333;
IP13(6,0) = 0.3333333; IP13(7,0) = 0.3333333;
IP13(8,0) = 0.3333333;

for ( i=0; i<13; i++ ) W13(i) = -0.14957;
W13(0) = -0.14957;
```

**Vector W7(7), W13(13);**

W7(0) = 0.1756152;
W7(1) = 0.1756152;
W7(2) = 0.1756152;
W7(3) = 0.1756152;
W7(4) = 0.1756152;
\[ W_{13}(5) = 0.0533472; \]
\[ W_{13}(6) = 0.0533472; \]
\[ W_{13}(7) = 0.0771136; \]
\[ W_{13}(8) = 0.0771136; \]
\[ W_{13}(9) = 0.0771136; \]
\[ W_{13}(10) = 0.0771136; \]
\[ W_{13}(11) = 0.0771136; \]
\[ W_{13}(12) = 0.0771136; \]
\[ W_{7}(0) = 0.225; \]
\[ W_{7}(1) = \frac{155.0 + \sqrt{15.0}}{1200.0}; \]
\[ W_{7}(2) = \frac{155.0 + \sqrt{15.0}}{1200.0}; \]
\[ W_{7}(3) = \frac{155.0 + \sqrt{15.0}}{1200.0}; \]
\[ W_{7}(4) = \frac{155.0 - \sqrt{15.0}}{1200.0}; \]
\[ W_{7}(5) = \frac{155.0 - \sqrt{15.0}}{1200.0}; \]
\[ W_{7}(6) = \frac{155.0 - \sqrt{15.0}}{1200.0}; \]
The QuadraticTetrahedron with 10 Nodes

Fig 20 show the node-numbering for the QuadraticTetrahedron finite element. This element is not seen as such during visualization because it is partitioned into subtetrahedra when meshes are output to VTK or other OpenGL based visualization tools as OpenGL shading can only represent linear property variations.

![Quadratic Tetrahedron](image)

Fig. 22: The QuadraticTetrahedron with coordinate reference frame and node numbering scheme.

Like the other higher-order elements in CSP, the QuadraticTetrahedron is isoparametric and can therefore be used in models of progressive deformation where element boundaries become curved after the first displacement increment. The interpolation functions in local coordinates are

\[
\begin{align*}
L_1 &= 1.0 - L_2 - L_3 - L_4; \\
N(0) &= L_1 \times (2.0 \times L_1 - 1.0); \\
N(1) &= L_2 \times (2.0 \times L_2 - 1.0); \\
N(2) &= L_3 \times (2.0 \times L_3 - 1.0); \\
N(3) &= L_4 \times (2.0 \times L_4 - 1.0); \\
N(4) &= 4.0 \times L_1 \times L_2; \\
N(5) &= 4.0 \times L_2 \times L_3; \\
N(6) &= 4.0 \times L_3 \times L_1; \\
N(7) &= 4.0 \times L_2 \times L_4; \\
N(8) &= 4.0 \times L_1 \times L_4; \\
N(9) &= 4.0 \times L_1 \times L_4;
\end{align*}
\]

Note that \(L_1 = 1 - r - s - t\), \(L_2 = r\), \(L_3 = s\), \(L_4 = t\), see Bathe, p. 256 for midside nodes and Huyakorn & Pinder (1983) p. 91 for corner node interpolation function specifications, respectively.

The local partial derivatives of the interpolation functions at arbitrary points inside of the tetrahedron are with respect to the \(r\) local coordinate

\[
\begin{align*}
\text{DNR}(0) &= 1. - 4. \times (1. - r - s - t); \\
\text{DNR}(1) &= -1. + 4. \times r; \\
\text{DNR}(2) &= 0.; \\
\text{DNR}(3) &= 0.; \\
\text{DNR}(4) &= -4. \times r + 4. \times (1. - r - s - t); \\
\text{DNR}(5) &= 4. \times s; \\
\text{DNR}(6) &= -4. \times s; \\
\text{DNR}(7) &= 4. \times t; \\
\text{DNR}(8) &= 0.;
\end{align*}
\]
\[ DNR(9) = -4. \times t; \]

and with respect to the \( s \) local coordinate

\[
\begin{align*}
DNS(0) &= 1. - 4. \times (1. - r - s - t); \\
DNS(1) &= 0.; \\
DNS(2) &= -1. + 4. \times s; \\
DNS(3) &= 0.; \\
DNS(4) &= -4. \times r; \\
DNS(5) &= 4. \times r; \\
DNS(6) &= -4. \times s + 4. \times (1. - r - s - t); \\
DNS(7) &= 0.; \\
DNS(8) &= 4. \times t; \\
DNS(9) &= -4. \times t;
\end{align*}
\]

as well as with respect to the \( t \) local coordinate

\[
\begin{align*}
DNT(0) &= 1. - 4. \times (1. - r - s - t); \\
DNT(1) &= 0.; \\
DNT(2) &= 0.; \\
DNT(3) &= -1. + 4. \times t; \\
DNT(4) &= -4. \times r; \\
DNT(5) &= 0.; \\
DNT(6) &= -4. \times s; \\
DNT(7) &= 4. \times r; \\
DNT(8) &= 4. \times s; \\
DNT(9) &= 4. \times (1. - r - s - t) - 4. \times t;
\end{align*}
\]

The local partial derivatives at the node points are

<table>
<thead>
<tr>
<th>node</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r = \xi )</td>
<td>-3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>-2</td>
<td>2</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>( s = \eta )</td>
<td>-3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>-2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>-2</td>
</tr>
<tr>
<td>( t = \zeta )</td>
<td>-3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>-2</td>
<td>0</td>
<td>-2</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6: Local derivatives of interpolation functions.

All four integration points have a weight of \( \frac{1}{4} \). Their locations in local coordinates are

\[
\begin{align*}
IP(0,0) &= 0.13819660, \ IP(0,1) = 0.13819660, \ IP(0,2) = 0.13819660; \\
IP(1,0) &= 0.58541020, \ IP(1,1) = 0.13819660, \ IP(1,2) = 0.13819660; \\
IP(2,0) &= 0.13819660, \ IP(2,1) = 0.58541020, \ IP(2,2) = 0.13819660; \\
IP(3,0) &= 0.13819660, \ IP(3,1) = 0.13819660, \ IP(3,2) = 0.58541020;
\end{align*}
\]

The volume of the tetrahedron is found by numerically integrating the element interpolation functions over the tetrahedron.

The QuadraticTetrahedron is documented in more detail in the HTML Reference Guide.
PDE OPERATORS LIBRARY (ALGORITHMS)

To solve PDEs with the finite-element method, they are converted into linear-algebraic matrix problems of the form

\[
\begin{bmatrix}
A
\end{bmatrix}\{x\} = \{b\}
\]

where, on the lefthand side, the square matrix \( A \) is called the global solution matrix, the vector \( x \), is called the solution vector, and the vector \( b \), is called the righthand vector. The size \( n \) of \( A \) is equivalent to the number of unknowns squared. For “scalar” problems where the dependent variable is a scalar, like “fluid pressure”, \( n \) is equivalent to the number of non-Dirichlet nodes. For problems where the dependent variable is a vector property, like displacement for instance, there are several degrees of freedom (DOF) per node, and the size of \( A \) is \( n \) DOF \( \times \) \( n \) DOF. Since \( A \) holds the elements contributions to the global conductance or stiffness of the equation system it is sometimes also called the stiffness or conductance matrix. The volume or area integrals from the elements are accumulated into \( A \) one by one until one obtains a piecewise integration over the whole model. Integration constraints and constants are supplied via \( b \). It will hold both, essential and initial conditions.

The PDE Operators in CSP are grouped into those which compute (matrix) contributions to be assembled into the lefthand global solution matrix and are derived from the MathOperatorLHS base class, and into those which compute (vector) contributions to the righthand vector and are derived from the MathOperatorRHS base class. It was a design intent to completely decouple PDE operators from the type of finite elements used, and to allow for the integration of both element and node properties. Necessarily, there is a difference in the form of PDE operators which use analytical integration as is only possible for straight-sided elements, and PDE operators which employ numerical integration. This distinction is expressed in the PDE operator names:

- Integral... analytical integration (LinearTriangle or LinearTetrahedron)
- NumIntegral... numerical integration (all higher order elements)

Matrix contributions from the elements are called element “stencils”. These are again square matrices with a size equal to the number of nodes per element times the DOF per node. Further naming conventions used for the PDE operators are (with reference to the number of nodes per element):

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>N, NT</td>
<td>interpolation function vector and its transposed with ( n )=nodes entries</td>
</tr>
<tr>
<td>N2</td>
<td>double rock interpolation function block matrix</td>
</tr>
<tr>
<td>dN, dNT</td>
<td>interpolation function derivatives matrix (dimensions x nodes)</td>
</tr>
<tr>
<td>P, PT</td>
<td>interpolation function matrix and its transposed for vector properties</td>
</tr>
<tr>
<td>B, BT</td>
<td>( dN ) expanded for multiplication with 3 x 3 (2D) or 6 x 6 (3D) matrix</td>
</tr>
<tr>
<td>( op )</td>
<td>the material property(ies) which enter the MTRL matrix in integral</td>
</tr>
<tr>
<td>( lhsop )</td>
<td>lefthandside ( op ) if there arises an ambiguity in names when same operator is used in the left and the righthandside of the equation</td>
</tr>
<tr>
<td>( dV )</td>
<td>area- or volume integral</td>
</tr>
</tbody>
</table>
When numerical integration is used or the properties are not constant on the element, interpolation functions and their derivatives must be computed at the integration points just like the material properties. This is how property variations are interpolated over the element and then integrated. In some cases the operator implementations are restricted such that only scalar properties or only element properties can be integrated by the specific operators. In these cases one will get a **FATAL_ERROR** message if one tries to use incompatible variables.

To reiterate, in detail, what is stated in the table, the matrix forms used further below are listed:

**Element interpolation functions:**

\[
\begin{bmatrix}
N_1 & \cdots & N_n
\end{bmatrix}, \quad
\begin{bmatrix}
N_1 \\
\vdots \\
N_n
\end{bmatrix},
\]

\[
\begin{bmatrix}
N_1 & \cdots & N_n
\end{bmatrix}
\]

\[
\frac{\partial N_1}{\partial x} \quad \cdots \quad \frac{\partial N_n}{\partial y}
\]

The expansion for multiplication with tensor variables (as is used in mechanics) is (for two-dimensional applications):
For three dimensional applications, an entry for a single node (where the last three rows refer to the off-diagonal tensor components) is:

\[
B_i = \begin{bmatrix}
\frac{\partial N_i}{\partial x} & 0 & \cdots & \frac{\partial N_i}{\partial x} & 0 \\
0 & \frac{\partial N_i}{\partial y} & \cdots & 0 & \frac{\partial N_i}{\partial y} \\
\frac{\partial N_i}{\partial y} & \frac{\partial N_i}{\partial x} & \cdots & \frac{\partial N_i}{\partial y} & \frac{\partial N_i}{\partial x} \\
\frac{\partial N_i}{\partial z} & 0 & \cdots & 0 & \frac{\partial N_i}{\partial z} \\
0 & \frac{\partial N_i}{\partial z} & \cdots & \frac{\partial N_i}{\partial z} & 0 \\
\frac{\partial N_i}{\partial x} & 0 & \cdots & 0 & \frac{\partial N_i}{\partial x}
\end{bmatrix}
\]

\[
B_T = \begin{bmatrix}
\frac{\partial N_i}{\partial x} & 0 & \frac{\partial N_i}{\partial y} \\
0 & \frac{\partial N_i}{\partial y} & \frac{\partial N_i}{\partial x} \\
\vdots & \vdots & \vdots \\
\frac{\partial N_i}{\partial d} & 0 & \frac{\partial N_i}{\partial d} \\
0 & \frac{\partial N_i}{\partial d} & \frac{\partial N_i}{\partial d}
\end{bmatrix}
\]

In the following, analytically integrated operators are described first and numerically integrated ones second.

**PDE Operators for Analytical Integration**

The available operators for analytical integration are essentially all those which are also available (and addressed via enumeration strings) in conjunction with the CSP_Algorithm:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Integration of PDE Term</th>
<th>Other Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integral_NT_lhsop_N_dV</td>
<td>$lhsop \nabla^2 N$</td>
<td>capacitance</td>
</tr>
<tr>
<td>Integral_dNT_op_dN_dV</td>
<td>$lhsop \nabla^2 N$</td>
<td>conductance</td>
</tr>
<tr>
<td>Integral_dNT_op_dN_NT_v_dNT_dV</td>
<td>$op \nabla^2 N + v \cdot \nabla N$</td>
<td>diffusion-advection</td>
</tr>
<tr>
<td>Integral_NT_op_N_dV</td>
<td>$N$</td>
<td>capacitance</td>
</tr>
<tr>
<td>Integral_N_op_dN_dV</td>
<td>e.g. $\rho g \nabla z$</td>
<td>buoyancy</td>
</tr>
</tbody>
</table>

Table 5: PDE operators for elements using analytical integration. The operators in red have not been tested rigorously yet.
The actual form of these operators and the contributions arising from them are for 2D triangular elements:

\[
\text{Integral\_NT\_lhsop\_N\_dV (conductance matrix)}
\]

\[
K^{(e)} = \iint_A dNT \begin{bmatrix} K_x & 0 \\ 0 & K_y \end{bmatrix} dN \, dx \, dy
\]

Analytical integration of this integral for 2D linear triangles yields

\[
\frac{K_x}{4A} \begin{bmatrix} b_i^2 & b_i b_j & b_i b_k \\ b_i b_j & b_j^2 & b_j b_k \\ b_i b_k & b_j b_k & b_k^2 \end{bmatrix} + \frac{K_y}{4A} \begin{bmatrix} c_i^2 & c_i c_j & c_i c_k \\ c_j c_i & c_j^2 & c_j c_k \\ c_k c_i & c_k c_j & c_k^2 \end{bmatrix}
\]

and for 3D linear tetrahedra (still wrong revise this wrong formula)

\[
\frac{K_x}{18V} \begin{bmatrix} b_i^2 & b_i b_j & b_i b_k \\ b_j b_i & b_j^2 & b_j b_k \\ b_k b_i & b_k b_j & b_k^2 \end{bmatrix} + \frac{K_y}{18V} \begin{bmatrix} c_i^2 & c_i c_j & c_i c_k \\ c_j c_i & c_j^2 & c_j c_k \\ c_k c_i & c_k c_j & c_k^2 \end{bmatrix} + \frac{K_z}{18V} \begin{bmatrix} d_i^2 & d_i d_j & d_i d_k \\ d_j d_i & d_j^2 & d_j d_k \\ d_k d_i & d_k d_j & d_k^2 \end{bmatrix}
\]

where \( B \) for 3D linear tetrahedra is

\[
B = \frac{1}{6V} \begin{bmatrix} b_1 & b_2 & b_3 & b_4 \\ c_1 & c_2 & c_3 & c_4 \\ d_1 & d_2 & d_3 & d_4 \end{bmatrix}
\]

Note that for linear finite elements, shape function derivatives are constants.

\[
\text{Integral\_dNT\_op\_dN\_dV (mass- or capacitance matrix)}
\]

consistent formulation

\[
C^{(e)} = \iint_A NT S^{(e)} N \, dx \, dy = \frac{S^{(e)} A}{12} \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}
\]

lumped formulation

\[
C^{(e)} = \iint_A NT S^{(e)} N \, dx \, dy = \frac{S^{(e)} A}{3} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}
\]
Integral \( dN/_{NT \cdot v \cdot dNT \cdot dV} \) (diffusion-advection matrix)

\[
D^{(e)} = \int_{A} dNT \begin{bmatrix} D_{xx} & D_{xy} \\ D_{yx} & D_{yy} \end{bmatrix} dN \begin{bmatrix} v_x \\ 0 \end{bmatrix} dN dxdy
\]

For 2D triangular elements, integration yields

\[
\frac{D_{xx}}{4A} \begin{bmatrix} b_i^2 & b_j b_k \\ b_j b_i & b_k b_j \\ b_k b_i & b_j b_k \end{bmatrix} + \frac{D_{yy}}{4A} \begin{bmatrix} c_i^2 & c_i c_j \\ c_j c_i & c_j c_k \\ c_k c_i & c_k c_j \end{bmatrix} \\
\frac{D_{xy}}{4A} \begin{bmatrix} b_j c_i & b_j c_k \\ b_k c_i & b_k c_j \\ b_i c_j & b_i c_k \end{bmatrix} + \frac{D_{yx}}{4A} \begin{bmatrix} c_j b_i & c_j b_k \\ c_k b_i & c_k b_j \\ c_i b_j & c_i b_k \end{bmatrix} \\
- \frac{v_x}{6} \begin{bmatrix} b_i & b_j & b_k \\ b_j & b_k & b_i \\ b_k & b_i & b_j \end{bmatrix} - \frac{v_y}{6} \begin{bmatrix} c_i & c_j & c_k \\ c_j & c_k & c_i \\ c_k & c_i & c_j \end{bmatrix}
\]

Additional decay and sorption terms also are based on “mass” matrices and therefore have the same consistent and lumped forms:

\[
RAD^{(e)} = \int_{A} NT(\lambda(\Phi + \rho_b K_d))N dxdy
\]

\[
A^{(e)} = \int_{A} NT(\rho_b K_d + \Phi)N dxdy
\]
### PDE Operators for Numerical Integration

<table>
<thead>
<tr>
<th>Operator</th>
<th>Integration of PDE Term</th>
<th>Other Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lefthand</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NumIntegral_NT_lhso_N_dV</td>
<td>$lhso{\nabla}^2 N$</td>
<td>capacitance</td>
</tr>
<tr>
<td>NumIntegral_dNT_op_dN_dV</td>
<td>$lhso{\nabla}^2 N$</td>
<td>conductance</td>
</tr>
<tr>
<td>NumIntegral_NT_op_dN_dV</td>
<td>e.g. $p g \nabla z$</td>
<td>buoyancy</td>
</tr>
<tr>
<td>NumIntegral_BT_D_B_dV</td>
<td>lhs force balance $\frac{\partial \sigma_{ij}}{\partial x_i} = F_j^b$</td>
<td>stiffness</td>
</tr>
<tr>
<td><strong>righthand</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NumIntegral_NT_op_dV</td>
<td>fluid body forces</td>
<td></td>
</tr>
<tr>
<td>NumIntegral_NT_op_N_dV</td>
<td>$N$</td>
<td>capacitance</td>
</tr>
<tr>
<td>NumIntegral_BT_op_dV</td>
<td>body forces, initial stresses</td>
<td></td>
</tr>
<tr>
<td>NumIntegral_BT_D_op_dV</td>
<td>volume strain</td>
<td></td>
</tr>
<tr>
<td>NumIntegral_PT_op_dS</td>
<td>surface stresses</td>
<td></td>
</tr>
<tr>
<td>PT_op</td>
<td>nodal forces</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: PDE operators for elements using numerical integration. The operators in red have not been tested rigorously yet.

In order to carry out the numerical integration interpolation function values and their derivatives must be computed at all integration points. Derivates need to be computed also in the assembly of mass matrices, since the Jacobian must be formed as its determinant post multiplies each integration point contribution.

**Lumping** of the mass matrix (if so desired for time-dependent calculations), is often implemented the crude way: The off-diagonal elements of the mass matrix are summed into its diagonal after forming the consistent form of the mass matrix:

$$b_{ij} = \sum_{j=1}^{N} b_{ij}$$

This adds computational effort rather than reducing it by a simpler form. An alternative approach circumventing this problem is discussed in Lapidus & Pinder (1999, p. 328-329). They advocate Simpson’s rule-based integration to arrive naturally at a diagonal form of the mass matrix. For the isoparametric elements in CSP lumping is implemented by multiplying the operand with the elements volume and dividing each of these nodal contributions by the number of nodes per element:

$$M^{(e)}_{lumped} = \frac{SV}{n} J \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
NumIntegral_dNT_op_dN_dV

This integral arises for elliptic partial differential equations, such as pressure diffusion or heat conduction. When the operand represents the hydraulic conductivity a conductance matrix results where $K$ values are interpolated to the integration points using $N$. Numerical integration amounts to the following summation over the contributions from each integration point:

$$
K^{(e)}_{nn} = \sum_{i=1}^{N_{\text{INT}}} \sum_{j=1}^{N_{\text{INT}}} W_i(\xi, \eta) W_j(\xi, \eta) \left( \frac{dN(\xi, \eta)}{dV} J(\xi, \eta) \right) 
\left\{ dNT(\xi, \eta) \left[ J(\xi, \eta) \right]^{-1} \right\} 
\left\{ \begin{array}{ccc}
K_x(\xi, \eta) & 0 \\
0 & K_y(\xi, \eta)
\end{array} \right\} 
\left\{ \begin{array}{c}
J(\xi, \eta) \\
1
\end{array} \right\}
$$

Integral_NT_op_N_dV

The capacitance or mass matrix computed by summing the contributions at the integration points has the form (3D):

$$
C^{(e)} = \iiint_{V} NT[S]N |J| d\xi d\eta d\zeta
$$

The storativity is inserted as a scalar operand.

NumIntegral_dNT_D_dN_NT3_v_dN

The diffusion-advection (dispersion) operator (not implemented yet) is based on the dispersivity tensor and the velocity vector. The accumulation of this term will lead to a non-symmetric solution matrix. If the matrix becomes strongly asymmetric the AMG solver can no longer be used and Gaussian Elimination may be the only method capable of inverting the solution matrix.

$$
D^{(e)} = \iiint_{V} dNT \left[ J^{-1} \right]^T \begin{bmatrix}
D_{xx} & D_{xy} & D_{xz} \\
D_{yx} & D_{yy} & D_{yz} \\
D_{zx} & D_{zy} & D_{zz}
\end{bmatrix} J^{-1} dN |J| d\xi d\eta d\zeta 
+ \iiint_{V} NT S |J| d\xi d\eta d\zeta
$$

Radioactive decay can be incorporated into this element matrix by adding a term in which the scalar operand is composed from the decay constant, the porosity, fluid density and the KD value.
and the (ad)sorption within the rock represented by the element is described by a similar NumIntegral_NT_op_N_dV term in which the operand contains the porosity, fluid density and the KD value

\[ A^{(e)} = \int \int \int_{V} NT[J \rho \Phi] d\xi d\eta d\zeta \]

Volume integrals for mechanical calculations differ from the previous ones in that the dependent variable is no longer a scalar but a vector or tensor variable. These operators describe the elements stiffness, body forces, and volume strains. Surface integrals describe boundary tractions, stresses etc.

**NumIntegral_BT_D_B_dV (stiffness)**

\[ K^{(e)} = \int \int \int_{V} BT[J^{-1}] [D][J^{-1}] B[J] d\xi d\eta d\zeta \]

**NumIntegral_BT_op_dV (initial stresses)**

\[ IS^{(e)} = \int \int \int_{V} BT[J^{-1}] \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{bmatrix} d\xi d\eta d\zeta \]

**NumIntegral_PT_op_dV (body forces)**

\[ F^{(e)} = \int \int \int_{V} NT[F_x \quad F_y \quad F_z] d\xi d\eta d\zeta \]

**NumIntegral_PT_op_dS (boundary tractions)**

\[ BS^{(e)} = \int \int_{S} NT[\sigma] dS \]

**NumIntegral_BT_D_op_dV (volume strain)**

\[ E^{(e)} = \int \int \int_{V} BT[J^{-1}] [D] d\xi d\eta d\zeta \]

**PT_op (nodal forces)**

\[ F^{(e)} = \begin{bmatrix} F_{x}^1 & F_{y}^1 & \cdots & F_{x}^n & F_{y}^n \end{bmatrix}^T \]
8. A Comprehensive Simulation Example: Carbonate Dissolution

This chapter is a step by step description how one can use CSP3D3.0 to carry out a reactive transport simulation with a feedback between carbonate dissolution and permeability, which (after thousands of iterations) will lead to wormholing. The source code for this example is provided in the examples directory on the distribution CD (example11.cpp). This description repeats concepts presented earlier in order to achieve a certain degree of independence and completeness of the chapter such that the reader is taken through all the necessary steps like in a cooking recipe.

0. Input Geometry

Build a geometrical model which will serve as an input geometry for the calculation:
2 possibilities exist:

1. Use NIH Image (current version >1.66) to create a rainbow color-coded representation of the model geometry and save it as TIFF file. Export an additional copy as text file from which a CSP mesher will automatically generate a regular triangular mesh with dimensions 2*pixel rows x 2*pixel columns. For simple runs limit the image dimensions to <50 x 100. The color values will be translated into permeabilities using the factors defined in “CSP_DataInterpreter.h.” (see table in Input/Output chapter).

2. Build a PSLG (planar straight line graph) of the geometry. It will consist of closed polylines with material points to which properties are attached. The PSLG is output to a dxf file and converted by the GeometryPreprocessor.exe to an input file for the Triangle mesher. This mesher produces a “*.poly” file with a constrained, conforming Delauney triangulation.

1. Physical Variables

The desired physical variables must be communicated to CSP. This is accomplished via a text file which initializes the CSP variable database (as managed by the Property class object). This file is named “CSP_carbonate_variables.txt”. Recall from Chapter 2 that notation in the file’s headline refers to symbols used for writing equations with the variable; units are SI units, and the index tells CSP what type of variable shall be constructed (scalar (index=1), a vector (2) or a tensor (3)). Min. max. defines a physically meaningful range for each new variable. All variables which appear in the reaction file must also be represented in the physical-variables database.

2. Initial Values and Boundary Conditions

For high-level interactions with CSP one most commonly uses the interfaces (functions) of the SuperGroup object. All of these are documented in the HTML documentation which can be browsed with any internet tool. For instance, one may set a fluid-volume source term to zero on all elements and assign Dirichlet boundary pressures on the left side of the geometry. This is done by the following code:

```c++
// SuperGroup interface, condition, your variable, its desired value
```
model.InputUniformValueWhere( PLAIN, "fluid volume source", 0.0 );

// model side, your variable, Dirichlet value-top, val-bottom
model.AssignBoundaryValues( LEFT, "fluid pressure", DIRICH, 1.0e+6, 1.0e+6 );

One must always initialize variables and specify at least one Dirichlet boundary value, before one carries out a computation.

The model geometry, as expressed in terms of spatial variations in the permeability, is perhaps the most important initial condition for the carbonate dissolution model. It ultimately determines where “wormholing” and “sealing” will occur by dissolution and precipitation, respectively. While one can try the carbonate-dissolution program with many different input geometries, the input file set “fractures.1” is recommended as a default. After starting the program, one enters this filename, when prompted for it by the program.

3. THE INTERRELATIONS

To compute the fluid-pressure distribution using the partial differential equation based on Darcy’s law, one must first calculate hydraulic conductivity values from the permeability values. This is done with an Interrelation object from a library of already implemented Interrelations. In summary, to use an existing Interrelation one must:

1. Include its definition into the program file before main():

```
#include "ConductivityFromPerm.h"
```

2. Create an instance of the Interrelation object:

```
// ConductivityFromPerm relation( p_ref, 1.6 );
```

Here, the viscosity of the fluid is supplied as second argument to the interrelation subclass constructor.

3. Execute the Interrelation by passing it to the SuperGroup object:

```
model.Pass( relation );
```

After this step, the variable ‘conductivity’ will have been assigned the desired value. Alternatively one may derive own Interrelations from the Interrelation base class. The code for this is as follows:

1. declare class name for the preprocessor

```
#ifndef __ConductivityFromPerm_h__
#define __ConductivityFromPerm_h__
```

2. include necessary declarations of base class and CSP variables

```
#include "CSP_definitions.h"
#include "Interrelation.h"
```

3. derive the subclass ‘ConductivityFromPerm’
class ConductivityFromPerm : public Interrelation {
  Operand&  k;
  Operand&  K;
  double    viscosity;
public:
  ConductivityFromPerm( const Property& p, double visc );
  ~ConductivityFromPerm() {}
  void Calculate();
};

3. construct the Interrelation object

    inline ConductivityFromPerm::ConductivityFromPerm( const Property& p,
    double          visc )
    : Interrelation(p), // initialize base class
      k( GlobalProperty("permeability") ), // register input property
      K( GlobalProperty("conductivity") ), // register result property
      viscosity(visc)                      // initialize variable
    {
      Name(" ConductivityFromPerm"); // name Interration for error reporting
      IOConditions( K, PLAIN, INIT_COND ); // define in and output flags
      ResultProperty("conductivity"); // define output property
    }

4. specify how the result property is calculated from the input properties. Here one
makes use of the overloaded capabilities of Operand objects:

    inline void ConductivityFromPerm::Calculate()
    {
      K = 1.0 / viscosity;
      K *= k;
    }

close the preprocessor 'ifdef' statement from above with #endif (omitting this may
send the compiler into an infinite loop if this file is used in several source code files).

4. THE CALCULATION

How to compute a steady-state fluid pressure distribution in a model, for spatial
variations in the permeability and Dirichlet (=fixed pressure) boundary conditions
was already discussed in Chapter 2 for the code example "mainUsersGuide.cpp".

In summary, one first constructs a SuperGroup representing the discretized version
of the physical model. Secondly, one specifies initial and boundary conditions. Thirdly,
one defines a computation and perhaps interrelations among the physical variables
used. Finally, the computation is carried out and the results are output for display.
The computed interstitial flow velocity field is used further by the TransportVisitor to
advect the concentrations of the aqueous solutes on a series of grids. After each
advection step, speciation calculations are carried out with the AlterationVisitor to
equilibrate the fluid with the rock.

5. SOLVING THE GLOBAL EQUATION

Global equations are specified in CSP using algorithm objects. Remember that two
classes are supported: The old CSP_Algorithm and the new Algorithm. In the
carbonate-dissolution program, a CSP_Algorithm subclass is applied:

    Velocity K-div2-P( p_ref );
K-div2-P.AddOperation( OPERAND__GRADIENT_N__GRADIENT_N, "conductivity", "fluid pressure","fluid pressure");
K-div2-P.AddOperation( OPERAND__N, "fluid volume source", "fluid pressure");
moab.Pass( K-div2-P );

The subclass constructor takes a reference to the physical-variable database as
argument such that it can obtain CSP_INDEX variables from the variable names. The
code above defines and applies the pressure-diffusion equation discussed in the
Algorithms library (Chapter 7).

6. THE GROUPS
In most geologically-oriented models one will discover the need to address specific
model regions selectively to obtain local results, assign regional properties, and/or
restrict the application of Interrelations and/or Algorithms to specific regions of the
model. For this purpose, group objects are defined by:

```cpp
supergroup.FormAndAddGroup("high perm", "permeability", 1.0e-12, 1.0e-5);
```

This line of code will include the model region where permeability ranges between
1.0e-12 and 1.0e-5 m², in a group with the name “high perm” (names are allowed to
contain white-space). The following statements assign new porosity values to the
elements in the new group and along the group boundary, respectively

```cpp
sg.ChangePropertyInGroupToWhere("high-perm", "porosity", 0.4, PLAIN);
sg.ChangePropertyInGroupToWhere("high-perm", "porosity", 0.4, BOUNDARY);
```

7. THE VISITORS
Remember that the term visitor comes from the object-oriented software design
pattern named Visitor. It facilitates a traversal of the SuperGroup hierarchy of
Groups, Elements, Nodes, and ConstraintPoints. A visitor in CSP can therefore visit
all Groups, Elements, ConstraintPoints, and Nodes of a SuperGroup object while
accessing these hierarchical objects via their public interfaces. The example at hand,
introduces the usage of the “TransportVisitor” and the “AlterationVisitor”.

8. FINITE DIFFERENCE GRIDS
Apart from unstructured-mesh finite-element capabilities, CSP also supports
structured, regular gridded data. Regular grids can be mapped to finite-elements and
vice versa, permitting mesh-assignment of geophysical data from regular-gridded
arrays and the output of mesh variables to structured-array visualization tools. If
animations shall be produced, pixel-type regular-gridded data can be output to JPEG
files or pixmaps, using the finite-element mesh interpolation functions. Regular grids
are represented in CSP by “FiniteDifferenceGrid” objects and can also be used for
finite-difference calculations, kernel type operations, image processing etc. To build
an instance of a finite-difference grid inside a CSP main() program one writes
FiniteDifferenceGrid mygrid;
mygrid.Initialize( x0, x1, y0, y1, RESOLUTION_x, RESOLUTION_y,
framewidth_x, framewidth_y );

An advantage of grids is that they require much less storage than a comparably refined finite-element mesh. For this reason the carbonate-dissolution example uses a FiniteDifferenceGrid for the advection of aqueous species. The fluid composition is stored by an array of grids on which the molalities of OH\(^-\), Cl\(^-\), CO\(_2\)\(_{aq}\), H\(^+\), HCO\(_3\)\(^-\) and Ca\(^{2+}\) are advected. H\(_2\)O is transported by the finite-element flow calculations, and calcite is stored as an element property. The TransportVisitor object advects the aqueous species on the grids over a distance that is given by the product of the local interstitial flow velocity (pore velocity) and the advection-time increment.

9. SPECIATION CALCULATIONS

In the carbonate dissolution example, reactions are treated as rate laws using a forward rate and the inverse of the equilibrium constant. Via the law of mass action, the typical equilibrium expressions are replaced by ordinary differential equations (ODE's) which are written in the form:

\[ A + B = C \]

is replaced by

\[ \frac{dA}{dt} = -kf [A][B] + \frac{kb [C]}{K} \]

where \( A \) and \( B \) are the reactants, \( C \) is the reaction product, \( kf \) is the forward (far from equilibrium reaction rate and \( K \) is the equilibrium constant. Because equilibrium is defined such that

\[ kf [A][B] = kb [C] \quad \text{and} \quad K = \frac{[C]}{[A][B]} \]

it follows that

\[ K = \frac{kf}{kb} \]

at equilibrium. Therefore, one can express \( kb \) in terms of \( K \) as is done in the equation above. This kinetic approach is outlined in detail in Carl Steefel's article in Reviews in Mineralogy 34, p. 93ff. For the carbonate-dissolution example, the system of ODE's is:

\[
\begin{align*}
\frac{dOH_{1m}}{dt} &= -KF H_{1p} OH_{1m} + KB1 H2O_{aq} \\
\frac{dCO2_{aq}}{dt} &= -KF CO2_{aq} H2O_{aq} + KB2 HCO3_{1m} H_{1p} \\
\frac{dCC}{dt} &= -KF CC H_{1p} + KB3 HCO3_{1m} Ca_{2p} \\
\frac{dH_{1p}}{dt} &= -KF H_{1p} OH_{1m} + KB1 H2O_{aq} + \\
&\quad + KF CO2_{aq} H2O_{aq} + -KB2 HCO3_{1m} H_{1p} + \\
&\quad + -KF CC H_{1p} + KB3 HCO3_{1m} Ca_{2p} \\
\frac{dH2O_{aq}}{dt} &= KF H_{1p} OH_{1m} + -KB1 H2O_{aq} + -KF CO2_{aq} H2O_{aq} + \\
&\quad + KB2 HCO3_{1m} H_{1p} \\
\frac{dHCO3_{1m}}{dt} &= KF CO2_{aq} H2O_{aq} + -KB2 HCO3_{1m} H_{1p} + \\
&\quad + KB2 HCO3_{1m} Ca_{2p} + KF CC H_{1p} + -KB3 HCO3_{1m} Ca_{2p} \\
\frac{dCa_{2p}}{dt} &= KF CC H_{1p} + -KB3 HCO3_{1m} Ca_{2p}
\end{align*}
\]
where $KBx$ is the inverse of $K$. The products in the equations which produce the left-hand species always have a rate coefficient with a negative sign. Also, as follows from the law of mass action, stoichiometric factors in the conventional equations like $2A+B=C$ translate into dependent variable exponents ($2A \rightarrow A^2$). Stoichiometric factors must also be taken into account in the rate coefficients. Thus, if a reaction produces $2A$, this implies for the total rate at which $A$ is produced or consumed that this reaction contributes twice the amount of $A$. Consequently, both, the forward and the backward rate must be multiplied by 2 in any expression involving $A$. This shall be illustrated using a wollastonite producing reaction:

$$WOL + 2H_{lp} = Ca_{2p} + \text{SiO}_2_{aq} + H_2O \quad (1/\text{equilibrium constant} = KB1)$$

The contribution of this reaction to an ODE which describes the $H^+$ concentration is:

$$\frac{d}{dt} H_{lp} = 2 \times -KF \times WOL \times H_{lp}^2 + 2 \times KB1 \times Ca_{2p} \times \text{SiO}_2_{aq} \times H_2O_{aq} + \ldots$$

since, during every reactive exchange, 2 moles of $H^+$ are produced or consumed. Stoichiometric factors must be multiplied into forward and backward rates, since the equation parser in the ODE solver cannot deal with 2 coefficients in a single product term. Alternatively, the number 2 is defined as an additional independent variable. This, however, is more costly than just generating

$$KB1 \times 2 = 2 \times KB1$$

The reaction ODEs are integrated over the reaction time interval with a Burlisch-Stoer solution scheme for stiff ODEs (Bader and Deuflhard 1986). This scheme is implemented as a “StiffODESolver” object which has a method that reads the example text file “calciteOH_25.txt” with the specification of the ODEs as above. The equilibrium constants and forward reaction rates for the corresponding reactions at 25°C are also defined in this file.

### 10. Initial Conditions

Before the simulation of reactive transport is started, the fluid on the grid is equilibrated with the rock via a visitation of each finite element by the alteration visitor. Before this visit, the grids are initialized to zero and the equilibration time given to the visitor is set to a very large value (in seconds) to ascertain that the fluid fully equilibrates with the rock. The speciation calculations are carried out element by element as the AlterationVisitor visits them. The code which specifies how this is done is contained in the method,

```cpp
virtual void AlterationVisitor::Visit( Element* e );
```

a purely virtual function of the base class CSP_FEM_Visitor. The Visit() method does the following:

1. Read the speciation of the fluid from the last timestep from the Node and assign it as initial condition to the ODE solver object.
2. Obtain the new speciation in the considered rock volume for 1 kg of fluid.

```cpp
// maxtime is just the time needed for full equilibration (no other meaning)
maxtime init.t-step save-steps desired tolerance
solver.Solve( time_increment, 1.0e-17, 0, fmol*tolerance );
```

3. Calculate how much calcite was dissolved from the difference of the Ca\(^{2+}\) molality between the old and the new speciation in 1 kg of fluid. Convert this amount to a unit volume of the fluid-rock assembly. Store the result back to the node.

```cpp
// dissolved calcite is derived from change in Ca+2 concentration
// result is converted into kg
n->Store( pmem, cc_key.index,
   (sc=BASIS[2]+(BASIS[6]-solver(7))*100.08*1.0e-3*fmass) );
```

### 11. The Overall Model

In the main program, reactive transport is simulated via a sequential iteration approach (SIA) as discussed by Yeh and Tripathi (1989, 1991) and summarized in Steefel (1996, see reference above). This technique means that transport and speciation steps are decoupled. An additional complexity arises as calcite-dissolution increases porosity which, in turn, increases the permeability. This is implemented via two Interrelation subclasses 'PorosityChange' and 'PermeabilityFromPorosity'. They are passed to the SuperGroup after each transport- and equilibration- pair of steps. Subsequently, the Darcy flow velocity distribution at \( t+1 \) is recomputed with the 'Velocity' Algorithm described in Chapter 7.

```cpp
K-div2-P.Reset();
K-div2-P.AddOperation( OPERAND__GRADIENT_N__GRADIENT_N,
   "conductivity","fluid pressure","fluid pressure");
K-div2-P.AddOperation( LUMPED__OPERAND__N__N,
   "storativity","fluid pressure","fluid pressure");
K-div2-P.AddOperation( LUMPED__OPERAND__N,
   "storativity","fluid pressure");
K-div2-P.AddOperation( LUMPED__OPERAND__N2,
   "fluid volume source","fluid pressure");
K-div2-P.SetTimeIncrement( time_increment );
```

Note that in order to solve the transient PDE, the CSP_Algorithm module is first Reset() and then newly defined. Two new terms enter the definition: A left-hand term LUMPED__OPERAND__N__N and the right-hand term LUMPED__OPERAND__N. They describe the reluctance of the pressure-distribution to change due to the storage capacity of the system. The differential operator for the fluid volume source term is different, also, as was explained for the PDE of transient fluid flow in Chapters 2 and 7. The transient flow equation is solved using fully implicit finite-difference time-stepping with a time-step size input by.

```cpp
K-div2-P.SetTimeIncrement( time_increment );
```

The solution alters the previous fluid-velocity distribution to the model state at \( t = \) previous time + time_increment.
To drive the chemical reactions, after each advection step, an acid but charge-balanced fluid composition is specified on the left margin of the grids. This acid fluid is advected to the right into the cross-sectional model where it dissolves carbonate until it is neutralized by the rate-controlled reaction with calcite.

The model is run while the H\(^+\) concentration on the first of the grids and the “porosity” are output from time to time to DXF and VTK files, respectively, to monitor the progress of carbonate dissolution.
9. Auxiliary Programs

Mesh Pre-processors

The GeometryPreprocessor.exe was already described in the chapter on data input and output. It constitutes the “glue” needed to use the Triangle mesher and other programs.

CSP also comes with a meshing module named Triangulator for the creation of two-dimensional triangulations from regular-gridded geometry representations. This module is used internally by the SuperGroup. The source code “example1.cpp” illustrates how to use the Triangulator to generate a mesh from which a SuperGroup is build.

The MeshInterface class provides further capabilities needed to pre-process GoCad and NASTRAN mesh files for the use in CSP. One of the tasks of these pre-processor methods is the identification of the neighbor elements of each finite-element. This type of data is rarely provided by third-party meshers. The CSP_VSetConverter module takes triangular and tetrahedral input meshes and inserts midside and barycentric nodes such that these meshes can be used with the higher-order finite elements from the Finite-Element Library. It also replaces those corner elements in 2D meshes which have sides on two model boundaries simultaneously. Such elements are unsuitable for computations since they would have zero degrees of freedom if boundary conditions were assigned to the two model boundaries.

Geometry Utilities

A set of topology manipulation classes is integrated into the CSP basic library. They comprise Point, Edge, Polygon, and Triangle classes for two-dimensional operations, and Point3D, Edge3D, and Triangle3D classes for finding line triangle intersections etc. in three dimensional space. Since these classes were based on a book on computational geometry by M. J. Lazlo, their names are prefaced with MJL_. The main reason for the integration of these classes in CSP, is the pre-processing of mesh input files containing parallel straight line graphs (PSLG’s) of models for the program Triangle.

It is an auxiliary program by SKM which reads DXF files prepared according to the specifications in Chapter 4. The interpreted files are converted into input files for the program Triangle. This program by J. Shewchuk performs constrained-conforming Delauney triangulations of excellent quality. The GeometryPreprocessor also converts raw triangulated surfaces from Rhino (R. McNeel & Associates) into GoCad TSurf format. This capability permits to circumvent GoCads or TSurf’s cumbersome geometry generation routines.
Appendix A: THE DISTRIBUTION

CSP3D comes as a set of header and binary library files for the specific platforms and compilers with which the simulations shall be carried out. The files are stored in subdirectories with names corresponding to the libraries: main-library/, interrelations/, algorithms/, visitors/, finite-elements/, finite-volumes/, pde-operators/, and distributed-examples/. Unix-linux distributions are based on the GNU C++ compiler (version 2.95.2 or later) and have bin/ and lib/ directories in which executables, input, and variables files, and the binary libraries themselves are stored, respectively. Makefiles are provided for the example programs, and, on request, CodeWarrior Pro 6.0 (or later) project files for the Macintosh, PC, and the Linux PC, can be obtained.

The class objects which constitute the current distribution of CSP are listed in alphabetical order in the tables below corresponding to the different CSP libraries:

- CSP3D Library
- CSP3D Interrelations
- CSP3D Algorithms
- CSP3D Visitors
- CSP3D Finite Elements
- CSP3D Finite Volumes
- CSP3D PDE Operators
- CSP3D Distributed Examples

The Meschach, Meschach++, AMG, HDF, JPEG, and F2C libraries which are used by CSP are documented elsewhere.

<table>
<thead>
<tr>
<th>CSP3D Base Library</th>
<th>Description / Role</th>
<th>Support</th>
<th>Doc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>solution of PDE's using MathOps.</td>
<td>yes</td>
<td>html</td>
</tr>
<tr>
<td>Array</td>
<td>basic array class with operators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BE_Time</td>
<td>measurement of runtime / processes</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>BinaryFileHandler</td>
<td>template: IO of any class objects</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>BoundaryElementPair2D</td>
<td>computes fluxes between elements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BoundarySegment</td>
<td>representation of line segments</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ColorPalette</td>
<td>conversion of double to RGB or HSV rainbow or grayscale (modified from Per Kistler's version)</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>ConstraintPoint</td>
<td>point on finite-element edge</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>CSP_Algorithm</td>
<td>solution of global equations</td>
<td></td>
<td>UG</td>
</tr>
<tr>
<td>CSP_DataContainer</td>
<td>storage of CSP_FEM_Data data sets</td>
<td>yes</td>
<td>html</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------------------</td>
<td>-----</td>
<td>------</td>
</tr>
<tr>
<td>CSP_Exception</td>
<td>information container for exceptions</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>CSP_FEM_Data</td>
<td>vector of scalar, vector, tensor variables</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>CSP_FEM_Visitor</td>
<td>base class for visitor subclasses</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>CSP_FileHandler</td>
<td>text I/O for run documentation</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>CSP_Index</td>
<td>class equivalent of CSP_INDEX struct</td>
<td>yes</td>
<td>html UG</td>
</tr>
<tr>
<td>CSP_IndexMapper</td>
<td>re-indexing of ID-key maps</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>CSP_MemoryManager</td>
<td>property value storage</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>CSP_MeshManager</td>
<td>mesh connectivity</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>CSP_Operand</td>
<td>model-wide calculations</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>CSP_Standard_IO_Handler</td>
<td>input queries etc.</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>CSP_StatisticalAnalyzer</td>
<td>property value histograms</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>CSP_String</td>
<td>operations with text strings</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>CSP_Triangle</td>
<td>determination of 2D model boundaries</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>CSP_VData</td>
<td>mesh-connectivity-data container</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>CSP_VSet</td>
<td>SuperGroup model container, persistence</td>
<td>yes</td>
<td>html DG</td>
</tr>
<tr>
<td>CSP_VTK_Interface</td>
<td>output of results for Vis. Tool Kit</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>CSP_VSetConverter</td>
<td>linear to quadratic mesh conversions</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>CuthillMcKhee</td>
<td>node no. matrix bandwidth reduction</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>DataInterpreter</td>
<td>1...256 integer to permeability converter</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>Element</td>
<td>connectivity and coordinates of finite element</td>
<td>yes</td>
<td>DG</td>
</tr>
<tr>
<td>ElementGrid</td>
<td>vector of grid cells overlapped by element</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>FemToGridVisitor</td>
<td>unstructured to structured mapping</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>FemFromGridVisitor</td>
<td>structured to unstructured mapping</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>FiniteDifferenceGrid</td>
<td>interpolation, advection, JPEG</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>FiniteElement</td>
<td>base class of specific finite-element type</td>
<td>yes</td>
<td>DG</td>
</tr>
<tr>
<td>GocadHeader</td>
<td>file header for GoCad files</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>GocadPropertyClassHeader</td>
<td>GoCad property-type descriptor</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>GridPoint</td>
<td>storage array for xy-point data on several but corresponding grids</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Group</td>
<td>sub-region of model</td>
<td>yes</td>
<td>DG</td>
</tr>
<tr>
<td>HashKey</td>
<td>element neighbor identification</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>IndexedMapper</td>
<td>renumber dis-contiguous lists</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Interrelation</td>
<td>base class for interrelation objects</td>
<td>yes</td>
<td>html</td>
</tr>
<tr>
<td>KD_BinaryTree</td>
<td>multidimensional search tree</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>MathOperatorLHS</td>
<td>base class for PDE operators for left-hand side of matrix equation</td>
<td>yes</td>
<td>DG</td>
</tr>
<tr>
<td>MathOperatorRHS</td>
<td>base class for PDE operators for right-hand side of matrix equation</td>
<td>yes</td>
<td>DG</td>
</tr>
<tr>
<td>MBST_Node</td>
<td>disk in KD_BinaryTree object</td>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>
MeshInterface | Triangle, GoCad, ICEM interface | yes | UG
---|---|---|---
MJL_Point | topology: 2D point object | Lasl.
MJL_Edge | topology: 2D straight edge | Lasl.
MJL_Polygon | topology: 2D straight-edge polygon | Lasl.
MJL_Point3D | topology: 3D point object | Lasl.
MJL_Edge3D | topology: 3D straight edge | Lasl.
MJL_Triangle3D | topology: 3D triangle | Lasl.
Node | nodal point in finite element | yes | DG
ODE_StiffSolver | stiff ordinary differential equation solver | yes | html
Operand | variable handle in computations | yes | html
Operation | composition of equations inside of CSP_Algorithm | yes
Output | output of selected timesteps | yes
Parameter | property specifications record used by PropertyDatabase | yes
PropertyDatabase | variable management in a model | yes | html
ScalarVariable | double plus DATA_STYLE | yes | UG
SkmErrorHandler | global platform-independent error-handler | yes | html
skmReadWrite | binary File I/O functions | yes
SKM_DataObject | data storage for KD_BinaryTree | yes
Solver | sparse-matrix solver, AMG, conjugate gradient, Gaussian elimination etc. | yes | UG, DG, html
SumOfProducts | algebraic equation handler for ODE solver | yes | DG
SumOfProductsWith-Exponents | algebraic equation with exponents handler for ODE solver | yes | DG
SuperGroup | model representation of real-world system | yes | html
TensorVariable | space-dimensions squared double matrix with flags | yes | UG
Triangle | finite-element forms of PDE operators for 2D CSP_Algorithm | yes | UG
Triangulator | 2D mesh generator for pixel input | yes | UG
VectorVariable | space-dimension-sized double vector with flags | yes | UG

Table A.1: Class objects contained in the CSP3D base library. Doc. = documentation in html (HTML) format, in the User’s Guide (UG, this document), or the Developer’s Guide (DG).

<table>
<thead>
<tr>
<th>Class</th>
<th>Description / Role</th>
<th>Support</th>
<th>Doc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbsoluteFluidDensity_HILL</td>
<td>nodal fluid density as calculated from Hill EOS</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>AvgFluidDensity_HILL</td>
<td>element average of nodal fluid density</td>
<td>yes</td>
<td>UG</td>
</tr>
</tbody>
</table>
Appendix A: CSP3D.0 Distribution

### ConductivityFromPerm
- **Description**: hydraulic conductivity from permeability and fixed fluid viscosity
- **Support**: yes
- **Doc.**: UG

### Conductivity
- **Description**: hydraulic conductivity from permeability and fluid viscosity
- **Support**: yes
- **Doc.**: UG

### Dilatancy
- **Description**: element volume change
- **Support**: yes
- **Doc.**: UG

### EnthalpySource
- **Description**: heat source /sink due to $H$ change
- **Support**: no
- **Doc.**: no

### ExtractTensorVariable-Component
- **Description**: Assign value (i,j) of tensor to scalar
- **Support**: yes
- **Doc.**: UG

### ExtractVectorVariableLength
- **Description**: compute length & assign to scalar
- **Support**: yes
- **Doc.**: UG

### ExtractVectorVariable-Component
- **Description**: Assign value (i) of vector to scalar
- **Support**: yes
- **Doc.**: UG

### FluidExpansionSource
- **Description**: source term due to PT change
- **Support**: no
- **Doc.**: no

### FluidHeatCapacity
- **Description**: fluid heat capacity as calculated from Hill EOS
- **Support**: yes
- **Doc.**: UG

### FluidViscosity_HILL
- **Description**: IAPS style fluid viscosity from Hill EOS fluid density
- **Support**: yes
- **Doc.**: UG

### HeatTransfer
- **Description**: compute heat advection velocity from Darcy velocity and thermal properties of fluid and rock
- **Support**: yes
- **Doc.**: UG

### Hill.c
- **Support**: no
- **Doc.**: no

### HILL_H2O_Density
- **Description**: fluid density from Hill EOS
- **Support**: no
- **Doc.**: no

### HydraulicHead
- **Description**: from piezometric + elevation head
- **Support**: yes
- **Doc.**: UG

### IAPS_H2O_Viscosity
- **Description**: IAPS formula for fluid viscosity
- **Support**: yes
- **Doc.**: UG

### ModifyPropertyWhere
- **Description**: selectively change property values
- **Support**: yes
- **Doc.**: UG

### NodeToElementProperty
- **Description**: element average of node property
- **Support**: yes
- **Doc.**: UG

### NodalScalarVolumeFlux
- **Description**: nodal velocity -> scalar flux
dissolution or precipitation due to gradient reaction
- **Support**: yes
- **Doc.**: UG

### QuartzPrecipitation
- **Description**: Craig Manning’s formula
- **Support**: yes
- **Doc.**: UG

### QuartzSolubility
- **Description**: Fournier & Potter, Crerar
- **Support**: yes
- **Doc.**: UG

### RelativeFluidDensity_HILL
- **Description**: $\rho/\rho_0$ as calculated from Hill EOS
- **Support**: yes
- **Doc.**: UG

### ScalarMassFlux
- **Description**: $\nabla P \cdot \mathbf{K}$
- **Support**: yes
- **Doc.**: UG

### ScalarVolumeFlux
- **Description**: advection step-size determination
- **Support**: yes
- **Doc.**: UG

### TransportStepSize
- **Description**: advection step-size determination
- **Support**: yes
- **Doc.**: UG

#### Table A.2: Class objects contained in the CSP3D Interrelations library. Doc. = documentation in html (HTML) format, in the User’s Guide (UG, this document), or the Developer’s Guide (DG).

#### CSP3D Algorithms

<table>
<thead>
<tr>
<th>Class</th>
<th>Description / Role</th>
<th>Support</th>
<th>Doc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GravityCorrectedVelocity</td>
<td>buoyancy-driven fluid flow</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>InterstitialVelocity</td>
<td>pore-scale flow velocity</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>Velocity</td>
<td>Darcy velocity</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>LinearElasticity</td>
<td>linear-elastic deformation</td>
<td></td>
<td>UG</td>
</tr>
</tbody>
</table>

#### Table A.3: Class objects contained in the CSP3D Algorithms library. Doc. = documentation in html (HTML) format, in the User’s Guide (UG, this document), or the Developer’s Guide (DG).
### CSP3D Visitors

<table>
<thead>
<tr>
<th>Class</th>
<th>Description / Role</th>
<th>Support</th>
<th>Doc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlterationVisitor</td>
<td>element-by-element carbonate dissolution / precipitation reaction prototype</td>
<td></td>
<td>UG</td>
</tr>
<tr>
<td>BoundaryTransferVisitor</td>
<td>fluxes across model and group boundaries</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>CSP_ID_Visitor</td>
<td>example, collects ID numbers into IVec</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>FiniteVolumeAdvection-Visi</td>
<td>FV first order volume-conservative transport scheme</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>HeatTransportVisitor</td>
<td>advection of properties on internal grid</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>HydroFractureVisitor</td>
<td>overpressure dissipation via conductivity enhancement</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>InterfaceVisitor</td>
<td>fluxes between adjacent groups</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>TransportVisitor</td>
<td>property advection on several grids</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>VelocityAndVolumeFlux-Visi</td>
<td>group-restricted post-processing and nodal extrapolation</td>
<td>yes</td>
<td>UG</td>
</tr>
</tbody>
</table>

Table A.4: Class objects contained in the CSP3D Visitors library. Doc. = documentation in html (HTML) format, in the User’s Guide (UG, this document), or the Developer’s Guide (DG).

### CSP3D Finite Elements

<table>
<thead>
<tr>
<th>Class</th>
<th>Description / Role</th>
<th>Support</th>
<th>Doc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinearTriangle</td>
<td>isoparametric triangular element with linear interpolation functions</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>LinearTetrahedron</td>
<td>isoparametric tetrahedral element with linear interpolation functions</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>QuadraticTriangle</td>
<td>6-noded isoparametric element</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>BarycentricQuadraticTri</td>
<td>7-noded isoparametric element</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>QuadraticTetrahedron</td>
<td>10-noded isoparametric element</td>
<td>yes</td>
<td>UG</td>
</tr>
</tbody>
</table>

Table A.5: Class objects contained in the CSP3D finite-element library. Doc. = documentation in html (HTML) format, in the User’s Guide (UG, this document), or the Developer’s Guide (DG).

### CSP3D Finite Volumes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description / Role</th>
<th>Support</th>
<th>Doc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FiniteVolumeVisitor</td>
<td>base class FV mesh visitors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FiniteVolumeAdvection-Visitor</td>
<td>2D volume-conservative FV transport scheme</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>FiniteVolumeManager</td>
<td>2D finite-volume mesh</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>FiniteVolume</td>
<td>mesh-representation of FV’s</td>
<td>yes</td>
<td>UG</td>
</tr>
</tbody>
</table>

Table A.6: Class objects contained in the CSP3D finite-volume library. Doc. = documentation in html (HTML) format, in the User’s Guide (UG, this document), or the Developer’s Guide (DG).
### CSP3D PDE Operators

<table>
<thead>
<tr>
<th>Class</th>
<th>Description / Role</th>
<th>Support</th>
<th>Doc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integral_NT_op_N_dV</td>
<td>e.g., capacitance right vector</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>Integral_NT_lhsop_N_dV</td>
<td>e.g., capacitance left matrix</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>Integral_dNT_op_dN_dV</td>
<td>e.g., conductance left matrix</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>Integral_dNT_op_dN_v_dNT</td>
<td>diffusion-advection left matrix</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>Integral_N_op_dN_dV</td>
<td>e.g., buoyancy forces</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NumIntegral_BT_D_B_dV</td>
<td>e.g., stiffness left matrix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NumIntegral_BT_D_op_dV</td>
<td>e.g., volume strain right vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NumIntegral_dNT_op_dN_dV</td>
<td>e.g., conductance left matrix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NumIntegral_NT_lhsop_N_dV</td>
<td>e.g., capacitance left matrix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NumIntegral_PT_op_dS</td>
<td>e.g., surface stresses (tractions)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NumIntegral_PT_op_dV</td>
<td>e.g., initial stresses</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NumIntegral_op_PT_P_dV</td>
<td>mass, right vec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NumIntegral_PT_op_dV</td>
<td>body forces, right vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NumIntegral_NT_op_dN_dV</td>
<td>capacitance right vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PT_op</td>
<td>nodal point forces, right vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>StressesAndStrain</td>
<td>postprocessing, nodal extrap.</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>VelocityAndVolumeFlux</td>
<td>postprocessing, nodal extrap.</td>
<td>yes</td>
<td>UG</td>
</tr>
</tbody>
</table>

Table A.7: Class objects contained in the CSP3D PDE Operator library. Doc. = documentation in html (HTML) format, in the User’s Guide (UG, this document), or the Developer’s Guide (DG).

### CSP3D Distributed Examples

<table>
<thead>
<tr>
<th>Function</th>
<th>Description / Role</th>
<th>Support</th>
<th>Doc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>examples_main.cpp</td>
<td><code>main()</code> program with menu to choose examples from</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>example_UsersGuide.cpp</td>
<td>2D pressure-driven flow, K variations, Group computation</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>example1.cpp</td>
<td>Triangulator-based meshing, 2D pressure-driven flow, model to binary file and back, CSP_Operands</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>example2.cpp</td>
<td>2D solute transport in a constant velocity field</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>example3.cpp</td>
<td>2D fluid pressure-diffusion during well testing linear-elastic displacements in an inhomogeneous solid due to boundary displacements</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>example4.cpp</td>
<td>3D pressure-driven fluid flow using an Algorithm, output of Groups</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>example5.cpp</td>
<td>integration of ordinary differential equations describing chemical reactions</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>example6.cpp</td>
<td>topography-driven fluid flow</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>example8.cpp</td>
<td>transient thermal convection</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>example9.cpp</td>
<td>3D elastic deformation / stress fields computed with quadratic elements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>example11.cpp</td>
<td>2D reactive transport: rate-controlled carbonate dissolution or precipitation with K feedback</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>example12.cpp</td>
<td>2D transport with the finite-volume method</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>example13.cpp</td>
<td>fluid-pressure equilibration using Algorithm and PDE operators</td>
<td>yes</td>
<td>UG</td>
</tr>
<tr>
<td>example14.cpp</td>
<td>3D hybride-element volume &amp; surface flow model</td>
<td>yes</td>
<td>UG</td>
</tr>
</tbody>
</table>

Table A.6: Functions contained in the CSP3D Distributed Examples library. Doc. = documentation in html (HTML) format, in the User’s Guide (UG, this document), or the Developer’s Guide (DG).
**B: INSTALLATION**

The CSP3D libraries require the third-party support libraries f2c (g2c), JPEG, Meschach, Meschach++, and AMG. The optional HDF C-library adds functionality for writing and reading platform-independent binary data files, but it is no longer actively supported since NCSA is replacing it with a new C++ library. The AMG and Meschach++ libraries were written by the co-author Stephen G. Roberts. CSP3D is divided into the basic CSP3D library and further derived libraries which provide high-level functionality, see Fig. A1.

![Fig. A1: Library packages of the CSP3D distribution.](image)

The complete CSP3D distribution therefore contains the additional components:

<table>
<thead>
<tr>
<th>Library Name</th>
<th>ftp site</th>
</tr>
</thead>
<tbody>
<tr>
<td>f2C</td>
<td><a href="http://www.netlib.no/netlib/f2c/">http://www.netlib.no/netlib/f2c/</a></td>
</tr>
<tr>
<td>Meschach++</td>
<td>in this form only from the authors</td>
</tr>
<tr>
<td>AMG</td>
<td><a href="http://www.netlib.no/netlib/AMG">http://www.netlib.no/netlib/AMG</a></td>
</tr>
<tr>
<td>HDF</td>
<td><a href="http://hdf.ncsa.uiuc.edu/">http://hdf.ncsa.uiuc.edu/</a></td>
</tr>
<tr>
<td>JPEG</td>
<td></td>
</tr>
<tr>
<td>VTK</td>
<td><a href="http://kitware.com">http://kitware.com</a></td>
</tr>
<tr>
<td>Blitz++</td>
<td><a href="http://oonumerics.com/blitz">http://oonumerics.com/blitz</a></td>
</tr>
</tbody>
</table>

In order to adapt to the particularities of the various platform a bit of conditional compilation is used in CSP3D3.0. The user must therefore uncomment a `#define`
statement in the header file “CSP_definitions.h” to unveil the name of the operating system before compiling CSP-library based code on the platform of choice.

The user may also need to define which version of the STL library shall be used. There are different implementations which add multithread safety and specific exception handling capabilities to Stepanov’s original STL resource. Again, different versions can be invoked on different platforms by un-/commenting #define statements in the “CSP_definitions.h” header file.

CSP Projects in CodeWarrior

CodeWarrior is a powerful C++ code development environment with IDE, debugger, class browser etc. It exists on the Mac, the PC, and on Linux operating systems. SKM recommends it for the development of CSP applications, especially now after version 6.0 is out and actually compiles Todd Verhuizen’s Blitz++ library.

To use the CSP library in an original C++ Project in CodeWarrior (with exact references to CW Pro 6.0), a series of configuration steps must be performed. In the following description of these steps a right arrow (→) indicates menu and submenu choices, for instance:

File → Save → TIFF

means go to the “File” menu, choose the submenu “Save”, and select the option which will save the data in a tagged image file.

1. Start Code Warrior and choose: File → New...

A box will appear prompting the user to select the kind of supplementary files he or she wishes to include into a new project. Click into the list of options for the current operating system, e.g.:

Win32 C++ Stationary

Define a project name and enter the directory in which the new project shall be created. Then choose after clicking O.K. from the new pop-up box

Win32 Console App -> C++ Console App (on PPC, or NT)

as project subcategory. Make sure that the project will be created in an intended subdirectory. Importantly, this should not be the default bin/ directory of CodeWarrior since files stored in this location are likely to be deleted when Code Warrior is upgraded.

A new project window will now be created, containing the three folders: Sources, MSL ANSI Libraries, and operating-system dependent Win32 SDK libraries. Before adding source code and libraries to the new project, the project must be configured:
2. Choose: Edit → C++ App Console Settings...
and a window with a topic panel will appear. Several of the listed panels need to be
opened to set options:

Target → Target Settings: Enter the intended name of your project configuration.

Target → Access Paths: Set paths to the subdirectories in which the header files of the
desired objects live in. The typical set of paths for a CSP project might look like this:

```
Current-Work/Programming/Libraries/CSP3D
Current-Work/Programming/Libraries/CSP3D-Algorithms
Current-Work/Programming/Libraries/CSP3D-Interrelations
Current-Work/Programming/Libraries/CSP3D-Visitors
Current-Work/Programming/Libraries/CSP3D-Finite-Elements
Current-Work/Programming/Libraries/f2c/include
Current-Work/Programming/Libraries/Meschach
Current-Work/Programming/Libraries/Meschpp
Current-Work/Programming/Libraries/AMG
Current-Work/Programming/Libraries/JPEG
Current-Work/Programming/Libraries/HDF
```

Its exact form depends on the directory structure of the machine used and the
operating system. In the CSP3D Windows NT distribution, all CSP header files are
stored in the single directory:

```
CSP3D-headers/
```

Add paths by using the Add... button with the “User Paths” window selected. Do no
alter the “System Paths”. IMPORTANT is that the last folder in each path must
contain the header file which declares the desired object. It does not suffice to specify
the next higher folder. To find the location of a file use the operating system’s “Find”
tools with its file name as input.

```
Target → "..." Target: Enter the name for the executable program which shall be
produced in the “File Name” box, and, on the Mac, give the program enough heap
memory to operate. The more highly resolved the mesh is, the more memory will be
required.
```

```
Language Settings → C/C++ Language: Check the following boxes in addition to those
which are already selected:

11. Activate C++ Compiler
12. ANSI Strict
```

The “Code Generation → "..." Processor” menu allows the user to optimize the code
for the processor which is being used. Different optimizations may be chosen. The
“Code Generation → Global Optimizations” panel offers additional options. The main
choices are executable size versus execution speed and a series of optimization levels.
Beware of levels beyond 2, if the code still has to be debugged. If a high level of
optimization is chosen variables may be used for a number of purposes and their
values will seem to change erratically. Also Code Warrior may crash trying to find
the best options.
Linker → “...” Linker: Check the “Generate SYM File” option to be able to view source code in the debugger window and to set breakpoints in it.

Save the configuration and close the “C++ App Console Settings” panel.

3. Now, libraries and source code files can be added to the project. Make sure that there is always just a single main() function in the code, otherwise the linker will not know which executable to build after the compilation. Before adding source code files delete the default folder “Sources” by selecting it in the project window and choosing:

Project → Remove Selected Items...

To add libraries choose:

Project → Add Files... → Add

and pick the desired files which will have the extension “.lib”. Under Windows NT, the file type must be set to library files, otherwise the user will not see the libraries.

To add source code files:

Project → Add Files... → Add

Source code files are shown by default.

4. Compile the project:

Project → Make

will invoke a compilation monitored by a compilation progress box. This box will also report when the application program is linked. If one gets no error messages in this process, an executable is produced and can be run by selecting:

Project → Run

Compiler and linker error and warning messages are documented in the Code Warrior reference manual which comes in “.pdf” and HTML formats on the documentation CD shipped with CW. All the best of luck with the programming.

Mac / OS >8.6

On the Mac CSP3D3.0 is distributed as a set of Metroworks Code Warrior Professional library files. Only CW version 4 or later is supported since CSP depends on the MSL++ implementation of the standard template library (STL) which was not properly support iterators in earlier versions. The library binaries can be included into projects in order to gain access to the CSP capabilities. Code Warrior can be ordered electronically from:
PC / Windows NT4.0

Similar to the Macintosh platform, CSP has been ported to Windows NT 4.0, using CodeWarrior Professional Vs 4.0 and greater. Importantly, no HDF library is provided on this platform, but a binary library file of the 4.2r2 library release can be downloaded from NCSA’s website at:

ftp.ncsa.uiuc.edu/pub/HDF

Linux

Earlier this year (1999) a Red Hat Linux distribution of CodeWarrior was released for the Linux operating system using the EGCS compiler. CSP3D3.0 is compatible with the EGCS compiler (vs. 1.1.2 or later), and the GNU C++ compiler 2.95.2 or later. CSP3D can be obtained either as a g-zipped tar file of the CSP binary libraries or as binary libraries for CodeWarrior on the Linux platform. The EGCS compiler can be downloaded from

http://egcs.cygnus.com/

The C++ specific improvements of the recent EGCS 1.1.1 are listed at:

http://egcs.cygnus.com/egcs-1.1/c++features.html

Notice that member function templates are now supported by many compilers, which means that much simpler code can be written and overloaded-method cluttered class interfaces can finally disappear in future CSP releases.

SGI / Irix 6.5 or Later

On the Silicon Graphics platform, the freely available EGCS compiler (vs. 1.1.2 or later), the GNU compiler, and theoretically also the Silicon Graphics MIPS Pro C++ compiler (vs. 7.3 or later), can be used to develop CSP3D3.0 applications. Since Stepanov, who wrote the original STL library, currently (2001) works for SGI, a very good implementation of this library is available on this platform and the Mips Pro compiler in the Developer’s Workshop package is robust and has relatively good warning and error reporting. However, this compiler always creates automatically a directory “ii_files” in which it stores linker information for the object files and it fails to track changes of header files in this repository. Often enough it is therefore necessary to delete the “ii_files”, for instance, if a compilation will fail to produce a library at the link stage. The compiler sometimes also fails to instantiate templates such that the object code for these is reported as missing when the libraries are linked at the end of the compilation. In this case, a sure way to get the necessary object code is to create dummy template instances in the main program which match the type of the missing ones. It seems much easier just to use the GNU compiler!
The Developer Pro Workshop on the SGI also includes the CaseVision debugger (cvd) which facilitates graphical debugging of C++ code. However, this debugger was not written for C++ and it is terribly slow and cumbersome to take a look into class objects. Given that the SGI developers workshop also is a commercial add-on software package, the academically freely available Sniff development environment is recommended for the SGI platform.
DEPRECATED CAPABILITIES AVAILABLE ON REQUEST

HDF Interface

CSP provides an interface to the NCSA HDF (http://hdf.ncsa.uiuc.edu/index.html) library. This allows you to save model results to hdf files which can be opened by Spyglass Transform and CSP auxiliary viewers on a range of different platforms. CSP uses the HDF library 4.1r2 but does not support HDF Windows NT. The format of HDF files which the CSP methods of the FileHandler class write is the following:

Fig. 12: Records stored in CSP HDF data files.
CSP Glossary

The present Glossary is a brief and by far not comprehensive alphabetical listing of the terminology employed in the documentation of CSP. Please give me feedback where items are missing and where explanations are too cryptic. I have also included some explanations of C++ jargon. If the listed expression is C++ terminology, I have indicated this by putting (C++) behind it. Specific CSP terminology like CSP object names is underlined.

Stephan K. Matthäi, July 22, 1999

**AMG.** Algebraic multigrid method (Ruge & Stueben 1987) used inside the CSP solver. See Briggs, W.L.’s multigrid tutorial (SIAM 1987) for a description of the method.

**Argument.** Input or output value of a method (associated with a class) or a function. This value is entered into or retrieved from the function by writing it into the braces behind the function name, e.g. MyFunction(argument1, argument2, argument3). In the function declaration placeholders are used for the arguments. These defined the type of the expected input or output, e.g., MyFunction(int argument1, House& argument2, Car& argument3). The & means that a reference to the actual input/output object is passed. Otherwise the function will make a copy of the argument for internal use (logically, large objects should not be copied and therefore always be passed by references).

**Baseclass.** A class object which describes the commonalities of an object hierarchy. The base class defines an interface for inherited objects, thereby specifying to the program how to handle these even when these are not known at the time when the base class is implemented.

**const.** C++ type qualifier which means that the expression which follows to the right is not modified after its definition or at least by the function to which it is passed, for example: const int pi=3.1414; or MyFunction(const Group& g ) const;

**constructor.** Default method with the same name as the class object, which it initializes when an instance of the class is created. Arguments to the constructor method provide initial data for the new object.

**enum.** C++ construct associating string variables with integer values such that these can be used as array indexing variables and for debugging.

**Group.** Collection of finite-elements, the global mesh ID numbers of which, are stored in an STL map. Use groups to manipulate geological entities of your model selectively.

**Interface.** List of operations supported by a piece of software or hardware; typically, the sum of the methods(=functions) of a class object which have been made accessible to the public.

**Operand.** A handle class (Operand) to a distributed CSP physical variable described in the PropertyDatabase object and stored in the CSP_MemoryManager (both objects inside the SuperGroup object). Operands allow you to access variables in
computations. To do this you construct an Operand with the variable name as constructor argument.

**Polymorphism.** The ability to treat different objects the same way which is defined in a base class.

**Subclass.** A class which is inherited from a base class. The base class thereby captures the commonalities of all objects to which the subclass belongs, for example, automobiles when the subclass implements a VW beetle. A class A is a subclass of a baseclass B when the statement “A is a B” is true.