FELyX - The Finite Element Library eXperiment

O. König *, M. Wintermantel, N. Zehnder and P. Ermanni

ETH Zurich, Centre of Structure Technologies, Leonhardstrasse 27, CH-8092 Zurich, Switzerland

Abstract

FELyX (Finite Element Library eXperiment) is an object oriented finite element library written entirely in C++. It is designed to meet the demands of researchers and developers who want to have a flexible and easy-to-handle library allowing them to implement their own ideas and projects quickly. Therefore, the main focus of FELyX is laid on computational efficiency and ease of extensibility. FELyX provides all features for structural static analysis of complex real-world structures, as the relevant elements, an interface to a commercial FEA package, smart node reordering strategies and efficient solver algorithms. Beyond this, FELyX is an open source project freely accessible to everyone.

Key words: Finite element library, Structural analysis, Structural optimization, Object oriented, C++, Generic programming

1 Introduction

The FELyX project was started at the Centre of Structure Technologies of the Swiss Federal Institute of Technology in Zurich. The idea and motivation was to provide a Finite Element library with all the advantages of modern generic and object–oriented programming techniques.

The need for such a library arose within ongoing research projects in the field of structural optimization. To find optimum structural designs using numerical

* Corresponding author.
Email addresses: okoenig@imes.mavt.ethz.ch (O. König), wintermantel@imes.mavt.ethz.ch (M. Wintermantel), nzehnder@imes.mavt.ethz.ch (N. Zehnder), ermanni@imes.mavt.ethz.ch (P. Ermanni).

optimization methods, the performance of different designs is often evaluated and compared using the Finite Element Method (FEM). In our research we mainly rely on evolutionary algorithms as numerical optimization methods, typically demanding the evaluation of thousands of different designs. All designs have to be rated with regard to a quality norm (fitness) such as weight, stiffness, strength, or some combination of them. These evaluations normally consume most of the time of an optimization run and pose, even with the computational capabilities of nowadays computers, the main restriction of the applicability of evolutionary methods within the field of structural optimization.

This leads to a strong need for a numerically efficient and flexible FEM tool, able to analyze the structural behavior of complex CAD parts. Nevertheless, the library should be a framework usable in a general Finite Element context, not just restricted to structural mechanics.

Starting with the above requirements we defined the following design criteria in detail:

- *Numerical efficiency:* The computational resources needed for an FE calculation increase nonlinearly with the model size which is the number of nodes and elements. Therefore at some level of discretization, numerical efficiency will always become an issue, no matter how much computational power is available. This is especially true in the field of structural optimization, where, as outlined above, a large number of design evaluations have to be computed. Although increasing only linearly with the model size, memory requirements have to be considered as well.
- *Flexibility in use:* In research and development the programmer is always interested in a tool that allows him to try out and test new ideas and approaches with the least effort possible. Furthermore in the development of structural optimization concepts, it is important to have access to the Finite Element code at all levels. For instance, one can utilize the effect of many similar designs in an optimization process by omitting some parts of the FE analysis. Or looking at gradient–based optimization methods, one can save computational time if the stiffness matrices are directly available [1].
- *Extensibility:* Finite Element Methods, no matter in what context they are used, are always similar at least to some level of abstraction. Therefore the library should be a framework for general finite element analysis, allowing the implementation of new element or analysis types without unnecessary coding (although the developers primary interest was static structural analysis). This can be realized almost perfectly by means of modern generic programming techniques.
- *Easy handling of complex real–world structures:* The Finite Element library should be able to handle complex real-world structures and models as they

are naturally produced with modern CAD packages. Therefore an adequate interface has to be implemented as well.
- *Simple Handling:* Last but not least, the resulting programs should be easy to use which can be achieved by using a programming language with high abstraction capabilities.

In the following sections, we show our approach to realize the above design criteria in the Finite Element Library eXperiment FELyX. Section 2 starts with an introduction to the Finite Element Method and shows the general structure of our library. Section 3 gives an overview on the implemented features and capabilities of FELyX. The practical use of the library is demonstrated on an example in Section 4. Performance comparisons of our library with the commercial FEA package ANSYS [2] are accomplished in Section 5. Finally, Section 7 lists the conclusions and shows the next steps in the development of FELyX.

2 Concept

2.1 Basics of the Finite Element Method

The Finite Element Method is a technique for the numerical solution of a field problem. The solution to a field problem is the spatial distribution of one or more dependent variables. Mathematically, a field problem is described by differential equations or by an integral expression. Finding a solution for these equations for complex spatial domains can be very demanding. The Finite Element Method achieves the solution of field problems in any arbitrary spatial domain. Because it undergoes always the same steps this method is easy to automate and can be coded efficiently in computer programs. That is why FEM is widely used.

The idea of the Finite Element Method is to subdivide a given complex spatial domain into smaller elements, whose behavior is well understood. The variation of the field quantity in each element is assumed small and is therefore mostly described by polynomials up to order two or three. The actual variation within the region spanned by a single element is generally more complicated, that is why the FEM normally provides only approximate solutions.

Within a single element the field quantities are given as functions of unknown values in the nodes. The nodes are well defined points mostly lying on the border of the domain spanned by an element. The individual elements are connected at the nodes to form the mesh. The mesh forms a surrogate model of the real domain and its physical behavior. Such a model is in a mathemat-

ical sense a system of linear equations to be solved for the unknown nodal values [3].

Although FELyX provides a framework for general finite element analyses the authors main interest lies in the field of static structural analysis. For this reason and to simplify the explanations the remaining sections will focus on static structural analysis.

The stress–state in a loaded structure is a typical field problem applied to the domain filled by the structure. The basic steps of a structural finite element analysis are as follows. First of all, the mesh must be built in order to discretize the structure in finite elements. After this step all nodal coordinates are determined and all elements are assigned their nodes. Now, all elements are geometrically defined. Additionally, all elements must contain the material properties of the domain they span. After assigning the geometrical and physical properties the surrogate model of the structure is finished.

The loads applied on the real structure must be translated to the surrogate model. Distributed loads like pressure must be mapped to forces acting on nodes. Geometric boundary conditions are defined as predefined displacements of single nodes. One can distinguish two different types of such boundary conditions. A geometric boundary condition is called homogeneous if it specifies a zero displacement, else it is called inhomogeneous.

The surrogate model of the whole loaded structure is now finished and the appropriate mathematical function can be built up. Every single finite element must fulfill the equations of equilibrium. All stresses acting on the boundaries of an element are represented as statically equivalent nodal forces. The stress state within the element is represented by a function, mapping nodal displacements to nodal forces. This mapping is often given as a system of linear equations, expressed through the so-called Element Stiffness Matrix (ESM).

The present load case is given as the following matrix equation containing the displacement vector $\{u\}$, the force vector $\{f\}$, and the Global Stiffness Matrix (GSM) $[K]$.

$$[K]\{u\} = \{f\} \tag{1}$$

The GSM $[K]$ is an assembly of the individual ESMs. The nodal displacement vector $\{u\}$ is unknown except for the specified geometric boundary conditions. If homogeneous boundary conditions $u_i = 0$ appear, the linear equation they appear in is trivial and can be removed from the global system of linear equations. Inhomogeneous boundary conditions $u_i \neq 0$ can be represented as a superposed load case $\{f_d\}$ and must be added to $\{f\}$. The force vector $\{f\}$ contains the nodal forces derived from the applied load case and is therefore

known.

Eq. (1) can now be solved for the vector of nodal unknowns $\{u\}$. Then the stress state within the single elements is a function of the nodal displacements $\{u\}$ and is easily evaluated once the primary solution is known.

2.2 Structure of FELyX

FELyX is implemented in the programming language C++. This is a widely available and standardized language providing high levels of abstraction for object oriented and generic programming. Furthermore there is a large amount of libraries of generic data containers and algorithms as for example the Standard Template Library (STL) [4] or the Boost Libraries [5]. With all these features, C++ supports efficient programming techniques.

As comes out from Section 2.1 a finite element library must be able to store large amounts of data, to hold the entire information of a discretized surrogate model. FELyX stores all this data in STL vectors templated with the appropriate data objects. The different objects of a FELyX surrogate model are the nodal coordinates, the elements, the material properties, the loads, the boundary conditions, and the nodal coordinate systems. The dependencies of these objects are shown in Figure 1. The different lists in Figure 1 symbolize the STL vectors containing objects of the FELyX library. The arrows stand for the data access of individual objects. This access is implemented with pointers which is substantially faster than indices lookup mechanisms. In such an architecture data is not stored redundantly.

A central object in the architecture of FELyX is the class `Element`. Every element has a unique set of data defining its individual physical behavior. FELyX organizes this data in three subsets, the material, general properties, and the nodes. As can be seen in Figure 1 different elements can access the same subsets of data. An object `Material` contains the physical material properties and methods providing different stress-strain relations. The class `PropertySet` is a pool for element-type dependent data such as thickness of a shell, cross-sectional area of a beam, etc. The shape of an element is defined by the coordinates of its nodes.

Besides its coordinates a node can have boundary conditions applied on it. These boundary conditions can be applied in other directions than the global cartesian ones. In that case the node is assigned a coordinate system, giving that direction.

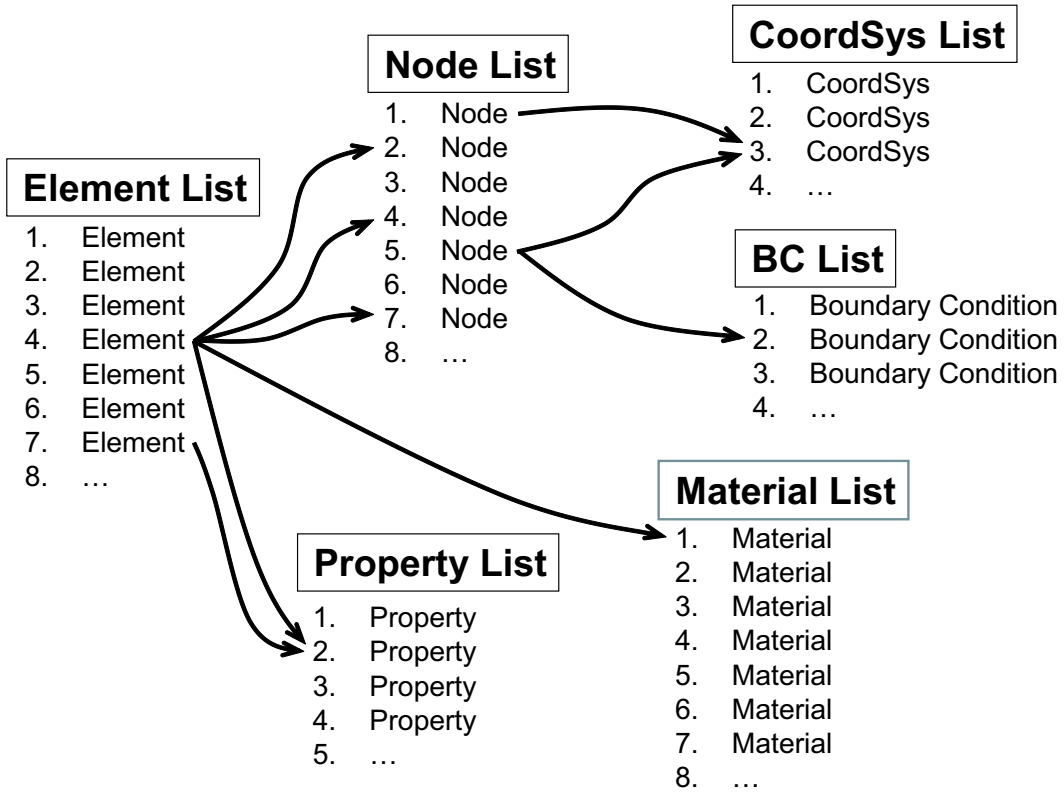

Fig. 1. Data structure of FELyX. The arrows represent direct access to objects.

3 Features

3.1 Matrix handling and linear algebra operations using MTL

Matrix handling and linear algebra operations build the backbone for every Finite Element analysis. The way how matrices and vectors are stored and processed is absolutely crucial for the potential and power of a Finite Element library in terms of computational efficiency, memory needs and usability.

The main focus for such a linear algebra library must lie in the efficient handling of large and sparsely populated matrices. The library must support different shapes and storage schemes for the matrices as banded, sparse, or envelope. In addition, building the element stiffness matrices invokes a lot of operations on small dense matrices. The library should provide the basic matrix operations and a comfortable interface to apply them.

After evaluating the C++ matrix libraries available and even starting to design our own matrix library we finally decided to use the Matrix Template Library (MTL) [6–8]. MTL fulfills our core requirements by a generic approach for expressing high performance numerical linear algebra routines for a class of

dense and sparse matrix formats and shapes.

As with STL, algorithms are separated from data structures through the use of generic programming techniques. Through the use of templates, the matrix types in MTL are parameterized in terms of the elements data type, the matrix storage scheme, the matrix format and the matrix orientation. That way, all the matrix types needed for Finite Element analysis are provided. Despite this high level of abstraction, MTL is considered to be fast. Basic linear algebra operations such as multiplication, inverse etc. are implemented in a generic way so they can be used for all matrix and vector formats. In FELyX, the performance of MTL is further improved using blocked routines from the Automatic Tuned Linear Algebra Software (ATLAS) [9] in matrix multiplications and solver routines.

A disadvantage of MTL for our needs lies in its performance driven user interface, that does not provide operator overloading and requires some more coding than with other libraries.

3.2 Data type as template argument

The templating capabilities of C++ and the structure of MTL give the possibility to change the precision of floating point numbers. A finite element analysis within FELyX can be switched from double to single precision for performance reasons. The importance of this subject is covered in detail in Section 5.

3.3 Element implementations

In FELyX the common elements for static structural analysis are implemented. In two dimensional space:

- 2–node link element with two translational DOFs per node
- 2–node beam element with two translational and one rotational DOFs per node
- Triangular 6–node plane element with quadratic displacement behavior and two translational DOFs per node
- Quadrilateral 4–node plane element with linear displacement behavior and two translational DOFs per node
- Quadrilateral 8–node plane element with quadratic displacement behavior and two translational DOFs per node

In three dimensional space:

- 2–node link element with three translational DOFs per node
- 2–node beam element with three translational and three rotational DOFs per node
- Tetrahedral 10–node element with quadratic displacement behavior and three translational DOFs per node
- Quadrilateral 8–node element with linear displacement behavior and three translational DOFs per node
- Quadrilateral 20–node element with quadratic displacement behavior and three translational DOFs per node
- 8–node shell element with quadratic displacement behavior, three translational DOFs and three rotational DOFs per node

All elements are derived from an abstract base class `Element`. This class contains all data members and functions which are common to most of the derived elements. In order to minimize code duplication most of the functions are implemented in the base class.

The derived element classes contain only data members and functions which are unique to their special type (e.g. number of nodes). The amount of data and functions in the derived element classes is kept as small as possible. The data members are the same for all instances of a derived element and therefore stored as static data members.

3.4 Assembly and application of boundary conditions

Assembly functions in FELyX guarantee that the element stiffness matrices of a mix of different elements with different numbers of DOFs are appropriately assembled to the global stiffness matrix (GSM). In addition, homogeneous geometric boundary conditions are already considered during assembly, resulting in a smaller global stiffness matrix.

The implementation of these features is based on a mapping process taking place before assembly and determining the appropriate index in the GSM for each degree of freedom of each node. The indices are evaluated using element type information and homogeneous geometric boundary conditions, omitting all non–necessary rows and columns of the GSM. During assembly, the values of element stiffness matrices can then be added at the appropriate position in the GSM using this index. To make the system ready to solve, the inhomogeneous boundary conditions such as forces on nodes or non–zero displacement constraints are applied.

Displacement constraints in directions other than the global cartesian ones are handled by coordinate transformations of the corresponding element stiffness matrices as proposed in [10].

3.5 Linear solvers and node reordering algorithms

The linear equation system (1) of typical linear structural Finite Element calculations is commonly solved using either direct (*Gauss/Cholesky–type*) or iterative solvers. Direct solvers scale with $O(n^3)$ in contrast to iterative ones that scale with $O(n)$ where n is the number of DOFs. This means that there is a break–even point in the model size from where on it is favorable to use iterative instead of direct solvers.

The matrix K of typical structural Finite Element systems has only very few non–zero entries, since each element contributes to the matrix K only at the positions of its own DOFs. Therefore there will be only non–zero entries in K between the DOFs of nodes belonging to the same element. Since most Finite Elements of an actual problem discretization are only neighbors to few other elements compared with the number of elements of the whole system, the population of K with non–zero entries is very sparse. Further the structure of K depends crucially on the sequence of the DOFs in the equation system. Since the sequence of the DOFs can be chosen freely, this opens up the possibility to affect this DOF–sequence, i.e. the ordering of the nodes, in a way that is favorable for the solution process.

Numbering the nodes heuristically in a way, that nodes connected to one another have numbers that are close together, results in a matrix structure with non–zero entries only around the diagonal. The maximum distance of a non–zero entry from the diagonal is denoted as the maximum bandwidth $b_{\max}$. Modifying a standard Cholesky decomposition in a way that it only operates on the band around the diagonal of the matrix delimited by the maximum bandwidth leads to a solver that scales with $O(nb_{\max}^2)$ instead of $O(n^3)$. Furthermore one has to store only the elements within the band, reducing the memory needs drastically.

But now, an algorithm that optimizes the ordering for a small maximum bandwidth is needed. This problem belongs to the field of graph theory, where graphs are abstract structures consisting of so–called vertices (equivalent to the nodes in the Finite Element context) and edges connecting these vertices. There is the rather heuristical but well established Cuthill–McKee algorithm [11] reordering vertices for minimum bandwidth very effectively. In FELyX, this algorithm is provided through the graph library of Boost [12].

Looking at the band structures of K matrices one notes that for most Finite Element problems, there are only few rows with non–zero entries up to the edge of the maximum bandwidth. Therefore it is a good idea to store the entries of each line only up to the last non–zero entry in a so–called Skyline storage format. Further one has to modify the Cholesky decomposition in a

way operating only within the Skyline of the matrix. This leads to a slight augmentation in terms of memory administration, which is far overcompensated by the operations saved for the Cholesky decomposition.

Analyzing this new Skyline solvers reveals that they do not scale anymore with the maximum bandwidth but as $O(nb_{rms}^2)$ with

$$b_{rms}^2 = \left(\frac{1}{n} \sum_{i=1}^{n} b_i^2 \right)^{\frac{1}{2}} \tag{2}$$

the root-mean-square bandwidth (RMS-bandwidth) where b_i is the i^{th} bandwidth of the matrix. So it holds $b_{rms} \leq b_{max}$. Therefore one is more interested in a node reordering algorithm providing minimal RMS-bandwidth which is not necessarily identical with minimum maximal bandwidth. And in fact graph theory knows an also rather heuristical Sloan algorithm [13,14], providing for almost all finite element discretizations a smaller RMS-bandwidth than Cuthill–McKee even tough the maximum bandwidth is typically larger. The algorithm has been implemented in FELyX as an extension of the Boost graph library [12]. This Sloan node reordering algorithm and the above described Skyline solver combined with efficient matrix handling as described in Section 3.1 leads to state of the art performance results for the direct solver in FELyX as described in Section 5.

Iterative solvers are also available in FELyX by integration of the so–called Iterative Template Library (ITL) [15]. Their performance most crucially depends on effective preconditioning and fast matrix multiplication. The latter is provided by effective matrix handling. Doing mainly multiplications of sparsely occupied matrices during the solution process even makes the reordering of the nodes obsolete since they can be done very efficiently using a sparse matrix storage format where each non–zero entry is just stored in a linear list together with it's position within the matrix.

Since the authors interest is mainly in problem sizes favoring direct solvers, there is not yet invested any effort into FELyX in terms of efficient preconditioning. But nevertheless in the context of structural optimization the iterative solver is sometimes interesting. This holds since in an optimization run similar designs have to be evaluated often. This similarity is also reflected in the linear system that has to be solved and the old solution can be used efficiently as starting point for the iterative process.

3.6 Interface to ANSYS

One of the major requirements for FELyX is its ability to handle complex real–world structures. To achieve this goal, we implemented an interface to the commercial FEM program ANSYS [2]. Starting from a CAD design of a structure, the geometry is imported to ANSYS, where all the preprocessing of the FEM model takes place. This includes the discretization of the geometry, definition of material and element properties as well as the application of boundary conditions. The resulting FE model is then exported to a text file, using the ANSYS archive format. This file can be read into FELyX, storing elements, nodes, materials, element properties, coordinate systems, and boundary conditions in the appropriate STL vectors as introduced in Section 2.2. All ANSYS interface classes are derived from general IO classes, facilitating the implementation of interfaces to other programs.

3.7 User interface `FelyxObject`

For easier application and adaption to specific tasks we built an interface class `FelyxObject`. An instance of this class contains all containers necessary to store an entire Finite Element model. Furthermore the class contains all functions to evaluate the stored model.

If one needs to expand the ability of the `FelyxObject`, e.g. in a special optimization problem, a new class should be derived. The data members are directly accessible from any derived class, because they possess protected accessibility rights.

The utilization of the `FelyxObject` is shown in more detail in the following Section.

4 Example

In this Section an introduction is given how FELyX can be used. First, the `FelyxObject` is presented in a simple example thereby being far from revealing all its functionality. Then it is outlined how custom functionality can be implemented into FELyX by creating a new class derived from the `FelyxObject` class.

The `FelyxObject` is a class containing all data and functionality of a FEM calculation. In terms of data members these are:

- A list of nodes
- A list of individual node coordinate systems
- A list of boundary conditions of a problem
- A list of all elements
- A list of materials assigned to the elements
- A list of other individual properties of elements
- The nodal solution vector

Besides the nodal solution vector these data members are initialized by either reading an ANSYS model or filled by hand for simple calculations. On the functional side the `FelyxObject` has class members managing

- File input/output functionality for the above listed data members
- All kind of functions to manipulate the data members
- A function to perform different node–reordering algorithms
- A direct solver function
- An iterative solver function
- A stress evaluation function

There are some further functions and data members that handle the level of information printed during the calculations, that give model statistics, etc.

In terms of code a simple FE–calculation from an existing ANSYS archive model would look just like this.

```
// Including the FelyxObject header
#include "FelyxObject.h"

//Setting the FELyX namespace
using namespace felyx;

int main(){
  //Initializing the FelyxObject
  FelyxObject
  FEM( "my_FEdata", "~/my_FEdir/", 3 );

  //Apply Sloan node-reordering algorithm
  FEM.NodesReordering( "sloan" );

  //Solving the system with a direct solver
  FEM.DirectSolver();

  //Evaluating the stresses
  FEM.EvalStresses();
```

```
  //Saving the model and the results
  FEM.SaveAnsysModel();
  FEM.SaveResults();

  return 0;
}
```

First a `FelyxObject FEM` is generated and the data members are initialized with the data from a file. The third argument in the initialization of the `FelyxObject` is called the noise level that determines how much messaging is performed when using the `FelyxObject`. `FEM.NodesReordering("sloan")` reorders the nodes using the Sloan–algorithm. This is followed by solving the problem with a direct solver through the function call `FEM.DirectSolver()` and stress evaluation via `FEM.EvalStresses()`. Finally the model as well as the results are saved by `FEM.SaveAnsysModel()` and `FEM.SaveResults()`.

In addition to simple FE–calculations, custom functionalities can be implemented in an elegant way in C++ by deriving the `FelyxObject`. In terms of code this could look like this:

```
#include "FelyxObject.h"

using namespace felyx;

class myFEobject:FelyxObject{

public:
  anyType myFunction(anyType arg1, ...)
    {
      /* The code to manipulate the
         FelyxObject */
    }
}
```

The new class `myFEobject` provides the newly implemented functions as well as all the old functionality of the `FelyxObject`. This reveals how easy custom Finite Element codes can be implemented in FELyX.

5 Performance experiments

In this Section, the performance of FELyX in comparison to the commercial FEM software ANSYS [2] is demonstrated. ANSYS represents a high–end software for structural Finite Element analysis. Performance comparisons focus

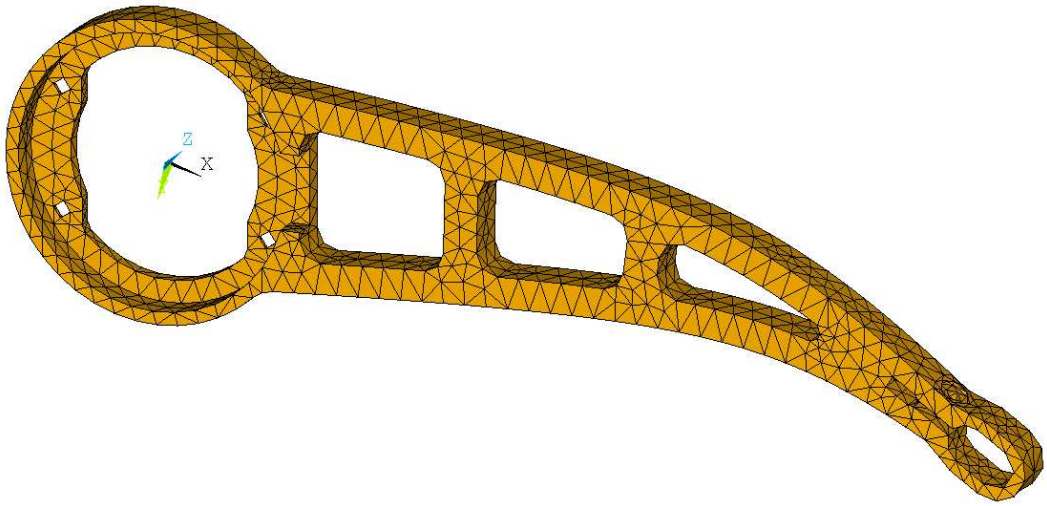

Fig. 2. Finite Element model of the cantilever arm with 3559 elements (22650 DOFs).

on problem sizes favoring direct solvers as outlined in Section 3.5. Therefore, the direct skyline solver of FELyX is compared to the direct frontal (wavefront) solver and direct sparse solver of ANSYS. For verification purposes, we also list the measured runtimes of the iterative preconditioned gradient solver (PCG) of ANSYS.

5.1 Description of test models

Three test structures based on CAD geometries are used. The Finite Element models were built as described in Section 3.6 using preprocessor functionality of ANSYS. In the following the models are described in detail:

Solid model of a cantilever arm. The cantilever arm as shown in Figure 2 represents a typical solid CAD part consisting of a single volume (courtesy of Tribecraft AG, Zurich). The volume is meshed using 10–node tetrahedrons and loaded with a bending force on one side.

The mesh size for this structure is varied, resulting in 26 different FE models from 2657 up to 6833 elements. This corresponds to models with 17385 to 39321 DOFs. With this models, the scaling of different solvers with respect to the number of DOFs will be investigated.

Mixed 3D model. The intention of the mixed model as shown in Figure 3 is to demonstrate the correct interaction of different element types with different number of DOFs in a single FE model. It is based on a fictive geometry

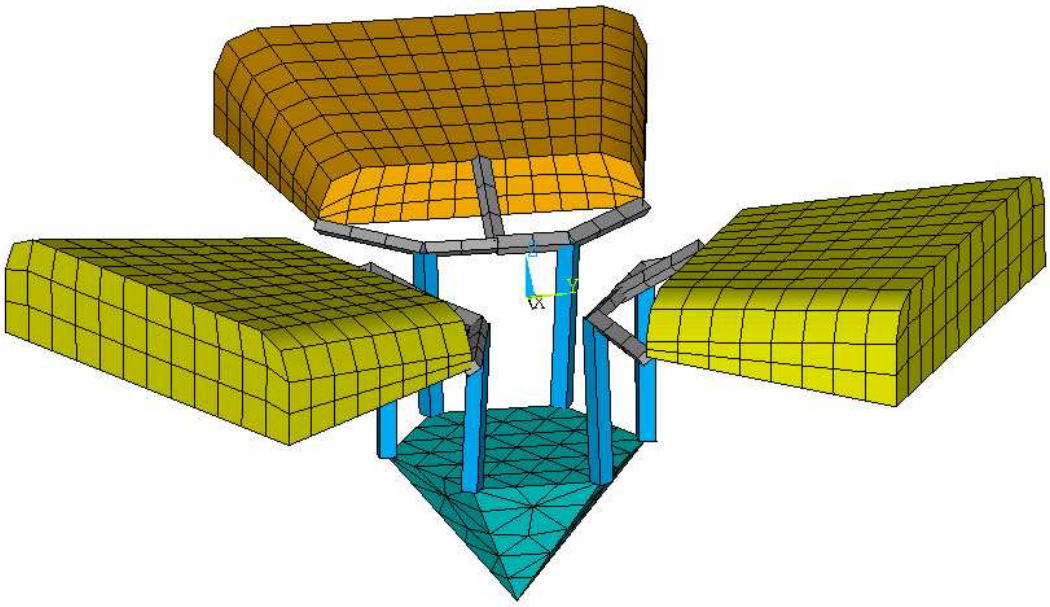

Fig. 3. Finite Element model of the mixed test model with 1133 elements (12568 DOFs).

and consists of 2–node links, 2–node beams, 10–node tetrahedrons, 8–node bricks and 20–node bricks. The three block solids are clamped on one side and deformation results by a downward force at the hexagonal solid in the center. The model consists of 1133 elements and 4298 nodes, resulting in 12568 DOFs.

Shell model of a wind turbine wing. The wing of a wind turbine shown in Figure 4 finally represents a thin walled structure that is analyzed with 8–node shell elements (courtesy of Aventa AG, Winterthur). The circular junction at one side of the wing is clamped, and the wing is loaded with aerodynamic forces. It has an overall length of 6m and is discretized with 1092 elements and 3318 nodes, resulting in 19404 DOFs.

5.2 Test conditions

To compare the performance of FELyX against ANSYS, identical conditions in terms of hardware, operating system, and time measurement are accomplished for both programs. Therefore, runtime measurements include the following steps for both programs :

(1) Profile/bandwidth minimization by reordering nodes or elements
(2) Building of element stiffness matrices and assembly to the GSM
(3) Application of boundary conditions

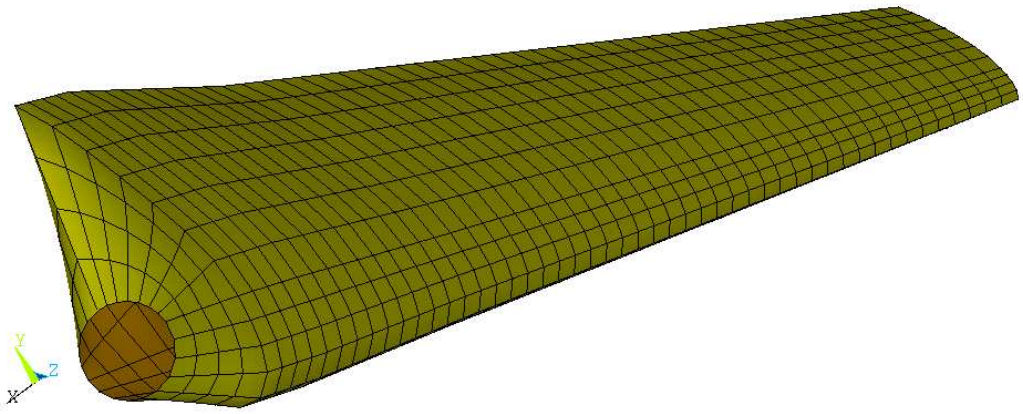

Fig. 4. Finite Element model of the wind turbine wing with 1092 elements (19404 DOFs).

(4) Solution of the system of linear equations

Loading and storing of the different models is not included in runtime measurements.

Test programs for FELyX were compiled with GNU g++ 2.95.3. ATLAS/BLAS routines for the Skyline solver and matrix multiplications were included as introduced in Section 3.1. For time measurement we used the Boost timer functions.

Runtimes in ANSYS were evaluated by simply measuring how long the execution of the `SOLVE` command took. This can be done by using a `*GET` command for the cpu time. The tests were run with ANSYS 6.0, starting the program in batch mode from command line and reserving 712 MB memory for the workspace and 128 MB memory for the database. For the iterative preconditioned gradient solver in ANSYS, all evaluations were run with a solver tolerance value of 10^{-8}.

All evaluations were run under SuSE Linux 8.0 on an AMD Athlon XP 2000+ workstation with 1.5 GB RAM.

5.3 Results

Evaluations of the test models described in Section 5.1 under the test conditions outlined in Section 5.2 lead to the following results. All three test models are analyzed using the FELyX skyline solver with double and single precision, which is easy possible as mentioned in Section 3.2. In the tables 1, 2 and 3 the resulting runtimes are compared with ANSYS. There, the models are evaluated using the direct frontal and sparse solvers and, for verification

Table 1
Performance results for the cantilever arm with 3559 elements (22650 DOFs).

Program & Solver	Runtime [s]	Max. def. [mm]
FELyX skyline double precision	9.42	3.83
FELyX skyline single precision	6.66	3.85
ANSYS frontal	9.71	3.83
ANSYS sparse	9.78	3.83
ANSYS PCG (iterative)	10.86	3.83

reasons, the iterative preconditioned gradient solver (PCG). The third column in these tables lists the maximum deformation for each solution to compare the accuracy of the results.

Results for the cantilever arm model evaluated in Table 1 show that for this structure FELyX is fully competitive to ANSYS in terms of solution times. Switching to single precision, runtimes can be reduced by another 30%, worsening the accuracy of the maximum deflection by less than 1%.

Deformation results for the 3D mixed model in Table 2 confirm, that FELyX accurately evaluates the deformations of such a mix of different element types with different numbers of DOFs. Switching to single precision again, maximum deformation changes with 3% more significantly than for the cantilever arm. Although the mixed model consists of little more than half of the DOFs of the cantilever arm model, computation times are around 10 seconds for both models. The reason lies in the more complex geometry of the mixed model. Node numbers can not be enumerated from one side to the other for this structure, resulting in a much larger profile than for the cantilever arm. This fact is underlined by the good performance of the ANSYS PCG solver for this model that does not depend on profile/bandwidth minimization. The runtimes of the direct solvers show that FELyX is also competitive for this kind of problems.

Finally, maximum deformation values in Table 3 show that thin walled models consisting of shell elements are also evaluated accurately in FELyX. But analyzing this structure with single precision values results in a significant variation of the maximum deformation by 7.5% compared to double precision results. Using single precision values can lead to substantially large errors. Therefore, results have to be checked carefully. But nevertheless, in cases where accuracy of the obtained results is not that important, switching to single precision can save computation time. Looking at the measured runtimes for the wing structure, the competitive abilities of FELyX in comparison to

Table 2
Performance results for the 3D mixed model with 1133 elements (12568 DOFs).

Program & Solver	Runtime [s]	Max. def. [mm]
FELyX skyline double precision	13.43	3.49
FELyX skyline single precision	9.55	3.36
ANSYS frontal	10.23	3.49
ANSYS sparse	10.26	3.49
ANSYS PCG (iterative)	7.23	3.49

Table 3
Performance results for the wing model with 1092 elements (19404 DOFs).

Program & Solver	Runtime [s]	Max. def. [mm]
FELyX skyline double precision	12.53	538.75
FELyX skyline single precision	6.9	578.77
ANSYS frontal	8.92	538.75
ANSYS sparse	13.06	538.75
ANSYS PCG (iterative)	30.31	538.75

ANSYS are outlined again.

Scalings of solution times for varying mesh sizes of the cantilever arm model are compared in Figure 5. The viewed range of number of DOFs reflects model sizes interesting for our projects in structural optimization. Basically, the plots show one more time, that the solver performance of FELyX is competitive compared with ANSYS's direct solvers. For this rather good conditioned problem it is shown that the skyline solver of FELyX tends to be even faster for small problems. On the other hand it is also apparent that the solver in FELyX scales worse for increasing problem sizes than the direct solvers in ANSYS. In our opinion this fact is founded on better compiler performance and better integration of blocked linear algebra routines in ANSYS. Figure 5 also shows that direct solvers are competitive to iterative solvers for the problem sizes considered.

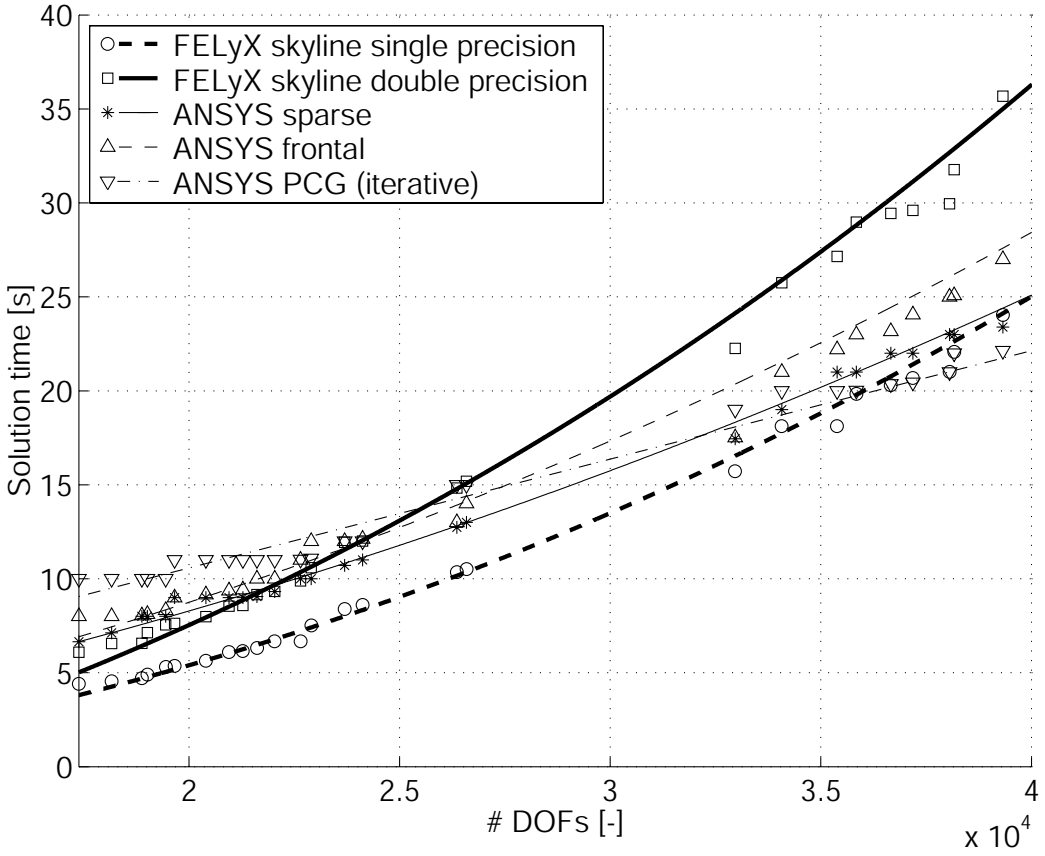

Fig. 5. Scaling of solution times for varying mesh sizes of the cantilever arm model.

6 Technical Remarks

FELyX was developed under the Linux operating system and the GNU compiler gcc 2.95.3. It is also tested under gcc 3.2 (Linux and Solaris) and under Intel C++ 6.0 (Linux). But porting the library to other modern C++ compilers should not be a big task.

For documentation, we use the in–code system doxygen [16], that automatically creates html and latex pages from the documented source code. This is the only way for a small group of developers to maintain an up–to–date documentation.

The whole library is released on Sourceforge [17] under GNU General Public License [18]. Any contributions from you would be appreciated very much and everybody is welcomed to join the developer team of FELyX.

7 Conclusions

FELyX is a numerically efficient and expandable C++ library for Finite Element analysis. The library has been developed using object-oriented and generic programming techniques to the best of our knowledge. We are convinced that with the limited manpower involved in this project only this approach made it possible to achieve the design criteria layed out in Section 1. Section 5 shows that FELyX is competitive to well known commercial Finite Element analysis programs in terms of numerical efficiency. This also proves one more time that object-oriented and generic programming approaches in C++ do not have performance disadvantages compared to other languages or programming techniques. Flexibility in use is mainly provided through the FELyX object as described in Section 3.7, that can easily be derived and extended with custom functionality. Since the library is distributed under GNU Public License and therefore relies only on freely available libraries, all code can be expanded and tailored to specific needs. Together with the high level of abstraction of this programming approach this provides the demanded ease of extensibility. Finally, it is shown that complex real-world structures can be handled in FELyX using interfaces to the commercial computer aided engineering world.

FELyX is still under active development. It will be used extensively in structural optimization since it provides the basis for the authors PhD theses. In the near future the following features and capabilities will be added by different people:

- Triangular shell elements as needed to mesh complex shell structures
- Multi layered and single layered shell elements together with orthotropic and transversely isotropic materials to analyses composite structures
- Capabilities to calculate the time-dependent flow of liquid through a porous media (Darcy's law)
- Capabilities for eigenfrequency analysis

Although FELyX was developed with structural analysis in mind, the architecture of this library should be useful for many people working in the broad field of Finite Element Analysis. We invite everybody to include their own ideas into FELyX and any contribution to improve the library is welcomed.

Acknowledgements

This work was supported by the Swiss National Foundation (SNF-project 21-66879.01) and by the Swiss Federal Institute of Technology Zurich (TH-project

01112).

References

[1] M. Bendsoe, N. Kikuchi, Generating optimal topologies in structural design using a homogenization method, Computer Methods in Applied Mechanics and Engineering 71 (1988) 197–224.

[2] ANSYS - Commercial FEA package, url: `http://www.ansys.com` (Version 6.1, 2002).

[3] R. Cook, D. Malkus, M. Plesha, R. Witt, Concepts and Applications of Finite Element Analysis, 4th Edition, John Wiley and Sons, 2002.

[4] Standard Template Library Programmer's Guide, url: `http://www.sgi.com/tech/stl/` (1993).

[5] Boost Libraries, url: `http://www.boost.org` (Version 1.28.0).

[6] A.Lumsdaine, J. Siek, L. Lee, The Matrix Template Library (MTL), url: `http://www.osl.iu.edu/research/mtl` (Version 2.1.2-21).

[7] J. Siek, A modern framework for portable high performance numerical linear algebra, Master's thesis, University of Notre Dame, Indiana (April 1999).

[8] J. Siek, A. Lumsdaine, The matrix template library: Generic components for high-performance scientific computing, Computing in Science and Engineering 1 (6) (1999) 70–78.

[9] Automatically Tuned Linear Algebra Software (ATLAS), url: `http://math-atlas.sourceforge.net` (Version 3.3.15).

[10] K. Bathe, Finite Elemente Methoden, Springer-Verlag Berlin Heidelberg New York, 1990, Ch. 4, p. 156.

[11] E. Cuthill, J. McKee, Reducing the bandwidth of sparse symmetric matrices, in: Proc. ACM Nat. Conf, Association of Computing Machinery, New York, 1969.

[12] J. Siek, L. Lee, A. Lumsdaine, The Boost Graph Library, C++ In-Depth Series, Addison Wesley, 2002.

[13] S. W. Sloan, An algorithm for profile and wavefront reduction of sparse matrices, Int. J. Numer. Methods Eng. 23 (1986) 239–251.

[14] S. W. Sloan, A fortran program for profile and wavefront reduction, Int. J. Numer. Methods Eng. 28 (1989) 2651–2679.

[15] A.Lumsdaine, J. Siek, L. Lee, The Iterative Template Library (ITL), url: `http://www.osl.iu.edu/research/itl` (Version 4.0.0).

[16] D. Hesch, Doxygen - A documentation system for C++, C, Java, IDL and others, url: `http://www.stack.nl/ dimitri/doxygen` (Version 1.2.14, 2002).

[17] O. Koenig, M. Wintermantel, N. Zehnder, The Finite Element Library Experiment, url: `http://felyx.sourceforge.net` (Version 0.21).

[18] Free software foundation, Inc., 59 Temple Place Suite 330, Boston, MA 02111, USA, GNU General Public License, url: `http://www.gnu.org/licenses/gpl.html` (Version 2, 1991).