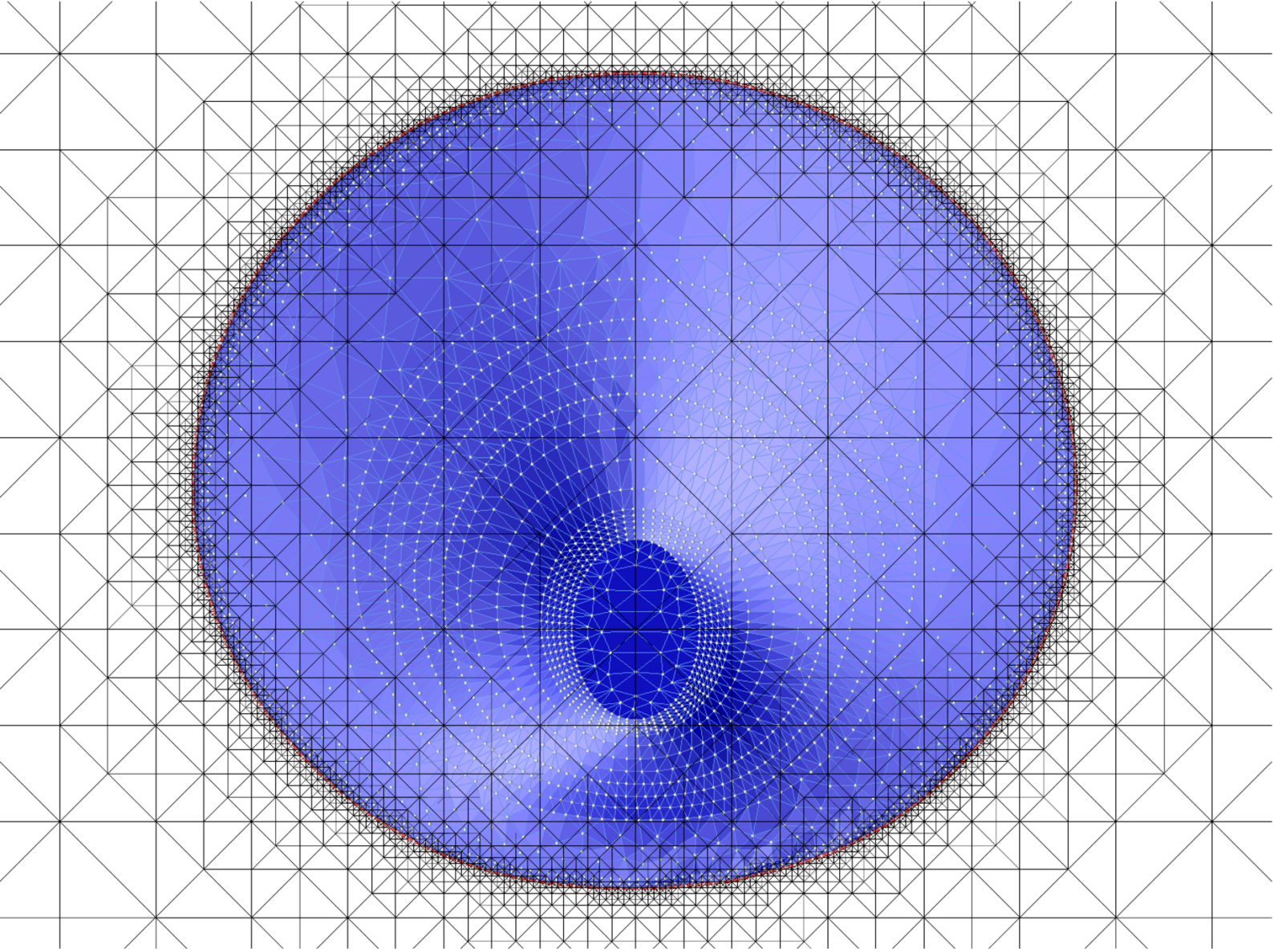
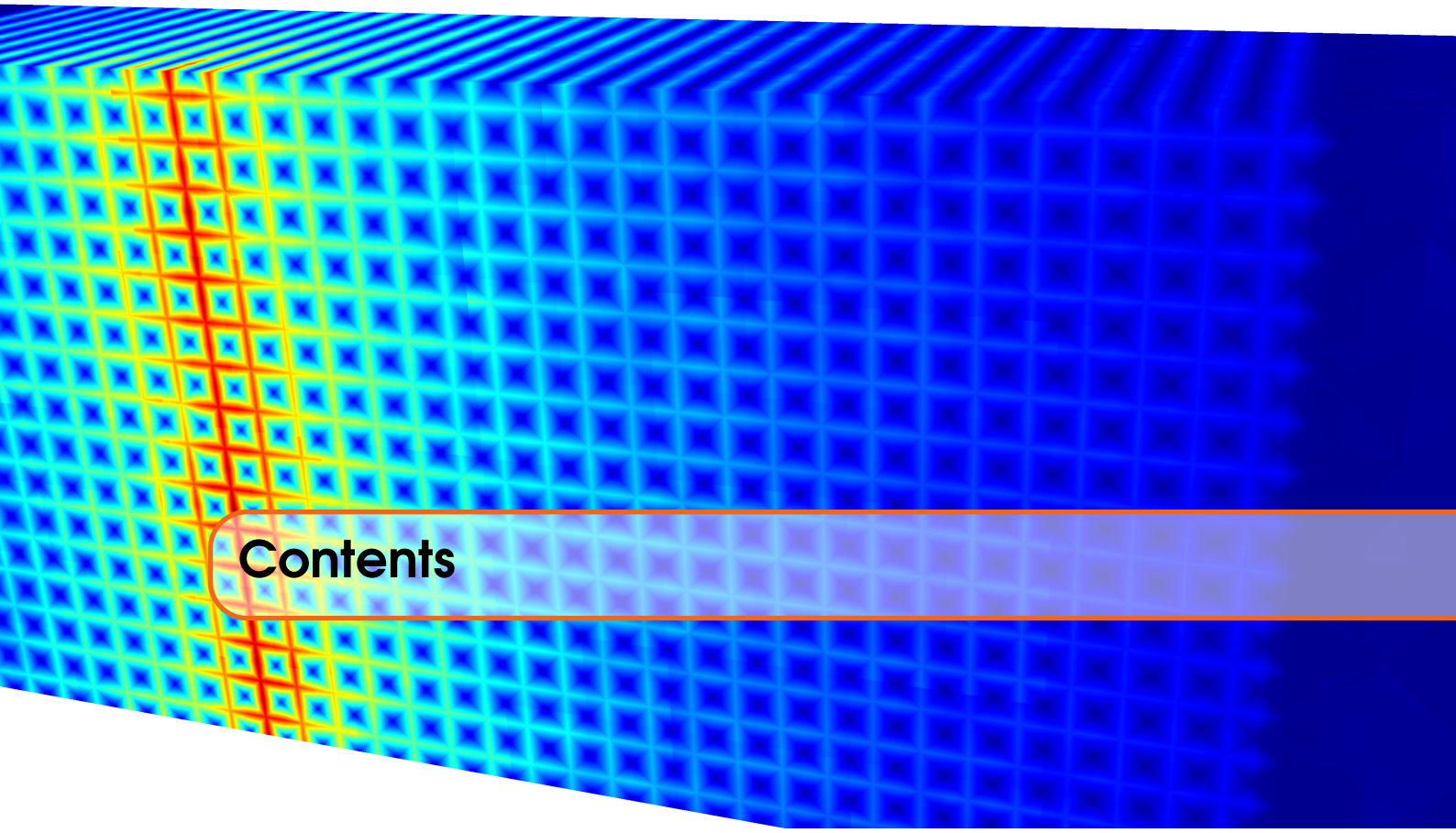


PUBLISHED BY ISET DEVELOPERS

GFEM.CEE.ILLINOIS.EDU

First printing, December 2016



Contents

1. Introduction

[heading=subbibliography]

2. The Grid File

The data defining a generalized finite element model to be solved (geometry, material data, loads, boundary conditions, etc.) is defined in a `phlex` formatted input file. The format used in a `phlex` formatted file is keyword based, straightforward to generate and very readable. The suffix is by default `*.grf` but this is not required. This chapter presents an overview of the basic data packets recognized by the `phlex` input format (*.grf) file reader. These packets describe the generalized finite element model by defining the nodes, element connectivity, material properties, boundary conditions and application specific data. Each packet of data is delimited by a set of keywords which define the beginning, the end, and the type of input to follow. Complete details describing this format are described in the next sections.

A `phlex` model definition file may be given any acceptable ASCII file name. A `'.grf'` extension has traditionally been used which refers to a grid file type of format. In addition to the model definition file, a companion `'.tcl'` file must be provided by the user. This file defines various parameters used by the solver as well as a command for loading the `'.grf'` file described here. Details on the `'.tcl'` file are provided in Chapter 4.

The `phlex` input file reader is written in C++ using a parser, and thus, spacing is quite arbitrary. New lines, spaces, etc., serve as delimiters. The characters '=' (equal sign), and ',' (comma) are allowed in the input file for better readability and are treated as separators equivalent to a blank. All of the data following a #, % or $ sign to the end of the line is treated as a comment. The input data *is case sensitive*, and thus, all keywords must appear exactly as shown.

A large number of `'.grf'` files can be found in *ISET* Quality Assurance (QA) Test Suite. To find them use the following commands from the command prompt at folder `SetSolver/qa-tests`

```
find . -name \*.grf | xargs grep PtTemperature
```

This will print a (long) list of `'.grf'` files where command `PtTemperature` is used. Files with examples of other commands can be found using the above commands with `PtTemperature` replaced by another command name.

2.1 General Packet Syntax

The phlex input file consists of a number of discrete data packets. Each packet is delimited by the keywords BEGIN packet and END as follows.

```
BEGIN packet
...
END
```

The keyword packet must be one of the following predefined options:

DOMAIN,
NODES,
ELEMENTS,
MATERIALS,
BOUNDARY,
BCASSIGN,
PROBLEMDATA,
SYSTEMS.

Each packet must be defined only once. The order of packets as listed above is recommended.

2.2 DOMAIN Packet

This packet of data simply defines the dimension of the problem. Valid keywords in this packet are

THREEDIMENSIONAL, TWODIMENSIONAL and ONEDIMENSIONAL.

These keywords defined the valid types of elements, materials, etc. that can be used in a .grf file. Example:

```
BEGIN DOMAIN
  THREEDIMENSIONAL
END
```

2.3 NODES Packet

This packet of data in the phlex input file contains the nodal point coordinates. The array of coordinate data (numerical values only) can optionally be preceded by one keyword:

```
NUMBERED
```

The NUMBERED option informs the input file reader that each set of node data will be preceded by a node ID. Node IDs are positive integers and are not necessarily ordered in a consecutive manner.

When NUMBERED is not used then each node is identified by its consecutive number (i.e., IDs are automatically generated).

Each set of node data contains an ID (if NUMBERED) and the X, Y, Z coordinates. It is an error to use the same ID for two or more nodes. Currently, only one NODES packet is allowed in the input file.

The general format of the NODES packet is:

```
BEGIN NODES
 NUMBERED
 ID x y z
 ...
END
```

A sample NODES packet where IDs are given and the for the case where the ID of the first two nodes are 10 and 50:

```
BEGIN NODES NUMBERED
 10      0.0, 1.0, 0.0
 50      1.0, 1.0, 0.0
 ...
END
```

A sample NODES packet for the case corresponding to no node IDs is:

```
BEGIN NODES
 0.0, 1.0, 0.0
 1.0, 1.0, 0.0
 ...
END
```

Since the NUMBERED keyword is not used, consecutive node IDs (1, 2, etc.) are automatically assigned to the nodes shown in the example above.

2.4 ELEMENTS Packet

This packet of data is used to define element connectivity in terms of global node numbers (i.e., node IDs defined in the NODES packet).

The ELEMENTS keyword may be followed by three optional keywords which define the format and type of data to appear with the element definitions. The additional keywords include:

```
NUMBERED
MATID
```

The NUMBERED option indicates that each element definition will be preceded by an element ID. Similar to node numbers, element numbers are required to be positive and unique integers, but not necessarily occurring in sequence. The MATID keyword indicates that each element definition will be followed by a string denoting the material ID. If a string is supplied it should be recognizable as a character string and not an integer or a real data value. This keyword is currently *not* optional and must be given.

Each element appearing in this packet must be preceded by a keyword defining the element type. In *ISET*, the available element types are:

LINE_3D,
PIPE_3D,
TRIA3,
TRIA3_3D,
TRIA6,
TRIA6_3,
TRIA6_3D,
QUAD4,
QUAD8,
TET4,
TET10,
TET10_4,
HEX8,
HEX20,
PlateRM_QUAD4,
PlateRM_QUAD4SI,
PlateRM_QLLL,
ShellRM_QLLL.

For these element classes, the defining node numbers must be appropriately sequenced as indicated in figures 2.1-2.4. Improper node sequencing will typically lead to negative Jacobian calculations which may or may not be detected until the solver is fired.

The sequencing and format for element data must be as follows:

- ID (if NUMBERED keyword given)
- Node IDs (in number and sequence appropriate for the element type)
- Material ID. The ID must be a string.

An example of an ELEMENTS packet with element IDs is:

```
BEGIN ELEMENTS NUMBERED MATID
  TET4  10  { 2 5 4 14  steel  }
  TET4  20  { 11 13 14 4  steel  }
END
```

The same example but with no element IDs is:

```
BEGIN ELEMENTS MATID
 TET4  {  2 5 4 14  steel  }
 TET4  {  11 13 14 4  steel  }
END
```

Since the NUMBERED keyword is not used, consecutive element IDs (1, 2, etc.) are automatically assigned to the elements.

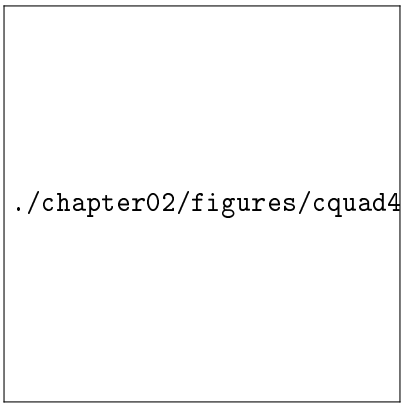


Figure 2.1: Node numbering sequence for quadrilateral QUAD4 elements.

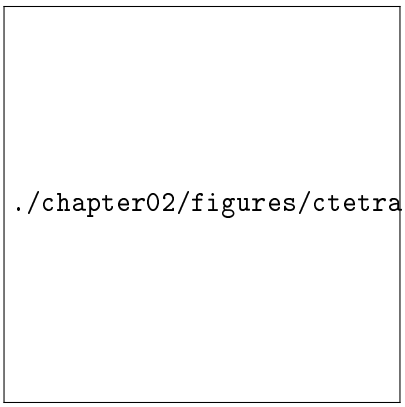


Figure 2.2: Node numbering sequence for linear TET4 tetrahedral elements.

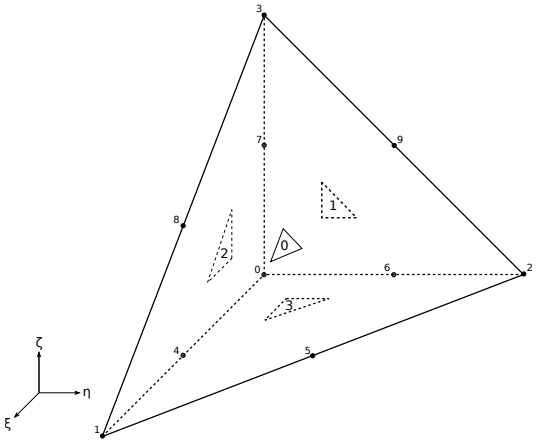


Figure 2.3: Node and face numbering sequences for quadratic `TET10` tetrahedral elements.

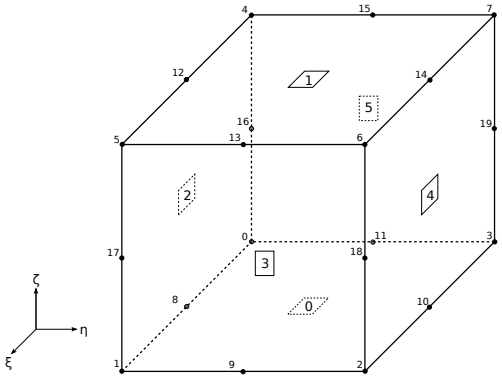


Figure 2.4: Node and face numbering sequences for quadratic `HEX20` hexahedral elements.

2.5 MATERIALS Packet

Materials are defined within a `MATERIALS` data packet. The material definitions includes the material type name – like `IsoHook3D, IsoHeat3D`, a user-supplied material name, and appropriate material constants. Material definitions end with an `END` keyword.

The material types currently available in *ISET* are:

- Solid Mechanics Materials:
  `IsoHook2D`, `IsoHook3D`, `OrthoHook3D`, `OrthoHookAbaqus3D`, `HeteroHook3D`, `RandHook3D`, `IsoPlas3D`, `IsoPlasAbaqus3D`, `HeteroPlas3D`, `RandPlas3D`, `IsoVisco3D`, `IsoHookPF2D`.
- Structural Mechanics Materials:
  `PlateRM`, `ShellRM`.
- Poisson and Heat Equation Materials:
  `IsoHeat2D`, `IsoHeat3D`, `FGMHeat3D`, `HeteroHeat3D`, `RandHeat3D`.
- Thermo-elasticity Materials:
  `IsoThermoElas3D`, `IsoThermoPlas3D`.
- Phase Field Materials:
  `IsoPF2D`, `IsoMPHookPF2D`.
- Newtonian Fluid and Lubrication Equation Material:
  `NewtonFluidLub2D`;
- Hagen-Poiseuille fluid (incompressible and Newtonian fluid in laminar flow flowing through a long cylindrical pipe of constant cross section):
  `Pipe1D`;
- Krauklis Wave Material:
  `Krauklis2D`;
- L^2 Projection "Materials":
  `MatL22D`, `MatL23D`.

`IsoHook2D` and `IsoHook3D` are 2-D and 3-D linear isotropic Hookean materials, respectively. `MatL22D` and `MatL23D` are L^2 projection of functions in 2-D and 3-D, respectively. `IsoHeat2D` and `IsoHeat3D` are used for problems of heat transfer. The next sections describe the different material types and gives a detailed overview of the user input. Here we focus on the general specifications and examples of the relevant syntax for the `MATERIALS` packet.

The `MATERIALS` packet associates a material name with a material type and defines the constants required by each type. The general format for the materials specification in *ISET* for elasticity problems is:

```
BEGIN MATERIALS
  IsoHook3D    matname1   {  material constants  }
  IsoHook3D    matname2   {  material constants  }
END
```

A sample `MATERIALS` packet including examples of materials for elasticity problems follows:

```
BEGIN MATERIALS
 IsoHook3D steel { E =  350000, nu = 0.4 }
 IsoHook3D steel_bf { E =  350000, nu = 0.4, BodyForce = 1 0 0 }
 IsoHook2D Mat1l {E =  2.0e5, nu = 0.3, StressState = PlaneStress,
                  Thickness =  10.0, BodyForce = -1.0e-4 0.0 }
END
```

The general format for the materials specification in *ISET* for heat transfer problems is:

```
BEGIN MATERIALS
  IsoHeat3D    matname1  {  material constants  }
END
```

A sample `MATERIALS` packet including examples of material types for heat transfer problems follows:

```
BEGIN MATERIALS
 IsoHeat3D steel { Conductivity =  43 }
 IsoHeat3D steel_capacity { Conductivity =  43, Capacity =  10 }
 IsoHeat3D steel_intsource { Conductivity =  43, HeatSource = 10 }
 IsoHeat3D steel_intsource_funct { Conductivity =  43,
       HeatSourceFunctionNum = 3 }
END
```

2.6 Solid Mechanics Materials

Syntax convention adopted for all materials:

```
[ ] Encloses optional parameters
{ } Encloses mandatory parameters
 | Separates two or more items within brackets and braces, only one of
which may be chosen
```

R This is a remark.

2.6.1 IsoHook3D

This material represents an isotropic Hookean material in *ISET*. Since the elastic properties of an isotropic material are independent of any specific direction, only two independent material constants are needed. The first material constant is called `E` for elastic modulus often called Young's modulus. The second material constant is `nu` known as the Poisson's ratio.

The general format for the `IsoHook3D` material specification in *ISET* is:

```
IsoHook3D mat_name { nu = value E = value [rho = value]
                     [ BodyForce = fx fy fz ]
                     [ alpha = value [ Temperature = value |
                       TempDistFunctNum = value ]
                     ] }
```

- `nu` is the Poisson's ratio of the material.
- `E` is the Young's modulus of the material.
- `rho` is mass density of the material. This is relevant only when solving dynamics problems.
- `BodyForce` are the components of the body force acting on the elements with this material. Components are with respect to the global Cartesian coordinate system.
- `alpha` is the coefficient of thermal expansion of the material.
- `TempDistFunctNum` is the index of a function representing the temperature distribution over elements with this material. This function is hardwired in the `IsoHook3D` material class. This parameter is mainly used by *ISET* developers.

Parameters `alpha`, `Temperature` and `TempDistFunctNum` are used only if solving a thermoelasticity problem.

The following example illustrates the definition of two `IsoHook3D` materials called *steel*, and *rubber*.

```
BEGIN MATERIALS
  IsoHook3D steel { E = 29000000, nu = 0.285 }
  IsoHook3D rubber { E = 520, nu = 0.490 }
END
```

The values of Poisson's ratio typically range from 0.0 to 0.5 with values near 0.5 representing a near incompressibility condition. In some instances, values of `nu` very close to 0.5 may cause computational difficulties. Therefore in *ISET* an upper limit of 0.499 for `nu` is suggested. The lower limit is 0.0 to prevent anti-compressibility.

2.6.2 IsoHook2D

This is the two-dimensional counterpart to `IsoHook3D`. The general format for the `IsoHook2D` material specification in *ISET* is:

```
IsoHook2D mat_name { nu = value E = value [rho = value]
            StressState = [PlaneStress | PlaneStrain | AxisSymmetric]
            [ Thickness = value  ]
            [ BodyForce = fx fy  ]
            [ alpha = value [ Temperature = value |
              TempDistFunctNum = value ]
            ] }
```

- `nu` is the Poisson's ratio of the material.
- `E` is the Young's modulus of the material.
- `rho` is mass density of the material. This is relevant only when solving dynamics problems.
- `StressState` is used to select the 2-D stress state of the elements with this material.
- `Thickness` is the thickness of the 2-D domain.

R `Thickness` is used only if `StressState = PlaneStress`.

- `BodyForce` are the components of the body force acting on the elements with this material. Components are with respect to the global Cartesian coordinate system.
- `alpha` is the coefficient of thermal expansion of the material.
- `TempDistFunctNum` is the index of a function representing the temperature distribution over elements with this material. This function is hardwired in the `IsoHook3D` material class. This parameter is mainly used by *ISET* developers.

R Parameters `alpha`, `Temperature` and `TempDistFunctNum` are used only if solving a thermoelasticity problem.

2.6.3 OrthoHook3D

This material represents an orthotropic Hookean material in *ISET*. The elastic properties of this material are symmetric with respect to three mutually orthogonal planes. This leads to nine independent material constants. It is also necessary to define the orientation of the preferred material directions (the base vectors perpendicular to the planes of symmetry). These directions are also known as principal material directions.

`OrthoHook3D` materials can be specified in *ISET* using one of the fallowing formats

Format 1:

This format is similar to the `*ENGINEERING CONSTANTS` format in Abaqus. The only difference between *ISET* and Abaqus formats is that the material properties and the orientation of the material coordinate system are defined separately in Abaqus while in *ISET* they are defined as shown below.

```
OrthoHook3D mat_name { E_1 = value E_2 = value E_3 = value
                       nu_12 = value nu_23 = value nu_31 = value
                       G_12 = value G_23 = value G_31 = value
                       Orientation_1 = value value value
                       Orientation_2 = value value value
                       Orientation_3 = value value value}
```

- `E_1`, `E_2` and `E_3`, are the Young's modulus of the material in the material directions `Orientation_1`, `Orientation_2` and `Orientation_3`, respectively.
- `nu_12`, `nu_23` `nu_31` are the Poisson's ratio of the material with respect to material directions 1-2, 2-3 and 3-1, respectively. The quantity ν_{ij} has the physical interpretation of the Poisson's ratio that characterizes the transverse strain in the j-direction, when the material is stressed in the i-direction.

R In general, $\nu_{ij} \neq \nu_{ji}$ but they are related by

$$\nu_{ij}/E_i = \nu_{ji}/E_j$$

- `Orientation_1`, `Orientation_2` and `Orientation_3` are base vectors defining the material coordinate system.

R The base vectors *must* be of unity length and orthogonal to each other. *ISET* does not check if this is indeed the case!

R Transversely isotropic materials are special cases of orthotropic ones. Thus, they can be defined in *ISET* using an `OrthoHook3D` material as described above.

Format 2:
This format is only suitable when the material model is implemented in an Abaqus [**?**] UMAT_ELAS FORTRAN subroutine compiled and linked with *ISET* as described in Section 2.6.4. This format provides only two material parameters and the directions of the material coordinate system. *The actual material properties are built in the UMAT_ELAS subroutine and parametrized by* `g1` *and* `g2`*.*

The same units adopted for the material properties defined in subroutine UMAT_ELAS must be used for the definition of the GFEM mesh, loads, etc!

Details on how to integrate this and other Abaqus UMAT subroutines in *ISET* can be found in Section 2.6.4.

This material format is given by

```
OrthoHook3D mat_name { g1 = value g2 = value
                       Orientation_1 = value value value
                       Orientation_2 = value value value
                       Orientation_3 = value value value
                       effective_E = value
                       effective_nu = value}
```

- `Orientation_1`, `Orientation_2` and `Orientation_3` are base vectors defining the material coordinate system.
- `effective_E` and `effective_nu` are the effective values of Young's Modulus and Poisson's ratio used to compute penalty for the application of Dirichlet boundary conditions.

The base vectors *must* be of unity length and orthogonal to each other. *ISET* does not check if this is indeed the case!

Input arguments `g1` and `g2` have no effect on the elasticity tensor provided by the material model defined in subroutine UMAT_ELAS shipped with *ISET* (see file `umat_elas.f`). The user is expected to replace this default UMAT_ELAS by an actual implementation of such material model. See also file README_UMAT.txt shipped with *ISET*.

The following two examples lead to *exactly* the same material model used in *ISET* (see Section 2.6.4 for details on material `OrthoHookAbaqus3D`):

```
OrthoHook3D matName { g1 = 0.009247 g2 = 0.337048
  Orientation_1 = 0.003859, 0.017532, 0.999839
  Orientation_2 = -0.54203, 0.840262, -0.01264
  Orientation_3 = -0.84035,  -0.5419, 0.012746
  effective_E = 100000000 effective_nu = 0.3}

OrthoHookAbaqus3D matName { num_props = 11 props = 0.009247 0.337048
 0.003859 0.017532 0.999839
-0.54203 0.840262 -0.01264
-0.84035 -0.5419 0.012746
 effective_E = 100000000 effective_nu = 0.3}
```

2.6.4 OrthoHookAbaqus3D

The *ISET* material class `OrthoHookAbaqus3D` is essentially a wrapper to a linearly elastic material model defined in an Abaqus [**?**] UMAT_ELAS FORTRAN subroutine. Therefore an *ISET*

`OrthoHookAbaqus3D` material replicates a user-provided UMAT_ELAS material subroutine. The format used to provide the material parameters is analogous to the one adopted in Abaqus

```
OrthoHookAbaqus3D mat_name { num_props = value
                             props = value(s)
                             [effective_E = value
                              effective_nu = value]}
```

- `num_props` is the number of material properties for the material defined in the user-provided UMAT_ELAS material subroutine.
- `props` is a list of dimension `num_props` with the values of the material properties. The order of the items in the list must be compatible with the expected order of data in argument `PROPS` of the corresponding UMAT_ELAS subroutine.
- `effective_E` and `effective_nu` are optional parameters which can be given with this material model definition. They represent the effective values of Young's modulus and Poisson's ratio used to compute penalty for the application of Dirichlet boundary conditions. If these values are not given, then the first and second members of the list `props` will be taken as Young's modulus and Poisson's ratio for the computation of penalty to apply Dirichlet boundary conditions.

Before using an `OrthoHookAbaqus3D` material in *ISET*, the user must compile the corresponding FORTRAN UMAT_ELAS subroutine using the following command (single line issued in folder with file `file_with_UMAT_ELAS.f`)

```
gfortran -cpp -shared -fPIC umat.f {file_with_UMAT_ELAS.f} \
         -o {full_path_to_iset_dyn_lib_folder}/libabaqususersubs.so
```

The above command compiles FORTRAN file `file_with_UMAT_ELAS.f` and generates a shared library named `libabaqususersubs.so`. If there are multiple FORTRAN UMAT user files, e.g. umat_elas.f, umat_plas.f, etc., they must be added to the compilation of the shared library. Similarly, if other Abaqus subroutines are also used, they should be compiled along with file `file_with_UMAT_ELAS.f` to generate the shared library `libabaqususersubs.so`. For instance, if the user subroutine DLOAD from Section 2.16.1 and DFLUX from Section 2.17.1 are used, the command to generate the shared library will be

```
gfortran -cpp -shared -fPIC umat.f \
         {file_with_UMAT_ELAS.f} {file_with_DLOAD.f} {file_with_DFLUX.f} \
         -o {full_path_to_iset_dyn_lib_folder}/libabaqususersubs.so
```

Dynamic ibrary `libabaqususersubs.so` should then be copied to the folder with other shared libraries used by *ISET*. This folder is named `lib` and already contains a file named `libabaqususersubs.so` on a Linux system, or `libabaqususersubs.dylib` on Mac OS system. The existing file should be replaced by the new library created as described above. String `full_path_to_iset_dyn_lib_folder` must be replaced by the full path to the folder with existing shared (dynamic) libraries used by *ISET*. As an example, in my system and for a UMAT_ELAS in file named `umat_elas.f`, the command is (single line)

```
gfortran -cpp -shared -fPIC umat.f umat_elas.f \
 -o /Users/armando/SetSolver_hg/ProjectMakefile/lib/libabaqususersubs.so
```

The above procedure must be repeated every time the UMAT_ELAS subroutine changes or a new UMAT_ELAS implementation is required by the user. *ISET* will load the shared library `libabaqususersubs.so` in folder `lib` at run time and thus making the UMAT_ELAS defined in file_with_UMAT_ELAS.f available for the *ISET* user.

R File `umat.f` listed in the above compilation commands is shipped with ISET. It simply provides an interface to call different UMAT subroutines defined by the user. This allows ISET to be compiled with more than one UMAT subroutine at a time and solve problems using different material models, each one defined in a different UMAT-type subroutine.

R The compilation flag `-DABAQUS_USER_SUBS` must be defined when compiling *ISET* in order to activate the support for Abaqus user subroutines. This is done by setting `CMake` variable `ISET_COMPILE_FORTRAN_FILES` to true while in `CMake`. This is only relevant for those users that have the source code of *ISET*.

2.6.5 HeteroHook3D

```
HeteroHook3D mat_name { nu1 = value nu2 = value E1 = value E2 = value
                        Interpolate = int_value
                      [ BodyForce = fx fy fz ]
                      [ alpha1 = value alpha2 = value
                        [ AmbientTemp = temp ] [ Temperature = value ]
                      ] }
```

R Parameters `alpha1`, `alpha2`, `AmbientTemp` and `Temperature` are used only if solving a thermoelasticity problem.

2.6.6 RandHook3D

R

2.6.7 IsoPlas3D

R

2.6.8 IsoPlasAbaqus3D

The *ISET* material class `IsoPlasAbaqus3D` is essentially a wrapper to a plastic material model defined in an Abaqus [**?**] UMAT_PLAS FORTRAN subroutine. Therefore an *ISET* `IsoPlasAbaqus3D` material replicates a user-provided UMAT_PLAS material subroutine. The format used to provide the material parameters is analogous to the one adopted in Abaqus.

```
IsoPlasAbaqus3D mat_name { num_props = value
                           props = value(s)
                           num_depvar = value
                           [ effective_E = value
                             effective_nu = value ]}
```

- `num_props` is the number of material properties required by the user-provided UMAT_PLAS material subroutine.
- `num_depvar` is the number of dependent variables defined in the user-provided UMAT_PLAS material subroutine.
- `props` is a list of dimension `num_props` with the values of the material properties. The order of the items in the list must be compatible with the expected order of data in argument `PROPS` of the corresponding UMAT_PLAS subroutine.
- `effective_E` and `effective_nu` are optional parameters which can be given with this material model definition. They represent the effective values of Young's modulus and Poisson's ratio used to compute penalty for the application of Dirichlet boundary conditions. If these values are not given, then the first and second members of the list `props` will be taken as Young's modulus and Poisson's ratio for the computation of penalty to apply Dirichlet boundary conditions.

R The same units adopted for the material properties defined in subroutine UMAT_PLAS must be used for the definition of the GFEM mesh, loads, etc!

For instructions on compilation of ISET with ABAQUS UMAT_PLAS subroutines, please see section 2.6.4.

Example:

```
IsoPlasAbaqus3D Steel { num_props = 4 props = 4.0 0.0 4.0 1.0
                        num_depvar = 7 }
```

2.6.9 HeteroPlas3D

R

2.6.10 RandPlas3D

R

2.6.11 IsoVisco3D

This material represents an isotropic visco-elastic material in *ISET* using Prony series for the shear and bulk moduli given in terms of *normalized* coefficients

$$G(t) = G_0 \left[1 - \sum_{k=1}^{N} \bar{G}_k \left(1 - e^{(-t/\tau_k)} \right) \right]$$

$$K(t) = K_0 \left[1 - \sum_{k=1}^{N} \bar{K}_k \left(1 - e^{(-t/\tau_k)} \right) \right]$$

where G_0 and K_0 are the instantaneous (elastic) shear and bulk moduli determined from the values of the user-defined instantaneous elastic moduli E_0 and ν_0 using

$$G_0 = \frac{E_0}{2(1+\nu_0)} \qquad K_0 = \frac{E_0}{3(1-2\nu)}$$

$\bar{G}_k = G_k/G_0$ is the normalized modulus the k^{th} term in the Prony series expansion of the shear relaxation modulus.
$\bar{K}_k = G_k/G_0$ is the normalized modulus the k^{th} term in the Prony series expansion of the bulk relaxation modulus.

The general format for the `IsoVisco3D` material specification in *ISET* is:

```
IsoVisco3D mat_name { nu = value E = value
                      numPronyCoeff = value
                      normal_shear_relaxation = value1 value2 ... valuen
                      normal_bulk_relaxation  = value1 value2 ... valuen
                      relaxation_time  = value1 value2 ... valuen
                      [TRS = Tactual Tref C1 C2]
                      [ BodyForce = fx fy fz ] }
```

- `nu` is the instantaneous (elastic/initial) Poisson's ratio of the material (ν_0).
- `E` is the instantaneous (elastic/initial) Young's modulus of the material (E_0).
- `numPronyCoeff` is the number of terms in the Prony series used to represent the material (N).
- `normal_shear_relaxation` are normalized Prony series coefficients of shear relaxation modulus: $\bar{G}_k,\ k = 1,\ldots,N$.
- `normal_bulk_relaxation` are normalized Prony series coefficients of bulk relaxation modulus: $\bar{K}_k,\ k = 1,\ldots,N$.
- `relaxation_time` are the relaxation times in the Prony series for shear and bulk modulus: $\tau_k,\ k = 1,\ldots,N$.
- `TRS` are the constants for time-temperature superposition equation based on William-Landell-Ferry (WLF) equation. This data is *optional*.
  `TRS` = Tactual Tref C1 C2
  - `Tactual` = solid temperature
  - `Tref` = reference temperature,
  - `C1, C2` = calibration constants of William-Landell-Ferry equation.
- `BodyForce` are the components of the body force acting on the elements with this material. Components are with respect to the global Cartesian coordinate system. This data is *optional*.

2.6.12 IsoHookPF2D

This material is the elastic part of a phase field material. It is an isotropic Hookean elastic material that makes use of the phase field order parameter to penalize stiffness.

```
IsoHookPF2D mat_name { nu = value E = value
                       StressState = [PlaneStress | PlaneStrain ]
                       [ Thickness = value  ] }
```

2.7 Structural Mechanics Materials

2.7.1 PlateRM

This class represents material and structural properties (like thickness) of Reissner-Mindlin plate elements. The general format for the `PlateRM` material specification in *ISET* is:

```
PlateRM mat_name { nu = value E = value [rho = value]
                   Thickness = value
                   [ BodyForce = {fz mx my} ] }
```

- `nu` is the Poisson's ratio of the material.
- `E` is the Young's modulus of the material.
- `rho` is mass density (mass per unity area) of the material. This is relevant only when solving dynamics problems.
- `BodyForce` has the components of the force and moments (per unity area) acting on elements with this material.
  Components are with respect to the global Cartesian coordinate system.

2.7.2 ShellRM

This class represents material and structural properties (like thickness) of Reissner-Mindlin flat shell elements. The general format for the `ShellRM` material specification in *ISET* is:

```
ShellRM mat_name { nu = value E = value [rho = value]
                   Thickness = value
                   [ BodyForce = {fx fy fz mx my mz} ] }
```

- `nu` is the Poisson's ratio of the material.
- `E` is the Young's modulus of the material.
- `rho` is mass density (mass per unity area) of the material. This is relevant only when solving dynamics problems.
- `BodyForce` has the components of the force and moments (per unity area) acting on elements with this material.
  Components are with respect to the global Cartesian coordinate system.

2.8 Poisson and Heat Equation Materials

2.8.1 IsoHeat3D

```
IsoHeat3D mat_name { Conductivity = value
                     [ Capacity = value  ]
                     [ HeatSource = value
                      | HeatSourceFunctionNum = source_num
                     ] }
```

This class represents an isotropic thermal material in *ISET*. Since the thermal properties of an isotropic material are independent of any specific direction, only one material constant is needed. The material constant is called `Conductivity`, for the thermal conductivity of the material, often denoted by κ.

The following example illustrates the definition of two `IsoHeat3D` materials called *steel*, and *titanium*.

```
BEGIN MATERIALS
  IsoHeat3D steel { Conductivity = 43 }
  IsoHeat3D titanium { Conductivity = 21 }
END
```

2.8.2 IsoHeat2D

```
IsoHeat2D mat_name { Conductivity = value
                   [ Thickness = value  ]
                   [ Capacity = value  ]
                   [ HeatSource = value
                     | HeatSourceFunctionNum = source_num
                   ] }
```

2.8.3 FGMHeat3D

2.8.4 HeteroHeat3D

2.8.5 RandHeat3D

2.9 Thermo-elasticity Materials

2.9.1 IsoThermoElas3D

2.9.2 IsoThermoPlas3D

2.10 Phase Field Materials

2.10.1 IsoPF2D

This material represents an isotropic phase field material. It solves for a scalar order parameter that is used to penalize the stiffness of elastic materials based on the phase field model of fracture.

```
IsoPF2D mat_name { Gc = value delta = value
                   model = [AT1 | AT2 | AT1Nucleation | CZM]
                   [ crackIrrev = [ penalty | historyVar ]  ]
                   [sigCS = value ] [sigTS = value] [deltaEps = value]
                   [E = value] [ softeningLaw = [ LINEAR |
                                                  EXPONENTIAL |
                                                  HYPERBOLIC |
                                                  CORNELISSEN ] ]}
```

- `Gc` is the material's critical strain energy release rate.
- `delta` is the regularization length for the phase field fracture.
- `model` set the phase field model used, this is either AT1 or AT2 – based on the Ambrosio-Tortorelli models.
- `crackIrrev` this determines how irreversibility of fracture is enforced. The penalty and history variable implementations are in place. Penalization is used by default.
- `sigCS` is the compressive strength of the material used to build the Drucker-Prager strength surface.
- `sigTS` is the tensile strength of the material used to build the Drucker-Prager strength surface.
- `deltaEps` is the regularization length to be sure the nucleation phase field model accurately captures nucleation from existing fractures.
- `E` is the Young's modulus of the material. It should be the same value used for elastic material.
- `softeningLaw` is softening law to be used for the Cohesive Zone Model (CZM) model.

R The current implementation assumes zero Neumann boundary conditions on all boundaries of the phase field BVP.

R The parameters `sigCS, sigTS, deltaEps` are required for AT1Nucleation phase field. These commands cannot be passed for AT1, AT2, or CZM phase field materials.

R The parameters `sigCS, sigTS, E, softeningLaw` are required for CZM phase field. These commands cannot be passed for AT1, AT2, or AT1Nucleation phase field materials.

R AT1, AT2, and AT1Nucleation models do *not* follow the damage convection, meaning $z = 0.0$ and $z = 1.0$ for damage and undamaged, respectively. On the other hand, CZM follows damage convection. For CZM model do not use `phaseFieldSolOption` to initialize the phase-field solution with $z = 1.0$.

2.10.2 IsoMPHookPF2D

This is a wrapper material that carries the phase field elasticity and order parameter materials.

```
IsoMPHookPF2D mat_name { PFMat = pfMat_name ElasMat = elasMat_name }
```

R This material must be declared after the phase field and phase field elasticity materials have been created in the grf file.

2.11 Newtonian Fluid and Lubrication Equation Material

2.11.1 NewtonFluidLub2D

2.12 Hagen-Poiseuille Fluid Material

2.12.1 Pipe1D

This one-dimensional material implements the Hagen-Poiseuille equation governing an incompressible and Newtonian fluid in laminar flow flowing through a long cylindrical pipe of constant cross section.

```
Pipe1D mat_name { [ {Viscosity | mu} = value {Radius | rad} = value | kw = value ]
                    {Density | rho} = value
                    {NumberPerf | n} = value
                    {PerfDiam | Dp} = value
                    {DischargeCoeff | Cd} = value
                    EntryFrictionalCoeff = value
                }
```

where

- `Viscosity | mu` is the viscosity of the fluid.
- `Radius | rad` is the radius of the pipe.
- `kw` is the pipe conductivity and calculated as $\texttt{kw} = \frac{\pi\ \texttt{rad}^4}{8\ \texttt{mu}}$
- `Density | rho` is the mass density of the fluid.
- `NumberPerf | n` is the number of perforations.
- `PerfDiam | Dp` is the diameter of the perforations.
- `DischargeCoeff | Cd` is the discharge coefficient of the perforations.
- `EntryFrictionalCoeff` is the entry frictional coefficient calculated as $\frac{0.8072\ \texttt{rho}}{\texttt{n}^2\ \texttt{Dp}^4\ \texttt{Cd}^2}$

A standard Hagen-Poiseuille material can be defined with either the pipe radius (`rad`) and fluid viscosity (`mu`), or by defining the pipe conductivity (`kw`) directly.

A connecting pipe material that implements the pressure loss in the perforations can defined using the fluid density (`rho`), number of perforations (`n`), diameter of the perforations (`Dp`), and discharge coefficient of the perforations (`Cd`). Conversely, it can be defined by directly setting the entry frictional coefficient (`EntryFrictionalCoeff`).

2.13 Krauklis Wave Material

2.13.1 Krauklis2D

This is the two-dimensional Krauklis wave material. It implements Krauklis waves in fluid filled fractures.

For simulating fully coupled Krauklis waves, an `IsoHook3D` material also needs to be defined in the same `.grf` file. If no `IsoHook3D` is defined, the two-dimensional acoustic wave equations are solved.

The general format for the `Krauklis2D` material specification in *ISET* is:

```
Krauklis2D mat_name { {Density | rho} = value
                      {BulkModulus | kf} = value
                      [{Viscosity | mu} = value]  }
```

- `Density | rho` is the mass density of the fluid.
- `BulkModulus | kf` is the bulk modulus of the fluid.
- `Viscosity | mu` is the viscosity of the fluid.

2.14 L^2 Projection "Materials"

2.14.1 MatL22D

2.14.2 MatL23D

2.15 BOUNDARY and BCASSIGN Packets

In *ISET*, boundary conditions are created using a two step process. In the first step, boundary condition objects are defined. This definition begins with the keyword `BEGIN BOUNDARY`, and terminates with an `END` keyword. Between the keywords, face and nodal boundary condtions objects are defined. This entails given, for each boundary condition defined, (i) the boundary condition type, (ii) a unique name, and (iii) associated arguments. The current boundary condition classes available in *ISET* are:

- Solid Mechanics Problems, Face and Nodal Boundary Conditions Classes:

  `Traction`, `TractionAbaqusUserSub`, `Displacement`, `Spring`,

  `PtDisplacement`, `PtForce`, `PtForceAbaqusUserSub`, `PtSpring`;

- Poisson and Heat Equation Problems, Face and Nodal Boundary Conditions Classes:

  `Flux`, `FluxAbaqusUserSub`, `Flux2D`, `Flux2DAbaqusUserSub`, `Temperature`, `Temperature2D`, `Convection`, `Convection2D`,

  `PtTemperature`, `PtTemperature2D`, `PtFlux`, `PtConvection`;

- Thermo-elasticity Problems Face and Nodal Boundary Conditions Classes:

  `Traction`, `TractionAbaqusUserSub`, `Displacement`, `Spring`, `Flux`, `FluxAbaqusUserSub`, `Temperature`, `Convection`, `ThermoElasBC`, `ThermoPlasBC`,

  `PtDisplacement`, `PtForce`, `PtForceAbaqusUserSub`, `PtSpring`; `PtTemperature`, `PtFlux`, `PtConvection`;

- Newtonian Fluid and Lubrication Equation Face and Nodal Boundary Conditions Classes:

  `Flux`, `Flux2D`, `Pressure`, `Pressure2D`,

  `PtPressure`, `PtPressure2D`, `PtFlux`;

- Hagen-Poiseuille Equation Nodal Boundary Conditions Classes:

  `PtPressure, PtPressurePipe, PtFlux, PtFluxPipe, PtFlowPipe`

Once a *face* boundary object is created (i.e., defined in packet `BOUNDARY`), the second step is to assign its *name* to specific element faces. This is done in the `BCASSIGN` packet which begins with the keyword `BEGIN BCASSIGN` and terminates with an `END` keyword. Between the keywords, element identification numbers, face numbers, and boundary name identifications are provided for each element face associated with a particular boundary object. Further details are given in Section 2.15.1.

The general format for defining boundary condition objects and assigning them to element faces is:

```
BEGIN BOUNDARY
  Boundary_Class boundary_name { arguments }
END

BEGIN BCASSIGN
  element_id face_id name
END
```

Above, `Boundary_class` is one of the *face* boundary conditions classes listed earlier. The *boundary_name* is a user prescribed character string used to identify a particular instance of a boundary class. The *arguments* define the specific boundary condition data for the `Boundary_class`. Details are privided next.

2.15.1 BCASSIGN Packet

The `BCASSIGN` packet is used to assign face based boundary conditions to the faces of elements. Thus, all `Traction, Displacement, Spring`, etc., boundary conditions which were defined in the `BOUNDARY` packet *must* be assigned to element faces in this packet. If the face based boundary data is not assigned here, it will be ignored. Unlike many of the other packets described here, this packet must appear in your input file after the elements and boundary conditions are defined (i.e., after the `ELEMENTS` and `BOUNDARY` packets).

The general format for prescribing boundary condition assignments is:

```
BEGIN BCASSIGN
  element_id  face_number  boundary_name
END
```

Here *element_id* is the element ID either explicitly or implicitly given in the `ELEMENTS` packet, *face_number* is the element face number which must be defined appropriately for each element type (see for example Figure 2.3 for face numbers for the TET10 elements), and *boundary_name* is a user defined boundary name given in the `BOUNDARY` packet.

A sample `BCASSIGN` input packet assigning face based boundary conditions to element 1 face 0 and element 2 faces 4 and 5 is:

```
BEGIN BCASSIGN
  1 0 spring
  2 5 traction
  2 4 fixed
  ...
END
```

A few additional key points regarding the BCASSIGN packet should be kept in mind;

- Currently, only one boundary condition can be applied to a face of an element. If multiple boundary conditions are assigned to a single element face, they must belong to different load cases and it must be a Neumann boundary condition.
- It is not necessary to provide a boundary condition assignment for each element face. Those faces which are not explicitly assigned will default to a zero Neumann (traction, flux, etc.) condition.

2.16 Solid Mechanics Face Boundary Conditions

2.16.1 Surface Tractions: Traction and TractionAbaqusUserSub

The surface traction vector in *ISET* follows the Lagrangian rule: the direction of the force in the deformed and undeformed configuration is the same and always acts on the undeformed surface area. Surface tractions in *ISET* can be specified using the following formats.

```
Traction trac_name  { Components =  t1  t2  [ t3 ]
                      | LoadFunctions = trac_fun_num
                      | UserFunction = func_name   pOrder = p_ord_integ
                       [rhs = rhs_number]
                       [ CoordinateFrame = FaceAligned ] }
```

where

- LoadFunctions is the number of a hardwired function used to compute the traction vector.
- UserFunction is the name of a Python function used to compute the traction vector. This is a non-constant function of (x, y, z) and potentially time, defined in a Python script.
- pOrder is the p-order equivalent to the user-prescribed boundary condition function UserFunction. This is used for numerical integration over an element face.
- Paramenter rhs is taken as zero if not given.
- Components are w.r.t. global Cartesian system by default: $t1 \equiv t_x$, $t2 \equiv t_y$, $t3 \equiv t_z$.
- If optional argument CoordinateFrame = FaceAligned is given, Components are w.r.t. to coordinate system n,t,s aligned with the element face this BC is assigned to: $t1 \equiv t_n$, $t2 \equiv t_t$, $t3 \equiv t_s$.
  n = vector normal to element face
  t = tangent vector in the direction of the element ksi/rr coord. line
  s = tangent vector $s = n \times t$

R Option UserFunction can only be used with *ISETPy*, a Python interface to *ISET*.

```
TractionAbaqusUserSub trac_name  { [rhs = rhs_number]
                                   [ CoordinateFrame = FaceAligned |
                                                       GlobalCoordSys] }
```

where

- Paramenter `rhs` is taken as zero if not given.
- If optional argument `CoordinateFrame = GlobalCoordSys` is given, the traction defined in the user subroutine DLOAD will be transformed to the basic (X,Y,Z) system. The default option is `CoordinateFrame = FaceAligned` where the load is applied in the normal direction of the face.

The n,t,s coord. system changes, in general, from element-face to element-face. Thus, different `Tractions` may have to be created at `.grf` file (one per element-face that uses it): I.E. this Traction object can not in general be applied to different element faces. However that when prescribing t_n *only* (a pressure BC), it is OK to apply the same *Traction* BC to different element faces.

It is an error to provide both option `CoordinateFrame` and `LoadFunctions`.

`Traction` **Example:**

```
BEGIN BOUNDARY
 Traction STrac_name {
  Components =  100, 0, 100
 }
END

BEGIN BCASSIGN
 3  0  STrac_name
END
```

In the boundary class definition above, an arbitrary name, *STrac_name* (standing for Surface Traction name), is assigned to the boundary class, `Traction`, with one set of arguments. *STrac_name* is subsequently assigned in the `BCASSIGN` packet to the face of a given element by specifying the element identification number (*element_id* 3), and the face number (*face_id* 0). See Section 2.4 for the face numbering scheme used in *ISET*.

The `Components` argument represents three components of the surface traction vector. The components of the traction vector are assumed to occur in the basic Cartesian (X,Y,Z) coordinate system.

The following example illustrates the application of a Lagrangian surface traction called *tension* with a magnitude of 10000.0 in the y direction, acting on face 3 of element number 1:

```
BEGIN BOUNDARY
 Traction tension  {
  Components =  0, 10000, 0
 }
END

BEGIN BCASSIGN
 1 3 tension
END
```

In this example, the surface traction components are assumed to occur in the basic (X,Y,Z) system.

`TractionAbaqusUserSub` **Example:**

```
BEGIN BOUNDARY
 TractionAbaqusUserSub STrac_name { }
END

BEGIN BCASSIGN
 3  0  STrac_name
END
```

In the boundary class definition above, an arbitrary name, *STrac_name* (standing for Surface Traction name), is assigned to the boundary class, `TractionAbaqusUserSub`, with no arguments. *STrac_name* is subsequently assigned in the `BCASSIGN` packet to the face of a given element by specifying the element identification number (*element_id* 3), and the face number (*face_id* 0). See Section 2.4 for the face numbering scheme used in *ISET*.

The components of the surface traction vector is given by the Abaqus user subroutine DLOAD compiled in the shared library `libabaqususersubs.so`. Before using a `TractionAbaqusUserSub` boundary condition in *ISET*, the user must compile the corresponding FORTRAN DLOAD subroutine using the following command (single line issued in folder with file `file_with_DLOAD.f`)

```
gfortran -cpp -shared -fPIC {file_with_DLOAD.f} \
         -o {full_path_to_iset_dyn_lib_folder}/libabaqususersubs.so
```

The above command compiles FORTRAN file `file_with_DLOAD.f` and generates a shared library named `libabaqususersubs.so`.

If other Abaqus subroutines are also used, they should be compiled along with the `file_with_DLOAD.f` to generate the shared library. For instance, if the user subroutine UMAT from Section 2.6.4 and DFLUX from Section 2.17.1 are used, the command to generate the shared library will be

```
gfortran -cpp -shared -fPIC \
         {file_with_UMAT.f} {file_with_DLOAD.f} {file_with_DFLUX.f} \
         -o {full_path_to_iset_dyn_lib_folder}/libabaqususersubs.so
```

R The compilation flag `-DABAQUS_USER_SUBS` must be defined when compiling *ISET* in order to activate the support for Abaqus user subroutines. This is done by setting `CMake` variable `ISET_COMPILE_FORTRAN_FILES` to true while in `CMake`. This is only relevant for those users that have the source code of *ISET*.

2.16.2 Prescribed Displacements: `Displacement`

Face based displacement boundary conditions in *ISET* are specified using the following format:

```
Displacement disp_name { Components = u v [ w ]
                         Flags = flag_x flag_y [ flag_z ] }
```

This format for the prescribed displacement boundary condition assignment is very similar, in terms of key words, to the traction boundary conditions. The boundary class assignment `Displacement` is followed by an arbitrary character name and a list of arguments which includes the three `Components` of the displacement (*u, v, w*), three activation flags (*flag1, flag2, flag3*) which are the constraint indicators. The activation flags may be set to either 0, or 1. Zero is 'off' (i.e., ignore the constraint for this component). One is 'on' which means to use the value specified.

The following example illustrates the definition of a prescribed displacement called *fixed* which enforces a zero displacement in the *x* and *y* directions and 0.1 displacement in the *z* direction.

```
BEGIN BOUNDARY
 Displacement fixed {
  Components = 0.0, 0.0, 0.0
  Flags = 1, 1, 1
 }
END
```

The assignment of the above object to face *3* of element *2* can be done using

```
BEGIN BCASSIGN
 2 3 fixed
END
```

2.16.3 Cauchy or Spring Type Boundary Conditions: `Spring`

```
Spring   spr_name { Springs = Kx Ky [Kz]  Displacements = Dx Dy [Dz] }
```

Displacements are displacements applyed to the springs and are taken as (0.0,0.0,0.0) if not given. Note that we do NOT read coupling terms for Spring matrices.

TBD: Describe this object

2.16.4 Cohesive Boundary Conditions: `Cohesive`

This boundary condition implements a cohesive fracture law. The syntax is as follows

```
Cohesive  coh_name { CohesiveLawParm = #parm_1 #parm_2 ... #parm_11 }
```

Eleven parameters must be given after the keyword `CohesiveLawParm`. Their meaning are as follows

- `param_1` = Global direction to define a region in which crack is stress-free: 0: x, 1: y, 2: z
- `param_2` = Global coordinate below which the crack is stress-free.
- `param_3` = Type of cohesive model: 1: linear, 2: bilinear, 3: ZuNeedleman, 4: PPR.
- `param_4` = Cohesive strength in normal direction.
- `param_5` = Cohesive strength in tangential direction.
- `param_6` = Fracture energy in normal direction.
- `param_7` = Fracture energy in tangential direction.
- `param_8` = Parameter associated with kink point (bilinear law) or shape parameter (PPR law) in normal direction.

- `param_9` = Parameter associated with kink point (bilinear law) or shape parameter (PPR law) in tangential direction.
- `param_10` = Parameter associated with final separation (bilinear law) or slope indicator (PPR law) in normal direction.
- `param_11` = Parameter associated with final separation (bilinear law) or slope indicator (PPR law) in tangential direction.

R Even if a particular cohesive law does not use all parameters listed above, a zero value must be given. This is admittedly a poor design and it is expected to be improved in the future.

R `Cohesive` boundary conditions are defined in a `.grf` file but usually only referenced in a `Tcl` file where it is assigned to a `crackMgr` object using, for example,

```
crackMgr crackMgrID 1 setOptions useCohesive CohesiveBCName
```

where `CohesiveBCName` is defined in a `.grf` file using the syntax described above. Further details are provided in Section 4.5.5.

2.17 Face Boundary Conditions for Heat-Type Problems

2.17.1 Normal Fluxes: `Flux`, `Flux2D`, `FluxAbaqusUserSub` and `Flux2DAbaqusUserSub`

Face based normal fluxes in *ISET* are specified using the following formats.

```
Flux NFname  { NormalFlux = q  |
               FluxFunctionNum = flux_num  |
               UserFunction = func_name  pOrder = func_p_ord
               [rhs = rhs_number] }

Flux2D NFname  { NormalFlux = q |
                 FluxFunctionNum = flux_num
                 [rhs = rhs_number] }

FluxAbaqusUserSub NFname  { [rhs = rhs_number] }

Flux2DAbaqusUserSub NFname  { [rhs = rhs_number] }
```

where

- `FluxFunctionNum` is the number of a hardwired function used to compute the normal flux to an element face.
- `UserFunction` is the name of a Python function used to compute the normal flux. This is a non-constant function of (x, y, z) and potentially time, defined in a Python script.
- `pOrder` is the p-order equivalent to the user-prescribed boundary condition function `UserFunction`. This is used for numerical integration over an element face.
- Paramenter `rhs` is taken as zero if not given.

R Option `UserFunction` can only be used with *ISETPy*, a Python interface to *ISET*.

`Flux` **and** `Flux2D`

In the definition above, a user-defined name, *NFname* (standing for Normal Flux name), is assigned to the boundary object of class `Flux`/`Flux2D`. `Flux` and `Flux2D` are used in 3- and 2-D problems, respectively. Object *NFname* can subsequently be assigned in the `BCASSIGN` packet to the face of a given element by specifying the element identification number (*element_id*), and the face number (*face_id*). See Section 2.4 for the face numbering scheme used in *ISET*. The `NormalFlux` argument represents the value of the normal flux, `q`.

The following example illustrates the application of a normal flux called *flux* with a magnitude of 10.0, acting on face 3 of element number 1:

```
BEGIN BOUNDARY
 Flux flux {
  NormalFlux = 10
 }
END

BEGIN BCASSIGN
 1 3 flux
END
```

`FluxAbaqusUserSub` **and** `Flux2DAbaqusUserSub` **Example:**

The following example illustrates the application of a FluxAbaqusUserSub boundary condition acting on face 3 of element 1.

```
BEGIN BOUNDARY
 FluxAbaqusUserSub flux { }
END

BEGIN BCASSIGN
 1 3 flux
END
```

The normal flux is given by Abaqus user subroutine DFLUX. Before using an `FluxAbaqusUserSub` or `Flux2DAbaqusUserSub` boundary condition in *ISET*, the user must compile the corresponding FORTRAN DFLUX subroutine using the following command (single line issued in folder with file `file_with_DFLUX.f`)

```
gfortran -cpp -shared -fPIC {file_with_DFLUX.f} \
         -o {full_path_to_iset_dyn_lib_folder}/libabaqususersubs.so
```

The above command compiles FORTRAN file `file_with_DFLUX.f` and generates a shared library named `libabaqususersubs.so`.

If other Abaqus subroutines are also used, they should be compiled along with the `file_with_DFLUX.f` to generate the shared library. For instance, if the user subroutine UMAT from Section 2.6.4 and DLOAD from Section 2.16.1 are used, the command to generate the shared library will be

```
gfortran -cpp -shared -fPIC \
         {file_with_UMAT.f} {file_with_DLOAD.f} {file_with_DFLUX.f} \
         -o {full_path_to_iset_dyn_lib_folder}/libabaqususersubs.so
```

R The compilation flag `-DABAQUS_USER_SUBS` must be defined when compiling *ISET* in order to activate the support for Abaqus user subroutines. This is done by setting `CMake` variable `ISET_COMPILE_FORTRAN_FILES` to true while in `CMake`. This is only relevant for those users that have the source code of *ISET*.

2.17.2 Prescribed Temperatures: `Temperature` and `Temperature2D`

Face based temperature boundary conditions in *ISET* are specified using the following format:

```
Temperature PTname { Temperature = u }

Temperature2D PTname { Temperature = u }
```

This format for the prescribed temperature boundary condition assignment is very similar, in terms of key words, to the normal flux boundary conditions. The boundary class assignment `Temperature` or `Temperature2D` is followed by an arbitrary character name and an argument which consists of value of the prescribed `Temperature`. `Temperature` and `Temperature2D` are used in 3- and 2-D problems, respectively.

The following example illustrates the definition of a prescribed temperature called *SetTemp* which enforces a zero temperature on a 3-D problem. The boundary assignment is done elsewhere.

```
BEGIN BOUNDARY
 Temperature SetTemp {
  Temperature = 0.0
 }
END
```

The assignment of the above object to face *3* of element *2* can be done using

```
BEGIN BCASSIGN
 2 3 SetTemp
END
```

2.17.3 Convection-Type Boundary Conditions: `Convection` and `Convection2D`

Face based convective boundary conditions in *ISET* are specified using the following format:

```
Convection  conv_name { ConvCoeff = beta  Temperature = u0 }

Convection2D  conv_name { ConvCoeff = beta  Temperature = u0 }
```

`Convection` and `Convection2D` objects provide data for the following boundary condition in 3- and 2-D problems, respectively

$$\frac{\partial u}{\partial n} = \beta(u - u_0)$$

where

- n is the direction normal to the boundary of the domain;
- β is the convection coeffience of the medium surrounding the body;
- u is the body temperature at the boundary;
- u_0 is the ambient temperature;

2.18 Face Boundary Conditions for Thermo-elasticity-Type Problems

2.18.1 Traction

2.18.2 Displacement

2.18.3 Spring

2.18.4 Flux

2.18.5 Temperature

2.18.6 Convection

2.18.7 ThermoElasBC

2.18.8 ThermoPlasBC

2.19 Face Boundary Conditions for Newtonian Fluid and Lubrication Equation Problems

2.19.1 Normal Fluxes: `Flux` and `Flux2D`

```
Flux flux_name  { NormalFlux = q | FluxFunctioNum = #Number [rhs = rhs_number] }


Flux2D flux_name  { NormalFlux = q | FluxFunctioNum = #Number [rhs = rhs_number] }
```

2.19.2 Prescribed Pressure: `Pressure` and `Pressure2D`

```
Pressure disp_name { Pressure = u }

Pressure2D disp_name { Pressure = u }
```

2.20 Face Boundary Conditions for Krauklis Wave Problems

2.20.1 Prescribed Pressure: `Pressure`

```
Pressure press_name { Pressure = u | PressureFunctionNum = fn_num }
```

2.20.2 Normal Fluxes: `Flux`

```
Flux flux_name  { NormalFlux = q }
```

2.21 Point Loads and Point Constraints

In *ISET*, point "boundary conditions" (forces, displacements, and temperatures) are defined using the keywords `PtForce`, `PtDisplacement`, `PtSpring` and `PtTemperature`.

These "boundary conditions" are defined in the `phlex` input file starting with the keywords `BEGIN BOUNDARY` and terminating with `END`.

Point boundary conditions in general represent singularities. Therefore, special care should be exercised when using this class of problem data due to unbounded stresses/fluxes which ultimately accompany such specifications in 2- and 3-D problems. They are therefore inconsistent unless carefully applied to, for example, the faces of a linear element.

2.22 Solid Mechanics Point Boundary Conditions

The following general formats define the keywords associated with the three classes of pointwise problem data:

2.22.1 Point Forces: `PtForce` and `PtForceAbaqusUserSub`

```
PtForce PtForce_name { Node = nod#
                       Force = p_x p_y p_z }

PtForceAbaqusUserSub PtForce_name { Node = nod#}
```

WARNING: Point force always applied to load vector 0. NO multiple loads support!

`PtForce` **Example:**

```
BEGIN BOUNDARY
 PtForce force_name {
  Node = node_number
  Force = F1,F2,F3 }
END
```

The keyword `PtForce`, must be followed by a unique name and a list of arguments. The character name is an identifier used internally by *ISET*. The nodal point at which the point data is applied must be specified. The terms F_1, F_2, F_3, are the components of the applied force in the global (X, Y, Z) coordinate frame.

In the following example a point force called *load* is applied at Node number 101. The magnitude of the load is 1000.0 in the z direction

```
BEGIN BOUNDARY
 PtForce load {
  Node = 101
  Force = 0.0,0.0,1000.0
 }
END
```

`PtForceAbaqusUserSub` **Example:**

```
BEGIN BOUNDARY
 PtForceAbaqusUserSub load {
  Node = 101
 }
END
```

R It is currently assumed that the point force is only applied in the global X direction.

The point force will be given by the Abaqus user subroutine DLOAD. Before using an `PtForceAbaqusUserSub` boundary condition in *ISET*, the user must compile the corresponding FORTRAN DLOAD subroutine using the following command (single line issued in folder with file `file_with_DLOAD.f`)

```
gfortran -cpp -shared -fPIC {file_with_DLOAD.f} \
         -o {full_path_to_iset_dyn_lib_folder}/libabaqususersubs.so
```

The above command compiles FORTRAN file `file_with_DLOAD.f` and generates a shared library named `libabaqususersubs.so`.

If other Abaqus subroutines are also used, they should be compiled along with the `file_with_DLOAD.f` to generate the shared library. For instance, if the user subroutine UMAT from Section 2.6.4 and DFLUX from Section 2.17.1 are used, the command to generate the shared library will be

```
gfortran -cpp -shared -fPIC \
         {file_with_UMAT.f} {file_with_DLOAD.f} {file_with_DFLUX.f} \
         -o {full_path_to_iset_dyn_lib_folder}/libabaqususersubs.so
```

R The compilation flag `-DABAQUS_USER_SUBS` must be defined when compiling *ISET* in order to activate the support for Abaqus user subroutines. This is done by setting `CMake` variable `ISET_COMPILE_FORTRAN_FILES` to true while in `CMake`. This is only relevant for those users that have the source code of *ISET*.

2.22.2 Point Displacements: `PtDisplacement`

```
PtDisplacement PtDisp_name { Node      = nod#
                             Displacement = u v [ w ]
                             Flags = flag_x flag_y [ flag_z ] }
```

```
BEGIN BOUNDARY
 PtDisplacement disp_name {
  Node = node_number
  Displacement = D1 D2 [D3]
  Flags = flag1, flag2, flag3 }
END
```

The keywords `PtDisplacement` must be followed by a unique name and a list of arguments. The character name is an identifier used internally by *ISET*. The nodal point at which the point data is applied must be specified. The terms D_1, D_2, D_3 are the components of the applied displacement in the global XYZ coordinate frame. The terms $flag1, flag2, flag3$ are either 0 (for 'off'), or 1 (for 'on'). They are the constraint indicators for the displacement components. The value of the displacement is ignored (i.e., treated as free) if the associated *flag* component is turned off. The components of the pointwise vector are assumed to occur in the basic Cartesian (X, Y, Z) coordinate system.

The final example shows a point displacement called *support* that is applied at Node 210. The magnitude of the displacement is zero in the x and y directions only. The specified displacement in the z direction is ignored because the associated flag is set to zero.

```
BEGIN BOUNDARY
 PtDisplacement support {
  Node = 210
  Displacement = 0.0,0.0,2.0
  Flags = 1,1,0
 }
END
```

2.22.3 Point Springs: `PtSpring`

`PtSpring` is NOT implemented yet. Please implement and update this section.

2.23 Heat-Type Point Boundary Conditions

`PtTemperature, PtTemperature2D, PtFlux, PtConvection;`

2.23.1 Point Temperature: `PtTemperature` and `PtTemperature2D`

```
PtTemperature PtTemp_name { Node        = nod#
                            Temperature = u }

PtTemperature2D PtTemp_name { Node      = nod#
                              Temperature = u }
```

```
BEGIN BOUNDARY
 PtTemperature temp_name {
  Node = node_number
  Temperature = T1  }

END
```

The next example shows a point temperature. The prescribed point temperature, $temp1$, has a value of zero value at node 101.

```
BEGIN BOUNDARY
 PtTemperature temp1 {
  Node = 101
  Temperature = 0.0 }

END
```

2.23.2 Point Flux: `PtFlux`

`PtFlux` is NOT implemented. Please implement and update this section.

2.23.3 PtConvection: `PtConvection`

`PtConvection` is NOT implemented yet. Please implement and update this section.

2.24 Point Boundary Conditions for Thermo-Elasticity Problems

`PtDisplacement, PtForce, PtSpring; PtTemperature, PtFlux, PtConvection;`

2.24.1 PtDisplacement

2.24.2 PtForce

2.24.3 PtSpring

2.24.4 PtTemperature

2.24.5 PtFlux

2.24.6 PtConvection

2.25 Point Boundary Conditions for Newtonian Fluid and Lubrication Equation

`PtPressure, PtPressure2D, PtFlux;`

2.25.1 Prescribed Point Pressure: `PtPressure` and `PtPressure2D`

```
PtPressure PtPressure_name { Node      = nod#
                             Pressure = u }

PtPressure2D PtPressure_name { Node      = nod#
                               Pressure = u }
```

2.25.2 Prescribed Point Flux: `PtFlux`

`PtFlux` is NOT implemented. Please implement and update this section.

2.26 Point Boundary Conditions for Krauklis Wave Problems

`PtPressure, PtFlux`

2.26.1 Prescribed Point Pressure: `PtPressure`

```
PtPressure PtPressure_name { Node      = nod#
                             {Pressure = u | PressureFunctionNum = fn_num} }
```

2.26.2 Prescribed Point Flux: PtFlux

```
PtFlux PtFlux_name { Node      = nod#
                    {Flux = u | Flow = u
                     | FluxFromFile = "filename" | FlowFromFile = "filename"} }
```

The file with name `filename` should be a text file formatted in two columns, where the first column has times and the second column has fluxes/flows. An example of the file is given next:

```
0.0      0.0
1.0      10.0
1.25    31.1
4.5      2.0
5.31    0.12
```

The code will automatically use linear interpolation/extrapolation based on the data provided in the file if needed.

2.27 Point Boundary Conditions for Hagen-Poiseuille Equation

`PtPressure, PtPressurePipe, PtFlux, PtFluxPipe, PtFlowPipe`

2.27.1 Prescribed Point Pressure: PtPressure and PtPressurePipe

```
{PtPressure | PtPressurePipe}  PtPressure_name { Node      = nod#
                                                 Pressure = u }
```

2.27.2 Prescribed Point Flux: PtFlux, PtFluxPipe, and PtFlowPipe

```
{PtFlux | PtFluxPipe | PtFlowPipe}  PtFlux_name { Node  = nod#
                                                 {Flux | Flow} = u
                                                 {Compliance | U} = compl
                                                 {FluxFromFile | FlowFromFile} = filename }
```

The file with name `filename` should be a text file formatted in two columns, where the first column has times and the second column has fluxes/flows. An example of the file can be seen for the flow boundary condition `PtFlux` for Krauklis wave problems.

When solving an uncoupled pipe problem the flow boundary condition must be imposed using the command `PtFlux`. However, if it is a coupled problem - for instance hydraulic fracturing propagation with wellbore - the flow boundary condition on the pipe must be set using either `PtFluxPipe` or `PtFlowPipe`.

2.28 PROBLEMDATA Packet

The `PROBLEMDATA` packet is used to define various classes of problem data that fall somewhat outside the general scope covered by the remainder of the input data packets. In *ISET*, the `PROBLEMDATA` packet includes;

```
EdgeElas2D,
EdgeElasHO2D,
EdgeElas3D,
EdgeElasShadow,
```

StepFunCyl2D,
StepFunCyl3D,
StepFunCylGroup2D,
StepFunCylGroup3D, and
SBGeneric.

2.28.1 Generic Special Basis: SBGeneric

```
SBGeneric ID { max_n_terms = # max_n_terms
               nStateVar = # nStateVar
               dim = #dim
               equiv_p_order = # equiv_p_order
               fn_num = # fn_num}
```

- ID = ID of this Generic Special Basis Object.
- max_n_terms = max number of terms, default of 1.
- nStateVar = number of state variables associate with material.
- dim = number of integrable dimensions.
- equiv_p_order = equivalent p-order used for integrating function.
- fn_num = function number to hard-wired function in .C file.

Example:

```
PROBLEMDATA
 SBGeneric 1 {nStateVar = 1, dim = 3, max_n_terms = 1,
equiv_p_order = 17, fn_num = 7}
END
```

2.28.2 Crack Tip Enrichment: EdgeElas2D and EdgeElasHO2D

Crack tip enrichment function for 2-D elasticity problems.

```
EdgeElas2D ID { [MaxNumTerms = max_n_terms]
                [Opening     = alpha_degress]
              NodeOrigin     = origin_ID
              NodeXAxis      = x_axis_ID }
```

- ID = ID of this edge (integer). It can be used in tcl commands.
- MaxNumTerms = maximum num terms in the asymptotic expansion to use. It is optional and it is taken equal to 1 if not given.
- Opening = opening of the edge in degrees. It is optional and it is taken as 360 degrees (crack) if not given.
- NodeOrig = ID of the node at the edge tip.
- NodeXAxis = ID of any node on the positive x-axis of the coordinate system associated with the edge.

```
EdgeElasHO2D ID { [MaxNumTerms = max_n_terms]
                  NodeOrigin     = origin_ID
                  NodeXAxis      = x_axis_ID }
```

R Objects of class EdgeElasHO2D use the *first* term of the asymptotic expansion multiplied by constant, linear, quadratic, etc., polynomials. The order of the polynomials is equal to MaxNumTerms - 1. Default value of MaxNumTerms is 1. This parameter of the object can be changed using Tcl command setSpBasis, as described in Section 4.2.9.

- MaxNumTerms = Order + 1 of branch function enrichment.

2.28.3 Crack Front Enrichment: EdgeElas3D

Crack front enrichment function for 3-D elasticity problems. NOTE: This is used only for very simple tests. Enrichment functions for fractures are automatically created and assigned to nodes by a Crack Manager.

```
EdgeElas3D ID { [MaxNumTerms = max_n_terms]
                [Opening     = alpha_degress]
                [StressState = PlaneStress | PlaneStrain]
              NodeOrig     = origin_ID
              NodeZAxis    = z_axis_ID
              NodeXZPlane  = xz_plane_ID }
```

- ID = ID of this edge (integer). It can be used in tcl commands.
- MaxNumTerms = maximum num terms in the asymptotic expansion to use. It is optional and it is taken equal to 1 if not given.
- Opening = opening of the edge in degrees. It is optional and it is taken as 360 degrees (crack) if not given.
- StressState = optional. It is taken as PlaneStrain if not given.
- NodeOrig = ID of any node at the edge.
- NodeZAxis = ID of any node on the positive z-axis of the coordinate system associated with the edge.
- NodeXZPlane = ID of any node on the x-z plane of the edge coordinate system (a point in the positive x-axis, for example)

2.28.4 Crack Front Enrichment with Shadow Functions: EdgeElasShadow

NOTE: This is used only for very simple tests. Enrichment functions for fractures are automatically created and assigned to nodes by a Crack Manager.

```
EdgeElasShadow ID { [MaxNumTerms = max_n_terms]

              Thickness = t_z
              NodeOrig     = origin_ID
              NodeZAxis    = z_axis_ID
              NodeXZPlane  = xz_plane_ID }
```

- ID = ID of this edge (integer). It can be used in tcl commands.
- MaxNumTerms = maximum num terms in the asymptotic expansion to use. It is optional and it is taken equal to 1 if not given. CAD NOTE: I think MaxNumTerms can not be greater than 1.
- Thickness = Thickness of the domain in z-direction.
- NodeOrig = ID of any node at the edge.
- NodeZAxis = ID of any node on the positive z-axis of the coordinate system associated with the edge.
- NodeXZPlane = ID of any node on the x-z plane of the edge coordinate system (a point in the positive x-axis, for example)

2.28.5 StepFunCyl2D

```
StepFunCyl2D ID {  NodeOrigin    = origin_ID
                   NodeXAxis     = x_axis_ID
                   Theta_0       = theta_0
                   Theta_1       = theta_1 }
```

- ID = ID of this Step Function (integer). It can be used in tcl commands.
- NodeOrig = ID of the node.
- NodeXAxis = ID of any node on the positive x-axis of the coordinate system associated with the edge.
- Theta_0 = see file sbstepfuncyl2d.h. The angle is in degrees.
- Theta_1 = see file sbstepfuncyl2d.h. The angle is in degrees.

2.28.6 StepFunCyl3D

```
StepFunCyl3D ID { NodeOrig     = origin_ID
                  NodeZAxis    = z_axis_ID
                  NodeXZPlane  = xz_plane_ID
                  Theta_0      = theta_0
                  Theta_1      = theta_1 }
```

- ID = ID of this Step Function (integer). It can be used in tcl commands.
- NodeOrig = ID of any node.
- NodeZAxis = ID of any node on the positive z-axis of the coordinate system associated with the origin.
- NodeXZPlane = ID of any node on the x-z plane of the origin coordinate system (a point in the positive x-axis, for example)
- Theta_0 = see file sbstepfuncyl3d.h. The angle is in degrees.
- Theta_1 = see file sbstepfuncyl3d.h. The angle is in degrees.

2.28.7 StepFunCylGroup2D

```
StepFunCylGroup2D ID { SpBasisID    = SpBasis_ID_1,
                       SpBasisID    = SpBasis_ID_2,
                       ...,
                       SpBasisID    = SpBasis_ID_n }
```

- ID = ID of this Step Function (integer). It can be used in tcl commands.
- SpBasisID = ID of any StepFunCyl2D previously defined. Any number of SpBasisID can be given.

2.28.8 StepFunCylGroup3D

```
StepFunCylGroup3D ID { SpBasisID    = SpBasis_ID_1,
                       SpBasisID    = SpBasis_ID_2,
                       ...,
                       SpBasisID    = SpBasis_ID_n }
```

- ID = ID of this Step Function (integer). It can be used in tcl commands.
- SpBasisID = ID of any StepFunCyl3D previously defined. Any number of SpBasisID can be given.

2.28.9 Crack Surface Enrichment: `BranchStepPair`

Object `BranchStepPair` defines a Branch Function for 2-D or 3-D fracture problems as a pair of functions: A Branch Function and a Step Function. This is typically used with geometrical enrichment at nodes whose support is intersected by the crack surface/line but not the crack front/tip.

```
BranchStepPair ID { BranchFnID = SpBasis_ID_1,
                    StepFnID   = SpBasis_ID_2 }
```

- `ID` = ID of `SpecialBasis` function to be created. It can be used in `Tcl` commands.
- `BranchFnID` = ID of any Branch Function previous defined in `.grf`. Supported branch functions are: `EdgeElas2D`, `EdgeElasHO2D`, `EdgeElas3D`, `EdgeElasShadow`.
- `StepFnID` = ID of any Step Function previous defined in `.grf`. Supported Step Functions are: `StepFunCyl2D` and `StepFunCyl3D`.

2.29 SYSTEMS Packet

The `SYSTEMS` packet is used to define coordinate systems. It is currently under development. [heading=subbibliography]

3. List of *ISET*Tcl commands

This chapter contains a list of all Tcl commands implemented in *ISET*. It can be used to quickly check the syntax of *ISET* Tcl commands. Chapter 4 provides descriptions and examples of usage.

```
Syntax Conventions:

[ ]   Encloses optional parameters
{ }   Encloses mandatory parameters
 |    Separates two or more items within brackets or braces, only one of
      which may be chosen
```

3.1 Physics-Independent Tcl Commands

```
readFile {file_name} {phfile | restart}
         [ [solids] | heat | fluids | thermomech | hydroFrac
           | phasefieldelas | krauklisfluid | krauklis
           | pipe | pipelub ]

readGlobalMatrixAndRHS [ abaqus | iset ]

createAnalysis [ multiPhys {heat | thermoelas | thermoplas} ]
               [ linear | linearReAnalysis [ Rebuild_F0 ] ]

analysis [localProb {probID #}]
         [multiPhys {heat | thermoelas | thermoplas}]
                [ create [linear | matNonlinear | matVisco | transient
                           | transientReAnalysis | nonlinearCohesive
                           | coupledHydroFracAnalysis | krauklisAnalysis
                           | phaseFieldAnalysis ]
```

```
[ solOption [ numTimeSteps {# steps}
            | simulationTime {time} ]
 [ nonSolOption [ solAlgorithm [newton | modifiedNewton]
                | numTimeSteps {# steps}
                | convCriterion { L2Norm | residual
                                | strainEnergy }
                | convTolerance {# tol}
                | absTolerance {# tol}
                | thresholdTolerance {# tol}
                | maxNumNonlinIters {# iters} ]
                | resetStateVars
                    [{true | on | 1} | {false | off | 0}]
                | loadIncrType [uniform | locUniform |
                                locProb | constant |
                                ramp | creep | hat | freq |
                                impulse | userDef ]
                | isLineSearch
                    [{true | on | 1} | {false | off | 0}]
                | simulationTime {time}
                | peakTime {ptime}
                | isShapeFnTimeDepen
                | isIncrementalSol
                    [{true | on | 1} | {false | off | 0}]
| nonlinearSolve [ allLocalProbs ] [ timeStep {# istep} ]
| nonlinearAssemble [ allLocalProbs ] { timeStep {# istep} }
| linearSolve [ allLocalProbs ] [ timeStep {# istep} ]
]
[ transSolOption [ transSolAlgorithm [ alphaMethod
                                     | newmarkMethod
                                     | centralDifference
                                     | krauklisTimeInt ]
                | transNumTimeSteps {# steps}
                | transStopTime {# tFinal}
                | initialTime {# tInitial}
                | isShapeFnTimeDepen
                    [{true | on | 1} | {false | off | 0}]
                | useLumpedMassForL2Proj
                    [{true | on | 1} | {false | off | 0}]
                | useLumpedMassForTransAnalysis
                    [{true | on | 1} | {false | off | 0}]
                | saveTransFactors
                    [{true | on | 1} | {false | off | 0}]
                | transSetParameters [ alpha {# alpha}
                                     | beta  {# beta }
                                     | gamma {# gamma} ]
                | accelerationBased
                    [{true | on | 1} | {false | off | 0}]
                  ]
```

```
| coupledHydroFracSolOption [ maxNumNewtonIters {# iters}
                    | timeIncrement {# deltaT}
                    | minTimeStep {# tmin}
                    | maxTimeStep {# tmax}
                    | convTolerance {# tol}
                    | totalSimulationTime {# tfinal}
                    | propagatingCrack {true | false}
                    | restart {true | false}
                    | startTime {# init_time}
                    | assignCrackMgr {# crackMgrID} ]
                    | parametersDCM { extrDist_min {extrDist_min #}
                                      deltaDist {deltaDist #}
                                      numExtrPoints {numExtrPoints #}
                                    }
                    | minOpeningAllowed {# wmin}
            | initPropagStepWithSmallVisco factorSmaller {# factor}
            | useFillOfFractureWithSmallVisco factorSmaller {# factor}
                                              fillingTime {# time}
                    | pOrdForFluidEls {# pOrd}
                    | identifyAndSetCornerNodesToHalfInj {true | false}
                    | oneDeltaTPerDeltaA {true | false}
                    | leakOffCarterCoeff {#}
                    | spurtLoss {#}
                    | initLeakOffExposureTime {#}
                   ]

| coupledHydroFracSolve
| coupledHydroFracSolveAndPropagate

| krauklisSolOption [ pOrdForFluidEls {# pOrd} ]
                      neglectSolidInertia {#}
                      solveCtePressProblemForIniOpening {#}
                      constantW0 {# ctew0} ]
                      minimumW0 {# minimumW0} ]
| krauklisComputePhaseVelAndAttenuation {initCoord {# x} {# y} {# z}
                                         direction {# x} {# y} {# z}
                                         dist {#} npts {#} }
| phaseFieldSolOption [initializePFSol]
| quasiStaticSolve [Time {# Time}]
| transientSolve  {# Time}
| hardCodeProjectSolutionForP1
| L2ProjectInitialCond [{approxOrder #}]

| transAssembleReAn {# Time}
| transSolveReAn
| transSolveReAnFast
| setParallelAssemble

| outputOption [ saveResultsFreq {# freq}
```

```
                              | extractResultsFreq {# freq}
                              | graphicalResol {# resol} ]

                | writeSolution                 filename [binary | ascii]
                | processSolutionInitialProblem filename [binary | ascii]
                | computeDumpPseudoLoads        filename [binary | ascii]
                | readPseudoLoads               filename [binary | ascii]
                | computeDumpPseudoSolutions    filename [binary | ascii]
                | readPseudoSolutions           filename [binary | ascii]
                | solveEnrichedProbReAnalysisTwoSolvers
                | sortLocalProbs {ascend | descend}
                ]

enrichApprox  [localProb {probID #}]
              [multiPhys {heat | thermoelas | thermoplas}]
              [iso | xi | eta | zeta] approxOrder {order #}

enrichCompNod [localProb {probID #}] [multiPhys {heat | thermoelas | thermoplas}]
              [all [ hasStepFn | hasBranchFn | hasLocalSol ]
              | cnode {cnodeID #}
              | onSegment A {Ax #} {Ay #} {Az #}
                          B {Bx #} {By #} {Bz #}
              | inBBox xyzMin {Xmin #} {Ymin #} {Zmin #}
                       xyzMax {Xmax #} {Ymax #} {Zmax #}
              | inCylz center {x #} {y #} {z #}
                       radius {r #}
                       height {h #}             ]
              [nodeType {vertex | edge | face
                         | interior | any}]
              [iso | xi | eta | zeta] order {order #}

setCompNodSpBasis [localProb {probID #}]
                  [multiPhys {heat | thermoelas | thermoplas}]
                  [ all
                  | cnode {cnodeID #}
                  | locProb
                  | onSegment A {Ax #} {Ay #} {Az #}
                              B {Bx #} {By #} {Bz #}
                  | inBBox xyzMin {xMin #} {yMin #} {zMin #}
                           xyzMax {xMax #} {yMax #} {zMax #}
                  | inCylz center {x #} {y #} {z #}
                           radius {r #}
                           height {h #}          ]
                  [nodeType {vertex | edge | face
                             | interior | any}]
                  spBasis {spBasisID #}

assignPtBC  [localProb {probID #}]
            [Elas2D | Elas3D | Plas3D | ViscoElas3D]
```

```
            [cnode {cnodeID #}
            |inBBox xyzMin {Xmin #} {Ymin #} {Zmin #}
                    xyzMax {Xmax #} {Ymax #} {Zmax #}]
            PtDisplacement
            Displacement {ubar_x} {ubar_y} {ubar_z}
            Flags {flag_x} {flag_y} {flag_z}

setSpBasis [localProb {probID #}]
           [multiPhys {heat | thermoelas | thermoplas}]
           [ all { hasStepFn | hasBranchFn | hasLocalSol }
           | spBasis {spBasisID #} ]
           maxNumCustomTerms {maxTerms #}

delete [localProb {probID #}] spBasis {spBasisID #}

refine [localProb {probID #}]
       [ all
       | element {elemID #}
       | hasNode {geoNodeID #}
       | hasNodeInBBox xyzMin {Xmin #} {Ymin #} {Zmin #}
                       xyzMax {Xmax #} {Ymax #} {Zmax #}  ]

refineIfIntersect  [localProb {probID #}]
             [ [segment] A {Ax #} {Ay #} {Az #}  B {Bx #} {By #} {Bz #}
             |  plane    P {Px #} {Py #} {Pz #}  N {Nx #} {Ny #} {Nz #} ]

unRefine [localProb {probID #}]
         toInitialMesh

meshReport [localProb {probID #}]

colorMesh [localProb {probID #}] [multiPhys {heat | thermoelas | thermoplas}]
      [maxColorSize {# size}]

assemble [ localProb {probID #} ]
         [ multiPhys { heat | thermoelas | thermoplas |
                       phasefield | phasefieldelas } ]
         [ allLocalProbs
           [transient Time {# Time}] ]
         [ reAnalysis ]
         [ step {# stepNum} ]

parallelAssemble [localProb {probID #}]
                 [multiPhys {heat | thermoelas | thermoplas}]

scaleGlobalMatrix [ localProb {probID #} ]

solve [ localProb {probID #} ]
      [ multiPhys { heat | thermoelas | thermoplas |
```

```
                    phasefield | phasefieldelas } ]
      [ allLocalProbs
          [transient Time {# Time}] ]
      [ reAnalysis | reAnalysisFast | reAnalysisPCG {BGS | BJ | CG}  {#tol} |
        reAnalysisBGS {#tol} | reAnalysisScratch ]

printSolCoeff [localProb {probID #}] [all | cnode {cnodeID #}]

computeNodReaction [localProb {probID #}]  cnode {cnodeID #}

extractQuantity [multiPhys {heat | thermoelas | thermoplas} ] {Quantity}
                point {x#} {y#} {z#} [elemDim {dim#}] [reducedVerbosity {bool}]

work [localProb {probID #}] [multiPhys {heat | thermoelas}] [addIntOrder {order #}]

solInnerRHS [localProb {probID #}]

strainEnergy [localProb {probID #}]
             [multiPhys {heat | thermoelas | thermoplas}] [addIntOrder {order #}]
             [matName {Name_of_the_material}]

parallelStrainEnergy [localProb {probID #}]
                     [multiPhys {heat | thermoelas}] [addIntOrder {order #}]
                     [matName {Name_of_the_material}]

strainEnergyAndNorms [localProb {probID #}]
                     [multiPhys {heat | thermoelas}] [addIntOrder {order #}]
                     [matName {Name_of_the_material}]

parallelStrainEnergyAndNorms [localProb {probID #}]
                             [multiPhys {heat | thermoelas}] [addIntOrder {order #}]
                             [matName {Name_of_the_material}]

computeResidual [localProb {probID #}

LinfNorm [ solution | derivative | trueError | trueErrorDeriv | clear]

extractFile [localProb {probID #}]  [create | close]

graphMesh [localProb {probID #}]
          [ create {file_name}
                   [DXStyle | TecPlotStyle | HyperMeshStyle | VTKStyle]
                   [2DManifold | coupledHydroFrac | krauklis ] [ascii | binary]
          | setParallelSolution
          | open   {file_name}
          | appendName     [multiPhys {heat | thermoelas | thermoplas}]
                           [coupledHydroFrac | krauklis {solid | fluid}]
            {vec_or_scal_name}
          | appendVecName  [multiPhys {heat | thermoelas | thermoplas}]
```

```
                          [coupledHydroFrac | krauklis {solid | fluid}]
            {vec_name}
          | appendScalName [multiPhys {heat | thermoelas | thermoplas}]
                          [coupledHydroFrac | krauklis {solid | fluid}]
            {scal_name}
          | postProcess [multiPhys {heat | thermoelas | thermoplas}]
                        resolution {resolution}
                        [sampleInside] [solution {sol_number}]
          | postSpBasisEnrich {file_name}
          | postSignedDistance crackMgrID {#crackMgrID}
          ]

createExactSol [localProb {probID #}]
               [ PolySol3D porder {px #} {py #} {pz #}
                           POUBasisT [Pascal | Tensor]
                           basisT [Pascal | Tensor]

               | PolySol2D porder {px #} {py #}
                           POUBasisT [Pascal | Tensor]
                           basisT [Pascal | Tensor]

               | EdgeSol3D edgeElasID  {edgeID #}
                           [mode [ModeI | ModeII | ModeIII | ModeI_II
                                 | ModeI_II_III]
                           | setKVals KI {KI #} KII {KII #} KII {KIII #}]
                           term {term #}

               | EdgeSolShadow edgeElasID  {edgeID #}
                           mode [ModeI]
                           term {term #}

               | EdgeSol2D edgeElasID {edgeID #}
                           mode [ModeI | ModeII | ModeI_II]
                           [ term {term #} | numTerms {num_terms #} ]

               | UserFunction userFn {functionID #}
               ]

enrichExactSol [localProb {probID #}] order {px #} {py #}  [ {pz #} ]

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       exactSolAsBC [ [true | on | 1] | [false | off | 0] ]
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       exactSolAsPtBC [ [true | on | 1] | [false | off | 0] ]
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       exactSolAsBF [ [true | on | 1] | [false | off | 0] ]
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       exactSolAsIC [ [true | on | 1] | [false | off | 0] ]
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
```

```
       ignoreBCs    [ [true | on | 1] | [false | off | 0] ]
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       directMethodForPtDirichletBC [ [true | on | 1] | [false | off |0] ]

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       scaleAndStabAlgo   [ [true | on | 1] | [false | off | 0] ]
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       perturbationForBabuskaAlgorithm {epsilon #}
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       maxStabIterationsForBabuskaAlgorithm  {maxStabIterations #}
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       tolForStabilization     {tolForStab #}
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       scaleGlobalMatrixB4Fact   [ [true | on | 1] | [false | off | 0] ]

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       addPorderForInteg {addPorder #}
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       reducePorderForInteg {reducePorder #}
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       singularIntegrationOrder  {sing_p_order #}
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       lowerPorderForIntegEG [ [true | on | 1] | [false | off | 0] ]
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
        minNumDesc4LowerIntPorderEG {minNumDesc #}

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       SGFEMSpecialBasisEnrichments [ [true | on | 1] | [false | off | 0] ]
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       SGFEMBranchFnEnrichments [ [true | on | 1] | [false | off | 0] ]
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       SGFEMBranchFnEnrichmentsDiscInterp [ [true | on | 1] | [false | off | 0] ]
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       SGFEMPolyEnrichments [ [true | on | 1] | [false | off | 0] ]

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       ShiftedSpBasisEnrichments [ [true | on | 1] | [false | off | 0] ]
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       ShiftedNotBranchFnEnrichments [ [true | on | 1] | [false | off | 0] ]

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       pouDegree {p-order_pou #}

parSet [multiPhys {heat | thermoelas}]
       useCompElPfemGfem    [ [true | on | 1] | [false | off | 0] ]
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       maxElemSideDimWithBranchFnTopoEnrich  {sideDim #}
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       maxElemSideDimWithBranchFnGeomEnrich  {sideDim #}
```

```
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       maxElemSideDimWithHeavisideFnEnrich  {sideDim #}
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       maxpOrderHeavisideFnEnrich  {sideDim #}
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       useBranchHeavisideFnPairWithGeomEnrich [ [true | on | 1] | [false | off | 0] ]

Obsolete:
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       maxElemSideDimBranchHeavisidePairGeomEnrich  {sideDim #}

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       maxElemSideDimWithLocalSolEnrich {sideDim #}

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       solveResidual [ [true | on | 1] | [false | off | 0] ]

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       enforceDescSearchGlobLocBC  [ [true | on | 1] | [false | off |0] ]

parSet factorForScaledRoundOffTolerance {factor #}


parGet [localProb {probID #}] [multiPhys {heat | thermoelas}]  minEdgeLen

parGet [localProb {probID #}] [multiPhys {heat | thermoelas}]
  condNumber [numZeroEigval {Num_zero_eigenvals} | tol {toler for
                "zero" eigenvals}]

parGet [localProb {probID #}] [multiPhys {heat | thermoelas}]  normResidual

parGet [localProb {probID #}] [multiPhys {heat | thermoelas}]  lastTimeStep

parGet [localProb {probID #}] [multiPhys {heat | thermoelas}]  normResidual_IGL


getISETConfig [whichSolver isMatlab isAbaqusUserSubs]
```

3.2 Tcl Commands for a GFEM^{gl} Analysis

```
createLocalProblem probID {probID #} [[linear] | matNonlinear |
                                                 | matVisco | transient
                                                 | transientLocal ]
                    [ geoNodeID {geoNodeID #}
                    | xyzMin {Xmin #} {Ymin #} {Zmin #}
                      xyzMax {Xmax #} {Ymax #} {Zmax #}
                    | crackMgID {crackMgID #}
                       [[frontOnly] | allElems ] ]
                    | readFromFile {file_name} [[solid] | heat] ]
                     numLayers {nLayers #}
```

```
                    userBC {userBCName}

assembleTransLoc [localProb {probID #}] [multiPhys {heat}] Time {Time #}]

solveTransLoc [localProb {probID #} [multiPhys {heat}] Time {Time #} deltaT {deltaT #}]

l2ProjectInitialCondLocal [ localProb {probID #} ]
                        [ multiPhys { heat | thermoelas } ]
                        useGlobalInitCond { true | false }
                        Time { Time # }
                        deltaT { deltaT # }
                        storeSolution { true | false }

printFPOSLocProbs

MasterLocalProblem msProbID {probID #}
                   [ multiPhys {heat | thermoelas | thermoplas} ]
                   [ create  [[linear] | matNonlinear | transient]
                             [ geoNodeID {geoNodeID #}
                             | xyzMin {Xmin #} {Ymin #} {Zmin #}
                               xyzMax {Xmax #} {Ymax #} {Zmax #}
                             | readFromFile {file_name} [[solid] | heat]
                             | crackMgID {crackMgID #} ]
                             numLayers {nLayers #}
                             userBC {userBCName}
                   | defineSubProblems
                        [ oneLocProbPerCloud |
                          clusterSeedNodes |
                          numSubLocProbs {numSub #} ]
                   | defineSeedNodes [ geoNodeID {geoNodeID #}
                                     | xyzMin {Xmin #} {Ymin #} {Zmin #}
                                       xyzMax {Xmax #} {Ymax #} {Zmax #} ]
                   | defineSeedNodsCluster [ xyzMin {Xmin #} {Ymin #} {Zmin #}
                                             xyzMax {Xmax #} {Ymax #} {Zmax #} ]
                   | defineClustersWithHSFC [ numClusters {nClusters #}
                                            | xyzMin {Xmin #} {Ymin #} {Zmin #}
                                              xyzMax {Xmax #} {Ymax #} {Zmax #} ]
                   | process
                        [ noManager | crackMg | grainMg ]
                   | enrichCompNodSubProbs
                        [multiPhys {heat | thermoelas | thermoplas}]
                        all [ hasStepFn | hasBranchFn ]
                        [nodeType {vertex | edge | face
                                   | interior | any} ]
                        [iso | xi | eta | zeta] order {order #}
                   | setSpBasisSubProbs
                        all [ hasStepFn | hasBranchFn ]
                        maxNumCustomTerms {maxTerms #}
                   | setCompNodSubLocalProb
```

```
                  | sortSubProbs {ascend | descend}
                  | setParallel
                  | setAnalysisOptionsSubProbs
                       [multiPhys {heat | thermoelas | thermoplas}]
                  | assemble [multiPhys {heat | thermoelas}]
                  | solve [multiPhys {heat | thermoelas}]
                  | parallelAssemble [multiPhys {heat | thermoelas}]
                  | parallelSolve [multiPhys {heat | thermoelas}]
                  | transientAssemble Time {# Time}
                  | transientSolve Time {# Time}
                  | nonlinearSolve [multiPhys {thermoplas}]
                                   [ timeStep {# istep} ]
                  | nonlinearAssemble [multiPhys {thermoplas}]
                                   { timeStep {# istep} }
                  | linearSolve [multiPhys {thermoplas}]
                                   [ timeStep {# istep} ]
                  | graphMesh
                      [ create {file_name} [DXStyle | TecPlotStyle |
                                            HyperMeshStyle | VTKStyle]
                                           [ascii | binary]
                      | open   {file_name}
                      | postProcess resolution {resol #} [sampleInside]
                        [solution {sol_number}] ]
                      | postSurface
                      | postEnrichment
                  ]
```

3.3 Tcl Commands for an IGL-GFEM^{gl} Analysis

```
IGLAlgorithm {  create [useShellSolidCoupling {true | on | 1 | false | off | 0} ]
              | readAndProcessInterfaceData {file_name}
              | computeAndDumpInterfaceData {file_name}
              | setElemListLoc xyzMin {Xmin #} {Ymin #} {Zmin #}
                               xyzMax {Xmax #} {Ymax #} {Zmax #}
              | useShellSolidCoupling {true | on | 1 | false | off | 0}
              | readAndApplyInterfaceResForce {file_name}
              | readInterfaceResForceLoc {file_name}
              | readInterfaceResForceLocPrev {file_name}
              | computeAndDumpInterfaceResForce {file_name}
                               { NoAcceleration | DynamicRelaxation }
            }
```

3.4 Tcl Commands for a Fracture Analysis

```
crackMgr [ localProb {probID #} ]  crackMgrID {crackMgrID #}
       {  create
          { crackFile {file_name}
            |
            crackSurfaceFromCrackMgr {crackMgrIDforCreation #}
          }
```

```
| process
        [ localProb {probID #} [ frontOnly | allElems] ]
| postSurface {file_name}
        [ DXStyle | crfStyle | VTKStyle | ParaviewStyle ]
| postComputationalSurface [aboveCrack | belowCrack] {file_name}
        [ DXStyle | VTKStyle | ParaviewStyle ]
| postEnrichment {file_name}
        [ DXStyle ]
| postAllCrackCutElInfo {file_name}

| refineMesh
            { crackFronts [ maxEdgeLen {max_len#} | minEdgeLen {min_len#}
                            [inCylinderRadius {radCyl#}]
                          ]
            | crackVertices
            | crackSurface [maxEdgeLen {max_len#} [distToFront {dis_front#}] ]
            }

| enrichApprox
     {numElemLayers {nLayers #} | inCylinder radius {R #} }
     [iso | xi | eta | zeta] approxOrder {approxOrder #}

| branchFnGeometricEnrich { numElemLayers {nLayers #} | inCylinder radius {R #}
                           | inBBox xyzMin {xMin #} {yMin #} {zMin #}
                                    xyzMax {xMax #} {yMax #} {zMax #} }

| setCompNodLocalProb
     localProb {probID #}

| extract
   {     DCM extrDist_1 {extrDist_1 #} extrDist_2 {extrDist_2 #}
          [ projPntOnCrackSurface ]
    |
         DCM extrDist_min {extrDist_min #} deltaDist {deltaDist #}
             numExtrPoints {numExtrPoints #} [averLinExtrapl | leastSqrExtrapl]
          [ projPntOnCrackSurface ]
    |
         CIM radius {radius #} [numIntegPoints {nint #}]
     |
         CFM  radius_1 {rho1 #} radius_2 {rho2 #}
              cylThickness {zThickness}
             [numIntegPoints_r {nint_r #}
              numIntegPoints_t {nint_t #}]
   }
     [ tracFace+ {t_n #} {t_t #} {t_b #}
       tracFace- {t_n #} {t_t #} {t_b #} ]

     [ start_time {stime #} end_time {etime #}  nSteps {nsteps #} ]
```

```
| cleanUp { intersectData | afterPropagation }

| advanceFront
    {    DCM  extrDist_1 {extrDist_1 #} extrDist_2 {extrDist_2 #}
          [ projPntOnCrackSurface ]
     |
         DCM extrDist_min {extrDist_min #} deltaDist {deltaDist #}
            numExtrPoints {numExtrPoints #} [averLinExtrapl | leastSqrExtrapl]
          [ projPntOnCrackSurface ]
     |
         CIM radius {radius #}
            [ numIntegPoints {nint #} ]
     |
         CFM radius_1 {rho1 #} radius_2 {rho2 #}
             cylThickness {zThickness}
            [numIntegPoints_r {nint_r #}
             numIntegPoints_t {nint_t #}]
    }
      [ tracFace+ {t_n #} {t_t #} {t_b #}
        tracFace- {t_n #} {t_t #} {t_b #} ]

    [ start_time {stime #} end_time {etime #} ]

| remeshCrackSurface
| cleanUpStoredCrackFronts

| setFluidMatForCrackMgr {fluidMat_name} pressureBCForInitialGuess {bc_name}

| setConnectingPipeMatForCrackMgr {fluidMat_name}

| prescribeFluidFlow { inBBox xyzMin {xMin #} {yMin #} {zMin #}
                                xyzMax {xMax #} {yMax #} {zMax #}
                      | inSemiTorus r1 {#} r2 {#} z1 {#} z2 {#} [xc {#} yc {#}]
                      | atPoint {x #} {y #} {z #} }
                    {volume {vol #} | volumeFromFile {filename}} }

| prescribePressureKrauklis { inBBox xyzMin {xMin #} {yMin #} {zMin #}
                                       xyzMax {xMax #} {yMax #} {zMax #}
                             | inSemiTorus r1 {#} r2 {#} z1 {#} z2 {#} [xc {#} yc {#
                             | atPoint {x #} {y #} {z #} }
                          { pressure {pres #}
                          | pressureFunctNumber {fn #} [pressureFrequency {freq #}]
                          | pressureFromFile {filename} }
                            }

| connectPipeWithFrac {  inSemiTorus r1 {#} r2 {#} z1 {#} z2 {#}
          | inBBox xyzMin {Xmin #} {Ymin #} {Zmin #}
                          xyzMax {Xmax #} {Ymax #} {Zmax #}
          | atPoint {x #} {y #} {z #} }
```

```
            pipeNodeID {ID #}

      | setCornerNodes { node1 {x #} {y #} {z #} node2 {x #} {y #} {z #} }

      | {setOptions | parSet}
          {
            usePlanarCutsSignedDist |
            useBranchFn |
            useSingularBranch |

            useMLSForExtraction |
            useMLSForSkippedFrontEndVert |

            ignoreKII |
            ignoreKIII |

            useMLSForCrackFrontVertices |
            useMLSForCrackFrontVerticesWhileRemeshing |
            useUniformCrackFrontVertices |

            useSimplifyCrackSurface |
            useCompleteSimplifyCrackSurface |
            useRemeshCrackSurface|
            storeCrackFront |

            viscoExtractionCP |
            viscoFourierTransf |
            viscoForceControlled

            { [true | on | 1] | [false | off | 0] }
          |

            MLS_overlappingFactor  |

            delta_a_max            |
            frontEdgeLengthLimit   |
            crackSurfEdgeLength    |

            scaleForCrackFrontVerticalDisplacement |
            accumulatedCrackFrontAdvanceLimit |
            frontStretchTol |

            viscoPeakTime

            {double_precision_option_value #}
          |
            useViscoExtractionCP viscoMatName {name_viscoelas_mat}
          |
            viscoLoadType {constant | ramp | creep | hat | freq | haversine}
```

```
        |
          viscoIntegrType {LaplaceTransf | GaussIntegr | incremental}
        |

          MLS_degree                     |
          MLS_numSamplePointsFrontEdges |

          compSIFsAfterEveryNVerts      |
          numFrontEndVertToSkip

          {integer_option_value #}
        |

          applyBCtoCrackFaces {bc_name}
        |
          useCohesive {bc_name}
        |
          branchFunctionType { straightFrontOD | straightFrontBB | straightFrontOY
                             | curvedFrontBB | curvedFrontOD
                             | curvedFrontOD6 | curvedFrontOD9 }

        | IDNeighborMgr4Coalesce {crackMgrID #}
        | surfCoalesceTol        {tol_distance #}

        | snapNodesToSurfAndFront snappingTolSurf {tol #} [snappingTolFront {tol #}]
        | snapNodesToSurfAndFront { [true | on | 1] | [false | off | 0] }

        }

    | parGet { minElemEdgeLenAtCrackFront
             | isCrackArrested
             }

    | crGrowthPhysicsLaw
        {
          create

        | crGrowthType
          { FatigueCrackGrowth
            |
            StableCrackGrowth {Gc_material {user_defined_Gc # }}
          }

        | crFrontIncrementType
          { fixedDeltaA {us_def_incr #}
            |
            adaptiveDeltaA
                     [ setOptions
                         {
```

```
                    Alpha {#Double_value} |
                    Beta  {#Double_value} |
                    restart
                  }
              ]
  }

|
  frontKinkingAngleLimitDEG {double_precision_option_value #}
|
  frontTwistingAngleLimitDEG {double_precision_option_value #}

| print

| crFrontScalingLaw
  { ParisLaw
            { C {us_def_C #}
              m {us_def_m #}
              R {us_def_R #}
            }
    |
    MearLaw
            { aZero {us_def_aZero #}
              m {us_def_m #}
              KZero {us_def_KZero #}
              KIC {us_def_KIC #}
            }
    |
    MearLawQuasiStatic
            { m {us_def_m #}
              KIC {us_def_KIC #}
            }
    |
    FormanLaw
            { C {us_def_C #}
              m {us_def_m #}
              R {us_def_R #}
              Kth {us_def_Kth #}
              KC {us_def_KC #}
            }
    |
    MearLaw2
            { alpha {us_def_alpha #}
              beta {us_def_beta #}
              KIC {us_def_KIC #}
            }
    |
    GuptaDuarteLaw
            { KIC_material {KIC_material #}
```

```
                          alpha {alpha #}
                          [ m_alpha {m_alpha #} ]
                          beta {beta #}
                          [ m_beta {m_beta #}   ]
                        }
                        | { setVariationInAlphaRegion {true | false} }
                |
                ConstFrontDispl
                |
                setConstFrontDispl [{true | on | 1} | {false | off | 0}]

              } # end of sub-sub command crFrontScalingLaw

          } # end of crGrowthPhysicsLaw sub commands

      } # end of crackMgr sub-commands


crackMgr2D [ localProb {probID #} ]  crackMgrID {crackMgrID #}
      { create
         {  signedDistFunction {function #}

          | straightCrackEndPoints tail {x #} {y #}
                                   tip  {x #} {y #} }

      | process

      | branchFnGeometricEnrich { inCylinder radius {R #}
                                | inBBox xyzMin {xMin #} {yMin #} {zMin #}
                                        xyzMax {xMax #} {yMax #} {zMax #} }
      | {setOptions | parSet}
         {  branchFunctionType {straightFrontOD | straightFrontBB
                               | straightFrontHOOD }
          | useBranchFn        { [true | on | 1] | [false | off | 0] }
          | useSingularBranch  { [true | on | 1] | [false | off | 0] }

          | snapNodesToSurfAndFront snappingTolSurf {tol #}
                                    [snappingTolFront {tol #}]
          | snapNodesToSurfAndFront { [true | on | 1] | [false | off | 0] }
        }
      } # end of crackMgr2D sub-commands


createSIFExtractor [ [CIM2D | CIM3D | Jintegral2D | Jintegral3D | DCM3D ]
                       edgeElasID  {edgeID #}
                     | CFM2D nodeOrigin {nodeID #} nodeXAxis {nodeID #}
                     | CFM3D nodeOrigin {nodeID #} nodeZAxis {nodeID #}
                       nodeXZPlane {nodeID #}
        [ [ tracFace+ {t_n #} {t_t #}  tracFace- {t_n #} {t_t #} |
```

```
             tracFace+ {t_n #} {t_t #} {t_b #}
             tracFace- {t_n #} {t_t #} {t_b #} ] ]

extractSIF [discreteExtract | DCM]
            {  [ extrDist_1 {extrDist_1 #} extrDist_2 {extrDist_2} ]
             | [ extrDist_min {extrDist_min #} deltaDist {deltaDist #}
                 numExtrPoints {numExtrPoints #}
                 [averLinExtrapl | leastSqrExtrapl]
               ]
               [ projPntOnCrackSurface ]
            }
           | [contourInt] radius {radius}
                          [ numIntegPoints {nint #} [ term {term #} ] ]
           | [domainInt]  radius_1 {radius_1} radius_2 {radius_2}
                          [ cylThickness {zThickness} ]
                          [ numIntegPoints_r  {nint_r #}
                            numIntegPoints_t  {nint_t #} ]

graphMeshMng  [localProb {probID #}]
              MngID {MngID #}
             [ create {file_name} [DXStyle | TecPlotStyle | HyperMeshStyle
                                  | VTKStyle] [ascii | binary]
             | open   {file_name}
             | postMesh
             ]

masterCrackMgr
  [ create

  | advanceFront
      {  [ DCM  extrDist_1 {extrDist_1 #} extrDist_2 {extrDist_2 #} ]
       |
         [ DCM extrDist_min {extrDist_min #} deltaDist {deltaDist #}
              numExtrPoints {numExtrPoints #} [averLinExtrapl | leastSqrExtrapl] ]
           [ projPntOnCrackSurface ]
       | [ CIM radius {radius #}
              [ numIntegPoints {nint #} ]
       |
         [ CFM  radius_1 {rho1 #} radius_2 {rho2 #}
              [ cylThickness {zThickness} ]
              [ numIntegPoints_r {nint_r #}
                numIntegPoints_t {nint_t #} ]
      }
        [ tracFace+ {t_n #} {t_t #} {t_b #}
          tracFace- {t_n #} {t_t #} {t_b #} ]

      [ start_time {stime #} end_time {etime #}]

  | forAllCracks
```

```
{any sub-command of crackMgr (except create and advanceFront)
 after crackMgrID {crackMgrID #} } ]
```

3.5 Tcl Commands for Analysis of Polycrystals

```
grainManager [ localProb {probID #}]
             [ create grainMgID {grainMgID #}
                  grainFile {file_name}
                  [ GrainAnalysis | CrackAnalysis ]
             | process grainMgID {grainMgID #}
                  [ MaxNumTermsStepFn {max_n_tems #} ]
             | refineProbMesh grainMgID {grainMgID #}
                  [ crackFronts | crackVertices ]
```

3.6 Tcl Commands for a Coupled FTSI Analysis

```
coupledFTSIManager  [ create
                     | UExternalDB kstep {#} \n
                                   kinc {#}  \n
                                   lop {#}   \n
                                   curr_time {#}  \n
                     | URDFIL [ Temperature | Displacement |
                                Coordinates ]
                    ]
```

3.7 Tcl Commands for a Multi-Physics Analysis

```
multiPhysDir  [localProb {probID #}]
              [ create {thermoelas1way | thermoplas1way | phasefield2way}
              | {cloneProblem | cloneProblemWithCrackMg}
              | activate {heat | thermoelas | thermoplas |
                          phasefield | phasefieldelas}
              | staggeredSolOption [ setMaxStaggeredIter { # iterLim }
                                  | setStaggeredConvTol { # conTol } ]
              | solveStaggeredStep { # stepNum } [ adapStep ]
              ]
```

[heading=subbibliography]

4. Description of *ISET*Tcl commands

This chapter contains short descriptions all Tcl commands implemented in *ISET*. Examples are also provided in some cases. Many additional examples can be found the *ISET* Quality Assurance (QA) Test Suite. To find them use the following commands from the command prompt at folder SetSolver/qa-tests

```
find . -name \*.tcl | xargs grep enrichApprox
```

This will print a (long) list of Tcl files where *ISET* Tcl command enrichApprox is used. Files with examples of other commands can be found using the above commands with enrichApprox replaced by another command name.

4.1 *ISET* Tcl commands

ISET uses *Tcl* (Tool Command Language) as its control command language. The *Tcl* library provides a full-function scripting language that includes both the commands provided by *Tcl* (called *Tcl* core commands) and those that were specifically added for *ISET*. The following syntax conventions are adopted for all *ISETTcl* commands:

- Square brackets [ ] used in the *Tcl* commands described below indicate that a single entry (or entry set) must be selected from those options provided. Brackets are not to be taken literally so don't use them when executing a command.
- Items within curly braces { } are options, of which you must choose only one. Braces are not to be taken literally so don't use them when executing a command.
- A vertical line | separates two or more items within square brackets and curly braces, only one of which may be chosen.
- Any text not contained in a bracket or brace is required. In the syntax of many commands, the only text not surrounded by one or more brackets or braces is the command name itself.

TBD: Move this to GFEMgl section

Several commands such as enrichCompNod, graphMesh, assemble, etc. can be used with option [localProb {probID #}]. The operation of the *Tcl* commands with this option is the same as those without it except the fact that it is used in the local problem with ID probID.

Regarding the entire format of the command which contains this option, refer to the definition of the corresponding command.

4.2 Physics-Independent Tcl Commands

4.2.1 readFile

The *Tcl* command readFile provides an interface for reading new models as well as for reading restart files from a prior session. The format for this command is:

```
readFile {file_name} {phfile | restart}
         [ [solids] | heat | fluids | thermomech | hydroFrac
           | phasefieldelas | krauklisfluid | krauklis
           | pipe | pipelub ]
```

- *file_name* is a user prescribed file name.
- phfile identifies a phlex formatted input file.
- restart option is not currently available.
- Parameters solids, heat, fluids, thermomech, hydroFrac etc. define the type of problem described in *file_name*. The default problem type is solids.

4.2.2 readGlobalMatrixAndRHS

The *Tcl* command readGlobalMatrixAndRHS provides an interface for reading the global stiffness matrix and load vector generated by other codes on in a previous run of *ISET*. The format for this command is:

```
readGlobalMatrixAndRHS [ abaqus | iset ]
```

R Filenames must be: stiff_abaqus.mtx and load_abaqus.mtx or stiff_iset.mtx and load_iset.mtx.

4.2.3 createAnalysis

The *Tcl* command createAnalysis is used to create an object for a linear or nonlinear analysis. The format for this command is:

```
createAnalysis [ multiPhys {heat | thermoelas | thermoplas} ]
               [  linear | linearReAnalysis [ Rebuild_F0 ] ]
```

- If optional parameter multiPhys is provided, a multi-physics analysis will be performed. The type of multi-physics analysis is defined provided one of the parameters: heat, thermoelas or thermoplas.
- Parameter linear defines that a linear analysis will be performed.
- Parameter linearReAnalysis is similar to parameter linear but in this case a re-analysis algorithm will be used if the problem is solved more than one.
- Parameter Rebuild_F0 requests that the load vector used when solving the problem the first time, be rebuilt when re-solving the problem with a re-analysis algorithm. This leads to more accurate computations of load vector but is more computationally demanding.

R This command is deprecated. Users should use instead command analysis with parameter create. See Section 4.2.4 for details.

4.2.4 analysis

The *Tcl* command analysis is used to create an object for a linear, nonlinear or transient analysis. The format for this command is:

```
analysis [localProb {probID #}]
         [multiPhys {heat | thermoelas | thermoplas}]
                [ create [linear | matNonlinear | matVisco | transient
                         | transientReAnalysis | nonlinearCohesive
                         | coupledHydroFracAnalysis | krauklisAnalysis
                         | phaseFieldAnalysis ]

                [ solOption [ numTimeSteps {# steps}
                            | simulationTime {time} ]
                  [ nonSolOption [ solAlgorithm [newton | modifiedNewton]
                                 | numTimeSteps {# steps}
                                 | convCriterion { L2Norm | residual
                                                 | strainEnergy }
                                 | convTolerance {# tol}
                                 | absTolerance {# tol}
                                 | thresholdTolerance {# tol}
                                 | maxNumNonlinIters {# iters} ]
                                 | resetStateVars
                                    [{true | on | 1} | {false | off | 0}]
                                 | loadIncrType [uniform | locUniform |
                                                 locProb | constant |
                                                 ramp | creep | hat | freq |
                                                 impulse | userDef ]
                                 | isLineSearch
                                    [{true | on | 1} | {false | off | 0}]
                                 | simulationTime {time}
                                 | peakTime {ptime}
                                 | isShapeFnTimeDepen
                                 | isIncrementalSol
                                    [{true | on | 1} | {false | off | 0}]
                | nonlinearSolve [ allLocalProbs ] [ timeStep {# istep} ]
                | nonlinearAssemble [ allLocalProbs ] { timeStep {# istep} }
                | linearSolve [ allLocalProbs ] [ timeStep {# istep} ]
                ]
                [ transSolOption [ transSolAlgorithm [ alphaMethod
                                                     | newmarkMethod
                                                     | centralDifference
                                                     | krauklisTimeInt ]
                                 | transNumTimeSteps {# steps}
                                 | transStopTime {# tFinal}
                                 | initialTime {# tInitial}
                                 | isShapeFnTimeDepen
                                    [{true | on | 1} | {false | off | 0}]
                                 | useLumpedMassForL2Proj
                                    [{true | on | 1} | {false | off | 0}]
                                 | useLumpedMassForTransAnalysis
```

```
                        [{true | on | 1} | {false | off | 0}]
                    | saveTransFactors
                        [{true | on | 1} | {false | off | 0}]
                    | transSetParameters [ alpha {# alpha}
                                         | beta  {# beta }
                                         | gamma {# gamma} ]
                    | accelerationBased
                        [{true | on | 1} | {false | off | 0}]
                      ]
| coupledHydroFracSolOption [ maxNumNewtonIters {# iters}
                     | timeIncrement {# deltaT}
                     | minTimeStep {# tmin}
                     | maxTimeStep {# tmax}
                     | convTolerance {# tol}
                     | totalSimulationTime {# tfinal}
                     | propagatingCrack {true | false}
                     | restart {true | false}
                     | startTime {# init_time}
                     | assignCrackMgr {# crackMgrID} ]
                     | parametersDCM { extrDist_min {extrDist_min #}
                                       deltaDist {deltaDist #}
                                       numExtrPoints {numExtrPoints #}
                                     }
                     | minOpeningAllowed {# wmin}
             | initPropagStepWithSmallVisco factorSmaller {# factor}
             | useFillOfFractureWithSmallVisco factorSmaller {# factor}
                                               fillingTime {# time}
                     | pOrdForFluidEls {# pOrd}
                     | identifyAndSetCornerNodesToHalfInj {true | false}
                     | oneDeltaTPerDeltaA {true | false}
                     | leakOffCarterCoeff {#}
                     | spurtLoss {#}
                     | initLeakOffExposureTime {#}
                    ]

| coupledHydroFracSolve
| coupledHydroFracSolveAndPropagate

| krauklisSolOption [ pOrdForFluidEls {# pOrd} ]
                       neglectSolidInertia {#}
                       solveCtePressProblemForIniOpening {#}
                       constantW0 {# ctew0} ]
                       minimumW0 {# minimumW0} ]
| krauklisComputePhaseVelAndAttenuation {initCoord {# x} {# y} {# z}
                                         direction {# x} {# y} {# z}
                                         dist {#} npts {#} }
| phaseFieldSolOption [initializePFSol]
| quasiStaticSolve [Time {# Time}]
| transientSolve  {# Time}
```

```
| hardCodeProjectSolutionForP1
| L2ProjectInitialCond [{approxOrder #}]

| transAssembleReAn {# Time}
| transSolveReAn
| transSolveReAnFast
| setParallelAssemble

| outputOption [ saveResultsFreq {# freq}
              | extractResultsFreq {# freq}
              | graphicalResol {# resol} ]

| writeSolution                filename [binary | ascii]
| processSolutionInitialProblem filename [binary | ascii]
| computeDumpPseudoLoads       filename [binary | ascii]
| readPseudoLoads              filename [binary | ascii]
| computeDumpPseudoSolutions   filename [binary | ascii]
| readPseudoSolutions          filename [binary | ascii]
| solveEnrichedProbReAnalysisTwoSolvers
| sortLocalProbs {ascend | descend}
]
```

- TBD: Please describe the parameters for this command.
- `transSolOption`
  - `transSolAlgorithm` is used to create a transient analysis object. The `alphaMethod` includes the `forwardEuler` and `backwardEuler` methods. The `newmarkMethod` and `centralDifference` are currently at a *beta* developement stage.
    The time-step loop is controlled from the *Tcl* file.
  - `transSetParameters`: Parameter `alpha` is used in transient simulations of the parabolic heat equation with the `alphaMethod`. Unconditional stability is obtained with `alpha` $\geq \frac{1}{2}$. The backward Euler algorithm is obtained with `alpha` = 1.0 and the forward Euler algorithm is obtained with `alpha` = 0.0.
    Parameters `beta` and `gamma` are used with the `newmarkMethod`.
  - Option `isShapeFnTimeDepen` sets a boolean variable in the transient analysis class to `true` or `false`, indicating whether time-dependent shape functions are being used.
- `quasiStaticSolve` is used for the computation of the initial solution vector used to setup initial conditions for the time-dependent problem.
- `transientSolve Time` produces the solution at time = `Time`.
- `solOption` is used to set solution parameters. It allows the user to set `simulationTime`, `numTimeSteps` for an analysis. This allows the use of load stepping in all analysis classes.
- `nonSolOption` is used to set parameters for a nonlinear analysis. Users can specify nonlinear solvers, convergence critieria, time step parameters, and more.
- `phaseFieldSolOption` is used to interface with the phasefield analysis class. Currently this function is only used to initialize the phase field solution with $z = 1.0$ and avoid an unnecessary solve. Other parameters associated with the nonlinear solving of phase field equations are set via the `nonSolOption` function.

4.2.5 enrichApprox

The *Tcl* command enrichApprox is used to enrich the GFEM approximation to a given polynomial order. By default a linear ($p = 1$) GFEM approximation is adopted. This command combines the functionality of *Tcl* commands setSpBasis and enrichCompNod described in sections 4.2.9 and 4.2.6, respectively. Command enrichApprox simplifies the setting of the nodal polynomial order when there are many different special basis enrichment functions in a GFEM discretization. It automatically sets the maximum number of custom terms for all special basis and the resulting polynomial order of the approximation. The syntax of this command is as follows:

```
enrichApprox  [localProb {probID #}]
              [multiPhys {heat | thermoelas}]
              [iso | xi | eta | zeta] approxOrder {order #}
```

- The polynomial enrichment can be isotropic (all directions) or in the direction xi, eta or zeta, where xi, eta, zeta are directions in the coordinate system associated with each node. Currently, these directions are the same as the global directions x, y, z.
- Parameter approxOrder defines the order of the GFEM approximation.

> R In contrast with parameter order in command enrichCompNod, approxOrder sets the *resulting* polynomial order of the GFEM approximation, *not* the order of the enrichment functions since command approxOrder takes into account the polynomial order of the FE partition of unity.
>
> Command approxOrder assumes that a linear FE partition of unity is used when selecting the order of the enrichment functions.

- localProb: See section 4.3.2.
- multiPhys: See command analysis in Section 4.2.4.

Example: A quadratic GFEM approximation can be set using the command

```
enrichApprox iso approxOrder 2
```

4.2.6 enrichCompNod

The *Tcl* command enrichCompNod provides an interface for enriching the nodes in a GFEM discretization.

```
enrichCompNod [localProb {probID #}] [multiPhys {heat | thermoelas}]
              [all [ hasStepFn | hasBranchFn | hasLocalSol ]
              | cnode {cnodeID #}
              | onSegment A {Ax #} {Ay #} {Az #}
                          B {Bx #} {By #} {Bz #}
              | inBBox xyzMin {Xmin #} {Ymin #} {Zmin #}
                       xyzMax {Xmax #} {Ymax #} {Zmax #}
              | inCylz center {x #} {y #} {z #}
                       radius {r #}
                       height {h #}             ]
              [nodeType {vertex | edge | face
                         | interior | any}]
              [iso | xi | eta | zeta] order {order #}
```

- Parameter order defines the order of the polynomial enrichment.

- The polynomial enrichment can be isotropic (all directions) or in the direction `xi`, `eta` or `zeta`, where `xi`, `eta`, `zeta` are directions in the coordinate system associated with each node. Currently, these directions are the same as the global directions x, y, z.
- `all`: All nodes is the mesh are enriched with polynomial functions of degree `order` if this option is selected.
- If `hasStepFn`, `hasBranchFn` or `hasLocalSol` parameters are used, all nodes enriched with step (Heaviside) functions, singular crack tip functions or local problem solutions, are also enriched with polynomial functions of degree `order`.
- `cnode`: A single node with ID `cnodeID` is enriched if option `cnode` is selected.
- `onSegment`: All nodes on the defined segment will be enriched if this option is selected.
- `inBBox`: All nodes contained within a defined bounding box will be enriched.
- `inCylz`: All nodes contained within a cylinder along the global z-axis, with given radius and height, will be enriched.
- `nodeType`: This option provides a filter for the set of nodes selected with one of the previous options (`all`, `inBBox`, etc.). A node will be enriched only if it is of the specified type: `vertex`, `edge`, `face`, `interior` or `any`. This option should be used only if parameter `useCompElPfemGfem` is true.
- `localProb`: See section 4.3.2.
- `multiPhys`: See command `analysis` in Section 4.2.4.

Command `enrichCompNod` allows the enrichment of only a set of nodes in the domain while command `enrichApprox` enriches all nodes in the domain. However, the latter command is recommended for all new users since it automatically takes care of selecting the appropriate order of polynomial enrichments at each node.

4.2.7 `setCompNodSpBasis`

The *Tcl* command `setCompNodSpBasis` provides an interface for assigning analytically or numerically constructed special enrichment functions to nodes of a GFEM mesh. Examples of these functions are Heaviside functions and solution of local problems.

All the nodes are enriched if the option `all` is selected or a single node with ID `cnodeID` if option `cnode` is selected or the nodes on the segment AB if option `onSegment` is selected. `Ax`, `Ay`, and `Az` denote x, y, and z values of point A in the global coordinate system and the enrichment is always done isotropically (in x, y and z direction). The format for this command is:

```
setCompNodSpBasis [localProb {probID #}]
                  [multiPhys {heat | thermoelas | thermoplas}]
                  [ all
                  | cnode {cnodeID #}
                  | locProb
                  | onSegment A {Ax #} {Ay #} {Az #}
                              B {Bx #} {By #} {Bz #}
                  | inBBox xyzMin {xMin #} {yMin #} {zMin #}
                           xyzMax {xMax #} {yMax #} {zMax #}
                  | inCylz center {x #} {y #} {z #}
                           radius {r #}
                           height {h #}          ]
                  [nodeType {vertex | edge | face
                             | interior | any}]
                  spBasis {spBasisID #}
```

- spBasisID defines the ID is the special basis. This special basis must, in most cases, have been predefined in the .grf file.
- all: Special function with ID spBasisID is assigned to all nodes.
- cnode: Special function with ID spBasisID is assigned to a single node with ID cnodeID.
- locProb: This option will
  - Find local problem (which is a SpecialBasis) object with ID spBasisID.
  - Set the found local problem SpecialBasis to all global nodes with ID equal to seed nodes that were used in the definition of the local problem.
- onSegment: All nodes on the defined segment will be assigned the special basis.
- inBBox: All nodes contained within a defined bounding box will be assigned the special basis.
- inCylz: All nodes contained within a cylinder along the global z-axis, with given radius and height, will be assigned the special basis.
- nodeType: This option provides a filter for the set of nodes selected with one of the previous options (all, inBBox, etc.). A node will be assigned a special basis only if it is of the specified type: vertex, edge, face, interior or any. This option should be used only if parameter useCompElPfemGfem is true.
- localProb: See section 4.3.2.
- multiPhys: See command analysis in Section 4.2.4.

R This command is mostly used by developers. Special basis are automatically assigned to nodes using, for example, a Crack Manager as described in Section 4.5.5.

This command should be used *before* command enrichApprox is used.

This commands assigns a special basis to a set of nodes. However, if the special basis will actually be used depends on the order set for the node. For example, EdgeElas basis is only used by a node if its order is equal to one or higher.

4.2.8 assignPtBC

The *Tcl* command assignPtBC provides an interface for assigning point boundary conditions to nodes of a GFEM mesh. All nodes in a bounding box are assigned the point BC is option inBBox is selected or a single node with ID cnodeID if option cnode is selected.

This command is useful after mesh refinement of a GFEM model with point BCs since nodes created by the mesh refinement are not automatically assigned point BCs as there are no generic and robust way of doing this. Thus, the user may want to assigned additional point BCs to the GFEM model after performing mesh refinement.

R Only point Dirichlet BC for elasticity problems is supported at this time.

```
assignPtBC  [localProb {probID #}]
            [Elas2D | Elas3D | Plas3D | ViscoElas3D]
            [cnode {cnodeID #}
            |inBBox xyzMin {Xmin #} {Ymin #} {Zmin #}
                    xyzMax {Xmax #} {Ymax #} {Zmax #}]
            PtDisplacement
            Displacement {ubar_x} {ubar_y} {ubar_z}
            Flags {flag_x} {flag_y} {flag_z}
```

- localProb: See section 4.3.2.

- Elas2D, Elas3D, Plas3D or ViscoElas3D should be selected depending on the type of elasticity problem being solved.
- cnode and cnodeID define the ID of the node to which the point BC will be applied.
- inBBox: All nodes in a bounding box are assined the point BC if this option is used. Parameters xyzMin and xyzMax define the bounding box relative to the global coordinate system.
- PtDisplacement defines that a PtDisplacement BC will be assigned. This is the only type of point BC currently supported by this command. See Section 2.22.2 for further details on this type of point BC.
- Displacement provides the components of the prescribed displacement vector.
- Flags are activation flags. The activation flags may be set to either 0, or 1. Zero is 'off' (i.e., ignore the constraint for this component). One is 'on' which means to use the value specified. See also Section 2.22.2.

4.2.9 setSpBasis

The *Tcl* command setSpBasis provides an interface for setting parameters used by special basis classes. The only parameter currently available is the maximum number of custom terms for enriching with Special Basis objects. The user must provide the type of special basis to be set. The maximum number of custom terms is set for a single Special Basis with ID spBasisID if option spBasis is selected. Another option is to set the maximum number of custom terms for all Special Basis that are of type *Step Function* or *Branch Function* by using options all and hasStepFn or all and hasBranchFn .

```
setSpBasis [localProb {probID #}]
           [multiPhys {heat | thermoelas}]
           [ all { hasStepFn | hasBranchFn | hasLocalSol }
           | spBasis {spBasisID #} ]
           maxNumCustomTerms {maxTerms #}
```

R This command is mostly used by developers.
This command should be used *before* command enrichApprox is used.

4.2.10 A Comparison Between enrichCompNod and enrichApprox Commands

Commands enrichCompNod and enrichApprox are explained and contrasted here in the context of a specific example of a problem solved with the GFEM[gl], wherein some nodes of the global problem are enriched with the solution of the local problem.

Let's assume that there are a total of 100 nodes in the global problem. We want to enrich only 5 nodes of the global problem with the local solution. Let's say that those 5 nodes have ID (defined in .grf file) 11, 23, 42, 55, and 77. We also want that the resulting polynomial order of our approximation be 2 in all three global directions.

Procedure Based on Command enrichCompNod

To achieve our goals with the help of the enrichCompNod command, what we will do is the following. First, set the enrichment order of all nodes to be 1, using the following statement

```
enrichCompNod all iso order 1
```

We could also have used

```
enrichCompNod all xi   order 1
enrichCompNod all eta  order 1
enrichCompNod all zeta order 1
```

R We are assuming that the FEM shape functions are linear. Therefore the enrichment order of 1 means that the resulting polynomial order of the GFEM shape functions will be 2, because the enrichment functions are multiplied by the linear FEM shape functions.

Next, suppose that a local problem with ID = 10 was created and solved. The solution of this problem is assigned as an enrichment function to the above mentioned 5 global nodes using the following commands:

```
setCompNodSpBasis cnode 11 spBasis 10
setCompNodSpBasis cnode 23 spBasis 10
setCompNodSpBasis cnode 42 spBasis 10
setCompNodSpBasis cnode 55 spBasis 10
setCompNodSpBasis cnode 77 spBasis 10
```

This commands assigns a special basis with spBasisID = 10 to a set of nodes. However, if the special basis will actually be used depends on the order set for the nodes. In this example, the order of those nodes must be at least one, which is the case.

Next, to enrich those 5 global nodes with the local problem solution *and* with the polynomial enrichment functions, the following commands must be used

```
enrichCompNod cnode 11 iso order 2
enrichCompNod cnode 23 iso order 2
enrichCompNod cnode 42 iso order 2
enrichCompNod cnode 55 iso order 2
enrichCompNod cnode 77 iso order 2
```

Here, we are defining the order to be 2, because the nodes are also enriched with a spBasis with spBasisID = 10 and that counts as one "entry or slot" in the set of enrichments used at the node. If the above is not used, those nodes will have no polynomial enrichment, only the spBasis with spBasisID = 10. Therefore, we have to keep track of the number of "slots" taken by special basis enrichment functions when using the command enrichCompNod in order to make sure the special basis assigned to the nodes are actually used and we get the polynomial enrichment we want. This is an error prone process which can be avoided by using command enrichApprox, as we will see next.

Procedure Based on Command enrichApprox

The same thing as done above can also be implemented in the following manner:

```
setCompNodSpBasis cnode 11 spBasis 10
setCompNodSpBasis cnode 23 spBasis 10
setCompNodSpBasis cnode 42 spBasis 10
setCompNodSpBasis cnode 55 spBasis 10
setCompNodSpBasis cnode 77 spBasis 10
```

Next, use the enrichApprox command as follows:

```
enrichApprox iso approxOrder 2
```

This approxOrder (which has been assigned a value of 2 in this case), defines the resulting polynomial order of approximation. This command takes care of the special basis functions and the polynomial order as well, so we need not take care of the nodes to be enriched with the special basis functions, separately.

As illustrated above, enrichApprox is easier to use and is more comprehensive as compared to enrichCompNod.

4.2.11 delete

This *Tcl* command deletes from computational mesh data base the special basis object with ID = spBasisID.

```
delete [localProb {probID #}] spBasis {spBasisID #}
```

- localProb: See section 4.3.2.

4.2.12 refine

This *Tcl* command refines tetrahedron elements in the mesh using the bisection algorithm proposed by Arnold and Mukherjee. The command can only be used with TET4 elements.

```
refine [localProb {probID #}]
       [ all
       | element {elemID #}
       | hasNode {geoNodeID #}
       | hasNodeInBBox xyzMin {Xmin #} {Ymin #} {Zmin #}
                       xyzMax {Xmax #} {Ymax #} {Zmax #}  ]
```

- all Bisects all elements in the mesh.
- element Bisects element with ID elemID.
- hasNode Bisects all elements sharing GeoNode with ID geoNodeID.
- hasNodeInBBox Bisects all elements which has a GeoNode inside the user-defined bounding box.
- localProb: See section 4.3.2.

4.2.13 refineIfIntersect

The *Tcl* command refineIfIntersect has basically the same operation as *Tcl* command refine. The difference is that this command bisects elements which intersect a defined geometric entity. The default entity is a line segment connecting points A and B. The second option, requiring the flag plane is a plane defined by a point P, and unit normal vector, defined by N. The following convention is used: Ax, Ay, and Az denote x, y, and z values of point A in the global coordinate system. The format for this command is:

```
refineIfIntersect  [localProb {probID #}]
             [ [segment] A {Ax #} {Ay #} {Az #}  B {Bx #} {By #} {Bz #}
             |  plane    P {Px #} {Py #} {Pz #}  N {Nx #} {Ny #} {Nz #} ]
```

4.2.14 unRefine

This *Tcl* command recovers the mesh read from input .grf file by unrefining all elements that have beem refined using one of the mesh refinement commands.

```
unRefine [localProb {probID #}]
         toInitialMesh
```

- localProb: See section 4.3.2.

4.2.15 meshReport

This *Tcl* command prints information about element size and element quality in a GFEM mesh. An example is shown below.

```
meshReport [localProb {probID #}]
```

- localProb: See section 4.3.2.

Example of a mesh report:

```
Range of element edge sizes at crack front after refinement:
Maximum Edge Size = 0.0106563
Minimum Edge Size = 0.0059525
Ratio = 1.79023
Maximum Edge lenght ratio in an element = 1.79023
Max. radius-length = 0.895113
this value should be bounded as small as possible.
Q = 0.612 is optimal, if we do not consider sliver elems.
```

4.2.16 colorMesh

This Tcl command "colors" elements in the GFEM mesh in order to improve the parallel efficiency during assembly of the global matrices.

```
colorMesh [localProb {probID #}] [multiPhys {heat | thermoelas}]
      [maxColorSize {# size}]
```

- maxColorSize this optional parameter defines the maximum number of elements with the same color.
- localProb: See section 4.3.2.
- multiPhys: See command analysis in Section 4.2.4.

This command is deprecated since OpenMP provides more efficient alternatives which were implemented in *ISET* after colorMesh.

4.2.17 assemble

The *Tcl* command assemble provides an interface for requesting the assembling of the global matrix (or matrices) and load vector(s). The format for this command is:

```
assemble [ localProb {probID #} ]
         [ multiPhys { heat | thermoelas | thermoplas |
                       phasefield | phasefieldelas } ]
         [ allLocalProbs
           [transient Time {# Time}] ]
         [ reAnalysis ]
         [ step {#stepNum} ]
```

- reAnalysis is used when option linearReAnalysis can be used with Tcl command createAnalysis. It will assemble only the part of the global matrices and vectors that changed since the previous solution step.
- localProb: See section 4.3.2.
- multiPhys: See command analysis in Section 4.2.4.
- step allows the user to set a time step for the analysis. This allows the loads to be factored based on the numTimeSteps, simulationTime provided in the analysis commands in Section 4.2.4

4.2.18 parallelAssemble

The *Tcl* command parallelAssemble is equivalent to the following commands

```
analysis setParallelAssemble
assemble
```

```
parallelAssemble [localProb {probID #}]
                 [multiPhys {heat | thermoelas | thermoplas}]
```

- localProb: See section 4.3.2.
- multiPhys: See command analysis in Section 4.2.4.

4.2.19 scaleGlobalMatrix

The *Tcl* command scaleGlobalMatrix scales the global matrix (e.g. stiffness, tangent matrix) and saves scaling factors. Scaling is performed upon invocation of this command, instead of just before the factorization of the global matrix. This command is typically used before requesting the computation of the condition number of the global matrix.

```
scaleGlobalMatrix [ localProb {probID #} ]
```

- localProb: See section 4.3.2.

4.2.20 solve

The *Tcl* command solve provides an interface for firing the linear equation solver. The format for this command is:

```
solve [ localProb {probID #} ]
      [ multiPhys { heat | thermoelas | thermoplas |
                    phasefield | phasefieldelas } ]
      [ allLocalProbs
          [transient Time {# Time}] ]
      [ reAnalysis | reAnalysisFast | reAnalysisPCG {BGS | BJ | CG}  {#tol} |
        reAnalysisBGS {#tol} | reAnalysisScratch ]
```

- reAnalysis and reAnalysisFast can be used when option linearReAnalysis is used with Tcl command createAnalysis.
- localProb: See section 4.3.2.
- multiPhys: See command analysis in Section 4.2.4.

4.2.21 printSolCoeff

The *Tcl* command printSolCoeff is used to print the coefficients of the solution vector associated with a node (a CompNod) or with all nodes. The coefficients of all nodes are printed if the option all is selected or the coefficients of a single node with ID cnodeID if option cnode is selected. Coefficients associated with standard FEM shape functions have the usual physical meaning while those associated with enrichment functions have no physical meaning. The format for this command is:

```
printSolCoeff [localProb {probID #}] [all | cnode {cnodeID #}]
```

- all Prints solution coefficient for all nodes of the mesh.
- cnode Prints solution coefficient for node with ID cnodeID.
- localProb: See section 4.3.2.
- multiPhys: See command analysis in Section 4.2.4.

R This command is mostly used by developers. It can also be useful to check what are the shape functions assigned to a node either by the user in the Tcl file or automatically by the code.

4.2.22 computeNodReaction

The *Tcl* command computeNodReaction is used to compute the reaction (external force or flux) at a CompNod of the GFEM mesh. The physical meaning of computed reaction depends on the type of problem (Material) being solved. It will, for example, be a flux in the case of heat problems. The reaction is equal to the sum of internal forces exerted on the node by all elements sharing the node. Reactions are printed to standard output and saved in the extract file if it is available when the command computeNodReaction is used.

R Reactions can not be computed at CompNods enriched with custom functions like Heaviside or singular functions since at this time these functions are not necessarily zero at the nodes. Therefore, the reaction computed at a node with custom enrichments will not have the expected physical meaning.

The format for this command is:

```
computeNodReaction [localProb {probID #}]  cnode {cnodeID #}
```

- cnode Computes the reaction at a node with ID cnodeID.

  R The ID of a CompNod is the same as the ID of the GeoNod at the same location.

- localProb: See section 4.3.2.

4.2.23 extractQuantity

The *Tcl* command ExtractQuantity is used to extract the value of any quantity that can be computed for the tye of problem (material class) being solved. All extracted quantities are saved in a file named

```
jobName + "_extractQuantity.rst"
```

where jobName is the name of the Tcl file without its .tcl extension.

The format for this command is:

```
extractQuantity [multiPhys {heat | thermoelas | thermoplas} ] {Quantity}
                point {x#} {y#} {z#} [elemDim {dim#}] [reducedVerbosity {bool}]
```

- `Quantity` Name of the quantity to be extracted (computed).
- `Point` Global coordinates of the point where `Quantity` will be computed.
- `localProb`: See section 4.3.2.
- `multiPhys`: See command `analysis` in Section 4.2.4.
- `elemDim` Dimension of the element used to extract quantity. Useful for hydraulic fracture problems where the fluid elements are 2D, and the solid elements are 3D.
- `reducedVerbosity` Prints the output with minimal text. Useful when extracting quantities through many time steps.

4.2.24 `work`

The *Tcl* command `work` is used to request the computation of the work of all applied loads (Neumann boundary conditions). It does *not* compute the work of reaction forces which is non-zero in the case of non-homogeneous Dirichlet boundary conditions. In the case of problems with Neumann only or with homogeneous Dirichlet boundary conditions, the work of prescribed Neumann boundary conditions is equal to the strain energy of the GFEM solution over the analysis domain. Body forces must also be zero for this to be the case.

```
work [localProb {probID #}] [multiPhys {heat | thermoelas}] [addIntOrder {order #}]
```

- `addIntOrder` option is used to increase the order of the default quatrature rule used when computing work over each element face with Neumann boundary condition. Default is zero (adopt same quadrature rule as used when computing the load vector). This parameter must be an integer.
- `localProb`: See section 4.3.2.
- `multiPhys`: See command `analysis` in Section 4.2.4.

R This approach to compute strain energy is *much* faster than using cammand `strainEnergy`. It is also more accurate than the latter when singular enrichment functions are used.

R The strain energy in the analysis domain is not equal to the work of applied loads if nonhomogenous Dirichlet boundary conditions or body forces are prescribed.

4.2.25 `solInnerRHS`

The *Tcl* command `solInnerRHS` is used to request the computation of the inner product between the solution vector and the load vector. The format for this command is:

```
solInnerRHS [localProb {probID #}]
```

R The *Tcl* command `solInnerRHS` is deprecated and should not be used.

4.2.26 strainEnergy

The *Tcl* command strainEnergy is used to compute the strain energy of the GFEM solution in the analysis domain. The strain energy from the internal stresses and strains is computed over in each element. In contrast with the work command, this command can compute the strain energy even if nonhomogenous Dirichlet boundary conditions or body forces are prescribed. However, strain energy values computed by this function are not exactly correct when the internal stress or strain is described by non-polynomial functions with singularities due to integration errors. Command work is much less sensitive to integration errors. If the exact solution of the problem is also known (see Section 4.2.34), the strain energy of the exact solution and the energy norm of the error are also computed.

```
strainEnergy [localProb {probID #}]
             [multiPhys {heat | thermoelas}] [addIntOrder {order #}]
             [matName {Name_of_the_material}]
```

- addIntOrder option is used to increase the order of the default quatrature rule used when computing the strain energy over each element. Default is zero (adopt same quadrature rule used when computing the stiffness matrix). This parameter must be an integer.
- localProb: See section 4.3.2.
- multiPhys: See command analysis in Section 4.2.4.
- matName option is used to select the set of elements over which the strain energy is computed. Only elements with material name Name_of_the_material are used in the computations. The name of the material used by each element is defined in the .grf file (cf. Chapter 2). It is common that materials with different names adopt the same parameter values. Multiple materials are used just as a way to define multiple sets of elements in an analysis domain. If the name of the material provided does not match any material defined in the .grf file, the code will exit with an error.

R Option matName is ignored if option multiPhys is used.

4.2.27 parallelStrainEnergy

The *Tcl* command parallelStrainEnergy computes strain energy of the GFEM solution in the analysis domain in parallel. It is equivalent to commands

```
analysis setParallelAssemble
strainEnergy
```

```
parallelStrainEnergy [localProb {probID #}]
                     [multiPhys {heat | thermoelas}] [addIntOrder {order #}]
                     [matName {Name_of_the_material}]
```

- addIntOrder option is used to increase the order of the default quatrature rule used when computing the strain energy over each element. Default is zero (adopt same quadrature rule used when computing the stiffness matrix). This parameter must be an integer.
- localProb: See section 4.3.2.
- multiPhys: See command analysis in Section 4.2.4.
- matName option is used to select the set of elements over which the strain energy is computed. Only elements with material name Name_of_the_material are used in the computations. The name of the material used by each element is defined in the .grf file (cf. Chapter 2). It is common that materials with different names adopt the same parameter values. Multiple

materials are used just as a way to define multiple sets of elements in an analysis domain. If the name of the material provided does not match any material defined in the .grf file, the code will exit with an error.

R Option matName is ignored if option multiPhys is used.

4.2.28 strainEnergyAndNorms

The *Tcl* command strainEnergyAndNorms is analogous to command strainEnergy but it also computes the L^2 of the GFEM solution and of the exact solution, if known.

```
strainEnergyAndNorms [localProb {probID #}]
                     [multiPhys {heat | thermoelas}] [addIntOrder {order #}]
                     [matName {Name_of_the_material}]
```

- addIntOrder option is used to increase the order of the default quatrature rule used when computing the strain energy over each element. Default is zero (adopt same quadrature rule used when computing the stiffness matrix). This parameter must be an integer.
- localProb: See section 4.3.2.
- multiPhys: See command analysis in Section 4.2.4.
- matName option is used to select the set of elements over which the strain energy is computed. Only elements with material name Name_of_the_material are used in the computations. The name of the material used by each element is defined in the .grf file (cf. Chapter 2). It is common that materials with different names adopt the same parameter values. Multiple materials are used just as a way to define multiple sets of elements in an analysis domain. If the name of the material provided does not match any material defined in the .grf file, the code will exit with an error.

R Option matName is ignored if option multiPhys is used.

4.2.29 parallelStrainEnergyAndNorms

The *Tcl* command parallelStrainEnergyAndNorms is analogous to command strainEnergyAndNorms but the element-by-element computations are performed in parallel. It is equivalent to commands

```
analysis setParallelAssemble
strainEnergyAndNorms

parallelStrainEnergyAndNorms [localProb {probID #}]
                  [multiPhys {heat | thermoelas}] [addIntOrder {order #}]
                  [matName {Name_of_the_material}]
```

- addIntOrder option is used to increase the order of the default quatrature rule used when computing the strain energy over each element. Default is zero (adopt same quadrature rule used when computing the stiffness matrix). This parameter must be an integer.
- localProb: See section 4.3.2.
- multiPhys: See command analysis in Section 4.2.4.
- matName option is used to select the set of elements over which the strain energy is computed. Only elements with material name Name_of_the_material are used in the computations. The name of the material used by each element is defined in the .grf file (cf. Chapter 2). It is common that materials with different names adopt the same parameter values. Multiple materials are used just as a way to define multiple sets of elements in an analysis domain. If

the name of the material provided does not match any material defined in the .grf file, the code will exit with an error.

R Option matName is ignored if option multiPhys is used.

4.2.30 computeResidual

The *Tcl* command computeResidual when solving a linear problem. Residue is computed from the difference between internal and external forces.

```
computeResidual [localProb {probID #}
```

- localProb: See section 4.3.2.

R This command was implemented at early stages of the implementation of the non-linear solver for elasto-plastic problems. It s *not* meant to be used when solving non-linear problems.

4.2.31 LinfNorm

The *Tcl* command LinfNorm provides an interface for requesting the L_∞ norm of the solution, derivatives, error of solution (trueError) or error of derivatives (trueErrorDeriv). The options trueError and trueErrorDeriv should be used only when the exact solution is known. This command just prints the values stored in the data base. These values are computed when generating files for post-processing. The option clear is used to reset all the values (set them to zero). The format for this command is:

```
LinfNorm [ solution | derivative | trueError | trueErrorDeriv | clear]
```

The L_∞ norm is taken as the maximum value of the selected quantity among all values dumped to a grapMesh file when executing the command

```
graphMesh postProcess
```

Therefore, the same quantity selected in the LinfNorm command, must be selected for post-processing using command

```
graphMesh appendName Solution
```

or

```
graphMesh appendName Derivative
```

or

```
graphMesh appendName TrueError
```

or

```
graphMesh appendName TrueErrorDeriv
```

depending which quantity one wants the L-infinity norm.

R Note that the captalization of arguments to command LinfNorm is not the same as for arguments to command graphMesh appendName.

4.2.32 extractFile

The *Tcl* command extractFile opens or closes a file to dump extract quantities. The content of extract files are compared with reference values when running Quality Assurance tests. If a discrepancy is found, there is a problem with *ISET*. Quality Assurance (QA) tests are regularly performed with *ISET* during the development of the code and after the installation of the code in a user machine. This command is relevant only when performing QA tests. The name of the extract file created with the create option is the same name as the Tcl file used to solve a problem but with extension .extract instead of .tcl.

```
extractFile [localProb {probID #}]  [create | close]
```

- localProb: See section 4.3.2.

4.2.33 graphMesh

The *Tcl* command graphMesh is used to create or open files used for post-processing of solutions computed by *ISET*. Command graphMesh also manipulates parameters used for the generation of post-processing files. The format for this command is:

```
graphMesh [localProb {probID #}]
          [ create {file_name}
                   [DXStyle | TecPlotStyle | HyperMeshStyle | VTKStyle]
                   [2DManifold | coupledHydroFrac | krauklis] [ascii | binary]
          | setParallelSolution
          | open   {file_name}
          | appendName     [multiPhys {heat | thermoelas | thermoplas}]
                           [coupledHydroFrac | krauklis {solid | fluid}]
            {vec_or_scal_name}
          | appendVecName  [multiPhys {heat | thermoelas | thermoplas}]
                           [coupledHydroFrac | krauklis {solid | fluid}]
            {vec_name}
          | appendScalName [multiPhys {heat | thermoelas | thermoplas}]
                           [coupledHydroFrac | krauklis {solid | fluid}]
            {scal_name}
          | postProcess [multiPhys {heat | thermoelas | thermoplas}]
                        resolution {resolution}
                        [sampleInside] [solution {sol_number}]
          | postSpBasisEnrich {file_name}
          | postSignedDistance crackMgrID {#crackMgrID}
          ]
```

- create option creates an object that handles post-processing data and opens a file to dump it. The optional arguments DXStyle, TecPlotStyle, HyperMeshStyle and VTKStyle define the data format of the file. Only DXStyle and VTKStyle are operational.
  Option VTKStyle produces files in the VTU format used by Paraview (http://www.paraview.org/).
  Option DXStyle is deprecated.
  Options TecPlotStyle and HyperMeshStyle are not currently supported.
  Options ascii and binary define the encoding used for data dumped to graphMesh file.
  Option binary is the default.
- setParallelSolution activates the paralllel generation of post-processing data.
- open is used to close existing graphMesh file and open another one named file_name.

- `appendName` option adds quantity `vec_or_scal_name` to the list of quantities that will be dumped to `graphMesh` file. It can be a scalar- or a vector-valued quantity. Valid quantity names are problem type (material type) specific. A list is given later in this section.
- `appendVecName` (deprecated) Same as option `appendName` but for vector-valued quantities only.
- `appendScalName` (deprecated) Same as option `appendName` but for scalar-valued quantities only.
- `postProcess` option activates the computation of post-processing quantities and the generation of `graphMesh` files. Parameter `resolution` specifies how many times each GFEM element will be diced when generating graphical elements. These elements are dumped to `graphMesh` file. Resolution 0 means no dicing, resolution 1 means one level of refinement, etc. Dicing may be needed to visualized quantities that varies non-linearly over a GFEM element. Note that resolution 1 and higher leads to substantially larger `graphMesh` files.
  Optional parameter `sampleInside` causes post-processing quantities to be computed inside the elements, instead of at element nodes. It is computed slightly offset from element element nodes. This is needed when post-processing discontinuous quantities whose values are not uniquely defined at graphical element nodes that are located, e.g., along a crack surface.
  Optinal parameter `solution` specifies the index of the solution to be dumped. This is relevant only when solving a linear problem with multiple load cases. Default valus is zero.
- `postSpBasisEnrich` is only valid `DXStyle` is used, which is a deprecated file style. Special basis enrichment information is dumped automatically is file style is `VTKStyle`.
- `postSignedDistance` dumps the following signed distances and polar coordinates along the crack front for the given crack manager.
  - dist_phi_y: The signed distance $\phi_y(\boldsymbol{x})$ to the crack surface and its extension in the forward direction of the crack front. This extended surface is denoted by $\bar{\Gamma}_c$ and the signed distance to it is given by

    $$\phi_y(\boldsymbol{x}) = \min_{\bar{\boldsymbol{x}} \in \bar{\Gamma}_c} |\boldsymbol{x} - \bar{\boldsymbol{x}}| \text{sign}(\boldsymbol{n}^+ \cdot (\boldsymbol{x} - \bar{\boldsymbol{x}}))$$

    where sign is the sign function and $\boldsymbol{n}^+$ is a vector normal to the positive side of surface $\bar{\Gamma}_c$. Surface $\bar{\Gamma}_c$ corresponds to the iso surface $\phi_y(\boldsymbol{x}) = 0$.
  - dist_phi_x: The signed distance function $\phi_x(\boldsymbol{x})$ to a surface orthogonal to $\bar{\Gamma}_c$ and that intersects it along the crack front. This surface corresponds to the iso surface $\phi_x(\boldsymbol{x}) = 0$. The sign of ϕ_x is defined such that $\phi_y(\boldsymbol{x}) = 0, \phi_x(\boldsymbol{x}) < 0$ gives the crack surface Γ_c.
  - dist_crFront: Distance to the crack front computed from explicit crack surface representation.
  - dist_crFront_phi: Distance to the crack front computed using

    $$r(\boldsymbol{x}) = \sqrt{\phi_x^2(\boldsymbol{x}) + \phi_y^2(\boldsymbol{x})}$$

  - polar_theta_asin: Crack front polar coordinate of a point computed using

    $$\theta(\boldsymbol{x}) = \arcsin \frac{\phi_y(\boldsymbol{x})}{r(\boldsymbol{x})}$$

    where $r(\boldsymbol{x})$ is equal to dist_crFront.
  - polar_theta_atan: Crack front polar coordinate of a point computed using

    $$\theta(\boldsymbol{x}) = \arctan \frac{\phi_y(\boldsymbol{x})}{\phi_x(\boldsymbol{x})}$$

R This option is recommended for developers only.

- localProb: See section 4.3.2.
- multiPhys: See command analysis in Section 4.2.4.

R If option VTKStyle is selected,

- Three post-processing files are create with this option:
  - file_name_GFEM.vtu: Which contains post-processing data for the GFEM mesh.
  - file_name.vtu: Which contains post-processing data for the graphical mesh. A graphical mesh is created by further sub-dividing GFEM elements into graphical elements for the purpose of visualization of the solution (see option resolution). The GFEM solution may be discontinuous inside an element. Thus, the graphical elements are created to handle situations like this.
  -
  - file_name_BCs.vtu: Which contains data for the visualization of boundary conditions applied to the problem and enrichment types assigned to the nodes of the GFEM mesh.
- A valid file_name is of the form:
  - (a) name.extension
  - (c) name
- All VTK files output will have a uniform filename extension given by "extension", if given, or "vtu", otherwise.
- Valid extensions must start with "vtu".
- ParaView *only* recognizes sequences of files of the forms filename_{#}.vtu or filename.vtu.{#} automatically.

Supported Post-Processing Quantities

- IsoHook3D Material:

  - Displacement
  - Solution
  - State
  - SolU
  - SolV
  - SolW
  - Derivative
  - Flux
  - Stress
  - Sxx
  - Syy
  - Szz
  - Sxy
  - Syz
  - Sxz
  - VonMises
  - S1
  - S2
  - S3
  - Strain
  - MechStrain
  - ThermStrain
  - ElemEnergy

  - TrueSolution
  - TrueError
  - TrueDerivative
  - TrueErrorDeriv
  - TrueElemEnergy
  - ErrorElemEnergy
  - TrueRelErrorElemEnergyNorm
  - TrueErrorElemEnergyNorm
- IsoHeat3D Material:
  - Temperature
  - Solution
  - State
  - Derivative
  - Flux
  - Stress
  - Sxx
  - Syy
  - Szz
  - ElemEnergy
  - TrueSolution
  - TrueError
  - TrueDerivative
  - TrueErrorDeriv
  - TrueElemEnergy
  - ErrorElemEnergy
  - TrueRelErrorElemEnergyNorm
  - TrueErrorElemEnergyNorm
- **TBD** —
  Document quantities for other material classes.

Examples

A few examples of usage follows:
In this example we create a VTK graph mesh and connect to file "6tet4_p1.vtu"

```
graphMesh create "6tet4_p1.vtu"  VTKStyle
```

The examples below set quantities to be post-processed (dumped to the post-processing file). The name of the quantities must be compatible with the type of problem been solved. Each quantity may be a vector or scalar. Below, we are solving an elasticity problem with known exact solution:

```
graphMesh appendName  Stress
graphMesh appendName  VonMises
graphMesh appendName  Sxx
graphMesh appendName  Syy
graphMesh appendName  Szz
graphMesh appendName  SolU
graphMesh appendName  Solution
```

In the example below, we dump post-processing quantities to the file previously connected to the graph mesh

```
graphMesh postProcess resolution  0
```

The option `resolution` controls the refinement of the mesh used to generate the graph mesh. Resolution equals to 0 means no mesh refinement, resolution equals to 1 means one level of refinement, etc.

4.2.34 createExactSol

The *Tcl* command `createExactSol` creates an object used to compute the exact solution for a problem and possibly the boundary conditions. The format for this command is:

```
createExactSol [localProb {probID #}]
               [ PolySol3D porder {px #} {py #} {pz #}
                           POUBasisT [Pascal | Tensor]
                           basisT [Pascal | Tensor]

               | PolySol2D porder {px #} {py #}
                           POUBasisT [Pascal | Tensor]
                           basisT [Pascal | Tensor]

               | EdgeSol3D edgeElasID  {edgeID #}
                           [ mode [ModeI | ModeII | ModeIII | ModeI_II
                                  | ModeI_II_III]
                           | setKVals KI {KI #} KII {KII #} KII {KIII #} ]
                           term {term #}

               | EdgeSolShadow edgeElasID  {edgeID #}
                           mode [ModeI]
                           term {term #}

               | EdgeSol2D edgeElasID {edgeID #}
                           mode [ModeI | ModeII | ModeI_II]
                           [ term {term #} | numTerms {num_terms #} ]

               | UserFunction userFn {functionID #}
               ]
```

- `PolySol3D` are function objects that defines polynomial functions in 3D. The p-order of the functions are `px`, `py`, `pz` in the x- y- and z-directions, respectively. The polynomials may be those from the Pascal pyramid or Tensor cube. Please take a look at documentation in source code of the classes `PolySol3D`, `PolySol2D`, `EdgeSol3D`, `EdgeSol2D` for more details.
- `UserFunction` The option for `UserFunction` allows for the use of a user-defined function as an exact solution. The parameter `functionID #` is the ID of the particular function defined in source code file `esuserfn.C`.
- `localProb`: See section 4.3.2.
- `setKVal`: This option allows the user to manually set the magnitude of the SIFs K_I, K_{II}, and K_{III} used in the exact solution instead of the default value of one. All values must be set if option `setKVal` is used, even if some values are zero, and `mode` is set to `ModeI_II_III`.

The command is mostly used by developers to create manufactured solutions used to verify the code.

4.2.35 enrichExactSol

The *Tcl* command enrichExactSol provides an interface to change the polynomial order of the object functions used to compute the exact solution. These objects are created with *Tcl* command createExactSol. The format for this command is:

```
enrichExactSol [localProb {probID #}] order {px #} {py #}  [ {pz #} ]
```

- localProb: See section 4.3.2.

4.2.36 parSet

The *Tcl* command parSet is used to set from a Tcl file, parameters for a GFEM analysis. This section contains the list of the parSet options which may be included in your .tcl file. These parameters are grouped according to the parameter type. Note that case sensitivity is important when selecting a parameter name but distinction between real parameters and integer parameters is not. Thus the data values 2 and 2.0 are both acceptable inputs for a real or an integer parameter. As a final note, there are a number of parameters that are simply toggle switches that turn on/off a particular option. For these parameters, the following options are equivalent:

to activate: true, on, 1
to de-activate: false, off, 0

In the examples below the toggle option which appears to be the most intuitive is shown, however, any of the other equivalent selections will provide the same effect.

Boundary Condition Control Parameters

```
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       exactSolAsBC [ [true | on | 1] | [false | off | 0] ]

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       exactSolAsPtBC [ [true | on | 1] | [false | off | 0] ]

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       exactSolAsBF [ [true | on | 1] | [false | off | 0] ]

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       exactSolAsIC [ [true | on | 1] | [false | off | 0] ]

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       ignoreBCs    [ [true | on | 1] | [false | off | 0] ]

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       directMethodForPtDirichletBC [ [true | on | 1] | [false | off |0] ]
```

The boundary condition control parameters set up basic information about the boundary conditions. The parameters associated with this class of constants include:

These parameters are used to request the use of boundary conditions which come from the exact solution instead of using those defined in .grf file. If this command is being in use, only the information about the *type* of boundary conditions is obtained from .grf file. These parameters are typically used when a manufacture solution was created with Tcl command createExactSol. Current parameters in this category are:

parSet exactSolAsBC true

flag (true, false) to use the solution of a problem to set up the values for boundary conditions defined in the .grf data file ,(default = false).

R If exactSolAsBC is set to true, exactSolAsPtBC is *also* set to true.

parSet exactSolAsPtBC true
: flag (true, false) to use the solution of a problem to set up the values for point boundary conditions defined in the .grf data file ,(default = false).

parSet exactSolAsBF true
: flag (true, false) to use the solution of a problem to set up body forces for a problem, (default = false).

parSet ignoreBCs true
: flag (true, false) to ignore all the boundary conditions prescribed in the .grf file, (default = false). This parameter is typically used with L^2 projection calculations. This allows the use of a data file originally set up for, let's say, elasticity, to perform an L^2 projection without the need to remove the boundary conditions from the data file.

directMethodForPtDirichletBC
: Whether to enforce point Dirichlet BC using direct method, instead of penalty method. Default = false.

Parameters related to Babuska Algorithm

This set of parameters are used to control the algorithm used to handle positive semi-definite matrices. Details on this so-called Babuska algorithm can be found in [**?**].

```
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       scaleAndStabAlgo   [ [true | on | 1] | [false | off | 0] ]

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       perturbationForBabuskaAlgorithm {epsilon #}

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       maxStabIterationsForBabuskaAlgorithm  {maxStabIterations #}

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       tolForStabilization     {tolForStab #}
```

scaleAndStabAlgo
: Turns on/off the algorithm. Default = true.

perturbationForBabuskaAlgorithm
: Set the magnitude of the perturbation used with Babuska algorithm. Default = 1.0e-12.

maxStabIterationsForBabuskaAlgorithm
: Set the maximum number of iterations allowed in Babuska algorithm. Default = 10

tolForStabilization
: Set the telerance for convergence of Babuska algorithm. Default = 1.0e-12.

```
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       scaleGlobalMatrixB4Fact   [ [true | on | 1] | [false | off | 0] ]
```

scaleGlobalMatrixB4Fact
: Turns on/off scalling of the global matrix just before factorization of the matrix.
Default = false.
This parameter should be set to true only for discretizations with positive definite matrices. In fact, it should always be set to true and parameter scaleAndStabAlgo set to false if

the global matrix is positive definite since it will lead to substantial memory savings when the Pardiso solver is adopted.

R This parameter has no effect if parameter scaleAndStabAlgo is set to true.

R This parameter has no effect if Tcl command scaleGlobalMatrix has been used before the factorization of the global matrix.

Parameters Controling Numerical Integration

```
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       addPorderForInteg {addPorder #}

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       reducePorderForInteg {reducePorder #}

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       singularIntegrationOrder  {sing_p_order #}

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
        lowerPorderForIntegEG [ [true | on | 1] | [false | off | 0] ]

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
        minNumDesc4LowerIntPorderEG {minNumDesc #}
```

`addPorderForInteg`
: Additional p-order to use when selecting quadrature rule for volume and BC elements. Rule will be selected based on actual p-order and `addPorderForInteg`. This parameter is mainly used to handle QA problems that fail due to minor under-integration of weak form. Default value is zero.

`reducePorderForInteg`
: This parameter has the opposite effect as `addPorderForInteg`. It thus leads to a quadrature rule with fewer points than the default one.
This parameter was designed for GFEM^{gl} analyses with Hexahedron elements in the global problem and fine local tetrahedron meshes. The default quadrature rules in this case uses more points than what is actually needed and leads to long assembly times. Details can be found in the Appendix of [**?**].

R This parameter is currently only used, if set, by Hexahedron elements – it will have no effect in meshes with other types of elements. It has no effect on quadrature rules used by BC elements either.

`singularIntegrationOrder`
: If `singularIntegrationOrder` is greater than zero, this parameter activates use of singular rule and its value is the equivalent p-order used to select singular integraion rules. Default = 0.

GFEM, SGFEM and Shifted Enrichments

```
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       SGFEMSpecialBasisEnrichments [ [true | on | 1] | [false | off | 0] ]
```

```
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       SGFEMBranchFnEnrichments [ [true | on | 1] | [false | off | 0] ]

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       SGFEMBranchFnEnrichmentsDiscInterp [ [true | on | 1] | [false | off | 0] ]

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       SGFEMPolyEnrichments [ [true | on | 1] | [false | off | 0] ]
```

`SGFEMSpecialBasisEnrichments`
: Turns on/off use of SGFEM enrichments at all nodes of GFEM mesh. Applies to special basis (non-polynomial) enrichments *only*. Default = `false`.

`SGFEMBranchFnEnrichments`
: Turns on/off use of SGFEM enrichment modification for all SpecialBasis that return isBranchFunction() == true. Default = `false`.

`SGFEMBranchFnEnrichmentsDiscInterp`
: Turns on/off use of SGFEM enrichment modification based on a discontinuous Finite Element interpolant for all SpecialBasis that return isBranchFunction() == true. Default = `false`.

> **R** A Discontinuous Interpolant coefficient can only be used at *Partition of Unity (PoU) nodes* that have a Heaviside or a `BranchHeavisideFnPair` enrichment. In the case of *p*FEM-GFEM we can additionally use a Discontinuous Interpolant coefficient at the PoU nodes of a patch, if the *enriched* (patch/cloud) node has a `BranchHeavisideFnPair` enrichment. Thus, this option is typically used only if parameter `useBranchHeavisideFnPairWithGeomEnrich` is true (see below). See [**?**] for details.

> **R** In the case of `BranchHeavisideFnPair` enrichments, both `SGFEMBranchFnEnrichments` and `SGFEMBranchFnEnrichmentsDiscInterp` will act only on the branch functions of such pair, leaving its Heaviside enrichment part unmodified.

`SGFEMPolyEnrichments`
: Turns on/off use of SGFEM polynomial enrichments at all nodes of a GFEM mesh. Default = `false`.

```
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       ShiftedSpBasisEnrichments [ [true | on | 1] | [false | off | 0] ]
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       ShiftedNotBranchFnEnrichments [ [true | on | 1] | [false | off | 0] ]
```

`ShiftedSpBasisEnrichments`
: If on, all non-polynomial enrichments are shifted by their nodal values. As a result, the enrichments will be zero at the associated node. Default = `false`.

> **R** If a node is enriched with a `BranchHeavisideFnPair` a Discontinuous Shiftting is performed on the (discontinuous) branch functions of the pair. Discontinuous shifting can only be used at nodes with a `BranchHeavisideFnPair` enrichment since it makes the branch function continuous. A *continuous* shifting is applied to step functions (always) and to branch functions that are continuous. See [**?**] for details.

`ShiftedNotBranchFnEnrichments`
: Similar to `ShiftedSpBasisEnrichments` but applies only to SpecialBasis that return

`isBranchFunction() == false`. Default = `false`. This parameter is intended to be used in conjunction with parameter `useSGFEMBranchFnEnrichments` or `SGFEMBranchFnEnrichmentsDiscInterp`, by setting both parameters to true.

R This command also applies a *continuous* shifting to the Heaviside function part of a `BranchHeavisideFnPair` enrichment.

R Polynomial enrichments are always zero at nodes and scaled by the radius of the node patch. Thus, there is no need to shift them.

```
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       pouDegree {p-order_pou #}
```

PFEM-GFEM Approximation Parameters

```
parSet [multiPhys {heat | thermoelas}]
       useCompElPfemGfem    [ [true | on | 1] | [false | off | 0] ]
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       maxElemSideDimWithBranchFnTopoEnrich  {sideDim #}
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       maxElemSideDimWithBranchFnGeomEnrich  {sideDim #}
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       maxElemSideDimWithHeavisideFnEnrich  {sideDim #}
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       maxpOrderHeavisideFnEnrich  {pOrder #}
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       useBranchHeavisideFnPairWithGeomEnrich [ [true | on | 1] | [false | off | 0] ]

Obsolete:
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       maxElemSideDimBranchHeavisidePairGeomEnrich  {sideDim #}

parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       maxElemSideDimWithLocalSolEnrich {sideDim #}
```

`useCompElPfemGfem`
: Turns on/off the use of pFEM-GFEM approximation spaces. If set to `true`, computational elements that implement pFEM-GFEM shape functions will be created during model reading from a `.grf` file. Thus, this parameter must be used before reading that file. This allows using the same `.grf` file to solve problems using either a GFEM space (default) or a pFEM-GFEM approximation space. Default = `false`.

`maxElemSideDimWithBranchFnTopoEnrich`
: Element sides with dimention higher than this parameter are not enriched with Branch Functions, regardless of the order of the CompNod on the element side. This parameter is not relevant when using GFEM spaces since only vertex nodes (dim = 0) are enriched in this case.
Used only for element sides in topological enrichment zone.
Default value = 0.

R There is a one-to-one relation between the sides of a computational element and its CompNods. If the dimension of an element side is equal to

0 This is a node on an element Vertex
1 This is a node at the Interior of a 1-D element or this is a node on an Edge of 2- or 3-D elem.
2 This is a node at the Interior of a 2-D element or this is a node on a Face of a 3-D elem.
3 This is a node at the Interior of a 3-D element.

`maxElemSideDimWithBranchFnGeomEnrich`
Analogous to parameter `maxElemSideDimWithBranchFnTopoEnrich` but used only for element sides in the geometrical enrichment zone.
Default value = 0.

R This parameter *must* be less or equal to parameter `maxElemSideDimWithBranchFnTopoEnrich`.

`maxElemSideDimWithHeavisideFnEnrich`
Element sides with dimention higher than this parameter are not enriched with Heaviside Functions, regardless of the order of the CompNod on the element side. This parameter is not relevant when using GFEM spaces since only vertex nodes (dim = 0) are enriched in this case.
Used for element sides in topological *and* geometrical enrichment zones.
Default value = 1.

`maxpOrderHeavisideFnEnrich`
Polynomial order of high-order Heaviside enrichment functions is restricted with this parameter. The value of the parameter is the highest order of Heaviside enrichment functions that will be used in the entire domain. This parameter is only valid for pFEM-GFEM simulations. This parameter will only make changes if it is lower than the global order of approximation. Otherwise, the polynomial order of Heaviside enrichment functions is controlled by `maxElemSideDimWithHeavisideFnEnrich`.
Default value = 4.

`useBranchHeavisideFnPairWithGeomEnrich`
Whether to use {Branch, Heaviside} pair in elements fully cut by crack surface and in the geometrical enrichment zone.
Default = false.

`maxElemSideDimBranchHeavisidePairGeomEnrich`
This parameter is no longer used. Parameter `maxElemSideDimWithBranchFnGeomEnrich` also controls the maximum element side dimension with {Branch, Heaviside} pair.
Element sides with dimention higher than this parameter are not enriched with {Branch, Heaviside} pair.
Default value = 0.

R This parameter *must* be greater or equal than parameter `maxElemSideDimWithBranchFnGeomEnrich` and less or equal than parameter `maxElemSideDimWithHeavisideFnEnrich`.

`maxElemSideDimWithLocalSolEnrich`
Element sides with dimention higher than this parameter are not enriched with local problem solution, regardless of the order of the CompNod on the element side. This parameter is not relevant when using GFEM spaces since only vertex nodes (dim = 0) are enriched in this case.
Default value = 0.

Miscellaneous Parameters

```
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       solveResidual [ [true | on | 1] | [false | off | 0] ]
```

R Parameter `solveResidual` can only be used when solving a linear problem (linear `Analysis`).

```
parSet [localProb {probID #}]  [multiPhys {heat | thermoelas}]
       enforceDescSearchGlobLocBC  [ [true | on | 1] | [false | off |0] ]
```

Whether to enforce the search of local descendants when applying boundary conditions to the local problem in a GFEM^{gl} simulation of crack propagation. At a given crack propagation step, the global problem is enriched with the solution of a local problem solved in the previous propagation step. The enriched global solution is then used to apply boundary conditions for another local problem – the one that will be solved at this propagation step. The use of these two local problems requires the enforcement of the aforementioned search. This command is only relevant for crack propagation problems solved with GFEM^{gl}.
Default: false

R It must be set to true when simulating crack propagation problems with the GFEM^{gl}.

```
parSet factorForScaledRoundOffTolerance {factor #}
```

Sets the factor used to scale the scaledRoundOffTolerance which is a round-off tolerance that takes into account the scale of the mesh and is used to check jacobians, check against zero, etc.
Default: 1.0

R This parameter is sometimes used when performing crack propagation simulations. A factor less then 1.0 can lead to substantially fewer missing computational crack surface facets in crack problems where the 3-D mesh is very refined (relative to the crack surface triangulation) around the front. Missing facets may be detrimental to problems with some physics on the crack surface, like in hydraulic fracture simulations.

R This parameter is a C++ static variable. Therefore, in a GFEM^{gl} simulation both global and local problems use the same `factorForScaledRoundOffTolerance`.

R Developers: Check also comments in code where variables `CrackGeoEng3D2::factor_for_vol_descd_jacobian_test` and `CrackGeoEng3D2::factor_for_face_descd_jacobian_test` are set.

4.2.37 parGet

The *Tcl* command `parGet` is used to get a parameter from the *ISET* database. The value of the parameter can then be used in the `Tcl file` driving a GFEM analysis. This section contains the list of the `parGet` options which may be included in your `.tcl` file.

```
parGet [localProb {probID #}] [multiPhys {heat | thermoelas}]  minEdgeLen
```

- minEdgeLen is the length of the shortest edge in a GFEM mesh.
- localProb: See section 4.3.2.
- multiPhys: See command analysis in Section 4.2.4.

```
parGet [localProb {probID #}] [multiPhys {heat | thermoelas}] condNumber
       [numZeroEigval {Num_zero_eigenvals} | tol {toler for "zero" eigenvals}]
```

- condNumber: With this argument, parGet returns the effective condition number of the global matrix of the problem computed using MATLAB. The effective condition number is the ratio of the largest eigenvalue and the smallest non-zero eigenvalue.
- numZeroEig is the number of known zero eigenvalues of the matrix. This is used to pick the smallest non-zero eigenvalue of the matrix.
- tol The smallest non-zero eigenvalue is taken as the smallest eigenvalue larger than tol. This option is typically used if numZeroEig is not known.
- If neither optional argument is given, tol is taken as 1.0e-15.

R Option ISET_USE_MATLAB must be enabled in CMake to use this command.
The global matrix should be scalled before calling this method by invoking *Tcl* function scaleGlobalMatrix.

```
parGet [localProb {probID #}] [multiPhys {heat | thermoelas}]  normResidual
```

R Please describe this parameter.

```
parGet [localProb {probID #}] [multiPhys {heat | thermoelas}]  lastTimeStep
```

R Please describe this parameter.

```
parGet [localProb {probID #}] [multiPhys {heat | thermoelas}]  normResidual_IGL
```

R Please describe this parameter.

4.2.38 getISETConfig

The *Tcl* command getISETConfig is used to get configuration parameters selected at *ISET* compilation time.

```
getISETConfig [whichSolver isMatlab isAbaqusUserSubs]
```

- whichSolver returns either “Pardiso” or “Cholmod”.
- isMatlab returns either “true” or “false”.
- isAbaqusUserSubs returns either “true” or “false”.

4.3 Tcl Commands for a GFEM^{gl} Analysis

This section described the *Tcl* commands used in the GFEM with global-local enrichments (GFEM^{gl}). Details on this method can be found in [**?**].

4.3.1 createLocalProblem

The *Tcl* command createLocalProblem is used to create a local problem whose solution can be used as an enrichment function in the GFEM with global-local enrichments (GFEM^{gl}). The local problem is a special basis objects and can be assigned to nodes of the global problem like any other special basis. The format for this command is:

```
createLocalProblem probID {probID #} [[linear] | matNonlinear |
                                                | matVisco | transient
                                                | transientLocal ]
                    [ geoNodeID {geoNodeID #}
                    | xyzMin {Xmin #} {Ymin #} {Zmin #}
                      xyzMax {Xmax #} {Ymax #} {Zmax #}
                    | crackMgID {crackMgID #}
                       [[frontOnly] | allElems ] ]
                    | readFromFile {file_name} [[solid] | heat] ]
                     numLayers {nLayers #}
                     userBC {userBCName}
```

- probID defines the ID of the created local problem. This ID can be used by other *Tcl* commands through their optional parameter localProb as described later.
- linear, matNonlinear, matVisco and transient define the type of analysis to be performed by the local problem.
- The domain of the local problem is defined using a set of *seed nodes* and parameter numLayers. The set of *seed nodes* can be defined in several ways:
  - geoNodeID defines a *seed node* equal to the GeoNod with ID geoNodeID.
  - xyzMin and xyzMax defines a bounding box containing the set of *seed nodes*.
  - crackMgID define seed nodes for the local domain creation using either
    - frontOnly the nodes of all elements intersected by the front of the crack with ID crackMgID;
    - allElems the nodes of all elements intersected by the surface of the crack with ID crackMgID.

    R crackMgID option to define seed nodes is not operational with the current Crack Manager used in *ISET*.

- numLayers defines the number of layers of global elements used in the definition of the local domain, as described next. It must be greater or equal to one.
  Given a set of seed nodes, a domain Ω_L^1 is defined by the set of global elements sharing any one of the set of seed nodes. If numLayers is equal to one, the local domain is given by Ω_L^1. If numLayers is equal to two, the set of nodes used by all global elements in Ω_L^1 defines a new set of seed nodes used in the definition of Ω_L^2. The process is repeated as many times as needed if numLayers is greater than two. See [**?**, **?**] for further details.

  R If numLayers is greater than one, a buffer zone defined by layers of global elements is used in the definition of the local domain. Mode details on this concept can be found in [**?**].

  R The set of seed nodes defined by the user in command createLocalProblem is typically used to enrich those global nodes with the solution of the local problem defined by the command.

- `readFromFile` option specifies that the local problem is defined in a `.grf` file named `file_name`. The type of material used by the local problem can be either `solid` or `heat`. The boundary conditions applied to the local problem read from a `.grf` file are determined based on
  a. whether the local boundary (Γ_{loc}) intersects the global boundary (Γ_{glob}) or not;
  b. whether there are user-defined boundary conditions on Γ_{loc} in the local problem `.grf` file;
  c. whether there are user-defined boundary conditions on Γ_{glob} in the global `.grf` file.
  There can be four basic scenarios:
  1. a = true, b = true, c = true/false: The local BCs specified by the user in the local `.grf` are assigned to Γ_{loc}, as specified in the local `.grf`, even on portions of Γ_{loc} that intersects Γ_{glob}. Portions of Γ_{loc} with no BC assigned in the local `.grf` are assigned BCs following the usual rules (either a global-local BC or a BC assigned to Γ_{glob}).
  2. a = true, b = false, c = true/false: BCs are assigned to Γ_{loc} following the usual rules: either a global-local BC or a BC assigned to Γ_{glob}.
  3. a = false, b = true, c = not relevant: The local BCs specified by the user in the local `.grf` are assigned to Γ_{loc}, as specified in the local `.grf`. Portions of Γ_{loc} with no BC assigned in the local `.grf` are assigned global-local BCs.
  4. a = false, b = false, c = not relevant: Global-local BCs are assigned to entire Γ_{loc}.

  In summary, local BCs defined in the local `.grf` always overwrite the global BCs when the global and local boundaries coincide. If the local boundary is inside the global domain, the global-local type BCs are used.

  R Local problems do not inherit any Point BCs from the global problem even when a global node with a Point BC coincides with a local node. The user needs to define local Point BCs either in the local `.grf` file or through the *Tcl* command `assignPtBC`.

- `userBCName` defines the type of boundary conditions applied on the portions of the local boundary that do not intersect the boundary of the global problem. A boundary condition named `userBCName` must have been defined in the `.grf` file of the global problem.

4.3.2 Optional Paramerer `localProb`

The local problem created with `Tcl` command `createLocalProblem` can be refined, enriched, etc., like the problem used in a standard GFEM analysis. All `Tcl` commands that can be applied to either a global or a local problem have an optional parameter `localProb`. This parameter can be used to provide the `ID` of the local problem to which the `Tcl` command should be applied to.

4.3.3 `assembleTransLoc`

The *Tcl* command `assembleTransLoc` is used to assemble a transient local problem

```
assembleTransLoc [localProb {probID #}] [multiPhys {heat}] Time {Time #}]
```

- `localProb`: See section 4.3.2.
- `multiPhys`: See command `analysis` in Section 4.2.4.

R TBD: The description of this command needs improvements.

4.3.4 `solveTransLoc`

The *Tcl* command `solveTransLoc` is used to solve a transient local problem.

```
solveTransLoc [localProb {probID #} [multiPhys {heat}] Time {Time #} deltaT {deltaT #}]
```

- localProb: See section 4.3.2.
- multiPhys: See command analysis in Section 4.2.4.

TBD: The description of this command needs improvements.

4.3.5 l2ProjectInitialCondLocal

```
l2ProjectInitialCondLocal [ localProb {probID #} ]
                          [ multiPhys { heat | thermoelas } ]
                          useGlobalInitCond { true | false }
                          Time { Time # }
                          deltaT { deltaT # }
                          storeSolution { true | false }
```

- multiPhys: See command analysis in Section 4.2.4.

TBD: Please provide a description of this command.

4.3.6 printFPOSLocProbs

The *Tcl* command printFPOSLocProbs prints the number of floating point operations required for factorization of stiffness matrix of each (sub-)local problem. The syntax of the command is:

```
printFPOSLocProbs
```

TBD: The description of this command needs improvements.

4.3.7 MasterLocalProblem

The *Tcl* command MasterLocalProblem is used when performing a GFEM[gl] analsysis with the Master-Sub-LocalProblem concept introcuded in [**?**]. The syntax of the commands is a follows

```
MasterLocalProblem msProbID {probID #}
                   [ multiPhys {heat | thermoelas | thermoplas} ]
                   [ create  [[linear] | matNonlinear | transient]
                             [ geoNodeID {geoNodeID #}
                             | xyzMin {Xmin #} {Ymin #} {Zmin #}
                               xyzMax {Xmax #} {Ymax #} {Zmax #}
                             | readFromFile {file_name} [[solid] | heat]
                             | crackMgID {crackMgID #} ]
                             numLayers {nLayers #}
                             userBC {userBCName}
                   | defineSubProblems
                        [ oneLocProbPerCloud |
                          clusterSeedNodes |
                          numSubLocProbs {numSub #} ]
                   | defineSeedNodes [ geoNodeID {geoNodeID #}
                                     | xyzMin {Xmin #} {Ymin #} {Zmin #}
                                       xyzMax {Xmax #} {Ymax #} {Zmax #} ]
                   | defineSeedNodsCluster [ xyzMin {Xmin #} {Ymin #} {Zmin #}
```

```
                                xyzMax {Xmax #} {Ymax #} {Zmax #} ]
| defineClustersWithHSFC [ numClusters {nClusters #}
                          | xyzMin {Xmin #} {Ymin #} {Zmin #}
                            xyzMax {Xmax #} {Ymax #} {Zmax #} ]
| process
    [ noManager | crackMg | grainMg ]
| enrichCompNodSubProbs
     [multiPhys {heat | thermoelas | thermoplas}]
     all [ hasStepFn | hasBranchFn ]
     [nodeType {vertex | edge | face
                | interior | any} ]
     [iso | xi | eta | zeta] order {order #}
| setSpBasisSubProbs
     all [ hasStepFn | hasBranchFn ]
     maxNumCustomTerms {maxTerms #}
| setCompNodSubLocalProb
| sortSubProbs {ascend | descend}
| setParallel
| setAnalysisOptionsSubProbs
     [multiPhys {heat | thermoelas | thermoplas}]
| assemble [multiPhys {heat | thermoelas}]
| solve [multiPhys {heat | thermoelas}]
| parallelAssemble [multiPhys {heat | thermoelas}]
| parallelSolve [multiPhys {heat | thermoelas}]
| transientAssemble Time {# Time}
| transientSolve Time {# Time}
| nonlinearSolve [multiPhys {thermoplas}]
                 [ timeStep {# istep} ]
| nonlinearAssemble [multiPhys {thermoplas}]
                 { timeStep {# istep} }
| linearSolve [multiPhys {thermoplas}]
                 [ timeStep {# istep} ]
| graphMesh
    [ create {file_name} [DXStyle | TecPlotStyle |
                          HyperMeshStyle | VTKStyle]
                         [ascii | binary]
    | open   {file_name}
    | postProcess resolution {resol #} [sampleInside]
      [solution {sol_number}] ]
    | postSurface
    | postEnrichment
]
```

- `multiPhys`: See command `analysis` in Section 4.2.4.

R TBD: The description of this command needs LOTS of improvements.

4.4 Tcl Commands for a IGL-GFEM^{gl} Analysis

This section describes the *Tcl* commands used in the non-intrusive IGL-GFEM^{gl}. Details on this method can be found in [**?**, **?**].

```
IGLAlgorithm {  create [useShellSolidCoupling {true | on | 1 | false | off | 0} ]
              | readAndProcessInterfaceData {file_name}
              | computeAndDumpInterfaceData {file_name}
              | setElemListLoc xyzMin {Xmin #} {Ymin #} {Zmin #}
                               xyzMax {Xmax #} {Ymax #} {Zmax #}
              | useShellSolidCoupling {true | on | 1 | false | off | 0}
              | readAndApplyInterfaceResForce {file_name}
              | readInterfaceResForceLoc {file_name}
              | readInterfaceResForceLocPrev {file_name}
              | computeAndDumpInterfaceResForce {file_name}
                               { NoAcceleration | DynamicRelaxation }
            }
```

R Please describe this command.

4.5 Tcl Commands for a Fracture Analysis

This section describes *Tcl* commands used in the solution of fracture mechanics problems. Such analyzes are usually performed with the aid of a *Crack Manager*. However, in some simple problems, the analysis can be performed without a *Crack Manager*. The *Tcl* commands for these simple cases are described first, followed by the description of command `crackMgr`

4.5.1 createSIFExtractor

The *Tcl* command `createSIFExtractor` is used to create an object that extracts crack front quantities, such as the stress intensity factors (SIF) and energy release rate, from GFEM solutions. The syntax for this command is as follows:

```
createSIFExtractor [ [CIM2D | CIM3D | Jintegral2D | Jintegral3D | DCM3D ]
                        edgeElasID  {edgeID #}
                     | CFM2D nodeOrigin {nodeID #} nodeXAxis {nodeID #}
                     | CFM3D nodeOrigin {nodeID #} nodeZAxis {nodeID #}
                        nodeXZPlane {nodeID #}
          [ [ tracFace+ {t_n #} {t_t #}  tracFace- {t_n #} {t_t #} |
              tracFace+ {t_n #} {t_t #} {t_b #}
              tracFace- {t_n #} {t_t #} {t_b #} ] ]
```

The SIF in two-dimensional and three-dimensional problems can be extracted using *Contour Integral Method* (`CIM2D` and `CIM3D`), *Cut-off Function Method* (`CFM2D` and `CFM3D`), and *J-integral* (`Jintegral2D` and `Jintegral3D`). The J-integral implementation extracts crack front quantities for single mode only while CIM and CFM can extract crack front quantities in single as well as mixed modes. The extraction object requires an edgeElas object during its creation. Using the option `edgeElasID` the user *must* provide an edgelas object with ID `edgeID`. The current implementation of CIM and CFM also provides an interface to extract SIFs and energy release rate for loaded crack problems. Using options `tracFace+` and `tracFace-` the user can provide normal (`t_n`) and tangential (`t_t`) tractions on the crack faces.

- TBD

R TBD: The description of this command needs improvements.

4.5.2 extractSIF

The *Tcl* command `extractSIF` provides an interface for the extraction object. The syntax for this command is as follows:

```
extractSIF [discreteExtract | DCM]
            {  [ extrDist_1 {extrDist_1 #} extrDist_2 {extrDist_2} ]
             | [ extrDist_min {extrDist_min #} deltaDist {deltaDist #}
                 numExtrPoints {numExtrPoints #}
                 [averLinExtrapl | leastSqrExtrapl]
               ]
               [ projPntOnCrackSurface ]
            }
           | [contourInt] radius {radius}
                          [ numIntegPoints {nint #} [ term {term #} ] ]
           | [domainInt]  radius_1 {radius_1} radius_2 {radius_2}
                          [ cylThickness {zThickness} ]
                          [ numIntegPoints_r  {nint_r #}
                            numIntegPoints_t  {nint_t #} ]
```

Through the option `radius` the user can set the radius of integration around the crack front. Using the option `numIntegPoints` the user can also set the number of integration points used in the extraction. The CIM and CFM methods are also able to extract not only the stress intensity factors but also other terms of the assymptotic expansion. Option `term` provides an interface for extracting a given term with number `term` of the asymptotic expansion.

- TBD
- `localProb`: See section 4.3.2.
- `multiPhys`: See command `analysis` in Section 4.2.4.

R TBD: The description of this command needs improvements.

4.5.3 crackMgr

The *Tcl* command `crackMgr` is used when solving three-dimensional fracture mechanics problems. Non-planar and non-smooth three-dimensional crack surfaces are explictly represented by surface meshes composed of triangles [**?**, **?**]. The crack can growth under any combination of Mode I, II and III loading [**?**]. The creation of multiple `crackMgr` objects when solving problems with several cracks, is also supported. See description of command `masterCrackMgr` in Section 4.5.4. Command `crackMgr` provides several options to control the crack surface evolution during a crack growth simulations. They are described below.

```
crackMgr [ localProb {probID #} ]  crackMgrID {crackMgrID #}
       {  create
          { crackFile {file_name}
            |
            crackSurfaceFromCrackMgr {crackMgrIDforCreation #}
          }
```

```
| process
        [ localProb {probID #} [ frontOnly | allElems] ]
| postSurface {file_name}
        [ DXStyle | crfStyle | VTKStyle | ParaviewStyle ]
| postComputationalSurface [aboveCrack | belowCrack] {file_name}
        [ DXStyle | VTKStyle | ParaviewStyle ]
| postEnrichment {file_name}
        [ DXStyle ]
| postAllCrackCutElInfo {file_name}

| refineMesh
            { crackFronts [ maxEdgeLen {max_len#} | minEdgeLen {min_len#}
                            [inCylinderRadius {radCyl#}]
                          ]
            | crackVertices
            | crackSurface [maxEdgeLen {max_len#} [distToFront {dis_front#}] ]
            }

| enrichApprox
     { numElemLayers {nLayers #} | inCylinder radius {R #} }
     [iso | xi | eta | zeta] approxOrder {approxOrder #}

| branchFnGeometricEnrich { numElemLayers {nLayers #} | inCylinder radius {R #}
                           | inBBox xyzMin {xMin #} {yMin #} {zMin #}
                                    xyzMax {xMax #} {yMax #} {zMax #} }
| setCompNodLocalProb
     localProb {probID #}
| extract
   {     DCM extrDist_1 {extrDist_1 #} extrDist_2 {extrDist_2 #}
          [ projPntOnCrackSurface ]
     |
         DCM extrDist_min {extrDist_min #} deltaDist {deltaDist #}
             numExtrPoints {numExtrPoints #} [averLinExtrapl | leastSqrExtrapl]
          [ projPntOnCrackSurface ]
    |
         CIM radius {radius #} [numIntegPoints {nint #}]
     |
         CFM  radius_1 {rho1 #} radius_2 {rho2 #}
              cylThickness {zThickness}
             [numIntegPoints_r {nint_r #}
               numIntegPoints_t {nint_t #}]
   }
     [ tracFace+ {t_n #} {t_t #} {t_b #}
       tracFace- {t_n #} {t_t #} {t_b #} ]

     [ start_time {stime #} end_time {etime #}  nSteps {nsteps #} ]

| cleanUp { intersectData | afterPropagation }
```

```
| advanceFront
    {    DCM  extrDist_1 {extrDist_1 #} extrDist_2 {extrDist_2 #}
          [ projPntOnCrackSurface ]
     |
         DCM extrDist_min {extrDist_min #} deltaDist {deltaDist #}
            numExtrPoints {numExtrPoints #} [averLinExtrapl | leastSqrExtrapl]
          [ projPntOnCrackSurface ]
     |
         CIM radius {radius #}
            [ numIntegPoints {nint #} ]
     |
         CFM radius_1 {rho1 #} radius_2 {rho2 #}
             cylThickness {zThickness}
            [numIntegPoints_r {nint_r #}
             numIntegPoints_t {nint_t #}]
    }
      [ tracFace+ {t_n #} {t_t #} {t_b #}
        tracFace- {t_n #} {t_t #} {t_b #} ]

    [ start_time {stime #} end_time {etime #} ]

| remeshCrackSurface
| cleanUpStoredCrackFronts

| setFluidMatForCrackMgr {fluidMat_name} pressureBCForInitialGuess {bc_name}

| setConnectingPipeMatForCrackMgr {fluidMat_name}

| prescribeFluidFlow { inBBox xyzMin {xMin #} {yMin #} {zMin #}
                              xyzMax {xMax #} {yMax #} {zMax #}
                      | inSemiTorus r1 {#} r2 {#} z1 {#} z2 {#} [xc {#} yc {#}]
                      | atPoint {x #} {y #} {z #} }
                    {volume {vol #} | volumeFromFile {filename}} }

| prescribePressureKrauklis { inBBox xyzMin {xMin #} {yMin #} {zMin #}
                                    xyzMax {xMax #} {yMax #} {zMax #}
                             | inSemiTorus r1 {#} r2 {#} z1 {#} z2 {#} [xc {#} yc {#
                             | atPoint {x #} {y #} {z #} }
                         { pressure {pres #}
                         | pressureFunctNumber {fn #} [pressureFrequency {freq #}]
                         | pressureFromFile {filename} }
                            }

| connectPipeWithFrac {  inSemiTorus r1 {#} r2 {#} z1 {#} z2 {#}
          | inBBox xyzMin {Xmin #} {Ymin #} {Zmin #}
                   xyzMax {Xmax #} {Ymax #} {Zmax #}
          | atPoint {x #} {y #} {z #} }
          pipeNodeID {ID #}
```

```
| setCornerNodes { node1 {x #} {y #} {z #} node2 {x #} {y #} {z #} }

| {setOptions | parSet}
      {
        usePlanarCutsSignedDist |
        useBranchFn |
        useSingularBranch |

        useMLSForExtraction |
        useMLSForSkippedFrontEndVert |

        ignoreKII |
        ignoreKIII |

        useMLSForCrackFrontVertices |
        useMLSForCrackFrontVerticesWhileRemeshing |
        useUniformCrackFrontVertices |

        useSimplifyCrackSurface |
        useCompleteSimplifyCrackSurface |
        useRemeshCrackSurface|
        storeCrackFront |

        viscoExtractionCP |
        viscoFourierTransf |
        viscoForceControlled

        { [true | on | 1] | [false | off | 0] }
      |

        MLS_overlappingFactor  |

        delta_a_max            |
        frontEdgeLengthLimit   |
        crackSurfEdgeLength    |

        scaleForCrackFrontVerticalDisplacement |
        accumulatedCrackFrontAdvanceLimit |
        frontStretchTol |

        viscoPeakTime

        {double_precision_option_value #}
      |
        useViscoExtractionCP viscoMatName {name_viscoelas_mat}
      |
        viscoLoadType {constant | ramp | creep | hat | freq | haversine}
      |
        viscoIntegrType {LaplaceTransf | GaussIntegr | incremental}
```

```
        |

          MLS_degree                    |
          MLS_numSamplePointsFrontEdges |

          compSIFsAfterEveryNVerts      |
          numFrontEndVertToSkip

          {integer_option_value #}
        |

          applyBCtoCrackFaces {bc_name}
        |
          useCohesive {bc_name}
        |
          branchFunctionType { straightFrontOD | straightFrontBB | straightFrontOY
                             | curvedFrontBB | curvedFrontOD
                             | curvedFrontOD6 | curvedFrontOD9 }

        | IDNeighborMgr4Coalesce {crackMgrID #}
        | surfCoalesceTol        {tol_distance #}

        | snapNodesToSurfAndFront snappingTolSurf {tol #} [snappingTolFront {tol #}]
        | snapNodesToSurfAndFront { [true | on | 1] | [false | off | 0] }

        }

    | parGet { minElemEdgeLenAtCrackFront
             | isCrackArrested
             }

    | crGrowthPhysicsLaw
        {
          create

        | crGrowthType
          { FatigueCrackGrowth
            |
            StableCrackGrowth {Gc_material {user_defined_Gc # }}
          }

        | crFrontIncrementType
          { fixedDeltaA {us_def_incr #}
            |
            adaptiveDeltaA
                     [ setOptions
                         {
                           Alpha {#Double_value} |
                           Beta  {#Double_value} |
```

```
                    restart
                  }
              ]
  }

|
  frontKinkingAngleLimitDEG {double_precision_option_value #}
|
  frontTwistingAngleLimitDEG {double_precision_option_value #}

| print

| crFrontScalingLaw
  { ParisLaw
            { C {us_def_C #}
              m {us_def_m #}
              R {us_def_R #}
            }
    |
    MearLaw
            { aZero {us_def_aZero #}
              m {us_def_m #}
              KZero {us_def_KZero #}
              KIC {us_def_KIC #}
            }
    |
    MearLawQuasiStatic
            { m {us_def_m #}
              KIC {us_def_KIC #}
            }
    |
    FormanLaw
            { C {us_def_C #}
              m {us_def_m #}
              R {us_def_R #}
              Kth {us_def_Kth #}
              KC {us_def_KC #}
            }
    |
    MearLaw2
            { alpha {us_def_alpha #}
              beta {us_def_beta #}
              KIC {us_def_KIC #}
            }
    |
    GuptaDuarteLaw
            { KIC_material {KIC_material #}
              alpha {alpha #}
              [ m_alpha {m_alpha #} ]
```

```
                        beta {beta #}
                        [ m_beta {m_beta #}   ]
                      }
                      | { setVariationInAlphaRegion {true | false} }
                |
                ConstFrontDispl
                |
                setConstFrontDispl [{true | on | 1} | {false | off | 0}]

            } # end of sub-sub command crFrontScalingLaw

        } # end of crGrowthPhysicsLaw sub commands

    } # end of crackMgr sub-commands
```

Description of `CrackMgr` Sub-Commands and Options:

- `[ localProb {probID #} ]`

  A `CrackMgr` can be created for either a local or a global problem in a GFEM^{gl} analysis. Optional argument `localProb` is used to create a `CrackMgr` for a local problem with ID `probID`. See also section 4.3.2.

- `crackMgrID {crackMgrID #}`

  This argument specifies the unique integer ID of each `crackMgr` object.

- `{ crackFile {file_name} |`
  `crackSurfaceFromCrackMgr {crackMgrIDforCreation #} }`

  Creates a `CrackMgr` object with the specified ID using one of the following options to define the crack surface:
  - `crackFile`: The crack surface is defined in a `.crf` file. This file contains information about the 3-D triangulation representing the crack surface: The coordinates of surface vertices, connectivities of surface facets, and the crack front definition. The format of this file is described in Chapter 5.
  - `crackSurfaceFromCrackMgr`: The crack surface from a previously created `CrackMgr` object with ID `crackMgrIDforCreation` is *shared* with this crack manager object. This option is useful when solving fracture problems using the GFEM^{gl}.

- `process [ localProb {probID #} [ frontOnly | allElems] ]`

  Process the crack: Create and assign enrichments to nodes whose clouds/patches interact with the crack surface; creates integration descendants used for the numerical integration of weak form.

  Sub-options

  `[ localProb {probID #} [ frontOnly | allElems] ]`

  are currently not available.

- `postSurface {file_name} [DXStyle | crfStyle | VTKStyle | ParaviewStyle]`

  Writes the crack surface to a file named `file_name` using one of the file styles listed above.

File styles VTKStyle and ParaviewStyle are identical and produce a file in the VTK format used by Paraview (http://www.paraview.org/).
File style crfStyle is identical to the file format adopted in the .crf file used in the creation of a crackMgr object. This file style is typically used to re-start a simulation using the crack surface computed in a crack propagation simulation.
File style DXStyle is deprecated.

- postComputationalSurface [aboveCrack | belowCrack] {file_name}
  [DXStyle | VTKStyle | ParaviewStyle]

  Writes the computational crack surface to a file named file_name. Such surface is defined by the union of faces of 3-D integration elements that are on the crack surface. The computational crack surface is used to, e.g., compute boundary conditions on the crack surface. It also defines where the crack surface discontinuity is located and thus its visualization can be quite useful for debugging purposes. Reference [**?**, **?**] explains the difference the computational and the geometrical crack surfaces. There are two computational crack surfaces. One on the above (positive) side and one on the below (negative) side of the geometric crack surface. The user can select to dump the above surface using option aboveCrack or the below surface using option belowCrack. By default, both surfaces are dumped to the file. The edges of the triangulation defining the above and below surfaces do not necessarily match when the crack surface is at the boundary of computational elements. Further details can be found in [**?**, **?**].

  – The crackMgr sub-command postComputationalSurface should only be called after executing the sub-command processCrack.
  – The identification of integration element faces that are on the crack surface is only performed if crackMgr sub-command applyBCtoCrackFaces is used (before sub-command processCrack). crackMgr sub-command applyBCtoCrackFaces is described later. If a problem does not have boundary conditions on the crack surface, crackMgr sub-command applyBCtoCrackFaces with bc_name equal to "none" can be used to enable the construction and visualization of the computational crack surfaces.

  One of the following file styles can be selected for the computational crack surface

  [DXStyle | VTKStyle | ParaviewStyle]

  File styles VTKStyle and ParaviewStyle are identical and produce a file VTK format used by Paraview (http://www.paraview.org/).
  File style DXStyle is deprecated.

- postEnrichment {file_name} [DXStyle]

  This sub-command is deprecated. It has been replaced by sub-command postAllCrackCutElInfo.

- postAllCrackCutElInfo {file_name}

  Dumps to a file in VTK format all CompNod enrichments and all intersections (between the crack surface and GeoMesh) stored in all CrackCutElInfo2 objects of the Crack Manager.

  This command is meant to be used by developers for debugging purposes.

- refineMesh { crackFronts [maxEdgeLen {max_len#} | minEdgeLen {min_len#}
  [inCylinderRadius {radCyl#}]
  ]

```
| crackVertices
| crackSurface [ maxEdgeLen {max_len#} [distToFront {dis_front#}] ]
}
```

The Crack Manager refines the elements of the 3D problem mesh using one of the following criteria

- crackFronts [maxEdgeLen {max_len#} | minEdgeLen {min_len#} [inCylinderRadius {radCyl#}]]
  Any 3D element that intersects the crack front is refined (bisected) once. If optional arguments maxEdgeLen or minEdgeLen are provided, the refinement is performed until the maximum or minimum element edge length among all elements intersected by crack front, is smaller then maxEdgeLen or minEdgeLen, respectively.
  If optional argument inCylinderRadius is provided, refinement is performed on all elements (cracked or not) within a cylinder around the crack front(s) and with radius radCyl#.
- crackVertices
  Any 3D element that intersects the crack front ends is refined (bisected) once. This command is only relevant for surface breaking cracks since an interior crack has no crack front end.
- crackSurface [maxEdgeLen {max_len#} [distToFront {dis_front#}]]
  Any 3D element that intersects the crack surface is refined (bisected) once. If optional argument maxEdgeLen is provided, the refinement is performed until the maximum edge length among all elements intersected by the crack surface, is smaller then maxEdgeLen. If optional argument distToFront is provided, the following additional condition must be met for an element be refined: Its centroid must be within distance dis_front# from the crack front.

> R It is noted that since only cracked elements are refined by this option, no refinement ahead of the crack front is performed. This is in contrast with option crackFronts when used with optional argument inCylinderRadius.

For all options to command refineMesh, additional elements are also refined in order to recover a conforming mesh.

- enrichApprox {numElemLayers {nLayers #} | inCylinder radius {R #} } [iso | xi | eta | zeta] approxOrder { approxOrder #}

The Crack Manager enriches Computational Nodes (CompNods) of the 3D mesh such that the polynomial order of the GFEM approximation over the support of these nodes is equal to approxOrder. This command has the same functionality as the enrichApprox command described in Section 4.2.5 except that it is applied to a set of nodes, instead of all nodes of the GFEM mesh. The set of nodes is selected using one of the options:

- numElemLayers {nLayers #}
  Enrich CompNods from the problem mesh belonging to all elements within the first nLayers layers around the crack front(s).
- inCylinder radius {R #}
  Enrich CompNods from the problem mesh within a geometrical enrichment zone defined by a cylinder of radius R around the crack front(s).

There are two important points to remember when using this crackMgr sub-command

- This command must be called after a call, if any, to the "global" enrichApprox command described in Section 4.2.5. The latter command enriches the whole mesh and it will overwrite the crackMgr sub-command enrichApprox is called after it.

– When adopting a pFEM-GFEM space (parameter `useCompElPfemGfem` is true), new edge and face nodes may be activated with this sub-command. If, for example, Heaviside enrichment is desired on these new nodes, the command

```
parSet maxElemSideDimWithHeavisideEnrich {sideDim #}
```

must be called as described in Section 4.2.36 with `sideDim` set to the highest dimension of the newly created nodes. This will assign Heaviside Functions at all deserving nodes of dimension less than or equal to the set value, but only activate those where the local polynomial enrichment has been applied by the `crackMgr` sub-command.

- `branchFnGeometricEnrich { numElemLayers {nLayers #}`
  `| inCylinder radius {R #}`
  `| inBBox xyzMin {xMin #} {yMin #} {zMin #}`
  `xyzMax {xMax #} {yMax #} {zMax #} }`

  The commands activates and defines the branch function geometrical enrichment around the crack front(s). If this sub-command is used, branch functions are created and assigned to all `CompNods` within the geometrical enrichment region when the `Crack Manager` sub-command `process` is invoked. The geometrical enrichment region can defined using the following options:
  – `numElemLayers {nLayers #}`
  Assign branch functions for `CompNods` from the problem mesh belonging to all elements within the first `nLayers` layers around the crack front(s).
  – `inCylinder radius {R #}`
  Assign branch functions for `CompNods` from the problem mesh within a cylinder of radius `R` around the crack front(s).
  – `inBBox xyzMin {xMin #} {yMin #} {zMin #}`
  `xyzMax {xMax #} {yMax #} {zMax #}`
  Assign branch functions for `CompNods` from the problem mesh within a bounding box defined by the lower-left and upper right corners `xyzMin` and `xyzMax`, respectively.

  R This commands assigns branch functions to a set of nodes. However, if the special basis will actually be used depends on the order set for the nodes. The order of a node must be equal to one or higher for its branch function be activated during the solution of the problem. See also `Tcl` command `setCompNodSpBasis` in Section 4.2.7.

- `setCompNodLocalProb localProb {probID #}`

  This sub-command is currently not available.

- `extract`

```
{   [ DCM extrDist_1 {extrDist_1 #} extrDist_2 {extrDist_2 #}
          [projPntOnCrackSurface] ]
  |
    [ DCM extrDist_min {extrDist_min #} deltaDist {deltaDist #}
          numExtrPoints {numExtrPoints #} [averLinExtrapl | leastSqrExtrapl]
          [projPntOnCrackSurface]
    ]
  |
    [ CIM radius {radius #} [numIntegPoints {nint #}] ]
  |
    [ CFM  radius_1 {rho1 #} radius_2 {rho2 #}
           cylThickness {zThickness}
```

```
            [numIntegPoints_r {nint_r #}
             numIntegPoints_t {nint_t #}] ]
}
  [ tracFace+ {t_n #} {t_t #} {t_b #}
    tracFace- {t_n #} {t_t #} {t_b #} ]

  [ start_time {stime #} end_time {etime #}  nSteps {nsteps #} ]
```

This sub-command is used to extract stress intensity factors and energy release rate along the crack front(s). These quantities are extracted at every crack front vertex of the explicit representation of the crack surface (cf. Chapter 5). The following extraction methods are available:

- `DCM extrDist_1 {extrDist_1 #} extrDist_2 {extrDist_2 #} [projPntOnCrackSurface]`
  The Displacement Correlation Method is used with two extraction/sampling points at distances `extrDist_1` and `extrDist_2` from the crack front. The jump of the displacement is computed at the sampling points. If optional flag `projPntOnCrackSurface` is given, extraction points are projected to the (geometrical) crack surface. This only matters if crack surface is curved in the crack front forward direction. Details on this method can be found in [**?**].
- `DCM extrDist_min {extrDist_min #} deltaDist {deltaDist #} numExtrPoints {numExtrPoints #} [averLinExtrapl | leastSqrExtrapl] ] [projPntOnCrackSurface]`
  The Displacement Correlation Method is used with the following parameters:
  - `extrDist_min` is the distance from the crack front to the first point where the displacement jump is computed.
  - `deltaDist` is the distance between extraction/sampling points where the displacement jump is computed.
  - `numExtrPoints` is the number of extraction/sampling points where the displacement jump is computed.
  - `averLinExtrapl` SIFs are first computed using a linear extrapolation for each pair of extraction points. The average of the extrapolated values is then taken.
  - `leastSqrExtrapl` SIFs are computed using a linear least squares fitting based on all sampling points. A linear extrapolation based on the least squares fitting is then taken. If optional flag `projPntOnCrackSurface` is given, extraction points are projected to the (geometrical) crack surface. This only matters if crack surface is curved in the crack front forward direction. Further details on this and the last option can be found in [**?**].
- `CIM radius {radius #} [numIntegPoints {nint #}]`
  The Contour Integral Method is used with this option. Parameter `radius` define the circular contour used for extraction. Parameter `numIntegPoints` defines the number of integration points used for the contour integral. The default value is 49 points. Details on this method and its GFEM implementation can be found in [**?**, **?**, **?**].
- `CFM radius_1 {rho1 #} radius_2 {rho2 #} cylThickness {zThickness} [numIntegPoints_r {nint_r #} numIntegPoints_t {nint_t #}]`
  The Cut-off Function Method is used with this option.
  - `radius_1` is the inner radius of the extraction domain. The radius is measured from the crack front.

* textttradius_2 is the outer radius of the extraction domain.
* cylThickness is the thickness of the extraction domain which is measured in the tangential direction of the crack front.
* numIntegPoints_r and numIntegPoints_t define the number of integration points in the radial and circumferential directions used to evaluate the volume integral used with this method. The default value is 25 points in both directions. The number of integration points in the tangential direction cannot be specified at this time.

Details on this method and its GFEM implementation can be found in [**?**, **?**, **?**, **?**].

The extraction of SIFs from loaded cracks requires that the user provides the components of the traction vector acting on each face of the crack. The syntax for this optional data is as follows:

```
tracFace+ {t_n #} {t_t #} {t_b #}
tracFace- {t_n #} {t_t #} {t_b #}
```

where n, t and b are directions relative to each crack face. Each face has a face-aligned (n,t,b) coordinate system defined as

- n = Direction normal to the crack face, in the outward direction of the body.
- t = Direction tangent to the crack face and in the direction away from the crack front.
- b = Direction defined by cross product $t \times n$ in the case of the positive crack face, or direction defined by cross product $n \times t$ in the case of the negative crack face.

The positive crack face is the one at $\theta = \pi$ in the local cylindrical coordinate system used for extraction at each crack front vertex. This local coordinate cylindrical and a local Cartesian system are defined based on the explicit representation of the crack surface as explained in Chapter 5. The normal to the crack surface is defined by the orientation of the facets of the polyhedron crack surface. The local y-direction is equal to the positive normal direction of the crack surface. The local x-direction is the forward direction at the crack front. The local z-direction is defined based on the x- and y-direction.
This information is only required if the CIM or CFM are used for SIF extraction. The DCM does not require this information.

The above definition of face-aligned coordinate system and positive/negative crack faces needs to be double checked.

If solving a viscoelastic problem, the following additional data must be provided

```
start_time {stime #} end_time {etime #}  nSteps {nsteps #}
```

- cleanUp { intersectData | afterPropagation }
  - afterPropagation Delete and resets objects used or created by the CrackManager. These are objects that must be rebuild after advancement of crack front. Examples are special basis created by the CrackManager, and integration elements.
  - intersectData Similar to option afterPropagation but here only containers with intersection data are deleted. This is used to save memory during a simulation. This sub-command can, in general, be called after the sub-command process but should be used with care.
- advanceFront
  { [DCM extrDist_1 {extrDist_1 #} extrDist_2 {extrDist_2 #}
  [projPntOnCrackSurface]]

```
    |
      [ DCM extrDist_min {extrDist_min #} deltaDist {deltaDist #}
            numExtrPoints {numExtrPoints #} [averLinExtrapl | leastSqrExtrapl]
            [projPntOnCrackSurface]
      ]
    |
      [ CIM radius {radius #}
          [ numIntegPoints {nint #} ]
    |
      [ CFM radius_1 {rho1 #} radius_2 {rho2 #}
            cylThickness {zThickness}
           [numIntegPoints_r {nint_r #}
            numIntegPoints_t {nint_t #}] ]
  }
    [ tracFace+ {t_n #} {t_t #} {t_b #}
      tracFace- {t_n #} {t_t #} {t_b #} ]

  [ start_time {stime #} end_time {etime #} ]
```

This sub-command computes the new crack front position using SIFs extracted at each crack front vertex and update the explicit representation of the crack surface. The magnitude of the advancement of each vertex is computed using the SIFs at the vertex and a crack front scaling law as described in sub-command `crGrowthPhysicsLaw`. All other parameters have the same meaning as in sub-command `extract`.

- `remeshCrackSurface`

  This sub-command remeshes the geometrical crack surface. This is typically used to reduce the number of facets used for the representation of the crack surface and in turn speed up the sub-command `process` and, in some circumstances, the integration of the element matrix by reducing the number of element descendants. There are several parameters that can be used to control the remeshing algorithm. They are described in the `setOption` sub-command. A detailed definition of the geometrical crack surface can be found in [**?**, **?**, **?**]. One important property of the remeshing algorithm is that it enforces the preservation of the crack front geometry while elsewhere the geometry may change after remeshing depending on the requested value of parameter `crackSurfEdgeLength`.

  See also Crack Manager option `useRemeshCrackSurface` described later in this section. The main difference between this sub-command and the use of option `useRemeshCrackSurface` is that the later performs remeshing only after a crack propagation step.

- `cleanUpStoredCrackFronts`

  This sub-command cleans the previously saved fixed points for remeshing of the crack surface. For instance, it is useful to save the front vertices of a kinking propagating crack in the first step of propagation to exactly model the initial kink. However, it might be of interest to forget these points in further remeshes of the crack surface. In this case, this sub-command can be called.

- `setFluidMatForCrackMgr {fluidMat_name} pressureBCForInitialGuess {bc_name}`

  This sub-command sets the fluid material and a constant pressure for the initial guess in coupled hydraulic fracture propagation and Krauklis waves problems. Both the material

fluidMat_name and the constant pressure for the initial guess bc_name must be defined in the grid file. The material is defined in the MATERIALS section of the grid file and the constant pressure for the initial guess is set in the BOUNDARY section of the grid file. For example:

```
BEGIN MATERIALS

# solid material
IsoHook3D solidMat_name {E = 3300 nu = 0.41}

# fluid material
NewtonFluidLub2D fluidMat_name {Viscosity = 6.0e-5}

END


BEGIN BOUNDARY

# Cosntant pressure of magnitude 20 for initial guess
Traction bc_name  { Components = -20.0  0. 0.0
                      CoordinateFrame = FaceAligned }

# Another displacement boundary condition on the solid material
Displacement displacementBC {Components = 0.0 0.0 0.0 Flags = 1 0 0}

END
```

- `setConnectingPipeMatForCrackMgr {fluidMat_name}`

  This sub-command sets the connecting pipe material to be used by the Crack Manager. The material with name fluidMat_name has to be defined in the grid file. At each propagation step, the Crack Manager will automatically create the connecting pipes set with the sub-command connectPipeWithFrac using the material set by this sub-command.

- ```
  prescribeFluidFlow { inBBox xyzMin {xMin #} {yMin #} {zMin #}
                               xyzMax {xMax #} {yMax #} {zMax #}
                     | inSemiTorus r1 {#} r2 {#} z1 {#} z2 {#} [xc {#} yc {#}]
                     | atPoint {x #} {y #} {z #} }
                   {volume {vol #} | volumeFromFile {filename}} }
  ```

  This sub-command is used to prescribe the flow rate (volume per unity of time) entering a hydraulic fracture.
  Thus, this is a Neumann boundary condition for the equation governing the pressure distribution on a hydraulic fracture. Details can be found in [**?**, **?**]. The flow rate is applied to nodes of the fluid FE mesh used to solve for the pressure distribution on the fracture. A single node can be selected using option

  ```
  atPoint {x #} {y #} {z #} Volume {vol #} }
  ```

  In this case, the flow rate of magnitude Volume is prescribed to the FEM node closest to point (x,y,z). The flow rate can be applied to a set of nodes within a bounding box defined by the lower-left and upper right corners xyzMin and xyzMax, respectively, using option

  ```
  inBBox xyzMin {xMin #} {yMin #} {zMin #}
  ```
  ```

```
xyzMax {xMax #} {yMax #} {zMax #} Volume {vol #}
```

Additionally, the flow can be applied to a set of nodes within a semi torus with inner and outer radius r1 and r2, respectively, height between z1 and z2, and center in xc and yc using option

```
inSemiTorus r1 {#} r2 {#} z1 {#} z2 {#} [xc {#} yc {#}]
```

Note that the torus has to be defined in the xy-plane.

In the last two cases, the total flow rate (Volume) is distributed among the FE nodes found. If the fluid elements are linear, the flow is distributed equally between the nodes. For quadratic fluid elements, the distribution is based on the element shape functions. The distribution for a flow Q_{edge} applied to an edge of a quadratic triangle element can be seen in Figure 4.1.

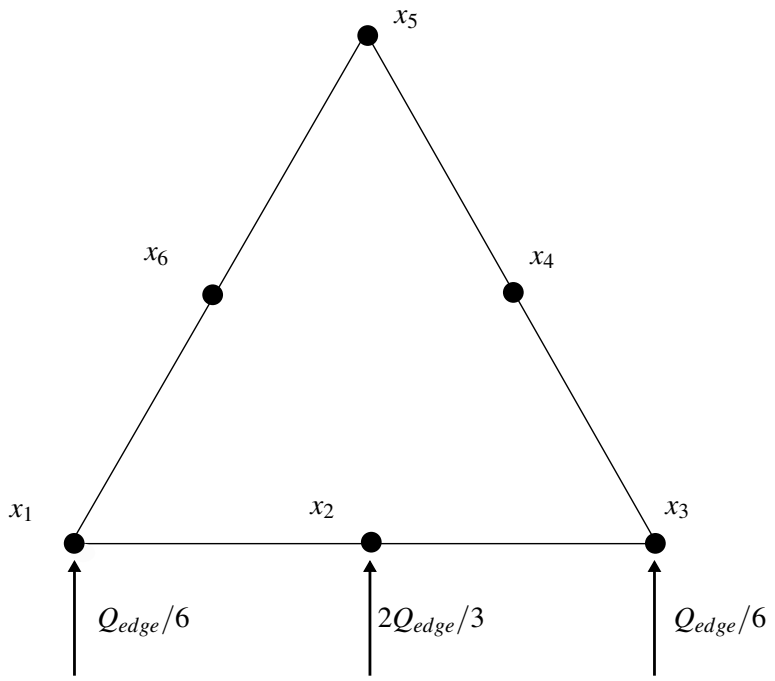


Figure 4.1: Graphical representation of the flow distribution for a quadratic standard FEM triangular element.

Also, see sub-command setCornerNodes for how to deal with nodes that are at the ends or corners of a segment where flow boundary conditions are applied.

- ```
  prescribePressureKrauklis { inBBox xyzMin {xMin #} {yMin #} {zMin #}
                                     xyzMax {xMax #} {yMax #} {zMax #}
                            | inSemiTorus r1 {#} r2 {#} z1 {#} z2 {#} [xc {#} yc {#}]
                            | atPoint {x #} {y #} {z #} }
                  { pressure {pres #}
                  | pressureFunctNumber {fn #} [pressureFrequency {freq #}]
                  | pressureFromFile {filename} }
  ```

  This sub-command is used to prescribe the fluid pressure in a hydraulic fracture when solving Krauklis wave problems. Thus, this is a Dirichlet boundary condition for the equations governing waves in fluid-filled fractures.

  The meaning of parameters inBBox, inSemiTorus, and atPoint is the same as in sub-command prescribeFluidFlow.
  ```

The pressure boundary condition to be applied on the requested nodes can either be set as a fixed value with `pressure {pres #}`, as a hard-coded pressure function number with `pressureFunctNumber {fn #} [pressureFrequency {freq #}]`, or from a file with `pressureFromFile {filename}`.

Some hard-coded pressure function numbers require supplying a frequency. This is, for instance, the case for pressure function number 5, where a pressure sine wave of amplitude 0.7 is applied on the selected nodes with frequency `freq`.

The file with name `filename` should be a text file formatted in two columns, where the first column has times and the second column has pressures. An example of the file is given next:

```
0.0 0.0
0.5 25.0
1.0 70.0
1.5 150.0
2.0 500.0
```

The code will automatically use linear interpolation/extrapolation based on the data provided in the file.

- `connectPipeWithFrac {  inSemiTorus r1 {#} r2 {#} z1 {#} z2 {#} [xc {#} yc {#}]`
  `| inBBox xyzMin {Xmin #} {Ymin #} {Zmin #}`
  `xyzMax {Xmax #} {Ymax #} {Zmax #}`
  `| atPoint {x #} {y #} {z #} }`
  `pipeNodeID {ID #}`

  This sub-command is used to inform the Crack Manager where connecting pipes should be created. The connecting pipes are connected between a pipe node with ID `pipeNodeID` and one of the options `inSemiTorus`, `inBBox`, or `atPoint`. The meaning of parameters `inBBox`, `inSemiTorus`, and `atPoint` is the same as in sub-command `prescribeFluidFlow`.
  At each propagation step, the Crack Manager will automatically create the connecting pipes based on the parameters set in this sub-command and using the material set with the sub-command `setConnectingPipeMatForCrackMgr`.

- `setCornerNodes { node1 {x #} {y #} {z #} node2 {x #} {y #} {z #} }`

  This sub-command finds the closest nodes to the coordinates provided in `node1` and `node2` and tags them as corner nodes. During the simulation of hydraulic fracturing, when flow boundary conditions are applied, the flow assigned to these corner nodes will be half than they would be if they were not tagged as corner nodes. This is done because their area of effect is smaller than the area of effect of an inner node.

Description of Sub-Command `SetOptions` or `parSet`:

The description of all parameters that can be set through `Tcl` command

```
crackMgr [ localProb {probID #} ] crackMgrID {crackMgrID #} setOptions
```

is given below.

Keyword `parSet` is just an alias to `SetOptions`. The former is preferred since it is more consistent with other `Tcl` commands.

Most of the parameters below must be set before sub-command `process` is used.

- `parSet usePlanarCutsSignedDist { [true | on | 1] | [false | off | 0] }`
  `Default = false`

  Whether to compute intersections between the crack surface and 3D elements using signed distance between element nodes and the crack surface. This approach leads to planar cuts at elements not intersected by the crack front while preserving the exact geometry of the crack front. Details on the cutting algorithm can be found in [**?**].

- `parSet useBranchFn { [true | on | 1] | [false | off | 0] }`
  `Default = true`

  Whether to create and assign branch functions to nodes of elements that are intersected by the crack front or has it at their boundary. Thus, if true, topological enrichment with branch functions is used when sub-command `process` is issued. The type of branch function adopted is controlled by parameter `branchFunctionType`.

- `parSet useSingularBranch { [true | on | 1] | [false | off | 0] }`
  `Default = true`

  Whether branch functions, if used, are singular or not. This flag must be set before branch functions are created, i.e., before processCrack is called. Singularity can not be turned on/off after that.

- `parSet useMLSForExtraction { [true | on | 1] | [false | off | 0] }`
  `Default = false`

  Whether a Moving Least Squares (MLS) approximation [**?**] for extracted Stress Intensity Factors is used. The MLS approximation provides smoother SIFs curves along the crack front than raw values. If set to true, the MLS approximation may also used to extrapolate SIF values to crack front ends, depending on the value of parameter `useMLSForSkippedFrontEndVert`. See also parameter `useMLSForCrackFrontVertices`.

- `parSet useMLSForSkippedFrontEndVert { [true | on | 1] | [false | off | 0] }`
  `Default = true`

  It is often not possible to extract SIFs at crack front ends due to, e.g., missing integration points caused by extraction domain being partially outside of the solution domain. If this parameter is set to true a Moving Least Squares approximation of extracted SIFs is used to extrapolate SIF values to front-end vertices that were skipped during extraction. If this parameter is set to false, the SIF values at front-end vertices that were skipped during extraction, are taken as the same value at the last front-end vertex that was not skipped. This is useful for problems with SIFs that are known to be constant along the crack front. See also parameters `useMLSForExtraction` and `numFrontEndVertToSkip`.

- `parSet ignoreKII { [true | on | 1] | [false | off | 0] }`
  `Default = false`

  If this parameter is set to true, SIF K_{II} is computed, printed to the .sif file, and then set to zero. Therefore, the computed value will not be used for crack propagation. This parameter should be used with care and only in problems that are known a-priori to have $K_{II} = 0$. The user should check if the computed values are indeed zero or nearly zero.

- `parSet ignoreKIII { [true | on | 1] | [false | off | 0] }`
  `Default = false`

  If this parameter is set to true, SIF K_{III} is computed, printed to the .sif file, and then set to zero. Therefore, the computed value will not be used for crack propagation. This parameter

should be used with care and only in problems that are known a-priori to have $K_{III} = 0$. The user should check if the computed values are indeed zero or nearly zero.

- `parSet useMLSForCrackFrontVertices {[true | on | 1] | [false | off | 0]}`
  `Default = false`

  Whether Moving Least Squares (MLS) approximations [**?**] for crack front vertices after each propagation step is to be performed. The MLS approximations are computed for each global coordinate of crack front vertices. The updated crack surface, i.e., after front advancement and after application of MLS smoothing to new crack front vertex positions, are used as input for the remeshing function, if parameter `useRemeshCrackSurface` if set to true. This MLS approximation is useful to remove numerical noise of crack front positions after a crack propagation increment. See also parameters `useMLSForCrackFrontVerticesWhileRemeshing` and `useMLSForExtraction`.

- `parSet useMLSForCrackFrontVerticesWhileRemeshing`
  `{ [true | on | 1] | [false | off | 0] }`
  `Default = false`

  If this parameter is set to true, Moving Least Squares (MLS) approximations [**?**] for each global coordinate of crack front vertices are used as input for the remeshing function *but the input crack surface for the remeshing function will have non-MLS crack front coordinates*. This MLS approximation is useful to remove numerical noise of crack front positions after a crack propagation increment. See also parameter `useMLSForCrackFrontVertices`.

  R The implementation supporting the functionality controlled by this parameter is still at alpha stage.
  Parameters `useMLSForCrackFrontVerticesWhileRemeshing` and `useMLSForCrackFrontVertices` cannot both be set to true.
  Parameters `useMLSForCrackFrontVerticesWhileRemeshing` should be set to true only if parameter `useRemeshCrackSurface` is set to true. This is not the case for parameter `useMLSForCrackFrontVertices`.

- `parSet useUniformCrackFrontVertices`
  `{[true | on | 1] | [false | off | 0]}`
  `Default = true`

  If this parameter is set to true, the crack front vertices after each propagation step will be uniformly distributed along the crack front. This is useful to preserve the accuracy of the geometrical approximation of the crack front when the propagation speed along the crack front is non-uniform. Portions of the crack front that propagates with higher speed will have longer edges than those that propagates slowly, if this parameter is not set to true. Longer edges along curved fronts lead to loss of geometrical accuracy and should be avoided.

  R The uniform redistribution of crack front edges can only be performed if parameter `useMLSForCrackFrontVertices` is set to true.

  R *This parameter must be used with caution*: Redistribution of front vertices requires tangential displacement of the vertices along the crack front. This may lead to overlapped (invalid) crack surface facets if the magnitude of tangential displacement is large. If this happens, use
  `useMLSForCrackFrontVerticesWhileRemeshing = true`
  instead of both
  `useMLSForCrackFrontVertices = true` and
  `useUniformCrackFrontVertices = true`.

- `parSet useSimplifyCrackSurface {[true | on | 1] | [false | off | 0]}`
  `Default = false`

  The functionality controlled by this parameter is no longer available,

- `parSet useCompleteSimplifyCrackSurface`
  `{[true | on | 1] | [false | off | 0]}`
  `Default = false`

  The functionality controlled by this parameter is no longer available,

- `parSet useRemeshCrackSurface {[true | on | 1] | [false | off | 0]}`
  `Default = false`

  If set to `true` the geometrical crack surface will be remeshed *after every crack propagation step*. This is typically used to reduce the number of facets used for the representation of the crack surface and in turn speed up the sub-command `process` and, in some circumstances, the integration of the element matrix by reducing the number of element descendants. There are several parameters that can be used to control the remeshing algorithm. They are described in the `setOption` sub-command. A detailed definition of the geometrical crack surface can be found in [**?**, **?**, **?**]. One important property of the remeshing algorithm is that it enforces the preservation of the crack front geometry while elsewhere the geometry may change after remeshing depending on the requested value of parameter `crackSurfEdgeLength`.

- `parSet storeCrackFront { [true | on | 1] | [false | off | 0] }`
  `Default = false`

  If set to true, the crack front points before a propagation step are preserved after the front propagation and remeshing of the crack surface. By default, the remeshing function preserves only the crack front points *after* the propagation, i.e., the front points of the triangulation passed to the remeshing function. This option is important when simulating the propagation of non-planar surfaces with sharp features such as kinks. These features are located at a previous crack front. Thus, by preserving an old crack front, the geometrical features are preserved while remeshing the crack surface. The front points are cumulatively preserved as long as this parameter is set to *true*. Thus, this parameter should only be set to *true* when the crack surface is expected to make sharp turns, like in the first propagation step of a mixed mode problem.

- `parSet viscoExtractionCP { [true | on | 1] | [false | off | 0] }`
  `Default = false`

  This option is used to activate the extraction of time-dependent Energy Release Rate (ERR) in problems with linear viscoelastic materials and solved using the elastic-viscoelastic Correspondence Principle (C.P.). Further details can be found in [**?**]. The reference elastic problem in the C.P. is solved using an `IsoVisco3D` material, even though it is just a linear elastic problem.

  This approach to extract ERR is deprecated. Use instead `parSet useViscoExtractionCP` as described later.

- `parSet viscoFourierTransf { [true | on | 1] | [false | off | 0] }`
  `Default = true`

This option is relevant when solving viscoelastic problems using the elastic-viscoelastic correspondence principle. If true, the transformation from Laplace to physical domain is performed using Fourier transformation. If false, Zakian's method [**?**] is used. Further details can be found in [**?**].

Fourier method is recommended. Zakian's method is in general not accurate.

- `parSet viscoForceControlled { [true | on | 1] | [false | off | 0] }`
  `Default = true`

  This option is relevant when solving viscoelastic problems using the elastic-viscoelastic correspondence principle. It is used to switch between displacement and force controlled problems. Further details can be found in [**?**].

- `parSet MLS_overlappingFactor {double_precision_option_value #}`
  `Default = 2.0`

  This parameter controls the overlapping of weighting functions used to compute Moving Least-Squares (MLS) approximations of, e.g., SIFs. Further details can be found in [**?**].

- `parSet delta_a_max {double_precision_option_value #}`
  `Default = none (must be set by user)`

  This parameter controls the magnitude of the maximum advancement of any crack front vertex. The magnitude of crack front vertex advancement is based on this parameter and a `crGrowthPhysicsLaw`.

- `parSet frontEdgeLengthLimit {double_precision_option_value #}`
  `Default = Maximum crack front length read from .crf file`
  `used to create this Crack Manager.`

  This parameter controls the maximum allowed length of crack front edges. If after a propagation step this limit is violated, the crack front is automatically refined, thus preserving the geometric accuracy of the crack front as it propagates.

This parameter is only used while remeshing the crack surface. Thus, for it to be enforced, parameter `useRemeshCrackSurface` must be set to `true` or sub-command `remeshCrackSurface` must be used.

- `parSet crackSurfEdgeLength {double_precision_option_value #}`
  `Default = 1.1 times the average of crack surface edge lenght`
  `read from .crf file used to create this Crack Manager.`

  This parameter controls the maximum allowed length of crack surface edges while remeshing the crack surface. It is relevance only if parameter `useRemeshCrackSurface` is set to true or sub-command `remeshCrackSurface` is called.

- `parSet scaleForCrackFrontVerticalDisplacement`
  `{double_precision_option_value #}`
  `Default = 0.5`

  This parameter is used with the beam on elastic foundation model used to compute the contribution of the crack front twisting angle to the vertical displacement (kinking angle) along the crack front. See [**?**] and Section 5.2.1.1 of [**?**] for details.

This parameter is currently not available to the user.

- `parSet accumulatedCrackFrontAdvanceLimit {double_precision_option_value #}`
  `Default = 5 * delta_a_max`

  Advancement of crack front is performed using either stretching (Propagate And Stretch (PAS)) of existing front vertices or by adding/extruding a new layer of facets along the crack front (Propagate And Extrude (PAE)) [**?**].
  This parameter sets the magnitude of the maximum crack front vertex advancement (from repeated application of PAS) before forcing the creation of a new layer of facets (i.e., force use of PAE). This is used to avoid repeated updates of the crack surface using PAS, which leads to very stretched facets along the crack front. Repeated PAS updates can happen if parameter `frontStretchTol` is set to a high value. See [**?**] for details.

- `parSet frontStretchTol {double_precision_option_value #}`
  `Default = 0.0 This implies that by default, a new layer of facets is created along the entire crack front (PAE) at every crack propagation step.`

  Tolerance used to select the type of crack front advance [**?**]:
  - If the displacement of a front vertex *relative to the maximum displacement* is smaller than `frontStretchTol`, the vertex will be stretched (PAS). Otherwise a new layer of facets will be added (PAE). The value for this option is given *as a percentage of the maximum vertex displacement*, `delta_a_max`. If set to zero, PAE is always used (default). If set to 100.0, PAS is always used. For any value in between, if the displacement of a front vertex relative to the maximum displacement is smaller than `frontStretchTol` then it will be stretched, otherwise a new layer of faces will be created (PAE). For example, if `frontStretchTol = 60.0`, then all vertices with displacement less than $0.6\times$`delta_a_max` will be stretched. Those with larger displacement with have a new layer of facets added ahead of them.
  - If the displacement at a front vertex is larger than this relative tolerance, a new layer of facets and vertices are created. Otherwise the front vertex is stretched.

  It is noted that this parameter is given in terms of percentages and hence the range of acceptable values is $[0.0, 100.0]$.

  
  Expert tip: A value of `frontStretchTol = 25.0` usually works well.

- `parSet viscoPeakTime {double_precision_option_value #}`
  `Default = 1.0`

  This option is relevant when solving viscoelastic problems using the elastic-viscoelastic correspondence principle. `viscoPeakTime` is the time at which the prescribed time-dependent force or displacement boundary condition reaches the maximum value. Further details and examples can be found in [**?**].

- `parSet useViscoExtractionCP viscoMatName {name_viscoelas_mat}`
  `Default = none`

  This option is used to activate the extraction of time-dependent Energy Release Rate in problems with linear viscoelastic materials and solved using the elastic-viscoelastic Correspondence Principle (C.P.). Further details can be found in [**?**]. The reference elastic problem in the C.P. is solved using an `IsoHook3D` material defined in the `.grf` file of the problem being solved. The viscoelastic material properties used for the extraction of ERR(t) is provided by an `IsoVisco3D` material with name `name_viscoelas_mat` and defined in the `.grf` file of the problem being solved.

R This approach to extract ERR(t) should be used instead of `parSet viscoExtractionCP` which is deprecated.

- `parSet viscoLoadType {constant | ramp | creep | hat | freq | haversine}`
  `Default = constant`

  This option is relevant when solving viscoelastic problems using the elastic-viscoelastic correspondence principle. It is used to set the type of function that describes the prescribed force or displacement boundary conditions. Further details can be found in [**?**].

- `parSet viscoIntegrType {LaplaceTransf | GaussIntegr | incremental}`
  `Default = LaplaceTransf`

  This option is relevant when solving viscoelastic problems using the elastic-viscoelastic correspondence principle. Further details can be found in [**?**].

  R `LaplaceTransf` should be used. The other options are either not effiencient or still experimental at this time.

- `parSet MLS_degree {integer_option_value #}`
  `Default = 2`

  Polynomial degree of (local) Moving Least-Squares (MLS) approximations of, e.g., SIFs. Further details can be found in [**?**].

- `parSet MLS_numSamplePointsFrontEdges {integer_option_value #}`
  `Default = 2 (leads to sampling at edge ends + 2 at interior of front edges)`

  Number of sampling points at the interior of each crack front edge when dumping SIF data to file .cm_sif If zero, sampling is done only at front edge vertices. These are samplings of the MLS approximation of SIFs.

- `parSet compSIFsAfterEveryNVerts {integer_option_value #}`
  `Default = 0 (Extract SIFs at every crack front vertex)`

  Number of crack front vertices to skip/jump after extracting at a given vertex. For example, if `compSIFsAfterEveryNVerts = 1`, SIFs are extracted at every other front vertex.

- `parSet numFrontEndVertToSkip {integer_option_value #}`
  `Default = 1 (skip front-end vertices only -- first and last vertex at each front)`

  Number of crack front end vertices to skip *at each front end.* Typically skipping the front-end vertices is sufficient to address issue of missing integration points when extracting SIFs at front ends. However, there are situations where skipping more vertices may be required. The SIFs values at skipped vertices are computed from the extrapolation of values extracted close to front ends. See also parameters `useMLSForCrackFrontVertices` and `useMLSForExtraction`.
  This parameter is not relevant for interior cracks since they have no ends.

- `parSet applyBCtoCrackFaces {bc_name}`
  `Default = No default value`

  If set, this parameter turns on face boundary conditions on crack surface faces. Argument `bc_name` is the name of a boundary condition (a Material BC) defined in the .grf file of the problem.

R If bc_name = "none", face boundary condition elements defining the computational crack surface will be created but not actually used by CompMesh at time of assembly. This option is used to visualize the computational crack surface in problems that have no BC on crack faces. See also sub-command postComputationalSurface.

- parSet useCohesive {bc_name}
  Default = No default value

  Just an alias to parameter applyBCtoCrackFaces as far as the crackManager is concerned.

- parSet branchFunctionType {straightFrontOD
  | straightFrontBB | straightFrontOY
  | curvedFrontBB | curvedFrontOD
  | curvedFrontOD6 | curvedFrontOD9 }

  Default = straightFrontOD

  Type of branch function to be used when executing sub-command process. The following branch function types are supported
  - straightFrontOD: Oden-Duarte branch function enrichment described in, e.g., [**?**, **?**, **?**, **?**]. These enrichments are defined on a rectangular Cartesian coordinate system. Thus, they are not recommended for problems with curved crack fronts or curved surfaces. The definition of the coordinate systems at nodes close to the crack front is presented in [**?**, **?**].
  - straightFrontBB: The branch function enrichments presented in [**?**].These enrichments are defined on a rectangular Cartesian coordinate system. Thus, they are not recommended for problems with curved crack fronts or curved surfaces. See also [**?**].
  - straightFrontOY: These enrichments are described in [**?**].
  - curvedFrontBB: Same basis as straightFrontBB but defined in a curvilinear coordinate system based on signed distances as described in [**?**, **?**].
  - curvedFrontOD: Same basis as straightFrontOD but defined in a curvilinear coordinate system based on signed distances as described in [**?**, **?**]. This is the same as option curvedFrontOD6.
  - curvedFrontOD6: This is an improvement of OD basis implementation described in [**?**] which avoids spurious coupling between Mode I/II basis functions and Mode III basis functions. It can represent exactly a 2D Mode I and Mode II solution, but not a Mode III.
  - curvedFrontOD9: Same basis as straightFrontOD6 but able to represent exactly all three crack deformation modes (at the cost of three additional dofs per node enriched with this basis).

    R The OD basis presented in [**?**] uses two functions to approximate a Mode III deformation. If option curvedFrontOD, curvedFrontOD6, or curvedFrontOD9 is selected, the second function is given by

    $$\sqrt{r}\cos\frac{\theta}{2}$$

    which is different from the one proposed in [**?**]. All other basis functions are exactly the same as in [**?**]. Further details on this modified OD basis is presented in [**?**].

- parSet IDNeighborMgr4Coalesce {crackMgrID #}
  Default = none

Activates algorithm for coalescence of two crack surfaces after each propagation step and sets the ID of the crack manager whose crack surface that can potentially coalesce with the surface of this manager.

- `parSet surfCoalesceTol {tol_distance #}`
  `Default = none`

  Set tolerance used to check whether two crack surfaces have coalesced.

- `parSet snapNodesToSurfAndFront snappingTolSurf {tol #}`
  `[snappingTolFront {tol #}]`
  `Default = 0.05 for both parameters (if snapping is activated)`

  Turns on snapping of Nodes to the crack surface and to crack front. Set scaled tolerance for snapping to the surface (`snappingTolSurf`) and, optionally the tolerance to snap nodes to the crack front (`snappingTolFront`). If the latter is not provided, the default value of 5% is adopted. Optionally, the user can simply turn on or off snapping of nodes using

  ```
  parSet snapNodesToSurfAndFront { [true | on | 1] | [false | off | 0] }
  ```

  In this case, default values for both `snappingTolSurt` and `snappingTolFront` are adopted.

  R Node snapping is not supported when solving a problem with the GFEM^{gl} since it will break the nesting of local and global meshes.

Description of Sub-Command `parGet`

The description of all parameters that can be accessed through `Tcl` command

```
crackMgr [ localProb {probID #} ] crackMgrID {crackMgrID #} parGet
```

is given below. The retrieved values are used in the `Tcl` script being executed.

- `parGet minElemEdgeLenAtCrackFront`

  Retrieves from the C++ data base the minimum edge length among all elements cut or touched by the crack front(s).

- `parGet isCrackArrested`

  Retrieves Boolean flag indicating whether in a stable crack propagation simulation the crack has arrested.

Description of Sub-Command `crGrowthPhysicsLaw`

The description of all parameters that can be set through `Tcl` command

```
crackMgr [ localProb {probID #} ] crackMgrID {crackMgrID #} crGrowthPhysicsLaw
```

is given below.

R The parameters below must be set before sub-command `advanceFront` is used.

- `crGrowthPhysicsLaw create`

  This sub-command creates a `crGrowthPhysicsLaw` for the `crackMgr`. The user must also set the `crGrowthType`, the `crFrontIncrementType`, and the `crFrontScalingLaw` before the `crGrowthPhysicsLaw` can be used.

- `crGrowthPhysicsLaw crGrowthType`
  `{ FatigueCrackGrowth`
  `| StableCrackGrowth {Gc_material {user_defined_Gc #}} }`

  This sub-command of `crGrowthPhysicsLaw` defines the type of crack propagation to be performed:
  - `FatigueCrackGrowth`

    In a `FatigueCrackGrowth`, all crack front vertices propagates at a crack propagation step. The magnitude of propagation at each vertex is controlled by a fatigue-like "scaling" law, like the `crFrontScalingLaw` defined by a `ParisLaw`.
  - `StableCrackGrowth {Gc_material {user_defined_Gc # }`

    In a `StableCrackGrowth` simulation, Irwin's criterion is adopted as the propagation criterion: A crack front vertex only propagates if the energy release rate at the vertex is greater or equal to the material critical value $G_c =$ `Gc_material`.
- `crGrowthPhysicsLaw crFrontIncrementType`

```
{ fixedDeltaA {us_def_incr #}
|
  adaptiveDeltaA
         [ setOptions
             {
               Alpha {#Double_value} |
               Beta  {#Double_value} |
               restart
             }
         ]
}
```

  This sub-command of `crGrowthPhysicsLaw` defines how the magnitude of the maximum crack front vertex advancement is to be computed:
  - `fixedDeltaA {us_def_incr #}`

    If a `fixedDeltaA` is selected, the crack front vertex with the highest energy release rate will propagate the user-defined magnitude `us_def_incr` at a propagation step. The magnitude of advancement at other front vertices is defined based on `us_def_incr` and the selected `crFrontScalingLaw`. This parameter is often changed during a fracture propagation simulation. Small values, relative to the initial crack size, are typically used at the early stages of a simulation where strong changes in crack propagation direction can happen. Later in the simulation, larger values can be used to speed up the simulation (reduce the number of propagation steps required to reach a given crack size).
  - `adaptiveDeltaA`

```
       [ setOptions {
                    Alpha {#Double_value} |
                    Beta  {#Double_value} |
                    restart
                    }
      ]
```

    If an `adaptiveDeltaA` is selected, the magnitude of the maximum advancement at a crack front vertex is automatically computed by the adaptive algorithm described in Section 5.3 of [**?**].

* Alpha denotes the percentage change in crack surface area for the first propagation step. The default value is $\alpha = 0.05$, denoting a 5% change in surface area. A smaller value of α should be used if the initial crack is large or if restarting the simulation after a number of propagation steps.
* Beta denotes the change in strain energy associated with each crack propagation step. The default value of β is 0.05 representing 5% change in strain energy per propagation step. Higher values can be selected after the crack has propagated several steps, while lower values should be used just before the crack arrests when simulating a stable crack propagation (i.e., crGrowthType is stableCrackGrowth).

The implementation supporting this functionality is still at alpha stage. The parameters for the adaptive algorithm are described in Section 5.3 of [?].

- crGrowthPhysicsLaw frontKinkingAngleLimitDEG
  {double_precision_option_value #}

  Default = 2

  Sets the minimum value for the crack kinking angle at any crack front vertex. The computed kinking angle at a front vertex is set to zero if it is smaller than frontKinkingAngleLimitDEG. Values smaller than frontKinkingAngleLimitDEG are taken as numerical noise. This is usually the case when K_{II} is much smaller than K_I, leading to nearly zero values of the kinking angle. The input value must be in degrees. The kinking angle in a crack propagation simulation is described, for example, in Appendix A of [?], Section 7.3.3 of [?], and in [?].

  Expert tip: If the crack surface shows some oscillations as the crack transions from a mixed to a mode I propogation, set frontKinkingAngleLimitDEG to 0 degrees.

- crGrowthPhysicsLaw frontTwistingAngleLimitDEG
  {double_precision_option_value #}

  Default = 2

  Sets the minimum value for the crack twisting angle at any crack front vertex. The computed twisting angle at a front vertex is set to zero if it is smaller than frontTwistingAngleLimitDEG. Values smaller than frontTwistingAngleLimitDEG are taken as numerical noise. This is usually the case when K_{III} is much smaller than K_I, leading to nearly zero values of the twisting angle. The input value must be in degrees. The twisting angle in a crack propagation simulation is described, for example, in Appendix A of [?], Section 7.3.3 of [?], and in [?].

  Expert tip: If the crack surface shows some oscillations as the crack transions from a mixed to a mode I propogation, set frontTwistingAngleLimitDEG to 0 degrees.

- crGrowthPhysicsLaw print

  Prints the crGrowthPhysicsLaw object to std output.

Description of Sub-Command crFrontScalingLaw **of** crGrowthPhysicsLaw

The description of all parameters that can be set through Tcl command

```
crackMgr [ localProb {probID #} ] crackMgrID {crackMgrID #}
                                  crGrowthPhysicsLaw crFrontScalingLaw
```

is given below.

The parameters described below must be set before sub-command advanceFront is used.

- `crGrowthPhysicsLaw crFrontScalingLaw`
  `ParisLaw`
  `{ C {us_def_C #}`
  `m {us_def_m #}`
  `R {us_def_R #}`
  `}`

  Creates a crack front scaling law based on Paris law

  $$\frac{da}{dN} = C(\Delta K)^m$$

  where da/dN is the crack growth rate with respect to the number of load cycles (N). C and m are model parameters based on experimental data and

  $$\Delta K = K_{\max} - K_{\min} = (1-R)K_{\max}$$

  where $K_{\max}$ ($K_{\min}$) is the SIF corresponding to the maximum (minimum) load, and $R = K_{\min}/K_{\max}$. See Section 3.2.3 of [**?**] for details.
  Using the above, the magnitude of advancement at crack front vertex j, Δa_j, is given by

  $$\Delta a_j = \Delta a_{\max} \left(\frac{\Delta K_j^{\text{eq}}}{\Delta K_{\max}^{\text{eq}}} \right)^m$$

  where
  - $\Delta a_{\max}$ is the magnitude of advancement at the crack front vertex with the highest energy release rate. This is the same as parameter `fixedDeltaA` of `crFrontIncrementType`;
  - ΔK_j^{eq} is the *equivalent Mode I stress intensity factor*, at crack front vertex j computed using an expression proposed by Schöllmann [**?**, **?**] (see also Appendix A of [**?**]);
  - $\Delta K_{\max}^{\text{eq}}$ is ΔK_j^{eq} computed at the crack front vertex with the highest energy release rate.

- `crGrowthPhysicsLaw crFrontScalingLaw`
  `GuptaDuarteLaw`
  `{ KIC_material {KIC_material #}`
  `alpha {alpha #}`
  `[ m_alpha {m_alpha #} ]`
  `beta {beta #}`
  `[ m_beta {m_beta #}  ]`
  `}`

  This is a scaling law based on a regularization of Irwin's criterion as described in Section 4.1 of [**?**]. This law is typically used in hydraulic fracture simulations. Default value for `m_alpha` and `m_beta` is 1.0. Figure 4.2 illustrates this scaling law. In the figure, $\Delta a_{\max}$ is the same as parameter `fixedDeltaA` of `crFrontIncrementType`; K_I^{eq} is the *equivalent Mode I stress intensity factor*, computed using an expression proposed by Schöllmann [**?**, **?**]; Δa_j is the magnitude of advancement at crack front vertex j; `m_alpha = 1` and `m_beta = 1`.

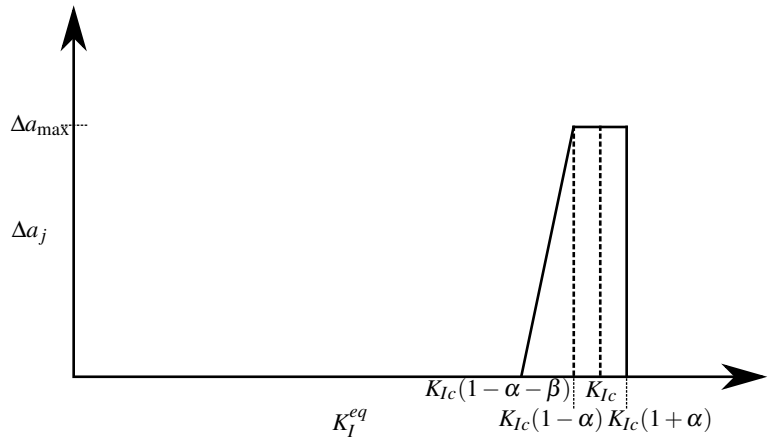


Figure 4.2: Graphical representation of the GD model with parameters $\alpha = 0.05$, $\beta = 0.10$, `m_alpha = 1` and `m_beta = 1` [**?**].

- `crGrowthPhysicsLaw crFrontScalingLaw ConstFrontDispl`

  If this scaling law is selected, all crack front vertices will propagate `ConstFrontDispl`, regardless of their SIFs. The magnitude of the advancement is given by parameter `fixedDeltaA` of `crFrontIncrementType`. This law is typically used then simulating the propagation of through-the-thickness cracks and the variation of SIF through the thickness is negligible.
- `crGrowthPhysicsLaw crFrontScalingLaw`
  `setConstFrontDispl [{true | on | 1} | {false | off | 0}]`

  Here, a scaling law is *not* actually set. Instead, this option tells an existing scaling law whether it should apply a constant displacement along the crack front or not. Thus, `setConstFrontDispl` can only be used if a scaling law has already been set. Not all scaling laws support this action.
- `crGrowthPhysicsLaw crFrontScalingLaw`

  TBD remaining scaling laws

4.5.4 masterCrackMgr

The *Tcl* command `masterCrackMgr` is used when solving three-dimensional multiple fracture mechanics problems. Command `masterCrackMgr` supports all commands implemented for a single crackMgr except create.

```
masterCrackMgr
  [ create

  | advanceFront
      {  [ DCM  extrDist_1 {extrDist_1 #} extrDist_2 {extrDist_2 #} ]
       |
         [ DCM extrDist_min {extrDist_min #} deltaDist {deltaDist #}
              numExtrPoints {numExtrPoints #} [averLinExtrapl | leastSqrExtrapl] ]
           [ projPntOnCrackSurface ]
       | [ CIM radius {radius #}
              [ numIntegPoints {nint #} ]
       |
         [ CFM  radius_1 {rho1 #} radius_2 {rho2 #}
              [ cylThickness {zThickness} ]
              [ numIntegPoints_r {nint_r #}
                numIntegPoints_t {nint_t #} ]
     }
       [ tracFace+ {t_n #} {t_t #} {t_b #}
         tracFace- {t_n #} {t_t #} {t_b #} ]

     [ start_time {stime #} end_time {etime #}]

  | forAllCracks
    {any sub-command of crackMgr (except create and advanceFront)
     after crackMgrID {crackMgrID #} } ]
```

4.5.5 crackMgr2D

The *Tcl* command `crackMgr` is used when solving two-dimensional fracture mechanics problems.

TBD: Describe main assumptions and limitations: Static cracks only; Cracks defined by a

signed distance function;

```
crackMgr2D [ localProb {probID #} ]  crackMgrID {crackMgrID #}
      { create
         {  signedDistFunction {function #}

          | straightCrackEndPoints tail {x #} {y #}
                                   tip  {x #} {y #} }

      | process

      | branchFnGeometricEnrich { inCylinder radius {R #}
                                | inBBox xyzMin {xMin #} {yMin #} {zMin #}
                                         xyzMax {xMax #} {yMax #} {zMax #} }
      | {setOptions | parSet}
         {  branchFunctionType {straightFrontOD | straightFrontBB
                              | straightFrontHOOD }
          | useBranchFn        { [true | on | 1] | [false | off | 0] }
          | useSingularBranch  { [true | on | 1] | [false | off | 0] }

          | snapNodesToSurfAndFront snappingTolSurf {tol #}
                                    [snappingTolFront {tol #}]
          | snapNodesToSurfAndFront { [true | on | 1] | [false | off | 0] }
        }
      } # end of crackMgr2D sub-commands
```

Description of `CrackMgr2D` Sub-Commands and Options:

- `[ localProb {probID #} ]`

  A `CrackMgr2D` can be created for either a local or a global problem in a GFEM^{gl} analysis. Optional argument `localProb` is used to create a `CrackMgr2D` for a local problem with ID `probID`. See also section 4.3.2.

  R `CrackMgr2D` has not been tested in a GFEM^{gl} analysis.

- `crackMgrID {crackMgrID #}`

  This argument specifies the unique integer ID of each `crackMgr2D` object.

- `create`
  ```
  {  signedDistFunction {function #}
   | straightCrackEndPoints tail {x #} {y #} tip  {x #} {y #} }
  ```

  Creates a `CrackMgr` object with the specified ID using one of the following options to define the crack surface:
  - `signedDistFunction`: TBD
  - `straightCrackEndPoints`: TBD

- `process`

  Process the crack: Create and assign enrichments to nodes whose clouds/patches interact with the crack surface; creates integration descendants used for the numerical integration of weak form.

- TBD: Describe other sub-commands and options

4.6 Tcl Commands for a Coupled FTSI Analysis

This section described the *Tcl* commands used in *ISET* to couple with an external fluid-solver to simulate a coupled Fluid-Thermal-Structural interaction example. Details on the implementation methodology can be found in Section 6.1

4.6.1 coupledFTSIManager

The *Tcl* command `coupledFTSIManager` is used to create and call Abaqus functionalities to simulate a coupled FTSI example. The syntax of the command is as follows:

```
coupledFTSIManager  [ create
                    | UExternalDB kstep {#} \n
                                  kinc {#}  \n
                                  lop {#}   \n
                                  curr_time {#}  \n
                    | URDFIL [ Temperature | Displacement |
                               Coordinates ]
                    ]
```

Description of `coupledFTSIManager` Sub-Commands:

- `create`: Creates a `coupledFTSIManager` object for the simulation. This should be created *soon* after reading of the grf file. It can be created before or after the creation of analysis object and is not dependent on the type of analysis object.
- `UExternalDB`: It calls the UExternalDB subroutine of the Abaqus which is used to call/manage the fluid-solver. UExternalDB subroutine can be used to manage user-defined external databases and calculate model-independent history information. This *Tcl* command has to be called once
  - at the beginning of the analysis,
  - at the beginning of each time step increment,
  - at the end of each time step increment, and
  - at the end of the analysis.

  There are 4 additional sub-options for this *Tcl* command as follows:
  - `kstep`: Current step number.
    `kstep = 0` at the beginning of the analysis, and
    `kstep = 1` during the analysis.
  - `kinc`: Current time increment.
  - `LOP`: This is an internal variable in Abaqus.
    `LOP = 0` indicates that the subroutine is being called at the start of the analysis,
    `LOP = 1` indicates that the subroutine is being called at the start of the current analysis increment (time increment),
    `LOP = 2` indicates that the subroutine is being called at the end of the current analysis increment (time increment),
    `LOP = 3` indicates that the subroutine is being called at the end of the analysis, and
    `LOP = 4` indicates that the subroutine is being called at the beginning of a restart analysis. This capability is not currently available for ISET.
  - `LRESTART`: This is an internal variable in Abaqus.
    `LRESTART = 0` indicates that an analysis restart file is not being written for this increment,
    `LRESTART = 1` indicates that an analysis restart file is being written for this increment.

The user is advised to look at one of the QA examples of the coupled fluid-thermal-structural interaction simulation to understand the usage of these *Tcl* commands in detail.

4.7 Tcl Commands for a Multi-Physics Analysis

This section described the *Tcl* commands used in *ISET* to solve multi-physics problems.

4.7.1 multiPhysDir

This command allows the user to set up and run a multiphysics simulation.

```
multiPhysDir  [localProb {probID #}]
              [ create {thermoelas1way | thermoplas1way | phasefield2way}
              | {cloneProblem | cloneProblemWithCrackMg}
              | activate {heat | thermoelas | thermoplas |
                          phasefield | phasefieldelas}
              | staggeredSolOption [ setMaxStaggeredIter { # iterLim }
                                  | setStaggeredConvTol { # conTol } ]
              | solveStaggeredStep { # stepNum } [ adapStep ]
              ]
```

- `localProb`: See section 4.3.2.
- `create` create a MultiPhysicsDirector object for a given physics type
- `cloneProblem` clone a defined problem (Analysis, GeoMesh, and CompMesh) to create a problem for the other physics.
- `activate` activate the problem (Analysis and CompMesh) associated with the specified physics.
- `staggeredSolOption` set options for the staggered solution scheme implemented in `ISET`. `setMaxStaggeredIter` sets the maximum number of iterations allowed for a given step in the staggered scheme. `solvStaggeredStep` attempts to solve the specified load step of the multiphysics problem with the the staggered scheme.
- `solveStaggeredStep`

R Please document this command and its options/sub-commands.

[heading=subbibliography]

5. The Crack Surface File

The data defining a geometrical crack surface used in a 3-D fracture analysis is defined in `crf` formatted input file. This format is very similar to the one adopted in the `phlex` formatted input file (cf. Chapter 2). The `crf` format is keyword based, straightforward to generate and very readable. The suffix is by default `*.crf` but this is not required. This chapter presents an overview of the basic data packets recognized by the `crf` input format (*.crf) file reader. These packets define a crack surface by defining the vertices and facets of the triangulation defining a crack surface and the vertices and segments defining crack fronts. Each packet of data is delimited by a set of keywords which define the beginning, the end, and the type of input to follow. Complete details describing this format are described in the next sections. Detailed definition of the concept of *geometrical crack surfaces* adopted in *ISET* can be found in [**?**, **?**, **?**]. The concept of *computational crack surfaces* adopted in *ISET* can be found in [**?**, **?**].

Application specific data is also saved on the geometrical crack surface when solving non-linear fracture propagation problems. This is described in [**?**, **?**, **?**].

A geometrical crack surface is also used by *ISET* to re-start a fracture propagation simulation from a later step. The data saved on the `.crf` when re-starting a simulation is also described in this chapter. [heading=subbibliography]

6. Additional ISET Modules

This chapter provides description about additional ISET modules, which are not part of main ISET repository.

6.1 *ISET* Coupled Fluid-Thermal-Structural Interaction Simulation

This section details the methodology behind coupling a fluid solver with ISET for multi-physics simulations. It also provides a brief overview of the current fluid solver, developed by the research group of Prof. Jack. McNamara at the Ohio State University.

A tutorial on the usage of the ISET-FTSI coupled solvers is presented in the *ISET Tutorial* available at hg repository:
`ssh://hg@faaws2.cee.illinois.edu/repositories/ISET_Tutorial`

6.1.1 Coupling Methodology

The idea and design for the coupling methodology is to have minimal non-intrusiveness in ISET and to maximize the use of coupling infrastructure provided by OSU. This will allow for easy extension for other fluid models developed in the future.

The basic premise of the coupled FTSI simulation is to compute the time-dependent aerodynamic pressure and heat flux applied on the structure. These quantities are only dependent on the solution of multi-physics thermo-mechanical problem i.e., displacement and temperature. Thus, *ISET* utilizes Abaqus user subroutines `DLOAD` and `DFLUX` to read the time-dependent varying pressure on the structure and an equivalent implementation of user subroutine `URDFIL` to compute the solution at each node. The current coupling infrastructure also uses the user subroutine `UEXTERNALDB` to manage/call the fluid-solver.

6.1.2 Fluid Model

The time-dependent aerodynamic pressure applied on the top surface of the plate is computed using piston theory [**?**]. The heat flux, dependent on the displacement of the plate and the aerodynamic pressure, is computed using Eckert's reference enthalpy method [**?**].

Details on how to solve an example are provided in detail with the QAs provided with the CoupledFTSI module. A `README` file is also provided explaining the changes required to run other coupledFTSI examples.

6.2 *ISET* Gen3D

See `SetSolverTools/MeshGenerator` in hg repository
`ssh://hg@faaws2.cee.illinois.edu/software_repositories/SetSolverTools`

6.3 *ISET* SIF Plotter

Use Mathematica scripts available at `SetSolverTools/SIFPlotterMathematica`. See hg repository
`ssh://hg@faaws2.cee.illinois.edu/software_repositories/SetSolverTools`

See also `SIFPloting Gabriel_Albacarys_REU`. This has not been mantained for a long time though.

6.4 GID to ISET Converter

See `SetSolverTools/GiD-ISET` in hg repository
`ssh://hg@faaws2.cee.illinois.edu/software_repositories/SetSolverTools`

6.5 Abaqus to ISET Converter

See `SetSolverTools/AbaqusToISETConverter` (uses Mathematica),
`SetSolverTools/ModelDataFileConverters/` (C++ code),
`SetSolverTools/ModelDataFileConverters_PGupta/` (more recent version of C++ code)
in hg repository `ssh://hg@faaws2.cee.illinois.edu/software_repositories/SetSolverTools`

6.6 Code testing

Since the ISET code is in constant development, a script called qa.py that runs qa-tests (quality assurance tests) was created in order to verify the reliability of a compiled executable. The purpose of the script is to detect executable differences in the solution, adaptive process, etc., by simply diffing an existing *.extract file with a newly created *.extract file. This script performs QA (Quality Assurance) on executables only in batch mode and is not intended to test the gui or postprocessor.

6.6.1 Usage

The script uses two ASCII files (QA.files and QA.constants) which define

- paths to: *.tcl files; *.grf; *crf, etc. files; *.extract files
- QA constants

To use the QA script simply follow the steps outlined below:

1) In a terminal, go to folder SetSolver/qa-tests/scripts
2) Call qa.py script with the desired parameters. A quick explanation of the parameters is given next:

```
usage: qa.py [-h] [--speed [SPEED]] [--group [GROUP]]
[--num-procs [NUM_PROCS]] [--test-number [TEST_NUMBER]]
```

```
[--results-dir [RESULTS_DIR]] [--script-type [SCRIPT_TYPE]]
exec QAfile

positional arguments:
exec                  executable to test with QA script
QAfile                QA file containing tests to run

optional arguments:
-h, --help            show this help message and exit
--speed [SPEED], -s [SPEED]
            upper limit on test speed (A=fastest, X=slowest;
            X=default)
--group [GROUP], -g [GROUP]
            test groups to run, formatted as a sequence of letters
            (a,b,c; ab=default)
--num-procs [NUM_PROCS], -j [NUM_PROCS]
            number of tests to run (in parallel) simultaneously
            (1=default)
--test-number [TEST_NUMBER], -n [TEST_NUMBER]
            number of the specific test in the given file to run
            (None=default)
--results-dir [RESULTS_DIR], -d [RESULTS_DIR]
            name for results directory; must be a path relative to
            ./results/ (yy_mm_dd_HH_MM=default)
--script-type [SCRIPT_TYPE], -l [SCRIPT_TYPE]
            script file extension to use (default=tcl)
```

Type `qa.py --help` at the command line for more help on usage. A video tutorial is also available in ISET Documentation YouTube channel.

6.6.2 Extract Files

Before you begin using the QA script some additional details concerning the *.extract file might be helpful. First, the name of the *.extract file, automatically created by the code, is rootname.extract where rootname is typically obtained by removing the .tcl extension from the input *Tcl* file. The *.extract file contains extracted data from a given executable for a particular job that has been run. This data may include min/max values, the number of dofs, the current time, etc. The only restriction on the *.extract data is that the following format must be used when writing the file;

```
Line_number  Variable_Name  Value
.
.
.
999 None 0.0
```

In this format Line_Number is an integer representing the the current line number. The Variable_Name is a character string identifier and Value is real numerical value for the variable. Note only one variable name and value is allowed per line. Thus if you want to output a coordinate (x,y,z) then you need to put it on three separate lines. Finally, each group of extracted data in the file may be separated by a line beginning with 999 as shown above. This provides a quick reference for separating results from Newton iterations, global-local iterations, etc.

Creating Extract Files

Creating an .extract file requires no additional coding for your application. All you need to do is add the tcl option *extractFile create* to the *.tcl file and fire it from the command line, and set the environment variable DO_QA ON. This option writes the step number, number of dofs, min and max coordinates, min and max solution components, etc to the *.extract file.

6.6.3 QA.files script

This file contains a complete path to *.tcl, *.grf, *.crf, *.extract, etc. files to be used in the QA process. It can also contain information particular to a determined test.

When defining a new test, the user has to define a speed and a group. The format is as following:

```
test  speed+group
```

Groups are defined by the user and usually depend in some specific characteristic of the batch of tests. Speed is defined using one of the following categories:
a < 2 sec, b < 10 sec, c < 1 min, d < 5 min, e < 30 min

The user can also define specific parameters in a test-by-test basis. The available parameters are:

- Needs_Abaqus_User_Subs: If test needs ISET to be compiled with Fortran Abaqus User Subs
- Needs_Pardiso: Tests that only work with Pardiso solver
- Needs_Matlab: Tests that only work if ISET was linked with the Matlab engine
- NeedsIgnore_NumEquations_AsSeriousFail: Ignores number of equations diffs as a serious failure. This might happen in propagation qa-tests where a different solver may lead to slightly a different propagation that eventually may lead to different enrichments.
- Diff_Tolerance #: Changes the diff tolerance for a determined test
- Zero_Tolerance #: Changes the zero tolerance for a determined test

Next is an example QA.files script.

Example QA.files file:

```
test ba
tcl     ../L2proj/2dimension/cl_gfe_ele2d_quad_p1-3.tcl
copy    ../L2proj/2dimension/cl_gfe_ele2d_quad.grf
extract ../L2proj/2dimension/cl_gfe_ele2d_quad_p1-3.extract
go

Needs_Abaqus_User_Subs
test bb
tcl     ../abaqus_user_subs/umat/3DBeamBending/JHUMatModel/Hex20BeamBending.tcl
copy    ../abaqus_user_subs/umat/3DBeamBending/JHUMatModel/Hex20MeshSurfDisp.grf
extract ../abaqus_user_subs/umat/3DBeamBending/JHUMatModel/Hex20BeamBending.extract
go

Diff_Tolerance 0.001
Zero_Tolerance 0.00005
Needs_Pardiso
test aa
```

```
tcl     ../coupledHydroFrac/fullyCutCuboid1/mesh_1x1x1_GFEM_p1_crack.tcl
copy    ../coupledHydroFrac/fullyCutCuboid1/mesh_1x1x1_GFEM.grf
copy    ../coupledHydroFrac/fullyCutCuboid1/crack_surface_for_mesh_1x1x1_on_edge.crf
extract ../coupledHydroFrac/fullyCutCuboid1/mesh_1x1x1_GFEM_p1_crack.extract
go

Needs_Matlab
test ab
tcl     ../matlab_engine/gfem_8x8_tria_branch_step_1by8_Model.tcl
copy    ../matlab_engine/square_2d_domain_tria_8x8_withConstraints.grf
extract ../matlab_engine/gfem_8x8_tria_branch_step_1by8_ModeI.extract
go
```

6.6.4 QA.constants file

Contains various constants used by the QA script, including tolerances and environment variables.

Example QA.constants file:

```
#
# set a relative diff tolerance
Diff_Tolerance 0.0001
#
# set batch mode
setenv FLAVOR BATCH
#
# turn the qa option on in the code
setenv DO_QA ON
#
```

6.6.5 Results

The results of a run of the qa-tests will be saved in the folder SetSolver/qa-tests/scripts/results. The screen output and overall results can be seen at the file QA.summary. Information of tests that diffed or that failed will be saved on different folders for further analysis. The most important files in a folder for a failed test are:

- gold.extract: Set of results that the test has to match to determined tolerance.
- run.extract: Results from executed test. These results were compared with the results in gold.extract.
- diff.out: Diff between gold.extract and run.extract.
- run.err: Shows the error message in case the code crashed before finishing execution.
- run.out: Screen output of the test.

[heading=subbibliography]