# Developing numerical programs with Jem and Jive

**pre-course material**

Dynaflow Research Group

# Contents

# Introduction

This stack of slides will help you getting started with Jem and Jive, two C++ libraries aimed at developing numerical programs.

It is meant to be studied before starting the course *Developing numerical programs with Jem and Jive* from Dynaflow Research Group.

Its main goal is to make you more familiar with Jem/Jive so that you will be more productive during the course.

These slides explain what Jem and Jive are; how to get started with Jem/Jive; and how to use some essential components provided by Jem/Jive.

Do not worry if something is not clear because all information in these slides will come back during the course in more detail.

The pre-course material includes a number of exercises that will help you getting familiar with Jem/Jive. You are encouraged to try completing the exercises.

The solutions to the exercises are presented during the course.

The pre-course material also includes an example program that shows how to solve a (simple) partial differential equation with Jem and Jive.

The program computes the displacement field in a 1-D bar by means of the finite element method.

It will be familiar to those who have participated in the course *Programming in C++* from Dynaflow Research Group.

Because Jem and Jive are aimed at developing C++ programs, you must have a working knowledge of C++.

In particular, you must be familiar with basic concepts (including pointers and references), classes, overloading and templates.

You also need to understand the numerical algorithms you are going to implement or use.

*Note that solving more difficult problems requires more programming skills; Jive can not change that.*

In order to complete the exercises and run the example program, you must have installed a recent C++ compiler and Jem/Jive on your system.

On Linux and MacOS X you can use the GNU compiler (**`g++`**) or the LLVM compiler (**`clang++`**).

On Windows you can use the GNU compiler (**`g++`**) from the MinGW project or the Microsoft C++ compiler (**`cl`**) from Visual Studio.

If you do not want to use Visual Studio on Windows, you must also install the MSys package so that you can use **`make`** to build Jem/Jive programs.

When using **make** to build Jem/Jive programs, you must set the environment variables **JEMDIR** and **JIVEDIR** to the path names of the directories where Jem and Jive, respectively, are installed.

On Linux and MacOS X add something like the following lines to your shell startup script:

```
export JEMDIR=/path/to/jem/directory
export JIVEDIR=/path/to/jive/directory
```

On Windows, use the Control Panel to set the environment variables. Note that you should forward slashes (`/`) to separate path components.

For instance:

```
JEMDIR  : D:/JemJive/jem
JIVEDIR : D:/JemJive/jive
```

# Jem/Jive overview

# Jem/Jive in a nutshell

Jem and Jive form a C++ toolkit for building numerical applications.

Jive is ***not*** a ready-to-run program; you must implement your own program.

Jive is similar to Matlab, but uses C++ instead of a custom interpreter.

Jem and Jive can be used to solve linear and non-linear partial differential equations (PDE's). They help you:

- transform a PDE into a system of equations;
- solve that set of equations;
- compute quantities derived from the solution.

Jem/Jive provide most support for the finite element method, but other methods can certainly be used too.

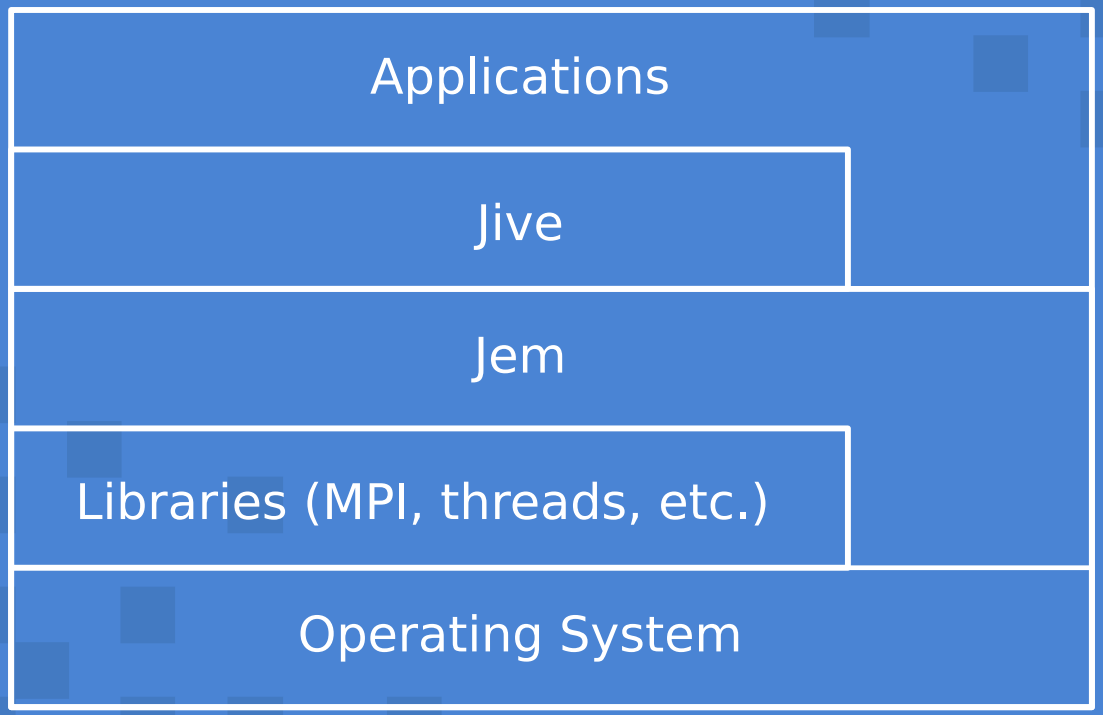
Jem is a generic C++ programming toolkit; it is a mix between the standard C++ library and the standard Java packages.

Jem provides a portable interface to system-level services and serves as the foundation for Jive.

Jive is specifically aimed at running numerical simulations, often involving large data sets.

Jive does not contain platform-specific code.

Design of Jem and Jive:



# Main features

Jem/Jive enable you to work on a high level where possible, and a low level where needed.

They use flexible and efficient data structures; degrees of freedom can be added on the fly.

Jive provides a flexible way for specifying input data and runtime parameters.

Two types of input files:

- data files, containing (large) data sets;
- property files, containing configuration data.

Data files are based on an XML syntax.

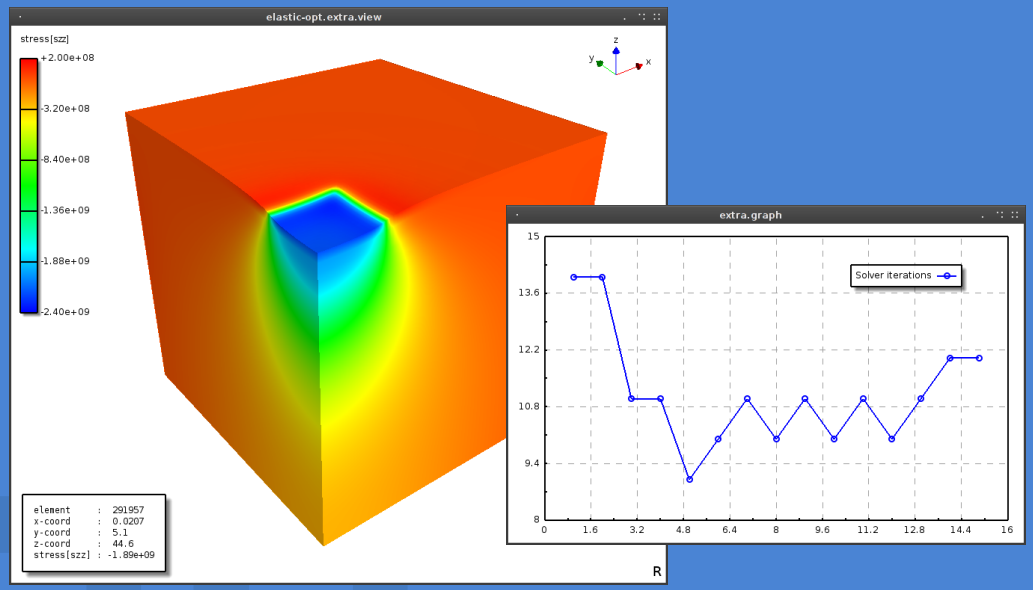
Property files are based on a C++-like syntax.

```
<Nodes>
  1 0.0 0.0;
  2 1.0 0.0;
  3 0.0 1.0;
  4 1.0 1.0;
</Nodes>

<Elements>
 1 1 2 3 4;
</Elements>

<NodeConstraints>
  dx[1] = 0.0
  dx[4] = 0.0
  dy[1] = 0.0
  dx[2] = dx[3]
</NodeConstraints>
```

```
linsolve =
{
  solver = "GMRES"
  {
    maxIter = 40;
  };
}

control =
{
  runWhile = "maxLoad < 2e6";
};

model = "Matrix"
{
  model = "Multi"
  {
    models = [ "elastic",
               "plastic",
                "load" ];

    // ...
  };
};
```

Jive implements advanced linear and non-linear solution algorithms.

Jive provides support for real-time graphical feedback.



# The programming interface

Jem and Jive provide a large collection of classes that can be combined into numerical applications.

Jem/Jive do not impose a pre-defined flow of execution; you must implement the `main` function.

Jive runs from the command line; it does not provide a graphical user interface.

Jem provides classes for:

- handling multi-dimensional (sparse) arrays;
- memory management and sharing of data;
- handling application configuration data;
- performing I/O operations;
- using system-level services in a portable way.

## Multi-dimensional array example:

```
Array<double,2>  a ( 4, 4 );
Array<double,2>  b ( 4, 4 );
Array<double,2>  c ( 4, 4 );
Array<double>    u ( 16 );
Array<int>       v ( 2 );

TensorIndex      i, j, k;

v[i]      = 2 + i;
b         = 0.0;
b(v,2)    = 1.0;
b(2,ALL)  = 2.0;
c         = sqrt ( b ) + 0.5 * b.transpose();
a(i,j)    = sum ( b(i,k) * c(k,j), k );
u         = flatten ( a );
```

Jive provides classes for:

- storing and handling (unstructured) meshes/grids;
- managing degrees of freedom;
- assembling and solving (large) systems of equations;
- handling constraints and other boundary conditions;
- helping find errors.

Jive also provides a large collection of classes for computing element shape functions, including:

- parametric shape functions;
- NURBS and T-Splines;
- 4-D (space-time) shape functions.

Shape functions can be evaluated within a domain and on a boundary.

# Getting started with Jem

# Organisation of Jem

Jem is composed of ***packages*** that bundle related components (classes, types, functions and constants).

Each package has its own namespace containing all classes, functions and other entities from that package.

The namespace of a package **`pkg`** is **`jem::pkg`**. One exception: the namespace of the **`base`** package is **`jem`**.

Header files from a package **`pkg`** are located in the include directory **`jem/pkg`**.

For instance, to include the header **`System.h`** from the package **`base`**, put the following directive in your code:

```
#include <jem/base/System.h>
```

Each package provides its own library that must be linked with a Jem program.

Essential packages in Jem:

**`base`**
: contains fundamental classes and functions.

**`io`** contains input/output components.

**`util`**
: contains utility components, including container classes.

**`numeric`**
: contains components for implementing numerical algorithms.

# Essential Jem classes

The **`System`** class provides output and logging functions.

The **`String`** class can be used for handling character strings.

The **`Array`** class provides support for Matlab/Fortran-style array expressions and is used extensively in Jem/Jive.

The **`Ref`** class is a kind of “smart pointer” that automatically deletes objects that are no longer being used.

The **`Properties`** class acts as a kind of in-memory database for storing arbitrary objects. It can also be used to manage configuration parameters.

The **`Reader`** and **`Writer`** classes represent textual input and output streams.

# How to use classes from Jem

To use a Jem class, you first need to include the header file containing the class definition.

The header file typically has the same name as the class. For instance, the class **`Reader`** is defined in the header file **`Reader.h`**.

The location of the header file is determined by the package containing the class. The class **`Reader`** is part of the **`io`** package so its header file is located in **`jem/io`**.

Optionally, put a using declaration in your code so that you can omit the namespace:

- if a class is used only in a few functions, then put the using declaration in those functions;
- if a class is used only in a few source files, then put the using declarations at the top of those files;
- if a class is used globally, then put the using declaration in a separate header file; see next slide.

Use ***forward declarations*** as much as possible, especially in header files.

Put common using declarations in a separate header file, for instance **`import.h`**.

Instead of repeating the same using declarations in multiple source files, you can simply include the file **`import.h`**.

Forward declarations are needed to make this work. Try not to include other header files.

## import.h

```
#ifndef IMPORT_H
#define IMPORT_H

// Forward declarations.

namespace jem
{
  class String;

  namespace io
  {
    class Reader;
    class Writer;
  }
}

// Using declarations.

using jem::String;
using jem::io::Reader;
using jem::io::Writer;

#endif
```

This is how the **`import.h`** header can be used:

**`input.cpp`**

```
#include "import.h"

void        readValue

   ( Reader&   in,
     String&   name,
     double&   value )

{
  String  dummy;

  in >> name >> dummy >> value;
}
```

**`output.cpp`**

```
#include "import.h"

void                 prinValue

   ( Writer&          out,
     const String&    name,
     double           value )

{
  out << name  << " = "
      << value << "\n";
}
```

Example source file using classes from Jem:

```
#include <jem/base/Array.h>
#include <jem/base/String.h>
#include <jem/base/System.h>
#include <jem/io/FileReader.h>
#include "import.h"

using jem::Array;

Array<String> readLines ( Reader& in )
{
  using jem::io::FileReader;

  // ...
}
```

Note that the class **`FileReader`** is derived from the **`Reader`** class. It is also located in the **`io`** package.

# Jem example program

The next slide shows the typical layout of a Jem program.

This example shows how to:

- include Jem header files;
- use using declarations;
- use some basic Jem classes;
- deal with exceptions.

```
#include <jem/base/String.h>
#include <jem/base/System.h>
#include <jem/base/Exception.h>

using jem::System;
using jem::String;
using jem::Exception;
using jem::io::endl;

int main ()
{
  try
  {
    String  name;
    int     age;

    System::out() << "Enter your name and age: ";
    System::in()  >> name >> age;
  }
  catch ( const Exception& ex )
  {
    System::err() << "ERROR: " << ex.what() << endl;
  }

  return 0;
}
```

## When using the standard C++ library:

```
#include <string>
#include <iostream>
#include <exception>

using std::string;
using std::exception;
using std::endl;

int main ()
{
  try
  {
    string  name;
    int     age;

    std::cout << "Enter your name and age: ";
    std::cin  >> name >> age;
  }
  catch ( const exception& ex )
  {
    std::cerr << "ERROR: " << ex.what() << endl;
  }

  return 0;
}
```

The two programs have a similar structure.

Main difference concerns the standard input/output streams.

In the standard library, the streams are global objects.

In Jem, the standard streams are obtained by calling a static member functions of the `System` class.

This difference has to do with support for parallel *threads* that is built into Jem.

# How to compile Jem programs

A Jem program can be built like any other C++ program.

First you need to invoke the C++ compiler on each source file. Then you need to run the linker to combine the object files and the Jem libraries into an executable.

You must specify the required compiler options, including the header file search path(s).

You can simplify this process by using **`make`**.

The program **`make`** reads a set of dependencies and rules from a text file called a ***makefile***.

The dependencies indicate which file depends on which other file(s). The rules indicate how to update a file when one of its dependencies has changed.

Jem can help you setting up a makefile by providing a collection of pre-defined makefiles that you can include in your own makefile.

First, make sure that you have GNU **`make`** (this is the default on Linux and MacOS X). Also make sure that the environment variable **`JEMDIR`** points to the Jem installation directory.

Create a file named **`Makefile`** in your source directory.

Specify the name of your program by setting the **`program`** variable.

Include all required package-specific makefiles. The makefiles can be found in the directory **`$(JEMDIR)/makefiles/packages`**.

Include the makefile **`$(JEMDIR)/makefiles/prog.mk`**.

Optionally, specify any sub-directories containing source files by setting the **`subdirs`** variable.

Example of a makefile:

**Makefile**

```
program = progName

include $(JEMDIR)/makefiles/packages/*.mk
include $(JEMDIR)/makefiles/prog.mk

subdirs = subdir1 subdir2 subdir3

MY_CXX_FLAGS = -O3
```

Note that the same makefile can be used on all platforms supported by Jem (including Windows).

Invoke **`make`** with a ***target*** like this:

```
make debug
```

Standard targets:

**`opt`** Build an optimised executable.

**`debug`**
: Build an executable for debugging.

**`clean`**
: Delete object files.

**`clean-all`**
: Delete object files and executable.

The makefiles provided by Jem will compile and link all source files with the extension `.cpp` in the directory containing your makefile.

One of the source files must contain the `main` function. You can not have multiple `main` functions.

If you want to exclude a source file you must move it to another directory or give its file name another extension.

When using an integrated development environment (IDE) like XCode, CodeBlocks or Visual Studio, you need to set up a project (or something similar) that indicates how to build your Jem program.

The exact details depend on the IDE. In general you need to specify the search paths for the Jem header files and library files.

In some cases you can specify that **`make`** should be used to build the program.

# The `System` class

The `System` class (package `base`) manages some global resources, such as the standard input and output streams. These are instances of the `Writer` class and are similar to `std::cin` and `std::cout`.

The `System` class also provides streams for writing log messages and progress information.

Note that the `System` class only has static member functions; you can not create an instance of the `System` class.

`System` member functions:

`out` Returns a reference to the standard output stream.

`in` Returns a reference to the standard input stream.

`log` Returns a reference to a log stream. An optional ***tag*** can be associated with a log message.

`exec`
Executes a function and catches exceptions and memory errors.

## Example use of the **System** class:

```
#include <jem/base/System.h>

using jem::System;
using jem::io::endl;

int run ( int argc, char** argv )
{
  int  age;

  System::log( "start" ) << "Starting " << argv[0] << endl;
  System::out() << "what is your age? ";
  System::in()  >> age;
  System::out() << "Your age is " << age << endl;

  return 0;
}

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

The member function **`exec`** calls another function and handles any exceptions and memory errors that occur during the execution of that function.

The function to be called must be passed as a parameter to the **`exec`** function.

Optionally, the command-line arguments can be passed to the **`exec`** function too. They will be forwarded to the function to be called.

The function to be called must return zero on success and a non-zero integer otherwise.

## Example with the **exec** function:

```
int run ()
{
  int  n;

  System::out() << "Enter a positive number: ";
  System::in()  >> n;

  return (n > 0) ? 0 : 1;
}

int main ()
{
  return System::exec ( run );
}
```

Any exception thrown during the execution of the function **run** will be handled by the **exec** function.

# Exercise: hello world

Implement a simple Jem program that prints the string “hello world” ten times to the standard output.

Use the **`System`** class for writing to the standard output. Create a makefile for compiling the program.

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

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

# The **`String`** class

The **`String`** class (package **`base`**) represents a (8-bit) character string.

Unlike the **`std::string`** class, a **`String`** is ***immutable***: its contents can not be modified after it has been created.

A **`String`** can be created from a literal string without overhead:

```
String  s = "this is a string";
```

The string **`s`** refers to the original string literal; no copy is made.

**`String`** objects can be copied with very little overhead by using an implicit data sharing scheme.

```
void example ( const String& s )
{
  String  t = s;  // t and s share the same data.

  // ...
}
```

Integers and floating point numbers can be easily converted to **`String`** objects.

```
void example ()
{
  String  s, t;

  s = String ( 101 );
  t = String ( 1.01 );
}
```

**`String`** member functions and operators:

**`size`**
: Returns the number of characters in a string.

**`find`**
: Returns the next location of a character or sub-string.

**`operator ==`**
: Checks whether two strings are equal.

**`operator +`**
: Concatenates two strings.

**`operator []`**
: Returns a character or sub-string.

A sub-string can be created by passing a **`Slice`** object to the **`[]`** operator. A slice object is created by the **`slice`** function.

Here are some examples:

```
void example ()
{
  String  s = "Hello World";

  System::out() << s[slice(1,5)]     << endl;  // "ello"
  System::out() << s[slice(BEGIN,5)] << endl;  // "Hello"
  System::out() << s[slice(6,END)]   << endl;  // "World"
  System::out() << s[slice(ALL)]     << endl;  // "Hello World"
}
```

| | |
|---|---|
| **`BEGIN`** | Denotes the start of the string. |
| **`END`** | Denotes the end of the string. |
| **`ALL`** | Denotes the entire string. |

If the second parameter of the **`slice`** function is an integer index, then the characters ***until*** that are index are selected; the specified index is not included in the selection.

For instance, the expression

```
str[slice(2,6)]
```

selects the sub-string from index 2 until (and not including) index 6; the length of the sub-string is (6 - 2) = 4.

## Example use of the **String** class:

```
#include <jem/base/String.h>
#include <jem/base/System.h>

using namespace jem;
using namespace jem::io;

String simplify ( const String& s )
{
  idx_t  i = s.find ( "one" );

  if ( i >= 0 )
  {
    return (s[slice(BEGIN,i)] + String( 1 ) + s[slice(i+3,END)]);
  }
  else
  {
    return s;
  }
}
```

# The `idx_t` type

Jem and Jive use the type `jem::idx_t` to represent indices and sizes of array-like objects.

The type `idx_t` is either an alias for `int` or `long int`, depending on how Jem has been configured.

By default, `idx_t` is an `int` (32-bit). If you need to handle very large data sets, then you can select `idx_t` to be a `long int` (64-bit).

By using `idx_t` you make your code more flexible.

The type `idx_t` is declared in the header file `<jem/defines.h>`.

Normally you do not have to include this header file explicitly beacuse it is included by all header files provided by Jem.

You only need to include `<jem/defines.h>` if you need to use the type `idx_t` and your code does not include any other header file from Jem.

# The `Array` class

Use the **`Array`** class (package **`base`**) for handling multi-dimensional rectangular arrays.

Because it is a class template, an **`Array`** can be used to store different types of elements.

The **`Array`** class mimics arrays from Matlab and Fortran 90.

The **Array** class has two template parameters. The first is the element type; the second the number of dimensions (one by default).

The array elements can be accessed with the `[]` operator (1-D arrays) or the `()` operator (multi-dimensional arrays).

```
#include <jem/base/Array.h>

void example ()
{
  using jem::Array;

  Array<int>       a ( 2 );
  Array<float,2>   b ( 2, 2 );
  Array<bool,3>    c ( 2, 2, 2 );

  a[0]      = 1;
  b(0,0)    = 2.0F;
  c(0,0,0)  = true;
}
```

**`Array`** member functions:

**`size`**
: Returns extent of an array in a given dimension.

**`resize`**
: Changes the shape of an array, discarding its current contents.

**`reshape`**
: Changes the shape of an array, keeping its current contents.

**`operator []`**
: Access an element of a one-dimensional array.

**`operator ()`**
: Access an element of a multi-dimensional array.

**`Array`** member functions (continued):

**`operator =`**
: Sets all array elements.

**`operator =`**
: Copies another array. The array shapes must match.

**`ref`** Creates a shallow copy of another array.

**`transpose`**
: Exchanges the dimensions of an array.

**`clone`**
: Creates a deep copy of an array.

## Example use of the **Array** class.

```
void example ()
{
  Array<double>    v ( 5 );
  Array<double,2>  a, b;

  a.resize ( 10, v.size() );

  v = 1.0;

  for ( idx_t i = 0; i < a.size(0); i++ )
  {
    for ( idx_t j = 0; j < a.size(1); j++ )
    {
      a(i,j) = v[j];
    }
  }

  b = a.transpose ();

  v.reshape ( v.size() + 1 );

  v[v.size() - 1] = 2.0;
}
```

# Data sharing

An array has a ***shape*** and a ***data block***. The shape determines the extent in each dimension. The data block stores the array elements.

Data blocks may be shared between multiple arrays.

The copy constructor creates a ***shallow*** copy and the assignment operator creates a ***deep*** copy. Use the `ref()` member function to create a shallow copy.

With a shallow copy, the two arrays point to the same data block.
With a deep copy, the contents are copied between data blocks.

When the shape of an array is changed, it allocates a ***new*** data block.

The `clone` member function returns a deep copy of an array.

A data block is deleted when the last array associated with that block is destroyed.

Here is an example that shows how data blocks are shared:

```
void example ()
{
  Array<int>  a; // ......... Empty array; no data block.

  a.resize ( 10 ); // ....... Allocates a data block containing
                   //         ten integers.
  {
    Array<int>  b = a; // ... Copy constructor; b points to the
                       //     data block of a.

    b = 3; // ............... Sets all elements of a and b to 3.

  } // ..................... Destroys b; the data block is kept
    //                       by a.

} // ......................... Destroys a; the data block is
  //                         destroyed too.
```

## The same example, but without sharing:

```
void example ()
{
  Array<int>  a; // ........ Empty array; no data block.

  a.resize ( 10 ); // ....... Allocates a data block containing
                   //         ten integers.
  {
    Array<int>  b; // ....... Empty array; no data block.

    b.resize ( 10 ); // ..... Allocates another data block
                     //       containing ten elements.

    b = a; // ............... Copies all elements from the data
           //                 block of a to the data block of b.

  } // ..................... Destroys b; its data block is
    //                       destroyed too.

} // ....................... Destroys a; its data block is
  //                         destroyed too.
```

## More sharing of data between arrays:

```
void example ()
{
  Array<int>  a ( 10 );
  Array<int>  b = a;           // Creates a shallow copy.
  Array<int>  c = a.clone();   // Creates a deep copy.

  a = 1;                       // Sets both a and b.

  a.resize ( 8 );              // Creates new data block.

  a = 2;                       // Sets only a.

  c.ref ( a );                 // Shallow copy.

  c = 3;                       // Sets both a and c.
}
```

# Exercise: arrays

Implement a simple Jem program that reads a series of positive integers from `System::in()` and stores those in an array.

The program should stop reading when it encounters a non-positive number.

Exercise location: `exercises/array`.

Solution location: `solutions/array`.

## Element-wise array expressions

All unary and binary operators can be applied to arrays; the operators are applied ***element wise***. All arrays in an expression must have the same shape.

The overloaded operators are implemented with ***expression templates***; no temporary arrays are created.

Most intrinsic math functions, such as **`abs`** and **`sin`**, can be used with arrays. These functions are also applied element wise.

## Example of array expressions:

```
void example ()
{
  Array<double>  a ( 10 );
  Array<double>  b ( 10 );
  Array<double>  c ( 10 );
  Array<bool>    v ( 10 );

  for ( idx_t i = 0; i < a.size(); i++ )
  {
    a[i] = i;
  }

  b = 2 * a + sin( a ) - 1.0 / a;
  c = where ( abs( b ) > 0.5, b, 0.0 );
  v = (c > 0.0);
}
```

# Sub-arrays

Use ***slices*** to create sub-arrays. There are different types of slices that can be created conveniently with the `slice` function.

A sub-array is represented by a normal `Array` object that points to the same data block as the original `Array`.

Use ***select*** expressions to create non-regular sub-arrays.

Use the `[]` operator on a multi-dimensional array to create a slice in the last dimension.

## Example of sub-arrays:

```
void example ()
{
  Array<double,2>  a  ( 10, 20 );
  Array<idx_t,1>   ix ( 3 );

  ix[0] = 2;
  ix[1] = 3;
  ix[2] = 7;

  a(ALL,3)              = 1.0;
  a[3]                  = 1.0; // Same as statement above.
  a(slice(BEGIN,3),2)   = 2.0;
  a(ALL,slice(2,END))   = 3.0;

  select(a,ix,ix)      += 4.0;
  select(a,ALL,ix)      = 5.0;
  a[ix]                 = 5.0; // Same as statement above.
}
```

## Array-related functions

**`min`** Returns the minimum value in an array.

**`max`** Returns the maximum value in an array.

**`sum`** Returns the sum of the values in an array.

**`dot`** Returns the dot product of two arrays or of an array with itself.

**`count`**
: Counts the number of array elements for which a given expression is true.

**`testany`**
: Checks whether at least one array element is true.

**`testall`**
: Checks whether all array elements are true.

**`reshape`**
: Changes the shape and number of dimensions of an array.

**`sort`**
: Sorts the elements in an array.

## Example of array functions:

```
void example ()
{
  Array<double>     a ( 10 );
  Array<double,2>   b = reshape ( a, 2, 5 );
  double            x;
  idx_t             k;

  a    =  1.0;
  b[2] = -1.0;                  // Also sets a[4] and a[5].
  k    = count ( a < 0.0 );     // k == 2.

  if ( testany( a < 0.0 ) )
  {
    x = dot ( a );
  }
  else
  {
    x = dot ( b, b );
  }

  sort ( a );
}
```

# Tensor expressions

Use ***tensor expressions*** to replace hand-written for-loops.

A tensor expression involves the use of the **`TensorIndex`** class as a dummy array index.

The expression on the left of the = operator determines the bounds of the tensor indices. Example:

```
Array<double,2>  a ( 3, 5 );
Array<double,2>  b ( 5, 3 );
TensorIndex      i, j;

a(i,j) = 1.0 + i + j;
b(i,j) = a(j,i);
```

The functions **min**, **max**, **dot** and **sum** can performs a reduction in a given dimension.

You must specify the **TensorIndex** over which the reduction is to be performed.

Example:

```
Array<double,2>  a ( 10, 20 );
Array<double,1>  u ( 10 );
Array<double,1>  v ( 20 );
TensorIndex      i, j;


u[i] = dot ( a(i,j), v[j],  j );  // Matrix-vector multiplication.
v[j] = sum ( u[i] * a(i,j), i );  // Vector-matrix multiplication.
u[i] = max ( abs( a(i,j) ), j );
```

# Header files

The **`Array`** class and all related functions and operators are defined in the header file **`jem/base/Array.h`**.

This file contains a ***lot*** of template code and including it will slow down the compilation of your source code.

Note: to use tensor expressions you ***must*** include the extra header file **`jem/base/array/tensor.h`**.

To speed up compilation, include smaller header files:

**`jem/base/array/Array.h`**
: Contains the definition of the **`Array`** class.

**`jem/base/array/operators.h`**
: Contains the overloaded operators.

**`jem/base/array/intrinsics.h`**
: Contains the overloaded intrinsic functions.

**`jem/base/array/utilities.h`**
: Contains the reduction functions and other utility functions.

**`jem/base/array/select.h`**
: Contains the implementation of select expressions.

# Exercise: matrix-matrix multiplication

Implement a matrix-matrix multiplication in three ways: using nested for-loops; using array slices and the `dot()` function; and using tensor expressions.

Test which implementation is fastest.

Exercise location: `exercises/matmul`.

Solution location: `solutions/matmul-1`.

# Const-qualified arrays

Most `Array` member functions and operators are const qualified.

Only member functions that change the shape or the address of the data block associated with an `Array` are not const qualified.

This means that the copy assignment operator and the subscript operator are const qualified.

***It is possible to modify the contents of a const-qualified array***.

## Example with const-qualified arrays:

```
void example ( const Array<int>& a )
{
  Array<int>  b ( a.size() );

  b    = 2;
  a    = b;          // OK
  a[4] = 3;          // OK

  a.resize ( 8 );    // Error: changes shape.
  a.ref    ( b );    // Error: changes data block.
}
```

The advantage of this design is that it enables operations to be performed on sub-arrays.

```
void fill ( const Array<double>& a );

void example ()
{
  Array<double>  a ( 10 );

  a              = 0.0;
  a[slice(2,6)] = 3.0;       // OK because assignment operator
                             //    is const qualified.

  fill ( a[slice(4,END)] ); // OK because array parameter
                             //    is const qualified.
}
```

A disadvantage is that a function declaration can not indicate whether the contents of an array are modified.

# The **Ref** class

Use the **Ref** class (package **base**) for automatic memory management. It acts like a pointer to an object; you can apply the usual ***** and **->** operators.

A **Ref** object not pointing to any object has the value **NIL**, similar to a null pointer.

```
Ref<Number>  num;  // Null pointer.
```

A **Ref** instance must point to objects derived from the **Collectable** class.

These objects must be created with the function **newInstance**; this is similar to the **new** operator.

```
Ref<Number>  num = newInstance<Number> ( 1 );
```

Arguments passed to **newInstance** are passed to the constructor of the object to be created.

# Example of the **Ref** class:

```
class Example : public Collectable
{
 public:
          Example  ( int n );
  int     getValue () cons;
};

void doit ( Example& e );

void example ()
{
  Ref<Example>  e = newInstance<Example> ( 1 );

  if ( e != NIL )
  {
    System::out() << "value = " << e->getValue() << endl;

    doit ( *e );
  }
}
```

A **Collectable** object will automatically be deleted if there are no more **Ref** objects pointing to it. Example:

```
void example ()
{
  Ref<Example>  e = newInstance<Example> ( 1 );

  e = newInstance<Example> ( e->getValue() + 1 );
}
```

Same code with normal pointers:

```
void example ()
{
  Example*  e = new Example ( 1 );
  Example*  t = new Example ( e->getValue() + 1 );

  delete e;
  e = t;
  delete e;
}
```

Use the copy assignment operator to copy a **Ref** to another **Ref** of a compatible type.

Use the functions **staticCast** and **dynamicCast** to cast a **Ref** object to a different type. These functions are similar to the **static_cast** and **dynamic_cast** operators.

```
void example ()
{
  Ref<Example>      e = newInstance<Example> ( 1 );
  Ref<Collectable>  c;

  c = e;                                  // OK
  e = c;                                  // Error
  e = dynamicCast<Example> ( c );  // OK
}
```

# Exercise: data sharing

Implement two classes that share data through a third class.
See the next slides for the details.

The idea is to mimick a common pattern that is used in Jive.

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

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

# Details

Implement the classes **SharedMatrix**, **Builder** and **Solver**.

The class **SharedMatrix** should be derived from the **Collectable** class and have a (public) member of type **Array<double,2>**.

```
class SharedMatrix : public jem::Collectable
{
 public:
  Array<double,2>   matrix;
};
```

The class **Builder** stores a reference to a **SharedMatrix** object. It should look like this:

```
class Builder
{
 public:
  void                  build ();

 private:
  Ref<SharedMatrix>  matrix_;
};
```

The member function **build** should store some values in the **matrix** member of the **SharedMatrix** object.

You must also implement a constructor for the **Builder** class.

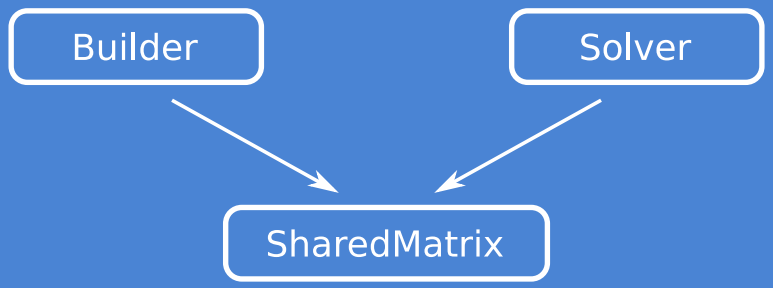
The class **Solver** also stores a reference to a **SharedMatrix** object. It should look like this:

```
class Solver
{
 public:
  void                  solve ();

 private:
  Ref<SharedMatrix>  matrix_;
};
```

The member function **solve** should print the values in the **matrix** member of the **SharedMatrix** object.

You must also implement a constructor for the **Solver** class.

Implement a **`test`** function that creates a **`Builder`** object and a **`Solver`** object that refer to the ***same*** **`SharedMatrix`** object.




Call the **`build`** member function on the **`Builder`** object and then the **`solve`** member function on the **`Solver`** object. Do you get the expected result?

You can put all class and member function definitions in one source file.

How can you verify that the `SharedMatrix` object is destroyed when it is no longer being used?

Hint: implement a destructor.

# The `Properties` class

Use the **`Properties`** class (package **`util`**) to store objects with different types in a tree structure.

Each object in a **`Properties`** object has a name (a **`String`**); different objects must have different names.

A (name,object) pair is called a ***property***. The type of the object is said to be the property type.

Use the **`set`**, **`get`** and **`find`** member functions to store/retrieve data into/from a **`Properties`** object.

An error is raised if a property type does not match the requested destination type. For instance, if a property is retrieved as an integer while the property type is a string.

The **`Properties`** is mainly used as a flexible, hierarchical database, and to manage configuration data and application data.

**`Properties`** members and operators:

**`size`**
: Returns the number of stored objects.

**`set`** Stores an object with a given name.

**`get`** Returns an object given a name; the object must exist. An exception is thrown if the property does not exist.

**`find`**
: Returns an object given a name if the object exists.

**`clone`**

Returns a deep copy of a **`Properties`** object.

**`operator []`**

Sets or gets an object with a given name.

**`operator =`**

Creates a shallow copy of a **`Properties`** object.

# Example of the **Properties** class:

```
#include <jem/util/Properties.h>

void example ()
{
  using jem::util::Properties;

  double       x = 0.0;
  int          i = 1;

  Properties  props;

  props.set  ( "first",  i );
  props.set  ( "second", x );

  props.get  ( i, "first" );     // OK
  props.get  ( x, "third" );     // Error: does not exist.
  props.find ( x, "third" );     // OK
  props.find ( i, "second" );    // Error: type mismatch.

  props["third"] = i;            // Same as set().
  i = props["third"];            // Same as get().
}
```

A properties set can also contain 1-D arrays.

Note that shallow array copies are stored and returned.

Example:

```
void example ()
{
  Properties  props;
  Array<int>  a ( 10 );
  Array<int>  b;

  a = 1;
  props.set ( "array", a );
  props.get ( b, "array" );

  b[2] = 2;  // Also changes a[2]!
}
```

The **get** and **find** member functions can be passed optional bounds for the value to be retrieved.

Example:

```
void example ()
{
  Properties  props;
  Array<int>  a ( 10 );
  int         i;

  a = 1;
  i = 3;

  props.set ( "first",  a );
  props.set ( "second", i );

  props.get ( a, "first",  0, 1 );  // OK
  props.get ( i, "second", 0, 1 );  // Error: out of bounds.
}
```

A **Properties** object can also contain a **Properties** object. This is called a ***sub-properties*** object.

Properties in a sub-properties object can be accessed by using a '.' (dot) in their names. Example:

```
void example ()
{
  Properties  main, sub;
  int         i;

  main.set ( "sub.i", i );
  main.get ( i, "sub.i" );
  main.get ( sub, "sub" );
  sub .get ( i, "i" );
}
```

When setting a property with a '.' in its name, sub-properties are created automatically.

Properties can be read from a text file by calling the **parseFile** member function.

They can be written to a text file by calling the **printTo** member function.

A **Properties** object can also be printed with the << operator.

```
void example ()
{
  Properties  props;
  int         i;

  props.parseFile ( "input.pro" );
  System::out() << props << endl;
  props.get ( i, "i" );
}
```

## Elaborate example; setting properties:

```
void example ()
{
  Ref<PrintWriter>  out;
  Properties        dbase;
  Properties        person;

  dbase.set  ( "dave.job", "dentist" );
  dbase.set  ( "dave.age", 34 );
  dbase.set  ( "dave.length", 1.77 );

  person.set ( "job", "cleaner" );
  person.set ( "age", 22 );
  person.set ( "length", 1.93 );

  dbase.set  ( "joe", person );

  // Save the properties to a file.

  out = newInstance<PrintWriter> (
    newInstance<FileWriter> ( "people.data" )
  );

  dbase.printTo ( *out );
}
```

## Example properties file:

```
// People database.

dave.job    = "dentist";
dave.age    = 34;
dave.length = 1.77;

joe =
{
  job    = "cleaner";
  age    = 22;
  length = 1.93;
};
```

## Getting properties:

```
void example ()
{
  Properties  dbase;
  Properties  person;

  String      job;
  int         age;

  dbase.parseFile ( "people.data" );

  dbase.find ( job, "dave.job" );
  dbase.get  ( age, "dave.age", 0, 200 );

  dbase.get  ( person, "joe" );

  person.find ( job, "job" );
  person.get  ( age, "age", 0, 200 );
}
```

# Exercise: read configuration data

Use a **`Properties`** object to store and read configuration data for an application.

Implement some dummy algorithm in the **`main`** function that uses some of the configuration data.

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

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

# Hints

Create a text file named `input.pro` (or whatever name you like) containing the configuration data.

Use the member function `parseFile` to read that file and store the configuration data in a `Properties` object.

You can try to experiment with different types of properties, including sub-properties.

# The `Reader` class

The `Reader` class (package `io`) represents a text input stream.

It provides various (virtual) member functions for reading and converting text from an input source, such as the standard input.

Various classes in Jem implement an overloaded `>>` operator for reading a class instance from a `Reader` object.

This operator is also overloaded for fundamental types.

**`Reader`** member functions:

**`read`**
: Extracts and returns the next character (as an **`int`**) from the stream, or **`-1`** if the end of the stream has been reached.

**`pushBack`**
: Pushes a character back into the stream. This essentially undoes a previous call to **`read`**.

**`skipWhite`**
: Skips over the next sequence of whitespace characters.

**`readLine`**
: Extracts and returns the next line from the stream.

The **Reader** class is an abstract base class; it can not be instantiated.

You must instantiate a derived class such as the **FileReader** class.

Because the **Reader** class is derived from the **Collectable** class you must use the **Ref** class to keep a reference to a **Reader**; you can not declare a **Reader** variable.

The next slide shows an example in which a **`FileReader`** is used to read an integer or a string from a file.

The function **`std::isdigit`** is used to determine whether to read an integer or string.

Note that a **`Ref<FileReader>`** can be assigned to a **`Ref<Reader>`** because a pointer to a derived class can be implicitly converted to a pointer to a base class.

```
#include <cctype>
#include <jem/io/FileReader.h>

void example ()
{
  using namespace jem;
  using namespace jem::io;

  Ref<Reader>  input;
  String       str;
  int          i, c;

  input = newInstance<FileReader> ( "input.dat" );

  input->skipWhite ();
  c = input->read ();

  if ( std::isdigit( c ) )
  {
    input->pushBack ( c );
    *input >> i;
  }
  else
  {
    *input >> str;
  }
}
```

Note the use of the dereference operator ***** in the statement

```
*input >> str >> i;
```

The dereference operator is required because the type **Ref<Reader>** represents a kind of pointer and the overloaded operator **>>** expects a reference.

# The `Writer` class

The **`Writer`** class (package **`io`**) represents a text output stream; it is the “twin” of the **`Reader`** class.

It provides various (virtual) member functions for writing text to an output destination, such as the standard output.

Various classes in Jem implement an overloaded `<<` operator for writing a class instance to a **`Writer`** object.

This operator is also overloaded for fundamental types.

`Writer` member functions:

**`write`**
: Writes one or more characters to the stream.

**`flush`**
: Writes all buffered data to the destination. This function is implicitly called by the destructor.

**`close`**
: Closes this stream. All subsequent write operations will lead to an exception.

Like the **`Reader`** class, the **`Writer`** class is an abstract base class; it can not be instantiated.

You must instantiate a derived class such as the **`FileWriter`** class.

You must use the **`Ref`** class to store a pointer to a **`Writer`**; you can not declare a **`Writer`** variable.

## Example of the **Writer** class:

```
#include <jem/io/FileWriter.h>

void example ()
{
  using namespace jem;
  using namespace jem::io;

  Ref<Writer>  output = newInstance<FileWriter> ( "output.dat" );
  String       str    = "this is a string";
  int          i      = 5;

  *output << "str = " << str << endl;
  *output << "i   = " << i   << endl;
}
```

The class **`PrintWriter`** (also from the **`io`** package) provides more control over the layout of the output.

A **`PrintWriter`** does not determine the destination of the text output stream. It needs to be wrapped around a **`Writer`** object.

The **`PrintWriter`** class has a public data member named **`nformat`** that controls the way that numbers are formatted.

The next slide shows an example in which a **`PrintWriter`** is wrapped around the standard output.

```
#include <jem/base/System.h>
#include <jem/io/PrintWriter.h>

void example ()
{
  using namespace jem;
  using namespace jem::io;

  Ref<PrintWriter>  prn;
  double            x;
  int               i;

  prn = newInstance<PrintWriter> ( &System::out() );
  x   = 2.2;
  i   = 6;

  prn->nformat.setScientific     ( true );
  prn->nformat.setFractionDigits ( 10 );
  prn->nformat.setIntegerWidth   (  8 );

  *prn << "x = " << x << endl;
  *prn << "i = " << i << endl;
}
```

# Getting started with Jive

# Organisation of Jive

Jive is organised in a similar way as Jem. It is composed of ***packages*** that bundle related components (classes, types, functions and constants).

Each package has its own namespace containing all classes, functions and other entities from that package. The namespace of a package **`pkg`** is **`jive::pkg`**.

Header files from a package **`pkg`** are located in the include directory **`jive/pkg`**.

For instance, to include the header file **`Table.h`** from the package **`util`**, put the following directive in your source file:

```
#include <jive/util/Table.h>
```

Each package provides its own library that must be linked with a Jive program.

Essential Jive packages:

**`util`**
: contains classes for storing (large) data sets.

**`fem`** provides general support for finite element applications.

**`algebra`**
: contains classes for building (large) matrices and for performing linear algebra operations.

**`solver`**

contains components for solving linear systems of equations.

**`geom`**

contains classes for evaluating shape functions and for executing geometrical operations.

# Mutable vs immutable classes

Many families of Jive classes come in two versions: an ***immutable*** and a ***mutable*** version.

The immutable version only provides functions for reading data from an instance of the class, while the mutable version adds functions for modifying an instance.

By convention, the name of the mutable version equals the name of the immutable version with the prefix character ‘X’.

For instance, the class **`XPointSet`** is the mutable version of the class **`PointSet`**.

Another example: the class **`XTable`** is the mutable version of the **`Table`** class.

The mutable class is always derived from the immutable class, and adds member functions for modifying an object instance.

## Example of an immutable and a mutable class:

```
class PointSet : public ItemSet
{
 public:
  virtual void getPointCoords ( const Vector&  coords,
                                idx_t          ipoint ) const;

  // ...
};

class XPointSet : public PointSet
{
 public:
  virtual void setPointCoords ( idx_t          ipoint,
                                const Vector&  cooords );

  // ...
};
```

The types **`Vector`** and **`ItemSet`** are described later.

An immutable class makes it possible to implement a derived class that represents a mathematical function instead of storing a large data set.

For instance, one could implement a **`Grid`** class, derived from the **`PointSet`** class, that calculates the coordinates of a grid point on the fly.

## Example implementation of the `Grid` class:

```
class Grid : public PointSet
{
 public:
  virtual void getPointCoords ( const Vector&  coords,
                                idx_t          ipoint ) const;
  // ...

 private:
  int           m_,  n_;    // Number of grid points.
  double        dx_, dy_;   // Grid spacing.
};

void Grid::getPointCoords ( const Vector&  coords,
                            idx_t          ipoint ) const
{
  int  i = ipoint / m_;
  int  j = ipoint % m_;

  coords[0] = i * dx_;
  coords[1] = j * dy_;
}
```

The `Grid` class requires only a trivial amount of memory.

This can make a significant difference when solving very large problems.

You should use immutable classes in your (member) functions and classes whenever that is possible. This expands the scope in which those functions and classes can be used.

# Function call conventions

Whenever a (member) function from Jive returns data through function parameters, those parameters are listed first.

That is, the output parameters are listed before the input parameters.

For instance, the member function:

```
bool findValue ( double& v, int i, int j ) const;
```

returns a value in the first parameter **v**.

When data are returned through an array parameter, the caller of the function must ensure that the array has the correct shape; the shape of the array is usually not modified in the function.

Consider this code:

```
void example ( const ElementSet& elems, idx_t ielem )
{
  IdxVector  inodes;

  elems.getElemNodes ( inodes, ielem );
}
```

The member function **`getElemNodes`** returns data in the array **`inodes`**. In this case it will raise an error because the array is empty.

The correct way of calling the function is:

```
void example ( const ElementSet& elems, idx_t ielem )
{
  IdxVector  inodes ( elems.getElemNodeCount( ielem ) );

  elems.getElemNodes ( inodes, ielem );
}
```

Here the function **`getElemNodeCount`** is used to determine the size of the **`inodes`** array.

There are two advantages of the caller being responsible for setting the correct array shape.

The first advantage has to do with performance: the overhead of allocating an array may be spread out over multiple function calls.

For instance, if all elements have four nodes:

```
void example ( const ElementSet& elems )
{
  IdxVector  inodes ( 4 );

  for ( idx_t ielem = 0; ielem < elems.size(); ielem++ )
  {
    elems.getElemNodes ( inodes, ielem );

    // ...
  }
}
```

The array **`inodes`** only needs to be allocated once for all elements.

The second advantage has to do with flexibility: the caller of a function may store the output data within a larger array.

Example:

```
void example ( const ElementSet& elems, idx_t ielem, idx_t jelem )
{
  const idx_t  k = elems.getElemNodeCount ( ielem );
  const idx_t  n = elems.getElemNodeCount ( jelem );

  IdxVector    inodes ( k + n );

  elems.getElemNodes ( inodes[slice(BEGIN,k)], ielem );
  elems.getElemNodes ( inodes[slice(k,END)],   jelem );
}
```

This collects the nodes of two elements in one array. Only one array allocation is required.

# Essential Jive components

The header file **<jive/Array.h>** provides a number of typedefs for common types of arrays. These types are declared in the namespace **jive** and they are used by many Jive classes and functions.

**Array.h**

```
typedef Array<double,1>  Vector;
typedef Array<idx_t,1>   IdxVector;
typedef Array<String,1>  StringVector;
typedef Array<double,2>  Matrix;
typedef Array<idx_t,2>   IdxMatrix;
typedef Array<double,3>  Cubix;
```

The `ItemSet` class represents a homogeneous set of ***items***. An item can be a point, an element, a volume, a particle, or some other type of object.

An item has an index and an identifier. The first ranges from zero to the number of items minus one, and the second is an arbitrary, unique integer.

An item has other data too, depending on its specific type.

The `ItemSet` class is the base class for all classes that describe a discrete representation of space and/or time.

The **`PointSet`** and **`XPointSet`** classes represent a collection of points (or nodes) in an $n$-dimensional space. The number of dimensions is limited only by the range of an integer.

Both these classes are derived from the **`ItemSet`** class.

The **`GroupSet`** class, also derived from **`ItemSet`**, stores topology information; each item in a **`GroupSet`** is a group of items in another group set.

The **`GroupSet`** class, and its mutable version **`XGroupSet`**, are used to store element connectivities, for example.

The **`DofSpace`** class maps degrees of freedom (DOFs) to a contiguous range of indices or equation numbers. It plays an important role when assembling a system of equations.

A **`DofSpace`** must be associated with an **`ItemSet`**; the DOFs are attached to the items in that set.

For instance, if a **`DofSpace`** is associated with a **`PointSet`**, then the DOFs are attached to the points/nodes in that set.

The **`Constraints`** class stores linear constraints between DOFs. It can be used to specify different types of boundary conditions.

Note that Jive applies constraints when a system of equations is solved, and not when that system is assembled.

The class **`MatrixBuilder`** can be used to assemble a (large) matrix. There are various derived classes that make different trade-offs between speed and memory usage.

The **`Solver`** class can be used to solve a linear system of equations subjected to a set of linear constraints (stored in a **`Constraints`** object.

Jive provides a substantial family of solvers with different performance characteristics.

The **`NodeSet`** and **`ElementSet`** classes represent (unstructured) FE meshes. They are essentially wrappers around the **`PointSet`** and **`GroupSet`** classes, respectively.

The **`Shape`** class encapsulates the geometrical properties of a finite element. It can be used to evaluate integrals over the domain of an element, and to calculate the shape functions.

The derived class **`InternalShape`** represents a regular, internal finite element. It can de used to compute the gradients of the shape functions.

The derived class **`BoundaryShape`** represents a boundary of an element. It can be used to evaluate integrals over a boundary, and to compute the normal vectors on a boundary.

The following table provides a summary of the classes that have been mentioned in the previous slides.

The classes are grouped according to the roles they play.

| Classes | Role |
|---|---|
| **`ItemSet`**<br>**`PointSet`**<br>**`GroupSet`**<br>**`NodeSet`**<br>**`ElementSet`** | Store a discrete representation of the problem domain (space/time). |

| Classes | Role |
|---|---|
| `DofSpace` `Constraints` | Manage degrees of freedom and boundary conditions. |
| `MatrixBuilder` `Solver` | Build and solve linear systems of equations. |
| `Shape` `InternalShape` `BoundaryShape` | Evaluate integrals and element shape functions. |

# Jive example program

The following slides show a Jive program that executes the following steps:

1. define a 1-D finite element mesh;
2. associate a degree of freedom with each node;
3. assemble a system of equations;
4. solve the system of equations.

***The aim of the example program is to give you an idea what a Jive program could look like. You do not need to understand the details at this point.***

The program instantiates some classes that provide a specific implementation of the classes that have been described earlier.

For instance, the class **`StdPointSet`** provides a “standard” implementation of the interface defined by the **`XPointSet`** class.

Most operations in the program are defined in terms of the base classes. This makes it possible to switch to a different implementation without having to modify the program.

## Part 1: include and using directives.

```
#include <jem/base/System.h>
#include <jive/util/Constraints.h>
#include <jive/util/StdPointSet.h>
#include <jive/util/StdGroupSet.h>
#include <jive/util/FlexDofSpace.h>
#include <jive/algebra/AbstractMatrix.h>
#include <jive/algebra/FlexMatrixBuilder.h>
#include <jive/solver/SparseLU.h>

using namespace jem;
using namespace jive;
using namespace jive::util;
using namespace jive::algebra;
using namespace jive::solver;
```

The example program contains a number of using directives for the sake of brevity. In a real program it is better to use using declarations, possibly in a separate header file **`import.h`**.

## Part 2: variable declarations.

```
int run ()
{
  const idx_t           NELM = 10;
  const idx_t           NNOD = NELM + 1;
  const double          DX   = 1.0 / NELM;

  Ref<MatrixBuilder>  mbuilder;
  Ref<XPointSet>      nodes;
  Ref<XGroupSet>      elems;
  Ref<XDofSpace>      dofs;
  Ref<Constraints>    cons;
  Ref<Solver>         solver;

  Matrix              elmat  ( 2, 2 );
  IdxVector           inodes ( 2 );
  IdxVector           idofs  ( 2 );
  Vector              coords ( 1 );
  Vector              rhs    ( NNOD );
  Vector              lhs    ( NNOD );

  idx_t               jtype;
```

The **`Ref`** class must be used for all classes that are derived (indirectly) from **`jem::Collectable`**.

## Part 3: define the mesh.

```
nodes = newInstance<StdPointSet> ( "nodes",    "node" );
elems = newInstance<StdGroupSet> ( "elements", "element",
                                   nodes );

for ( idx_t inod = 0; inod < NNOD; inod++ )
{
  coords[0] = inod * DX;

  nodes->addPoint ( coords );
}

for ( idx_t ielm = 0; ielm < NELM; ielm++ )
{
  inodes[0] = ielm;
  inodes[1] = ielm + 1;

  elems->addGroup ( inodes );
}
```

The strings **"node"** and **"element"** describe the kind of items stored in the **PointSet** and **GroupSet**, respectively.

## Part 4: define the degrees of freedom (DOFs).

```
dofs  = newInstance<FlexDofSpace> ( nodes );
jtype = dofs->addType ( "u" );

for ( idx_t inod = 0; inod < NNOD; inod++ )
{
  dofs->addDof ( inod, jtype );
}
```

Each DOF must have a type that is represented by a string. In this case all DOFs have the type `"u"`. You can use arbitrary type names and define multiple types.

The number of DOFs per node can vary, but in this case each node has one DOF.

## Part 5: assemble the system of equations.

```
mbuilder   = newInstance<FlexMatrixBuilder> ();
elmat(0,0) =  1.0;
elmat(0,1) = -1.0;
elmat(1,0) = -1.0;
elmat(1,1) =  1.0;

mbuilder->setToZero ();

for ( idx_t ielm = 0; ielm < NELM; ielm++ )
{
  // Get the nodes and DOFs attached to this element.

  elems->getGroupMembers ( inodes, ielm );
  dofs ->getDofIndices   ( idofs,  inodes, jtype );

  // Add the element matrix to the global matrix.

  mbuilder->addBlock ( idofs, idofs, elmat );
}

mbuilder->updateMatrix ();
```

The **`DofSpace`** is used to determine how each element matrix should be added to the global matrix.

## Part 6: solve the system of equations.

```
  rhs    = 0.0;
  cons   = newInstance<Constraints> ( dofs );
  solver = newInstance<SparseLU>    ( "solver",
                                      mbuilder->getMatrix(),
                                      cons );

  cons->addConstraint ( 0,         0.0 );
  cons->addConstraint ( NNOD - 1, 1.0 );

  solver->solve ( lhs, rhs );

  System::out() << "solution: " << lhs << "\n";

  return 0;
}

int main ()
{
  return System::exec ( run );
}
```

The constraints specify that the first DOF should be zero and the last should be one.

# Why this complexity?

When you are new to Jive you may be overwhelmed by the seemingly complicated class hierarchy.

Why not use multi-dimensional arrays and be done with it?

There are a number of good reasons why Jive is structured the way it is. Knowing these reasons may help appreciating the design of Jive.

By using specific classes instead of multi-dimensional arrays, it is possible to exploit special properties of a solution method.

One can, for instance, implement a special `Grid` class (see previous example) or a special sparse matrix class, without having to make large structural changes to a program.

This advantage has been used to implement some non-standard methods, including 4-D space-time methods and stochastic finite element methods.

Another advantage of the Jive class hierarchy is that objects can automatically adapt themselves when underlying objects changes.

For instance, when nodes are removed, the element connectivities are modified automatically. When DOFs are added or deleted, the solution vector is resized automatically.

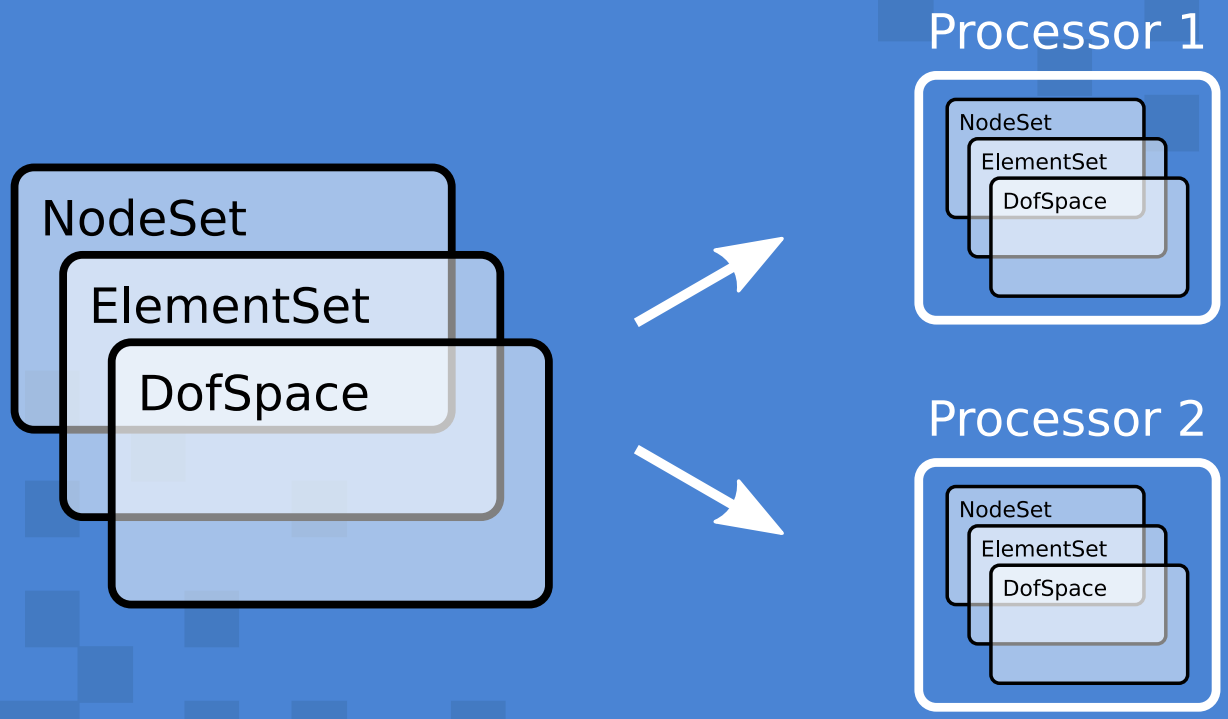
This means that relations between objects remain consistent.

A third advantage has to do with parallel computing.

Because Jive knows how objects are related to each other, it can automatically partition a computation into parallel sub-computations using the domain decomposition approach.

Jive will take care that the results are consistent between sub-domains.

## Domain decomposition:



# How to compile Jive programs

Building a Jive program is similar to building a Jem program.

One difference is that you must specify the header file and library search paths for Jive, in addition to the search paths for Jem.

Another difference is that you must link the program with both the required Jive libraries and the required Jem libraries.

Fortunately, you can use makefiles to automate the build process.

Like Jem, Jive provides a set of makefiles (one for each package) that can be used to create a makefile that works on all systems supported by Jem/Jive.

To use these makefiles you must have set the environment variable **`JIVEDIR`** equal to the Jive installation path.

Here is an example makefile for a Jive program:

**Makefile**

```
program = progName

include $(JIVEDIR)/makefiles/packages/*.mk
include $(JIVEDIR)/makefiles/prog.mk

subdirs = subdir1 subdir2 subdir3

MY_CXX_FLAGS = -O3
```

Each package-specific makefile from Jive includes the makefiles from the Jem packages on which it depends.

# The `ItemSet` class

The `ItemSet` class (package `util`) represents a homogeneous set of ***items***. An item can be a point, an element or some other type of object.

Each item in an `ItemSet` is associated with an index and an ID.

The index ranges from zero to the number of items minus one and is normally used to access an item.

The ID is an arbitrary integer that uniquely identifies an item. The index of an item may change when items are reordered, but the ID of an item is immutable.

`ItemSet` member functions:

**`size`**
: Returns the number of items.

**`findItem`**
: Returns the index of an item given an ID. A negative number is returned if no item with the ID exists.

**`getItemID`**
: Returns the ID of an item with a given index.

**`getItemName`**
: Returns the type name of an item, such as `"node"` or `"element"`.

Jive provides two families of classes that are derived from the **`ItemSet`** class: **`PointSet`** and **`GroupSet`**.

The **`PointSet`** class stores a set of points in an $n$-dimensional space. That is, each item in a **`PointSet`** has a coordinate vector in addition to an ID.

The **`PointSet`** class is immutable; use the **`XPointSet`** class if you need to add or modify points.

Both the **`PointSet`** and **`XPointSet`** classes are abstract classes. They define an interface but not an implementation; they can not be instantiated.

The **`StdPointSet`** class provides an efficient implementation in which all point coordinates are stored in a single array.

The **`GroupSet`** class stores topology information; each item in a **`GroupSet`** is a group of items in another group set.

For instance, elements or grid cells are represented by items in a **`GroupSet`**. Each element/cell is a group of points that are stored in a **`PointSet`**.

The **`XGroupSet`** class is the mutable version of the **`GroupSet`** class. Both classes are abstract.

The **`StdGroupSet`** class provides an implementation in which the topology information is stored in a few arrays.

## Example with the **PointSet** and **GroupSet** classes:

```
void example ()
{
  Matrix           coords ( 2, 3 );
  IdxVector        inodes ( 3 );
  Ref<XPointSet>   nodes;
  Ref<XGroupSet>   cells;

  nodes = newInstance<StdPointSet> ( "nodes", "node" );
  cells = newInstance<StdGroupSet> ( "cells", "cell", nodes );

  coords      = 0.0;
  coords(0,1) = 1.0;
  coords(1,2) = 1.0;

  inodes[0]   = nodes->addPoint ( coords[0] );
  inodes[1]   = nodes->addPoint ( coords[1] );
  inodes[2]   = nodes->addPoint ( coords[2] );

  cells->addGroup ( inodes );
}
```

The index assigned to an item is based on the order in which they are added to an item set.

Consider these statements from the previous example:

```
inodes[0] = nodes->addPoint ( coords[0] );
inodes[1] = nodes->addPoint ( coords[1] );
inodes[2] = nodes->addPoint ( coords[2] );
```

After the final statement the array **`inodes`** will contain the values 0, 1, 2.

As the node IDs are not specified in this example, they will be assigned automatically, also based on the order in which the nodes are added.

The **`ItemSet`** class and its derived classes are very general but also a bit abstract.

Hence, there are several application-specific *wrapper classes*, such as the **`NodeSet`** and the **`ElementSet`** class, that provide more logical member function names.

The idea is that the member function names of the wrapper classes conform better to the terminology used in a specific solution method such as the finite element method.

These classes are the subject of the slides following the next exercise.

# Exercise: create a 2-D mesh

Use the **`XPointSet`** and **`XGroupSet`** classes to create a rectangular 2-D mesh consisting of 4-node quadrilateral elements.

Exercise location: **`exercises/mesh`**

Solution location: **`solutions/mesh-1`**

# The `NodeSet` class

The **`NodeSet`** class (package **`fem`**) and its mutable version **`XNodeSet`** represent a set of nodes in a finite element mesh.

They are often used together with the **`ElementSet`** and **`XElementSet`** classes; more about those later.

The node coordinates and the element connectivities are stored in different classes to increase the possibilities for code re-use. For instance, when using mesh-less methods, only a **`NodeSet`** would be required.

**`NodeSet`** member functions:

**`size`**
: Returns the number of nodes.

**`rank`**
: Returns the number of dimensions.

**`getNodeID`**
: Returns the ID of a given node.

**`findNode`**
: Returns the node index associated with an ID, or a negative number if the ID is undefined.

**`getNodeCoords`**
: Returns the coordinates of a given node.

**`getSomeCoords`**
: Returns the coordinates of multiple nodes.

**`getCoords`**
: Returns the coordinates of all nodes.

**`getData`**
: Returns a pointer to the underlying **`PointSet`**.

**`isNull`**
: Returns **`true`** if this **`NodeSet`** has no underlying **`PointSet`**, and **`false`** otherwise. This is explained later.

The **`XNodeSet`** class adds these member functions:

**`addNode`**
: Adds a node with a given coordinate and an optional ID. The first node determines the number of dimensions; all other nodes must have the same number of dimensions.

**`setNodeCoords`**
: Updates the coordinates of a given node.

**`setSomeCoords`**
: Updates the coordinates of multiple nodes.

**`eraseNodes`**
: Deletes multiple nodes.

## Example with the **NodeSet** and **XNodeSet** classes:

```
#include <jive/fem/XNodeSet.h>

using jive::fem::NodeSet;
using jive::fem::XNodeSet;

void makeNodes ( XNodeSet& nodes )
{
  Vector  coords ( 2, 3 );

  coords      = 0.0;
  coords(0,1) = 1.0;
  coords(1,2) = 1.0;

  nodes.addNode ( coords[0] );
  nodes.addNode ( coords[1] );
  nodes.addNode ( coords[2] );
}
```

## Example, continued:

```
void printNodes ( const NodeSet& nodes )
{
  using jem::System;

  Vector  xpos ( nodes.rank() );

  for ( idx_t inode = 0; inode < nodes.size(); inode++ )
  {
    nodes.getNodeCoords ( xpos, inode );

    System::out() << nodes.getNodeID( inode ) << " : "
                  << xpos << "\n";
  }
}
```

A **`NodeSet`** instance is essentially a wrapper around a **`PointSet`** instance. The same holds for an **`XNodeSet`** instance.

All member functions of the **`NodeSet`** and **`XNodeSet`** classes simply call the equivalent member functions of the **`PointSet`** and **`XPointSet`** classes, respectively.

The **NodeSet** class definition looks something like this:

```
class NodeSet
{
 public:
  void getNodeCoords ( const Vector&   xpos,
                       idx_t           inode ) const;
  // ...
 private:
  Ref<PointSet>  nodes_;
};

void NodeSet::getNodeCoords ( const Vector&   xpos,
                              idx_t           inode ) const
{
  nodes_->getPointCoords ( xpos, inode );
}
```

The default constructor of the **NodeSet** class does not create an underlying **PointSet**; the private member **nodes_** is a null pointer.

You must pass a **PointSet** to a constructor of the **NodeSet** class in order to call its member functions.

A **NodeSet** can be seen as a kind of reference to a **PointSet**; it does not create the **PointSet** itself.

This also holds for the **XNodeSet** class, except that you can also use the function **newXNodeSet** to create an **XNodeSet** with an underlying **XPointSet**.

## How to create an **XNodeSet**:

```
#include <jive/fem/XNodeSet.h>

using namespace jive::fem;

void example ()
{
  XNodeSet  first = newXNodeSet ();
  XNodeSet  second;
  Vector    xpos  ( 2 );

  first .addNode ( xpos );  // OK
  second.addNode ( xpos );  // Error: no underlying XPointSet.
}
```

Like a true reference, a **NodeSet** or **XNodeSet** must be associated with a **PointSet** or **XPointSet** when it is constructed.

You can not associate a **NodeSet** or **XNodeSet** with another **PointSet** or **XPointSet** after it has been constructed.

This implies that the copy assignment operator has been disabled.

```
void example ()
{
  XNodeSet  nodes;

  nodes = newXNodeSet ();   // Error: disabled copy
                            // assignment operator.
}
```

By using the class template **Assignable**, however, it is possible to re-enable the copy assignment operator.

```
#include <jive/fem/XNodeSet.h>
#include <jive/util/Assignable.h>

using namespace jive::fem;
using jive::util::Assignable;

void example ()
{
  Assignable<XNodeSet>  nodes;
  Vector                xpos ( 2 );

  nodes = newXNodeSet ();  // OK

  nodes.addNode ( xpos );
}
```

Note that the **Assignable** class comes from the **util** package.

The copy assignment operator of the **Assignable** class creates a shallow copy; it copies only the address of the underlying **PointSet** or **XPointSet**.

```
void example ()
{
  Assignable<XNodeSet>  first;
  Assignable<NodeSet>   second;
  Vector                xpos ( 2 );

  first  = newXNodeSet ();
  second = first;                          // Shallow copy.

  first.addNode ( xpos );

  System::out() << second.size() << endl;  // Prints 1.
}
```

Note the conversion from a **XNodeSet** to a **NodeSet**.

The **`Assignable`** class is necessary, unfortunately, to prevent obscure errors related to *object slicing*.

While such errors are relatively rare, finding and fixing them can be a non-trivial affair.

# The `ElementSet` class

The **`ElementSet`** class (package **`fem`**) and its mutable version **`XElementSet`** represent a set of finite elements.

A **`ElementSet`** or **`XElementSet`** instance must be associated with a **`NodeSet`**. Each element is essentially an array with node indices, specifying the nodes to which en element is attached.

Jive views the elements simply as a set of connectivities; you must associate a particular shape with each element in your program. This is explained later.

Member functions of the **`ElementSet`** class:

**`size`**
: Returns the number of elements.

**`getElemID`**
: Returns the ID of a given element.

**`findElement`**
: Returns the element index associated with an ID, or a negative number if the ID is undefined.

**`getElemNodes`**
: Returns the indices of the nodes attached to a given element.

**`getNodesOf`**
: Returns the indices of the nodes attached to multiple elements.

**`getUniqueNodesOf`**
: Like the above function, but each node index occurs only once in the returned array.

**`getElemNodeCount`**
: Returns the number of nodes attached to a given element.

**`maxElemNodeCount`**
: Returns the maximum number of nodes attached to an element.

**`getNodes`**

Returns the **`NodeSet`** associated with this **`ElementSet`**.

**`getData`**

Returns a pointer to the underlying **`GroupSet`**.

**`isNull`**

Returns **`true`** if there is no underlying **`GroupSet`**, and **`false`** otherwise.

The class **`XElementSet`** adds the following member functions:

**`addElement`**
: Adds an element with a given array of node indices and an optional ID.

**`setElemNodes`**
: Updates the connectivity of a given element.

**`eraseElements`**
: Deletes a given set of elements.

As indicated, Jive does not impose any restrictions on the element connectivities.

However, the order in which the nodes are specified for an element is significant when evaluating the element shape functions (with or without the **Shape** class).

The **ElementSet** class guarantees that the order of the nodes attached to an element is preserved. Thus, **getElemNodes** returns the same index array as the one that was passed to **addElement** or **setElemNodes**.

The **`ElementSet`** and **`XElementSet`** classes are essentially wrappers around the **`GroupSet`** and **`XGroupSet`** classes, respectively.

A **`ElementSet`** or **`XElementSet`** does not create its underlying group set by default; you must either pass a group set to the constructor, or use the function **`newXElementSet`**.

The copy assignment operators are disabled. Use the **`Assignable`** class to re-enable them.

## Example with the **ElementSet** and **NodeSet** classes:

```
void example ()
{
  Matrix                    coords ( 2, 3 );
  IdxVector                 inodes ( 3 );
  Assignable<XNodeSet>      nodes;
  Assignable<XElementSet>   elems;

  nodes = newXNodeSet    ();
  elems = newXElementSet ( nodes );

  coords      = 0.0;
  coords(0,1) = 1.0;
  coords(1,2) = 1.0;

  inodes[0]   = nodes.addNode ( coords[0] );
  inodes[1]   = nodes.addNode ( coords[1] );
  inodes[2]   = nodes.addNode ( coords[2] );

  elems.addElement ( inodes );
}
```

Include and using directives not shown for the sake of brevity.

## Another example with the **ElementSet** class:

```
#include <jive/fem/ElementSet.h>

using namespace jive;
using namespace jive::fem;

void printElems ( const ElementSet& elems )
{
  using jem::System;

  IdxVector  inodes;

  for ( idx_t ielem = 0; ielem < elems.size(); ielem++ )
  {
    inodes.resize ( elems.getElemNodeCount( ielem ) );
    elems.getElemNodes ( inodes, ielem );

    System::out() << elems.getElemID( ielem ) << " : "
                  << inodes << "\n";
  }
}
```

# Exercise: create another 2-D mesh

Use the **`XNodeSet`** and **`XElementSet`** classes to create a rectangular 2-D mesh; replace the **`XPointSet`** and **`XGroupSet`** classes in the previous exercise.

Print the mesh to verify that it is correct.

Exercise location: **`exercises/mesh`**

Solution location: **`solutions/mesh-2`**

# The `DofSpace` class

The `DofSpace` class (package `util`) maps degrees of freedom (DOFs) to a contiguous range of indices or equation numbers.

It is the primary mechanism for mapping a discrete model to a (non-linear) system of equations.

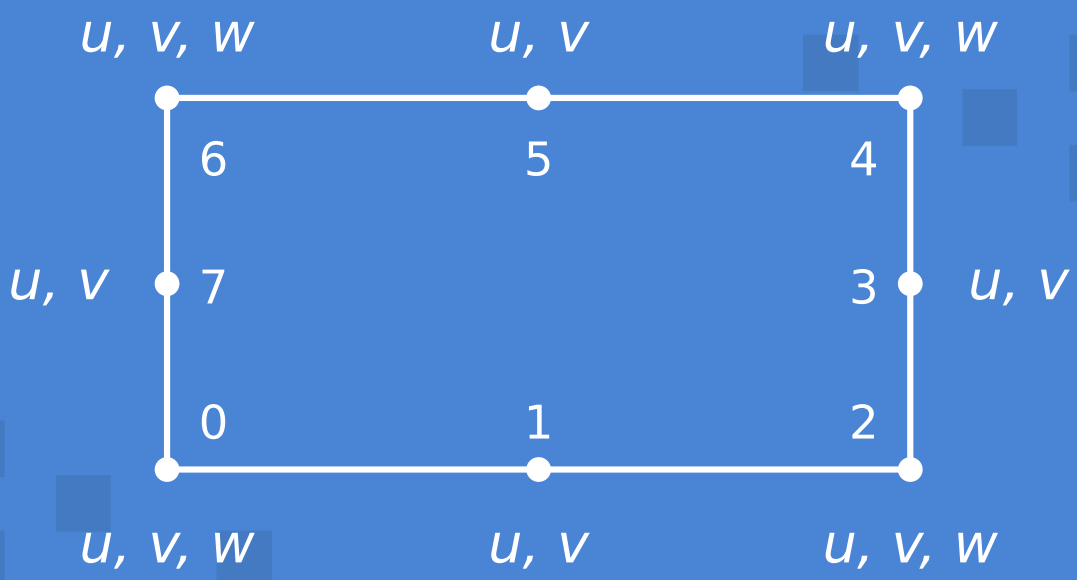
Without a `DofSpace` you would need another mechanism to figure out which entry in the global solution vector corresponds with which degree of freedom.

A **`DofSpace`** is basically an integer matrix in which each row represents an item in an **`ItemSet`**, and each column represents a DOF type like temperature or pressure.

The value of a matrix element $(i, j)$ is the global index, or equation number, of the DOF that is attached to item $i$ and has type $j$.

To illustrate this, consider a mesh with 8 nodes and three degree of freedom types: $u$, $v$ and $w$. The corner nodes have three DOFs and the mid nodes have only two DOFs; see the next slide.

Example mesh with 8 nodes and three DOF types:




The numbers denote the node numbers.

Here are the corresponding `DofSpace` and the solution vector.

| | DOF Index | | |
|---|---|---|---|
| Node | $u$ | $v$ | $w$ |
| 0 | 0 | 1 | 2 |
| 1 | 3 | 4 | |
| 2 | 5 | 6 | 7 |
| 3 | 8 | 9 | |
| 4 | 10 | 11 | 12 |
| 5 | 13 | 14 | |
| 6 | 15 | 16 | 17 |
| 7 | 18 | 19 | |

| Index | Value |
|---|---|
| 0 | $u_0$ |
| 1 | $v_0$ |
| 2 | $w_0$ |
| 3 | $u_1$ |
| 4 | $v_1$ |
| ⋮ | |
| 18 | $u_7$ |
| 19 | $v_7$ |

The DOF $v$ in node 2 has DOF index 6, which means that that DOF is stored at index 6 in the global solution vector.

Note that some entries in the `DofSpace` are blank because no DOF of the corresponding type has been defined in the corresponding node.

If the DOFs are ordered in a different way, then the contents of the `DofSpace` will be different. The entries in the solution vector will also be ordered in a different way.

A Jive program would not be aware of this change as long as it uses the `DofSpace` to determine which DOF is associated with which item and which DOF type.

A **`DofSpace`** must be associated with an **`ItemSet`**; the DOFs are associated with the items in this **`ItemSet`**.

When items are removed from the **`ItemSet`** associated with a **`DofSpace`**, then the DOFs associated with those items are automatically removed too.

The **`DofSpace`** class is immutable; use the **`XDofSpace`** class to add and remove DOFs.

The `DofSpace` member functions:

**`dofCount`**
: Returns the total number of DOFs. This is equal to the size of the solution vector.

**`typeCount`**
: Returns the number of of DOF types.

**`getTypeIndex`**
: Returns the index associated with a DOF type given a label (`String`).

**`getTypeName`**
: Returns the label (`String`) associated with a given DOF type index.

**`getDofIndex`**

Returns the DOF index for a given item index and a given DOF type index.

**`getDofIndices`**

Returns the DOF indices for multiple items and/or DOF types.

**`decodeDofIndex`**

Returns the item index and DOF type index for a given DOF index.

**`getItems`**

Returns a pointer to the **`ItemSet`** associated with this **`DofSpace`**.

## Example: print a solution vector.

```
void printSolution ( Writer&          out,
                     const Vector&    sol,
                     const DofSpace&  dofs )
{
  Ref<ItemSet>  items = dofs.getItems ();
  idx_t         iitem;
  idx_t         jtype;

  for ( idx_t i = 0; i < sol.size(); i++ )
  {
    dofs.decodeDofIndex ( iitem, jtype, i );

    out << dofs.getTypeName( jtype )  << "["
        << items->getItemsID( iitem ) << "] = "
        << sol[i] << "\n";
  }
}
```

## Output:

```
u[0] = 2.0;
v[0] = 2.5;
w[0] = 0.1;
u[1] = 2.1;
v[1] = 2.5;

   :

u[7] = 3.0;
v[7] = 1.9;
```

The **`XDofSpace`** class adds these member functions:

**`addType`**

Adds a DOF type with a given label (**`String`**). If the type has already been defined, then this function simply returns the index associated with that type. This function essentially adds a column to the table encapsulated by this **`DofSpace`**.

**`addDof`**

Add a DOF with a given item index and DOF type index. If that DOF has already been defined, then this function simply returns the DOF index.

**`addDofs`**

Adds multiple DOFs for multiple items and/or DOF types.

An **`XDofSpace`** is empty by default. You must first add one or more DOF types by calling the function **`addType`**.

After that, you must define the DOFs by calling the function **`addDof`** or **`addDofs`**.

The following slide shows an example in which a different number of DOFs are defined for items with even and odd indices.

## Example with the **XDofSpace** class:

```
void addDofs ( XDofSpace& dofs )
{
  Ref<ItemSet>  items  = dofs.getItems ();
  IdxVector     jtypes ( 3 );

  jtypes[0] = dofs.addType ( "u" );
  jtypes[1] = dofs.addType ( "v" );
  jtypes[2] = dofs.addType ( "w" );

  for ( idx_t iitem = 0; iitem < items->size(); iitem++ )
  {
    if ( (iitem % 2) == 0 )
    {
      dofs.addDofs ( iitem, jtypes );
    }
    else
    {
      dofs.addDofs ( iitem, jtypes[slice(BEGIN,2)] );
    }
  }
}
```

Both the **`DofSpace`** and **`XDofSpace`** classes are abstract base classes.

Jive provides several derived classes, each making a different trade-off between memory use and performance/flexibility. They are all defined in the **`util`** package.

Use the **`DenseDofSpace`** class if the DOF index table is mostly filled. That is, if all DOF types are associated with most items.

Use the **`SparseDofSpace`** class if the DOF index table is mostly empty.

Use the **`FlexDofSpace`** class if the sparsity pattern of the DOF index table is not known beforehand. It will automatically switch between a **`SparseDofSpace`** and a **`FlexDofSpace`**.

Use the **`UniformDofSpace`** class if all DOF types are associated with all items. This class is derived from the **`DofSpace`** class and not the **`XDofSpace`** class.

In most cases the **`FlexDofSpace`** class is a safe choice.

## Example with the **FlexDofSpace** class:

```
void example ()
{
  XNodeSet         nodes  = newXNodes ();
  Vector           coords ( 2, 2 );
  Ref<XDofSpace>   dofs;

  dofs   = newInstance<FlexDofSpace> ( nodes.getData() );
  coords = 0.0;

  coords(0,1) = 1.0;
  coords(1,0) = 1.0;

  nodes.addNode ( coords[0] );
  nodes.addNode ( coords[1] );

  addDofs ( *dofs );  // From the previous example.
}
```

Note the call **nodes.getData()** to get the underlying item set (**XPointSet**).

# Exercise: build a mesh and define DOFs

Extend the program from the previous exercise by associating DOFs with the nodes.

Print the DOFs associated with each element.

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

Solution location: **`solutions/mesh-4`**.

# Hints

Use the member function **getDofIndices** to get the DOFs associated with an element.

This function is declared as follows:

```
void getDofIndices ( const IdxVector&  idofs,
                     const IdxVector&  iitems,
                     const IdxVector&  jtypes ) const;
```

The DOF indices are returned in the array **idofs**. The arrays **iitems** and **jtypes** should contain the node indices and DOF type indices, respectively. What should be the size of **idofs**?

# The `Constraints` class

The `Constraints` class (package `util`) can be used to specify linear constraints for a selected set of DOFs.

To be precise, it can be used to specify that a DOF has a constant value, or that a DOF is a linear combination of other DOFs.

Use the `Constraints` class to handle essential boundary conditions.

The constraints are taken into account when solving a linear system of equations, and ***not*** when assembling that system (although that is also possible).

This has two advantages. First, the same matrix can be used with varying constraints.

Second, each linear solver can use the most efficient method to take the constraints into account.

For instance, an iterative solver can apply the constraints without having to modify the matrix.

Suppose that the DOFs are represented by the symbols $u_1, \ldots, u_n$. Then a constraint can be written as:

$$u_i = r_i + \sum_{j=1}^{n} a_{ij} u_j$$

where $r_i$ and $a_{ij}$ are given values, which may be zero.

The DOF $u_i$ is called the ***slave*** DOF and the DOFs $u_j$ with $a_{ij} \neq 0$ are called the ***master*** DOFs. The slave DOFs will be removed from the system of equations by the solver.

The previous formulation of a constraint can be used to:

- specify prescribed values for particular DOFs;
- specify periodic boundary conditions;
- connect non-matching meshes/grids;
- implement contact models;
- and implement local mesh refinement.

When a **Constraints** object is created, it must be associated with a **DofSpace**:

```
#include <jive/util/Constraints.h>

using namespace jem;
using namespace jive::util;

void example ()
{
  Ref<DofSpace>     dofs = ...;
  Ref<Constraints>  cons = newInstance<Constraints> ( dofs );

  // ...
}
```

The **Constraints** object uses the **DofSpace** object to check whether the specified DOF indices are valid, and to update the constraints when DOFs are removed.

The `Constraints` member functions:

**`slaveDofCount`**
: Returns the number of slave DOFs.

**`masterDofCount`**
: Returns the number of master DOFs.

**`getSlaveDofs`**
: Returns the indices of the slave DOFs.

**`getMasterDofs`**
: Returns the indices of the master DOFs associated with a given slave DOF.

**`clear`**
: Removes all constraints.

**`addConstraint`**
: Adds a constraint for a given slave DOF. This function is overloaded for defining different types of constraints.

**`eraseConstraint`**
: Erases the constraint associated with a given slave DOF.

**`getDofSpace`**
: Returns a pointer to the **`DofSpace`** associated with this **`Constraints`**.

**`printTo`**
: Prints the constraints to a **`PrintWriter`**.

Example: set all DOFs with type **u** associated with a given set of items to a constant value.

```
void setConstraints ( Constraints&       cons,
                      const IdxVector&   iitems,
                      double             value )
{
  Ref<DofSpace>  dofs  = cons->getDofSpace  ();
  idx_t          jtype = dofs->getTypeIndex ( "u" );

  for ( idx_t i = 0; i < iitems.size(); i++ )
  {
    idx_t  idof = dofs->getDofIndex ( iitems[i], jtype );

    // This sets: DOF[idof] = value.

    cons.addConstraint ( idof, value );
  }
}
```

# The `MatrixBuilder` class

The **`MatrixBuilder`** class (package **`algebra`**) can be used to assemble a (large) matrix without having to know how the matrix is stored.

It provides functions for adding and removing matrix elements and sub-matrices.

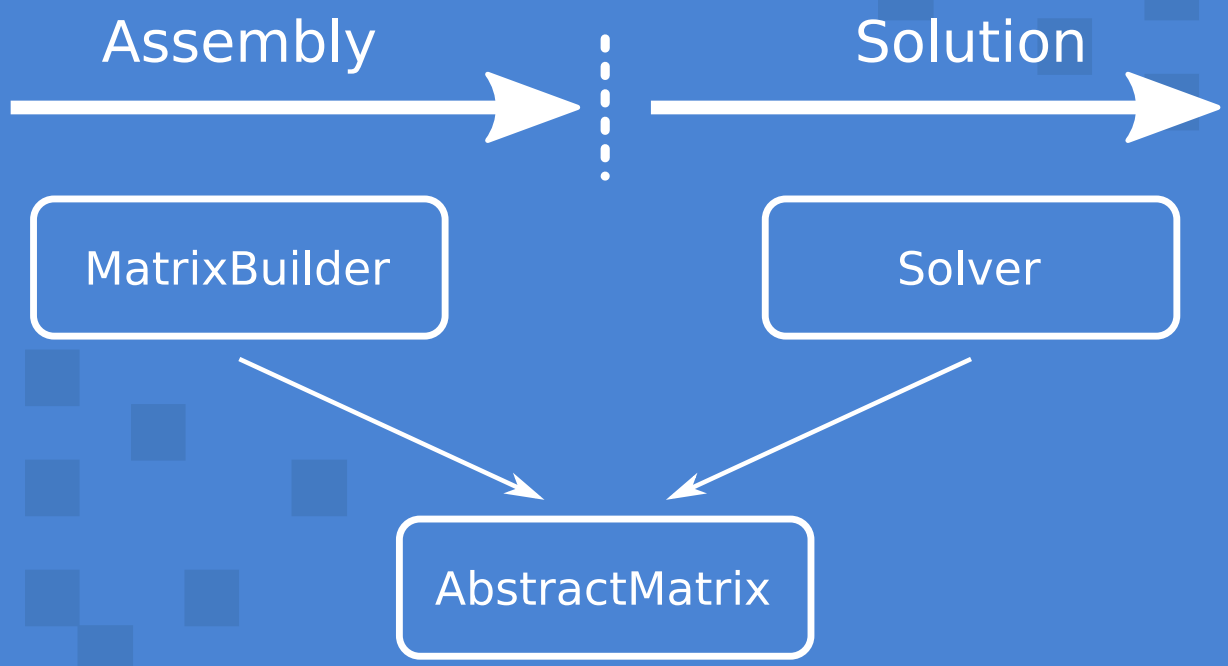
The **`MatrixBuilder`** class decouples the assembly procedure from the representation of a matrix.

A **`MatrixBuilder`** has an underlying **`AbstractMatrix`** object in which it stores the matrix elements.

The **`AbstractMatrix`** object is normally associated with a **`Solver`** that solves a system of equations (more about that later).

Different data structures are used in a **`MatrixBuilder`** and a **`AbstractMatrix`** because the assembly and solution of a system of equations involve different matrix access patterns.

The role of the **MatrixBuilder** class:



The **MatrixBuilder** member functions:

**clear**
: Deletes all matrix elements.

**setToZero**
: Sets all matrix elements to zero.

**getMatrix**
: Returns a pointer to the underlying **AbstractMatrix**.

**updateMatrix**
: Flushes all changes to the underlying **AbstractMatrix**.
This function must be called before using the matrix.

**`setValue`**
: Sets a matrix element at an index pair $(i, j)$.

**`addValue`**
: Adds a value to a matrix element at an index pair $(i, j)$.

**`getValue`**
: Returns a matrix element at an index pair.

**`eraseValue`**
: Deletes a matrix element at an index pair. This function deletes the matrix element from the underlying data structure.

**`setBlock`**
: Stores a sub-matrix spanned by two index arrays.

**`addBlock`**
: Adds a sub-matrix spanned by two index arrays.

**`getBlock`**
: Returns a sub-matrix spanned by two index arrays.

**`eraseBlock`**
: Deletes a sub-matrix spanned by two index arrays. This function deletes the sub-matrix from the underlying data structure.

# Example with the **MatrixBuilder** class:

```
#include <jive/algebra/MatrixBuilder>

using jive::algebra::MatrixBuilder;

void assemble ( MatrixBuilder& mb, idx_t n )
{
  mb.clear ();

  // Assemble a tri-diagonal matrix:

  for ( idx_t i = 0; i < n; i++ )
  {
    if ( i > 0 )
    {
      mb.setValue ( i, i - 1, -1.0 );
    }

    mb.setValue ( i, i, 1.0 );

    if ( i < (n - 1) )
    {
      mb.setValue ( i, i + 1, -1.0 );
    }
  }
}
```

The **`MatrixBuilder`** class is often used together with the **`DofSpace`** class. The latter is used to determine where to store the matrix elements.

The next slide shows an example in which a **`DofSpace`** is used to determine where to store an element matrix in a **`MatrixBuilder`**.

```
Matrix  makeElemMatrix ();

void    assemble ( MatrixBuilder&      mb,
                   const ElementSet&   elems,
                   const DofSpace&     dofs )
{
  IdxVector  jtypes ( 2 );
  IdxVector  inodes;
  IdxVector  idofs;
  Matrix     elmat;

  jtypes[0] = dofs.getTypeIndex ( "u" );
  jtypes[1] = dofs.getTypeIndex ( "v" );

  mb.setToZero ();

  for ( idx_t ielem = 0; ielem < elems.size(); ielem++ )
  {
    // Resize inodes, idofs and elmat ...

    elems.getElemNodes  ( inodes, ielem );
    dofs .getDofIndices ( idofs,  inodes, jtypes );

    elmat = makeElemMatrix ();

    mb.addBlock ( idofs, idofs, elmat );
  }
}
```

In the statement

```
mb.addBlock ( idofs, idofs, elmat );
```

the array **idofs** specifies how the entries in the element matrix **elmat** are to be mapped to the global matrix.

The **idofs** array is specified twice because the rows and columns of **elmat** are mapped in the same way.

Different index arrays can be used when assembling a coupled system of equations, for instance.

The `MatrixBuilder` class is an abstract base class. Jive provides multiple derived classes that make different trade-offs between speed and memory usage.

There are also derived classes that are aimed at specific types of applications or that apply a special transform to the matrix elements added to the global matrix.

Classes derived from **`MatrixBuilder`**:

**`FlexMatrixBuilder`**

Can be used to assemble matrices with an arbitrary non-zero pattern. Use this class when the non-zero pattern is not known beforehand.

**`FEMatrixBuilder`**

Can be used in “traditional” finite element applications in which the non-zero pattern of the matrix is determined by the element connectivities. This class is more efficient than the **`FlexMatrixBuilder`** class.

**`LumpedMatrixBuilder`**

Can be used to assemble a lumped matrix. This class automatically lumps all matrix elements and creates a diagonal matrix.

**`ConMatrixBuilder`**

Can be used to apply constraints when assembling the global matrix. Use this class when the number of constraints is of the same order as the number of DOFs.

# Exercise: assemble a matrix

Use the **`FlexMatrixBuilder`** class to assemble a block-diagonal matrix (see next slide).

Can you use the **`addBlock`** member function?

Print the matrix to verify that it is correct.

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

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

The matrix should look like this:

$$\begin{bmatrix} 1 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

# The `Solver` class

The `Solver` class (package `solver`) can be used to solve a system of linear equations represented by an `AbstractMatrix` and – optionally – a `Constraints` object.

The `Solver` class provides a common interface for various solution algorithms, including direct algorithms based in Gauss elimination and iterative solution algorithms.

The `Solver` member functions:

**`solve`**
: Computes the solution of the system of equations given a right-hand side vector.

**`improve`**
: Computes the solution given a right-hand side vector and an initial guess for the solution.

**`setPrecision`**
: Sets the relative precision with which the solution is to be computed.

A **Solver** must be given an **AbstractMatrix** and an optional **Constraints** object through its constructor.

Example:

```
void example ()
{
  Ref<AbstractMatrix>  matrix;
  Ref<Constraints>     cons;
  Ref<Solver>          solver;

  // Initialize the matrix and the constraints ...

  solver = newInstance<SparseLU> ( "solver", matrix, cons );
}
```

The class **SparseLU** is derived from the **Solver** class and implements a sparse LU-factorization algorithm.

The **`AbstractMatrix`** associated with a **`Solver`** is normally obtained from a **`MatrixBuilder`**.

The next slide shows an example in which a **`MatrixBuilder`** and a **`Solver`** are used to assemble and solve a (trivial) system of equations.

Note that the object **`jem::NIL`** is used to indicate that there are no constraints.

```
void example ()
{
  const idx_t          n = 20;

  Ref<MatrixBuilder>  mbuilder;
  Ref<Solver>          solver;

  Vector               lhs ( n );
  Vector               rhs ( n );

  mbuider = newInstance<FlexMatrixBuilder> ();
  solver  = newInstance<SparseLU> ( "solver",
                                    mbuilder->getMatrix(), NIL );
  mbuilder->setToZero ();

  for ( idx_t i = 0; i < n; i++ )
  {
    mbuilder->addValue ( i, i, 1.0 );
  }

  mbuilder->updateMatrix ();

  rhs = 1.0;

  solver->solve ( lhs, rhs );
}
```

# The `Shape` class

The `Shape` class (package `geom`) encapsulates the geometrical properties of a finite element.

A `Shape` comprises a set of nodes, a set of shape functions and an integration scheme.

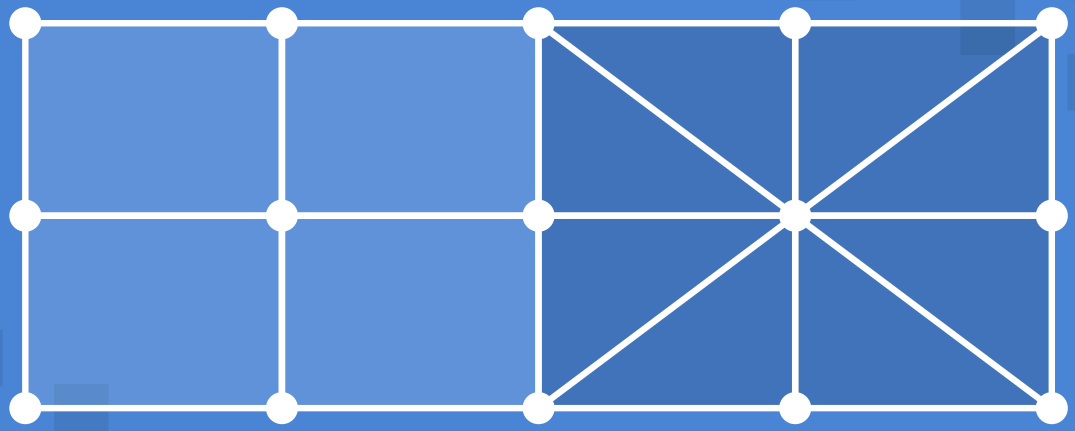
It can be used to evaluate integrals over the domain of an element, and to calculate the shape functions.

A `Shape` does not store the coordinates of its nodes; these must be passed to the member functions that require those coordinates.

This has the advantage that a single `Shape` instance can be used for multiple elements in a mesh.

For instance, if a mesh consists of quadrilateral and triangular elements, then only two `Shape` objects are needed: a quadrilateral shape and a triangle shape.

Re-use of **Shape** objects for multiple elements:



Quad Shape

Triangle Shape

## Re-use of **Shape** objects for multiple elements:

```
void example ( const ElementSet& elems )
{
  Ref<Shape>  tria = Triangle3::getShape ( ... );
  Ref<Shape>  quad = Quad4    ::getShape ( ... );
  Ref<Shape>  shape;
  Matrix      sf;

  for ( idx_t ielem = 0; ielem < elems.size(); ielem++ )
  {
    if ( elems.getElemNodeCount( ielem ) == 3 )
    {
      shape = tria;
    }
    else
    {
      shape = quad;
    }

    // Get the shape functions for this element.

    sf.ref ( shape->getShapeFunctions() );

    // Do something with the shape functions ...
  }
}
```

The **Shape** member functions:

**nodeCount**
: Returns the number of nodes associated with this **Shape**.

**ipointCount**
: Returns the number of integration points associated with this **Shape**.

**localRank**
: Returns the number of dimensions (rank) of the local coordinate system. More about this later.

**globalRank**
: Returns the rank of the global coordinate system.

**`getShapeFunctions`**
: Returns the values of the shape functions in the integration points.

**`evalShapeFunctions`**
: Computes the shape functions in a given point within this `Shape`.

**`getIntegrationWeights`**
: Returns the integration point weights in the global coordinate system, given the node coordinates.

**`getGlobalIntegrationPoints`**
: Returns the integration point coordinates in the global coordinate system.

## Example with the **Shape** class:

```
void example ()
{
  Ref<Shape>  shape  = Triangle3::getShape ( ... );
  Matrix      sfuncs = shape->getShapeFunctions ();

  for ( idx_t j = 0; j < shape->ipointCount(); j++ )
  {
    for ( idx_t i = 0; i < shape->nodeCount(); i++ )
    {
      // Print the value of the i-th shape function in the
      // j-th integration point:

      System::out() << sfuncs(i,j) << "\n";
    }
  }
}
```

Note that the statement

```
Ref<Shape>  shape = Triangle3::getShape ( ... );
```

creates a **Shape** instance representing a 3-node triangle. More about this later.

In traditional finite element applications the number of shape functions is equal to the number of nodes; one shape function is associated with each node.

## Another example with the **Shape** class:

```
void example ()
{
  Ref<Shape>  shape = Triangle3::getShape ( ... );
  Matrix      coords  ( 2, 3 );
  Vector      weights ( shape->ipointCount() );

  coords(0,0) = 0.0;  // X-coordinate first node.
  coords(0,1) = 0.0;  // Y-coordinate first node.
  coords(1,0) = 2.0;  // X-coordinate second node.
  coords(1,1) = 0.0;  // Y-coordinate second node.
  coords(2,0) = 2.1;  // X-coordinate third node.
  coords(2,1) = 3.0;  // Y-coordinate third node.

  shape->getIntegrationWeights ( weights, coords );

  for ( idx_t i = 0; i < shape->ipointCount(); i++ )
  {
    // Print the weight associated with the i-th
    // integration point:

    System::out() << weights[i] << "\n";
  }
}
```

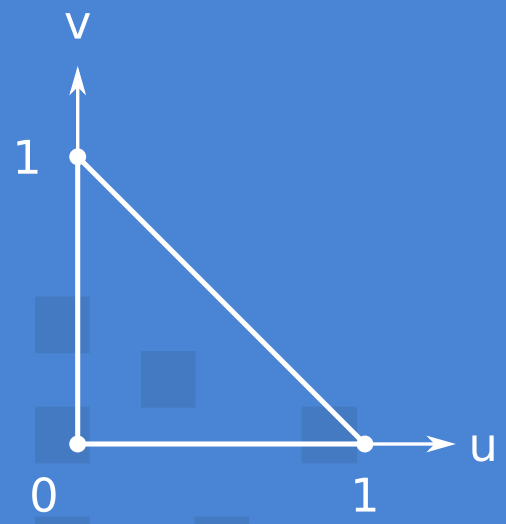
# Local and global coordinate systems

A **Shape** has a local coordinate system in which it has a fixed geometry.

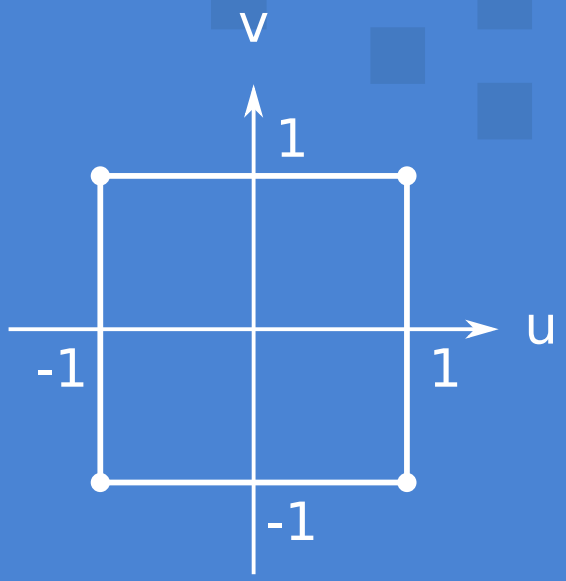
In other words, the nodes of a **Shape** are located at fixed points in the local coordinate system of that **Shape**.

The rank (number of dimensions) of the local coordinate system is said to be the ***local rank*** of a **Shape**.

## Local coordinate systems:




Triangle




Quadrilateral

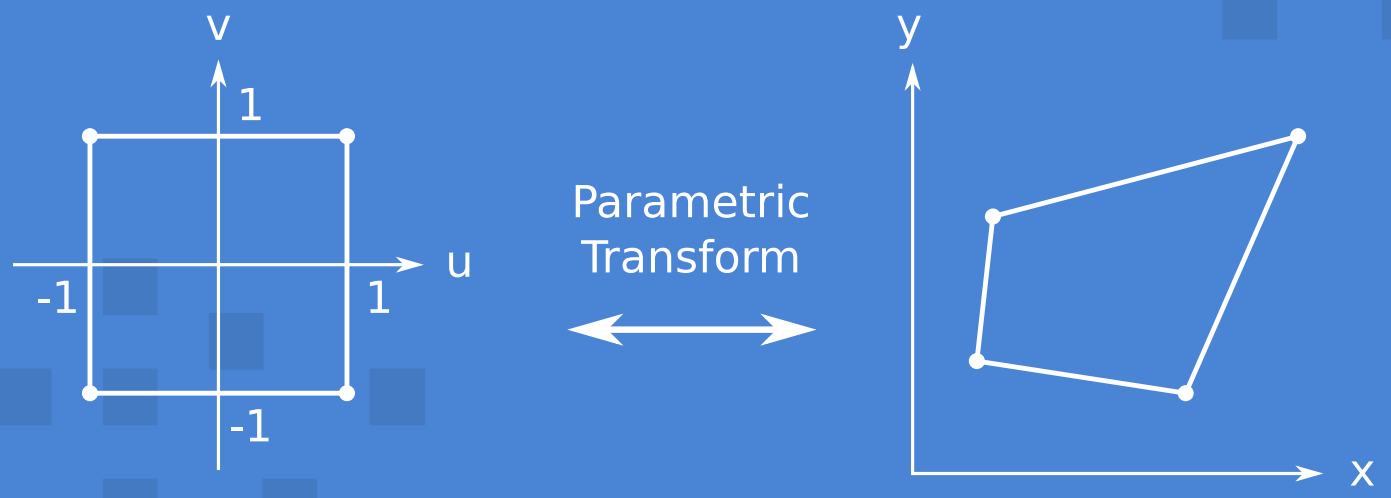
A `Shape` evaluates its shape functions and integration point weights in a global coordinate system that is connected to a finite element mesh or some region in space and/or time.

The rank of the global coordinate system is said to be the ***global rank*** of a `Shape`.

Note that the global rank may differ from the local rank. More about this later.

All shape classes in Jive implement a parametric transform to map their local coordinate system to the global coordinate system.

## Local and global coordinate system:



# Numerical integration

Each **Shape** object encapsulates a numerical integration scheme such that:

$$\int_\Omega f(\boldsymbol{x})\,\mathrm{d}\Omega \approx \sum_{i=1}^{n} w_i\, f(\boldsymbol{x}_i)$$

with:

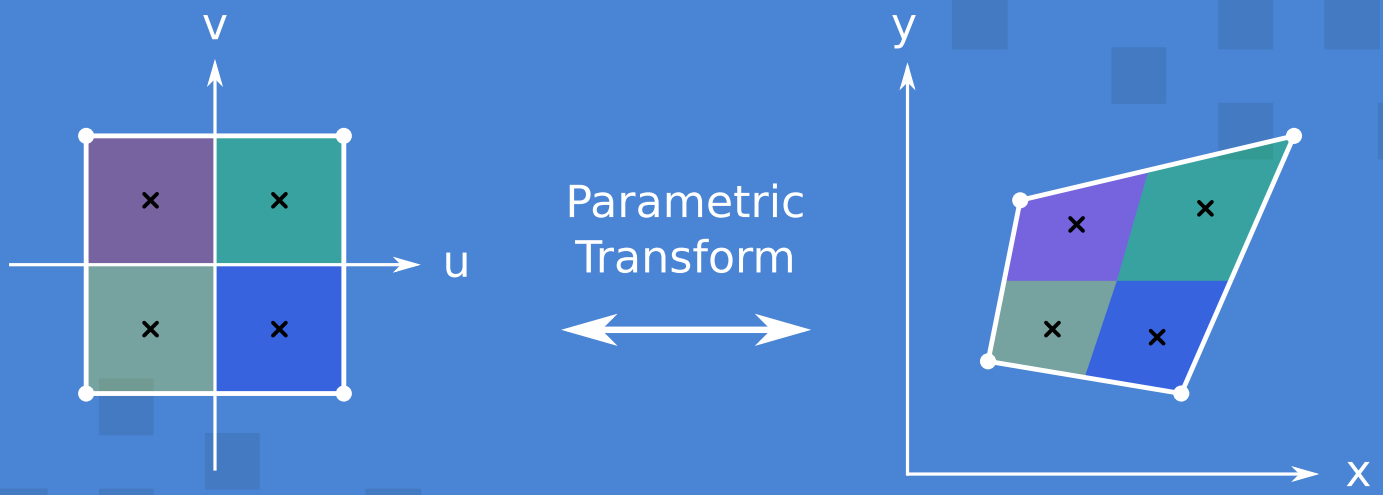
| | |
|---|---|
| $\Omega$ | the domain of an element; |
| $w_i$ | an integration point weight; |
| $\boldsymbol{x}_i$ | an integration point; |
| $n$ | the number of integration points. |

An integration scheme is specified in the local coordinate system of a **`Shape`**.

The **`Shape`** maps the integration scheme to the global coordinate system using a parametric transformation.

This means that the weights returned by the function **`getIntegrationWeights`** can be used to evaluate an integral in the global coordinate system.

Transformation of an integration scheme:




The colored areas indicate the integration point weights.

## How to determine the surface area of a triangle:

```
void example ()
{
  Ref<Shape>  shape   = Triangle3::getShape ( ... );
  Matrix      coords  ( 2, 3 );
  Vector      weights ( shape->ipointCount() );

  coords(0,0) = 0.0;  // X-coordinate first node.
  coords(0,1) = 0.0;  // Y-coordinate first node.
  coords(1,0) = 2.0;  // X-coordinate second node.
  coords(1,1) = 0.0;  // Y-coordinate second node.
  coords(2,0) = 2.1;  // X-coordinate third node.
  coords(2,1) = 3.0;  // Y-coordinate third node.

  shape->getIntegrationWeights ( weights, coords );

  System::out() << "Triangle surface area = "
                << sum( weights ) << "\n";
}
```

Jive implements various integration schemes, including:

- Gauss-Legendre of an arbitrary order;
- Newton-Cotes of various orders;
- and uniform.

The uniform integration scheme is aimed at evaluating integrals of discontinuous functions.

You can also define your own integration scheme if necessary.

## Exercise: calculate the area of a mesh

Use the `Shape` class to calculate the area of a mesh consisting of quadrilateral elements.

Use the code from a previous `mesh` exercise to create the mesh.

Exercise location: `exercises/integrate`.

Solution location: `solutions/integrate-1`.

# The `InternalShape` class

The **`InternalShape`** class (package **`geom`**) is derived from the **`Shape`** class and can be used to evaluate integrals over the interior of a mesh.

It can also be used to calculate the gradients of the shape functions within the global coordinate system.

It typically plays an important role in the assembly of the element matrices.

The `InternalShape` class extends the `Shape` class with the following member functions:

**`getShapeGradients`**
: Returns the gradients of the shape functions in the integration points, given the global node coordinates. This function also return the integration point weights in the global coordinate system.

**`evalShapeGradients`**
: Calculates the gradients of the shape functions in a given point in the local coordinate system, given the global node coordinates.

Note that the function **getShapeGradients** returns both the gradients of the shape functions and the integration point weights.

The reason is efficiency; when computing the shape function gradients you get the integration point weights more or less for free.

The shape function gradients are returned in a 3-D array, as illustrated in the next example. Note the use of the **Cubix** type that is defined as

```
typedef Array<double,3> Cubix;
```

```
void example ( const InternalShape&  shape,
               const Matrix&         coords )
{
  const idx_t  rank      = shape.globalRank  ();
  const idx_t  nodeCount = shape.nodeCount   ();
  const idx_t  ipCount   = shape.ipointCount ();

  Cubix        grads     ( rank, nodeCount, ipCount );
  Vector       weights   ( ipCount );

  shape.getShapeGradients ( grads, weights, coords );

  for ( idx_t k = 0; k < ipCount; k++ )
  {
    for ( idx_t j = 0; j < nodeCount; j++ )
    {
      for ( idx_t i = 0; i < rank; i++ )
      {
        // Print the derivative with respect to the i-th
        // coordinate of the j-th shape function in the
        // k-th integration point:

        System::out() << grads(i,j,k) << "\n";
      }
    }
  }
}
```

Instead of using the long name **InternalShape**, you can also use the abbreviated name **IShape** that is defined as:

```
typedef InternalShape IShape;
```

The function in the previous exercise could therefore also be declared as:

```
void example ( const IShape&  shape,
               const Matrix&  coords );
```

Jive provides a collection of classes that are derived from the `InternalShape` class and that implement a variety of shape functions for different geometries.

To instantiate these classes, however, one needs to know more about the `geom` package than has been explained so far.

Fortunately, Jive also provides a set of classes that simplify the construction of internal shape objects.

All these classes define a static member function named **getShape** that returns an **InternalShape** given a few parameters.

Here is an example that shows how to create an **InternalShape** representing a 4-node quadrilateral:

```
#include <jive/geom/Quad.h>

using namespace jem;
using namespace jive::geom;

void example ()
{
  String        ischeme = "Gauss2 * Gauss2";
  String        bscheme = "Gauss2";

  Ref<IShape>   quad4   = Quad4::getShape ( "quad",
                                            ischeme, bscheme );
}
```

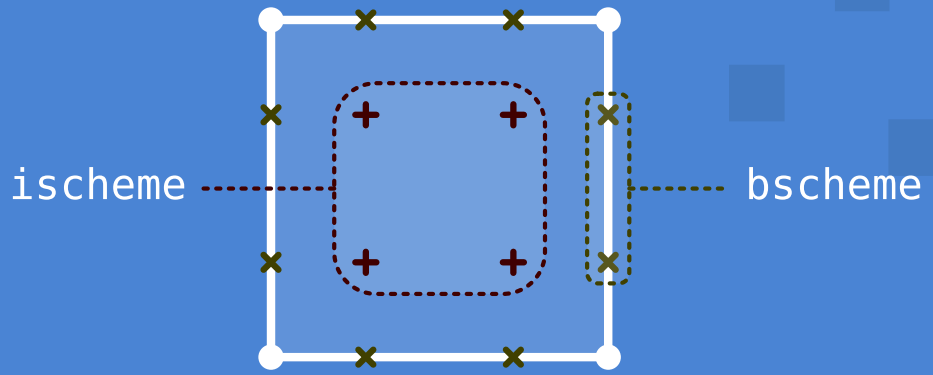
The **getShape** function has the following signature:

```
Ref<IShape> getShape ( const String&  name,
                       String&        ischeme,
                       String&        bscheme );
```

The parameter **name** specifies the name of the shape object. This is used primarily for error messages. You can specify any string you want, even an empty one.

The parameters **ischeme** and **bscheme** specify the integration schemes to be used for integrals over the domain and over the boundaries, respectively, of the shape.

The **`ischeme`** and **`bscheme`** parameters:




Both **`ischeme`** and **`bscheme`** are input/output parameters; if the specified integration scheme is not available, then the closest matching one is selected instead and the parameter is modified accordingly.

The **bscheme** parameter is only relevant if one uses the **BoundaryShape** objects associated with the boundaries of an **InternalShape**. It will be ignored here.

An integration scheme is specified by the type of the scheme followed by the number of desired integration points.

Examples:

```
"Gauss4"
"NewtonCotes6"
"Uniform8"
```

For quadrilaterals and hexahedra the integration scheme can be specified as the tensor product of 1-D integration schemes using the * token.

For instance:

```
"Gauss4 * Gauss4"
"Gauss4 * NewtonCotes4"
"Guass4 * Gauss4 * Gauss4"
```

The table below shows a selection of the classes with a **`getShape`** function. They all create **`InternalShape`** objects based on Lagrange polynomials.

| Class | Description | Header file |
|---|---|---|
| **`Line2`** | 2-node line | **`Line.h`** |
| **`Line3`** | 3-node line | |
| **`Triangle3`** | 3-node triangle | **`Triangle.h`** |
| **`Triangle6`** | 6-node triangle | |
| **`Quad4`** | 4-node quadrilateral | **`Quad.h`** |
| **`Quad8`** | 8-node quadrilateral | |
| **`Quad12`** | 12-node quadrilateral | |

| Class | Description | Header file |
|---|---|---|
| `Tetrahedron4` | 4-node tetrahedron | `Tetrahedron.h` |
| `Tetrahedron10` | 10-node tetrahedron | |
| `Hexahedron8` | 8-node hexahedron | `Hexahedron.h` |
| `Hexahedron20` | 20-node hexahedron | |
| `Wedge6` | 6-node wedge | `Wedge.h` |
| `Wedge18` | 18-node wedge | |

Note that the `getShape` functions of the `Line` classes have no `bscheme` parameter.

# Exercise: assemble an element matrix

Use the **`InternalShape`** class to implement a function named **`makeElmat`** that assembles an element matrix.

Print the element matrix to verify that it is correct.

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

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

## Details

The element matrix to be computed is given by the following equation:

$$\boldsymbol{K} = \int_{\Omega} (\boldsymbol{\nabla} \boldsymbol{N})^T \, \boldsymbol{\nabla} \boldsymbol{N} \, \mathrm{d}\Omega$$

with $\boldsymbol{N}$ a matrix containing the shape functions $\phi$:

$$\boldsymbol{N} = \boldsymbol{N}(\boldsymbol{x}) = \begin{bmatrix} \phi_1(\boldsymbol{x}) & \phi_2(\boldsymbol{x}) & \dots & \phi_k(\boldsymbol{x}) \end{bmatrix}$$

where $k$ denotes the number of nodes.

In 2-D the matrix $\nabla N$ is therefore given by:

$$\nabla N = \begin{bmatrix} \frac{\partial \phi_1}{\partial x} & \frac{\partial \phi_2}{\partial x} & \cdots & \frac{\partial \phi_k}{\partial x} \\ \frac{\partial \phi_1}{\partial y} & \frac{\partial \phi_2}{\partial y} & \cdots & \frac{\partial \phi_k}{\partial y} \end{bmatrix}$$

Note that you only need to implement the function **`makeElmat`**; the required “scaffolding” has already been implemented.

Use the **`matmul`** function from Jem to compute a matrix-matrix multiplication. Use the **`transpose`** member function of the **`Array`** class to get a transposed matrix.

## Example with the **matmul** function:

```
#include <jive/Array.h>
#include <jem/numeric/algebra/matmul.h>

void example ()
{
  using jive::Matrix;
  using jem::numeric::matmul;

  Matrix  a ( 4, 7 );
  Matrix  b ( 7, 4 );
  Matrix  c ( 4, 4 );

  a = 1.0;
  b = 2.0;
  c = matmul ( a, b );              // Returns a  * b
  c = matmul ( a.transpose(), b ); // Returns a' * b

  matmul ( c, a, b ); // Same as above; stores result in c.
}
```

# Example program

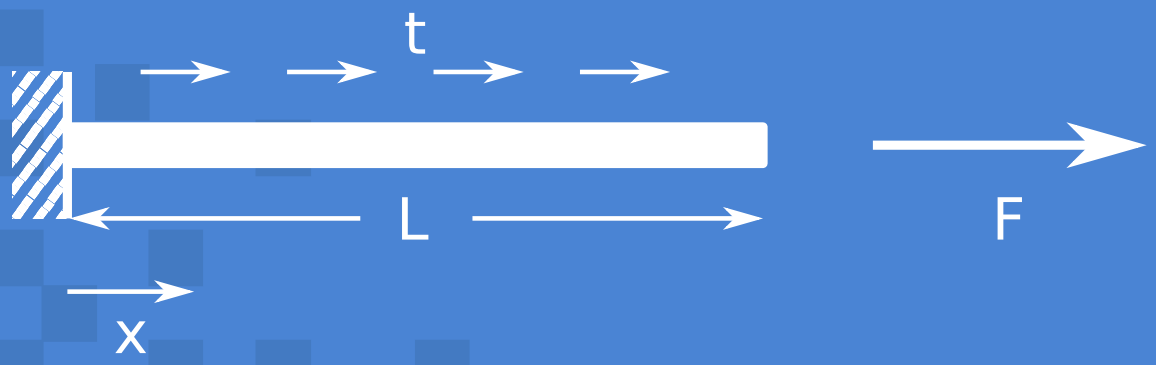
In addition to the exercises, a somewhat larger example program is used to show and practice different Jive components in a larger context.

The example program solves a linear 1-D elasticity problem with the finite element method.

The example program will be familiar if you have participated in the course *Programming in C++* from Dynaflow Research Group. In that case you can skip the next two sub-sections.

# Problem description

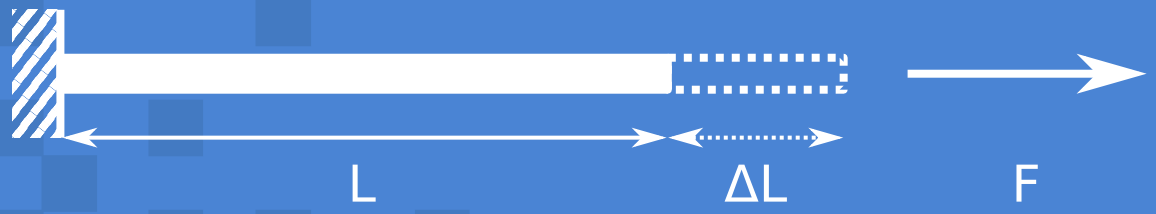
The program computes the deformation of a 1-D elastic bar of length $L$ that is clamped at the left and subjected to a force $F$ on the right. In addition, the bar is subjected to a traction $t$ (force per length unit) along the bar.



When assuming a zero traction, the elongation of the bar is given by Hooke's law:

$$F = E\,A\,\frac{\Delta L}{L}$$

with $E$ the elasticity or Young's modulus; $A$ the cross-sectional area of the bar; and $\Delta L$ the elongation of the bar.



When considering a small section of the bar, the following equilibrium equation can be derived:

$$\frac{d}{dx}(A\,\sigma) = \frac{d}{dx}\left(A\,E\,\frac{du}{dx}\right) = -t(x)$$

with $u$ the displacement field and $t$ the traction acting on the bar.

To compute the displacement field one therefore needs to solve:

$$\frac{d}{dx}\left(A\,E\,\frac{du}{dx}\right) = -t(x) \qquad \text{for} \qquad 0 \le x \le L$$

$$E\,A\,\frac{du}{dx} = F \qquad \text{for} \qquad x = L$$

$$u(0) = 0$$

Note that the Young's modulus and cross-sectional area may vary along the bar.

# Solution procedure

The equations are solved by means of the finite element method.

The first equation is therefore written in its weak form:

$$\int_0^L v \frac{d}{dx}\left(A\,E\,\frac{du}{dx}\right) dx = -\int_0^L v\,t\,dx$$

This equality must hold for all viable test functions $v(x)$.
Because $u$ is given at $x = 0$, $v(0)$ must be zero.

Integration by parts results in:

$$A\,E\;v\frac{du}{dx}|_L - A\,E\;v\frac{du}{dx}|_0 - \int_0^L \frac{dv}{dx}\,A\,E\,\frac{du}{dx}\,dx = -\int_0^L v\,t\,dx$$

Because the first term is equal to $(v(L)\,F)$ and the second term is zero, this can be written as:

$$\int_0^L \frac{dv}{dx}\,A\,E\,\frac{du}{dx}\,dx = v(L)\,F + \int_0^L v\,t\,dx$$

In each element $i$ the displacement field $u$ and its derivative are approximated as follows:

$$u(x) = \boldsymbol{N}_i(x)\boldsymbol{P}_i\,\boldsymbol{a} \qquad \frac{du}{dx} = \boldsymbol{G}_i(x)\boldsymbol{P}_i\,\boldsymbol{a}$$

with

$$\boldsymbol{N}_i(x) = \begin{bmatrix} \phi_{i,1}(x) & \dots & \phi_{i,k}(x) \end{bmatrix}$$

$$\boldsymbol{G}_i(x) = \begin{bmatrix} \frac{d\phi_{i,1}(x)}{dx} & \dots & \frac{d\phi_{i,k}(x)}{dx} \end{bmatrix}$$

with $\phi_{i,j}$ the shape functions, $k$ the number of nodes per element, and $\boldsymbol{a}$ a vector containing the displacements in all nodes.

The matrix $P_i$ selects the displacements in the nodes of element $i$. It is filled with ones and zeroes and has shape $(k \times n)$, with $n$ the total number of nodes.

The matrices $P_i$ are never formed in practice; it suffices to know which element is attached to which nodes.

In Jive the matrices $P_i$ are typically represented by a `DofSpace` instance.

When the test functions are approximated in the same way as the displacement field, then the weak formulation becomes:

$$\boldsymbol{w}^T \sum_{i=1}^{m} \boldsymbol{P}_i^T \int_{x_{1,i}}^{x_{2,i}} \boldsymbol{G}_i^T \, A \, E \, \boldsymbol{G}_i \, dx \, \boldsymbol{P}_i \, \boldsymbol{a} = w_n F + \boldsymbol{w}^T \sum_{i=1}^{m} \boldsymbol{P}_i^T \int_{x_{1,i}}^{x_{2,i}} \boldsymbol{N}_i^T \, t \, dx$$

with $m$ the number of elements, and $x_{1,i}$ and $x_{2,i}$ the lower and upper bounds of element $i$.

As the equality must hold for all test functions, and therefore all vectors $\boldsymbol{w}$, the following equality must hold:

$$\boldsymbol{K}\,\boldsymbol{a} = \boldsymbol{f}$$

with

$$\boldsymbol{K} = \sum_{i=1}^{m} \boldsymbol{P}_i^T \boldsymbol{K}_i \boldsymbol{P}_i \qquad \boldsymbol{K}_i = \int_{x_{1,i}}^{x_{2,i}} \boldsymbol{G}_i^T\, A\, E\, \boldsymbol{G}_i\, dx$$

$$\boldsymbol{f} = \sum_{i=1}^{m} \boldsymbol{P}_i^T \boldsymbol{f}_i + \boldsymbol{F} \qquad \boldsymbol{f}_i = \int_{x_{1,i}}^{x_{2,i}} \boldsymbol{N}_i^T\, t\, dx$$

$$\boldsymbol{F} = \begin{bmatrix} 0 \ldots 0\, F \end{bmatrix}^T$$

The vector $\boldsymbol{F}$ essentially adds the force $F$ to the vector $f$, assuming that the last node is located at the end of the bar.

# Structure of the program

The example program performs five tasks:

1. read model parameters;
2. set up the finite element mesh;
3. assemble the system of equations;
4. solve the system of equations;
5. save the solution to a file.

# Model parameters

The program defines a class named **`Params`** for storing model parameters, such as the number of nodes and elements, and other quantities such as $A$ and $E$.

The initial values for these parameters are read from a text file named **`elastic.pro`** using the **`Properties`** class.

## The finite element mesh

The program creates a uniform mesh consisting of $m$ elements and $n$ nodes.

Each element has size $\Delta x = \frac{L}{m}$.

Each node $i$ has coordinate $x_i = (i-1)\,\Delta x$.

The mesh is stored in two arrays: one containing the node coordinates, and one containing the element connectivities (which element is attached to which nodes).

## Assembly of the system of equations

The program assembles the global stiffness matrix and the external force vector in one loop over the elements.

Linear shape functions are used:

$$\phi_{1,i} = \frac{x_{2,i} - x}{\Delta x} \qquad \phi_{2,i} = \frac{x - x_{1,i}}{\Delta x}$$

with $x_{1,i}$ and $x_{2,i}$ the coordinates of the two nodes attached to element $i$. The element matrices are obtained by means of exact integration.

The **`FlexMatrixBuilder`** class is used to assemble the global stiffness matrix. The external force vector (right-hand side vector) is stored in an array.

A dirty trick is used to enforce the boundary condition $u(0) = 0$: a large value is added to the first diagonal entry in the global stiffness matrix $K$.

The program assumes a constant traction $t$.

## Solution of the system of equations

The system of equations is solved by means of the **`SparseLU`** class. This class implements a sparse LU-factorization algorithm with partial pivoting. It is the most robust solver in Jive.

The solution (displacement vector) is stored in an array.

# Exercise: use the `ElementSet` class

Replace the two arrays **`coords`** and **`elcon`** that represent the finite element mesh by instances of the **`XNodeSet`** and **`XElementSet`** classes.

Update the functions **`assemble`** and **`saveDisp`**.

Exercise location: **`exercise/elastic-1`**.

Solution location: **`exercise/elastic-1a`**.

# Hints

Modify the function **makeMesh** so that it looks like this:

```
ElementSet makeMesh ( const Params& params )
{
  XNodeSet     nodes = newXNodes       ();
  XElementSet  elems = newXElementSet ( nodes );

    :

  return elems;
}
```

Note that each node coordinate must be specified as a **Vector** with length one.

Modify the functions **assemble** and **saveDisp** so that their declarations are as follows:

```
void assemble ( MatrixBuilder&      mbld,
                Vector&             rhs,
                const ElementSet&   elems,
                const Params&       params );

void saveDisp ( const Vector&       disp,
                const ElementSet&   elems );
```

Modify the main function **run** so that it looks like this:

```
int run ()
{
  Assignable<XElementSet>  elems;

    :

  elems = makeMesh ( params );

  assemble ( *mbld, rhs, elems, params );

    :

  saveDisp ( disp, elems );

    :

  return 0;
}
```

Do not forget to add the required include directives and using declarations.

# Exercise: use the `DofSpace` class

Use a `DofSpace` to map the degrees of freedom to the global system of equations and back.

The `elastic` program currently uses the node connectivities to perform this mapping. Use a `DofSpace` instead.

Exercise location: `exercise/elastic-1`.

Solution location: `exercise/elastic-1b`.

# Hints

Add a function **makeDofs** that looks like this:

```
Ref<DofSpace>  makeDofs ( const ElementSet& elems )
{
  NodeSet          nodes = elems.getNodes ();
  Ref<XDofSpace>   dofs  =

    newInstance<DenseDofSpace> ( nodes.getData() );

  // Add the DOF type and the DOFs ...

  return dofs;
}
```

Modify the functions **assemble** and **saveDisp** so that their declarations are as follows:

```
void assemble ( MatrixBuilder&      mbld,
                Vector&             rhs,
                const DofSpace&     dofs,
                const ElementSet&   elems,
                const Params&       params );

void saveDisp ( const Vector&       disp,
                const DofSpace&     dofs,
                const ElementSet&   elems );
```

Modify the main function **run** so that it looks like this:

```
int run ()
{
  Ref<DofSpace>   dofs;

    :

  dofs = makeDofs ( elems );

  assemble ( *mbld, rhs, *dofs, elems, params );

    :

  saveDisp ( disp, *dofs, elems );

    :

  return 0;
}
```

You will need to use the following **`DofSpace`** and **`XDofSpace`** member functions:

- **`addType`**, **`addDof`** or **`addDofs`**, **`getDofIndices`**, and **`getDofIndex`**.

Examples with these functions can be found in the section about the **`DofSpace`** class.

You may assume that the DOF type index equals zero as there is only one DOF type.

The order of the DOFs should have no effect on the results if the program is correct. Add the DOFs in a different order to test this.

# Exercise: use the `InternalShape` class

Use the `InternalShape` class to assemble the element matrices and vectors.

The program currently uses exact integration to compute the element matrices and vectors. Change that to numerical integration.

Exercise location: `exercise/elastic-1`.

Solution location: `exercise/elastic-1c`.

# Hints

Use the **`Line2`** class to create an **`InternalShape`** of the appropriate type in the function **`assemble`**.

Use the function **`getShapeGradients`** to compute the element matrices, and the function **`getShapeFunctions`** to compute the element vectors.

You can re-use the code from the exercise at the end of the section about the **`InternalShape`** class.