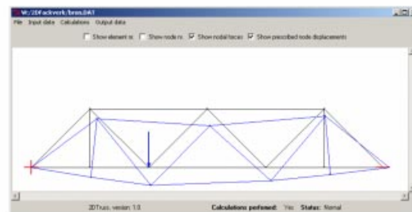
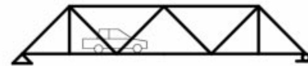
LUND UNIVERSITY





METHODS FOR INTEGRATING FORTRAN BASED FINITE ELEMENT APPLICATIONS IN A SCRIPTING ENVIRONMENT

MARTIN ROOS

Structural Mechanics

Master's Dissertation

Structural Mechanics

ISRN LUTVDG/TVSM--05/5129--SE (1-XXX)
ISSN 0281-6679

METHODS FOR INTEGRATING FORTRAN BASED FINITE ELEMENT APPLICATIONS IN A SCRIPTING ENVIRONMENT

Master's Dissertation by

MARTIN ROOS

Supervisors:

Jonas Lindemann and Ola Dahlblom,
Div. of Structural Mechanics

Copyright © 2005 by Structural Mechanics, LTH, Sweden.
Printed by KFS I Lund AB, Lund, Sweden, Month, 2005.

For information, address:
Division of Structural Mechanics, LTH, Lund University, Box 118, SE-221 00 Lund, Sweden.
Homepage: http://www.byggmek.lth.se

Preface

The work presented in this Master's thesis was carried out at the Division of Structural Mechanics at Lund's University, Sweden, during September 2004 - February 2005.

I would like to express my gratitude to my supervisors, Professor Ola Dalhblom and PhD Jonas Lindemann. Ola Dalhblom for helping me to find this Master's thesis and start it up. Jonas Lindemann for supporting me during the entire process and for the discussions we had regarding the topics in this work.

I am also very grateful to have had the possibility to use the resources at the Division of Structural Mechanics and thanks to Mr Bo Zadig for helping me with the graphics and printing.

Finally I would like to thank all other students in the Master's thesis room for giving me motivation and laughter.

Lund February 2005,

Martin Roos

Abstract

The course Software Development for Technical Systems at the division of Structural Mechanics teaches students techniques for implementing finite element applications with both user interface and computational code. Currently Borland Delphi environment is used for the user interface and Fortran is used for the computational part. This master's thesis is an effort to study methods of integrating Fortran with script languages and user interface to create more flexible and easier to use software. An interesting candidate for user interface and computational environment is Python. Python is a script language, which is easy to use and learn for non-programmers and yet powerful enough to use as a scripting environment for computational codes.

A finite element application is developed where the computational code is written in Fortran and the user interface in Python. The main task to solve is how to link Fortran and Python. To perform the link a tool, Fortran to Python interface generator (F2PY) is used after an evaluation of different tools. The test application shows that F2PY creates an interface in a single step, no advantages of Fortran are lost, all major Fortran functions are supported and a F2PY created module is as easy to use as a built-in function in Python. The disadvantage of F2PY is that it need five prerequisites. Overall F2PY fulfills all requirements on a link between Fortran and Python. Python has all features a scripting languages shall have, easy code structure, rapid development and the ability to glue components together. Python also gives the user possibility to create a graphical user interface (GUI) with or without a GUI-builder. Finally all parts that are used are platform independent. This result in that it is recommended to replace Borland Delphi with Python in the course Software Development for Technical Systems.

Sammanfattning

Kursen Programutveckling för Tekniska System på avdelningen för Byggnadsmekanik lär ut tekniker för att implementera finita element program som betsår av en beräkningsdel och ett användargränssnitt. För tillfället används utvecklingsmiljön Borland Delphi för användargränssnittet och Fortran för beräkningskoden. Detta examensarbete syfte är att studera metoder för att integrera Fortran med skriptspråk och användargränssnitt för att utveckla mer flexibla och lättanvända applikationer. Ett intressant programmeringsspråk för användargränssnittet och som beräknings skal är Python. Python är ett skriptspråk som är användarvänligt och lätt att lära för nybörjare men ändå tillräckligt kraftfullt för att användas som en skript miljö för beräkningskod.

Ett finita element program är utvecklat där beräkningskod är skriven i programmeringsspråket Fortran som integreras i scriptspråket Python, huvuduppgiften är att skapa en fungerande länk mellan Fortran och Python. Gränssnittet till länken skapas med ett verktyg, Fortran to Python interface generator (F2PY) som valdes efter en genomgång av olika verktyg. Under utvecklingen av finita element programmet ses att F2PY skapar ett gränssnitt i ett steg, integreringen av språken görs utan att Fortrans beräkningsegenskaper försämras, alla Fortran funktioner stöds av F2PY och en modul som är skapad av F2PY är lika enkel att använda som en inbyggd funktion i Python. Nackdelen med F2PY är att det krävs fem andra program installerade. Som helhet fungerar F2PY bra för att skapa en länk mellan Python och Fortran. Python uppfyller också alla krav som ställs på ett scriptspråk, enkel struktur, snabb utveckling av applikationer och möjligheten att skapa grafiska användargränssnitt med eller utan en grafisk gränssnitts byggare. Fortran och Python är plattformsoberoende vilket F2PY också är. Detta resulterar i att det rekommenderas att byta ut Borland Delphi mot Python i kursen Programutveckling för Tekniska System.

Contents

Preface i

Abstract iii

Sammanfattning v

1 Introduction 1
1.1 Background 1
1.2 Objective 1
1.3 Limitations 2
1.4 Outline 2

2 Mixed Language Programming 3
2.1 Static linking 4
2.2 Dynamic linking 4
2.3 Tools for generating interfaces to script languages 5

3 Scripting languages and computational code 8
3.1 Python 9
3.1.1 User Interfaces for Python 10
3.1.2 Computational environment 12
3.2 Fortran 13
3.3 Tools used in this work 13
3.4 Fortran to Python Interface Generator (F2PY) 13
3.4.1 Creating a Python module 14
3.4.2 Using F2PY generated modules in Python 17

4 Application cases 21
4.1 2D Truss 21
4.1.1 Quick review of former course layout 23
4.1.2 Interfacing Python with Fortran 24
4.1.3 User interface implementation 30
4.1.4 Final outline for 2D Truss 35
4.1.5 Python help and hints 36

4.1.6 Python to executable file 40
4.2 Graphical user interface for F2PY 40

5 Summary and Conclusions **43**
5.1 Summary 43
5.2 Conclusions 43
5.3 Future Work 44

Bibliography **45**

Appendices

A Python Modules **A-1**
A.1 Start A-1
A.2 Mainframe A-1
A.3 Base A-8
A.4 Modeldraw A-14
A.5 Elementproperties A-21
A.6 Elementproperties ui A-24

B Fortran Modules **B-1**
B.1 Calcprog B-1

Chapter 1

Introduction

1.1 Background

The course *Software Development for Technical Systems* at the division of Structural Mechanics [21] teaches students techniques for implementing finite element applications with both user interface and computational code. Currently Borland Delphi environment is used for the user interface and Fortran is used for the computational part. This master's thesis is an effort to study methods of integrating Fortran with script languages and user interface to create more flexible and easier to use software. The user interface has become more important and a natural part of computational programs because users often demand better graphical tools for creating complex models and viewing results. An additional advantage with better graphical interfaces is that the application reaches a larger user base. An interesting candidate for user interface and computational environment is Python. Python is a script language, which is easy to use and learn for non-programmers and yet powerful enough to use as a scripting environment for computational codes. Another advantage is the cost, Delphi is an expensive commercial application, Python is free and open source. Most students in this course are not experienced programmers, so the language must be easy to learn and understand, so that they can concentrate on the task instead of learning the intricate details of a complicated language.

1.2 Objective

This master thesis will investigate how forthcoming courses can be implemented with focus on Fortran integration in a script environment such as Python. The thesis will also investigate how the Python script-language can be used to develop user interfaces, in the same way as the Borland Delphi development environment has been used in former courses. A test application similar to the assignment in the existing course Software Development

for Technical Systems will be developed. The objective with the application is to evaluate if it is possible to use it in the new version of the course, it will also be considered if the application is user-friendly enough and platform independent.

1.3 Limitations

In this work the application case is developed in Windows and the graphical parts are tested both in Windows and Linux, but as no usable free Fortran 95 compiler is available for Linux at the time the F2PY created Python module is implemented on Windows. The test application does not have full support for all functions for a bar element. Fortran is not examined at the same levels as Python and F2PY since only minor changes are made in Fortran from the existing course.

1.4 Outline

Chapter 2: Mixed Language Programming Introduction how to mix different programming languages, both in Linux and Windows. Different tools that creates interfaces are introduced.

Chapter 3: Scripting languages and computational code Distinguishing features for scripting languages are described. The programming languages and interface tools that will be used are described with focus on interface creation and graphical user interfaces.

Chapter 4: Application Cases In this chapter two application cases are studied. The first is an application similar to the assignment in Software Development for Technical Systems, which investigates how to implement a finite element application. The second application test if it is possible to use Python as a rapid application develop environment as well as a tool for integrating different computational software components.

Chapter 2

Mixed Language Programming

Integrating computational code with user interface code often has special requirements. These two parts often have different demands on the programming language. The graphical part must support rapid development and integrate well with other languages. The language implementing the computational part must be fast in execution and able to handle large amounts of data e.g. matrices. To combine these aspects it is often needed to mix languages to take the advantage of the features different languages can offer.

According to Arnholm [8] a number of difficulties has to be considered before using a mixed language support. Hence it is important not to add new difficulties that are larger than the advantages. The main things to consider are that no new constraints should be added and that the complexity should not increase compared to using one language. If new constraints are added the lifetime of a project reduces and more complexity makes development inefficient and increases maintenance costs.

The described difficulties above can be translated to the following requirements [8]:

- Source code must be possible to use on any platform.
- The procedure to make a link between languages must be easy to perform.
- Mixed languages shall not loose execution speed although a linking is made.
- All major features must be supported in the linking for the linked language so that the original code does not have to be changed. If the original code has to be changed then old tested and verified code will not be usable in a convenient way.

- It shall not be much harder to call the linked code than calling code within a language.

Interfacing languages can be difficult since calling conventions often differs between compilers. There are two types of linking, static linking and dynamic linking. To understand what happens when linking different programming languages and to see the difficulties involved a short description of static and dynamic linking will follow. Differences between linking in Windows and Linux/Unix are also mentioned.

2.1 Static linking

Static linking [22, 9] is the simplest form of linking and the most easy to perform. A static link is made when a number of functions and modules are made into one single executable file when the module is compiled. The execution is usually faster than for dynamic linking but is more inconvenient when many programs uses the same modules, for example the Windows API. If all programs should include all modules they use into one executable file, the program would become huge and it would be difficult to update a function that many program uses. Linux and Microsoft Windows use the same method when linking statically.

2.2 Dynamic linking

In dynamic linking [22, 9] the modules are not included at compile time, instead at execution time. In Windows dynamic linking is accomplished using dynamic link libraries (DLL). In Linux/Unix it is implemented using Shared Objects (SO). The DLL and SO objects are files with pre-compiled code that can be used by any application. The dynamic objects are only loaded when they are needed and are never loaded in multiple instances, though multiple applications can use the same copy of the dynamic objects.

Dynamic linking have two load types, run-time and load-time. When loading Windows and Linux API functions load-time is used. Load-time is when a module is loaded into the memory when the program is loaded, just before execution. When run-time loading is used, the DLL is loaded dynamically during execution. The dynamic files have two types of linking, implicit and explicit linking. In implicit linking the operating system handles the linking, while in explicit linking the application handles the linking and the operating system is not involved. Files that uses explicit linking can be of any sort (*.*) but files that use implicit linking must have the file extension DLL for Windows or SO for Linux. The most common is implicit load-time linking.

The dynamic linking process is divided into two parts. First, the module that will be linked has to be located. Second, the address of the function in the module must be found. When the two parts are successful the external module can be linked just as if was linked static. Windows and Linux/Unix have differences in these two steps. In Windows the DLL file can have a complementary library file (LIB file) that contains the functions in the DLL file and the location of the corresponding functions. An application examines the headers to find the desired module it will link dynamic. In Linux/Unix it is the SO file it self that contains the corresponding information. Two more difference that exists between Windows and Linux is that Linux can link variables and uses position independent code but Windows does not. Position independent code means that the loaded dynamic library can change place in the memory during runtime. This leads to a more complex architecture but is more flexible [22, 9].

The advantages of using dynamic loading are that modules can be updated easily by just replacing a file. All processes using this module are updated automatically. The size in bytes is reduced when several applications can use the same modules. The memory usage is also reduced because multiple applications can use the same copy of the dynamic objects. Dynamic linking also have some disadvantages, when linking does not work it can be hard to find the errors, because the architecture of a program with dynamic linking is complex [22].

2.3 Tools for generating interfaces to script languages

Creating interfaces for script languages is often a difficult task to do completely by hand. To make it easier special tools can be used to create interfaces between different languages. The most common tools are:

- Simplified Wrapper and Interface Generator (SWIG)
- Fortran to Python Interface Generator (F2PY)
- Common Object Request Broker Architecture (CORBA)
- Component Object Model (COM/DCOM)

Previously, in the course Software Development for Technical Systems a special tool, Fortran interface wizard (FIW), was used to create the interfaces between Fortran and Borland Delphi. This tool is also reviewed.

Fortran Interface Wizard (FIW)

FIW [3] is a tool to make it easier to bind Fortran DLL files to Delphi. In FIW, interfaced subroutines are declared with variables. From this FIW cre-

ates a Delphi import unit and skeleton code in Fortran, which is needed for creating a Fortran DLL. The created Delphi import unit is ready to use but as the name implies the skeleton file(s) needs additional code. The skeleton code contains declaration of variables that are used within the linking to Delphi and implementation to accomplish the linking. The skeleton only needs to be complemented with the code that perform what the subroutine shall do and declaration of local variables. The subroutine(s) are then compiled to a Fortran DLL file that the Delphi import unit links to.

The DLL file is dynamically linked when the Delphi project is loaded (explicit load-time is used). This interface also requires a compiler that can build DLL files of Fortran source files. FIW is a Windows-only-solution since Delphi is only available for this platform.

Simplified Wrapper and Interface Generator (SWIG)

SWIG [20] is a multitool that can generate interfaces to script languages such as Python from C/C++. SWIG can use a header file in C/C++ to create wrapper code that interfaces the script language with the underlying C/C++ code. To use SWIG, a special interface file has to be written by hand, which is used to define what goes in the wrapper. SWIG is distributed for Windows, Linux and UNIX. SWIG main users are C/C++ developers wanting to provide an interface to a script language. SWIG uses runtime loading. If SWIG is to be used to link Python and Fortran an additional interface generator is necessary for interfacing C/C++ with Fortran.

Fortran to Python Interface Generator (F2PY)

F2PY [13] is an extension module for Python that generates a direct interface between Fortran and Python. This means that a Fortran module processed by F2PY can create a ready-to-use Python module in a single step. A Python module is a dynamically loadable file that contains the binary compiled Fortran code. All Python applications can use the module that F2PY creates. Disadvantage with F2PY is that it needs five prerequisites. F2PY uses run-time loading and explicit linking when loading the Python module. Inside the module implicit load-time is used e.g. when the Fortran DLLs are loaded. The code that binds Python with the Fortran code is generated by a signature file created by F2PY. F2PY is platform independent and can be run under Window, Linux and Unix.

Common Object Request Broker Architecture (CORBA)

CORBA [1, 2, 12] is a middlelayer process in a server/client architecture, which makes objects and applications interact and interoperate. Applications and objects communicate through an Object Request Broker (ORB) that has to be present to run a CORBA process. ORB:s exist for almost all

platforms that makes CORBA platform independent. Interfaces in CORBA are defined using an interface definition language (IDL). The purpose of the IDL is to define the object interfaces in a way that is independent of any programming language, which makes it possible for applications implemented in many different languages to interact. Client and server source codes are generated with an IDL compiler that is specific for each programming language. The source code is compiled to an executeable that links to other CORBA applications. CORBA can be used both locally and over networks. If CORBA is used locally, the server and client parts can be made into one binary file, which is very similar to SWIG. For more details see Henning and Vinoski [1].

The disadvantages with CORBA is that there is no straight connection to Fortran and that an IDL file always has to be created. CORBA is made for object oriented programming and is flexible and dynamic but this also make it complex. F2PY and SWIG compile code for the linked language, which means that only a specified language can use the created object. CORBA makes the object accessible for all languages.

Component Object Model (COM/DCOM)

COM/DCOM [2] is a multitool very similar to CORBA and is built in to Windows. A difference is that DCOM/COM uses a modified IDL. DCOM is used for remote connection and COM for local linking. DCOM/COM is written to work in most operating systems but in practice it almost only works in Windows.

Chapter 3

Scripting languages and computational code

Scripting languages are often used as a tool for gluing components together, which often results in reduced development time. System languages as C/C++ are often compared to scripting languages. System languages are designed to build data structures and algorithms with focus on fast execution in well defined and slowly changing applications, while script languages are designed to glue, handle strings, being easy and fast in development. Script- and system languages complement each other and do not replace each other. Using a script language on top of a low level language makes it more flexible. Changes in code are much faster when script languages has less developing time and is easier to understand. Script languages often have good possibilities to create graphical user interfaces, in a way that is easy both for the developer and the user of the finished product, Ousterhout [6].

Scripting language has given up execution speed and strength for getting better reuse of code and higher programmer productivity. Each statement in the source code of a script language contains more instructions for the cpu than a system language. But a scripting language is about 10-20 times slower in execution than a system language, Ousterhout [6]. Execution time for code is getting less important when computers become cheaper and faster in comparison to the cost of a programmer. The main problem for most applications today is not the computing time. Generally, the tasks a script language executes are not the tasks that need to be fast, e.g. it can be waiting for user inputs. Scripting languages are based on built in functions implemented and compiled in a system language, which makes built-in functions almost as fast as system language. When a script language is used the user shall try to use as many of the built-in modules and not to create custom modules that duplicates the built-in modules.

A distinguishing feature for scripting languages is that they are typeless to simplify connections between components. System languages are designed

for implementing complex components and have a strong degree of typing. In a typeless language variables do not have to be declared before they are used. They can be created on the fly and can change type during execution, and are sometimes called dynamic variables. This makes it possible to execute code in script languages at once. Compilation and execution is done in one step. In system languages the entire code first has to be compiled and then linked before execution.

When dynamic variables are used, the debugging procedure between system languages and scripting language differ. System languages check for errors during compilation, and that is impossible for scripting languages as they are typeless. Dynamic variables cannot be checked for errors at compilation because the variables type are not set at this stage. This problem is solved by debugging applications during execution. This result in an error check that is performed as late as possible, which often makes scripting languages more flexible. Scripting languages seem as they can allow errors to be undetected but is as safe as the debugging for system languages, Ousterhout [6].

Scripting languages had a breakthrough when the Internet and component frameworks started to expand, graphical user interfaces became popular and script languages started to be good to glue other components. Most programs today have a scripting languages built-in, enabling the user to change the program behavior with custom scripts. Updating programs and using plug-ins are also made easier.

3.1 Python

Python is a programming language created by Guido van Rossum in 1990 and named after the Monty Python's Flying Circus. Python is best described in programming terms as a modern object-oriented script language. Python has a good mix of engineering language and scripting language, full support for object-oriented programming and a powerful code structure. Python source code is platform independent and can be run on operating systems as Windows, Linux and Unix [4].

As described earlier, script languages can be used to glue components together, which is a distinguishing feature for Python too. This results in better code reuse and fewer rewrites. Python has a very close relation to C/C++ and Java, which makes this languages very easy to use with these languages. Python also has good integration capabilities to most other languages as well e.g. Fortran. Python's easy code structure makes it straightforward to translate Python to Java and C/C++. This illustrates the flexibility of Python as it can be used to glue components together as well as being embedded in other languages. To enhance speed most built-in modules are made in C/C++. Python can also be extended with modules written

in C, C++ or in Python. The modules can define new functions, classes, variables and objects.

Python has a large library including functions for Internet access, operating system (OS) integration and user interface. The code structure in Python is simple and that makes it easy to write and understand. Python can also be used as a command line interpreter or in an integrated development environment (IDE) to execute code at once. When the code is executed in Python, the text in the Python file (PY) is parsed and converted to byte-code stored in a PYC-file, which is executed by Python's virtual machine. This makes the second execution of a PY file faster since the PYC file can be executed immediately since no parsing of the Python file is needed, as illustrated in figure 3.1.

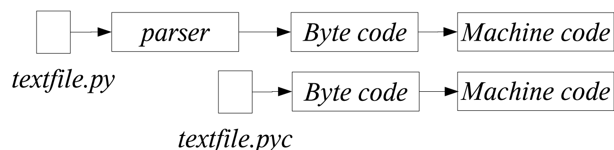


Figure 3.1: *Difference between PY and PYC files*

To facilitate the development of Python code an IDE can be used, an IDE often has parts that help the developer for writing, editing, testing, debugging and executing code. Komodo 3.1 from Activestate [7] is a professional development environment for dynamic languages, which supports the programmer with autocompletion, highlighting built-in commands and debugging code in the development, see figure 3.2. Komodo also has an interactive shell where single lines of code can be executed.

3.1.1 User Interfaces for Python

The user interface is the graphical part of the application that communicates with the user. Often this is used to define input data and view results. Python has a number of different graphical user interface (GUI) toolkits, such as Tkinter and wxPython [23]. More toolkits can be found on Python's GUI section [15]. Tkinter is built in to Python that makes it de facto standard GUI toolkit for Python. The same Tkinter source code can be used on all platforms supporting Python. wxPython is another good toolkit that could have been the main one if Tkinter was not first. wxPython is as simple and easy to use as Tkinter but contains more graphical objects. wxPython's two disadvantages are that it is not built-in to Python and that it can be hard to run on all platforms even though it said to be platform independent [16].

The graphical objects in a GUI toolkit are often called Widgets. The widgets that make Tkinter (Python) better than other toolkits are the text-

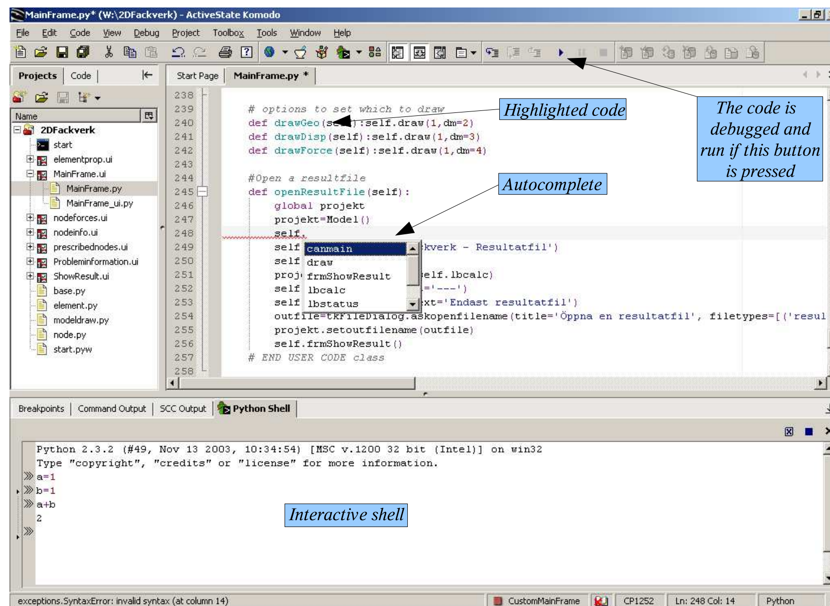


Figure 3.2: *Komodo IDE with explanations*

and the canvas widgets [4]. Using these widgets text, lines or pictures can be added and in an easy way deleted by setting tags (a group of objects) to different objects. Python also support many other functions for graphical interfaces. Most types of applications can be developed.

GUI builders

A large amount of source code is often needed to implement a user interface. To make this task easier a program can be used, which automatically generates source code for the graphical parts of a program e.g a button. This sort of program is called a GUI-builder. Often objects/widgets can be added to a window using drag and drop, see figure 3.3. The GUI builder can also change the attributes for the created widgets eliminating editing in the source code. The code for the graphical parts of a program that needs to be written manually is the event code. The event code can be code that describes what happens if a button is pressed or code for loading values into entries. Komodo 3.1 that is described earlier also includes a GUI builder, as shown in figure 3.3.

One disadvantage with the Komodo GUI-builder environment is that Python is not the best supported language. Some functions does not work at all, the documentation is also lacking in many areas. Komodo GUI builder generates two files. One read only file, which contains the graphical commands for the widgets and one, which can be edited that contains the event code and other settings that are added manually. To understand the gener-

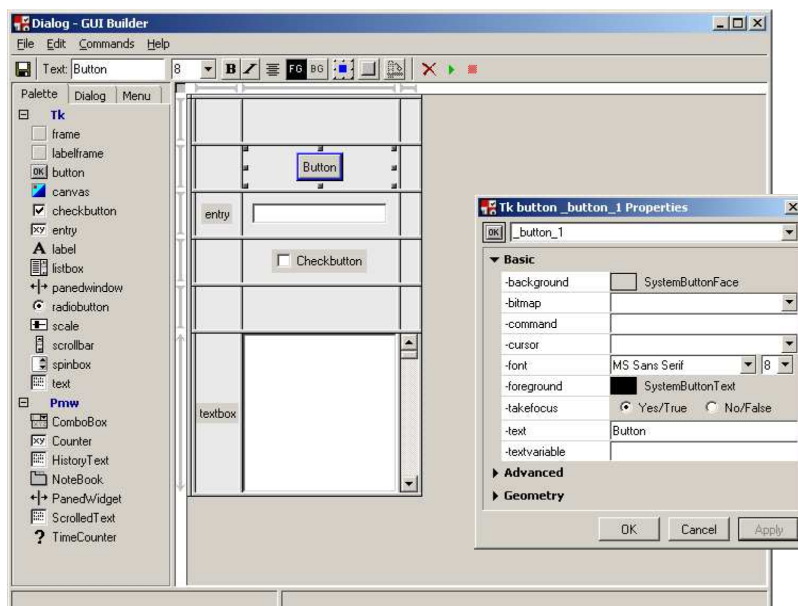


Figure 3.3: *Komodo GUI builder.*

ated code and to add own code, the user must have knowledge about object oriented programming including the self variable and inheriting as the two generated files are built up with classes. The preferences that can be change in the GUI builder will also easily confuse. All preferences that can be edited does not make any difference or does not work. Despite some weaknesses Komodo is an good GUI builder and reduces the time to implement a graphical user interface. Komodo is available for Linux, Microsoft Windows and Solaris [7].

3.1.2 Computational environment

Python has no built-in support for matrices and advanced mathematical computations. To handle this sort of functions in Python, additional modules have to be used. There are two main modules available, Numarray and Numeric [11]. Numeric is the predecessor to Numarray and they are very similar except that Numarray is faster for larger systems. They support both matrices and advanced computation, such as solving linear equation systems. Numeric and numarray creates a MATLAB [10] like environment in Python with similar syntax and functionality. The preferred module is Numarray, but Numeric is still frequently used. None of the modules are included in the Python distribution and they have to be installed separately.

3.2 Fortran

Fortran [8, 3] was developed in the 1950, since then new standards has been developed and the latest is Fortran 95, Fortran 2003 is under development. All new standards are backward compatible with the three latest standards. Since Fortran is standardized it is a platform independent language, so a program written in Fortran can be compiled on any platform having a Fortran compiler. Fortran is mostly used by engineers and researchers, as it is a fast system language that is often used in software requiring fast numerical calculations. A large amount of well established numerical code is written in Fortran. The main disadvantage with Fortran is that it can be complex and does not support object orientated programming. Fortran is about 15-20 times faster than Python. This is due to the fact that Fortran compiles the entire code to machine code executing directly on the target processor.

3.3 Tools used in this work

Python was chosen as the script language in this work. The main reasons for this are that it is an easy-to-learn language, integrates well with Fortran and supports GUI development. The graphical user interfaces are implemented with the Tkinter toolkit as it is built-in to Python. As interface generation tool F2PY was chosen after evaluating the different interface generation tools available. F2PY creates an interface to Fortran in a simple single step. The SWIG interface generator is mostly used to generate interfaces for C/C++. Generating Fortran interfaces would require extra C layer, which makes it less useful. Almost a new programming language has to be learned if CORBA or DCOM/COM is used. Numeric is used for matrices that are passed between Fortran and Python.

3.4 Fortran to Python Interface Generator (F2PY)

F2PY [13] is an extension module to Python that make it possible to link Fortran source code to Python. F2PY creates Python C/API modules from signature files. The signature files are made by hand or generated automatically with F2PY from Fortran source code. The signature files contains information about functions and variables in the Fortran source code. From the signature files the bindings between Python and Fortran functions can be built. F2PY is simple to use because binary Python modules can be built in a single step from Fortran source. The interface is generated without loosing any advantages in Fortran as the Python modules are binary.

The main features for F2PY are:

- Fortran 77/90/95 functions, F90/95 modules and C function are supported.

	Supported in F2PY	Passed as reference	Output
Scalar numbers	yes	no	yes
Numeric arrays	yes	yes*	yes
Numarray arrays	yes	no	no
Strings	yes	no	yes
Dictionaries	no	no	no
Lists	yes**	no	no
Tuples	yes**	no	no

Table 3.1: *Overview of supported variables in F2PY. * The variables are only passed as reference if the arrays are set to column major storage and declared as an array in Fortran. ** Lists and Tuples are converted to numeric variables in F2PY and are not recommended to use with F2PY.*

- Functions and data in Python can be accessed from modules created in F2PY with a callback function.
- Arguments in linked functions can be optional, required or hidden and dependencies between arguments are solved automatically.
- F2PY detects automatically if an array that is sent to a F2PY created module is of column major storage or row major storage.

3.4.1 Creating a Python module

To create a Python module some issues have to be considered. Even though F2PY detects what sort of storage an array has, this is the main problem when interfacing Fortran and Python. The problem is that matrices in Fortran and Python are stored differently in memory. Python uses row major storage (C contiguous) and Fortran use column major storage (F contiguous). The input to F2PY is solved automatically but returned arrays from Fortran are always column major storage. This introduces some complications in Python, as described later. An array of Numeric type stored in column major storage is passed by reference between Python and Fortran. If the variable is not stored in the correct format and is of the right variable type, the variable will be copied before function call. Passing scalars variables as references requires the scalars to be declared as an array with length one, the default is to pass scalars by value. Python variables also have to be declared with the same type as the corresponding variable in Fortran if they should be passed by reference. For example a double variable in Fortran needs to be linked with a Python variable declared as a double. Numarray arrays cannot be passed by reference to F2PY created modules. An overview of variables supported and how they are passed is shown in table 3.1.

Changing a Python array to column major storage can be done by transposing or using the `as_column_major_storage(<array>)` function, which

is built-in to all F2PY created modules. The latter function converts the matrix to column major storage without flipping the rows and columns as transpose does. With both these functions some important properties of the matrices are lost, therefore the returned matrices must be treated carefully in Python. E.g rows or columns that are not next to each other in a matrix cannot be change at the same time, partial indices can't be used. Both Numeric and Numarray has a function `copy`, which creates a exact copy of a F-contiguous array but it is C-contiguous.

Special F2PY attributes controlling the behavior of variables can be set in the signature file or directly in the Fortran source code. Adding these attributes does not have any disadvantages for the Fortran code. The most common attributes are:

- **In**, which is the default setting if no other attributes are set to a variable. This means that a variable is a required input in a subroutine or function.
- **Out**, which is set to a variable if it is to be returned to Python. If a variable is both an input and output variable **in, out** is used.
- **Copy**, if a Numeric variable that is Fortran contiguous is used it is linked as a reference. If this Numeric variable does not want to be passed as a reference, copy can be used.
- **C**, defines the array variable as row major storage. It is not recommended to use since a Numeric C-contiguous array will be passed as reference, which results that position [2,2] of the matrix in Python does not correspond to position [2,2] in Fortran. This is confusing and makes the variable less usable.

Attributes are added after the normal Fortran declaration of the variable by using the `intent` F2PY keyword, as shown in the following code.

```
!variable declaration
real(kind=8) :: eprop(6,*)
!adding attribute
!f2py   intent(in,out) eprop
```

F2PY keywords are declared in Fortran as comments, so that they do not interfere with the compiler. All variables that are given an attribute are declared in the subroutine's header. More detailed documentation can be found on F2PY's homepage [13].

There are as well some minor Fortran features not supported functions in F2PY. F2PY has problems compiling files with the extension .f95 but files with the extension .f, .f77 or f90 are supported. The accuracy of a variable cannot be set by a parameter, it has to be set with a value, e.g. kind=8.

Before starting to use F2PY some prerequisites have to be installed. They are listed below.

- Python
- Numpy (Numeric), Matrix handling
- Numarray (optional), Matrix handling
- SciPy, Compiler handling
- Fortran compiler
- C compiler

The C and Fortran compilers should preferably be from the same vendor. This is specially important for Windows since the object file formats differ between compilers. In Linux any Fortran and C compilers can be used in any combination. F2PY supports a number of major Fortran and C compilers as listed in F2PY's documentation [13]. When all the prerequisites are installed and configured properly F2PY is easy to use.

To compile the Fortran files to Python modules, F2PY is called with the command F2PY in the command prompt in Windows or Linux. To build a Python module in a single step the following command is used.

```
f2py -c -m <Name Python module> <Fortran files>
```

Additional options can be set to F2PY in the same command. The reader is referred to F2PY's documentation [13] for more information. The example below illustrates a session of using F2PY from the command line.

```
W:\f2pyGUI>f2py -c -m testmodule hello.f90
numpy_info:
FOUND:
  define_macros = [('NUMERIC_VERSION', '"\\"23.1\\""')]
  include_dirs = ['C:\\Python23\\include']

running build running config_fc running build_src building
extension "testmodule" sources
.
.
.
```

```
.
Removing build directory
  c:\docume~1\bmxjmr\locals~1\temp\tmpsb2max

W:\f2pyGUI>
```

The modules are also automatically documented with LaTeX in F2PY. To make it easier to use F2PY a GUI is developed in Python for F2PY. The application is described in section 4.2.

3.4.2 Using F2PY generated modules in Python

Using F2PY generated modules in Python is done in the same way as normal Python modules. To illustrate how to use F2PY two examples are given.

Example 1

Example 1 shows very basic commands for F2PY. A Fortran module `test` is located in a file `file.f90` and this file is compiled in F2PY and the Python module `module.pyd` in Windows or `module.so` in Linux is created. The Fortran code is presented first and then commands in a Python interpreter with results. The `#` or `!` signs is a comment.

Fortran code

```
! File module.f90

subroutine test (a)
    integer :: a

    print*, "Hello Python"
    print*,"a= ",a

end subroutine test

! End file module.f90
```

The Fortran code is compiled with F2PY by giving the following command in Windows command prompt.

```
f2py -c -m module file.f90
```

Python interpreter

To import the F2PY created module module.pyd the command import is used.

```
>>> import module
```

The help documentation is printed on the screen with the following command.

```
>>> module.test.__doc__

    test - Function signature:
        test(a)
    Required arguments:
        a : input int
```

The instructions in the documentation shows how to use the module, the module requires one input argument. The module is executed underneath.

```
>>> module.test(5)

    Hello Python
    a = 5
```

Example 2

This example illustrates how Numeric arrays are used with F2PY created modules. The example shows how a Numeric array is handled when interfaced as a reference and as a copy. It also describes how a return-variable is given an attribute.

Fortran code

```
!File test2.f90

subroutine test (a,b,c)

    real(kind=8) :: a(2,*), b, c(2,*)

!f2py   intent(out) b
!f2py   intent(in,out) a,c

    a(1,2)=a(1,2)+10
    b=a(1,2)
    c(1,2)=c(1,2)+10

end subroutine test

!End file test2.f90
```

Compiling Fortran code to Python module:

```
f2py -c -m test2 test2.f90
```

Python interpreter

Importing Numeric module for matrix support.

```
>>> from Numeric import *
```

Importing F2PY generated module.

```
>>> import test2
```

Print the documentation, which shows that there are two required arguments (a,c) and three returned objects (a,b,c).

```
>>> print test2.test.__doc__

    test - Function signature:
        a,b,c = test(a,c)
    Required arguments:
        a : input rank-2 array('d') with bounds
             (2,*)
        c : input rank-2 array('d') with bounds
             (2,*)
    Return objects:
        a : rank-2 array('d') with bounds (2,*)
        b : float
        c : rank-2 array('d') with bounds (2,*)
```

Creates variable "a" that shall be a matrix with two rows and the number of columns is defined by the user. The 'd' means that it is a variable of type double. The variable is transformed to be column major storage.

```
>>> a=test2.as_column_major_storage(array(((1,2)
    ,(2,3)),'d'))
```

Creates the array 'c' that looks exactly as the variable 'a' except that it is row major storage array.

```
>>> c=array(((1,2),(2,3)),'d')
```

The function test is executed with the input variables a and c, the returned variables are stored in A,B and C.

```
>>> A,B,C=test2.test(a,c)
```

Variable A is printed, this is the original "a" value, which is returned and changed.

```
>>> print 'A= ',A

A=
[[  1.  12.]
 [  2.   3.]]
```

Variable B is printed, this variable shows how to set an attribute to a variable, which is only output.

```
>>> print 'B= ',B

B=  12.0
```

Variable C is printed, This is the original "c" value that is returned and changed.

```
>>> print 'C= ',C

C=
[[  1.  12.]
 [  2.   3.]]
```

Variable a is printed, it is passed as a reference since it is column major storage and that is the reason why it is change from the original value.

```
>>> print 'a= ',a

a=
[[  1.  12.]
 [  2.   3.]]
```

Variable c is printed, it is passed as a copy since it is not column major storage, this result in that it is not changed from the original value.

```
>>> print 'c= ',c

c=
[[ 1.  2.]
 [ 2.  3.]]
```

Chapter 4

Application cases

Python, Fortran and F2PY are evaluated in combination to so see if they fulfill the demands computational applications has both from the user and in a developers perspective. Test application cases are implemented to evaluate this combination of programming languages, where both difficulties and possibilities for Python and F2PY are mentioned. Fortran is not covered extensively as there is nearly no change in Fortran from the existing course layout of Software Development for Technical Systems. The creation and structuring of the test applications are also described.

Two applications cases are studied. The larger application case is a two dimensional finite bar element program (2D Truss). The smaller application is a graphical user interface for F2PY. The latter application illustrates how Python can be used to implement an user interface to an application rapidly and be used as a good gluing language.

All Python code is generated with help from the books by Mark Lutz & David Ascher [4, 5] and the Python.org web site [14] if nothing else is written.

4.1 2D Truss

2D Truss is a two dimensional finite element truss program where all elements consist of bar elements, see figure 4.1. A finite element solver implemented in Fortran is used for the computations, the graphical user interface (GUI) is implemented in Python and F2PY is used to implement the interface between Fortran and Python.

In the existing course of Software Development for Technical Systems there is an assignment that is very similar to 2D Truss. This assignment is used as a skeleton for 2D Truss and can be found on Structural Mechanics homepage [21]. The writer who has taken the existing course has there developed the application in the assignment, the computational part will be partly reused in 2D Truss. A rough layout for a finite element computational

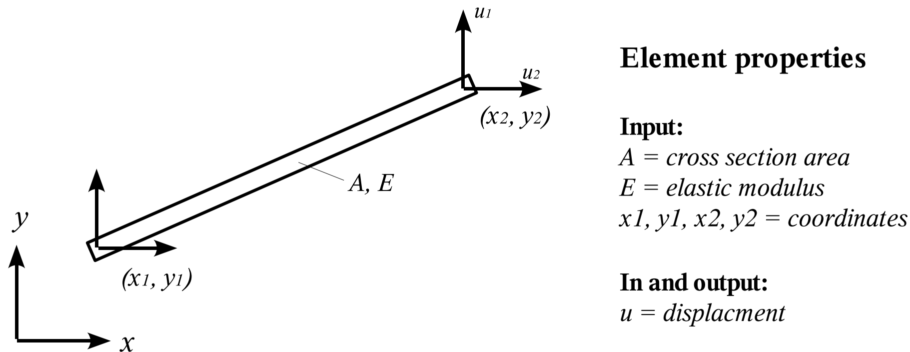


Figure 4.1: *Bar element with properties*

program and the requirements for 2D Truss are described in the assignment. In the list below the rough layout of a computational application is show. After a introduction to 2D Truss a quick review will be given for the existing layout of the assignment.

- Give input data in text form and/or graphically.
- Perform calculations.
- Presentation of result in text form and graphically.

2D Truss have goals for each item in the list above and the goals are taken from the existing assignment and are described underneath. Figure 4.2 shows an example how the application will look like when it is finished.

Input data: The input data is set in the graphical part and gives the user the possibility to create elements, define element topology, set nodal forces and set prescribed displacement of nodes. The element properties are presented in figure 4.1, which shows a bar element for a plane truss. Before calculation the geometry can be drawn to verify the input.

Calculation: Calculations are made with a banded finite element solver. The calculations also include assembling to stiffness matrix and force vector.

Result: The results are presented both in text form and graphically. The results in text form are presented in a result file and contains reaction forces in nodes, displacement of nodes and normal forces in elements. The result file can be viewed in 2D Truss. Normal forces in elements and displacement of nodes are also presented graphically in combination with the geometry of the model.

Other requirements: 2D Truss also follows the requirements to be able to save, save as, open and create new models.

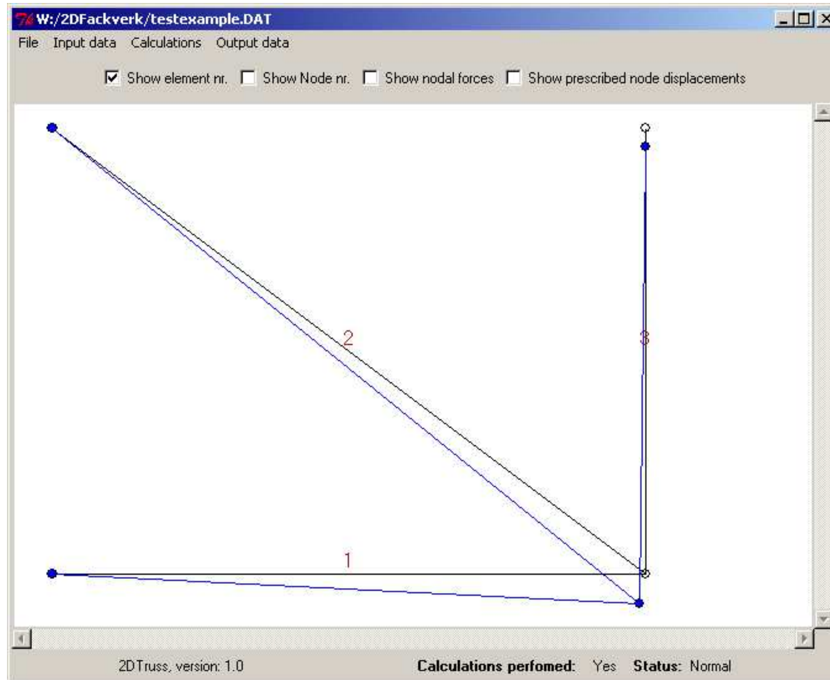


Figure 4.2: *Main window in 2D Truss with displacement and element number drawn for a test model.*

4.1.1 Quick review of former course layout

In former courses the goal was to create a program with the same structure and requirements as the program described above. The layout was constructed so that Fortran was used for computational code and Borland Delphi is used for the user interface. The interface between Fortran and Delphi is made with help from Fortran Interface Wizard (FIW) and a Fortran Compiler.

First, the computational part in Fortran is developed. It consists of functions to create element matrices, a boundary vector, assemble element matrices to a global stiffness matrix and to create a force vector. These vectors and matrices generate a system of linear equations, which is solved with a finite element solver that can handle boundary conditions. The results from the solver are displacements of elements, from which the normal force in each element can be calculated. To be able to test and verity the calculations, two functions are created, one to read input files and one to write a result file.

The Fortran Interface Wizard is used to create an interface between Fortran and Delphi. FIW is described in section 2.3. Three main Fortran subroutines are interfaced: read size of the problem, read input data and execute the solver. Delphi has to read the size of the problem before all input data can be read because the matrices in Delphi have to be allocated before the cells can be filled. The three main subroutines are not run in a single step as the user shall have the ability to make changes in the input data before a calculation is performed. The read and write routines could have been implemented in Delphi but in order not to rewrite any code, Fortran continued to implement these functions.

To build the graphical user interface, Borland Delphi's GUI builder is used. When the finished application built with the GUI builder and started, a main window appears. This window shows the menu and graphics of a created model. The input data and results of a model are stored in a matrix and vector controlled by a unit `Model`. The input data is edited in sub windows that are reached from the menu. If the size of a model is changed, a function that resizes the matrix and vectors without loosing the stored values is called and all new cells are set to zero. If the changes needs to be saved a save command is found in the menu. The save command creates a input file that follows the input data manual that Fortran uses when reading input data. To perform a calculation, calculate is chosen in the menu. Then the input data in the matrices and vectors are passed straight to the Fortran DLL. The result from the calculations is returned both in arrays for the graphical presentation and in a text file to show the result in numbers. A unit called `ModelDraw` controls the graphical presentation in the canvas of the main window. All sub windows have one unit each that handles event code and build up the windows.

This assignment shows that Delphi has a simple structure that is easy to follow. In the project, no object oriented programming was used, which makes the application less flexible and complex if larger applications are built. Delphi GUI builder is an advanced builder with many built-in preferences and all objects can easily be linked to an event. The application made with Delphi looks almost the same as the application in figure 4.2.

4.1.2 Interfacing Python with Fortran

The prerequisite and the review from the former course assignment makes it possible to start to create a more detailed outline for 2D Truss. The outline for the computational part is made first since most of the Fortran source code will be reused from the Fortran-Delphi project described above. The graphical interface is not dependent on the structure of the computational part but the GUI is dependent on the computational part.

The computational part of Fortran (FE-solver) defines the output and input variables (arguments) for the subroutine that is called from Python.

To continue further it has to be decided how the input data (input to FE-solver) for a model is stored and treated in the Python part of 2D Truss. The FE-solver requirements are matrices, vectors and scalars. Either they are stored in Python exactly as the FE-solver requires them or the input data is stored in any form and the required variables are created just before calculation. The first variant that is used in the existing course is not possible to use conveniently, as Python does not have a variable type, which can change size in combination with saving the old values in the same cells and set zeros to new cells. Because Python has good support for object orientated programming it can handle the input data in instances of classes. From the instances matrices, vectors and scalars are create just before computation. A consequence with the latter method is that Python will read the input data files. Python has a good function `cpickle` to save instances of a class to file. The latter method is used in 2D Truss. The routines in Fortran for computation and writing the result to a result file are reused. FE-solver returns displacement and normal forces to Python for graphical presentation. Input data to 2D Truss are described with three classes: model, element and node.

Classes

The classes are implemented so an instance of the model object is created for each opened project in 2D Truss. The model class have methods for adding and removing instances of element and node objects (aggregates), see figure 4.3. An example of a class implementation is shown in a listing after the three classes are described.

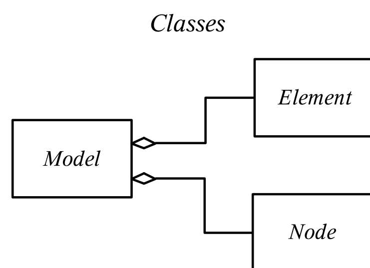


Figure 4.3: *Structure of classes in 2D Truss*

Model An instance of the `model` class handles most of the properties of a structure/model such as the names for the input data files and result files. Lists with element and nodes are stored and the instance also contains the number of elements, nodes, nodal forces and prescribed node values. After calculations the results that are used in the graphical presentation are stored in arrays. Functions to set and retrieve variables are also implemented.

Node An instance of the node class contains the variables, number of degrees of freedom, x and y coordinates of the node, nodal forces and prescribed node values. To control the variables, functions are implemented to set values and get values of the variables.

When an instance of the node class is created, the number of degrees of freedoms a node has can be defined. The default value is set to two degrees of freedom and is not changed in this application, as other functions does not support other than two degrees of freedom.

Element The element class implements the properties of an element, which are area, E-modulus, and nodes. The nodes contains the x and y coordinates so the elements define their size and position in terms of nodes.

```
#   Class that handles bar element with two
    dimensions

class element:

    #   When an instance is created

    def __init__(self,n1,n2):
        self.nodes=[n1,n2]      #Nodes
        self.E=0.               #E-moduls
        self.A=0.               #Area

    #   Returns the area and the e-modulus

    def getprop(self):
        return [self.E, self.A]

    #   Returns the nodes

    def getnodes(self):
        return self.nodes

    #   Set properties

    def setprop(self,Emodule,Area):
        self.E=Emodule
        self.A=Area

    #   Set nodes of the element

    def setnodes(self,n1,n2):
        self.nodes=[n1,n2]
```

Implementing the Fortran Python module

The decision how a project handles input data makes it possible to change the Fortran source code so it can be reused. After the Fortran source code is modified to suit F2PY it can be run in F2PY and tested.

The Fortran solver is built with reused parts of the Fortran code from the former course. A main module `calcprog` is created that includes a subroutine `execute`, which is the subroutine that interfaces to Python. `Execute` calls the other Fortran sub modules that implements the components part of the application, illustrated in figure 4.4. As the subroutines are tested and verified in the former assignment of Software Development for Technical Systems this results in that only the new main module has to be tested and verified. It is easier to manage, update and find errors when the Fortran DLL is structured as in figure 4.4. The Submodule `barelement` creates element matrices and force vectors, which are assembled to a global stiffness matrix and force vector. `Barelement` also calculates the bandwidth of the global stiffness matrix. `Solve` solves the linear equation system and `barelement` calculates the normal forces in the elements from the displacement that are given from the solver. As a last operation `outputfile` creates the result file. Main module `calcprog` is shown in appendix B.1.

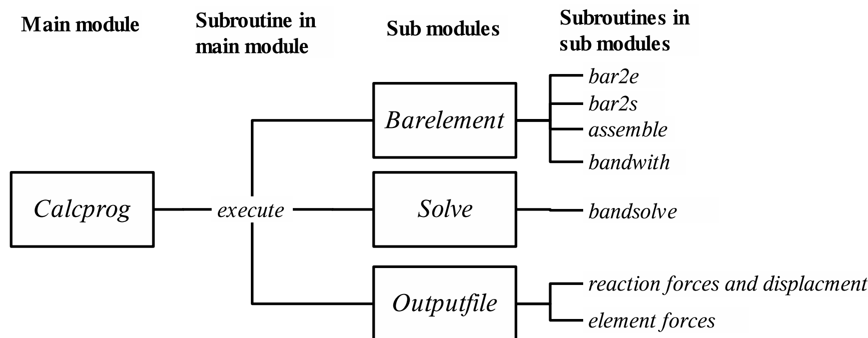


Figure 4.4: *Structure of Fortran DLL `solvepro.pyd`*

The interfaced variables from Python to `calcprog` does not need any F2PY attributes, since the arrays in the interface are Numeric variables and converted to column major storage so they will be passed as references. Displacement and normal forces arrays are returned from the Fortran DLL are created in Python before the calculation and then passed as reference, which make them change to the result automatically. The returned arrays in Python do not impose any difficulties as they are Fortran contiguous and can be treated as vectors and not matrices since they are rank-1 arrays. Rank-1 arrays of column major storages do not affect the properties of the arrays. The only code that is changed in the Fortran code before the compilation in F2PY is the accuracy of a variable. The interface is shown below where

the header to FE-solver is shown and the entire module in appendix B.1.

Header of solver in Fortran

```
subroutine execute(nel,neq,nnl,npv,eprop,edof,f,bc,
    b,eforces,outfile,ierr,displace)

    !---- Declaration of exported variables.

    implicit none

    integer :: nel
    integer :: neq
    integer :: nnl
    integer :: npv
    real(kind=8) :: eprop(6,*)
    integer :: edof(4,*)
    real(kind=8) :: f(*)
    real(kind=8) :: bc(*)
    integer :: b(*)
    real(kind=8) :: eforces(*)
    character(255) :: outfile
    integer :: ierr(1)
    real(kind=8) :: displace(*)
```

To call FE-solver in Python the following procedure is used, the complete module can be found in appendix A.3. The name of the PYD file is `solvepro` and then the subroutine `execute` is in a module `calcprog`. Before the FE-solver is called from Python the necessary variables are constructed with a function `createArrays` from the input data in the instances of `model`, `node` and `element`.

Python call procedure

```
#  creates the arrays which are linked.

[nel,neq,nnl,npv,eprop,edof,f,bc,b,ierr]=
    createArrays(projekt)

#  sets the returned vectors to zero and get them
    from the class object

projekt.setDispAndEforceToZeroAndSetSize()
disp,eforces=projekt.getDispAndEforces()

#  Calls execute in the module calcprog in the
    file solvepro.pyd to perform the calculations

solvepro.calcprog.execute(nel,neq,nnl,npv,eprop,
    edof,f,bc,b,eforces,outfile,ierr,disp)
```

```
#    ierr is a errorflag if the probelm is
     nonpositive definit
#    outfile is the name of the resultfile
#    eforces and disp are returned arrays and the
     rest are input variables
```

`createArrays` is shown in the listing below. Transpose is used to convert the matrices to column major storage and is done in order not to depend on the F2PY command `as_column_major_storage(<array>)`, which is only available if a F2PY module is imported. When doing this, the matrices have to be constructed transposed to the matrices that will be linked to Fortran.

CreateArrays

```
def createArrays(projekt):

    var=projekt.getprobleminfo()
    nel=var[0]
    neq=var[1]*2
    nnl=var[2]
    npv=var[3]

#    Creates the arrays with zeros

    ierr=array(0)
    eprop=zeros((nel,6),'d')
    edof=zeros((nel,4),'l')
    f=zeros(neq,'d')
    b=zeros(neq,'l')
    bc=zeros(neq,'d')

#    Fill the arrays with information about the
     problem after a patter from a indata manual
     for the fortransolver

    for i in xrange(nel):
        e=projekt.getelement(i)
        n1=projekt.getnode(e.getnodes()[0]-1)
        n2=projekt.getnode(e.getnodes()[1]-1)

        epropel=[e.getprop()[1],e.getprop()[0],n1.
            getcoord()[0],n1.getcoord()[1],n2.
            getcoord()[0],n2.getcoord()[1]]
        edofel=[n1.getdof()[0],n1.getdof()[1],n2.
            getdof()[0],n2.getdof()[1]]

        eprop[i]=epropel
```

```
        edof[i]=edofel

    for i in xrange(var[1]):
        n=projekt.getnode(i)
        f[i*2]=n.getnodeforces()[0]
        f[i*2+1]=n.getnodeforces()[1]
        b[i*2]=n.getb()[0]
        b[i*2+1]=n.getb()[1]
        bc[i*2]=n.getbc()[0]
        bc[i*2+1]=n.getbc()[1]

#   End filling the arrays with indata

#   Transpose the arrays to store them
    columnwise like fortran needs and not
    rowwise like python makes by default

    eprop=transpose(eprop)
    edof=transpose(edof)

    return nel,neq,nnl,npv,eprop,edof,f,bc,b,ierr
```

Because the input data is layout in the correct structure for the Fortran module, it can now be tested. If F2PY runs correctly it is now possible to compile the Fortran source code to a Python module in one single step.

4.1.3 User interface implementation

The main frame in the graphic part is used as a starting point for 2D Truss user interface and is implemented first. All windows and modules are implemented with approximately the same structure as in the assignment in the former course. The structure for 2D Truss is shown in figure 4.5. The user interface modules are built with Komodo 3.0 GUI builder.

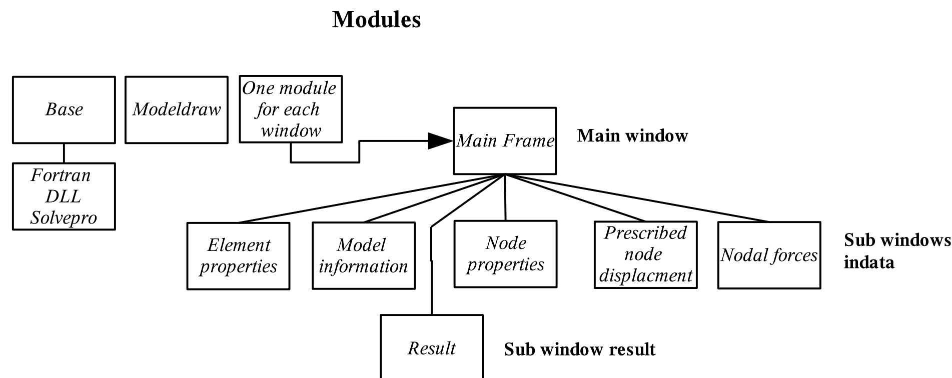


Figure 4.5: *Structure of GUI for 2D Truss*

Windows and Modules

In the main frame the drawing area and the menu are displayed, they are shown in figure 4.2. The menu is built with four main drop down menus: file, input data, calculations and output data. The sub items to each main menu can be seen in figure 4.6. All sub items of the file menu are handled by the module `base.`

File
New
Open
Save
Save as ...
Quit

Input data
Model information
Element properties
Node properties
Nodal forces
Prescribde node displacment

Calculations
Calculate

Output data
Resultfile
Graphical presentation ►
Open resultfile

Element geometry
Displacments
Normal forces

Figure 4.6: *Menu in 2D Truss*

Module base and file menu Module `base` contains the `model` class, `createArrays` function, `execute` function(perform a calculation) and the functions in the file menu. The class `model`, `createArrays` and `execute` are described earlier. The `new` option in the menu creates a new instance of the `model` object. `Save` calls a module `cpickle` in Python, which is a module that can save an instance of a class to a file [14]. In 2D Truss the instance of the class `model` is saved and since it contains all the instances of element and node they are saved as well. `Open` also uses `cpickle` to open the saved instance again. `Save as ...` and `Open` open a built-in module `tkFileDialog` that provides interfaces to native file dialogues, Secret Labs internet site [19] has good documentation.

Input data menu In this menu the properties for the model can be changed or set. All options are opened in a new window. Sub windows that handles input data use the same flow of event when the input data is set or changed, see figure 4.7. The figure also shows the window where the element properties are shown and can be changed. The code for the element properties window is presented in appendices A.5 & A.6. All input sub windows are describe below.

- **Model information** This window shows number of elements, number of nodes, number of nodal forces and number prescribed node values in the project. Nothing is set in this window, the purpose is to verify data.
- **Element properties** From this window the area, E-modulus, start and end node can be set for each element in the structure. Number of elements is also set here.

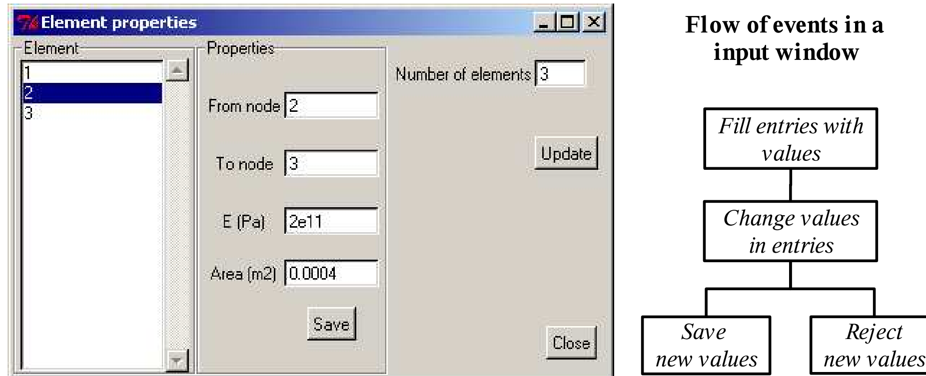


Figure 4.7: *The figure to the left shows a typical input data window in 2D Truss, The figure to the right describes how an input data window is implemented for changing input data.*

- **Node properties** From this window the properties for the nodes can be set or changed. The entries are number of nodes, x coordinate and y coordinate. The windows also shows which degrees of freedom that belongs to the node and are set automatically.

- **Nodal Forces** This is the window for setting nodal forces for each degree of freedom, one horizontal and one vertical degree of freedom for each node. The default value is zero.

- **Prescribed node values** The same as for nodal forces but here the displacement for each degree of freedom is set. A checkbox is ticked if the node is prescribed or not.

Calculations menu Once a complete model has been built or opened it can be solved by choosing Calculate in the Calculations menu.

Output data menu This is the last menu option. In this menu the result file can be shown in a new window or a graphical presentation of the model can be viewed in the drawing area, where deformations and normal forces can be shown graphically in combination with the model. The drawings are handled in a module `modeldraw`.

- **Result file window** The result file is shown in a text canvas in a new window.

Module Modeldraw The module `modeldraw` handles the drawing of the geometry and results in the drawing area, implemented using a `Canvas` widget. The `Canvas` is one of the strong parts in Tkinter, see figure 4.2. With

only a few lines of instructions, a complex geometry can be drawn, including lines with thickness and arrowheads, Secret Labs homepage [19] provides good documentation. The advanced drawing functions of the `Canvas` widget is illustrated in the code below, in appendix A.4 the complete module `modeldraw` is shown.

```
#   If tensile stress in a bar element

if force[i]>0:
    canvasMain.create_line(xd1,yd1,xd2,yd2, arrow='
        both', tag='arrow', fill='blue', arrowshape
        =(8,10,3))

#   If compressive stress in a bar element

else:
    canvasMain.create_line(xd1,yd1,xd2,yd2, arrow='
        both', tag='arrow', fill='red', arrowshape
        =(-8,-10,-5))
```

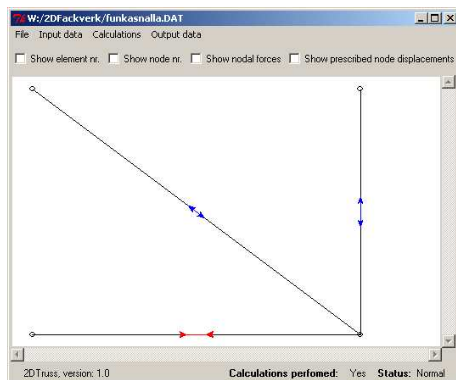


Figure 4.8: *Normal forces drawn in 2D Truss*

In the listing above two commands for drawing lines are shown. The option `arrow` is only written if arrows are wanted, `fill` means what color the line has. `Arrowshape` sets the shape of the arrow, a minus makes the arrow change direction, see figure 4.8

Before the geometry and results can be drawn a translation has to be done of the geometry as the global coordinate system for the FE-model is different from the canvas coordinate system. The structure also has to be scaled to fit the window. The canvas coordinate system is defined from the upper left corner with the x-axis going right and the y-axis going downwards and the FE system is defined as in figure 4.1. To translate the coordinate system two functions are implemented, `coordvalues` and `maxminXY`. `MaxminXY`

finds the size of the structure and the window to draw in so a ratio can be calculated, which is set to use as much of the window that is possible. `Coordvalues` finds a function with help from the ratio, which can translate and scale FE coordinates to the canvas coordinates.

The structure, displacement and normal forces are drawn in the canvas with three different functions. If displacement or normal forces are drawn the geometry function is called first to draw the geometry. When a drawing option is called in the menu, a `draw` function in the `mainframe` module is first called. This `draw` function then calls the functions in `modeldraw`.

Tkinter's canvas has a useful option that allows each object that is drawn to get a tag. In 2D Truss the drawn objects for element numbers, node numbers, nodal forces and prescribed node values have a tag each so that they can easily be deleted or drawn when the checkbuttons in the mainframe are ticked or unticked eliminating the need to redraw the whole figure. This option is available for all drawings in 2D Truss. The following code is extracted from the `DrawGeometry` function in the `modeldraw` module, which calls functions to draw if the checkbuttons are ticked.

```
#  Draws elementnumbers if the checkbutton is
   filled

if int(root.getvar('showElNr'))==1:printElementNr(
   canvasMain, projekt)

#  Draws nodenumbers if the checkbutton is filled

if int(root.getvar('showNodeNr'))==1:printNodeNr(
   canvasMain)

#  Draws nodal forces if the checkbutton is filled

if int(root.getvar('showNodeForce'))==1:
   printNodeForce(canvasMain, projekt)

#  Draws prescribde dof if the checkbutton is
   filled

if int(root.getvar('showPreNodes'))==1:printPreNode
   (canvasMain, projekt)
```

If the window that contains the drawing area is resized the drawing area has to be resized and the objects in the drawing area redrawn. In the following listing the method for doing this is shown. This example is taken from initialization of the main window where the drawing area is created. `Canmain` is the name of the instance of the `Canvas` widget (the drawing area) and `draw` is the drawing function in the program. This event happens

if something is changed to `Canmain`, not only if the drawing area is resized.

```
def __init__(self,root):
        .
        .
        self.canmain.bind('<Configure>',self.draw)
        .
        .
```

Another good characteristic of the canvas is that different objects can be moved forward and backwards on the screen.

4.1.4 Final outline for 2D Truss

The final structure of the application 2D Truss is presented in figure 4.9. All events in 2D Truss starts from the main frame and are initiated by the user. Figure 4.9 also illustrates the flow of events. The numbers represents the main tasks of the application.

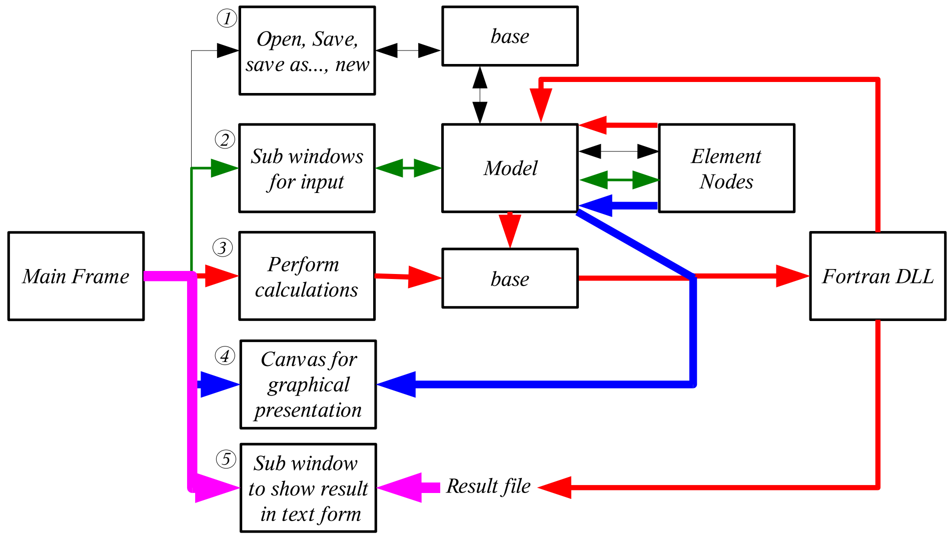


Figure 4.9: *How 2D Truss works*

1. When a model/project is saved, opened, saved as.. or a new is created, a function in module `base` is called, which retrieves or set the input data in the classes. Everything is set to zero when a new project is created.

2. When a input data window is opened input data is collected from the model instance, which in the other hand collect input data about elements and nodes from from the instances in the node and element

lists. If input data is changed by the user input data is sent in the other direction.

3. Calculations are performed by the Fortran computational module were a result file is created and displacements and normal forces of the structure are sent to the model instance. The input data is collected and transformed to arrays in base before the computations are performed.

4. When results are chosen to be shown graphically the canvas retrieves all input data for the structure from the model, element and node instances.

5. The result file is included in a text canvas, which is opened in a sub window when that menu option i chosen.

2D Truss has been tested and verified with a application example, see figures 4.10 and 4.11.

4.1.5 Python help and hints

Building the graphical user interface brings up a number of difficulties that are worth mentioning in this section.

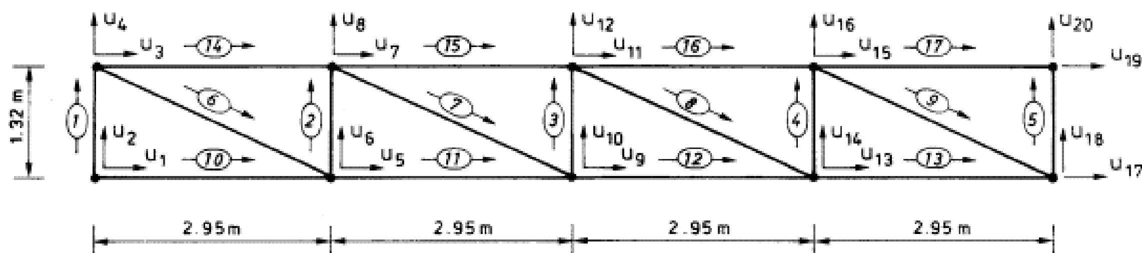
Some difficulties are introduce when linking frames/windows that are created with Komodo. All frames or windows in an application can first be built up and run separately with Komodo. At some stage in a project one window is linked to another window through a button or a menu. To implement this, a function that is run when the window is created has to be added to the sub window. This function also make it possible to setup variables and call functions when the window creates. After this is added the window cannot be run separately, if not a complementary `if` statement is added, which disables the initialize functions described above. In the listing below it is show how a new window is opened and then the extra functions that are added to the sub window's source code. In appendices A.2 & A.5 the entire files are shown.

```
#  Open a new window to set element properties,
   code from the main window

def frmElementProp(self):
    import elementprop
    window=Toplevel()
    elementprop.CustomElementprop(window, projekt)
    window.grab_set()
    window.focus_set()
    window.wait_window()

 _________________________________________
```

Application example



Prerequisite in application example

*E = 2.1 * 105 MPa*

Element 1,2,3,4:	*A = 21.2 * 10-4 m2*
Element 5:	*A = 10.6 * 10-4 m2*
Element 6,7,8,9:	*A = 21.2 * 10-4 m2*
Element 10,11,12,13:	*A = 21.2 * 10-4 m2*
Element 14,15,16,17:	*A = 38.8 * 10-4 m2*

*R4 = R20 = -8.5 * 10-3 MN*
*R8 = R12 = R16 = -17.0 * 10-3 MN*
u2 = u17 = u19 = 0

Application example built in 2DTruss

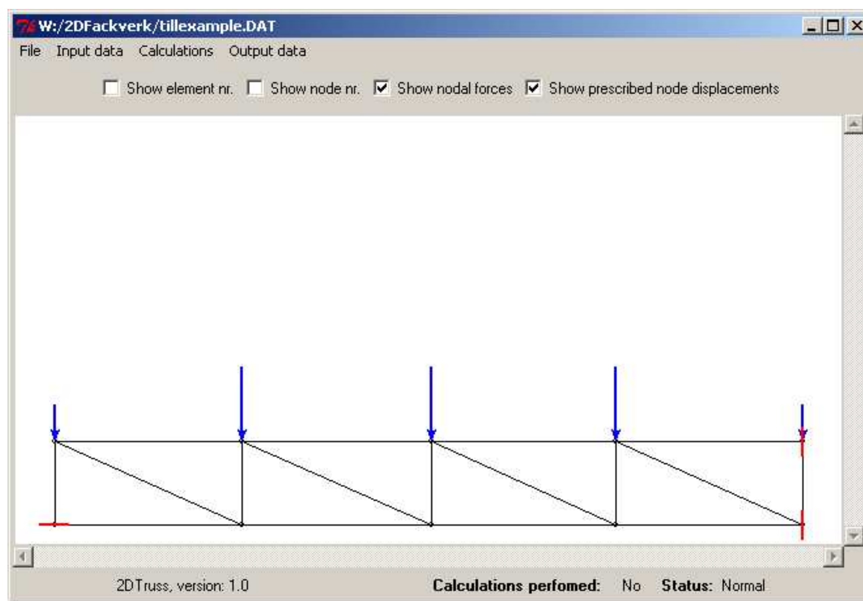
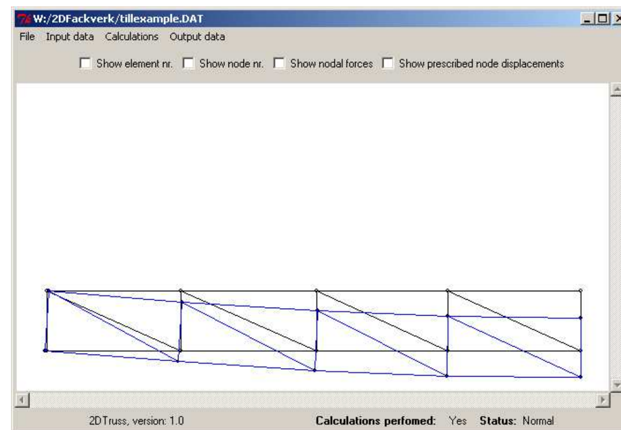


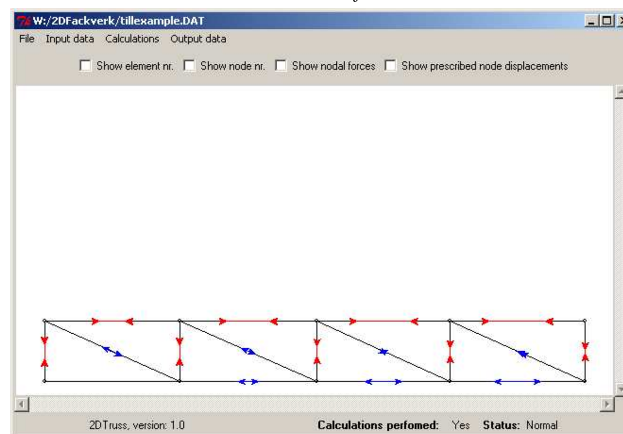
Figure 4.10: *Application example in 2D Truss*

Result Application example

Result: displacment



Result: normal forces



Result file

Resultat

Reaktioner

DOF	Q [N]
2	0.6800E+05
17	0.3046E+06
19	-0.3046E+06

Förskjutningar

DOF	a [m]
1	-0.4284E-02
2	0.000
3	0.3446E-02
4	-0.2016E-03
5	-0.4284E-02
6	-0.2007E-01
7	0.2964E-02
8	-0.2024E-01
9	-0.3388E-02
10	-0.3661E-01
11	0.2119E-02
12	-0.3673E-01
13	-0.1892E-02
14	-0.4661E-01
15	0.1103E-02
16	-0.4669E-01
17	0.000
18	-0.4952E-01
19	0.000
20	-0.4957E-01

Elementkrafter

Element	Normalkraft [N]	Spänning [Pa]
1	-0.6800E+05	-0.3208E+08
2	-0.5950E+05	-0.2807E+08
3	-0.4250E+05	-0.2005E+08
4	-0.2550E+05	-0.1203E+08
5	-8500.	-0.8019E+07
6	0.1457E+06	0.6872E+08
7	0.1055E+06	0.4978E+08
8	0.6155E+05	0.2903E+08
9	0.2081E+05	0.9817E+07
10	0.000	0.000
11	0.1330E+06	0.6272E+08

Stäng fönstret

Figure 4.11: *Result for application example in 2D Truss*

```
#   Extract from element properties window

#   Import class which draws the widgets

from elementprop_ui import Elementprop
.
.
.
#   Init is automatically called when the window is
    created

def __init__(self,root,projekt):
    Elementprop.__init__(self,root)
    self.root=root
    self.root.title("Element properties")
    self.project=project
    self.loadlistbox();
    self.entnrelement.insert(0,str(projekt.
        getprobleminfo()[0]))
.
.
.
#   Functins which is called when the close button
   is preset in the sub window

def btnClose_command(self, *args):
    self.root.destroy()
    pass
```

A start file is needed to change some preference as the title of the main window and the extra initialize function, which is described above is needed too in the mainframe's source code. This is also specific for Komodo. The start file is included in appendix A.1.

To quit an application from a menu created in Komodo use `root.quit` if root is the name of the instance of Tkinter.

The input data windows contains listboxes, entries, labels, checkbuttons, frames and buttons. All these widgets are easy to control with help form the documentation except the checkbutton. Checkbuttons are controlled with built in variables in Tkinter. These variables are created with the variables that are assigned to be a Tkinter class. If this main variables is root then the command `root.setvar(name,value)` creates a variable. To read the stored value of this variable the command `getvar` is used instead. This variable is always a string. Other types of built in variables can be used, but then the name of the variables are set automatically. This is an disadvantage in combination with Komodo GUI builder since the name of variables to control widgets are set manually. The good thing with these variables is

that the variable change automatically when e.g a checkbutton is ticked. The on and off value of a checkbutton can be set to any values, the default is 1 for on and 0 for off in Komodo.

To make a listbox in Komodo GUI builder hold a selection when other widgets are selected the preference `export selection` must be NO.

General for all widgets are that if they are disabled it is the same as if they are read-only.

The import module `tkMessageBox` is used to show short messages in new windows with an OK button.

A function that can be a bit hard to handle is the binding command for events e.g a mouse click or the enter button is pressed. More instructions on how to use a binding event can be found on Pythonware's homepage [17].

Python is started with a console but by giving a Python file the extension PYW it is run without a console.

4.1.6 Python to executable file

Running a Python application on Linux is easy as Python is built in to Linux. A Python application can only be run on Windows if Python is installed or Python scripts are converted into a standalone binary Windows application using a special tool. This tool is called Py2exe [18] and can be found on the Internet. Py2exe makes it possible to run Python program on any Windows computer with or without having Python installed. It is very easy to use, only a small setup file has to be written. To convert 2D Truss into a executable file the following setup file can be used.

```
#   setup file for 2DTruss

from distutils.core import setup
import py2exe

setup(windows=["start.pyw"])

#   Start.pyw is the file that is run to start 2
    DTruss
```

Some additional files can be necessary if the Fortran compiler uses a separate run-time library.

4.2 Graphical user interface for F2PY

F2PY is normally controlled from the command prompt in windows as mentioned in section 3.4.1. Students following the Software Development for Technical Systems course are often non-programmers, which often also means that they are not used to the command prompt. Therefore a graphical

interface has been implemented for F2PY, which makes it easier to use and more effort can be put into learning Fortran and Python instead. The F2PY user interface is made in Python with ActiveState Komodo 3.0 GUI builder, illustrated in figure 4.12. The user interface consist of a single window that makes it easy to understand and use.

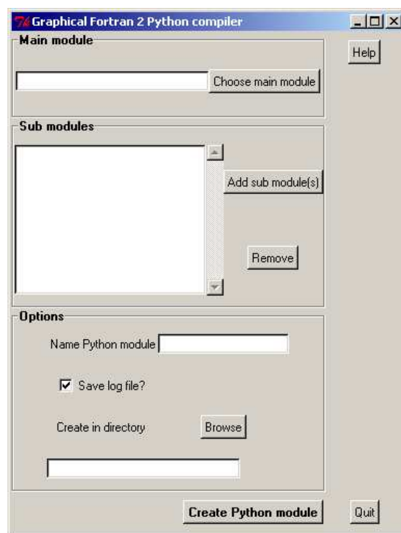


Figure 4.12: *Graphical F2PY*

The user interface for F2PY has multiple support for Fortran files so Fortran code can be divided into multiple Fortran files were the main module has to be specified. The main module must always be selected in the program. An unlimited number of submodules can be added and this is optional requirement. The only additional settings that is needed is the name of the Python module, the directory of the module that is built and if a log file shall be created. The log file is saved in a text file in the same directory as the generated Python module. If any errors occurs during the compilation the text file is displayed in any cases and the errors are highlighted. This makes the user interface very simple to use as only one Fortran file needs to be selected and then the program can create a Python module.

A Python module is created with F2PY when the create Python module button is pressed. The application creates a text string with the modules and settings to F2PY. A runtime operation with the text string is done in Windows command prompt that executes F2PY with the generated string.

```
#  Open commando promt to run the commandline
   generated earlyer
r,w=os.popen4(command)

#  Command is a string which is generated with the
```

```
 same commands as are used to run F2PY in the
command promt

#   Runs the commandoline and reads the output

text=w.readlines()
```

Disadvantage with the wizard is that not all setting for F2PY are supported, but normally it shall work without any problems if it is possible to write attributes for variables in the Fortran code. F2PY user interface is so far in its early stages as not all functions of the F2PY tool is reflected in the user interface, but they will be added when the need advises.

Chapter 5

Summary and Conclusions

5.1 Summary

This master thesis has investigated how forthcoming courses in Software Development for Technical Systems can be implemented with focus on Fortran integration in a script environment such as Python. A finite element application is created where Python is used for the graphical parts and Fortran for computational parts. A tool F2PY was used to create an interface between Python and Fortran. The focus was on the implementation of the linking with F2PY. The application should also have the possibility to be created by non-experienced programmers. They shall focus on Python and Fortran and not learning the intricate details of a complicated language or interface. Python is also examined to see if it fulfills the requirement a finite element application has on the user interface. Finally the application shall be platform independent.

5.2 Conclusions

F2PY is capable of creating Python modules in a single step from Fortran source code and if Numeric F-contiguous variables are used, not a single line of code have to be added to the Fortran code. This makes Python most suitable to create an interface as it is easy to use. Using the created modules in Python is also easy as the same calling conventions as calling internally are used. F2PY is a flexible tool as all major features in Fortran are supported, and the advantages of Fortran are conserved. The disadvantages with F2PY are that it requires five prerequisite and that matrices are stored differently in Python and Fortran. Overall F2PY is surprisingly good tool to create an interface between Python and Fortran and it fulfills all requirements when used in Microsoft Windows.

Python code is easy to implement and has an easy structure since Python is an advanced script language. The easy code structure render possibility

to rapid development and makes Python applications easy in development. User interfaces are developed with the toolkit Tkinter in a GUI-builder. This also decreases developing time and Tkinter contains all necessary widgets and functions for development of user interfaces. Most operations in Python requires a small amount of code if built-in functions are used. Creating complex and flexible applications are also possible as Python support object oriented programming and have a good capability to glue other components. Disadvantages with Python are that everything can be done in many different ways so it can be confusing. Extension toolkits can be need for Tkinter if more advanced user interfaces are wanted. Replacing Delphi with Python encourage new possibilities in the developing of finite element application. The possibilities can be updating software, adding plug-ins, create user interfaces rapid for applications and easy development.

The combination with Python, Fortran and F2PY can be used on most operating systems since all parts are platform independent. F2PY modules has to be compiled for each platform but the same Python source code can be run on any platform.

This results in, the next time Software Development for Technical Systems is given a change in programming languages to Python instead of Borland Delphi is recommended. Fortran is still recommended to be used as the computational language since a link between Python and Fortran is possible to perform. To perform the interface the tool F2PY is suggested to be used. Python has good possibilities to create a user interface if it is used with a GUI-builder.

5.3 Future Work

Some suggestions for future work in this area:

- Investigate if it is possible to create a finite element toolbox for Python, which consists of functions and modules implemented in Fortran.

- Study the gluing possibilities for Python, as gluing a complex mesh generator and solver to a Python user interface.

Bibliography

[1] HENNING, MICHI & VINOSKI, STEVE *Advance CORBA Programming with C++*, Addison Wesley Longman, USA, 1999

[2] LINDEMANN, JONAS *Programming and visualisation techniques in finite element software*, Licentiate Dissertation, KFS i Lund AB, Lund, Sweden, 2001

[3] LINDEMANN, JONAS & DAHLBLOM, OLA *Tillämpad Programmering*, 5:e utgåvan, KFS i Lund AB, Lund, Sweden, 2003

[4] LUTZ, MARK, *Programming Python second edition*, O'reilly, USA, 2001

[5] LUTZ, MARK & ASCHER, DAVID, *Learning Python*, O'reilly, USA, 1999

[6] OUSTERHOUT, JOHN K., *Scripting: Higher Level Programming for the 21st Century*, http://home.pacbell.net/ouster, 2004-12-02, The article appear in IEEE Computer magazine March 1998

Internet references

[7] ACTIVESTATE *Komodo 3.1 homepage*, www.activestate.com, 2005-02-13

[8] ARNHOLM, CARSTEN *Mixed language programming using C++ and FORTRAN 77*, Version 1.1, 28-May-1997, home.online.no/~arnholm

[9] BRAINYENCYCLOPEDIA, *Library (computer science)*, http://www.brainyencyclopedia.com/encyclopedia/l/li/library_computer_science_.html, 2005-02-16

[10] MATHWORKS, *MATLAB*, http://www.mathworks.se, 2005-02-16

[11] NUMERICAL PYTHON, *Numeric and Numarray*, http://numeric.scipy.org/, 2005-02-24

[12] Object Management Group *CORBA homsite*,
http://www.corba.org, 2005-02-10

[13] Peterson, Pearu *F2PY User Guide and Reference Manual*,
Revision 1.25, 2005-01-30,
http://cens.ioc.ee/projects/f2py2e/usersguide/f2py_usersguide.pdf

[14] Python.org *Documentation*, http://www.python.org/doc,
2005-02-18

[15] Python.org, *GUI programming in Python*,
http://www.python.org/moin/GuiProgramming, 2005-02-10

[16] Python.org, *wxPython*, http://www.python.org/moin/wxpython,
2005-02-10

[17] Pythonware *Chapter 7. Events and Bindings*,
http://www.pythonware.com/library/tkinter/introduction/events-and-bindings.htm,
2005-02-18

[18] Python to Executable homepage
http://starship.python.net/crew/theller/py2exe/, 2005-02-10

[19] Secret Labs *PythonWare and effbot.org*, www.Pythonware.com &
www.effbot.org, 2004-12-02

[20] Simplified wrapper and Interface Generator homepage,
http://www.swig.org, 2005-02-10

[21] Structural Mechanics, Lunds Institue of Technology,
http://www.byggmek.lth.se, 2005-02-10

[22] Tecfacs, *Overview of Dynamic Linking*,
http://www.tecfacs.com/dynamic.htm, 2005-02-16

[23] wxPython's official homepage http://www.wxpython.org/,
2005-02-13

Appendix A

Python Modules

A.1 Start

```
#   Start file for 2DTruss

#   Import modul with the mainframe for 2DTruss

from MainFrame import*

#   Make a loop

try: userinit()
except NameError: pass
root = Tk()
demo = CustomMainFrame(root)
root.title('2DTruss') # Title of the mainframe
try: run()
except NameError: pass
root.protocol('WM_DELEimport␣cpickleTE_WINDOW',
    root.quit)
root.mainloop()
```

A.2 Mainframe

```
"""␣MainFrame.py␣--

␣UI␣generated␣by␣GUI␣Builder␣Build␣107673␣on
    ␣2005-02-09␣18:18:22␣from:
␣␣␣␣W:/2DFackverk/MainFrame.ui
␣This␣file␣is␣auto-generated.␣␣Only␣the␣code␣within
␣␣␣␣'#␣BEGIN␣USER␣CODE␣(global|class)'
␣␣␣␣'#␣END␣USER␣CODE␣(global|class)'
```

```
␣and␣code␣inside␣the␣callback␣subroutines␣will␣be␣
    round-tripped.
 The 'main' function is reserved.
"""

from Tkinter import *
from MainFrame_ui import MainFrame

#   BEGIN USER CODE global
from base import*
import modeldraw
#   END USER CODE global

class CustomMainFrame(MainFrame):
    pass

    #   BEGIN CALLBACK CODE
    #   ONLY EDIT CODE INSIDE THE def FUNCTIONS.

    #   chkbtnShowEnr_command --
    #
    #   Callback to handle chkbtnShowEnr widget
        option -command

    def chkbtnShowEnr_command(self, *args):

        #   Check if the checkbutton is marked or
            not then the elementnumbers are drawn or
             deleted

        if int(self.root.getvar('showElNr')):
            modeldraw.printElementNr(self.canmain,
                projekt)
        else:
            self.canmain.delete('elnr')
        pass

    #   chkbtnShowNnl_command --
    #
    #   Callback to handle chkbtnShowNnl widget
        option -command

    def chkbtnShowNnl_command(self, *args):

        #   Check if the checkbutton is marked or
            not then the nodeforces are drawn or
            deleted

        if int(self.root.getvar('showNodeForce')):
```

```
            modeldraw.printNodeForce(self.canmain,
                projekt)
        else:
            self.canmain.delete('nodeforce')
        pass

#   chkbtnShowNnr_command --
#
#   Callback to handle chkbtnShowNnr widget
    option -command

    def chkbtnShowNnr_command(self, *args):

        #   Check if the checkbutton is marked or
            not then the nodenumbers are drawn or
            deleted

        if int(self.root.getvar('showNodeNr')):
            modeldraw.printNodeNr(self.canmain)
        else:
            self.canmain.delete('nodenr')
        pass

#   chkbtnShowNpv_command --
#
#   Callback to handle chkbtnShowNpv widget
    option -command

    def chkbtnShowNpv_command(self, *args):

        #   Check if the checkbutton is marked or
            not then the prescribde sign for the
            nodes are drawn or deleted

        if int(self.root.getvar('showPreNodes')):
            modeldraw.printPreNode(self.canmain,
                projekt)
        else:
            self.canmain.delete('prenode')
        pass

#   END CALLBACK CODE

#   BEGIN USER CODE class

#   When the frame is created

    def __init__(self,root):
        MainFrame.__init__(self,root)
```

```
        self.root=root

        #       Redraw if the window is modify like
            resized

        self.canmain.bind('<Configure>',self.draw)
        self.root.setvar('showElNr', 0)
        self.root.setvar('showNodeNr', 0)
        self.root.setvar('showNodeForce', 0)
        self.root.setvar('showPreNodes', 0)

#   Open window for showing the result

    def frmShowResult(self):
        import ShowResult
        window=Toplevel()
        ShowResult.CustomShowResult(window,projekt)
        window.grab_set()
        window.focus_set()
        window.wait_window()

#   Open window that shows the information
    about the problem

    def frmProblemInfo(self):
        import Probleminformation
        window=Toplevel()
        Probleminformation.CustomProbleminformation
            (window,projekt)
        window.grab_set()
        window.focus_set()
        window.wait_window()
        projekt.setCalcMade(0,self.lbcalc)

#   Open window to set nen and node properties

    def frmNodeInfo(self):
        import nodeinfo
        window=Toplevel()
        nodeinfo.CustomNodeinfo(window,projekt)
        window.grab_set()
        window.focus_set()
        window.wait_window()
        projekt.setCalcMade(0,self.lbcalc)

#   Open a new window to set element properties

    def frmElementProp(self):
        import elementprop
```

```
        window=Toplevel()
        elementprop.CustomElementprop(window,
            projekt)
        window.grab_set()
        window.focus_set()
        window.wait_window()
        projekt.setCalcMade(0,self.lbcalc)

#   Open window to set prescribde node values

    def frmPreNodes(self):
        import prescribednodes
        window=Toplevel()
        prescribednodes.CustomPrescribednodes(
            window,projekt)
        window.grab_set()
        window.focus_set()
        window.wait_window()
        projekt.setCalcMade(0,self.lbcalc)

#   Open window to set nodalforces

    def frmNodeForces(self):
        import nodeforces
        window=Toplevel()
        nodeforces.CustomNodeforces(window,projekt)
        window.grab_set()
        window.focus_set()
        window.wait_window()
        projekt.setCalcMade(0,self.lbcalc)

#   Function to open an existing project

    def open(self):
        global projekt
        projekt=Open()
        infile=projekt.getinfilename()
        if infile: self.root.title(infile)
        self.draw(1,dm=1)
        projekt.setCalcMade(projekt.isCalcMade(),
            self.lbcalc)
        self.lbstatus.config(text='Normal')

#   Function that calculates the displacment
    for each node and normalforces in evrey
    element

    def execute(self):
        Execute(projekt,self.lbcalc)
```

```
#   Function that creates a new projekt

def new(self):
    global projekt
    projekt=Model()
    New(projekt)
    self.root.title('2D␣Fackverk␣-␣New␣Project'
        )
    self.draw(1,dm=1)
    projekt.setCalcMade(0,self.lbcalc)
    self.lbstatus.config(text='Normal')

#   Fuction that saves your project

def save(self):
    infile=projekt.getinfilename()
    if infile:
        Save(projekt)
    else:self.saveas()

#   Fuction that save your project as ...

def saveas(self):
    SaveAs(projekt)
    infile=projekt.getinfilename()
    if infile: self.root.title(infile)

#   Function to draw elements, dispacement of
    the structure or normalforces in the
    elements

def draw(self,event,dm=None):
    if dm: modeldraw.Displaymode=dm

    #   get the size of the canvas

    height=self.canmain.winfo_height()
    width=self.canmain.winfo_width()

    #   Don't draw anything

    if modeldraw.Displaymode==1:
        self.canmain.delete('all')

    #   Draw the structure

    elif modeldraw.Displaymode==2:
        self.canmain.delete('all')
```

```
        modeldraw.DrawGeometry(width, height,
            self.canmain, projekt, self.root)

    #   Structure with displacement

    elif modeldraw.Displaymode==3:
        if projekt.isCalcMade():
            self.canmain.delete('all')
            modeldraw.DrawDisplacement(width,
                height, self.canmain, projekt, self
                .root)
        else:
            modeldraw.Displaymode=1
            showerror('Fatal␣Error','
                Beräkningar␣ej␣utförda')

    #   Structuer with normalforces

    elif modeldraw.Displaymode==4:
        if projekt.isCalcMade():
            self.canmain.delete('all')
            modeldraw.DrawForces(width, height,
                self.canmain, projekt, self.root)
        else:
            modeldraw.Displaymode=1
            showerror('Fatal␣Error','
                Beräkningar␣ej␣utförda')

#   options to set which to draw

def drawGeo(self): self.draw(1,dm=2)
def drawDisp(self): self.draw(1,dm=3)
def drawForce(self): self.draw(1,dm=4)

#   Open a resultfile

def openResultFile(self):
    global projekt
    projekt=Model()
    self.root.title('2D␣Fackverk␣−␣Resultatfil'
        )
    self.draw(1,dm=1)
    projekt.setCalcMade(0, self.lbcalc)
    self.lbcalc.config(text='−−−')
    self.lbstatus.config(text='Endast␣
        resultatfil')
    outfile=tkFileDialog.askopenfilename(title=
        'Öppna␣en␣resultatfil', filetypes=[('
        resultatfil','*.lst'),('Alla␣Filer','*')
```

```
            ])
        projekt.setoutfilename(outfile)
        self.frmShowResult()

    #   END USER CODE class

#   is only used if the frame is run seperatly, is
    auto generated.

def main():
    #   Standalone Code Initialization
    #   DO NOT EDIT
    try: userinit()
    except NameError: pass
    root = Tk()
    demo = CustomMainFrame(root)
    root.title('MainFrame')
    try: run()
    except NameError: pass
    root.protocol('WM_DELETE_WINDOW', root.quit)
    root.mainloop()

if __name__ == '__main__': main()
```

A.3 Base

```
#   Begin Module base


#   import other modules

import tkFileDialog
from tkMessageBox import*
from Numeric import*
from node import node
from element import element
import cPickle
import solvepro



#   Begin Class model

class Model:

    #   Runs when the class is created
```

```
def __init__(self):

    self.nel=0 # number of elements
    self.nen=0 # number of nodes
    self.nnl=0 # nuber of nodal loades
    self.npv=0 # number of prescribed
        nodevalues

    self.nodelist=[] # list that contains all
        nodes in the problem
    self.elementlist=[] # list that contains
        all elements in the problem

    self.infile='' # Name of the input file
    self.outfile=''# Name of the output file

#   Array that contains the displacement of
    the nodes (after calculations)

    self.disp=zeros(self.nen*2,'d')

#   Forces in the elements (after
        calculations)

    self.eforces=solvepro.
        as_column_major_storage(zeros((2,self.
        nel),'d'))

#   calcMade=1 if calculations are made

    self.calcMade=0

def setprobleminfo(self, nel=None,nen=None,nnl=
    None,npv=None):
    if nel : self.nel=nel
    if nen : self.nen=nen
    if nnl : self.nnl=nnl
    if npv : self.npv=npv

def getprobleminfo(self):
    return [self.nel, self.nen, self.nnl, self.
        npv]

#   Returns the node with index in the nodelist

def getnode(self,index):
    return self.nodelist[index]

#   Returns a element from elementlist
```

```
def getelement(self,index):
    return self.elementlist[index]

#   Make a new element

def addElement(self,element):
    self.elementlist.append(element)

#   Add a new node to the problem

def addNode(self,node):
    self.nodelist.append(node)
    node.setdof(len(self.nodelist)-1)

#   Remove a node

def removeNode(self,index):
    del self.nodelist[index]

#   Remove a element

def removeElement(self,index):
    del self.elementlist[index]

#   Change infilename

def setinfilename(self,name):
    self.infile=name

#   Change outfilename

def setoutfilename(self,name):
    self.outfile=name

def getinfilename(self):
    return self.infile

def getoutfilename(self):
    return self.outfile

def setDispAndEforceToZeroAndSetSize(self):
    self.disp=zeros(self.nen*2,'d')

    #   Fortran uses column major storage

    self.eforces=solvepro.
        as_column_major_storage(zeros((2,self.
        nel),'d'))
```

```
    def setDispAndEforce(self,disp,eforces):
        self.disp=disp
        self.eforces=eforces

    #   returns Disp and eforces

    def getDispAndEforces(self):
        return [self.disp,self.eforces]

    #   Controlls if a calculation is made

    def isCalcMade(self):
        return self.calcMade

    def setCalcMade(self,calc,label):
        self.calcMade=calc
        if calc:label.config(text='Yes')
        else:label.config(text='No')

#   End Class model

#   Model functions but not a part of the model
    class

#   Calculate the problem in fortran

def Execute(projekt,lbcalc):
    infile=projekt.getinfilename()

    #   Check if there is a inputfile

    if infile:
        projekt.setoutfilename(infile[:-3]+'lst')

        #   Creates the neccesary variabels from
            the model to calculate the problem

        [nel,neq,nnl,npv,eprop,edof,f,bc,b,ierr]=
            createArrays(projekt)

        #   lg in if ierrrorr

        outfile=projekt.getoutfilename()
        projekt.setDispAndEforceToZeroAndSetSize()
        disp,eforces=projekt.getDispAndEforces()

        #   solvepro uses a python module created
            with f2py from fortran. Matrix and
```

```
        arrays must be in numarray or numeric
        format to be abel to use

    solvepro.calcprog.execute(nel,neq,nnl,npv,
        eprop,edof,f,bc,b,eforces,outfile,ierr,
        disp)

    #   ierr is a errorflag if the probelm is
        nonpositive definit

    if ierr==1:
        projekt.setCalcMade(0,lbcalc)
        showerror('Problemet␣ej␣lÃűst','
            Problemet␣Ãd'r␣inte␣positivt␣definit␣
            och␣ej␣lÃűst')
        lbcalc.config(text='No')
    else:
        projekt.setDispAndEforce(disp,eforces)
        projekt.setCalcMade(1,lbcalc)
        showinfo('Klart','BerÃd'kningarna␣Ãd'r␣
            utfÃűrda')

else:
    showerror('Inte␣mÃűjligt!', "Ingen␣
        indatafil")

#   Function to open a projekt/inputfile

def Open():
    infile=tkFileDialog.askopenfilename(title='
        ÃŰppna␣ett␣projekt', filetypes=[('Projekt/
        indata','*.DAT'),('Alla␣Filer','*')])

    #   projekt.setinfilename(infile)

    file=open(infile, 'r')
    projekt=cPickle.load(file)
    file.close()
    return projekt

#   Creates a new projekt

def New(projekt):
    projekt.setinfilename('')

#   Saves a projekt

def Save(projekt):
```

```
    infile=projekt.getinfilename()
    file=open(infile, 'w')
    cPickle.dump(projekt, file)
    file.close()

#   save as

def SaveAs(projekt):
    infile=tkFileDialog.asksaveasfilename(title='
        Spara␣ett␣projekt',defaultextension='.DAT',
        filetypes=[('Projekt/indata','*.DAT'),('Alla
        ␣Filer','*')])
    projekt.setinfilename(infile)
    file=open(infile, 'w')
    cPickle.dump(projekt, file)
    file.close()

#   Function that creates the nessesarry arrays
   that are needed to solve the problem with
   execute

def createArrays(projekt):

    var=projekt.getprobleminfo()
    nel=var[0]
    neq=var[1]*2
    nnl=var[2]
    npv=var[3]

    #   Creates the arrays with zeros

    ierr=array(0)
    eprop=zeros((nel,6),'d')
    edof=zeros((nel,4),'l')
    f=zeros(neq,'d')
    b=zeros(neq,'l')
    bc=zeros(neq,'d')

    #   Fill the arrays with information about the
        problem after a patter from a indata manual
        for the fortransolver

    for i in xrange(nel):
        e=projekt.getelement(i)
        n1=projekt.getnode(e.getnodes()[0]-1)
        n2=projekt.getnode(e.getnodes()[1]-1)

        epropel=[e.getprop()[1],e.getprop()[0],n1.
            getcoord()[0],n1.getcoord()[1],n2.
```

```
            getcoord()[0],n2.getcoord()[1]]
        edofel=[n1.getdof()[0],n1.getdof()[1],n2.
            getdof()[0],n2.getdof()[1]]

        eprop[i]=epropel
        edof[i]=edofel

    for i in xrange(var[1]):
        n=projekt.getnode(i)
        f[i*2]=n.getnodeforces()[0]
        f[i*2+1]=n.getnodeforces()[1]
        b[i*2]=n.getb()[0]
        b[i*2+1]=n.getb()[1]
        bc[i*2]=n.getbc()[0]
        bc[i*2+1]=n.getbc()[1]

#   End filling the arrays with information
    about the problem

#   Transpose the arrays to get the stored
    columnwise like fortran needs and not
    rowwise like python makes by default

    eprop=transpose(eprop)
    edof=transpose(edof)

    return nel,neq,nnl,npv,eprop,edof,f,bc,b,ierr
```

A.4 Modeldraw

```
#   Module modeldraw to draw in the drawing area

#   percentage of the drawing area to be drawn

k=0.9

#   Handels what is going to be drawn, 1=nothing is
    drawn

Displaymode=0

#   Coordvalues calculates the variabels that
    transform the coordinates from normal to
    computer coordinatessytem and makes the
    strucutre to fit into the window

def coordvalues(width,height):
```

```
    global s,x0,y0
    s=min([k*width/(Xmax-Xmin),k*height/(Ymax-Ymin)
        ])
    x0=-s*Xmin+(1-k)/2*width
    y0=height+s*Ymin-(1-k)/2*height

#   Claculates max and min X,Y, it also returns the
    nodes X and Y coordinates in lists

def maxminXY(width, height, projekt):
    global Xmax,Ymax,Xmin,Ymin,var,X,Y
    var=projekt.getprobleminfo()
    X=[0]*var[1]
    Y=[0]*var[1]

    for i in xrange(var[1]):
        node=projekt.getnode(i)
        X[i]=node.getcoord()[0]
        Y[i]=node.getcoord()[1]


    if Displaymode==2 or Displaymode==4:
        Xmax=max(X)
        Ymax=max(Y)
        Xmin=min(X)
        Ymin=min(Y)

        coordvalues(width, height)

    #   Displaymode 3 is diffrent though the max
        min XY changes when the displacement is
        drawn

    elif Displaymode==3:
        dispX=[0.]*var[1]
        dispY=[0.]*var[1]
        disp=projekt.getDispAndEforces()[0]
        absdisp=[0.]*len(disp)

        for i in xrange(len(disp)):absdisp[i]=abs(
            disp[i])
        sd=0.05*max(abs(max(X)-min(X)),abs(max(Y)-
            min(Y)))/max(absdisp) # Scalefactor

        for i in xrange(var[1]):
            node=projekt.getnode(i)
            dispX[i]=X[i]+disp[i*2]*sd
            dispY[i]=Y[i]+disp[i*2+1]*sd
```

```
        Xmax=max(max(X),max(dispX))
        Ymax=max(max(Y),max(dispY))
        Xmin=min(min(X),min(dispX))
        Ymin=min(min(Y),min(dispY))

        coordvalues(width,height)

        return dispX,dispY

#   Function that draws the gometry for the
    structure with elementnumbers and nodes

def DrawGeometry(width,height,canvasMain,projekt,
    root):

    if Displaymode !=3: maxminXY(width,height,
        projekt)

    #   Draws one element att the time

    for i in xrange(var[0]):
        nodes=projekt.getelement(i).getnodes()

        x1=x0+s*X[nodes[0]-1]
        x2=x0+s*X[nodes[1]-1]
        y1=y0-s*Y[nodes[0]-1]
        y2=y0-s*Y[nodes[1]-1]

        canvasMain.create_line(x1,y1,x2,y2, tag='
            lines')
        canvasMain.create_oval(int(x1-0.03*s),int(
            y1-0.03*s),int(x1+0.03*s),int(y1+0.03*s)
            , tag='circ')
        canvasMain.create_oval(int(x2-0.03*s),int(
            y2-0.03*s),int(x2+0.03*s),int(y2+0.03*s)
            , tag='circ')


    canvasMain.tkraise('text')

    #   Draws elementnumbers if the checkbutton is
        filled

    if int(root.getvar('showElNr'))==1:
        printElementNr(canvasMain,projekt)

    #   Draws nodenumbers if the checkbutton is
        filled
```

```
    if int(root.getvar('showNodeNr'))==1:
        printNodeNr(canvasMain)

    #  Draws nodal forces if the checkbutton is
        filled

    if int(root.getvar('showNodeForce'))==1:
        printNodeForce(canvasMain, projekt)

    #  Draws prescribde dof if the checkbutton is
        filled

    if int(root.getvar('showPreNodes'))==1:
        printPreNode(canvasMain, projekt)

#  Function that first draws the geometry with "
    DrawGeometry" and then adds the displacement of
    the elements in blue color

def DrawDisplacement(width, height, canvasMain,
    projekt, root):

    dispX, dispY=maxminXY(width, height, projekt)
    DrawGeometry(width, height, canvasMain, projekt,
        root)

    # Draws one element att the time

    for i in xrange(var[0]):
        nodes=projekt.getelement(i).getnodes()
        xd1=x0+s*dispX[nodes[0]-1]
        xd2=x0+s*dispX[nodes[1]-1]
        yd1=y0-s*dispY[nodes[0]-1]
        yd2=y0-s*dispY[nodes[1]-1]
        canvasMain.create_line(xd1,yd1,xd2,yd2, fill
            ='blue', tag='dlines')
        canvasMain.create_oval(int(xd1-0.03*s),int(
            yd1-0.03*s),int(xd1+0.03*s),int(yd1
            +0.03*s),fill='blue', tag='dcirc')
        canvasMain.create_oval(int(xd2-0.03*s),int(
            yd2-0.03*s),int(xd2+0.03*s),int(yd2
            +0.03*s),fill='blue', tag='dcirc')

    canvasMain.tkraise('text')

#  Function that first draw the geometry with "
    DrawGeometry" and the adds arrow which are
    equivalent to tensile and compressive
    normalstress
```

```
def DrawForces(width, height, canvasMain, projekt, root
    ):
    from math import sqrt

    DrawGeometry(width, height, canvasMain, projekt,
        root)

    force=projekt.getDispAndEforces()[1]
    force=force[0]
    absforce=[0.]*len(force)
    for i in xrange(len(force)):absforce[i]=abs(
        force[i])
    sd=0.1*max(abs(Xmax-Xmin),abs(Ymax-Ymin))/max(
        absforce) #Skalfaktor

#   Draws one element att the time

    for i in xrange(var[0]):
        nodes=projekt.getelement(i).getnodes()
        x1=X[nodes[0]-1]
        y1=Y[nodes[0]-1]
        x2=X[nodes[1]-1]
        y2=Y[nodes[1]-1]

        # Half length of the force arrow

        halflength=abs(force[i]*sd/2)

        #   Calc half the length of the element

        lengthtomid=sqrt(((x2-x1)/2)**2+((y1-y2)/2)
            **2)

        #   Finds the pixel were to start drawing
            the force arrow

        deltax=(lengthtomid-halflength)/lengthtomid
            *(x2-x1)/2
        deltay=(lengthtomid-halflength)/lengthtomid
            *(y2-y1)/2
        xd1=x0+s*(x1+deltax)
        yd1=y0-s*(y1+deltay)
        xd2=x0+s*(x2-deltax)
        yd2=y0-s*(y2-deltay)

        #   If tensile stress

        if force[i]>0:
```

```
            canvasMain.create_line(xd1,yd1,xd2,yd2,
                arrow='both', tag='arrow',arrowshape
                =(8,10,3),fill='blue')

        #   If compressive stress

        else:
            canvasMain.create_line(xd1,yd1,xd2,yd2
                , tag='arrow',fill='red',arrow='both
                ',arrowshape=(-8,-10,-5))

#   Function to draw element number

def printElementNr(canvasMain,projekt):

    #   Goes through all elements and prints the
        number, var[0] is number of elements

    for i in xrange(var[0]):
        n=projekt.getelement(i).getnodes()
        canvasMain.create_text(int(((X[n[0]-1]+X[n
            [1]-1])*s+2*x0)/2),int((-(Y[n[0]-1]+Y[n
            [1]-1])*s+2*y0)/2),text=str(i+1),fill='#
            AA0000',tag='elnr',anchor='s',font='{MS␣
            Sans␣Serif}␣12')

#   Function print node numbers

def printNodeNr(canvasMain):

    #   Runs a for loop for each node and prints
        the number, var[1] is nuber of nodes

    for i in xrange(var[1]):
        canvasMain.create_text(x0+s*X[i],y0-s*Y[i],
            text=str(i+1),fill='blue',tag='nodenr',
            anchor='s',font='{MS␣Sans␣Serif}␣12')

#   Function that draws the nodeforces

def printNodeForce(canvasMain,projekt):

    #   Finds the biggest nodeforce and makes it
        10% of the size of the structure

    absf=[]
    for i in xrange(var[1]):
        ftemp=projekt.getnode(i).getnodeforces()
```

```
            absf.append(max(abs(ftemp[0]),abs(ftemp[1])
                ))
        sd=0.1*max(abs(Xmax-Xmin),abs(Ymax-Ymin))/max(
            absf) #Skalfaktor

    #   Make a loop for each node and draws an
        arrow to symbolize the nodeforce, one for x-
        direction and one for y

        for i in xrange(var[1]):
            f=projekt.getnode(i).getnodeforces()
            if f[0]:
                canvasMain.create_line(x0+s*(X[i]-f[0]*
                    sd),y0-s*Y[i],x0+s*X[i],y0-s*Y[i],
                    arrow='last',fill='blue',tag='
                    nodeforce',width='2')
            if f[1]:
                canvasMain.create_line(x0+s*X[i],y0-s*(
                    Y[i]-f[1]*sd),x0+s*X[i],y0-s*Y[i],
                    arrow='last',fill='blue',tag='
                    nodeforce',width='2')

#   Function to draw a sign if the node i
    sprescribed or not

def printPreNode(canvasMain,projekt):

    #   Draws the sign 2% of the size of the
        structure

    sd=0.02*max(abs(Xmax-Xmin),abs(Ymax-Ymin)) #
        Scalefactor

    #   Runs a loop for each node and check if the
        deegres of freedom at the node i prescribde
        or not.

    for i in xrange(var[1]):
        b=projekt.getnode(i).getb()
        if b[0]:
            canvasMain.create_line(x0+s*X[i],y0-s*(
                Y[i]+sd),x0+s*X[i],y0-s*(Y[i]-sd),
                fill='red',tag='prenode',width='2')
        if b[1]:
            canvasMain.create_line(x0+s*(X[i]+sd),
                y0-s*Y[i],x0+s*(X[i]-sd),y0-s*Y[i],
                fill='red',tag='prenode',width='2')
```

A.5 Elementproperties

```
"""␣elementprop.py␣--

␣UI␣generated␣by␣GUI␣Builder␣Build␣107673␣on
    ␣2005-02-08␣19:26:20␣from:
␣␣␣␣W:/2DFackverk/elementprop.ui
␣This␣file␣is␣auto-generated.␣␣Only␣the␣code␣within
␣␣␣␣'#␣BEGIN␣USER␣CODE␣(global|class)'
␣␣␣␣'#␣END␣USER␣CODE␣(global|class)'
␣and␣code␣inside␣the␣callback␣subroutines␣will␣be␣
    round-tripped.
␣The␣'main'␣function␣is␣reserved.
"""

from Tkinter import *
from elementprop_ui import Elementprop

#   BEGIN USER CODE global

from tkMessageBox import showerror

#   END USER CODE global

class CustomElementprop(Elementprop):
    pass

    #   BEGIN CALLBACK CODE
    #   ONLY EDIT CODE INSIDE THE def FUNCTIONS.

    #   btnClose_command --
    #
    #   Callback to handle btnClose widget option -
        command

    def btnClose_command(self, *args):
        self.root.destroy()
        pass

    #   btnSave_command --
    #
    #   Callback to handle btnSave widget option -
        command

    def btnSave_command(self, *args):
        try:
            n1=int(self.entfromnode.get())
            n2=int(self.enttonode.get())
```

```
            emodule=float(self.entemodule.get())
            area=float(self.entarea.get())
        except:
            showerror('Error!', "Det var inga 
                giltiga tal")
        else:
            index=self.lbelements.curselection();
            self.projekt.getelement(int(index[0])).
                setprop(emodule,area)
            self.projekt.getelement(int(index[0])).
                setnodes(n1,n2)
        pass

#   btnuppdate_command --
#
#   Callback to handle btnuppdate widget option
    -command

    def btnuppdate_command(self, *args):
        from element import element
        temp=self.projekt.getprobleminfo()[0]
        try:
            nel=int(self.entnrelement.get())
        except:
            showerror('Error!', "Det var inget 
                heltal")
        else:
            self.projekt.setprobleminfo(nel=nel)
            self.loadlistbox()
            if temp<=nel:
                for i in xrange(nel-temp):
                    self.projekt.addElement(element
                        (0,0))
            else:
                for i in xrange(temp-nel):
                    self.projekt.removeElement(temp
                        -i-1)
        pass

#   END CALLBACK CODE

#   BEGIN USER CODE class

#   Init happends when the window is opened

    def __init__(self,root,projekt):
        Elementprop.__init__(self,root)
        self.root=root
        self.root.title("Element properties")
```

```
        self.projekt=projekt
        self.loadlistbox();
        self.entnrelement.insert(0,str(projekt.
            getprobleminfo()[0]))

    #   Fills the lisbox with elementsnumbers

    def loadlistbox(self):
        self.lbelements.delete(0, END)
        nr=self.projekt.getprobleminfo()[0]
        for i in xrange(nr):
            self.lbelements.insert(i,str(i+1))

        #   This bind command controls what
            happends when a element i selected

        self.lbelements.bind('<ButtonRelease-1>',
            self.selectionlbelement)

    #   When a element is selected the entrys are
        filled with the actuall properties for the
        element and they can be change

    def selectionlbelement(self,event):
        index=self.lbelements.curselection();
        element=self.projekt.getelement
        n=element(int(index[0])).getnodes()
        prop=element(int(index[0])).getprop()
        self.entfromnode.delete(0,END)
        self.enttonode.delete(0,END)
        self.entemodule.delete(0,END)
        self.entarea.delete(0,END)
        self.entfromnode.insert(0,n[0])
        self.enttonode.insert(0,n[1])
        self.entemodule.insert(0,prop[0])
        self.entarea.insert(0,prop[1])

    #   END USER CODE class


#   is only used if the frame is run seperatly, is
    auto generated.

def main():
    #   Standalone Code Initialization
    #   DO NOT EDIT
    try: userinit()
    except NameError: pass
    root = Tk()
```

```
    demo = CustomElementprop(root)
    root.title('elementprop')
    try: run()
    except NameError: pass
    root.protocol('WM_DELETE_WINDOW', root.quit)
    root.mainloop()

if __name__ == '__main__': main()
```

A.6 Elementproperties ui

```
"""␣elementprop_ui.py␣--

␣UI␣generated␣by␣GUI␣Builder␣Build␣107673␣on
    ␣2005-02-08␣19:26:18␣from:
␣␣␣␣W:/2DFackverk/elementprop.ui
THIS␣IS␣AN␣AUTOGENERATED␣FILE␣AND␣SHOULD␣NOT␣BE␣
    EDITED.
The␣associated␣callback␣file␣should␣be␣modified␣
    instead.
"""

import Tkinter

class Elementprop:
    def __init__(self, root):


        # Widget Initialization
        self.labfrm = Tkinter.LabelFrame(root,
             text = 'Element',
        )
        self.frmegenskaper = Tkinter.LabelFrame(
            root,
             text = 'Properties',
        )
        self.lbelements = Tkinter.Listbox(self.
            labfrm,
             activestyle = 'dotbox',
             exportselection = '0',
             height = '15',
             takefocus = '1',
             width = '0',
        )
        self._scrollbar_1 = Tkinter.Scrollbar(self.
            labfrm,
        )
```

```
self.labfnode = Tkinter.Label(self.
    frmegenskaper,
     text = 'From␣node',
)
self.labtnode = Tkinter.Label(self.
    frmegenskaper,
     text = 'To␣node',
)
self.labe = Tkinter.Label(self.
    frmegenskaper,
     text = 'E␣(Pa)',
)
self.laba = Tkinter.Label(self.
    frmegenskaper,
     text = 'Area␣(m2)',
)
self.entfromnode = Tkinter.Entry(self.
    frmegenskaper,
     takefocus = '1',
     width = '10',
)
self.enttonode = Tkinter.Entry(self.
    frmegenskaper,
     takefocus = '1',
     width = '10',
)
self.entarea = Tkinter.Entry(self.
    frmegenskaper,
     width = '10',
)
self.entnrelement = Tkinter.Entry(root,
     width = '5',
)
self.labnrel = Tkinter.Label(root,
     text = 'Number␣of␣elements',
)
self.btnuppdate = Tkinter.Button(root,
     text = 'Update',
)
self.btnSave = Tkinter.Button(self.
    frmegenskaper,
     text = 'Save',
)
self.btnClose = Tkinter.Button(root,
     text = 'Close',
)
self.entemodule = Tkinter.Entry(self.
    frmegenskaper,
     width = '10',
```

```
)

# widget commands

self.lbelements.configure(
    xscrollcommand = self.
        lbelements_xscrollcommand
)
self.lbelements.configure(
    yscrollcommand = self._scrollbar_1.set
)
self._scrollbar_1.configure(
    command = self.lbelements.yview
)
self.entfromnode.configure(
    invalidcommand = self.
        entfromnode_invalidcommand
)
self.entfromnode.configure(
    validatecommand = self.
        entfromnode_validatecommand
)
self.entfromnode.configure(
    xscrollcommand = self.
        entfromnode_xscrollcommand
)
self.enttonode.configure(
    invalidcommand = self.
        enttonode_invalidcommand
)
self.enttonode.configure(
    validatecommand = self.
        enttonode_validatecommand
)
self.enttonode.configure(
    xscrollcommand = self.
        enttonode_xscrollcommand
)
self.entarea.configure(
    invalidcommand = self.
        entarea_invalidcommand
)
self.entarea.configure(
    validatecommand = self.
        entarea_validatecommand
)
self.entarea.configure(
    xscrollcommand = self.
        entarea_xscrollcommand
```

```
)
self.entnrelement.configure(
    invalidcommand = self.
        entnrelement_invalidcommand
)
self.entnrelement.configure(
    validatecommand = self.
        entnrelement_validatecommand
)
self.entnrelement.configure(
    xscrollcommand = self.
        entnrelement_xscrollcommand
)
self.btnuppdate.configure(
    command = self.btnuppdate_command
)
self.btnSave.configure(
    command = self.btnSave_command
)
self.btnClose.configure(
    command = self.btnClose_command
)
self.entemodule.configure(
    invalidcommand = self.
        entemodule_invalidcommand
)
self.entemodule.configure(
    validatecommand = self.
        entemodule_validatecommand
)
self.entemodule.configure(
    xscrollcommand = self.
        entemodule_xscrollcommand
)


# Geometry Management
self.labfrm.grid(
    in_    = root,
    column = 1,
    row    = 1,
    columnspan = '1',
    ipadx = '0',
    ipady = '0',
    padx = '0',
    pady = '0',
    rowspan = '4',
    sticky = 'news'
)
```

```
self.frmegenskaper.grid(
    in_     = root,
    column = 2,
    row     = 1,
    columnspan = '1',
    ipadx = '0',
    ipady = '0',
    padx = '0',
    pady = '0',
    rowspan = '4',
    sticky = 'news'
)
self.lbelements.grid(
    in_     = self.labfrm,
    column = 1,
    row     = 1,
    columnspan = '1',
    ipadx = '0',
    ipady = '0',
    padx = '0',
    pady = '0',
    rowspan = '1',
    sticky = 'news'
)
self._scrollbar_1.grid(
    in_     = self.labfrm,
    column = 2,
    row     = 1,
    columnspan = '1',
    ipadx = '0',
    ipady = '0',
    padx = '0',
    pady = '0',
    rowspan = '1',
    sticky = 'nsw'
)
self.labfnode.grid(
    in_     = self.frmegenskaper,
    column = 1,
    row     = 1,
    columnspan = '1',
    ipadx = '0',
    ipady = '0',
    padx = '0',
    pady = '0',
    rowspan = '1',
    sticky = ''
)
self.labtnode.grid(
```

```
    in_     = self.frmegenskaper,
    column  = 1,
    row     = 2,
    columnspan = '1',
    ipadx = '0',
    ipady = '0',
    padx = '0',
    pady = '0',
    rowspan = '1',
    sticky = ''
)
self.labe.grid(
    in_     = self.frmegenskaper,
    column  = 1,
    row     = 3,
    columnspan = '1',
    ipadx = '0',
    ipady = '0',
    padx = '0',
    pady = '0',
    rowspan = '1',
    sticky = ''
)
self.laba.grid(
    in_     = self.frmegenskaper,
    column  = 1,
    row     = 4,
    columnspan = '1',
    ipadx = '0',
    ipady = '0',
    padx = '0',
    pady = '0',
    rowspan = '1',
    sticky = ''
)
self.entfromnode.grid(
    in_     = self.frmegenskaper,
    column  = 2,
    row     = 1,
    columnspan = '1',
    ipadx = '0',
    ipady = '0',
    padx = '0',
    pady = '0',
    rowspan = '1',
    sticky = ''
)
self.enttonode.grid(
    in_     = self.frmegenskaper,
```

```
        column = 2,
        row     = 2,
        columnspan = '1',
        ipadx = '0',
        ipady = '0',
        padx = '0',
        pady = '0',
        rowspan = '1',
        sticky = ''
)
self.entarea.grid(
        in_     = self.frmegenskaper,
        column = 2,
        row     = 4,
        columnspan = '1',
        ipadx = '0',
        ipady = '0',
        padx = '0',
        pady = '0',
        rowspan = '1',
        sticky = ''
)
self.entnrelement.grid(
        in_     = root,
        column = 4,
        row     = 1,
        columnspan = '1',
        ipadx = '0',
        ipady = '0',
        padx = '0',
        pady = '0',
        rowspan = '1',
        sticky = 'w'
)
self.labnrel.grid(
        in_     = root,
        column = 3,
        row     = 1,
        columnspan = '1',
        ipadx = '0',
        ipady = '0',
        padx = '0',
        pady = '0',
        rowspan = '1',
        sticky = 'e'
)
self.btnuppdate.grid(
        in_     = root,
        column = 4,
```

```
        row     = 2,
        columnspan = '1',
        ipadx = '0',
        ipady = '0',
        padx = '0',
        pady = '0',
        rowspan = '1',
        sticky = 'w'
    )
    self.btnSave.grid(
        in_     = self.frmegenskaper,
        column = 2,
        row     = 5,
        columnspan = '1',
        ipadx = '0',
        ipady = '0',
        padx = '0',
        pady = '0',
        rowspan = '1',
        sticky = ''
    )
    self.btnClose.grid(
        in_     = root,
        column = 4,
        row     = 4,
        columnspan = '1',
        ipadx = '0',
        ipady = '0',
        padx = '0',
        pady = '0',
        rowspan = '1',
        sticky = ''
    )
    self.entemodule.grid(
        in_     = self.frmegenskaper,
        column = 2,
        row     = 3,
        columnspan = '1',
        ipadx = '0',
        ipady = '0',
        padx = '0',
        pady = '0',
        rowspan = '1',
        sticky = ''
    )


    # Resize Behavior
```

```
root.grid_rowconfigure(1, weight = 0,
    minsize = 35, pad = 0)
root.grid_rowconfigure(2, weight = 0,
    minsize = 40, pad = 0)
root.grid_rowconfigure(3, weight = 0,
    minsize = 64, pad = 0)
root.grid_rowconfigure(4, weight = 0,
    minsize = 27, pad = 0)
root.grid_columnconfigure(1, weight = 0,
    minsize = 127, pad = 0)
root.grid_columnconfigure(2, weight = 0,
    minsize = 133, pad = 0)
root.grid_columnconfigure(3, weight = 0,
    minsize = 102, pad = 0)
root.grid_columnconfigure(4, weight = 0,
    minsize = 51, pad = 0)
self.labfrm.grid_rowconfigure(1, weight
     = 0, minsize = 217, pad = 0)
self.labfrm.grid_columnconfigure(1, weight
     = 0, minsize = 102, pad = 0)
self.labfrm.grid_columnconfigure(2, weight
     = 0, minsize = 2, pad = 0)
self.frmegenskaper.grid_rowconfigure(1,
    weight = 0, minsize = 40, pad = 0)
self.frmegenskaper.grid_rowconfigure(2,
    weight = 0, minsize = 40, pad = 0)
self.frmegenskaper.grid_rowconfigure(3,
    weight = 0, minsize = 40, pad = 0)
self.frmegenskaper.grid_rowconfigure(4,
    weight = 0, minsize = 31, pad = 0)
self.frmegenskaper.grid_rowconfigure(5,
    weight = 0, minsize = 40, pad = 0)
self.frmegenskaper.grid_columnconfigure(1,
    weight = 0, minsize = 40, pad = 0)
self.frmegenskaper.grid_columnconfigure(2,
    weight = 0, minsize = 40, pad = 0)
```

Appendix B

Fortran Modules

B.1 Calcprog

```fortran
! File calcprog.f90

module calcprog

! Importing sub modules

use barelement
use solve
use outputfile

contains

subroutine execute(nel,neq,nnl,npv,eprop,edof,f,bc,
    b,eforces,outfile,ierr,displace)

     !---- Declaration of exported variables.

     implicit none

     integer, parameter :: fp = 8

     integer :: nel
     integer :: neq
     integer :: nnl
     integer :: npv
     real(kind=8) :: eprop(6,*)
     integer :: edof(4,*)
     real(kind=8) :: f(*)
     real(kind=8) :: bc(*)
     integer :: b(*)
     real(kind=8) :: eforces(2,*)
```

```
character(255) :: outfile
integer :: ierr(1)
real(kind=8) :: displace(*)

! Local declarations

integer :: bandbredd,i,Edofel(4),nlcs
real(kind=8) :: kel(4,4),Epropel(6)
real(kind=8), allocatable :: Kmatrix(:,:)
real(kind=8) :: ftemp(neq)

! Open file to write result to

open(unit=unitoutfile,file=outfile,&
access='sequential', action='write',status='
    unknown')

! Calculates bandwith

bandbredd=BB(Edof,nel)

! Creates stiffness matrix

allocate(Kmatrix(neq,bandbredd))
Kmatrix=0.0_ap
nlcs=1

! Creates element matrices and assembles to
    stiffness matrix

     do i=1,nel
     Epropel=Eprop(:,i)
     Edofel=Edof(:,i)
     call bar2e(Epropel,kel)
     call assem(Kmatrix,kel,Edofel,bandbredd,neq
         )
end do

! Solves the system of linear equations which
    gives displacement and reaction forces

do i=1,neq
     displace(i)=bc(i)
     ftemp(i)=f(i)
end do

call bandsolve(Kmatrix,displace,ftemp,b,neq,
    bandbredd,nlcs,ierr(1))
```

```fortran
    ! Computes elementforces (normal forces)

    do i=1,nel
        Epropel=Eprop(:,i)
        Edofel=Edof(:,i)
        call bar2s(Epropel,Edofel,displace,neq,
            eforces(:,i))
    end do

    !Write displacement and reaction forces to file

    call write3(ftemp,displace,b,neq)

        !Writes element forces to file

        call write4(eforces,nel)

    close(unitoutfile)
    deallocate(Kmatrix)

    return

end subroutine execute

end module calcprog

! End file calcprog.f90
```