Developing numerical programs with Jem and Jive

Dynaflow Research Group

Contents

Introduction

Modules and models

Built-in Jive modules

Built-in Jive models

Non-linear example

Introduction

This stack of slides explain how to get started with the model and module framework provided by Jive.

It comes with multiple exercises that one can use to practice with the presented information.

The solutions to the exercises are included too.

These slides assume that the reader is familiar with the essential classes and concepts provided by Jem/Jive.

Part of the slides serve as a high-level reference manual for common modules and models.

The final part deals with a non-linear example program that implements a damage mechanics model. This program could be used as the basis of your own Jive program.

Modules and models

Introduction

A numerical program typically performs the following steps:

1. create a discrete representation of the problem domain;
2. assemble and solve a system of equations;
3. save the solution and derived data.

The second and third steps are repeated in non-linear and/or dynamic applications.

Many programs use the same algorithms, but applied to different models.

For instance, the Newton-Raphson algorithm is typically used to solve non-linear problems.

What differs from one program to the next is the underlying model that determines how the non-linear system of equations is to be evaluated.

It would be advantageous of the solution algorithms could be decoupled from the models.

This decoupling would make it possible to:

- use the same solution algorithms with different models;
- use the same models with different solution algorithms.

This, in turn, would enable one to re-use the implementation of solution algorithms and models in different programs.

Ultimately, this saves time and leads to a more robust and dependable implementation of the solution algorithms and models.

Jive implements this decoupling in the form of high-level building blocks called ***modules*** and ***models***.

- Modules implement the solution algorithms; they determine ***which*** steps are executed by a program.
- Models implement the equations to be solved; they determine ***how*** these steps are executed.

Both modules and models are represented by class instances.

Modules

A module can be viewed as a mini-program that executes one particular task, such as

- reading input data;
- solving the global system of equations;
- writing output data.

Modules can be chained together to build a complete program.

Modules are separate entities. That is, a module typically does not “know” about other modules. This makes it possible to replace a module by another one with the same functionality but with a different implementation.

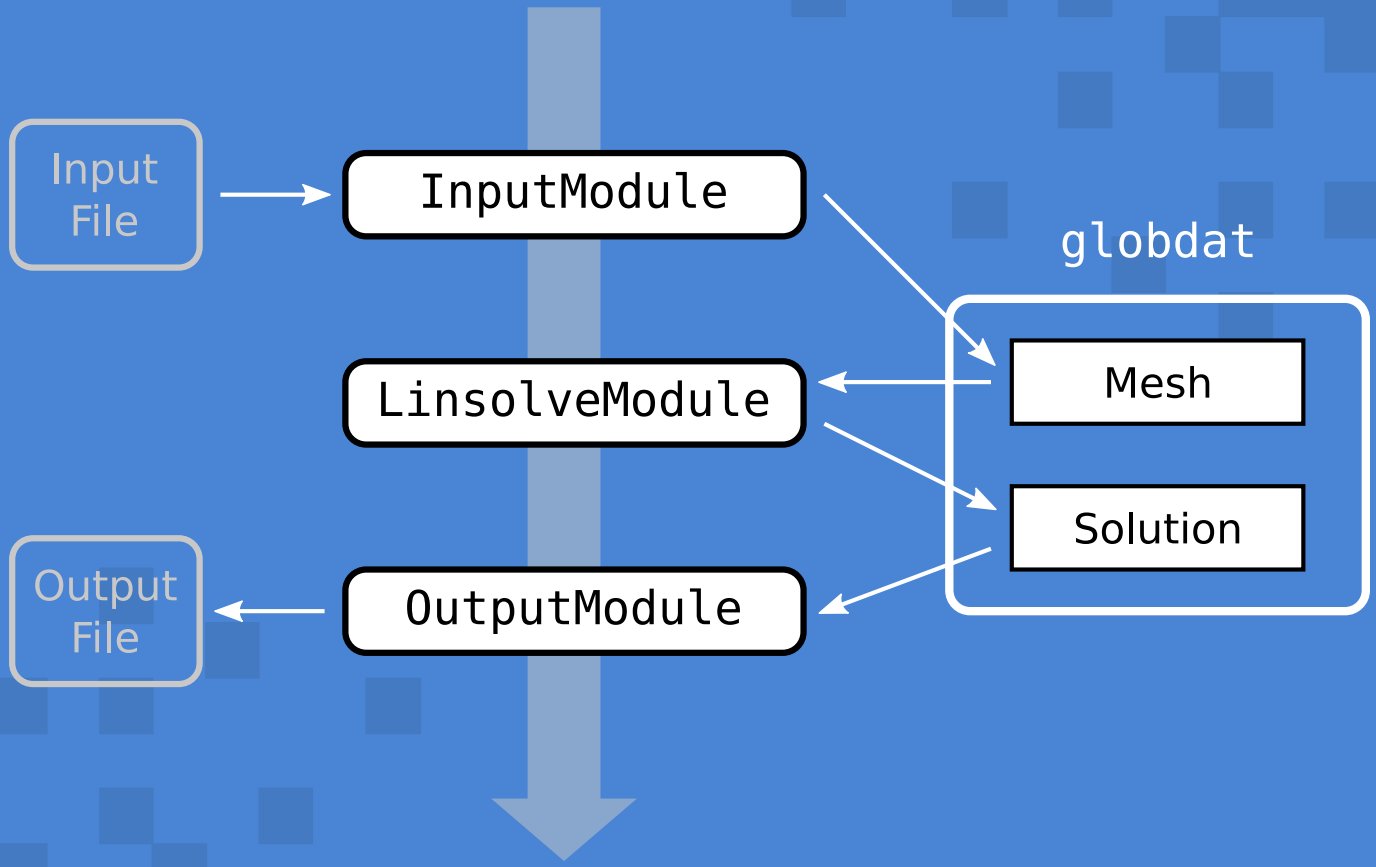
In particular, you can replace any Jive module by your own module that has been specifically tailored to your problem.

Modules communicate with each other by reading data from and writing data to an in-memory database named **`globdat`**.

For instance, the **`InputModule`** reads data from a file and stores these data in the database.

The **`LinsolveModule`** uses the data to assemble and solve the global system of equations. It then stores the solution vector in the database.

The **`OutputModule`** reads the solution vector from the database and writes it to a file.


Input File
InputModule
globdat
Mesh
LinsolveModule
Solution
Output File
OutputModule

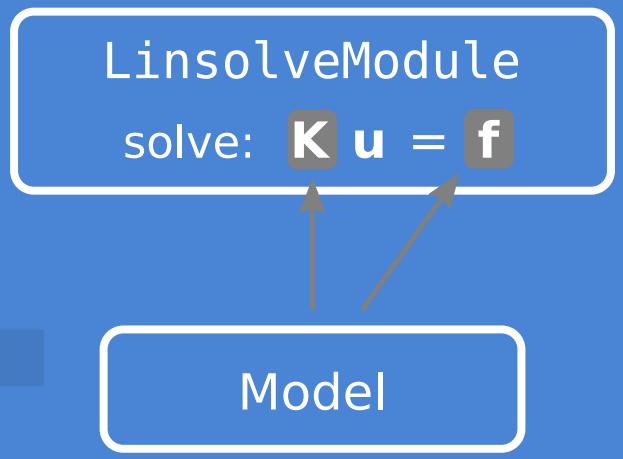
Models

A model represents a transformation from a physical model and its numerical representation.

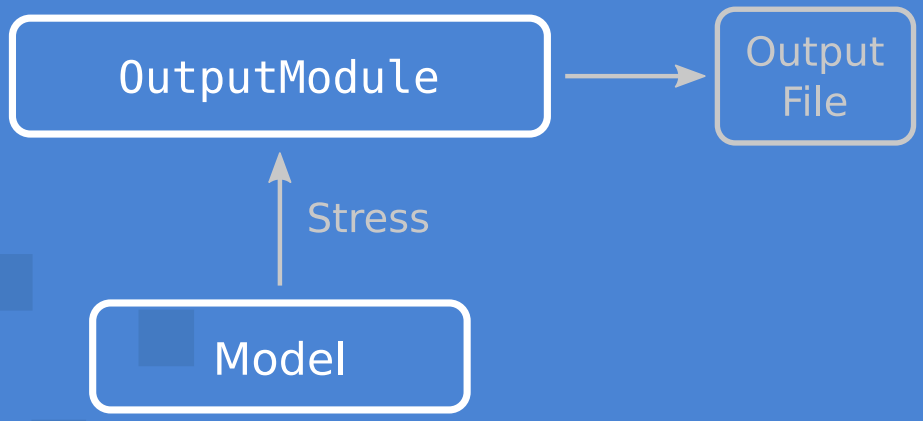
For instance, a model can encapsulate a finite element model describing the mechanical behavior of a solid material.

Models indicate ***how*** modules should execute particular operations in a program.

Example: the **`LinsolveModule`** asks the model to assemble the global matrix and the right-hand side vector.



Another example: the **`OutputModule`** asks the model to compute the stress field on basis of the displacement field.



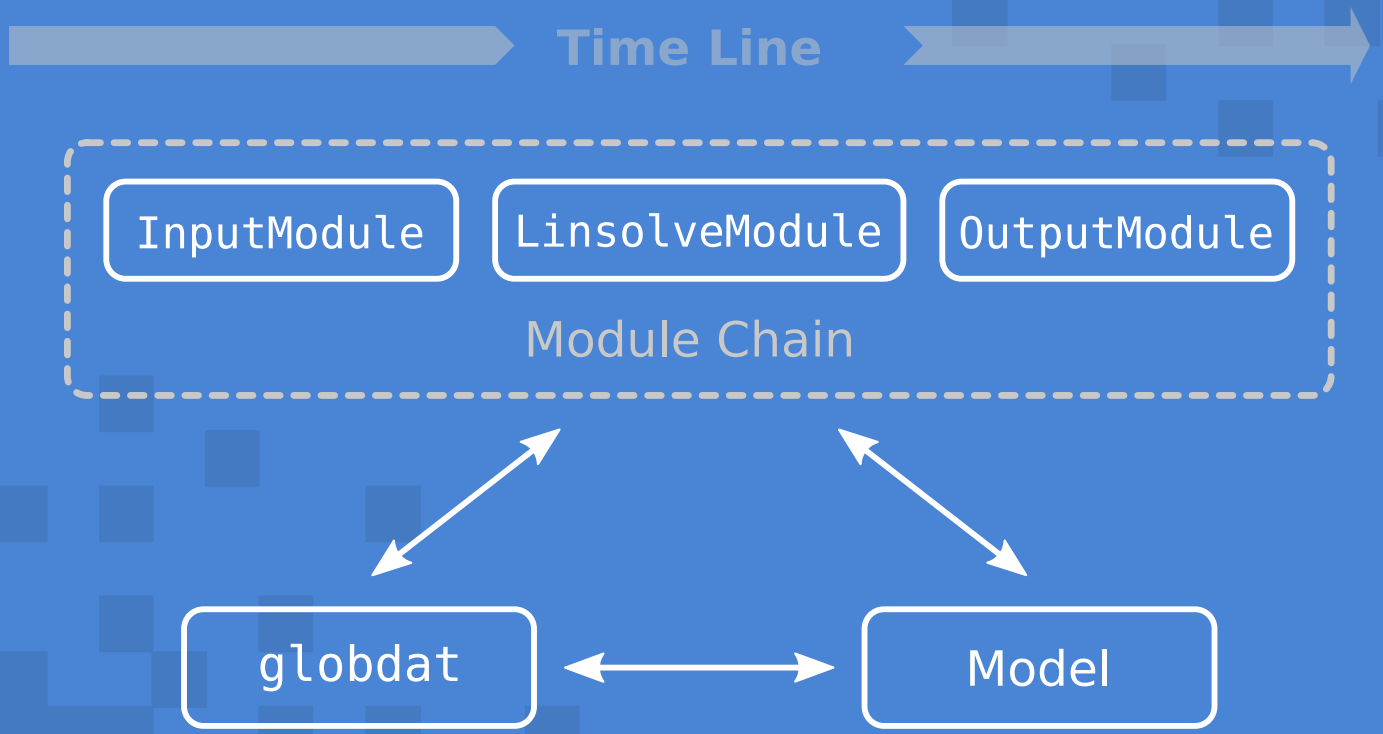
Models also have access to the database **`globdat`**.

They can use the data stored in **`globdat`** to fulfill the requests made by the modules.

For instance, a model can get the finite element mesh from the database in order to assemble the system of equations.

Note that the models themselves are stored in the database so that they can be found by the modules.

How it all fits together:



A model is often composed of multiple sub-models that are organised in a tree-like structure.

Each sub-model focuses on one aspect of the transformation from a physical model and its numerical representation.

For instance, the `LoadModel` might handle the assembly of the right-hand side vector.

More about sub-models later.

Exercise: use modules

Compile and run a Jive program that consists of modules and models.

The program solves a dynamic heat transfer problem.

Take a look at the source code and input files.

Exercise location: **`exercises/heat`**.

Details

Run the program like this:

```
./heat example.pro
```

Or, on Windows:

```
heat.exe example.pro
```

Try to change the heat source function by editing the file **`example.pro`**.

How to use modules

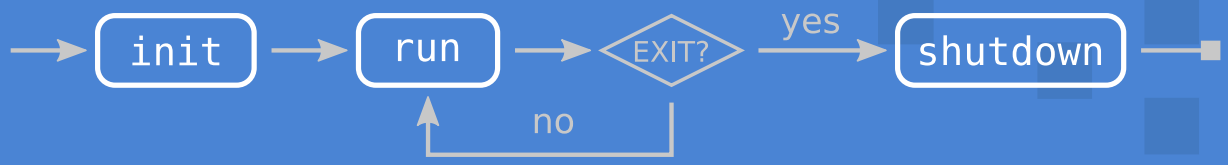
A module is an instance of a class that is derived from the **`Module`** class (package **`app`**). This class has three important member functions:

`init`
: This function is called once at the start of a program.

`run`
: This function is called multiple times during the execution of the program.

`shutdown`
: This function is called once at the end of the program.

Module function call flow chart:



The **`run`** function typically executes one time step or one loading step, but this is not strictly defined.

Some modules, such as the **`InputModule`**, only implement the **`init`** function; they are only active once after the program gas started.

The **Module** class definition:

```
class Module : public jem::NamedObject
{
 public:

  virtual Status           init

    ( const Properties&      conf,
      const Properties&      props,
      const Properties&      globdat );

  virtal Status            run

    ( const Properties&      globdat );

  virtual void             shutdown

    ( const Properties&      globdat );

};
```

The functions **`init`** and **`run`** return an integer code indicating the status of the module.

The code can be one of:

`OK` Indicates that the module desires to continue the current computation.

`DONE`
Indicates that the module has performed its task and can be discarded.

`EXIT`
Indicates that the program should be stopped.

The **`Module`** member functions are called with a **`Properties`** object that represents the global database.

Modules can store data in the global database by calling the **`set()`** function.

They can read data from the database by calling the **`get()`** function.

By convention, the global database object is named **`globdat`**.

When a module is initialised, it is also passed a **`Properties`** object, named **`props`**, that contains all runtime parameters.

A module can read its own runtime parameters from this **`Properties`** object.

A third **`Properties`** object, named **`conf`**, is used to report back all runtime parameters that have been used.

The **`conf`** object starts out empty and is then filled by each module. It will include the parameters in **`props`** and all default values that are not in **`props`**.

A **`Module`** has a name that is used for output messages and for finding its runtime parameters in the **`props`** object.

For instance, an **`OutputModule`** named **`output`** will search the **`props`** object for a sub-property object named **`output`** containing its runtime parameters.

You can create multiple modules of the same type; two output modules, for instance. By assigning them different names they can be configured differently.

More about runtime parameters later.

Modules example

The next couple of slides show an example involving three modules: the **`InputModule`**, the **`LinsolveModule`** and the **`OutputModule`**.

The runtime parameters are read from a file named **`input.pro`** and stored in the **`props`** variable.

The class **`Globdat`** is used to initialise the global database **`globdat`**.

Modules example: initialisation phase.

```
Ref<Module>  modules[3];
Properties   globdat;
Properties   props;
Properties   conf;
int          status;

modules[0] = newInstance<InputModule>     ();
modules[1] = newInstance<LinsolveModule> ();
modules[2] = newInstance<OutputModule>    ();

props.parseFile ( "input.pro" );
Globdat::init    ( globdat );

for ( int i = 0; i < 3; i++ )
{
  modules[i]->init ( conf, props, globdat );
}

System::out() << "Runtime parameters:\n\n";
System::out() << conf << "\n\n";

// Continued on the next slide ...
```

Modules example: run phase.

```
status = Module::OK;

while ( status != Module::EXIT )
{
  for ( int i = 0; i < 3; i++ )
  {
    if ( modules[i] != NIL )
    {
      status = modules[i]->run ( globdat );

      if       ( status == Module::DONE )
      {
        modules[i] = NIL;
      }
      else if ( status == Module::EXIT )
      {
        break;
      }
    }
  }
}

// Continued on the next slide ...
```

Modules example: shutdown phase.

```
for ( int i = 0; i < 3; i++ )
{
  if ( modules[i] != NIL )
  {
    modules[i]->shutdown ( globdat );
  }
}

// Only for debugging purposes.

System::out() << "Global database:\n\n";
System::out() << globdat << "\n\n";
```

If you would try to run the example code you would get an error.

The reason is that the **`LinsolveModule`** requires a model to assemble the system of equations and right-hand side vector.

As no model has been defined the **`LinsolveModule`** will raise an error when its **`init`** function is called.

More about models and their construction later.

Built-in modules

Jive provides a collection of built-in modules, including:

`DebugModule`
: Prints the state of the global database each time one of its member functions is called.

`ChainModule`
: Encapsulates a list of modules.

`InputModule`
: Reads input data from an XML-formatted file.

`MeshgenModule`
: Generates a finite element mesh.

`InitModule`

Initializes data structures that are required later in a program.

`InfoModule`

Prints a summary of the data stored in the global database and prints progress information.

`OutputModule`

Writes selected data sets to an XML-formatted output file.

`SampleModule`

Writes scalar values in a tabular format to an output file.

`ControlModule`

Controls when a simulation should be terminated and provides support for executing commands interactively.

`FemViewModule`

Provides support for real-time data visualization.

`GraphModule`

Shows scalar values in a graph.

Some of these modules are explained in more detail later.

Many generic modules are defined in the package **`app`**.
Application-specific modules are defined in other packages, such as the **`fem`** and **`implict`** packages.

Exercise: run modules

Implement a program that runs the **`InputModule`** (package **`fem`**) and the **`ControlModule`** (package **`app`**).

Read the runtime parameters (**`props`**) from the input file **`input.pro`**.

Print the global database when the program has finished.

Exercise location: **`exercises/modules`**.

Solution location: **`solutions/modules-1`**.

Using the `exec` function

You can create and manage a modules yourself, but you can also let Jive take care of that by calling the `exec()` function of the `Application` class.

Using the `exec` function saves you from writing boiler-plate code and makes your program more robust.

The `exec()` function:

- initializes the standard output streams;
- installs signal handlers for catching memory errors;
- reads the runtime parameters from a file specified on the command line;
- creates and initializes the global database;
- and creates the main module.

The **exec()** function also:

- calls the **init()** member function of the main module with the runtime parameters and the global database as arguments;
- repeatedly calls the **run()** member function of the main module until it returns **EXIT**;
- calls the **shutdown()** member function of the main module;
- restores the original signal handlers and output streams;
- and handles any exceptions that are thrown during the execution of the above tasks.

The main module is created by calling a ***module construction function*** that you must pass as an argument to the **exec()** function.

The module construction function may have any name as long as it has the following signature:

```
Ref<Module> func-name ();
```

As only one module can be returned, the module construction function typically creates a **ChainModule** to which a series of child modules are added.

Example of a module construction function:

```
Ref<Module> mainModule ()
{
  Ref<ChainModule>  mainChain = newInstance<ChainModule> ();

  mainChain->pushBack ( newInstance<InputModule>    ( "input"   ) );
  mainChain->pushBack ( newInstance<InitModule>     ( "init"    ) );
  mainChain->pushBack ( newInstance<InfoModule>     ( "info"    ) );
  mainChain->pushBack ( newInstance<NonlinModule>   ( "nonlin"  ) );
  mainChain->pushBack ( newInstance<ViewModule>     ( "view"    ) );
  mainChain->pushBack ( newInstance<ControlModule>  ( "control" ) );

  return mainChain;
}

int main ( int argc, char** argv )
{
  return Application::exec ( argc, argv, mainModule );
}
```

The **ChainModule** encapsulates an array of modules. Whenever one of its member functions is called, it will call the same member function for each of these modules.

Example:

```
void example ()
{
  Properties         globdat, props, conf;
  Ref<ChainModule>   chain;

  chain = newInstance<ChainModule> ();

  chain->pushBack ( newInstance<InputModule>    () );
  chain->pushBack ( newInstance<LinsolveModule>() );

  chain->init     ( conf, props, globdat );
  chain->run      ( globdat );
  chain->shutdown ( globdat );
}
```

In this example, the function call

```
chain->init ( conf, props, globdat );
```

will result in two calls to the **init** functions of the **InputModule** and **OutputModule**, in this order.

Likewise, the function call

```
chain->run ( globdat );
```

will call the **run** functions of the **InputModule** and **OutputModule**.

As an alternative to a module construction function, the main module chain could have been passed to the **exec** function:

```
int main ( int argc, char** argv )
{
  Ref<ChainModule>  chain = newInstance<ChainModule> ();

    :

  return Application::exec ( chain, argc, argv );
}
```

With this design, however, you would have to handle any error that might occur during the creation of the main module chain.

In addition, this would make it more difficult to run a Jive program on multiple processor cores.

Note that the module construction function is called only once!

This means, for instance, that any print statements added to the construction function are executed once at the start of a program when the main module chain is constructed.

They are ***not*** executed during the execution of the module chain!

How ***not*** to add print statements:

```
Ref<Module> mainModule ()
{
  Ref<ChainModule>  mainChain = newInstance<ChainModule> ();

  mainChain->pushBack ( newInstance<InputModule> ( "input" ) );

  System::out() << "after input\n";

  mainChain->pushBack ( newInstance<InitModule>  ( "init"  ) );

  System::out() << "after init\n";

  mainChain->pushBack ( newInstance<InfoModule>  ( "info"  ) );

  System::out() << "after info\n";

    :

  return mainChain;
}
```

Exercise: use the `exec` function

Modify the program from the previous exercise so that it uses the `exec` function.

Use the same modules and input files as before and add them to a `ChainModule`. Add the `InfoModule` to see what is going on.

Exercise location: `exercises/modules`.

Solution location: `solutions/modules-2`.

How to use models

A model represents a transformation between a physical model and its numerical representation.

For instance, a model could encapsulate a finite element model describing the mechanical behavior of a solid material.

This model would assemble the global stiffness matrix, and, optionally, compute the stress and strain field for a given displacement field.

Many models involve a (non-linear) system of equations that can be written as:

$$f(u, \dot{u}, \ddot{u}, t) = g(t)$$

in which:

- f is the so-called ***internal vector***;
- u is the so-called ***state vector*** (solution vector);
- $\dot{u}$ is the first-order time derivative of the state vector;
- $\ddot{u}$ is the second-order time derivative of the state vector;
- g is the so-called ***external vector***.

Note that the internal vector is a function of the state and its time derivatives, while the external vector is only a function of time.

The non-linear system of equations is typically linearised as follows:

$$\boldsymbol{f}_0 + \left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}} + \frac{\partial \boldsymbol{f}}{\partial \dot{\boldsymbol{u}}} \frac{\partial \dot{\boldsymbol{u}}}{\partial \boldsymbol{u}} + \frac{\partial \boldsymbol{f}}{\partial \ddot{\boldsymbol{u}}} \frac{\partial \ddot{\boldsymbol{u}}}{\partial \boldsymbol{u}} \right) \delta \boldsymbol{u} = \boldsymbol{g}(t)$$

in which:

$(\partial \boldsymbol{f} / \partial \boldsymbol{u})$ is the so-called ***zero-order*** tangent matrix;
$(\partial \boldsymbol{f} / \partial \dot{\boldsymbol{u}})$ is the ***first-order*** tangent matrix;
$(\partial \boldsymbol{f} / \partial \ddot{\boldsymbol{u}})$ is the ***second-order*** tangent matrix.

Note that these matrices are often called the ***stiffness matrix***, the ***damping matrix*** and the ***mass matrix***, respectively.

In many applications

$$\frac{\partial \dot{\boldsymbol{u}}}{\partial \boldsymbol{u}} = \alpha \boldsymbol{I}, \qquad \frac{\partial \ddot{\boldsymbol{u}}}{\partial \boldsymbol{u}} = \beta \boldsymbol{I}$$

resulting in the following linearised system of equations:

$$\boldsymbol{f}_0 + \left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}} + \alpha \frac{\partial \boldsymbol{f}}{\partial \dot{\boldsymbol{u}}} + \beta \frac{\partial \boldsymbol{f}}{\partial \ddot{\boldsymbol{u}}} \right) \delta \boldsymbol{u} = \boldsymbol{g}(t)$$

where α and β depend on the chosen time integration scheme.

A model essentially has to implement the algorithms for computing the internal and external vectors, and the relevant (linearised) matrices.

The input of a model consists of the state vector (and its time derivatives), a discrete representation of the domain, and all required model parameters.

The input data can be obtained from the global database **`globdat`** and the **`props`** object containing the runtime parameters.

The `Model` class

A model is an instance of a class derived from the `Model` class (package `model`).

The interface of this class essentially consists of the member function `takeAction()` that asks a model to perform a specific action.

This function is the primary channel through which modules and models communicate with each other.

The **Model** class definition:

```
class Model : public jem::NamedObject
{
 public:

  virtual bool             takeAction

    ( const String&          action,
      const Properties&      params,
      const Properties&      globdat );

 protected:

  String                   myName_;

};
```

The protected member **myName_** plays an important role when dealing with runtime parameters. More about this later.

The `takeAction` function has three parameters:

`action`
: A `String` that specifies what kind of action the model should take.

`params`
: A `Properties` object containing (temporary) data that are specific to the action.

`globdat`
: A `Properties` object representing the global database.

The `takeAction()` function should return `true` if the model has performed the requested action, and `false` otherwise.

A model can be requested to perform an action it does not know or care about. In that case the model should just ignore the action and pass it on to its child models, if any.

This will become clear when model trees are explained.

Example of the **takeAction()** function:

```
bool ExampleModel::takeAction

   ( const String&      action,
     const Properties&  params,
     const Properties&  globdat )

{
  if ( action == "GET_EXT_VECTOR" )
  {
    Vector  fext;

    params.get ( fext, "extVector" );

    fext += 1.0;

    return true;
  }

  return false;
}
```

In this example the **`ExampleModel`** handles the **`GET_EXT_VECTOR`** action by adding a contribution to the external (right-hand side) vector.

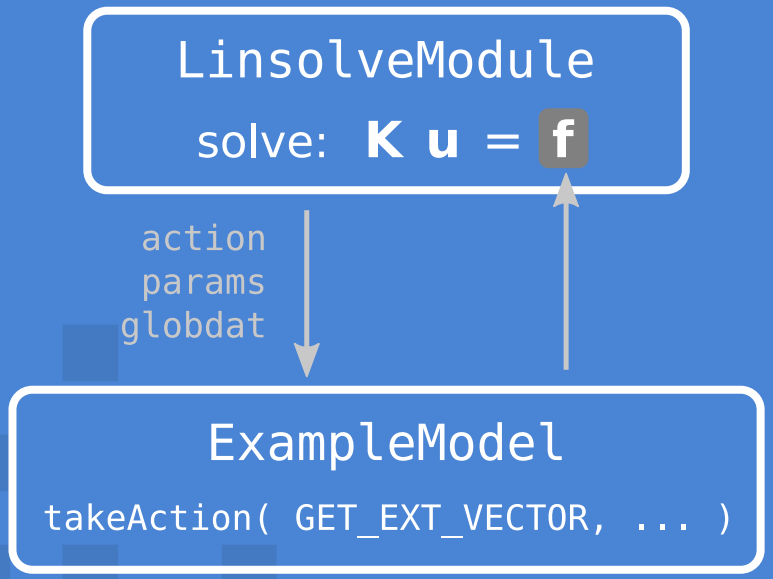
The external vector is obtained from the **`params`** parameter (note that a shallow copy is made).

The **`params`** parameter contains different data for different actions.

For instance, **`params`** will contain a **`MatrixBuilder`** when the actions equals **`GET_MATRIX0`**.

The **takeAction** function is called by the modules when they need data from a model.

For instance, the **LinsolveModule** calls the **takeAction** function to get the external vector:



This is what the code in the **LinsolveModule** looks like:

```
Status LinsolveModule::run ( const Properties& globdat )
{
  Ref<Model>     model;
  Ref<DofSpace>  dofs;
  Properties     params;
  Vector         fext;

  // Get the Model and DofSpace from globdat (not shown here).

  model = ...;
  dofs  = ...;

  fext.resize ( dofs->dofCount() );

  fext  = 0.0;

  params.set  ( "extVector", fext );

  model->takeAction ( "GET_EXT_VECTOR", params, globdat );

  return OK;
}
```

Jive defines a collection of standard actions and corresponding action parameters that are used by the built-in Jive modules.

These are described later.

Before that, let's take a closer look at how models are created.

Model construction

Models are (normally) constructed by the **`InitModule`**.

The **`InitModule`** searches the runtime parameters stored in **`props`** for a typed property set named **`model`** and uses that to create the model.

It then stores the model in **`globdat`** so that other modules can access the model.

Here is an example of the runtime parameters that the **`InitModule`** uses to create an **`ExampleModel`**:

```
model =
{
  type  = "Example";    // Model type.
  alpha = 1.0;          // Model parameter.
  beta  = 2.0;          // Another model parameter.
};
```

`InitModule` uses the **`type`** property to create an instance of the **`ExampleModel`**.

It does this by looking up a so-called ***model construction function*** that has been registered with the type string **`Example`**.

A model construction function is responsible for creating a model given an number of parameters.

It should be declared like this:

```
Ref<Model> newExampleModel ( const String&      name,
                             const Properties&  conf,
                             const Properties&  props,
                             const Properties&  globdat );
```

The name of the function is not relevant, as long as it has the same return type and parameter list.

The model construction function should have the following four parameters:

`name`
: The name of the model. This can be used to obtain the runtime parameters for the model. The name is determined by the **`InitModule`**.

`conf`
: A property set for storing the runtime parameters used by the model. This is for output.

`props`
: The runtime parameters.

`globdat`
: The global database.

The model construction function must be registered with the **`ModelFactory`** so that the **`InitModule`** is able to find it.

This is done as follows:

```
ModelFactory::declare ( "Example", newExampleModel );
```

The first parameter of the **`declare`** function is the type name of the model class. This can be any name, as long as it is equal to the type name specified in the runtime parameters.

The second parameter is a pointer to the model construction function.

A model must have been registered before the **`InitModule`** becomes active.

A good place to register a model is the main module construction function as this function will be called before any module becomes active.

A good practice is to implement a small function named **`declareExampleModel`**, say, that registers the model. This function is then called in the main module construction function.

This is illustrated in the next slides.

Recipe for implementing a new model

Suppose you want to implement a new model named **`ExampleModel`**. You can do that by means of the recipe outlined below.

Step 1: define a class named **`ExampleModel`** that is derived from the **`Model`** class.

You must (at least) declare a constructor and the **`takeAction`** function.

Definition of the **ExampleModel** class:

```
#include <jive/model/Model.h>

class ExampleModel : public jive::model::Model
{
 public:

                              ExampleModel

    ( const String&           name,
      const Properties&       conf,
      const Properties&       props,
      const Properties&       globdat );

  virtual bool                takeAction

    ( const String&           action,
      const Properties&       params,
      const Properties&       globdat );

};
```

Step 2: implement the constructor in which you initialise the data members of your class.

You will typically need to use the runtime parameters and get data from the global database. This is explained in more detail later.

The **ExampleModel** constructor:

```
ExampleModel::ExampleModel

  ( const String&      name,      // Name of the model.
    const Properties&  conf,      // Runtime parameters (out).
    const Properties&  props,     // Runtime parameters (in).
    const Properties&  globdat )  // Global database.

    : Model ( name )              // Initialise the base class.

{
  // Get the runtime parameters associated with this object.

  Properties  myProps = props.getProps  ( name );
  Properties  myConf  = conf .makeProps ( name );

  // Get data from the global database and get the runtime
  // parameters.

  globdat.get ( ... );
  myProps.get ( ... );
  myConf .set ( ... );
}
```

Step 3: implement the `takeAction()` function. In particular, determine which actions should be supported by your model and implement the code for executing those actions.

In many cases your model will need to support at least the `GET_INT_VECTOR` and `GET_MATRIX0` actions.

The `takeAction` function:

```
bool ExampleModel::takeAction

   ( const String&      name,
     const Properties&  params,
     const Properties&  globdat )

{
  if ( action == "GET_INT_VECTOR" )
  {
    Vector  fint;

    params.get ( fint, "intVector" );

      :

    return true;
  }

  return false;
}
```

Step 4: implement a model construction function that returns an instance of your model class.

Here is an example:

```
Ref<Model>              newExampleModel

   ( const String&       name,
     const Properties&   conf,
     const Properties&   props,
     const Properties&   globdat )

{
  return newInstance<ExampleModel> ( name, conf, props, globdat );
}
```

Step 5: implement a function that registers the model construction function with the **`ModelFactory`** class.

Here is an example:

```
#include <jive/model/ModelFactory.h>

void declareExampleModel ()
{
  using jive::model::ModelFactory;

  ModelFactory::declare ( "Example", newExampleModel );
}
```

Call this function in the main module construction function.

Exercise: implement a model

Implement a model called **`ExampleModel`** that prints the **`action`** string in its **`takeAction`** function.

Try to use the **`props`** object to get a value for the private member **`parameter_`**.

Exercise location: **`exercises/model`**.

Solution location: **`solutions/model-1`**.

Details

Run the program like this:

```
./model input.pro
```

On Windows:

```
model.exe input.pro
```

Do you understand the output of the program? What happens when you modify the input files?

Standard actions

Jive defines a number of standard actions that are used by the built-in Jive modules to communicate with models.

These standard actions are declared as string constants in the class **`Actions`** (package model).

The class **`ActionParams`** declares string constants for accessing the action-specific parameters.

Example with the **Actions** and **ActionParams** classes:

```
bool ExampleModel::takeAction

   ( const String&      action,
     const Properties&  params,
     const Properties&  globdat )

{
  using jive::model::Actions;
  using jive::model::ActionParams;

  if ( action == Actions::GET_EXT_VECTOR )
  {
    Vector  fext;

    params.get ( fext, ActionParams::EXT_VECTOR );

    fext += 1.0;

    return true;
  }

  return false;
}
```

You should use the string constants from the **`Actions`** and **`ActionParams`** classes to avoid runtime errors.

For instance, this expression

```
if ( action == "GET_EXT_VETCOR" )
```

is wrong but will compile without errors.

In contrast, the error in this expression

```
if ( action == Actions::GET_EXT_VETCOR )
```

will be caught by the compiler.

Here is an overview of the most important actions declared by the **`Actions`** class. The overview also lists the action-specific parameters.

`INIT`

Indicates that a model should initialise any data that has not yet been initialized in the constructor of the model.

Parameters: none.

`SHUTDOWN`

Indicates that a model is about to be destroyed.

Parameters: none.

GET_INT_VECTOR

Indicates that a model should assemble the internal vector.

Parameters: **INT_VECTOR**.

GET_EXT_VECTOR

Indicates that a model should assemble the external vector.

Parameters: **EXT_VECTOR**.

GET_MATRIX0

Indicates that a model should assemble the stiffness matrix ***together*** with the internal vector.

Parameters: **INT_VECTOR**, **MATRIX0**.

GET_MATRIX1

Indicates that a model should assemble the damping matrix.

Parameters: **MATRIX1**.

GET_MATRIX2

Indicates that a model should assemble the mass matrix.

Parameters: **MATRIX2**.

GET_CONSTRAINTS

Indicates that a model should update the constraints for the global system of equations.

Parameters: **CONSTRAINTS**.

ADVANCE

Indicates that a model should advance to the next time step or loading step.

Parameters: none.

COMMIT

Indicates that a model should commit the latest changes to its internal state because a new solution of the global system of equations has been determined.

Parameters: none.

CHECK_COMMIT

Indicates that a model should check whether the current solution should be committed or not. This action always comes before the **COMMIT** action.

Parameters: **ACCEPT**.

CANCEL

Indicates that a model should undo the changes to its internal state since the last commit.

Parameters: none.

GET_TABLE

Indicates a model should store output data in a **Table** object.

Parameters: **TABLE**, **TABLE_NAME**, **TABLE_WEIGHTS**.

The **ActionParams** class provides the following string constants for accessing the action-specific data in the **params** parameter of the **takeAction()** function.

INT_VECTOR
: The name of the internal vector. Used with the actions **GET_INT_VECTOR** and **GET_MATRIX0**.

EXT_VECTOR
: The name of the external vector. Used with the action **GET_EXT_VECTOR**.

MATRIX0
: The name of a **MatrixBuilder** object for assembling the stiffness matrix. Used with the action **GET_MATRIX0**.

MATRIX1

The name of a **MatrixBuilder** object for assembling the damping matrix. Used with the action **GET_MATRIX1**.

MATRIX2

The name of a **MatrixBuilder** object for assembling the mass matrix. Used with the action **GET_MATRIX2**.

CONSTRAINTS

The name of a **Constraints** object. Used with the action **GET_CONSTRAINTS**.

TABLE

The name of a **Table** object in which a model should store output data. Used with the action **GET_TABLE**.

TABLE_NAME

The name of a string specifying which data are to be stored in the **Table** object. Used with the action **GET_TABLE**.

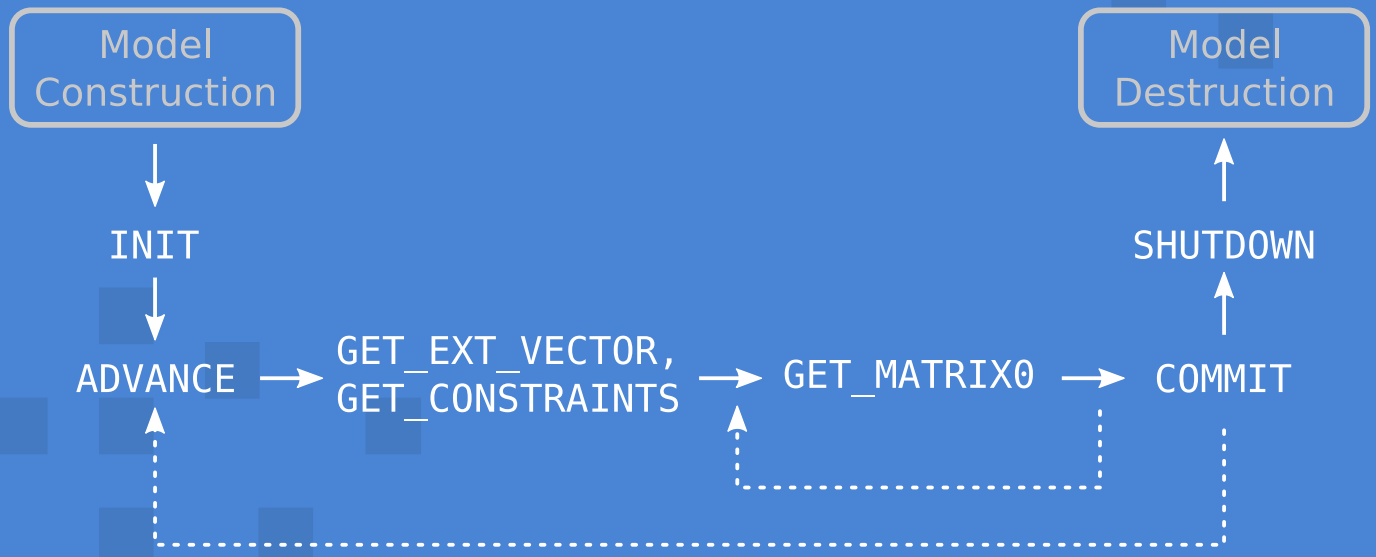
TABLE_WEIGHTS

The name of a vector for scaling the rows of the **Table**. Used with the action **GET_TABLE**.

ACCEPT

The name of a boolean value that indicates whether a model accepts the current solution of the global system of equations. Used with the action **CHECK_COMMIT**.

Standard actions flow chart:



In a linear application the flow chart is traversed once.

In a non-linear application the action **GET_MATRIX0** will be requested multiple times until a converged solution has been found.

In this case the action **CHECK_COMMIT** is requested before the **COMMIT** action.

In a time-dependent or step-wise non-linear application the actions from **ADVANCE** to **COMMIT** are repeated for each time step or loading step.

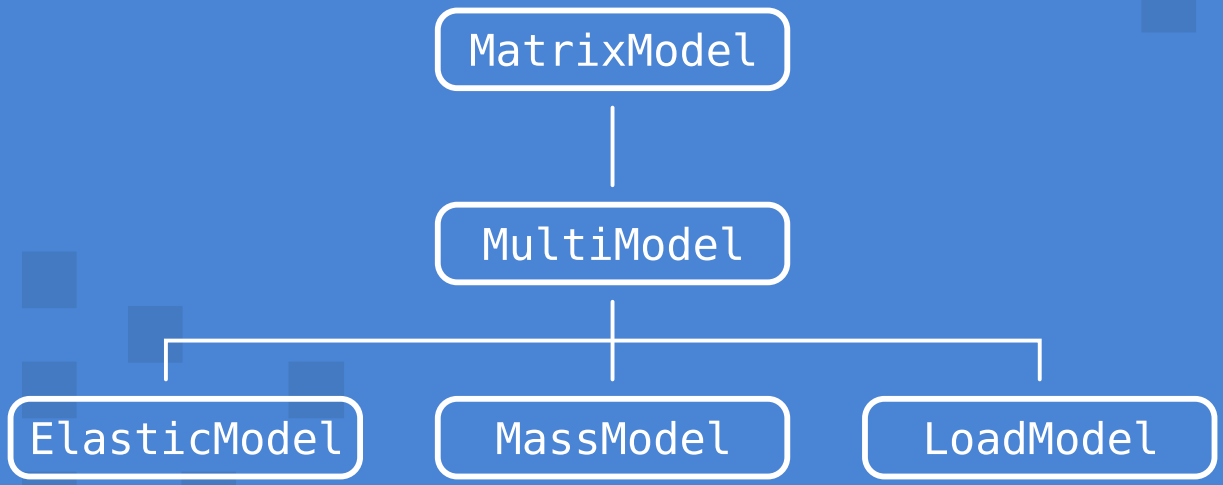
Model tree

A model is generally composed of multiple sub-models that are organised in a tree-like structure.

That is, there is one top-level model that has zero or more ***child models***. These child models, have zero or more child models, and so on.

Each model in the tree focuses on one particular aspect of the transformation from the physical model and its numerical representation.

A typical model tree:



The **`MatrixModel`** manages the **`MatrixBuilder`** that is used to assemble the global matrix.

The **`MultiModel`** has a similar role as the **`ChainModule`**; it essentially represents a list of models.

The **`ElasticModel`** assembles the global stiffness matrix and the internal vector.

The **`MassModel`** assembles the global mass matrix.

The **`LoadModel`** assembles the external vector containing the effects of external loads.

The model tree is created by the **`InitModule`**. To be precise, the **`InitModule`** creates the top-level model, that, in turn, will create its direct child model(s).

This process continues recursively until all models have been constructed.

Both the **`InitModule`** and the intermediate models use the runtime parameters stored in the **`props`** object to determine which models are to be created.

Ignoring the details for now, the next slide shows an input file for constructing the previous model tree.

Runtime parameters for the previous model tree:

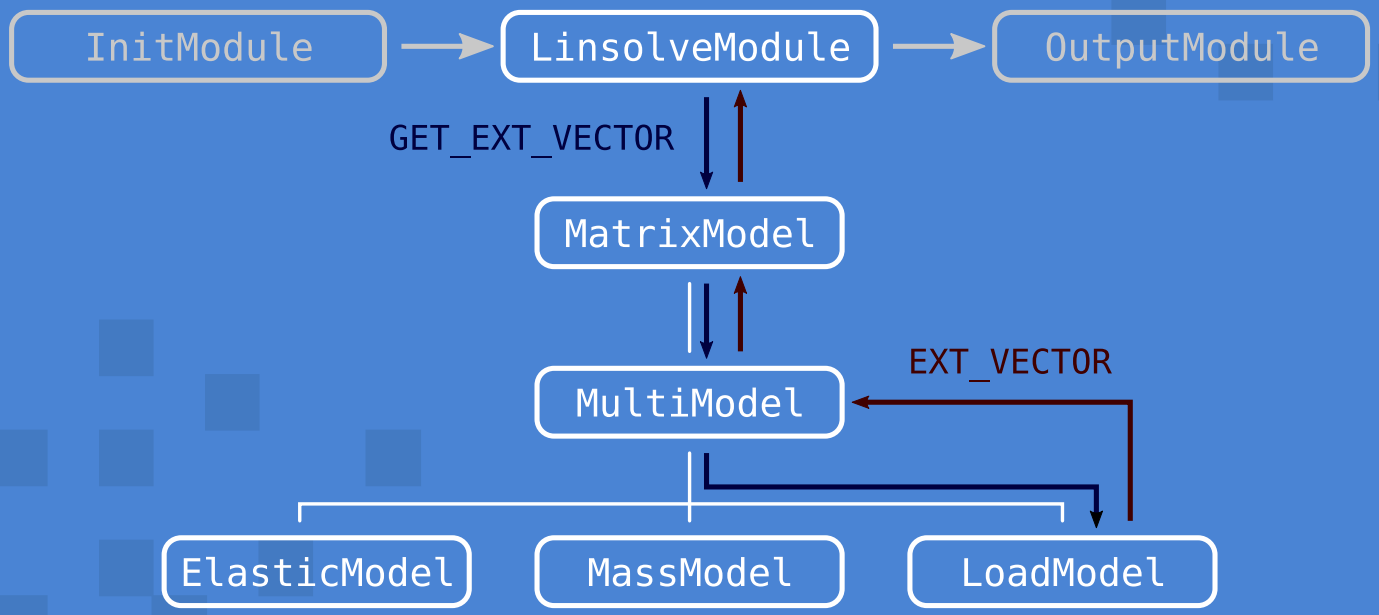
```
model =
{
  type  = "Matrix";
  model =
  {
    type   = "Multi";
    models = [ "elastic", "mass", "load" ];

    elastic =
    {
      type = "Elastic";

      // Material parameters ...
    };

    mass = { ... };
    load = { ... };
  };
};
```

Calls to the **takeAction** function are executed recursively. That is, a model calls the **takeAction** function of its child models when its own **takeAction** function is called.

In this way an action traverses the entire model tree even when an intermediate model does not handle the action.

For instance, the action **GET_EXT_VECTOR** will traverse the entire tree and will be handles by the **LoadModel**.

Action traversal of a model tree:



Built-in models

The **`model`** package provides a collection of generic models, including:

`DebugModel`
: Prints which action is being performed and prints the parameters passed to the **`takeAction`** function.

`MultiModel`
: Encapsulates an array of child models. This is a basic building block for composing model trees.

`MatrixModel`

Creates **`MatrixBuilder`** objects for assembling the global matrices. This model should be located at or near the root of the model tree.

`LoadScaleModel`

Scales the external vector with a fixed scalar value or a user-defined function.

`PointLoadModel`

Adds values from a **`Table`** object (from an input file, for instance) to the external vector.

`ContraintsModel`

Uses a **`Table`** object to modify the constraints for the global system of equations.

The **`femodel`** package also provides the **`LoadModel`** that evaluates a distributed load and adds it to the external vector.

The built-in Jive models are ***not*** registered with the **`ModelFactory`** class by default.

You will have to register the models that you want to use in the main module construction function. In this way you have complete control over which models are available in your program.

The **`model`** and **`femodel`** packages provide the function **`declareModels`** that will register all models in those packages.

Typical program set up:

```
#include <jive/model/declare.h>
#include <jive/femodel/declare.h>

// More header files ...

Ref<Module> mainModule ()
{
  Ref<ChainModule>  chain = newInstance<ChainModule> ();

  jive::model  ::declareModels ();
  jive::femodel::declareModels ();

  // Declare your own models ...
  // Build the module chain ...

  return chain;
}

int main ( int argc, char** argv )
{
  return Application::exec ( argc, argv, mainModule () );
}
```

Exercise: implement a model tree

Extend the program from the previous exercise so that it creates a model tree instead of a single model.

Use the **`MultiModel`** to add multiple instances of the **`ExampleModel`**. Print the name of each model as it is constructed.

Exercise location: **`exercises/model`**.

Solution location: **`solutions/model-2`**.

The global database

The data stored in the global database are organised in a specific way so that a module is able to find the data stored by another module.

To avoid name conflicts similar types of objects are often stored in a separate `Properties` object in the database.

One can view the global database as a kind of file system containing (sub) directories and objects.

The exact organisation of the global database is not that important because all relevant classes in Jive provide a static member function named **`get`** that retrieves an object from the database.

Here is an example:

```
void example ( const Properties& globdat )
{
  NodeSet     nodes = NodeSet   ::get ( globdat, "example" );
  ElementSet  elems = ElementSet::get ( globdat, "example" );

    :
}
```

The declaration of the **get** function differs somewhat from one class to the next, but in general it is declared as follows:

```
Ref<Type> get ( ...,
                const Properties&  globdat,
                const String&      context );
```

Where **Type** is the class type.

The three dots represent additional parameters that are needed to retrieve the object from the database.

The parameter **context** is a string that indicates from which context the **get** function is called.

The `context` string is used to compose an error message in the event that the requested object is not present in the database. It is mainly there to help you find the origin of the problem.

You can pass any string you like as the context string; Jive does not care about its contents.

Many classes, including the `Model` and `Module` class provide a member function `getContext` that will return an appropriate context string.

How to use the **`get`** functions to retrieve objects from the global database **`globdat`**:

```
ExampleModel::ExampleModel

  ( const String&       name,
    const Properties&   globdat ) : Model ( name )

{
  String          context = getContext ();

  NodeSet         nodes   = NodeSet    ::get ( globdat, context );
  ElementSet      elems   = ElementSet ::get ( globdat, context );

  Ref<XDofSpace>     dofs = XDofSpace  ::get ( nodes.getData(),
                                               globdat );
  Ref<Constraints>  cons = Constraints::get ( dofs, globdat );
}
```

The **get** function of the **XDofSpace** class requires an **ItemSet** with which the **XDofSpace** is to be associated.

This **get** function has no context string because it creates a new **XDofSpace** when it is called for the first time. The new **XDofSpace** is stored in the database so that it will be found the next time that **get** is called.

The **get** function of the **Constraints** class works in a similar way.

Classes that provide a **`get`** function also provide a (non-static) **`store`** member function that stores a class instance in the database.

Here is an example:

```
void example ( const Properties& globdat )
{
  XNodeSet     nodes = newXNodeSet    ();
  XElementSet  elems = newXElementSet ( nodes );

  // Store the mesh in the global database:

  nodes.store ( globdat );
  elems.store ( globdat );
}
```

Accessing the state vector

The global database stores both the solution vector, called the ***state vector***, and the previous version of the state vector.

That is, at the start of each time or load step a copy of the current state vector is stored in the “old” state vector.

In a time-dependent application, the global database also stores the first-order and, possibly, the second-order time derivative of the state vector.

The state vectors can be retrieved from the database by means of the **`StateVector`** class from the **`model`** package.

Here is an example:

```
#include <jive/model/StateVector.h>

void example ( Ref<DofSpace>      dofs,
               const Properties&  globdat )
{
  using jive::model::STATE1;
  using jive::model::StateVector;

  Vector  u, u0, v;

  StateVector::get    ( u,  dofs, globdat );
  StateVector::getOld ( u0, dofs, globdat );
  StateVector::get    ( v, STATE1, dofs, globdat );

  u0 = u;
  u  = 1.0;
  v  = u - u0;
}
```

The **get** functions of the **StateVector** class return shallow copies of the vectors stored in **globdat**.

That means that after the statement

```
StateVector::get ( u, dofs, globdat );
```

the local variable **u** will point to the same data block as the state vector in **globdat**.

Thus, any modifications to **u** will also affect the state vector in **globdat**.

The constants **`STATE1`** and **`STATE2`** can be used to retrieve the first-order and second-order time derivatives of the state vector.

The **`get`** function must be called with a **`DofSpace`** with which the state vector is associated. It is possible to have multiple **`DofSpace`** instances, each with its own set of state vectors.

The state vectors are automatically resized whenever the number of DOFs changes.

Runtime variables

The global database contains a sub-property set named **`var`** for storing ***runtime variables*** such as the current time/load step number and the simulation time.

Any model or module can store floating point or integer numbers in the **`var`** section of the database. These numbers can be used in user-defined expressions that are specified in the runtime parameters **`props`**.

For instance, the **LoadScale** can be configured with a user-defined expression for scaling the external vector.

This section from an example input file illustrates how this works:

```
load =
{
  type      = "LoadScale";
  scaleFunc = "0.5 + 2.0 * i";
  model     = { ... };
};
```

The **scaleFunc** parameter is a user-defined expression that uses the current load/time step **i**. This is the name of a runtime variable stored in the **var** section of the global database.

Here is a more elaborate example. Suppose that we want to stop a simulation if the maximum value in the state vector exceeds a specific value.

We can achieve this by implementing a model that stores the maximum value in the `var` section of the global database.

Alternatively, an existing model could be modified to store the maximum value in the database.

Here is what the **takeAction** function would look like:

```
bool ExampleModel::takeAction ( const String&  action,
                                const Properties&  params,
                                const Properties&  globdat )
{
  if ( action == Actions::COMMIT )
  {
    Vector  u;

    StateVector::get ( u, dofs_, globdat );

    globdat.set ( "var.maxval", max( u ) );

    return true;
  }

  return false;
}
```

Here it is assumed that the **ExampleModel** class has a private member **dofs_** of type **Ref<DofSpace>**.

The variable named **maxval** can now be used to configure the **ControlModule**.

Here is the relevant section from the input file:

```
control =
{
  runWhile = "maxval < 1.5";
};
```

You could also display the **maxval** variable in a graph by using the **GraphModule**, or save it to a file by using the **SampleModule**.

Bonus exercise: use runtime variables

Implement a model that stores a runtime variable named **`v`** in the global database. A different value should be stored in each iteration.

Display the runtime variable in a graph using the **`GraphModule`** and save it to a file with the **`SampleModule`**.

Exercise location: **`exercises/runvar`**.

Solution location: **`solutions/runvar`**.

Details

The prepared code for the exercise contains a custom module named **`StepModule`** that sends the **`COMMIT`** action to the model.

The input file **`input.pro`** already contains sections for the **`GraphModule`** and the **`SampleModule`**. You must still add those modules to the main module chain.

Check the contents of the output file **`runvar.out`**.

Configuration data

Previous examples and exercises have already illustrated how modules and models can be configured with the runtime parameters stored in the **`props`** object.

As explained before, Jive reads the runtime parameters from a properties file and stores them in the **`props`** object.

A module or model can then read its runtime parameters from the **`props`** object.

Because each model/module is free to make up its own parameter names, it is probable that different models and modules use the same names.

To avoid name conflicts, the runtime parameters are divided into sub-property sets, one for each module/model.

Each model/module is given a name that is essentially the name of the sub-property set containing its runtime parameters.

In other words, a model/module can use its name to extract the relevant sub-property set from the `props` object.

This is illustrated in the next slides.

Example input file:

```
model =
{
  type  = "Matrix";
  model =
  {
    type   = "Multi";
    models = [ "first", "second" ];
    first  =
    {
      type  = "Example";
      alpha = 1.0;
    };
    second =
    {
      type  = "Example";
      alpha = 2.0;
    };
  };
};
```

Implementation of **ExampleModel** constructor:

```
ExampleModel::ExampleModel ( const String&       name,
                             const Properties&   conf,
                             const Properties&   props,
                             const Properties&   globdat ) :

  Model ( name )

{
  Properties  myProps = propsgetProps ( myName_ );

  System::out() << "my name  : "  << myName_ << "\n";
  System::out() << "my props :\n" << myProps << "\n\n";
}
```

Note that **myName_** is a protected member of the **Model** class. It contains a copy of the **name** parameter.

Output:

```
my name  : model.model.first
my props :
type = "Example";
alpha = 1.0;

my name  : model.model.second
my props :
type = "Example";
alpha = 2.0;
```

Note that Jive does not care which properties are defined in the input file; it simply stores all properties in the **`props`** object.

This means that you can define whatever properties you need and retrieve their values in your own model.

When you add a module to your main module chain you may specify a name for that module. This name indicates where the runtime parameters of the module are located.

For instance, consider this module chain:

```
Ref<Module> mainChain ()
{
  Ref<ChainModule>  chain = newInstance<ChainModule> ();

  chain->pushBack ( newInstance<ControlModule> ( "ctrl" ) );
  chain->pushBack ( newInstance<OutputModule>  ( "out1" ) );
  chain->pushBack ( newInstance<OutputModule>  ( "out2" ) );

  return chain;
}
```

The input file might look like this:

```
ctrl =
{
  runWhile = "i < 100";
};

out1 =
{
  file    = "disp.out";
  vectors = [ "state" ];
};

out2 =
{
  file   = "stress.out";
  tables = [ "stress" ];
};
```

As illustrated, this makes it possible to add multiple modules of the same type and configure them in different ways.

Configuration output

The constructor of the **`ExampleModel`** class not only has a **`props`** parameter, but also a **`conf`** parameter.

This parameter is a kind of mirror of the **`props`** parameter and is meant to get a complete list of all runtime parameters that are used in a computation.

The contents of **`conf`** are printed by the **`ReportModule`** after all modules and models have been initialised.

The **conf** object starts out empty and is to be filled by all modules and models as they are initialised.

This is how it works:

```
ExampleModel::ExampleModel ( const String&       name,
                             const Properties&   conf,
                             const Properties&   props,
                             const Properties&   globdat ) :


  Model ( name )

{
  Properties  myProps = props.getProps  ( myName_ );
  Properties  myConf  = props.makeProps ( myName_ );

  double      a = 0.0;

  myProps.find ( a, "alpha" );
  myConf .set  ( "alpha", a );
}
```

By using the member function **`makeProps`** a new property set is created in **`conf`** with the same name as the corresponding property set in **`props`**.

The **`conf`** object typically ends up with more properties than are stored in **`props`** because it also stores the properties with default values.

For instance, the parameter **`alpha`** in the previous example may not be present in **`props`** but is will be stored in **`conf`**.

In this way the contents of **`conf`** tell you exactly which parameters have been used in a simulation.

Example

The **`elastic`** example program can be made more compact and more versatile by implementing it with modules and models.

This is explained in the next slides.

The new implementation essentially consists of a couple of header files containing common declarations, a new model class named **`ElasticModel`**, a module construction function, and two input files.

The table below lists all files making up the new implementation.

File	Purpose
`import.h`	Contains common using declarations.
`Array.h`	Contains using declarations for `Array` types.
`declare.h`	Declares a function for registering the `ElasticModel`.
`ElasticModel.cpp`	Contains the definition and implementation of the `ElasticModel` class.

File	Purpose
`main.cpp`	Contains the `main` function and the module construction function.
`example.pro`	Contains the runtime parameters.
`example.dat`	Contains a descriptions of the mesh and boundary conditions.

The header files `import.h` and `Array.h` are not essential but they will be useful when extending the program later.

import.h

```
#ifndef IMPORT_H
#define IMPORT_H

#include <jem/defines.h>

namespace jem
{
  class String;
  class Object;

  template <class T> class Ref;

  namespace util
  {
    class Properties;
  }
}

using jem::idx_t;
using jem::Ref;
using jem::String;
using jem::Object;
using jem::util::Properties;

#endif
```

Array.h

```
#ifndef ARRAY_H
#define ARRAY_H

#include <jive/Array.h>

using jive::Vector;
using jive::IdxVector;
using jive::Matrix;
using jive::Cubix;

#endif
```

With these two header files one does not have to repeat common using declarations in different source files; simply include one of these (or both) header files.

The header file **declare.h** declares a function that registers the **ElasticModel** so that it can be instantiated by the **InitModule**.

declare.h

```
#ifndef DECLARE_H
#define DECLARE_H

void declareElasticModel ();

#endif
```

By putting the **declareElasticModel** function in a separate header file the implementation of the class **ElasticModel** can be “hidden” in a source file.

Moreover, this set up simplifies adding more models later.

The `ElasticModel` class

The source file `ElasticModel.cpp` contains both the definition and implementation of the `ElasticModel` class.

This class is derived from the `Model` class and essentially implements the action `GET_MATRIX0`. That is, this class handles the assembly of the global stiffness matrix and the internal (force) vector.

The assembly of the external (right-hand side) vector is handled by other model classes.

Definition of the **ElasticModel** class:

```
class ElasticModel : public Model
{
 public:

                          ElasticModel

    ( const String&          name,
      const Properties&      conf,
      const Properties&      props,
      const Properties&      globdat );

  virtual bool              takeAction

    ( const String&          action,
      const Properties&      params,
      const Properties&      globdat );

  // Continued on the next slide ...
```

Definition of the **ElasticModel** class (continued):

```
  // ... continued from the previous slide.

private:

 void                        assemble_

    ( MatrixBuilder&           mbld,
      const Vector&            fint,
      const Vector&            disp );


private:

 Assignable<NodeSet>        nodes_;
 Assignable<ElementSet>     elems_;
 Ref<XDofSpace>             dofs_;
 Ref<IShape>                shape_;
 idx_t                      jtype_;   // DOF type index.

 double                     young_;
 double                     area_;

};
```

The private members of the `ElasticModel` class store all data that is needed to assemble the global stiffness matrix and the internal vector.

The `nodes_` and `elems_` members provide access to the finite element mesh.

The `shape_` member refers to an `InternalShape` that calculates the shape function gradients and the integration point weights.

The **`dofs_`** member refers to the **`Dof_Space`** that is used to map the degrees of freedom to the global system of equations.

The member **`jtype_`** refers to the DOF type representing the displacement field.

The members **`young_`** and **`area_`** store the relevant material parameters that are used to compute the element matrices.

The constructor uses the parameters **`props`** and **`globdat`** to initialise these members.

The **takeAction** function looks like this:

```
bool ElasticModel::takeAction

  ( const String&      action,
    const Properties&  params,
    const Properties&  globdat )

{
  if ( action == Actions::GET_MATRIX0 )
  {
    Ref<MatrixBuilder>  mbld;
    Vector              fint;
    Vector              disp;

    params      .get ( mbld, ActionParams::MATRIX0 );
    params      .get ( fint, ActionParams::INT_VECTOR );
    StateVector::get ( disp, dofs_, globdat );

    assemble_ ( *mbld, fint, disp );

    return true;
  }

  return false;
}
```

The private member function `assemble_` assembles the global stiffness matrix by adding the element stiffness matrix to the `MatrixBuilder`. It also adds the element vectors to the internal force vector `fint`.

Note that, because of the nature of the model, the internal force vector for an element i can be obtained by multiplying the element stiffness matrix with the displacements in the nodes of the element:

$$f_{\text{int},i} = K_i\, a_i = K_i\, P_i\, a$$

where a_i contains the displacements in the nodes of the element and a is the state vector.

Main program

The source file **`main.cpp`** contains both the **`main`** function and the main module construction function.

The latter creates a **`ChainModule`** containing six modules: **`InputModule`**, **`InitModule`**, **`InfoModule`**, **`LinsolveModule`**, **`OutputModule`** and **`ControlModule`**.

The **`InfoModule`** is not essential, but it enables one to monitor the progress of the program.

The main module construction function must register the **`ElasticModel`** and other classes that are used in the program.

This is what the function looks like:

```
Ref<Module> mainModule ()
{
  Ref<ChainModule>  chain = newInstance<ChainModule> ();

  jive::fem ::declareMBuilders ();
  jive::model::declareModels   ();
  declareElasticModel          ();

  chain->pushBack ( ... );

  return chain;
}
```

The call to the function **`declareMBuilders`** registers the **`FEMatrixBuilder`** class that is used to assemble the global matrix.

Properties file

This covers the source code of the program.

To run the program, two input files are required: one containing the runtime parameters, and one containing the description of the mesh and the boundary conditions.

The input file `example.pro`, called the ***properties file***, contains the runtime parameters for the modules and models.

By convention, it has the extension `.pro` but this is not required.

example.pro (part 1)

```
log =
{
  pattern = "*.info | *.debug";  // Filters the standard output.
  file    = "-$(CASE_NAME).log"; // Name of the log file.
};

input =
{
  file = "$(CASE_NAME).dat";     // Name of the data file.
};

control =
{
  runWhile = "i < 10";           // When to stop.
};

linsolve =
{
  solver = "SparseLU" {};        // Linear solver to be used.
};

output =
{
  file    = "$(CASE_NAME).out";  // Name of the output file.
  vectors = [ "state = disp" ];  // Which data to be saved.
  saveWhen = true;               // When to save data.
};
```

The sub-property set named **`log`** is used by the **`Application`** class to configure the standard output streams.

Many classes from Jive write messages with different priorities to the standard output stream to indicate what operations are being performed.

By default, only high-priority messages are printed to the terminal. The **`log.pattern`** property can be used to enable low-priority messages.

The current setting enables informational and debug messages.

Here are some alternative settings for the **`pattern`** property:

```
log =
{
  // Enable all informational messages:
  pattern = "*.info";

  // Only enable informational messages from the solver:
  pattern = "linsolve.solver.info";

  // Enable all informational messages, except from the solver:
  pattern = "*.info | ! linsolve.solver.info";
};
```

The property **`log.file`** specifies the file to which the messages are to be saved. If the file name starts with a dash (**–**) then the messages are also printed to the terminal.

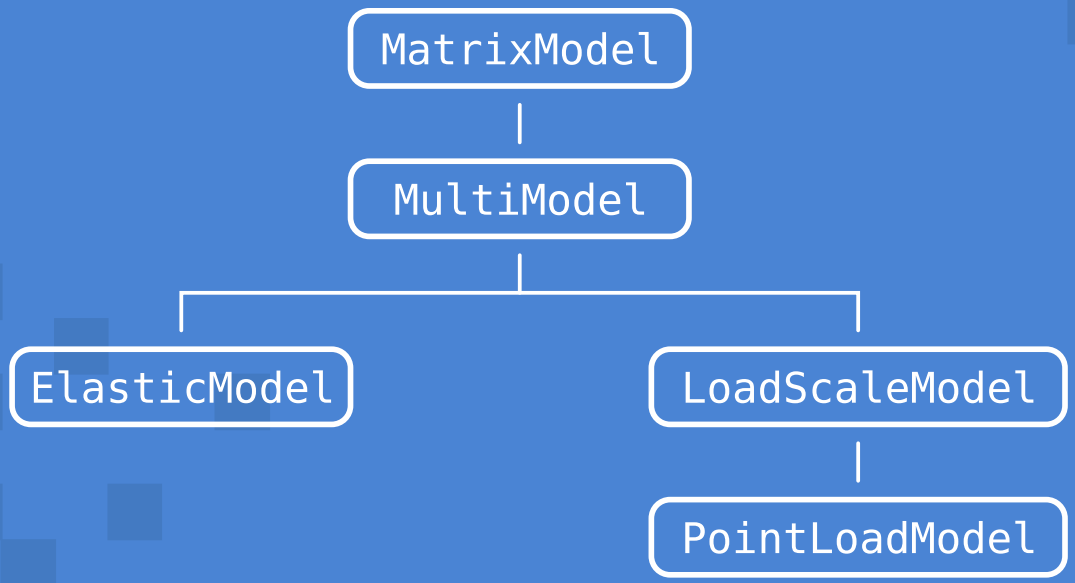
The special word **$(CASE_NAME)** is replaced by the base name of the properties file; **example** in this case.

This notation enables one to copy a properties file to a new name without having to modify all file names.

The property **output.vectors** specifies which vectors in the global database should be saved by the **OutputModule**.

In this case only the state vector is saved. It is assigned the label **disp** in the output file. If the label specification is omitted, the label **state** will be used by default.

The second part of the properties file (see next slide) defines the following model tree:



example.pro (part 2)

```
model = "Matrix"
{
  matrix =
  {
    type = "FEM";  // Use the FEMatrixBuilder.
  };
  model  = "Multi"
  {
    models  = [ "elastic", "load" ];
    elastic = "Elastic"
    {
      young = 0.0001;
      area  = 200e9;
    };
    load    = "LoadScale"
    {
      scaleFunc = "1 + i";
      model     = "PointLoad"
      {
        loadTable = "load";  // Refers to the table "load" in
                             // the data file.
      };
    };
  };
};
```

The **`MatrixModel`** creates and manages a **`MatrixBuilder`** that is used to assemble the global matrix.

The properties file specifies that it should create a **`FEMatrixBuilder`** that takes advantage of the specific structure of the global matrix in finite element applications.

The **`ElasticModel`** takes care of assembling the global matrix and the internal vector. This model is specific to the **`elastic`** program and has been explained earlier.

The **`LoadScaleModel`** and the **`PointLoadModel`** handle the force that is applied at the right end of the bar.

The first model determines a scale factor, specified by the **`scaleFunc`** property, that is applied to the force on the bar.

The **`PointLoadModel`** adds the scaled force to the external factor. It essentially implements the action **`GETEXT_VECTOR`**.

This model reads the force to be applied from a **`Table`** named **`load`** that is specified in the data file **`example.dat`**; see below.

Note the special syntax to specify the type of a model:

```
elastic = "Elastic"
{
  :
};
```

This is equivalent with:

```
elastic =
{
  type = "Elastic";
   :
};
```

In general, a string specified between the = and { tokens is translated into property named **`type`** that is part of the sub-property set. This is called a ***typed property set***.

Data file

The other input file, **`example.dat`**, called the ***data file***, contains a definition of the mesh and the boundary conditions.

This is a separate file because it can be large and because it could be generated by another program.

The data file is based on the XML syntax. This means that it is composed of multiple sections started by a ***begin tag*** and ended by an ***end tag***.

A begin tag has the following form:

<*tag-name*>

The corresponding end tag looks like this:

</*tag-name*>

A begin tag may have one or more attributes like this:

<*tag-name attrib-name*="*value*">

Multiple attributes are separated by one ore more spaces.

Comments are delimited by the tokens **<!--** and **-->**.

The next slide show the first part of the data file.

example.dat (part 1)

```
<Nodes>
  1 0.0;
  2 0.2;
  3 0.4;
  4 0.6;
  5 0.8;
  6 1.0;
</Nodes>

<Elements>
  1 1 2;
  2 2 3;
  3 3 4;
  4 4 5;
  5 5 6;
</Elements>
```

The sections **Nodes** and **Elements** define the nodes and the elements that make up the mesh.

Each row in the **`Nodes`** section defines one node. It starts with the identifier of the node, followed by its coordinates and a terminating semi-colon.

You can specify any number of coordinates per node, as long as all nodes have the same number of coordinates.

The number of coordinates determines the rank of the corresponding **`NodeSet`**.

A data file may have multiple **`Nodes`** sections. All nodes are stored in a single **`NodeSet`**.

Each row in the **`Elements`** section defines one element. It starts with the identifier of the element, followed by the identifiers of the nodes and a terminating semi-colon.

An element may have an arbitrary number of nodes; the number of nodes may be different for different elements.

A data file may have multiple **`Elements`** sections. All elements are stored in a single **`ElementSet`**.

`example.dat` (part 2)

```
<NodeGroup name = "left">
  { 1 }
</NodeGroup>

<NodeGroup name = "right">
  { 6 }
</NodeGroup>

<NodeConstraints>
  u[left] = 0.0;
</NodeConstraints>

<NodeTable name = "load">
  <Section columns = "u">
    right 1.0;
  </Section>
</NodeTable>
```

The second part of the data file specifies two node groups that are used to define the boundary conditions.

By using node groups instead of node IDs, you do not have to modify the definition of the boundary conditions when you refine the mesh.

The **`Constraints`** section defines the constraints. In this case the displacement in the left edge of the bar is set to zero. To be precise, the degree of freedom type **`u`** is set to zero in all nodes in the group named **`left`**.

Note that the degree of freedom type specified in the **`Constraints`** section must match the type declared by the **`ElasticModel`**:

```
dofs_->addType ( "u" );
```

The **`NodeTable`** section defines a table named **`load`** that is associated with the nodes in the mesh.

A table definition consists of multiple sections, each of which defines one or more columns. In this case there is one column named **`u`** that contains the value 1 for the nodes in the group **`right`**.

The contents of the table are added to the external vector (right-hand vector) by the **`PointLoadModel`**.

This means that the value in the table, multiplied with the scale factor determined by the **`LoadScaleModel`**, is added to the entry in the external vector that corresponds with the node at the right end of the bar.

Note that the table column name **`u`** must match the DOF type declared by the **`ElasticModel`**.

The table name **`load`** must match the value of the property **`loadTable`** of the **`PointLoadModel`**.

Exercise: finish the implementation

Finish the implementation of the new **`elastic`** program. In particular, you must complete the implementation of the main module function and the **`ElasticModel`** class.

Comments in the code indicate what you have to do.

Exercise location: **`exercises/elastic-2`**.

Solution location: **`solutions/elastic-2a`**.

Exercise: display displacement

Extend the **`elastic`** program with the **`GraphModule`** and display the largest displacement as function of the load step number.

Exercise location: **`exercises/elastic-2`**.

Solution location: **`solutions/elastic-2b`**.

Hints

Store the maximum displacement in the sub-property set **var** of the global database when handling the action **COMMIT**.

Add the **GraphModule** from the **gl** package to the main module chain and extend the properties file with the required runtime parameters.

To get a better view of the graph, extend the sub-property set for the **ControlModule** with:

```
fgMode = true;
```

This causes the **ControlModule** to wait after the last load step.

Bonus exercise: apply a traction

Implement a **`TractionModel`** that applies a (constant) traction along the bar. This model should implement the action **`GET_EXT_VECTOR`**.

Hint: copy the **`ElasticModel`** and start from there.

Exercise location: **`exercises/elastic-2`**.

Solution location: **`solutions/elastic-2c`**.

Built-in Jive modules

The built-in Jive modules define a (large) collection of runtime parameters or ***properties***, some of which have been explained earlier, and many of which are explained in the next slides.

A complete list of all properties is printed by the `ReportModule`. You are therefore advised to wrap your main module in a `ReportModule`.

This has the added benefit of being warned about properties in the input file that are not used in your program; maybe because of a typo in the property name.

The properties of a module should be specified in a property set that has the name of the module.

In all examples shown in the next slides the name of the enclosing property set is equal to the default name of a module.

For instance, the properties of the **`ControlModule`** are specified in a property set named **`control`** as this is the default name of the module.

Here is an example:

```
control =
{
  fgMode    = true;
  runWhile = "i < 10";
};
```

If one would assign a different name to the **`ControlModule`**, for instance **`ctrl`**, then the properties should be specified like this:

```
ctrl =
{
  fgMode    = true;
  runWhile = "i < 10";
};
```

The `ChainModule`

The **`ChainModule`** (package **`app`**) encapsulates a list of modules. It can be used in places where a single module is expected but multiple modules are desired.

Modules can be appended to the list by calling the **`pushBack`** member function.

The **`init`**, **`run`** and **`shutdown`** member functions of the **`ChainModule`** call the corresponding member functions of all modules in its list.

Implementation of the **run** member function:

```
Status ChainModule::run ( const Properties& globdat )
{
  Status  result = OK;

  for ( idx_t i = 0; i < list_.size(); i++ )
  {
    Status  stat = list_[i]->run ( globdat );

    if ( stat == EXIT )
    {
      result = EXIT;
    }
  }

  return result;
}
```

The variable **list_** stores the list of modules.

The **ChainModule** has no properties.

The `ReportModule`

The **`ReportModule`** (package **`app`**) is to be “wrapped” around another module. This is typically the main module.

The **`ReportModule`** keeps track of all properties that have been used by its child modules, and prints an overview of all properties at the end of its **`init`** member function.

To be precise, the **`init`** member function prints the contents of the **`conf`** property set after all modules (and models) have been initialised.

Typical use of the **ReportModule**:

```
Ref<Module> mainModule ()
{
  Ref<ChainModule>  chain = newInstance<ChainModule> ();

    :

  // Wrap the main module chain in a ReportModule:

  return newInstance<ReportModule> ( "report", chain );
}
```

Note that the **ReportModule** must be given a name. That name is currently not relevant as the **ReportModule** has no properties.

The `InputModule`

The **`InputModule`** (package fem) can be used to read a mesh and other data from an XML-formatted data file.

The contents of the data file are stored in different objects in the global database so that they are available to other modules and models.

The **`InputModule`** therefore should be one of the first modules in the main module chain.

The **InputModule** can be configured through the following properties:

file
: A string that specifies the name of the data file.

storageMode
: A string that controls the amount of memory required to store the nodes and the elements. It can have three values: **Default**, **Compact** and **Minimal**.

`storageMode` (cont)

The **`Default`** mode requires most memory and uses the fastest data structures to map node/element IDs to indices.

The **`Compact`** mode requires less memory and uses slower data structures to map IDs to indices.

The **`Minimal`** mode requires the least amount of memory but only supports regularly numbered IDs. This mode should only be used when the memory use of the program is critical.

Example properties for the **InputModule**:

```
input =
{
  file        = "$(CASE_NAME).dat";
  storageMode = "Compact";
};
```

The special symbol **$(CASE_NAME)** within the file name is replaced by the base name of the properties file. In fact, this works for any string containing this symbol.

In general, a symbol of the form **$(***name***)** within a string will be replaced by the contents of the environment variable *name*.

Data file format

The data file is composed of a number of sections that start with a ***begin tag*** of the form

<*tag-name*>

and end with a corresponding ***end tag*** of the form

</*tag-name*>

where *tag-name* denotes the name of the tag. This name essentially specifies the type of the section.

The special tags **<!--** and **-->** delimit comments and are ignored by the **InputModule**.

A begin tag may have one or more ***attributes*** of the form:

name = "*value*"

Multiple attributes are separated by spaces.

Here is an example:

```
<NodeGroup name = "left">
```

An attribute can be optional or required. In the latter case the attribute must be specified or an error is raised.

Note that the value of an attribute must always be specified within double quotes.

Example data file:

```
<Nodes>
  1 0.0 0.0;
  2 1.0 0.0;
  3 0.0 1.0;
</Nodes>

<Elements>
  1 1 2 3;
</Elements>

<NodeGroup name = "left">
  { 1, 3 }
</NodeGroup>

<ElementGroup name = "triangles">
  { 1 }
</ElementGroup>

<NodeConstraints>
  u[1] = 0.0;
  u[3] = u[1];
</NodeConstraints>

<Include source = "tables.dat"/>
```

The most common tags recognised by the **InputModule** are described below:

Nodes

Defines the nodes. Each line in this section consists of the node ID, followed by its coordinates and a terminating semi-colon.

Any number of coordinates can be specified as long as each node has the same number of coordinates.

Each node must have a unique ID (arbitrary integer).

The nodes are stored in a **NodeSet**.

`Elements`

Defines the elements. Each line in this section consists of the element ID, followed by the IDs of the element nodes and a terminating semi-colon.

An element may have an arbitrary number of nodes and the nodes per element may vary.

Each element must have a unique ID.

The elements are stored in an **`ElementSet`**.

`NodeGroup`

Defines a node group. The name of the node group must be specified with the required **`name`** attribute.

A node group is represented by a **`NodeGroup`** instance.

`NodeGroup` (cont)

A node group can be specified as a set of node IDs within curly braces. The notation $[i : j]$ can be used to specify a range of IDs, starting at i and ending at j; the number j is ***not*** included. Optionally, a skip factor can be specified: $[i : j; k]$.

The operators `|` and `&` can be used to combine two or more groups into a new group. The first yields the union of two groups, and the second yields the intersection. Parentheses can be used to group the operators.

The operands of these operators can be sets of IDs between curly braces, or names of previously defined groups.

Examples of **NodeGroup** sections:

```
<NodeGroup name = "first">
  { 1, 2, [4:10] }                    <!-- 1, 2, 4, 5, 6, 7, 8, 9 -->
</NodeGroup>

<NodeGroup name = "second">
  { [9:12], 20 }                      <!-- 9, 10, 11, 20 -->
</NodeGroup>

<NodeGroup name = "third">
  first & second                      <!-- 9 -->
</NodeGroup>

<NodeGroup name = "fourth">
  (first | second) & { 3, 5, 20 }     <!-- 5, 20 -->
</NodeGroup>
```

The node group named **third** contains only the node with ID 9; the group **fourth** contains the nodes with IDs 5 and 20.

ElementGroup

Defines an element group that is stored as an **ElmentGroup** instance in the global database. The syntax follows that of the **NodeGroup** section.

NodeConstraints

Defines the constraints (essential boundary conditions). They are stored in a temporary object that is converted by the **InitModule** into a **Constraints** object later.

Note that if the DOFs are not associated with the nodes but with the elements, then this section is named **ElementConstraints**.

NodeConstraints (cont)

A simple constraint (or support) is specified as follows:

dof-type`[`*node-id*`] = `*value*`;`

where *dof-type* is a DOF type name; *node-id* is a node ID; and *value* is a floating point value.

Instead of a single node ID one may also specify a node group between the square brackets like this:

```
u[left] = 1.0;
```

This sets the DOF type **u** to 1.0 in all nodes in the group named **left**.

`NodeConstraints` (cont)

A linear constraint involves a linear combination of DOFs to the right of the assignment token. Here are some examples:

```
u[1]    = u[2];
u[4]    = 1.0 + 0.5 * u[5] + 0.5 * u[6];
u[left] = u[right];
```

When using group names between the square brackets, all groups used in a single constraint specification must contain the same number of nodes.

Using group names instead of IDs is like specifying multiple constraints, one for each node in the group(s).

NodeTable

Defines a table associated with the nodes. That is, each row in the table is associated with one node. The name of the table must be specified with the required **name** attribute.

The optional **type** attribute indicates whether the table is mostly filled or empty and controls how the table is stored in memory. It can have the values **Dense** or **Sparse** (the default).

The data specified in a **NodeTable** section are stored in a **Table** instance.

NodeTable (cont)

A **NodeTable** section must contain one or more **Section** sections that define rectangular parts of the table. The required **columns** attribute specifies which table columns are affected by a section. It should contain the names of the columns separated by a | token.

Here is an example:

```
<Section columns = "Fx | Fy">
```

This starts a table section specifying data for the columns **Fx** and **Fy**.

`NodeTable` (cont)

Each line in a table section should like this:

node-id *value-1* . . . *value-n* ;

where *node-id* is the ID of a node, and *value-1* to *value-n* are the table values for that node and for the table columns specified by the `columns` attribute.

Instead of specifying a single node ID, you can also specify the name of a node group. This will copy the specified values to all table rows associated with the nodes in that group.

Example of a **NodeTable** section:

```
<NodeTable name = "load" type = "Dense">
  <Section columns = "Fx | Fy">
    1     2.0  0.0 ;
    2     2.0  2.0 ;
  </Section>
  <Section columns = "Fx">
    left  -1.0 ;
  </Section>
  <Section columns = "Fy">
    left   1.0 ;
  </Section>
</NodeTable>
```

Assuming that the node group **`left`** contains the node IDs 3 and 4, then the previous section in the data file defines the following table:

`load`

Node ID	**`Fx`**	**`Fy`**
1	2.0	0.0
2	2.0	2.0
3	-1.0	1.0
4	-1.0	1.0

Two more sections that can be specified in the data file:

ElementTable

Defines a table associated with the elements. The syntax is the same as that of the **NodeTable** section.

Include

Includes another file into the current data file. The required attribute **source** must specify the name of the file to be included.

When the **InputModule** encounters an **Include** tag, it stops reading the current file and continues with the included file. When that file heas been read, the **InputModule** continues after the **Include** tag.

Include (cont)

Data files may be included recursively. That is, one include file may include another one.

The **Include** tag specifies a so-called empty section. This means that instead of specifying a corresponding end tag, one can finish the begin tag with a slash:

```
<Include source = "mesh.dat"/>
```

Note that when the file name of a data file ends with the extension **.gz** then that data file is assumed to have been compressed with GNU zip.

The `InitModule`

The **`InitModule`** (package **`app`**) creates the model tree, the constraints and, optionally, initialises the state vector and/or its time derivatives.

It uses the typed property set named **`model`** in the properties file to construct the model tree.

After that it takes the temporary constraints object (created by the **`InputModule`** and converts it to a proper **`Constraints`** object.

The **Constraints** can not be constructed by the **InputModule** because this requires a **DofSpace**.

As the **DofSpace** is created during the construction of the model tree, the **Constraints** can only be created after the model tree has been constructed.

By default, the state vector (and its time derivatives) are set to zero when a Jive program starts. You can specify a different initial state vector through the **vectors** property.

Note that this must be specified in the property set named **init** or whatever name has been assigned to the **InitModule**.

How to specify the initial state vectors:

```
init =
{
  vectors = [ "state  = 0.0",
              "state1 = veloc0",
              "state2 = accel0" ];
};
```

This sets the state vector to zero; the first-order time derivative of the state to the values stored in the table named **veloc0**; and the second-order time derivative to the values stored in the table **accel0**.

Note that the **model** property should be defined ***outside*** the property set **init**.

The `OutputModule`

The **`OutputModule`** (package **`app`**) can be used to save the state vector (and its time derivatives) to an XML-formatted file.

It can also save other data associated with the nodes and/or elements, such as strains and stresses.

Each data set is saved as a **`NodeTable`** or **`ElementTable`**. This means that an output file can be used as an input file.

Example output file:

```
<NodeTable name = "disp" type = "Dense">
  <Section columns = "dx | dy">
    1  0.0  0.0;
    2  0.5  0.0;
    3  0.5  0.0;
    4  0.0  0.0;
  </Section>
</NodeTable>

<ElementTable name = "stress" type = "Dense">
  <Section columns = "sxx | syy | sxy">
    1 10.0 0.0 0.0;
  </Section>
</ElementTable>
```

Example properties for the **OutputModule**:

```
output =
{
  file       = "$(CASE_NAME).out.gz";
  append     = false;
  precision  = 8;
  saveWhen   = "(i % 10) < 0.5";
  vectors    = [ "state = disp" ];
  tables     = [ "elements/stress" ];
};
```

The **`OutputModule`** can be configured through the following properties:

`file`
: A string that specifies the name of the output file. If the name ends with the extension **`.gz`** then the file will be compressed according to the GNU zip format.

`append`
: A boolean value that indicates whether the data to be saved should be appended to an existing data file. The default value is **`false`**.

`precision`

An integer that specifies the number of digits that should be printed for each floating point value in the output file. The default value is 6.

`saveWhen`

An expression (string) that specifies when output should be saved. If the expression yields a non-zero value, then the data are saved to the output file.

saveWhen (cont)

The expression is evaluated in each time/load step when the **run** member function of the **OutputModule** is called. The expression may involve any runtime variable stored in the **var** property set in the global database.

vectors

A string array that specifies which vectors stored in the global database are to be saved to the output file.

Each entry in this array should contain the name of the vector to be saved, optionally followed by an = token and a label. The label will be used as the name of the table in the output file.

vectors (cont)

Here is an example:

```
vectors = [ "state  = disp",
            "state1 = veloc",
            "state2 = accel" ];
```

This indicates that the state vector, and its first and second-order time derivatives should be saved to tables named **disp**, **veloc** and **accel**, respectively.

If the label contains the character sequence **%i**, then that sequence will be replaced by the current time or load step number.

`tables` (cont)

A string array that specifies which tables are to be saved to the output file.

Each entry in this array should start with the name of the item set that is associated with the table. This is typically **`nodes`** or **`elements`**.

After the name of the item set there should be a slash (`/`), followed by the name of the table, optionally followed by an = token and a label.

tables (cont)

Here is an example:

```
tables = [ "nodes/flux = heat-flux" ];
```

This indicates that the node table named **flux** is to be saved to a table named **heat-flux** in the output file.

Note that the label of a table may also contain the special character sequence **%i**.

Saving tables

When the **OutputModule** saves a table to the output file, it first checks whether that table is stored in the global database.

If that is not the case, then the **OutputModule** will create a new **XTable** object and request the model to fill that table through the **GET_TABLE** action.

A warning is printed if the model does not fulfill the request.

To show how this works, the following slides explain how to extend the **ElasticModel** from the **elastic** example program with support for the **GET_TABLE** action.

To be precise, the **ElasticModel** will be extended with support for returning a table containing the average stress in each node of the mesh.

This means that the following **tables** property can be specified for the **OutputModule**:

```
tables = [ "nodes/stress" ];
```

Step 1: extend the **ElasticModel** class definition with a private member function **getStress_** that computes the average stress in the nodes.

The new definition of the **ElasticModel** class:

```
class ElasticModel : public Model
{
 public:
                    ElasticModel   ( ... );
  virtual bool      takeAction     ( ... );

 private:
  void              assemble_      ( ... );
  void              getStress_     ( ... );

 private:
  NodeSet           nodes_;
  ElementSet        elems_;
  Ref<IShape>       shape_;
  Ref<XDofSpace>    dofs_;
  idx_t             jtype_;
  double            young_;
  double            area_;
};
```

The function parameters are not shown for the sake of brevity.

Step 2: extend the implementation of the `takeAction` function with an if-statement that handles the `GET_TABLE` action.

The body of the if-statement extracts the table to be filled from the `params` object. It also extracts the name of the table and the so-called ***table weights***.

The weights are used by the `OutputModule` to scale each row in the table before it is saved to the output file. That is, each row is divided by the corresponding entry in the weights vector.

The weights are set to zero by the `OutputModule` and are to be filled by the model handling the `GET_TABLE` action.

How to handle the **GET_TABLE** action:

```
if ( action == Actions::GET_TABLE )
{
  Ref<XTable>  table;
  String       name;

  params.get ( table, ActionParams::TABLE );
  params.get ( name,  ActionParams::TABLE_NAME );

  if ( (name == "stress") &&
       (table->getRowItems() == nodes_.getData()) )
  {
    Vector  weights, disp;

    params      .get ( weights, ActionParams::TABLE_WEIGHTS );
    StateVector::get ( disp,    dofs_, globdat );

    getStress_ ( *table, weights, disp );

    return true;
  }
}
```

The if-statement

```
if ( (name = "stress") &&
     (table->getRowItems() == nodes_.getData()) )
```

makes sure that the **GET_TABLE** action is handled only if the name of the table is **stress** and the table is associated with the nodes.

This is important because a model might be requested to fill a table it does not know about. In this case the model should not handle the **GET_TABLE** action.

Step 3: implement the private member function `getStress_`.

This function adds a column labelled `sxx` to the table and then executes a for-loop over all elements.

For each element it computes the gradients of the shape functions in the nodes of the elements. From this it computes the stresses in the nodes and adds those to the table.

The table weights are set equal to the number of elements connected to each node so that the average stresses will be saved to the output file.

Implementation of the function **getStress_** (part 1):

```
void ElasticModel::getStress_ ( XTable&         table,
                                const Vector&   weights,
                                const Vector&   disp )
{
  const idx_t  jcol      = table.addColumn   ( "sxx" );
  const idx_t  rank      = nodes_.rank       ();
  const idx_t  nodeCount = shape_->nodeCount ();

  Cubix        grads     ( rank, nodeCount, nodeCount );
  Matrix       coords    ( rank, nodeCount );
  IdxVector    inodes    ( nodeCount );
  IdxVector    idofs     ( nodeCount );
  Vector       eldisp    ( nodeCount );
  Vector       stress    ( nodeCount );

  for ( idx_t ielem = 0; ielem < elems_.size(); ielem++ )
  {
    elems_.getElemNodes  ( inodes, ielem );
    nodes_.getSomeCoords ( coords, inodes );
    dofs_->getDofIndices ( idofs,  inodes, jtype_ );

    // Continued on the next slide ...
```

Implementation of the function `getStress_` (part 2):

```
    // ... continued from the previous slide.

    // Get the shape function gradients in the nodes of the
    // element and compute the stresses in those nodes.

    shape_->getVertexGradients ( grads, coords );

    eldisp = disp[idofs];

    for ( idx_t i = 0; i < nodeCount; i++ )
    {
      stress[i] = young_ * dot ( grads(0,ALL,i), eldisp );
    }

    // Update the table and the table weights.

    table.addColValues ( inodes, jcol, stress );

    weights[inodes] += 1.0;
  }
}
```

The member function **`getVertexGradients`** (defined by the **`InternalShape`** class) returns the gradients of the shape functions in the nodes of an element.

It is similar to the function **`getShapeGradients`** except that it does not return the integration point weights.

Note that in a non-linear model it is generally not possible to calculate the stresses directly from the displacements. In that case one typically extrapolates the stresses from the integration points to the nodes.

The complete implementation of the extended **`ElasticModel`** is located in the directory **`solutions/elastic-2d`**.

The **ExitModule**

The **ExitModule** (package **app**) does only one thing: return the **EXIT** code from its **run** member function to terminate a program.

This module is useful if one only needs to execute the main module chain once, for instance in a simple linear computation.

In this case the **ExitModule** should be added as the last module to the main module chain.

The `ControlModule`

The **`ControlModule`** (package **`app`**) can be used to terminate a program.

In contrast to the **`ExitModule`**, however, one can specify a condition that controls when the program is terminated.

The **`ControlModule`** can also be used to run a program in an interactive way, allowing the user to enter commands through a terminal or a file.

Example properties for the **ControlModule**:

```
control =
{
  pause     = 0.2;
  runWhile  = "i < 10";
  fgMode    = true;
  cmdFile   = "";
};
```

The **`ControlModule`** can be configured through the following properties:

`pause`

a floating point number indicating how long the **`ControlModule`** should wait before continuing with the next time/load step. The default value is zero.

This property can be set to a non-zero value when one wants to display the results of a simulation in real time.

runWhile

An expression (string) that specifies whether the program should continue with the next time/load step. If the expression yields a zero value then the program is terminated, unless the **ControlModule** is running in foreground mode; see below.

The expression is evaluated in each time/load step when the **run** member function of the **ControlModule** is called. The expression may involve any runtime variable stored in the **var** property set in the global database.

`fgMode`

A boolean that indicates whether the **`ControlModule`** should run in ***foreground mode***. In this mode the **`ControlModule`** will wait for input instead of terminating the program. The default value is **`false`**.

Commands can be entered through the terminal or through a file when the **`cmdFile`** property has been set (see below). Type the command **`help`** for a list of available commands.

Note that the **`ControlModule`** can always be switched to the foreground mode by entering the **`fg`** command.

`cmdFile`

A string specifying the name of a command file. If this string is not empty, then the `ControlModule` will read commands written to the command file. The default value is an empty string.

This property is meant to control long-running computations without the need to have a controlling terminal.

The `LinsolveModule`

The `LinsolveModule` (package `implict`) can be used to solve a linear system of equations of the form

$$K\,u = g$$

where K is the global stiffness matrix and g the external vector.

The solution u is stored in the global database as the state vector.

The **LinsolveModule** has one property named **solver**. This should be a typed property set that selects and configures the linear solver to be used.

Here are some examples:

```
linsolve =
{
  solver =
  {
    type = "SparseLU";       // Use the SparseLU solver.
  };

  solver = "SparseLU" {};    // Same as above.

  solver = "GMRES"           // Use the iterative GMRES solver.
  {
    precision      = 1.0e-5;
    preconditioner = "ILUd" {};
  };
};
```

The **init** member function of the **LinsolveModule** uses the **NEW_MATRIX0** action to get an **AbstractMatrix** that represents the global stiffness matrix K.

This action is normally handled by the **MatrixModel**. You can also handle it in your own model if you need complete control over the way that the global matrix is managed.

The **init** member function also creates a **Solver** instance that is associated with the global matrix and the **Constraints** stored in the global database.

Solution procedure

The **`run`** member function of the **`LinsolveModule`** executes the following operations:

1. Copy the current state vector to the “old” state vector.
2. Advance to the next load/time step by calling the **`takeAction`** function of the model with the **`ADVANCE`** action.
3. Assemble the global stiffness matrix K by means of the **`GET_MATRIX0`** action.

4 Assemble the external vector g by means of the **GET_EXT_VECTOR** action.

5 Update the constraints by means of the **GET_CONSTRAINTS** action.

6 Solve the system of equations by calling the **solver** member function of the **Solver**.

7 Store the solution vector as the state vector in the global database.

8 Signal that a new state vector has been computed by means of the **COMMIT** action.

The `NonlinModule`

The **`NonlinModule`** (package **`implict`**) implements the Newton-Raphson algorithm and can be used to solve non-linear systems of equations of the form:

$$f(u) = g$$

where f denotes the internal vector, u denotes the state vector and g denotes the external vector.

The external vector may not depend on the state vector in any way. If it does, then it should be “folded” into the internal vector.

The **NonlinModule** tries to find the solution by constructing a sequence of state vectors $\boldsymbol{u}_i$ that are progressively closer to the exact solution (in a numerical sense).

This sequence of state vectors is constructed by means of the following recipe:

$$\begin{aligned} \boldsymbol{r}_i &= \boldsymbol{g} - \boldsymbol{f}(\boldsymbol{u}_i) \\ \delta\boldsymbol{u}_i &= \boldsymbol{K}^{-1}\,\boldsymbol{r}_i \\ \boldsymbol{u}_{i+1} &= \boldsymbol{u}_i + \delta\boldsymbol{u}_i \end{aligned}$$

The vector $\boldsymbol{r}_i$ is called the ***residual vector***, or residual in short. It is a measure of the accuracy of approximate solution $\boldsymbol{u}_i$.

The matrix $\boldsymbol{K}$ should be equal to the tangent stiffness matrix or an approximation thereof:

$$\boldsymbol{K} \approx \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}$$

The closer $\boldsymbol{K}$ is to the tangent matrix, the higher the convergence rate, in general.

Note that the Newton-Raphson algorithm is not guaranteed to find the solution of a non-linear system. Indeed, it is not difficult to come up with (academic) non-linear systems for which the Newton-Raphson algorithm gets “stuck” or diverges.

The **NonlinModule** considers the solution u_i to be sufficiently accurate if

$$\frac{||\boldsymbol{r}_i||}{s} < \epsilon$$

where s is the ***residual scale factor*** and ϵ is a user-defined precision (through the property **precision**).

The residual scale factor is based on the norm of the internal and external vectors in the first two iterations. In this way a sensible value is obtained when constraints are the driving factor in a simulation.

A model can implement its own convergence criterion. More about this later.

Example properties for the **NonlinModule**:

```
nonlin =
{
  precision    = 1.0e-3;
  maxIter      = 20;
  tiny         = 1.0e-12;
  solver       = "AGMRES"
  {
    precision = 1.0e-4;
  };
};
```

The **NonlinModule** can be configured through the following properties:

precision

A floating point number specifying the relative precision ϵ that is used to determine whether the solution is sufficiently accurate.

The default value is 10^{-3}.

Setting **precision** to a very small number may prevent the **NonlinModule** from reaching convergence due to floating point errors.

maxIter

An integer that specifies the maximum number of iterations. An error is raised if a sufficiently accurate solution is not found within this number of iterations.

The default value is 20.

tiny

A floating point number that specifies a lower limit for the residual scale factor.

tiny (cont)

If the residual scale factor is smaller than **tiny** then the solution is considered to be sufficiently accurate.

The default value is the smallest possible positive floating point value (typically in the order of 10^{-300}).

solver

A typed sub-property set that selects and configures the linear solver.

The **init** member function of the **NonlinModule** is similar to that of the **LinsolveModule**.

That is, it uses the **NEW_MATRIX0** action to get an **AbstractMatrix** that is associated with a **Solver** instance.

The **MatrixModel** can be used to handle this action if there is no need to create a special matrix type or to manage the assembly of the matrix in a non-standard way.

Solution procedure

The **run** member function executes the following procedure:

1. Copy the current state vector to the “old” state vector.
2. Advance to the next load/time step by calling the **takeAction** function of the model with the **ADVANCE** action.
3. Assemble the external vector g by means of the **GET_EXT_VECTOR** action.
4. Update the constraints by means of the **GET_CONSTRAINTS** action.

5 Assemble the initial tangent stiffness matrix K_0 and the internal vector f_0 by means of the **GET_MATRIX0** action.

6 Compute the initial residual and the initial solution increment:

$$r_0 = g - f_0$$
$$\delta u_0 = K_0^{-1} r_0$$

7 Compute the residual scale factor:

$$s = \max\left(||f_0||, ||r_0||, ||g||\right)$$

Note that a model can provide an alternative residual scale factor by implementing the **GET_RES_SCALE** action. This is explained later.

8 Compute the new state vector:

$$u_1 = u_0 + \delta u_0$$

with u_0 the old state vector. The new state vector is stored in the global database **globdat**.

9 Save the current constraints and make all constraints homogeneous by setting the prescribed values to zero.

This is to make sure that the state vectors computed in the subsequent Newton-Raphson iterations satisfy the initial constraints.

10 Assemble the second tangent stiffness matrix K_1 and the internal vector f_1 by means of the **GET_MATRIX0** action.

11 Compute the new residual:

$$r_1 = g - f_1$$

12 Update the residual scale factor:

$$s = \max(s, ||f_1||, ||r_1||)$$

If s is smaller than the property **tiny** then skip to Step 20.

13 Set the iteration number i to one. The Newton-Raphson loop starts here.

14 Check for convergence:

$$\frac{r_i}{s} < \epsilon$$

If this criterion is satisfied then skip to Step 20.

Note that a model can provide an alternative convergence criterion by implementing the action **CHECK_CONVERGED**.

15 Increment the iteration number:

$$i = i + 1$$

If i is larger than **maxIter**, then the old state vector is restored and failure is signalled by means of the **CANCEL** action. After that an exception is thrown to indicate failure.

16 Update the state vector:

$$\boldsymbol{u}_i = \boldsymbol{u}_{i-1} + \delta\boldsymbol{u}_{i-1}$$

and store the new state vector in **globdat**.

17 Assemble the tangent stiffness matrix $\boldsymbol{K}_i$ and the internal vector $\boldsymbol{f}_i$ by means of the **GET_MATRIX0** action.

18 Compute the new residual:

$$\boldsymbol{r}_i = \boldsymbol{g} - \boldsymbol{f}_i$$

19 Go to Step 14 to check for convergence again.

20 Restore the constraints.

21 Check whether the converged solution is acceptable by means of the **CHECK_COMMIT** action. By default, the solution is considered to be acceptable.

If the solution is acceptable, then signal this fact by means of the **COMMIT** action.

Otherwise signal failure by means of the **CANCEL** action restore the state vector to its old value, and go back to Step 1.

The **NonlinModule** throws and exception when it fails to find a converged solution.

Typically this exception is caught by the **exec** function of the **Application** class that will print an error message and terminate the program.

If one needs an alternative way of handing this situation, one can implement a custom module that wraps around the **NonlinModule**.

The **run** member function of this custom module would then call the **run** function of the **NonlinModule** within a try-catch block.

Special actions

The solution procedure can be adapted to a specific application by implementing the actions **GET_RES_SCALE** and/or **CHECK_CONVERGED** in a model.

The first action enables one to implement a different method for computing the residual scale factor. The second actions enables one to implement a different convergence criterion.

Both actions has a number of action-specific parameters that are stored in the **params** object that is passed to the **takeAction** function.

The **GET_RES_SCALE** action has the following parameters (defined as string constants in the **ActionParams** class):

RESIDUAL
: The residual vector.

INT_VECTOR
: The internal vector.

EXT_VECTOR
: The external vector.

RES_SCALE
: The residual scale factor. The initial value is computed by the **NonlinModule**. The model must set this parameter.

Example implementation of the **GET_RES_SCALE** action:

```
bool ExampleModel::takeAction ( const String&       action,
                                const Properties&   params,
                                const Properties&   globdat )
{
  if ( action == Actions::GET_RES_SCALE )
  {
    Vector  res, fint, fext;
    double  scale;

    params.get ( res,  ActionParams::RESIDUAL );
    params.get ( fint, ActionParams::INT_VECTOR );
    params.get ( fext, ActionParams::EXT_VECTOR );

    scale = ...; // Compute the residual scale factor.

    params.set ( ActionParams::RES_SCALE, rscale );

    return true;
  }

  return false;
}
```

The **CHECK_CONVERGED** action has the following parameters:

RESIDUAL
: The residual vector.

RES_SCALE
: The residual scale factor.

CONVERGED
: A boolean that indicates whether convergence has been obtained. The initial value is computed by the **NonlinModule**. The model must set this parameter.

Example implementation of the **CHECK_CONVERGED** action:

```
bool ExampleModel::takeAction ( const String&       action,
                                const Properties&   params,
                                const Properties&   globdat )
{
  if ( action == Actions::CHECK_CONVERGED )
  {
    Vector  res;
    double  scale;
    bool    conv;

    params.get ( res,   ActionParams::RESIDUAL );
    params.get ( scale, ActionParams::RES_SCALE );

    conv = ...; // Determine whether convergence has been reached.

    params.set ( ActionParams::CONVERGED, conv );

    return true;
  }

  return false;
}
```

The `MeshgenModule`

The **`MeshgenModule`** (package **`mesh`**) can be used to generate a finite element mesh. It is ***not*** a generic mesh generator as it supports only a few (simple) geometries.

The aim of the **`MeshgenModule`** is to simplify the development and testing of new models; one can use the **`MeshgenModule`** to generate a simple mesh and test whether the model computes the expected results.

The **MeshgenModule** creates a **NodeSet** and **ElementSet** that store the nodes and elements in the mesh. Both objects are stored in the global database.

The **MeshgenModule** also creates a number of **NodeGroup** instances containing the indices of the nodes on the boundaries of the mesh.

These node groups are stored in the global database and can be used to define the boundary conditions in a data file.

The **MeshgenModule** should be located at the head of the module chain and should be followed by the **InputModule**.

Example module chain:

```
Ref<Module> mainModule ()
{
  Ref<ChainModule>  chain = newInstance<ChainModule> ();

  chain->pushBack ( newInstance<MeshgenModule> () );
  chain->pushBack ( newInstance<InputModule>   () );

    :

  return newInstance<ReportModule> ( "report", chain );
}
```

Example data file:

```
<NodeGroup name = "seCorner">
  leftEdge & lowerEdge
</NodeGroup>

<NodeConstraints>
  dx[leftEdge] = 0.0;
  dy[seCorner] = 0.0;
</NodeConstraints>

<NodeTable name = "load">
  <Section columns = "dx">
    rightEdge 1.0;
  </Section>
</NodeTable>
```

The node groups **leftEdge**, **lowerEdge** and **rightEdge** are defined by the **MeshgenModule**. See the example properties on the next slide.

Example properties for the **MeshgenModule**:

```
meshgen =
{
  elemSize     = 0.1;
  output       =
  {
    file       = "$(CASE_NAME).mesh";
    precision  = 8;
  };
  geometry     = "Rectange"
  {
    size       = [ 4.0, 2.0 ];
    elemType   = "Quad8";
  };
};
```

These properties define a rectangular mesh with dimensions (4×2) and square, 8-node elements with size 0.1.

The **MeshgenModule** can be configured through the following properties:

elemSize
: A floating point number that specifies the (average) size of the elements in the mesh. This property controls the number of elements.

output
: An optional sub-property set that specifies a file to which the generated mesh is to be saved. Note that the mesh is always stored in **globdat**, even if this sub-property set is present.

output.file

A string specifying the name of the output file.

output.precision

An integer that specifies the number of digits that should be printed for each node coordinate (default: 8).

geometry

A typed sub-property set specifying the geometry of the mesh to be created.

The following geometry types are supported: **Rectangle**, **Circle**, **Ellipse** and **Ring**. The latter is a disc with a hole in the middle.

Each geometry type has its own set of properties that can be used to control the exact shape of the mesh and the names of the node groups related to the boundaries of the mesh.

Use the **ReportModule** to get an overview of the available properties per geometry type.

The **SampleModule**

The **SampleModule** (package **app**) can be used to save runtime variables stored in the **var** property set of the global database to a text file.

When its **run** member function is called, the **SampleModule** will get the current values of the specified runtime variables from the database and append those values to the end of the output file.

Example properties for the **SampleModule**:

```
sample =
{
  file      = "$(CASE_NAME).var.gz";
  append    = false;
  header    = "# Column 1: iteration number\n"
              "# Column 2: displacement";
  separator = ",";
  precision = 8;
  dataSets  = [ "i", "disp" ];
};
```

The **SampleModule** can be configured through the following properties:

file
: A string specifying the name of the output file. If it ends with the extension **.gz** then the file is compressed according to the GNU zip format.

append
: A boolean that indicates whether the output data is to be appended to an existing output file. When **false** (the default) then a new, empty file is created.

header

A string that will be written to the start of the output file. This is an empty string by default.

separator

A string that will be used to separate the values on a single line in the output file. This is a space by default.

precision

An integer that specifies the number of digits that should be printed for each floating point value in the output file. The default value is 8.

dataSets

An array of strings specifying the expressions that are to be used to obtain the output data. The number of columns in the output file is equal to the size of this array.

The `GraphModule`

The **`GraphModule`** (package **`gl`**) can be used to plot runtime variables stored in the **`var`** property set of the global database in a graph

When its **`run`** member function is called, the **`GraphModule`** will get the current values of the specified runtime variables from the database and add those as a new point to the graph.

This can be a useful tool to monitor the progress of a program.

Example properties for the **GraphModule**:

```
graph =
{
  title    = "Displacement Graph";
  xLabel   = "Iteration"
  yLabel   = "Displacement";
  dataSets = [ "disp" ];
  disp     =
  {
    key    = "";
    xData  = "i";
    yData  = "disp";
  };
};
```

The **GraphModule** can be configured through the following properties:

title
: A string specifying the title of the window. By default this is an empty string.

xLabel
: A string specifying the label below the X-axis; it is empty by default.

yLabel
: A string specifying the label left to the Y-axis; it is empty by default.

`dataSets`

An array of strings referring to the data sets to be plotted. Each entry in this array must be the name of a sub-property set containing the parameters for the data set; see below.

A data set to be plotted can be configured through the following properties:

key A string specifying the key that identifies the data set in the graph. By default this is an empty string.

xData
A string specifying an expression that is used to obtain the X-coordinate of each new point in the graph. The expression may involve runtime variables stored in **globdat**.

yData
A string specifying an expression that is used to obtain the Y-coordinate of each new point in the graph.

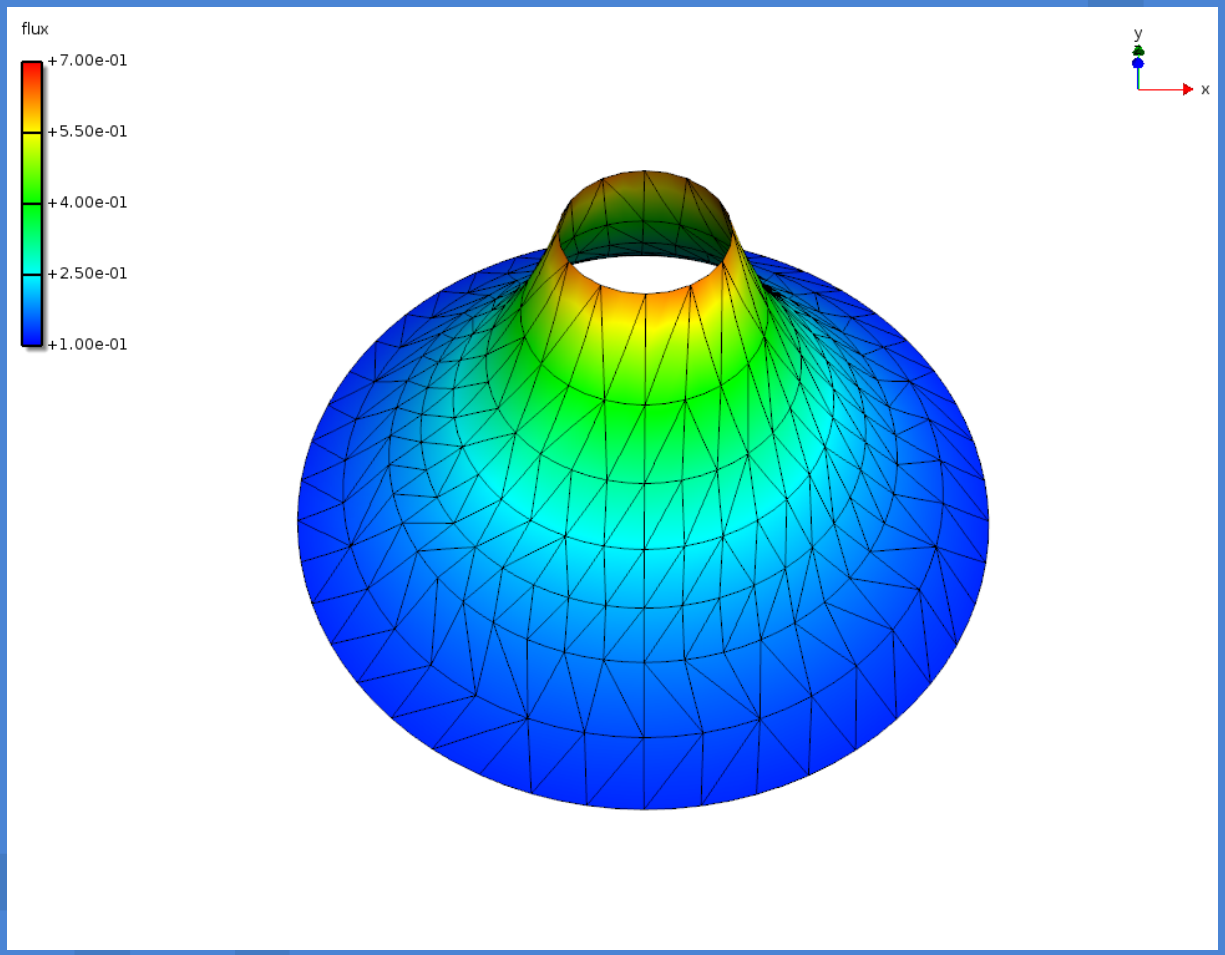
The `FemViewModule`

The **`FemViewModule`** (package **`gl`**) can be used to display a graphical representation of a mesh and the computed results in real time.

Like the **`GraphModule`**, it is primarily a tool to verify a program and to monitor the progress of a computation.

The **`FemViewModule`** therefore favors flexibility over image quality and a slick user interface.

Image created with the **FemViewModule**:



Example properties for the **FemViewModule**:

```
view =
{
  dataSets  = [ "state", "flux" ];
  state     = "Vector"
  {
    vector  = "state";
  };
  flux      = "Table"
  {
    table   = "nodes/flux";
  };
  mesh      =
  {
    deformation =
    {
      dz    = "state";
      scale = 0.5;
    };
    plugins = [ "colors" ];
    colors  = "MeshColorView"
    {
      data  = "flux{ abs( $$x ) }";
    };
  };
};
```

The **FemViewModule** can be configured through the following properties:

dataSets

An array of strings specifying the names of the data sets to be displayed. Each entry in the array should name a typed property set that selects and configures a data set.

Two data set types are supported: **Vector** and **Table**.

dataSets (cont)

A data set of type **Vector** can be used to display a vector stored in **globdat**. The sub-property **vector** must specify the name of the state vector. This should be **state**, **state1** or **state2**.

A data set of type **Table** can be used to display a table stored in **globdat** or a table that is filled by the model.

The sub-property **table** must specify the name of the item set associated with the table rows (**nodes** or **elements**), followed by a slash and the name of the table.

Example: **table = "nodes/stress";**

`mesh`

A property set that specifies how the mesh and the data sets are to be displayed.

`mesh.elements`

A string specifying the name of the element group to be displayed. If not specified, all elements in the mesh are selected.

`mesh.deformation`

A property set that controls if and how the displayed mesh should be deformed. If omitted, the mesh will be displayed with its original coordinates.

`mesh.displacement.dx`

The in the X-direction displacement to be added to the node coordinates. This can be specified as a floating point number (constant displacement), or a ***data set reference***.

The latter is a string that refers to one of the data sets that have been defined with the **`dataSets`** property. More about this later.

`mesh.displacement.dy`

The displacement in the Y-direction; similar to **`dx`**.

`mesh.displacement.dx`

The displacement in the Z-direction; similar to **`dx`**.

`mesh.plugins`

An array of strings specifying the so-called ***plugins*** that control how the mesh is rendered.

Each entry in this array must be the name of a typed property set that selects and configures a plugin.

Two useful plugins are the **`MeshViewColor`** plugin and the **`MeshNodeView`** plugin.

mesh.plugins (cont)

The **MeshColorView** plugin renders a contour plot of a data set. The sub-property **data** must specify a data set reference; see the **deformation** property.

The **MeshNodeView** plugin renders symbols (markers) at the nodes of the mesh. It can be used to display a node group, for instance.

The sub-property **nodes** (an array of strings) must specify the node groups and the marker styles.

mesh.plugins (cont)

Each entry in this array must contain the name of a node group, optionally followed by a colon (`:`) and a marker style name.

Example: `nodes = [ "innerEdge:orbs" ];`

If no marker style is specified, the node numbers will be displayed.

The **FemViewModule** has many more properties than the ones described here. Use the **ReportModule** to get a list of all properties.

Data set references

A data set reference is a string that refers to a data set that has been defined with the **`dataSets`** property.

It is used to obtain a value for each node/element in the mesh and must have one of the following forms:

name

: Indicates that the values should be equal to the 2-norm of all DOF types or table columns in the data set named *name*.

name`[`*var*`]`

Indicates that the values should be equal to the DOF type or table column named *var* in the data set named *name*.

name{*expr*}

Indicates that the values should be obtained by evaluating the expression *expr* in the context of the data set named *name*.

The expression may refer to the DOF types or table columns in that data set by prepending the token **$$** to the name of the DOF type or table column.

Here are some examples of data set references:

```
mesh.deformation.dx = "flux";
mesh.deformation.dy = "flux[ x ]";
mesh.deformation.dz = "flux{ abs( $$y ) }";
```

The first sets the X-displacement equal to the 2-norm of all columns in the data set **flux**.

The second sets the Y-displacement equal to the column named **x** in the data set **flux**.

The third sets the Z-displacement equal to the absolute values in the column **y** in the data set **flux**.

The **ShapeModule**

The **FemViewModule** expects to find a so-called ***shape table*** in the global database.

This shape table associates a **Shape** object with each element in the mesh. It is used by the **FemViewModule** to figure out how to render each element.

One can use the **ShapeModule** (package **fem**) to create the shape table. This module should come before the **FemViewModule** in the main module chain.

Example use of the **ShapeModule**:

```
Ref<Module> mainModule ()
{
  Ref<ChainModule>  chain = newInstance<ChainModule> ();

  chain->pushBack ( newInstance<InputModule>    () );
  chain->pushBack ( newInstance<ShapeModule>    () );
  chain->pushBack ( newInstance<FemViewModule> () );
    :

  return chain;
}
```

The **ShapeModule** has one property named **shapeTable** that selects and configures the shape table.

In most applications a shape table of type **Auto** is a good choice. This uses the number of nodes attached to an element to determine the corresponding **Shape** object.

Here is an example section of a properties file:

```
shapeTable = "Auto"
{};
```

Note that by default the **ShapeModule** has no name; the **shapeTable** property is therefore located in the global scope of the properties file.

The **DisplayModule**

The **FemViewModule** performs all render operations in a separate background thread of execution.

This means that the render operations are performed in parallel with the main simulation that runs in the main thread.

The result is a better user experience and no noticeable slow down of the simulation.

Unfortunately, on macOS (Apple's operating system) a background thread can not receive user input (mouse and keyboard events).

The **`DisplayModule`** (package **`gl`**) solves this problem by running the simulation in a background thread and performing the render operations in the main thread.

The **`DisplayModule`** is to be wrapped around the main module chain. It has no name and no properties.

Example use of the **DisplayModule**:

```
Ref<Module> mainChain ()
{
  Ref<ChainModule>  chain = newInstance<ChainModule> ();
  Ref<Module>       module;

  chain->pushBack ( newInstance<InputModule>    () );
  chain->pushBack ( newInstance<ShapeModule>    () );
  chain->pushBack ( newInstance<FemViewModule> () );

    :

  module = newInstance<ReportModule> ( "report", chain );

  return newInstance<DisplayModule>  ( module );
}
```

Example program

The directory **solutions/femview** contains an example program that shows how to use the **FemViewModule**.

The program solves a heat diffusion problem on a 2-D ring-shaped mesh.

The temperature is displayed as an offset in the Z-direction while the heat flux is displayed by means of the element colors.

One could use the example program as the basis of a stand-alone post-processor.

If you remove the **MeshgenModule**, the **InitModule** and the **LinsolveModule**, you have a program that can read a data file and display the mesh and the results in a graphical way.

This can be useful if you want to view the results of a simulation after it has ended, or if you want to run your program in the background without a graphics window (possibly on a remote server).

Built-in Jive models

The next slides explain the most relevant properties of some built-in Jive models.

As stated before, use the **`ReportModule`** to get a complete overview of all properties.

The properties of a model should be specified in a typed property set that has the name of the model.

The examples shown in the next slides will use the name **`model`**. In a real properties file a model may have a different name, depending on its position in the model tree.

The `MultiModel`

The **`MultiModel`** encapsulates a list of models. It is the basic building block for defining non-trivial model trees.

When its **`takeAction`** function is called, the **`MultiModel`** simply forwards that call to all its child models.

This is shown in the next slide.

Implementation of the **takeAction** member function:

```
bool MultiModel::takeAction ( const String&       action,
                              const Properties&   params,
                              const Properties&   globdat )
{
  bool  result = false;

  for ( idx_t i = 0; i < childs_.size(); i++ )
  {
    if ( childs_[i]->takeAction( action, params, globdat ) )
    {
      result = true;
    }
  }

  return true;
}
```

The private member variable **childs_** contains all child models.

Example properties for the **MultiModel**:

```
model = "Multi"
{
  models   = [ "elastic", "load" ];

  elastic = "Elastic"
  {
    // Properties for the ElasticModel ...
  };

  load     = "LoadScale"
  {
    // Properties for the LoadScaleModel ...
  };
};
```

The **MultiModel** has a property named **models** that is an array of strings. Each entry in the array must refer to a typed property set that selects and configures a child model.

The order of the array elements determines the order in which the **takeAction** function is called for each child model.

In the example shown in the previous slide the **takeAction** function is first called for the **ElasticModel** and then for the **LoadScaleModel**.

The **`MultiModel`** uses the **`ModelFactory`** class to instantiate a specified model type.

An exception is thrown if a specified model type has not been registered with the **`ModelFactory`**.

Classes are normally registered by calling the relevant declaration functions in the main module construction function.

In this way an application has complete control over the models that can be specified in the properties file.

The `MatrixModel`

The **`MatrixModel`** (package **`model`**) manages the **`MatrixBuilder`** instances that are used to assemble the global matrices (stiffness, damping and mass matrix).

The **`MatrixModel`** is typically the top-most model in the model tree.

You can omit the **`MatrixModel`** when using an explicit time integration method or when you need to manage the global matrices in a special way.

When the **MatrixModel** is requested to perform the action **NEW_MATRIX0**, **NEW_MATRIX1** or **NEW_MATRIX2** it creates a **MatrixBuilder** and returns the associated **AbstractMatrix** through the **params** parameter.

On the action **GET_MATRIX0**, **GET_MATRIX1** or **GET_MATRIX2** the **MatrixModel** stores the corresponding **MatrixBuilder** in the **params** parameter and forwards the action to its child model.

The next three slides show the definition and implementation of the **MatrixModel** class in simplified form.

Definition of the **MatrixModel** class:

```
class MatrixModel : public Model
{
 public:
                      MatrixModel ( ... );
  virtual bool        takeAction  ( ... );

 private:

  Ref<Model>          child_;
  Ref<MatrixBuilder>  mbld0_;
  Ref<MatrixBuilder>  mbld1_;
  Ref<MatrixBuilder>  mbld2_;
};
```

The function parameters are not shown for brevity.

Implementation of the **takeAction** function (part 1):

```
bool MatrixModel::takeAction ( const String&      action,
                               const Properties&  params,
                               const Properties&  globdat )
{
  if ( action == Actions::NEW_MATRIX0 )
  {
    mbld0_ = ...;  // Create a new MatrixBuilder using the
                   // matrix property.

    params.set ( ActionParams::MATRIX0, mbld0_->getMatrix() );

    return true;
  }

  // The actions NEW_MATRIX1 and NEW_MATRIX2 are
  // handled likewise ...

  // Continued on the next slide ...
}
```

Implementation of the **takeAction** function (part 2):

```
  // ... continued from the previous slide.

  if ( action == Actions::GET_MATRIX0 )
  {
    bool  result = false;

    mbld0_->setToZero ();
    params.set        ( ActionParams::MATRIX0, mbld0_ );

    result = child_->takeAction ( action, params, globdat );

    params.erase         ( ActionParams::MATRIX0 );
    mbld0_->updateMatrix ();

    return result;
  }

  // The actions GET_MATRIX1 and GET_MATRIX2 are
  // handled likewise ...

  return false;
}
```

Example properties for the **MatrixModel**:

```
model = "Matrix"
{
  matrix = "FEM"
  {
    symmetric = true;
  };

  matrix1 = "Sparse"
  {};

  matrix2 = "Lumped"
  {};

  model   = "Elastic"
  {
    // Properties for the ElasticModel ...
  };
};
```

The **MatrixModel** has the following properties:

matrix

A typed property set that selects and configures the **MatrixBuilder** for assembling the (tangent) stiffness matrix.

The following **MatrixBuilder** types can be selected: **FEM**, **Sparse** and **Lumped**.

The **FEM** type (from the package **fem**) is aimed at finite element applications. It can only be used when the global stiffness matrix has a standard finite element structure.

matrix (cont)

The **Sparse** type (package **algebra**) can be used to assemble a global matrix with an arbitrary structure.

The **Lumped** type (also package **algebra**) creates a diagonal matrix by adding all off-diagonal matrix entries to the diagonal.

To use the **FEM** type, the function **declareMBuilders** from the header file **<jive/fem/declare.h>** must be called in the main module construction function.

To use the **Sparse** or **Lumped** type, the function **declareMBuilders** from the header file **<jive/algebra/declare.h>** must be called first.

matrix.symmetric

A boolean value that indicates whether the global stiffness matrix is symmetric. The default value is **false**.

Some linear solvers use this property to select more efficient algorithms that are aimed at symmetric matrices.

matrix1

A typed property set that selects and configures the **MatrixBuilder** for assembling the damping matrix.

It is similar to the **matrix** property. It can be omitted if the application does not involve a damping matrix.

`matrix2`

A typed property set that selects and configures the **`MatrixBuilder`** for assembling the mass matrix.

It is similar to the **`matrix`** property. It can be omitted if the application does not involve a mass matrix.

`model`

A typed property set that selects and configures the child model.

The `PointLoadModel`

The **`PointLoadModel`** (package **`model`**) can be used to assemble the external vector.

It takes a table stored in the global data base and adds the table values to the external vector. The names of the table columns must match the DOF type names.

The table can be obtained from a data file (through the **`InputModule`**) or be created by another module or model.

Example properties for the **PointLoadModel**:

```
model = "PointLoad"
{
  loadTable = "load";
};
```

The property **loadTable** specifies the name of the table to be added to the external vector.

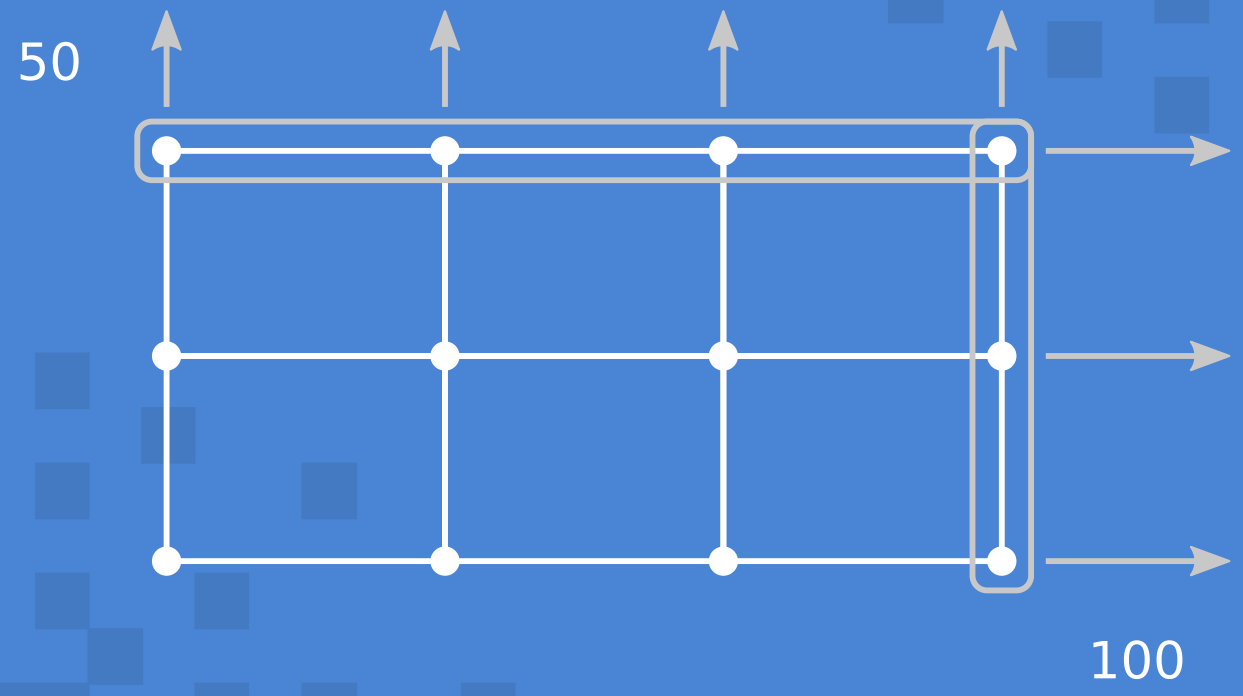
Suppose that the **load** table is defined as follows:

```
<NodeTable name = "load">
  <Section columns = "dx">
    rightEdge 100.0;
  </Section>
  <Section columns = "dy">
    upperEdge 50.0;
  </Section>
</NodeTable>
```

The **PointLoadModel** will then add the value 100 to the entries in the external vector that correspond with the DOFs of type **dx** in the nodes that are part of the group **rightEdge**.

It will also add the value 50 to the entries in the external vector that correspond with the DOFs of type **dy** in the nodes that are part of the group **upperEdge**.

Illustration of the **`load`** table used by the **`PointLoadModel`**:



The `ConstraintsModel`

The **`ConstraintsModel`** (package **`model`**) can be used to copy the values stored in a table to a **`Constraints`** object.

It plays a similar role as the **`PointLoadModel`** but for the constraints instead of the external vector.

The **`ConstraintsModel`** is primarily aimed at simulations in which the constraints change dynamically. It is often used together with the **`LoadScaleModel`**.

Example properties for the **ConstraintsModel**:

```
model = "Constraints"
{
  conTable = "disp";
};
```

The property **conTable** specifies the name of the table to be copied to the **Constraints**.

The names of the table columns must match the DOF type names.

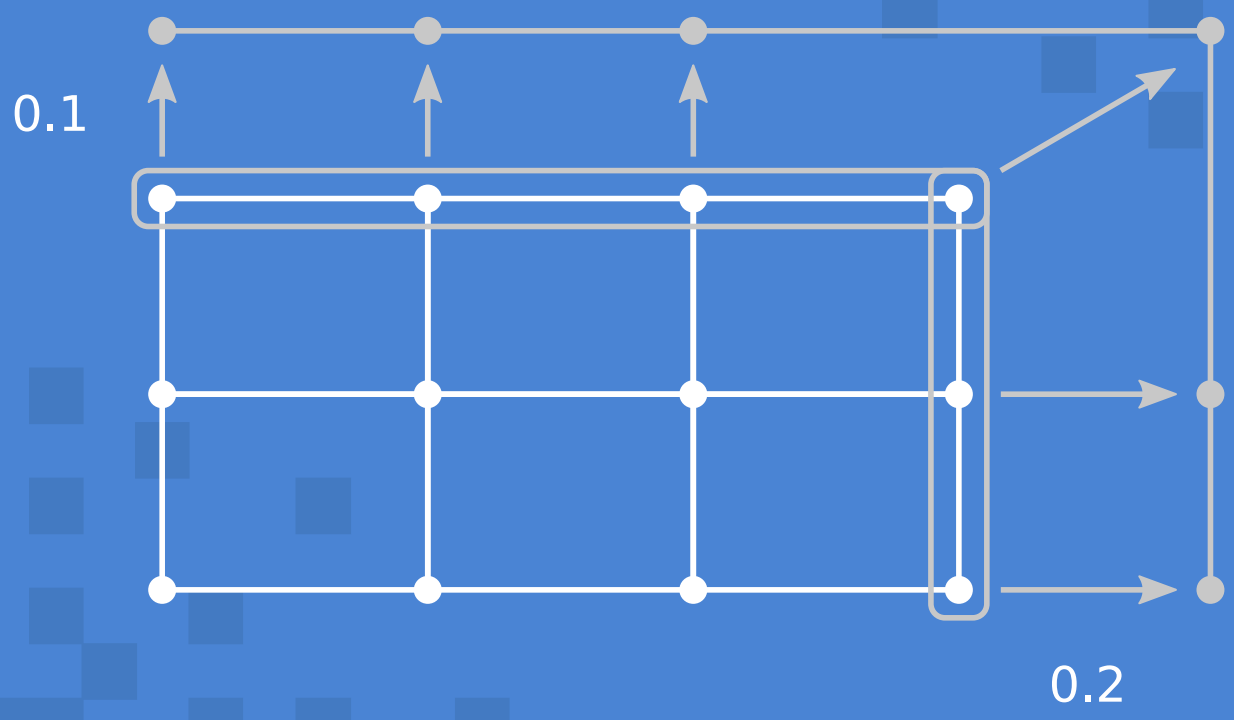
Suppose that the **disp** table is defined as follows:

```
<NodeTable name = "disp">
  <Section columns = "dx">
    rightEdge 0.2;
  </Section>
  <Section columns = "dy">
    upperEdge 0.1;
  </Section>
</NodeTable>
```

The **ContraintsModel** will then impose the prescribed value 0.2 on all DOFs of type **dx** in the nodes that are part of the group **rightEdge**.

It will also impose the prescribed value 0.1 on all DOFs of type **dy** in the nodes that are part of the group **upperEdge**.

Illustration of the **`disp`** table used by the **`ConstraintsModel`**:



The `LoadScaleModel`

The **`LoadScaleModel`** (package **`model`**) can be used to scale the external vector and/or the constraints with a user-defined function.

When it is requested to perform the action **`GET_EXT_VECTOR`** or **`GET_CONSTRAINTS`**, it calculates a scale factor.

The scale factor is stored in the **`params`** object with the name **`SCALE_FACTOR`** and passed to its child model.

Implementation of the **takeAction** member function:

```
bool LoadScaleModel::takeAction ( const String&       action,
                                  const Properties&   params,
                                  const Properties&   globdat )
{
  if ( action == Actions::GET_EXT_VECTOR
       action == Actoins::GET_CONSTRAINTS )
  {
    double  scale;

    scale = ...; // Calculate the load scale factor.

    params.set ( ActionParams::LOAD_SCALE, scale );
  }

  return child_->takeAction ( action, params, globdat );
}
```

The scale factor is not applied automatically; a child model must retrieve the scale factor from **`params`** and apply it to its contribution to the external vector or constraints.

Both the **`PointLoadModel`** and the **`ConstraintsModel`** look for the scale factor in **`params`**. If it has not been set, a scale factor of one will be used.

If you implement your own model for computing the external vector (or constraints) you can make it compatible with the **`LoadScaleModel`** by looking for the scale factor in **`params`**.

Example properties for the **LoadScaleModel**:

```
model = "LoadScale"
{
  scaleFunc   = "if ( i < 10, i * 0.1, 1.0 )";
  model       = "PointLoad"
  {
    loadTable = "load";
  };
};
```

This indicates that the entries in the table **load** are to be scaled with a factor that increases linearly from zero to 1.0 in the first ten time/load steps. The scale factor remains 1.0 after that.

The **LoadScaleModel** has the following properties:

scaleFunc

A string that specifies an expression for the load scale factor. The expression may involve any runtime variables stored in **globdat**.

model

A typed property set that selects and configures the child model.

This model should apply the load scale factor when handling the **GET_EXT_VECTOR** or **GET_CONSTRAINTS** action.

Non-linear example

Most programs developed with Jive feature non-linear models that need to keep track of one or more material state parameters.

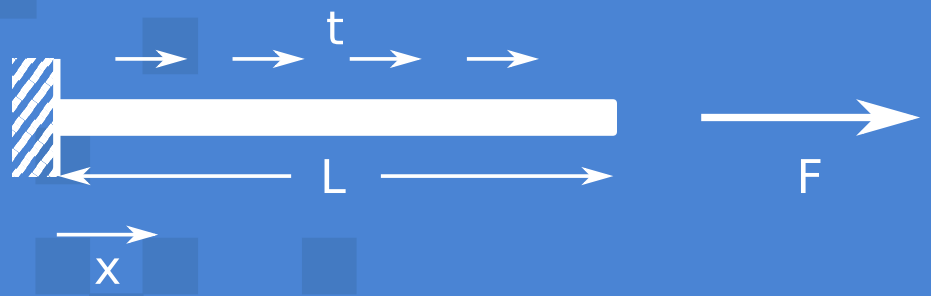
A second example program, named **`damage`**, illustrates how this works by implementing a non-linear damage mechanics model.

This program is based on the **`elastic`** example program and adds a **`DamageModel`**.

Problem description

The problem geometry and boundary conditions are the same as before.

That is, the **damage** program computes the deformation of a 1-D bar clamped at the left, and subjected to a force at the right and a traction along the bar.



The relation between the stress and the strain is no longer linear:

$$\sigma = (1 - \lambda)\ E\,\epsilon$$

where λ is the so-called ***damage parameter***.

The damage parameter starts at zero when the material has not been subjected to large deformations yet.

When the stress (or strain) exceeds a certain threshold, the damage parameter increases until it has reached the value one. At that point the material can no longer bear any load.

Note that the damage parameter can only increase; damage to the material can not be undone.

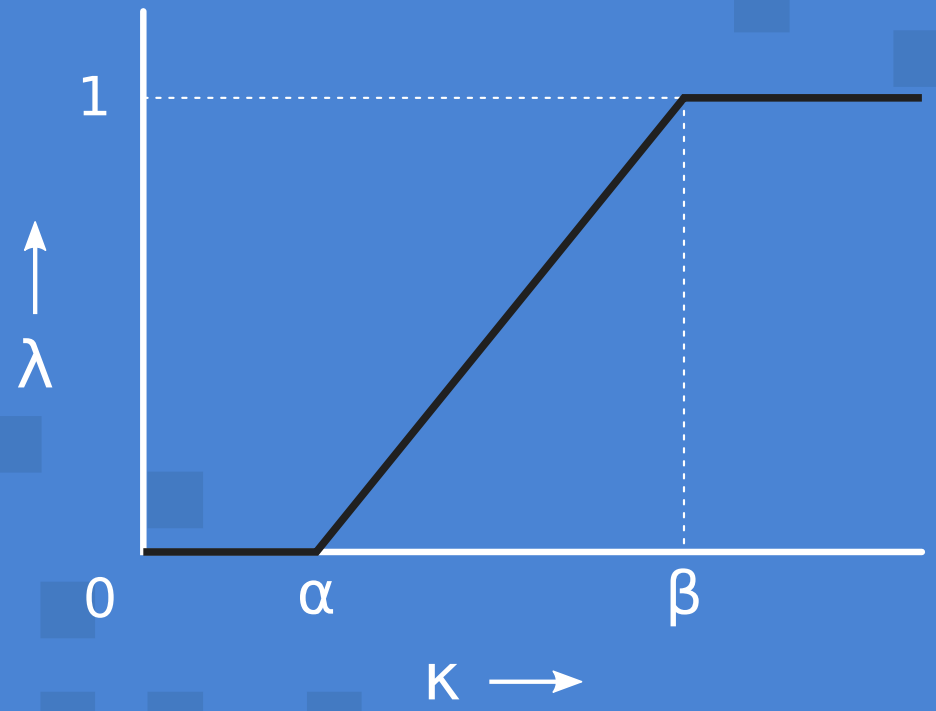
A very simple, linear strain-based damage formulation is used:

$$\lambda = \begin{cases} 0 & \text{for} \quad \kappa \leq \alpha \\ \frac{\kappa-\alpha}{\beta-\alpha} & \text{for} \quad \alpha < \kappa < \beta \\ 1 & \text{for} \quad \kappa \geq \beta \end{cases}$$

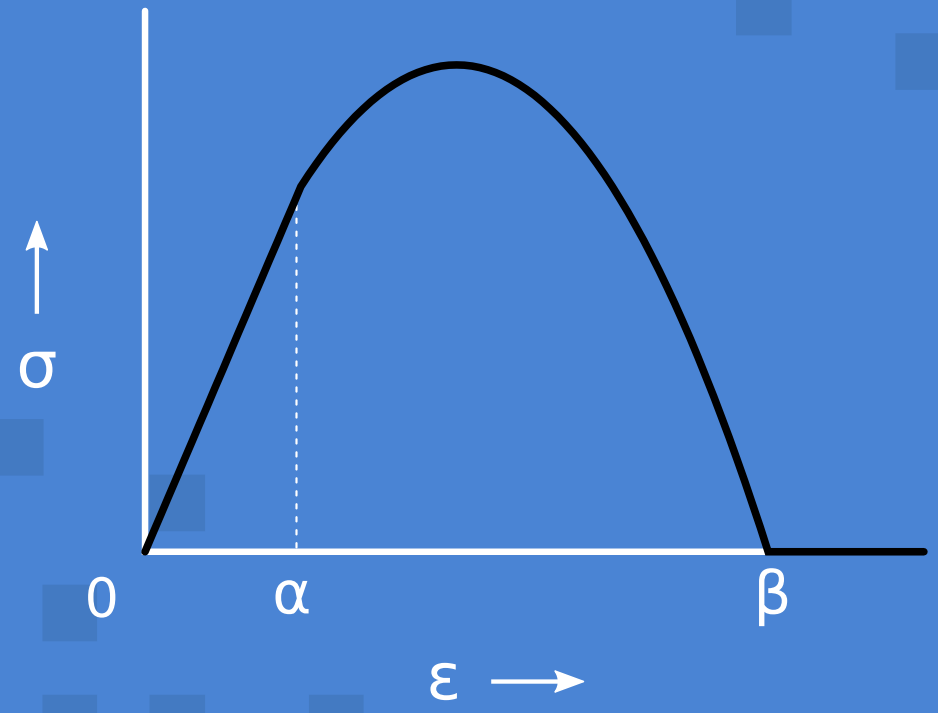
where κ denotes the maximum strain that has been obtained throughout the history of the material.

The parameter κ is called the ***history parameter***.

The damage as function of the history parameter:



The strain-stress curve:



The history parameter represents the state of the material. It must be stored and updated for all relevant ***material points***. That is, all points in which the strain (and stress) is evaluated.

In the finite element method, these points are (normally) the integration points of the elements.

Solution procedure

Like before, the displacement field is computed by means of the finite element method.

In order to follow the equilibrium path, the external force and traction are gradually increased in multiple ***load steps***.

In each load step the Newton-Raphson algorithm is used to determine the new displacement field and the new history parameter.

A so-called ***total incremental*** update procedure is used to avoid that the history parameter is affected by the convergence behavior of the Newton-Raphson algorithm.

This means that the history parameter in an integration point is updated as follows:

$$\kappa = \max(\epsilon, \kappa_0)$$

where κ_0 denotes the history parameter from the previous load step. This will be called the ***old*** history parameter.

The solution procedure is as follows:

1. Assemble the global tangent matrix $\boldsymbol{K}$ and the internal vector $\boldsymbol{f}$.
2. Compute the displacement increment:

$$\boldsymbol{\delta u} = \boldsymbol{K}^{-1}(\boldsymbol{g} - \boldsymbol{f})$$

3. Compute the new displacement vector:

$$\boldsymbol{u} := \boldsymbol{u} + \boldsymbol{\delta u}$$

4 For each element i, compute the new strain:

$$\epsilon = \boldsymbol{G}_i \boldsymbol{u}_i$$

with $\boldsymbol{G}_i$ the gradients of the shape functions, and $\boldsymbol{u}_i$ the displacements in the element nodes.

5 For each integration point, update the history parameter:

$$\kappa = \max\left(\epsilon, \kappa_0\right)$$

Also compute the new damage parameter.

6 Compute the tangent element stiffness matrix $\boldsymbol{K}_i$ and the internal element vector $\boldsymbol{f}_i$.

7 Assemble the global tangent matrix $\boldsymbol{K}$ and the internal vector f.

8 Check convergence. If not converged, then go back to Step 2.

9 Update the old history parameter:

$$\kappa_0 := \kappa$$

10 Increase the load and continue with Step 1.

Tangent element matrix

The tangent element matrix is obtained by taking the derivative of the internal element vector with respect to the element displacements:

$$\boldsymbol{K}_i = \frac{\partial \boldsymbol{f}_i}{\partial \boldsymbol{u}_i}$$

The internal element vector is given by:

$$\boldsymbol{f}_i = \int_{\Omega_i} \boldsymbol{G}_i^T \; (1-\lambda) \; EA \, \boldsymbol{G}_i \, \boldsymbol{u}_i \, d\Omega$$

The tangent element matrix therefore becomes:

$$\begin{aligned} \boldsymbol{K}_i &= \int_{\Omega_i} \boldsymbol{G}_i^T \, (1-\lambda) \, EA \, \boldsymbol{G}_i \, d\Omega \; - \\ &\qquad \int_{\Omega_i} \boldsymbol{G}_i^T \, EA \, \boldsymbol{G}_i \, \boldsymbol{u}_i \, \frac{\partial \lambda}{\partial \boldsymbol{u}_i} \, d\Omega \\ &= \bar{\boldsymbol{K}}_i - \tilde{\boldsymbol{K}}_i \end{aligned}$$

The first term $\bar{\boldsymbol{K}}_i$ is the elastic part of the matrix. The second term $\tilde{\boldsymbol{K}}_i$ is the non-linear part due to the damage model.

According to the damage formulation:

$$\frac{\partial \lambda}{\partial \boldsymbol{u}_i} = \begin{cases} \frac{1}{\beta - \alpha} \frac{\partial \epsilon}{\partial \boldsymbol{u}_i} & \text{if} \quad \alpha < \kappa_0 < \kappa < \beta \\ 0 & \text{otherwise} \end{cases}$$

In other words, the second term $\tilde{\boldsymbol{K}}_i$ in the tangent element stiffness matrix is non zero only if the damage increases.

Because

$$\epsilon = \boldsymbol{G}_i \, \boldsymbol{u}_i$$

the second term in the tangent matrix becomes:

$$\tilde{\boldsymbol{K}}_i = \int_{\Omega_i} \boldsymbol{G}_i^T \, \frac{EA}{\beta - \alpha} \, \boldsymbol{G}_i \, \boldsymbol{u}_i \, \boldsymbol{G}_i \, d\Omega$$

$$= \int_{\Omega_i} \boldsymbol{G}_i^T \, \frac{EA\,\epsilon}{\beta - \alpha} \, \boldsymbol{G}_i d\Omega \qquad \text{if} \quad \alpha < \kappa_0 < \kappa < \beta$$

Implementation

The implementation of the **`damage`** example program is similar to that of the **`elastic`** program with a few modifications and extensions.

One modification concerns the main module chain in which the **`LinsolveModule`** has been replaced by the **`NonlinModule`**.

The most significant extension concerns the new **`DamageModel`** that implements the damage formulation.

The main module construction function:

```
Ref<Module> mainModule ()
{
  Ref<ChainModule>  chain = newInstance<ChainModule> ();

  jive::fem ::declareMBuilders ();
  jive::model::declareModels   ();
  declareElasticModel          ();
  declareDamageModel           ();

  chain->pushBack ( newInstance<InputModule>    () );
  chain->pushBack ( newInstance<InitModule>     () );
  chain->pushBack ( newInstance<InfoModule>     () );
  chain->pushBack ( newInstance<NonlinModule>   () );
  chain->pushBack ( newInstance<OutputModule>   () );
  chain->pushBack ( newInstance<ControlModule> () );

  return newInstance<ReportModule> ( "report", chain );
}
```

The **`DamageModel`** is similar to the **`ElasticModel`**, except for three differences.

The first difference concerns five additional data members that store the history parameters, the damage parameters, and the material parameters α and β.

The second difference concerns the implementation of the **`takeAction`** function that needs to update the old history parameters.

The third difference concerns the implementation of the private member function **`assemble_`** that assembles the tangent stiffness matrix and the internal vector.

Definition of the **DamageModel** class:

```
class DamageModel : public Model
{
 public:
                              DamageModel ( ... );
  virtual bool                takeAction  ( ... );

 private:

  void                        assemble_   ( ... );

 private:

  Assignable<NodeSet>         nodes_;
  Assignable<ElementSet>      elems_;
  Ref<XDofSpace>              dofs_;
  Ref<IShape>                 shape_;
  idx_t                       jtype_;

  Matrix                      hist_, hist0_; // History parameters.
  Matrix                      damage_;       // Damage parameters.

  double                      young_;
  double                      area_;
  double                      alpha_;
  double                      beta_;
};
```

The data members **`hist_`** and **`hist0_`** store the current and old history parameters in all integration points.

They are matrices with shape $(m \times n)$, with m the number of integration points per element and n the number of elements.

The history parameter in integration point i of element j is obtained with the expression **`hist_(i,j)`**.

Both **`hist_`** and **`hist0_`** are resized and set to zero in the constructor of the **`DamageModel`**.

The data member **damage_** stores the current damage parameter in each integration point.

It, too, is a matrix with the same dimensions as the members **hist_** and **hist0_**. It is resized and set to zero in the constructor.

The **damage_** member is not strictly necessary as the damage parameters can be computed from the history parameters.

It is convenient, however, when handling the **GET_TABLE** action to compute the stresses.

The data members **`alpha_`** and **`beta_`** are used in the calculation of the damage parameter.

They indicate at which strain level damage starts to occur, and at which strain level damage is complete.

Both members are initialised in the constructor by extracting their values from the **`props`** object containing all runtime parameters.

The **takeAction** function needs to update the old history parameters at the end of a load step as follows:

```
bool DamageModel::takeAction ( const String&      action,
                               const Properties&  params,
                               const Properties&  globdat )
{
  if ( action == Actions::GET_MATRIX0 )
  {
    // Call the assemble_ function ...
  }

  if ( action == Actions::COMMIT )
  {
    hist0_ = hist_;

    return true;
  }

  return false;
}
```

The **`assemble_`** function calculates the global tangent matrix and the internal vector by assembling the element matrices and element vectors.

The element matrix is composed of a linear part $\bar{\boldsymbol{K}}_i$ and a non-linear part $\tilde{\boldsymbol{K}}_i$ that are calculated separately.

The internal vector for element i is obtained by multiplying the linear part of the element matrix with the displacements:

$$\boldsymbol{f}_i = \bar{\boldsymbol{K}}_i \, \boldsymbol{u}_i$$

The **assemble_** function also updates the current history parameters stored in the member **hist_**, and the damage parameters stored in **damage_**.

That is, for each integration point it calculates the strain and determines the new history parameter as follows:

$$\epsilon = \boldsymbol{G}_i \, \boldsymbol{u}_i \, ; \quad \kappa = \max\left(\kappa_0, \epsilon\right)$$

If the new history parameter is larger than the old one, then one must add a contribution to the non-linear part $\tilde{\boldsymbol{K}}_i$ of the element matrix.

Exercise: finish the implementation

Finish the implementation of the **`damage`** program by filling in the blank spots in the member function **`assemble_`**.

Comments with the word **`TODO`** indicate what is to be done.

Exercise location: **`exercises/damage-1`**.

Solution location: **`solutions/damage-1a`**.

Details

To validate the model, show the maximum internal force as function of the maximum displacement in a graph.

Store the two values as runtime parameters in the **`var`** section of the global database when handling the action **`GET_MATRIX0`**.

Use the names **`load`** and **`disp`** for the two values. The **`GraphModule`** has already been configured to display these two runtime variables.

The **`NonlinModule`** will fail after a few load steps. Why?

What can you do to follow the complete equilibrium path from zero to complete damage?

Using element groups

The **`damage`** program can either use the **`DamageModel`** or the **`ElasticModel`**, but not both.

The reason is that each model will assemble the element matrices for all elements in the mesh.

This limitation can be removed by using the **`ElementGroup`** class.

The idea is that both models are associated with an element group and that they only handle the elements in their group.

To do this, each model class definition is extended with a private data member named **`group_`** that stores the indices of the elements in the group.

Note that the **`Assignable`** class needs to be used in order to use the copy assignment operator in the constructor.

Extension of class definition with the **group_** member:

```
class DamageModel : public Model
{
 public:
                                DamageModel ( ... );
  virtual bool                  takeAction  ( ... );

 private:

  void                          assemble_   ( ... );

 private:

  Assignable<ElementGroup>  group_;  // The element group.
  Assignable<NodeSet>       nodes_;
  Assignable<ElementSet>    elems_;
  Ref<XDofSpace>            dofs_;
  Ref<IShape>               shape_;
  idx_t                     jtype_;

  // Other members ...

};
```

The **group_** member is initialised in the constructor:

```
DamageModel::DamageModel ( const String&       name,
                           const Properties&   conf,
                           const Properties&   props,
                           const Properties&   globdat ) :

  Model ( name )

{
  Properties  myProps = props.findProps ( myName_ );
  Properties  myConf  = conf .makeProps ( myName_ );

  group_ = ElementGroup::get   ( myConf,  myProps,
                                 globdat, getContext() );
  elems_ = group_.getElements ();
  nodes_ = elems_.getNodes    ();

    :
}
```

The function **`ElementGroup::get`** will look for a property named **`elements`** in the sub-property set associated with the model.

This property should be the name of an element group defined in the input file or by the **`MeshgenModule`**.

This is what the relevant section in the properties file looks like:

```
model = "Damage"
{
  elements = "all";  // Use the entire mesh.
  young    = 2.0e11;
    :
};
```

The degrees of freedom should only be defined for the nodes attached to the element in the element group.

This can be done by using the member function **`getUniqueNodesOf`** of the **`ElementSet`** class.

This function return an integer array containing the indices of the nodes attached to a given set of elements.

The array is filtered so that it contains no duplicate node indices.

How to define the DOFs in the constructor:

```
DamageModel::DamageModel ( const String&       name,
                           const Properties&   conf,
                           const Properties&   props,
                           const Properties&   globdat ) :

  Model ( name )

{
   :

  dofs_   = XDofSpace::get ( nodes_.getData(), globdat );
  jtype_  = dofs_->addType ( "u" );

  dofs_->addDofs (
    elems_.getUniqueNodesOf ( group_.getIndices() ),
    jtype_
  );

   :
}
```

The assemble function does not iterate over all elements but only over the elements in the group.

This is how it works:

```
void DamageModel::assemble_ ( ... )
{
  IdxVector  ielems = group_.getIndices ();

    :

  for ( idx_t ie = 0; ie < ielems.size(); ie++ )
  {
    idx_t  ielem = ielems[ie];  // Get the global element index.

    elems_.getElemNodes ( inodes, ielem );

      :
  }
}
```

The members **`hist_`**, **`hist0_`** and **`damage_`** only need to store the history and damage parameters for the elements in the group.

This means that their extent in the second dimension should be equal to the number of the elements in the group instead of the total number of elements.

Their extent in the first dimension remains equal to the number of integration points per element.

Initialisation of the history and damage parameters:

```
DamageModel::DamageModel ( const String&       name,
                           const Properties&   conf,
                           const Properties&   props,
                           const Properties&   globdat ) :

  Model ( name )

{
    :

  hist_   .resize ( shape_->ipointCount(), group_.size() );
  hist0_  .resize ( shape_->ipointCount(), group_.size() );
  damage_.resize ( shape_->ipointCount(), group_.size() );

  hist_    = 0.0;
  hist0_   = 0.0;
  damage_  = 0.0;

    :
}
```

To access an element in the **hist_**, **hist0_** and **damage_**, you need to use the index in the element group and ***not*** the global element index.

That is, instead of

```
hist_(ip,ielem)
```

you should write

```
hist_(ip,ie)
```

where **ip** is the integration point index, **ielem** the global element index and **ie** the index in the element group.

Correct use of the data **hist_**, **hist0_** and **damage_**:

```
void DamageModel::assemble_ ( ... )
{
  const idx_t  ipCount = shape_->ipointCount ();
  IdxVector    ielems  = group_ .getIndices  ();

    :

  for ( idx_t ie = 0; ie < ielems.size(); ie++ )
  {
      :

    for ( idx_t ip = 0; ip < ipCount; ip++ )
    {
        :

      hist          = max ( strain, hist0_(ip,ie) );
      hist_(ip,ie) = hist;

        :
    }
  }
}
```

Exercise: use element groups

Modify the **`damage`** program so that you can associate a model with an element group.

Create or modify two input files (properties file and data file) that define two models with different material parameters.

Exercise location: **`exercises/damage-1`**.

Solution location: **`solutions/damage-1b`**.

Hints

Add an **`ElementGroup`** member to the classes **`ElasticModel`** and **`DamageModel`**.

Update the constructors and the member functions **`assemble_`**, **`getStress_`** and **`getDamage_`**.

Use the **`ElementGroup`** tag to define at least two element groups in the data file.

Extension to two dimensions

The **damage** example program has many characteristics of a typical non-linear Jive program.

It is not very exciting, however, as it only solves a one-dimensional mechanics problem.

Fortunately, one can extend the program to two dimensions without having to make major changes to its structure; only smaller changes are required.

Modified class definition:

```
class DamageModel : public Model
{
 public:
                  DamageModel   ( ... );
  virtual bool    takeAction    ( ... );

 private:
  void            assemble_     ( ... );
  void            getStiffmat_  ( ... );  // Explained later.
  void            getBMatrix_   ( ... );  // Explained later.

 private:

    :

  IdxVector       jtypes_;  // Array instead of scalar.
  double          young_;
  double          poisson_; // Poisson ratio instead of area.

    :
};
```

The first change concerns the degrees of freedom types.

Instead of one, the **`ElasticModel`** and **`DamageModel`** classes need to declare two types: one for the X-displacements, and one for the Y-displacements.

These DOF types are labelled **`dx`** and **`dy`**, respectively.

The indices associated with these types are stored in a private member named **`jtypes_`** of type **`IdxVector`**.

Another change concerns the **`InternalShape`** object that is used to evaluate the shape function gradients.

This object is constructed with the **`Quad4`** class instead of the **`Line2`** class.

A third change involves the material parameters. Instead of the cross-section area of the bar, the models use the Poisson ratio as a runtime parameter.

Relevant changes to the model constructors:

```
DamageModel:: DamageModel ( const String&  name, ... ) :

  Model ( name )

{
  String  ischeme = "Gauss2 * Gauss2";
  String  bscheme = "Gauss2";

    :

  shape_ = Quad4::getShape ( "quad", ischeme, bscheme );

    :

  jtypes_.resize ( 2 );

  jtypes_[0] = dofs_->addType ( "dx" );
  jtypes_[1] = dofs_->addType ( "dy" );

    :

  myProps.get ( young_,   "young" );
  myProps.get ( poisson_, "poisson" );
}
```

The third change concerns the way that the strain and stress are calculated.

In two dimensions the strain is a vector with three components that is obtained from the displacements in the nodes of the i-th element as follows:

$$\boldsymbol{\epsilon} = \boldsymbol{B}_i \, \boldsymbol{u}_i$$

where the matrix is a $(3 \times n)$ matrix containing the gradients of the shape functions, and n is the number of nodes per element times two (because each node has two displacements).

This is what the B_i matrix looks like for an element with four nodes:

$$\boldsymbol{B}_i = \begin{bmatrix} \frac{\partial \phi_{i,1}}{\partial x} & 0 & \frac{\partial \phi_{i,2}}{\partial x} & 0 & \frac{\partial \phi_{i,3}}{\partial x} & 0 & \frac{\partial \phi_{i,4}}{\partial x} & 0 \\ 0 & \frac{\partial \phi_{i,1}}{\partial y} & 0 & \frac{\partial \phi_{i,2}}{\partial y} & 0 & \frac{\partial \phi_{i,3}}{\partial y} & 0 & \frac{\partial \phi_{i,4}}{\partial y} \\ \frac{\partial \phi_{i,1}}{\partial y} & \frac{\partial \phi_{i,1}}{\partial x} & \frac{\partial \phi_{i,2}}{\partial y} & \frac{\partial \phi_{i,2}}{\partial x} & \frac{\partial \phi_{i,3}}{\partial y} & \frac{\partial \phi_{i,3}}{\partial x} & \frac{\partial \phi_{i,4}}{\partial y} & \frac{\partial \phi_{i,4}}{\partial x} \end{bmatrix}$$

with $\phi_{i,j}$ the j-th shape function of element i.

The B matrices are obtained by passing the shape function gradients to the new private member function **`getBMatrix_`**.

The stress vector can be obtained from the strain vector as follows:

$$\sigma = (1 - \lambda)\ \boldsymbol{D}\,\epsilon$$

where the (3×3) matrix $\boldsymbol{D}$ is the linear strain-stress stiffness matrix.

This matrix is computed by the private member function **`getStiffmat_`** that uses the **`young_`** and **`poisson_`** members.

The final change has to do with the computation of the tangent stiffness matrix.

This matrix is given by the following expression:

$$\begin{aligned} \boldsymbol{K}_i &= \int_{\Omega_i} \boldsymbol{B}_i^T \, (1-\lambda) \, \boldsymbol{D} \, \boldsymbol{B}_i \, d\Omega \; - \\ & \qquad \int_{\Omega_i} \boldsymbol{B}_i^T \, \boldsymbol{\sigma} \, \frac{\partial \lambda}{\partial \kappa} \, \frac{\partial \kappa}{\partial \boldsymbol{u}_i} \, d\Omega \\ &= \qquad \bar{\boldsymbol{K}}_i - \tilde{\boldsymbol{K}}_i \end{aligned}$$

where κ is the history parameter.

Many different damage models have been developed that relate the history parameter in different ways to the evolution of the strain and stress.

The **`damage`** example program defines the history parameter as being the maximum value of the volumetric strain or ***dilatation***:

$$\kappa = \max\left(\epsilon_{xx} + \epsilon_{yy}\right) = \max\left(\boldsymbol{p}^T\,\boldsymbol{\epsilon}\right)$$

where the vector $\boldsymbol{p}$ is given by:

$$\boldsymbol{p}^T = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}$$

While this is not the most realistic definition of the history parameter, it does simplify the computation of the tangent stiffness matrix.

The non-linear part of the tangent stiffness matrix becomes:

$$\tilde{\boldsymbol{K}}_i = \int_{\Omega_i} \boldsymbol{B}_i^T \, \boldsymbol{\sigma} \, \boldsymbol{p}^T \frac{\partial \lambda}{\partial \kappa} \, \boldsymbol{B}_i \, d\Omega$$

The product $(\boldsymbol{\sigma}\, \boldsymbol{p}^T)$ is a (3×3) matrix of which the first two columns contain the stress vector and the last column is zero.

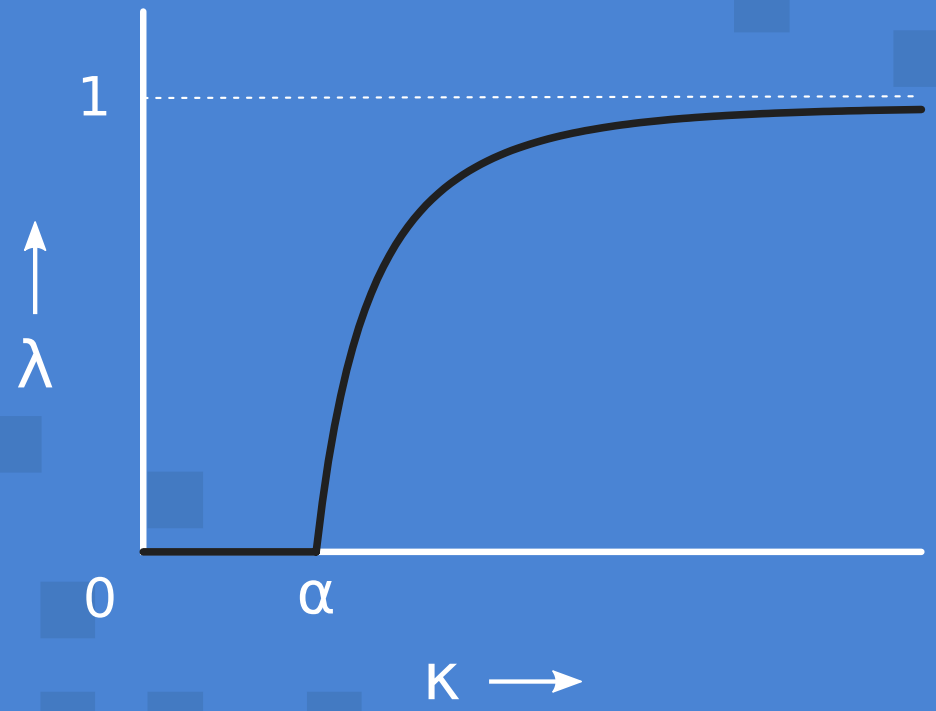
Instead of using a linear relation between the damage and the history parameters, the two-dimensional version of the **damage** program implements an exponential softening damage model:

$$\lambda = 1 - \frac{\alpha}{\kappa} e^{-\frac{\kappa-\alpha}{\beta-\alpha}}$$

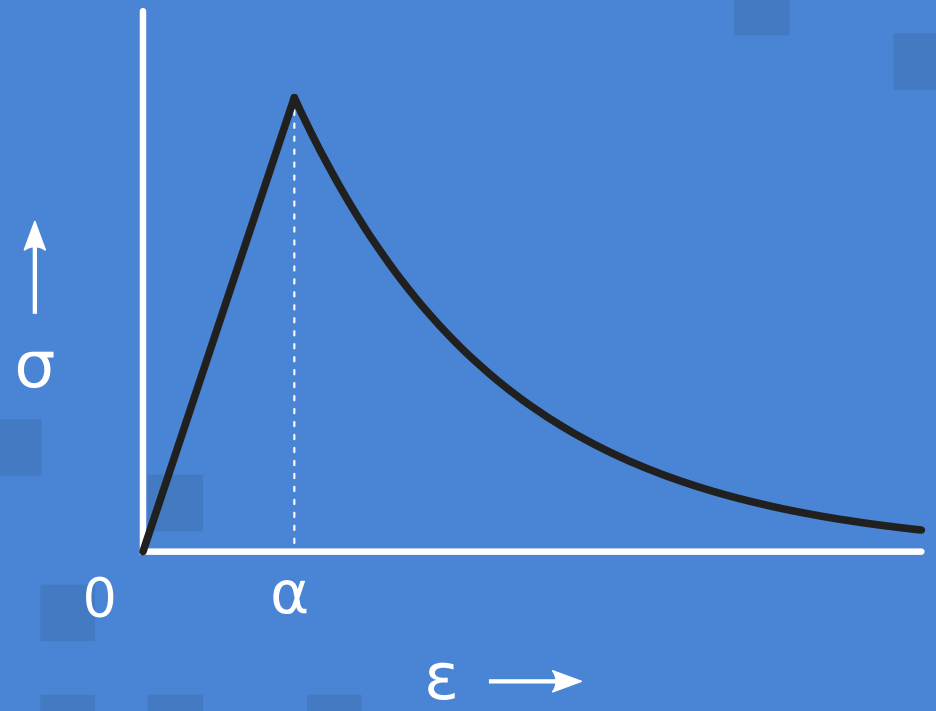
The parameter α is the value of the dilatation at which the damage starts to increase.

This model has the advantage that the stress-strain curve is a smooth function once the damage is non-zero.

The damage as function of the history parameter:



The strain-stress curve:



Program implementation

The two-dimensional version of the **damage** example program is located in the directory **solutions/damage-2**.

The implementation of the program is complete; there is no need to fill in the dots.

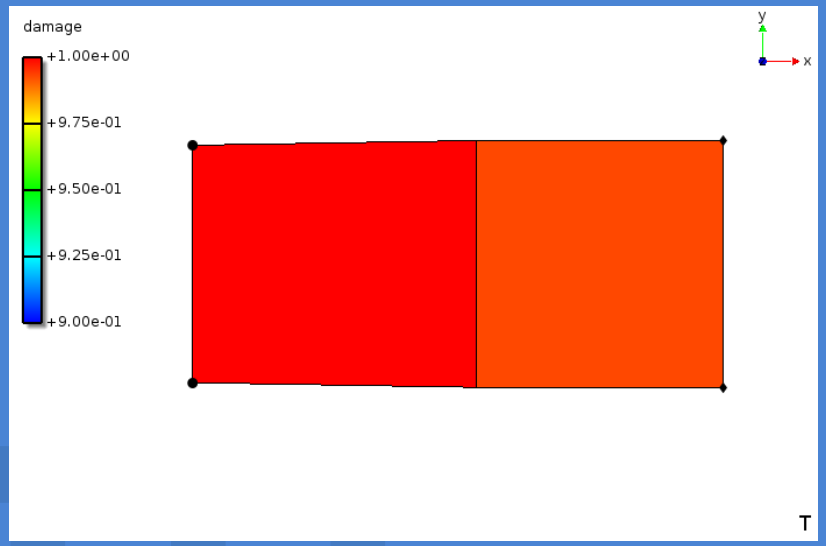
After the program has been compiled (with **make**), it can be run with one of the three sets of input files that are located in the same directory.

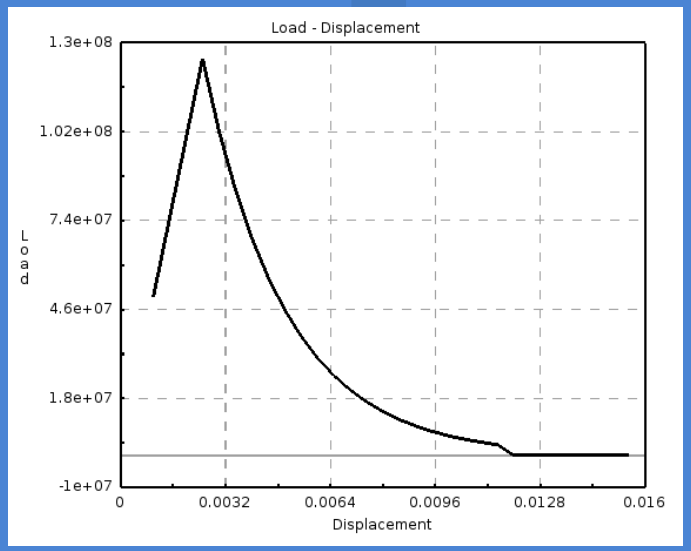
The first set of input files, **`simple.pro`** and **`simple.dat`**, model a two-element rectangular mesh subjected to a uniform displacement at the right edge.

The purpose of these input files is to verify that the implementation of the damage model yields the expected strain-stress curve.

Note that after about 20 load steps the damage pattern becomes non uniform due to small round off errors.

Results obtained with the `simple` input files:





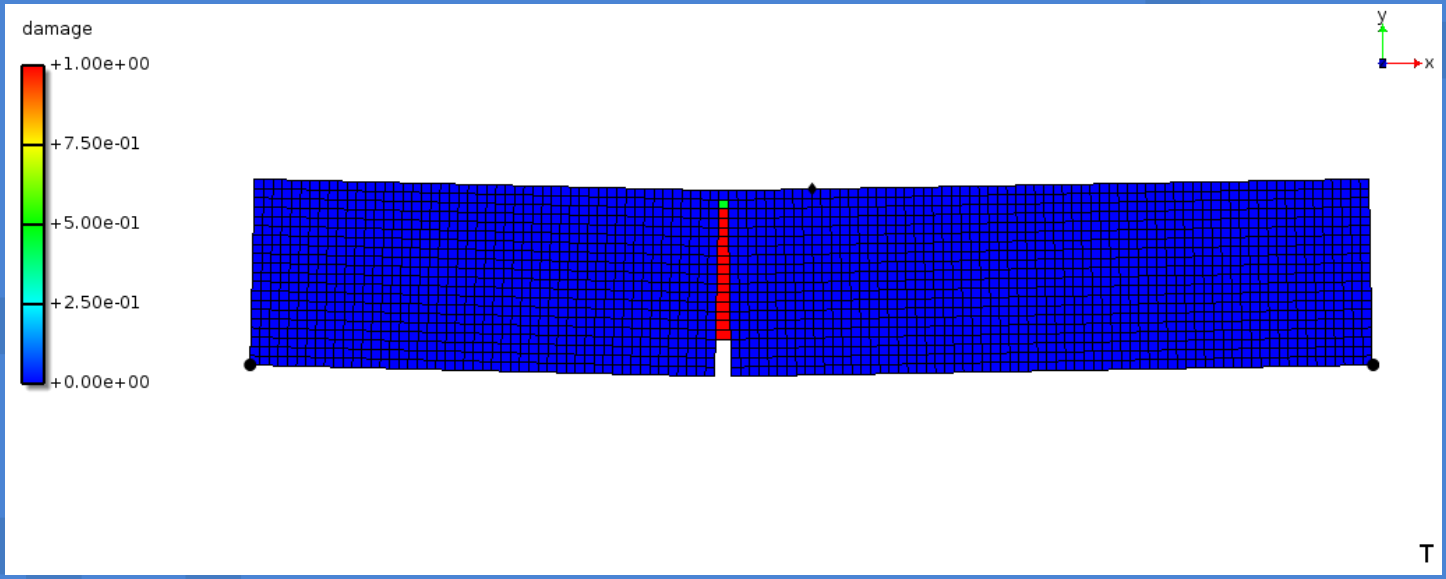
The second set of input files, **`beam.pro`** and **`beam.dat`**, model a three-point bending test.

The model involves a rectangular domain with a notch at which the damage evolution starts.

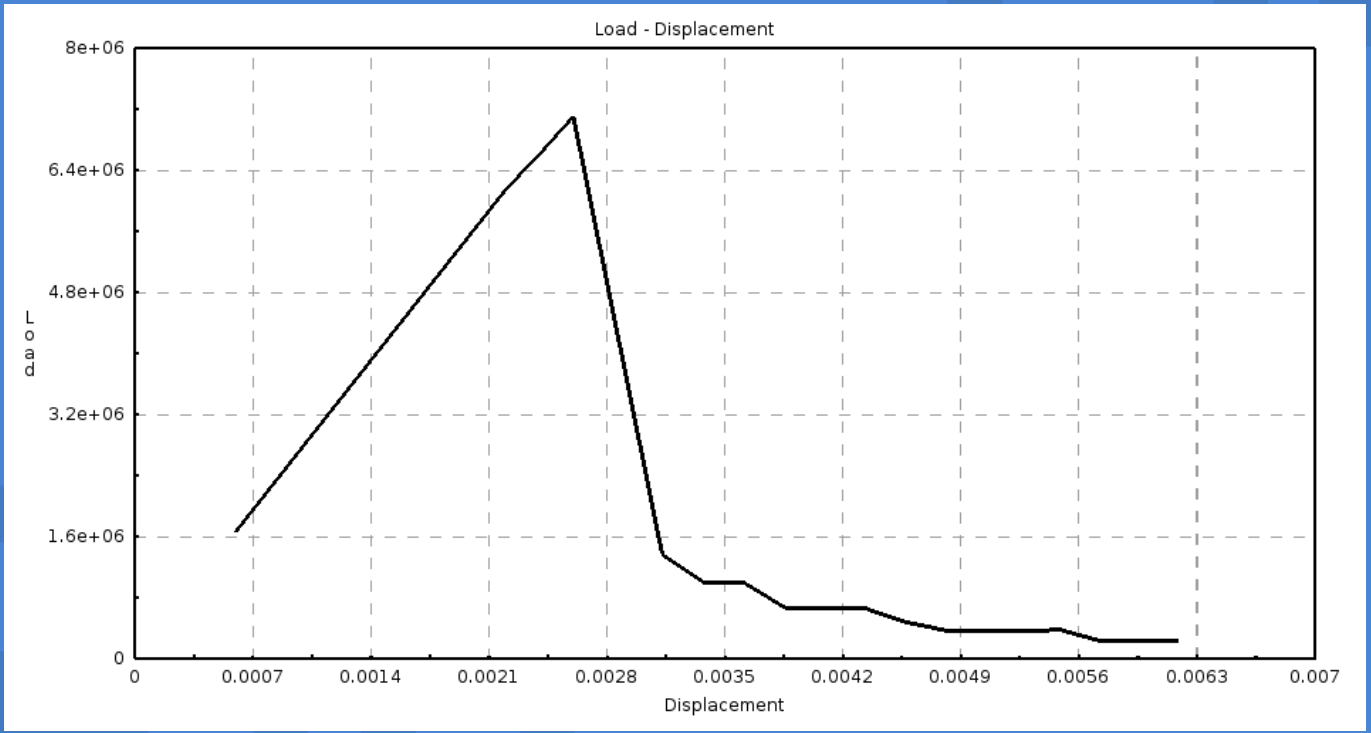
The non-linear solver is not able to follow the complete equilibrium path because that path exhibits snap-back behavior.

A more advanced path-following algorithm (such as the arc-length algorithm) is required to follow the complete path.

Damage distribution obtained with the **beam** input files:



Load-displacement graph obtained with the **beam** input files:



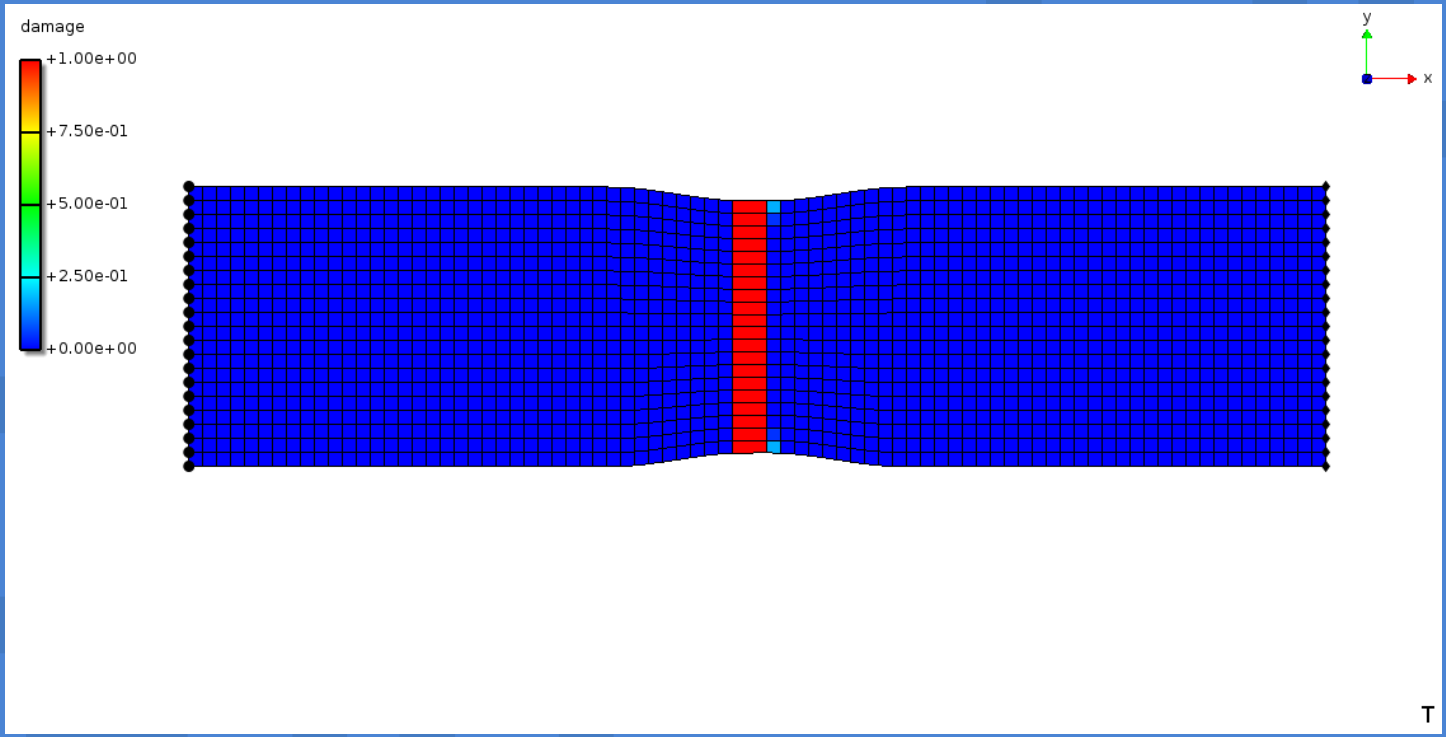
The last set of input files, **`impinge.pro`** and **`impinge.dat`**, model a rectangular domain with an impingement.

The rectangle is supported at the left edge, and subjected to an increasing displacement at the right edge.

Due to the local nature of the damage model, all the damage is concentrated in a one-element wide band in the middle of the impingement.

This model also exhibits a sharp snap-back behavior.

Damage distribution obtained with the **`impinge`** input files:



Note that the impingement has been created with a separate module named **`ImpingeModule`** that is part of the implementation of the **`damage`** program.

This module takes a mesh from the global database and applies a translation to the Y-coordinates of the nodes.

The **`ImpingeModule`** may serve as an example for creating similar modules that modify a mesh. For instance, a **`NotchModule`** that automatically inserts a notch in a mesh.

This is the END

my friend,

the very,

end.