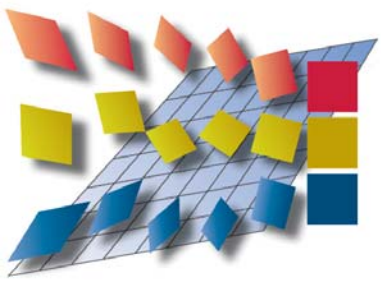
LUND UNIVERSITY



python™

IMPLEMENTATION OF CALFEM FOR PYTHON

ANDREAS OTTOSSON

Structural Mechanics

Master's Dissertation

Department of Construction Sciences
Structural Mechanics

ISRN LUTVDG/TVSM--10/5167--SE (1-47)
ISSN 0281-6679

IMPLEMENTATION OF CALFEM FOR PYTHON

Master's Dissertation by

ANDREAS OTTOSSON

Supervisors:

Jonas Lindemann, PhD,
Div. of Structural Mechanics

Examiner:

Ola Dahlblom, Professor,
Div. of Structural Mechanics

Printed by Wallin & Dalholm Digital AB, Lund, Sweden, August, 2010 *(Pl)*.

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 masters's thesis was carried out during the period June 2009 to August 2010 at the Division of Structural Mechanics at the Faculty of Engineering, Lund University, Sweden.

I would like to thank the staff of the Department of Structural Mechanics, especially my supervisor Jonas Lindemann, for help during this work. I would also like to thank my Jennie, and both our families, for their support throughout my education.

Lund, August 2010

Andreas Ottosson

Contents

Chapter 1

Introduction

1.1 Background

The computer program CALFEM [1] is written for the software MATLAB and is an interactive tool for learning the finite element method. CALFEM is an abbreviation of "Computer Aided Learning of the Finite Element Method" and been developed by the Division of Structural Mechanics at Lund University since the late 70's.

1.2 Why CALFEM for Python?

Unlike MATLAB, which have expensive licenses, Python is free to use and distribute both for personal and commercial use. An implementation to Python will make the functionality of CALFEM available to a larger group of users.

1.3 Objective

The objective of this dissertation is to implement a large part of CALFEM functions to Python and to explore libraries and software in addition to NumPy needed to achieve a comparable environment for CALFEM. No actions will be taken to improve or change existing programming. Comparison of code and description of the main differences when writing in Python is done to simplify the transition for existing users of CALFEM. The author assumes that readers have a basic knowledge of writing code with CALFEM in MATLAB.

Chapter 2

MATLAB

MATLAB is an numerical computing environment from the company The MathWorks. It is a high-level language and interactive environment that according to [18] enables users to perform computationally intensive tasks faster than with programming languages such as C, C++ and Fortran.

2.1 Background

MATLAB is short for Matrix Laboratory and was created in the late 1970s by Cleve Moler [8]. Cleve Moler wrote MATLAB in the program language Fortran and it was used as a tool for his students at University of New Mexico. It became popular and spread to other universities and later on John N Little and Steve Bangert joined Cleve Moler. Together they rewrote MATLAB in the program language C. They also founded The MathWorks in 1984 to continue the development of MATLAB [7]. Today The MathWorks have more than 2 000 employees in offices around the world and over 1 000 000 users in more than 100 countries [8].

2.2 Objects

Object types in MATLAB are scalars, vectors, matrices and multi-dimensional arrays. To create the last three, brackets are used. Multi-dimensional arrays are created by first defining a matrix and then adding the next dimension. Vectors and regular matrices are more often used and they can be created according to:

```
myrowvector = [1 2 3]
mycolumnvector = [1 2 3]'
mycolumnvector = [1; 2; 3]
mymatrix = [1 2 3; 4 5 6]
```

For indexing, MATLAB uses parentheses and is one-based, i.e., mymatrix(1,2) returns the value from the first row and the second column, myrowvector(2) and mycolumnvector(2) returns the second value. In MATLAB vectors are actually two dimensional so myrowvector(1,2) will return the same value as myrowvector(2). MATLAB use pass-by-value semantics so indexing a variable creates a partial copy of it, more about this in the next chapter.

Chapter 3

Python and NumPy

3.1 Python

Python is a dynamic object-oriented scripting language with the aim to combine remarkable power with very clear syntax. Its design philosophy emphasizes code readability [9] which make it easy to learn.

3.1.1 Background

Around 1990 Guido van Rossum created the programming language Python [2]. At the time he worked at Centrum Wiskunde & Informatica (CWI) in Amsterdam and the new programming language was supposed to be an advanced scripting language for the Amoeba operating system which he was currently involved with. The name Python comes from the British comedy series Monty Python's Flying Circus. According to [2] Guido van Rossum was watching reruns of the series when he needed a name for his new language. Different references to Monty Python can often be found in examples and discussions concerning Python programming [2].

3.1.2 Influences

During his time at CWI Guido van Rossum worked with the programming language ABC and therefore it had a major influence on Python. Similarities between the languages are the use of indentation to delimit blocks, the high-level types and parts of the object implementation. C was the second most important influence with identical keywords, such as `break` and `continue`. Other languages also influenced Guido such as Modula-2+ (exception handling, modules, 'self'), Algol-68 and Icon (string slicing) [3].

3.1.3 Objects

As mentioned above Python is an object-oriented programming language. While an object is anything that a variable can refer to (number, list, function, module etc), the term object-oriented is used for programming with class hierarchies, inheritance, polymorphism, and dynamic binding. For more information about these see [4]. The most common object types are int, float and str. Integers (object type int) are whole numbers, floats (object type float) are decimal numbers and strings (object type str) are pieces of text. These can be created according to:

```
myinteger = 5
myinteger = int(5.0)

myfloat = 5.
myfloat = 5.0
myfloat = float(5)

mystring = 'This is a piece of text'
mystring = "This is a piece of text"
mystring = """This is a piece of text"""
```

Three quotations are used when text is spread over several lines. Objects can be gathered within one variable using lists (object type list) or tuples (object type tuple). These are created using brackets for lists and parentheses or no parentheses for tuples. Both types can contain a sequence of arbitrary objects.

```
mylist = [1, 2.0, 'text']

mytuple = (1, 2.0, 'text')
mytuple = 1, 2.0, 'text'
```

According to [4] Python has great functionality for examining and manipulating the values within lists. Tuples on the other hand can be viewed as a constant list, i.e., changes of the content in a tuple are not allowed. Indexing is zero-based which means that mylist[0] and mytuple[0] returns the first value in mylist and mytuple. This differs from MATLAB indexing that is one-based. A nested list (list within a list) mynestedlist[0][1] returns the second value in the first list within the mynestedlist.

3.2 NumPy

NumPy (numeric python) is an extension module for Python and is written mostly in the programming language C. The NumPy module defines the numerical object types array and matrix and basic operations on them [5]. NumPy also include

mathematical operations and functions similar to the module package math. The NumPy module must be imported in order to get access to all the new functions. There are different ways of importing a module but the example below is considered the most convenient.

```
from numpy import *
```

All functions within the numpy module will now be available for the user.

3.2.1 Objects

NumPy introduces two new object types, multidimensional arrays (object type array) and 2-dimensional matrices (object type matrix). An array can be viewed as a type of list according to [4]. The difference is that while a list may contain arbitrary objects an array only consist of one object type. An array can be created using an existing list or assigning it manually.

```
mylist = [1, 2.0, '3']

myarray = array([1, 2, 3])
myarray = asarray(mylist)
```

The variable mylist contains an integer, a float and a string. The first myarray only holds one type, integer. The asarray function controls the input and change non-array input into an array. If the input is an array a copy is not created. In this case where the input consists of different object types they will be changed into one type. If a string is present all the other object will be changed into strings. If there is only integer and float objects they would be changed to float objects. The object type for the values in an array can also be set manually.

```
myarray = asarray(mylist, int)
myarray = asarray(mylist, float)
myarray = asarray(mylist, str)
```

List are more flexible but the benefits of using array are faster computations, less memory demands, and extensive support for mathematical operations on the data [4]. Arrays is intended to be a general-purpose n-dimensional array and can be used for many kind of numerical computing including linear algebra [6]. The matrix type is an subclass of the array class and is intended to facilitate linear algebra computations specifically. How linear algebra operations are used with the two types is shown in the end of this chapter. A matrix can be constructed in a couple of ways. Both matrix() and the shorter mat() can be used.

```
mylist = [[1,2,3],[4,5,6]]

mymatrix = matrix([[1, 2, 3], [4, 5, 6]])
mymatrix = mat([[1, 2, 3], [4, 5, 6]])
mymatrix = mat("1 2 3; 4 5 6")
mymatrix = asmat(mylist)
```

There is another major difference between MATLAB and Python in how matrix operations are handled. MATLAB uses a pass-by-value semantic while NumPy uses a pass-by-reference semantic [6]. This means that most operations in MATLAB make copies of values while Python extensively uses referencing instead. Slice operations in Python are therefor views into an array instead of partial copies of it. This is also true for function arguments. Using NumPy, functions recieve a reference to the argument rather than a copy of its value [11]. A modification of the value within the function will then also be seen outside the function. This is shown using the function assem where K is changed within the function.

MATLAB	Python
`K = assem(edof,K,Ke)`	`assem(edof,K,Ke)`

Making copies can be quite slow and inefficient if the variable contains many values. This is why pass-by-reference is considered more efficient in terms of time and space [11]. Even though still zero-based, indexing an 2-dimensional array and a matrix is more simular to MATLAB than a nested list. mymatrix[0,1] and my2Darray[0,1] returns the value from the first row and the second column in both variables.

3.2.2 Common matrixoperations

Linear algebra operations are not always the same for arrays and matrices. Here follows examples of the most common operations and how they differ. A is a 2-dimensional array and M is a matrix.

Matrix multiply:

Array	Matrix
`dot(A,A)`	`M*M`

Element-wise multiply:

Array	Matrix
`A*A`	`multiply(M,M)`

Inverse of square matrix:

Array	Matrix
`linalg.inv(A)`	`linalg.inv(M)` `M.I`

Transpose:

Array	Matrix
`A.transpose()`	`M.transpose()`
`transpose(A)`	`transpose(M)`
`A.T`	`M.T`

Determinant:

Array	Matrix
`linalg.det(A)`	`linalg.det(M)`

Chapter 4

Integrated Development Environments

Integrated development environments (IDE) are software applications designed to simplify programming and provide facilities to program more efficient. According to [10] an IDE normally consists of an editor, a complier and/or an interpreter (shell), build automation tools and a debugger. The editor is used to write the program code and the shell is used to execute program code. An editor called IDLE is included when installing Python but there are many different editors to choose from [4]. While the shell is used to display the output from programs written in the editor it can also be used as an environment where the user can work interactively with the programming [4]. When Python been installed on a computer the simplest Python shell will open when typing "python" in the command line in a terminal window [4]. In addition to an editor and a shell, an IDE offers helpful tools and they can vary from IDE to IDE. An example is syntax highlighting which is a standard feature on most. Syntax highlighting adds color and different fonts to the code in the editor to improve the readability and visualize the context. This chapter intends to give a brief overview of a couple different IDE, including MATLAB.

4.1 MATLAB

MATLAB has a built in interactive prompt and an editor in a separate window. The editor can hold different files at the same time and it is possible to rearrange them for a good overview and easy comparison, e.g., side-by-side. The way MATLAB is typed according to [7] is the reason why it lacks helpful tools such as code completion, references searches and refactoring. The names for variables and functions have a tendency to be shorter and sometimes cryptic compared to when code completion is available. MATLAB offers users the ability to buy different kinds of toolboxes for all kind of scientific applications. The most common toolbox is the Simulink which is an interactive graphical environment. MATLAB can be installed

and used on Windows, Mac and Linux and there are four licenses categories; academic, student, commercial and government [7]. Nor academic or student licenses are allowed for commercial use and are only offered to accredited institutions and students enrolled in classes. These licenses also cost less than those for commercial and government users. Figure 4.1 shows an image of the main window in MATLAB and its editor.

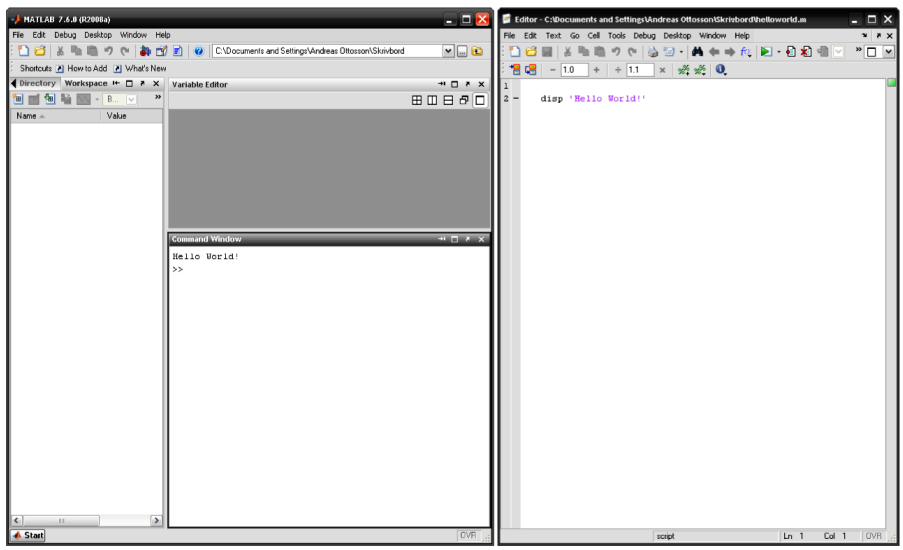


Figure 4.1: MATLAB

4.2 Python IDLE

As mentioned above, Python software comes with an editor called IDLE. It is however not just an editor, but it also contains a shell and basic tools such as syntax highlighting, automatic code completion and smart indent. There are some debugging functions built-in but these are only available in the Python shell window. IDLE is coded in 100% pure Python code, is free to use and works on both Windows and Unix [12]. It is a multi-windowed IDE, i.e., the shell and every program file is viewed in different windows. The learning curve is gentle according to [4] and IDLE can therefore be a suitable IDE for users to start with. IDLE is restricted to only write Python programs. Figure 4.2 shows an image of the IDLE shell and its editor.

4.3 IPython

IPython is an interactive shell superior to the basic Python shell. It adds features like object introspection and system shell access and can be embedded into other

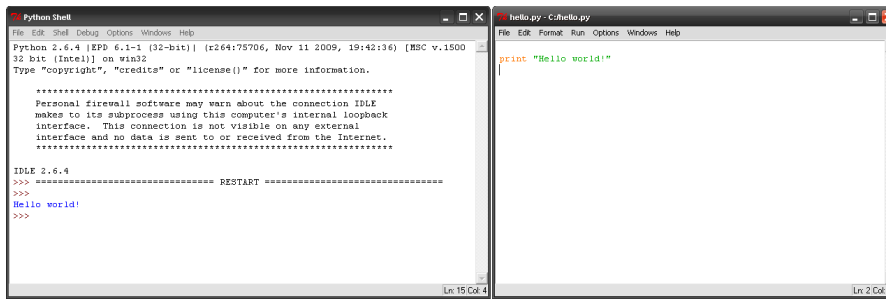


Figure 4.2: IDLE shell and editor

programs as a ready to use interpreter [15]. IPythons goal is according to [14] to create a comprehensive environment for interactive and exploratory computing. This is done with an enhanced interactive Python shell and an architecture for interactive parallel computing. IPython is open source and is therefore free to use. Figure 4.3 shows an image of IPython shell.

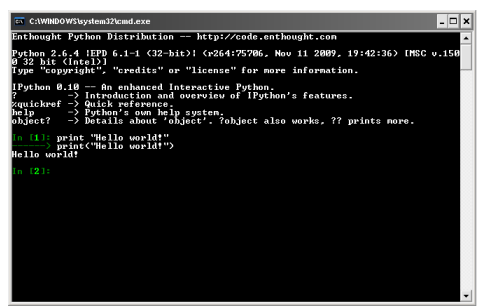


Figure 4.3: IPython shell

4.4 EPD

EPD is an abbreviation of Enthought Python Distribution and is developed by the company Enthought. Enthought support and participate in the maintenance of NumPy as well as hosting IPython almost since its inception. EPD includes a wide range of open source packages and tools for both data analysis and visualization. Their objective is to give users a solid and comprehensive Python environment for scientific computing. EPD can be installed on Windows, Mac, Linux and Unix [13]. Academic use of EPD is free and for individual and commercial use there is, in addition to a 30-day free trial, an annual cost. The cost is divided into different categories, whose difference is the degree of support desired [13]. Bundled packages and features are therefore the same for both highest and lowest annual cost. Figure 4.4 shows an image of the EPD software.

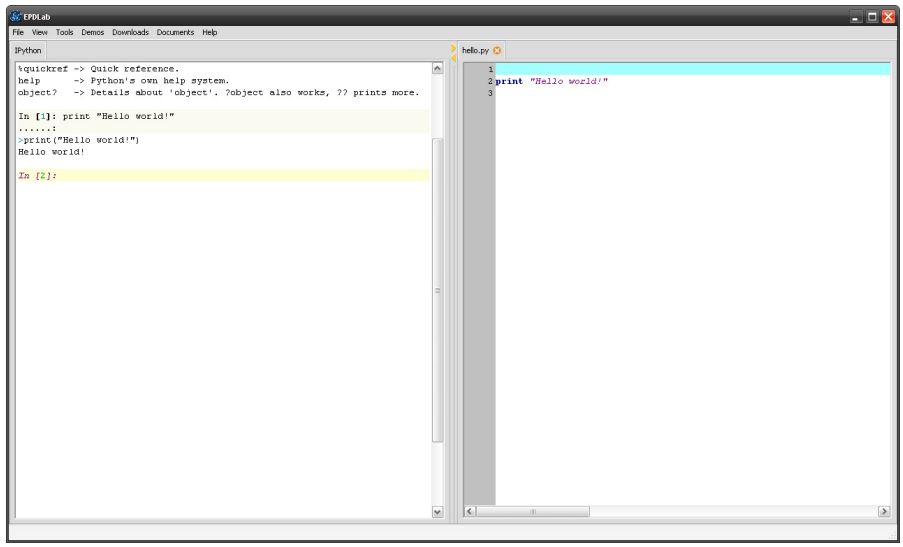


Figure 4.4: EPD

4.5 Spyder

Spyder is a free IDE that visually resembles MATLAB. Its editor and Python shell is built in. In addition to syntax highlighting the editor offers code completion, code analysis, function/class browser and horizontal/vertical splitting features [16]. It also has a MATLAB-like workspace for browsing global variables and a document viewer that automatically show documentation for any function call made in a Python shell [16]. Spyder developers recommended it for scientific computing and it works on Windows and Linux. Figure 4.5 shows an image of the Spyder software.

4.6 ActiveState Komodo IDE

The Komodo IDE from ActiveState is a professional IDE for dynamic languages and open technologies. Multiple programming languages like Perl, Python, Tcl, PHP, Ruby, JavaScript are supported in this IDE [17]. It includes a comprehensive graphical debugger and a whole range of helpful tools, e.g., code completion, code folding, bracket matching, source control integration and much more. It is however a commercial product and a license must be bought. Komodo works on Windows, Mac and Linux. Figure 4.6 shows an image of the Komodo IDE software.

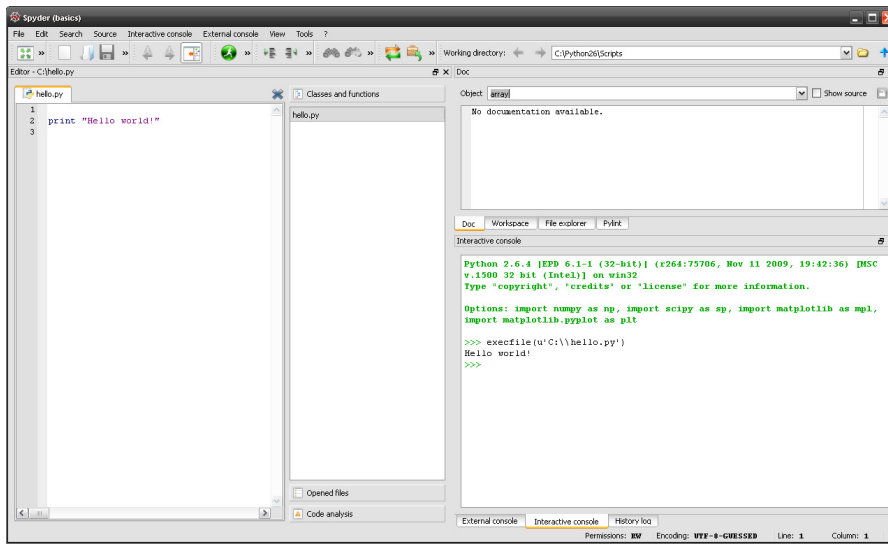


Figure 4.5: Spyder

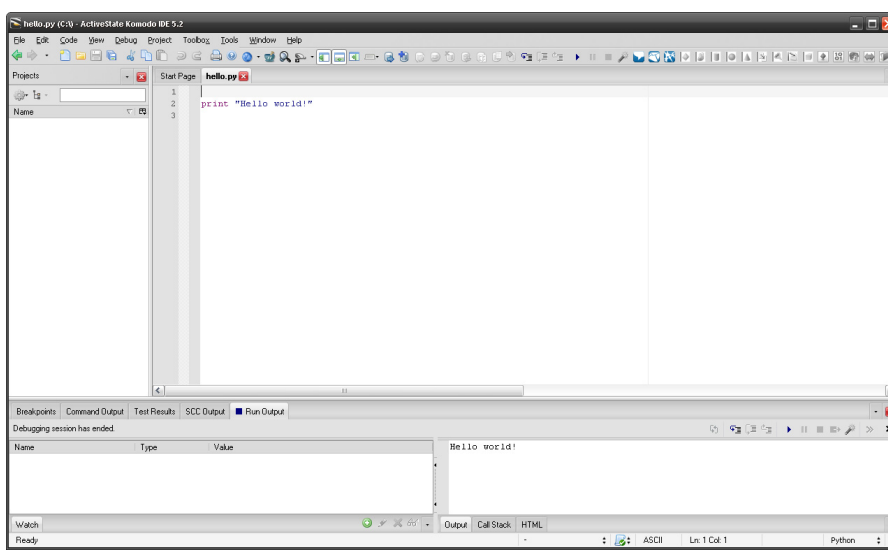


Figure 4.6: Komodo IDE

Chapter 5

Python version of CALFEM

Since CALFEM is written in MATLAB the objective is to recreate CALFEM's functions and calculations with programming language Python. All functions are then tested with different values and the results are compared with the output from MATLAB.

5.1 Common structure of code

Each function in the Python version of CALFEM is written with recurrent similarities. The general structure of an arbitrary function begins with information about the parameters that goes in and comes out of the function. A control of the inputs object type is then done and is changed if necessary. This is followed by the actual calculation code where both arrays and matrices are often used with the benefits of faster calculations. A final check is done to determine the number of outputs and these are based on the number of inputs in accordance to the CALFEM manual. Functions end with returning the outputs to the caller.

5.2 Code build-up

The main structure of the code is generally the same as in MATLAB with only minor differences. MATLAB has some built-in functionality which Python need to manually compensate for. This is done with different controls for determining object types and number of returned values. The information about every function is written as a string using three quotation signs in the beginning and end. The three quotations make it possible to write over several horizontal lines as mentioned in previous chapter. The information consists of a short description of what the function does, syntax, input parameters and returns. This information is a shorter version of what the CALFEM manual contains.

```
"""
Information about this function.

Parameters:
ex = [1, 2, 3]      X coordinates
ey = ...            ...
...
"""
```

This string is displayed when typing help('def'), where 'def' is the function name, in the interactive prompt. With the intension to be user-friendly the Python version of CALFEM supports different types of input such as lists, arrays and matrices. Functions are written for specific object types so the input data need to be, if necessary, changed to the correct type. Many functions also have one or more optional variables as inputs and when left unassigned they will be set to None. Since they are still a part of the following equations they need to be assigned a default value, e.g., if eq is unassigned by the caller it will first be set to eq = None and then get assigned its default value eq = 0. Python is more sensitive than MATLAB when it comes to defining the same number of output variables as the function will return with given input data. This means that the user need to assign the same number of returned values as the function actually returns. The number of returned values depends on the number of inputs given by the caller so it is necessary for functions to control this before returning values. For example, many functions have the element stiffness matrix Ke and element load vector fe as common returns. For these functions the input variable eq is optional and if not assign by the caller, the function only return Ke. If two outputs are called for according to

```
Ke,fe = ...
```

the user will receive an error message due to the function only returns one when eq=None. If only one variable is given and the functions returns more than one value, it will be assign a tuple containing all returned values. As showed, two or more variables will result in an error message if they do not match the number of returned values. The output object type depends on what information it holds. Variables that is defined by all of its values will be returned as a matrix, e.g., Ke and fe. Edof is an example of a variables that instead contain a collection of values and such variables will be returned as a array.

5.3 CALFEM code comparison

To demonstrate the difference between code written in MATLAB and Python we use two examples from the CALFEM manual, exs1 and exs7. See [1] for more information regarding the examples.

5.3.1 Static analysis of a linear spring system, exs1

The example starts with defining the topology matrix Edof. In Python Edof is defined as a 2-dimensional array. A list could also been used but as mentioned earlier array have the benefits of faster computations and extensive support for mathematical operations [4]. The first column in Edof indicating element numbering is not used in Python. This information is instead based on the row number, i.e., the first row contain element one.

MATLAB	Python

`Edof=[1 1 2;`
`2 2 3;`
`3 3 4];`
	`Edof=array([[1,2],`
	`[2,3],`
	`[3,4]])`

Defining the stiffness matrix K and the load vector f are similar using the zero function with the exception of defining the variable type matrix in Python. The input data for zeros is the same but in Python should information about the number of rows and column be collected in a tuple or a list. The extra parentheses turn the two inputs into one, a tuple. When defining a value in f it is important to remember that Python starts counting with 0 as the first number, i.e., the second row is f[1].

MATLAB	Python
`K=zeros(3,3)`	`K=matrix(zeros((3,3)))`
`f=zeros(3,1)`	`f=matrix(zeros((3,1)))`
`f(2)=100`	`f[1]=100`

Element stiffness matrix. The dot used when defining the variable k sets the variable type to float. This is important because multiplication and division using only int variables results in an int answer. The problem becomes clear when dividing 3 with 2 (both int) and getting the answer 1 (int). However, if either 3 or 2 (or both!) is a float the answer also will be an float.

MATLAB	Python
`k=1500;`	`k=1500.`
`ep1=k;`	`ep1=k`
`ep2=2*k;`	`ep2=2*k`
`Ke1=spring1e(ep1);`	`Ke1=spring1e(ep1)`
`Ke2=spring1e(ep2);`	`Ke2=spring1e(ep2)`

Assemble Ke into K. Here we can see that K is not defined as the output to assem. Since K is both an input and the output to assem there is no need to redefine the variable in Python. The function updates the value of K insted of making a copy of it.

MATLAB	Python
K=assem(Edof(1,:),K,Ke2)	assem(Edof[0,:],K,Ke2)
K=assem(Edof(2,:),K,Ke1)	assem(Edof[1,:],K,Ke1)
K=assem(Edof(3,:),K,Ke2)	assem(Edof[2,:],K,Ke2)

Solve the system of equations. The boundary condition variable in Python is divided into two variables, bc and bcVal. These contain the prescribed nodes and their given values separately. For each node in bc there must be a corresponding value in bcVal. If bcVal is not defined by the user, solveq prescribes bcVal to contain zero-values. Again, array are used for faster calculations. The use of brackets is not necessary in Python when functions return more than one value. The number of variables (a, r) must be consistent with the number of return values. If only one variable is given it will be a tuple containing both values from solveq.

MATLAB	Python
bc=[1 0; 3 0];	bc=array([1,3])
[a,r]=solveq(K,f,bc)	a,r=solveq(K,f,bc)

Element forces. For situations where you want to retrieve all the column in a specific row it is sufficient to only specify the desired row, as seen in Edof. Both the syntax and the input are the same for retrieving spring forces.

MATLAB	Python
ed1=extract(Edof(1,:),a)	ed1=extract(Edof[0],a)
ed2=extract(Edof(2,:),a)	ed2=extract(Edof[1],a)
ed3=extract(Edof(3,:),a)	ed3=extract(Edof[2],a)
es1=spring1s(ep2,ed1)	es1=spring1s(ep2,ed1)
es2=spring1s(ep1,ed2)	es2=spring1s(ep1,ed2)
es3=spring1s(ep2,ed3)	es3=spring1s(ep2,ed3)

5.3.2 Static analysis of a plane frame stabilized with bars, exs7

System matrices. Defining the variable type for K in the same manner as in exs1 but in an alternative way. Both mat() and matrix() creates the same variable type. Comments regarding creating and defining a value in f cited to previous example. Coord and Dof are created as array.

MATLAB

```
K=zeros(18,18);
f=zeros(18,1);
f(13)=1;

Coord=[0  0;
       1  0;
       0  1;
       1  1;
       0  2;
       1  2];

Dof=[ 1   2   3;
      4   5   6;
      7   8   9;
     10  11  12;
     13  14  15;
     16  17  18];
```

Python

```
K=mat(zeros((18,18)))
f=mat(zeros((18,1)))
f[12]=1

Coord=array([[0,0],
             [1,0],
             [0,1],
             [1,1],
             [0,2],
             [1,2]])

Dof=array([[ 1, 2, 3],
           [ 4, 5, 6],
           [ 7, 8, 9],
           [10,11,12],
           [13,14,15],
           [16,17,18]])
```

Element properties and topology. ep is defined as a list and it holds element properties. These are going to be used separately in the calculations and not together as a vector, so it makes no difference in calculation time if they are collected in an array or a list. As mentioned in previous example Edof do not contain the element number in the first column, instead row number is used to determine element number.

MATLAB

```
ep1=[1  1  1];
Edof1=[1   1   2   3   7   8   9;
       2   7   8   9  13  14  15;
       3   4   5   6  10  11  12;
       4  10  11  12  16  17  18;
       5   7   8   9  10  11  12;
       6  13  14  15  16  17  18];

ep2=[1  1];
Edof2=[7   1   2  10  11;
       8   7   8  16  17;
       9   7   8   4   5;
      10  13  14  10  11];
```

Python

```
ep1=[1,1,1]
Edof1=array([[ 1, 2, 3, 7, 8, 9],
             [ 7, 8, 9,13,14,15],
             [10,11,12,16,17,18],
             [ 7, 8, 9,10,11,12],
             [13,14,15,16,17,18]])

ep2=[1,1]
Edof2=array([[ 1, 2,10,11],
             [ 7, 8,16,17],
             [13,14,10,11]])
```

Element coordinates. ix_ is an build-in function in NumPy that lets user create new matrices based on specific rows and column in a existing matrix or array. The user can also rearrange the order of rows and column using ix_. Dof2 is created by all the rows and the first two columns in Dof. The reason for this step is that for now coordxtr only returns coordinates for element with the same number of DOF as Dof contain per node (row). Edof2 is the topology matrix for the bar element so the DOF input for coordxtr must be modified by removing the last column containing rotation DOF.

MATLAB

```
[Ex1,Ey1]=coordxtr(Edof1,Coord,Dof,2);

[Ex2,Ey2]=coordxtr(Edof2,Coord,Dof,2);
```

Python

```
Ex1,Ey1=coordxtr(Edof1,Coord,Dof)

Dof2=Dof[ix_(range(6),[0,1])]
Ex2,Ey2=coordxtr(Edof2,Coord,Dof2)
```

Draw the FE-mesh for a visual control of the model. Same syntax with one difference; eldraw2 does not support the variable plotpar where the user can choose e.g. linecolor. This feature might be added in future versions.

MATLAB	Python
`eldraw2(Ex1,Ey1,[1 3 1]);`	`eldraw2(Ex1,Ey1)`
`eldraw2(Ex2,Ey2,[1 2 1]);`	`eldraw2(Ex2,Ey2)`

Create and assemble element matrices. With zip it is possible to loop over several variables simultaneously instead of using a loop counter (i). The three variables (elx, ely, eltopo) will reprecent each row in the three variables within zip during the loop. The number of loops are determined by the variable within zip with the least amount of values i.e. the loop only goes on as long as every variable within zip has a new value per loop. As mentioned in previous example the variable K is updated in assem instead of redefined.

MATLAB

```
for i=1:6
  Ke=beam2e(Ex1(i,:),Ey1(i,:),ep1);
  K=assem(Edof1(i,:),K,Ke);
end

for i=1:4
   Ke=bar2e(Ex2(i,:),Ey2(i,:),ep2);
   K=assem(Edof2(i,:),K,Ke);
end
```

Python

```
for elx,ely,eltopo in zip(Ex1,Ey1,Edof1):
    Ke=beam2e(elx,ely,ep1)
    assem(eltopo,K,Ke);

for elx,ely,eltopo in zip(Ex2,Ey2,Edof2):
    Ke=bar2e(elx,ely,ep2)
    assem(eltopo,K,Ke)
```

Solve equation system. The function arange creates a 1-dimensional array with the first value 1 and the last 6 with steps of one between. Besides the lack of brackets there are no difference in solveq.

MATLAB

```
bc=[1 0;
    2 0;
    3 0;
    4 0;
    5 0;
    6 0];
[a,r]=solveq(K,f,bc);
```

Python

```
bc=arange(1,7)
a,r=solveq(K,f,bc)
```

Extract element displacements and display the deformed mesh. No need to comment Ed variables. The same goes for eldisp2 as for eldraw2 where the variable plotpar is not yet supported.

MATLAB

```
Ed1=extract(Edof1,a);
Ed2=extract(Edof2,a);
```

Python

```
Ed1=extract(Edof1,a)
Ed2=extract(Edof2,a)
```

MATLAB

```
[sfac]=scalfact2(Ex1,Ey1,Ed1,0.1);
eldisp2(Ex1,Ey1,Ed1,[2 1 1],sfac);
eldisp2(Ex2,Ey2,Ed2,[2 1 1],sfac);
```

Python

```
sfac=scalfact2(Ex1,Ey1,Ed1,0.1)
eldisp2(Ex1,Ey1,Ed1,sfac)
eldisp2(Ex2,Ey2,Ed2,sfac)
```

Chapter 6

Conclusions

Python is an easy programming language to learn due to its clear syntax. MATLAB syntax is also very clear so differences in code are small. This makes the transition to Python easy for existing MATLAB users. Linear algebra operations on vectors and matrices are both easy to use and fast to calculate with the library NumPy. Due to NumPy's pass-by-reference semantics fewer copies are created when running programs which reduce both computing time and memory usage. Large parts of CALFEM have been implemented and are now available to Python users. In short, this means that CALFEM can now be used without expensive licenses and in a programming environment of choice. The programming environment for Python varies widely, both visually and functionally. From a simple terminal window, like the basic Python shell or the more advanced IPython, to MATLAB-like environments such as Spyder. Since Python is open source, and therefore also most IDE, user can experiment with different IDE until they find the development environment that suits them. In conclusion, Python is considered to be an adequate alternative to MATLAB, due to only small differences in functionality, optional programming environment and great differences in costs.

Chapter 7

Future work

Even though most functions have been implemented there are still some functions left as can be seen in appendix. Several of these will probably be removed in the next official version of CALFEM so an implementation of these might be unnecessary. To aid users to find error in their code existing functions could be extended with specified error messages. These could for example inform the user if their input has the wrong shape. The functions could also be examined with regards to optimize them by applying pass-by-reference so that unnecessary copies are not created. Calculation time could also be shortened through integrations with high performance libraries and solvers, e.g., Intel's MKL (Math Kernel Library). Meshing functionality should be added from the new CALFEM mesh module. Even though close to identical, the Python syntax should also be added and presented next to the MATLAB syntax for each function in the CALFEM manual.

Appendix A

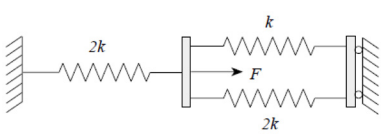
Example exs1

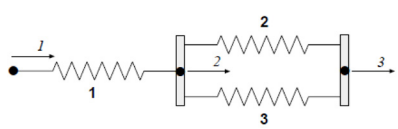
This example is from the CALFEM manual [1].
Purpose:
Show the basic steps in a finite element calculation.

Description:
The general procedure in linear finite element calculations is carried out for a simple structure. The steps are

- define the model
- generate element matrices
- assemble element matrices into the global system of equations
- solve the global system of equations
- evalueate element forces

Consider the system of three linear elastic springs, and the corresponding finite element model. The system of springs is fixed in its ends and loaded by a single load F.





Necessary modules are first imported.

```
>>> from numpy import *
>>> from pycalfem import *
```

The computation is initialized by defining the topology matrix Edof, containing element numbers and global element degrees of freedom,

```
>>> Edof = array([
... [1, 2],
... [2, 3],
... [2, 3]
... ])
```

the global stiffness matrix K (3x3) of zeros,

```
>>> K = matrix(zeros((3, 3)))
>>> print K
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
```

and the load vector f (3x1) with the load $F = 100$ in position 2.

```
>>> f = matrix(zeros((3, 1)));  f[1] = 100
>>> print f
[[   0.]
 [ 100.]
 [   0.]]
```

Element stiffness matrices are generated by the function spring1e. The element property ep for the springs contains the spring stiffnesses k and $2k$ respectively, where $k = 1500$.

```
>>> k = 1500;  ep1 = k;  ep2 = 2*k
>>> Ke1 = spring1e(ep1)
>>> print Ke1
[[ 1500. -1500.]
 [-1500.  1500.]]
>>> Ke2 = spring1e(ep2)
>>> print Ke2
[[ 3000. -3000.]
 [-3000.  3000.]]
```

The element stiffness matrices are assembled into the global stiffness matrix K according to the topology.

```
>>> assem(Edof[0, :], K, Ke2)
matrix([[ 3000., -3000.,     0.],
        [-3000.,  3000.,     0.],
        [    0.,     0.,     0.]])
>>> assem(Edof[1, :], K, Ke1)
matrix([[ 3000., -3000.,     0.],
        [-3000.,  4500., -1500.],
        [    0., -1500.,  1500.]])
>>> assem(Edof[2, :], K, Ke2)
matrix([[ 3000., -3000.,     0.],
        [-3000.,  7500., -4500.],
        [    0., -4500.,  4500.]])
```

The global system of equations is solved considering the boundary conditions given in bc.

```
>>> bc = array([1, 3])
>>> a, r = solveq(K, f, bc)
>>> print a
[[ 0.        ]
 [ 0.01333333]
 [ 0.        ]]
>>> print r
[[-40.]
 [  0.]
 [-60.]]
```

Element forces are evaluated from the element displacements. These are obtained from the global displacements a using the function extract.

```
>>> ed1 = extract(Edof[1, :], a)
>>> print ed1
[ 0.01333333  0.        ]
>>>
>>> ed1 = extract(Edof[0, :], a)
>>> print ed1
[ 0.          0.01333333]
>>> ed2 = extract(Edof[1, :], a)
>>> print ed2
[ 0.01333333  0.        ]
>>> ed3 = extract(Edof[2, :], a)
>>> print ed3
[ 0.01333333  0.        ]
```

The spring forces are evaluated using the function spring1s.

```
>>> es1 = spring1s(ep2, ed1)
```

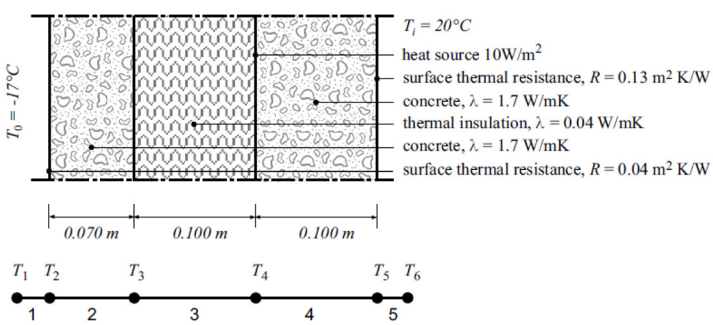
```
>>> print es1
40.0
>>> es2 = spring1s(ep1, ed2)
>>> print es2
-20.0
>>> es3 = spring1s(ep2, ed3)
>>> print es3
-40.0
```

Appendix B

Example exs2

This example is from the CALFEM manual [1].
Purpose:
Analysis of one-dimensional heat flow.

Description:
Consider a wall built up of concrete and thermal insulation. The outdoor temperature is $-17\,^{\circ}\mathrm{C}$ and the temperature inside is $20\,^{\circ}\mathrm{C}$. At the inside of the thermal insulation there is a heat source yielding 10 $\mathrm{W/m^2}$.



The wall is subdivided into five elements and the one-dimensional spring (analogy) element spring1e is used. Equivalent spring stiffnesses are $k_i = \lambda A/L$ for thermal conductivity and $k_i = A/R$ for thermal surface resistance. Corresponding spring stiffnesses per $\mathrm{m^2}$ of the wall are:

$$
\begin{aligned}
k_1 &= 1/0.04 &&= 25.0 \ W/K \\
k_2 &= 1.7/0.070 &&= 24.3 \ W/K \\
k_3 &= 0.040/0.100 &&= \ 0.4 \ W/K \\
k_4 &= 1.7/0.100 &&= 17.0 \ W/K \\
k_5 &= 1/0.13 &&= \ 7.7 \ W/K
\end{aligned}
$$

A global system matrix K and a heat flow vector f are defined. The heat source inside the wall is considered by setting $f_4 = 10$. The element matrices Ke are computed using spring1e, and the function assem assembles the global stiffness matrix.

The system of equations is solved using solveq with considerations to the boundary conditions in bc and bcVal. The prescribed temperatures are $T_1 = -17\,^{\circ}\mathrm{C}$ and $T_2 = 20\,^{\circ}\mathrm{C}$.

Necessary modules are first imported.

```
>>> from numpy import *
>>> from pycalfem import *
>>>
>>> Edof = array([
... [1, 2],
... [2, 3],
... [3, 4],
... [4, 5],
... [5, 6]
... ])
>>>
>>> K = mat(zeros((6, 6)))
>>> f = mat(zeros((6, 1)));  f[3] = 10
>>> print f
[[  0.]
 [  0.]
 [  0.]
 [ 10.]
 [  0.]
 [  0.]]
>>>
>>> ep1 = 25.0;  ep2 = 24.3
>>> ep3 = 0.4;  ep4 = 17.0
>>> ep5 = 7.7
>>>
>>> Ke1 = spring1e(ep1);  Ke2 = spring1e(ep2)
>>> Ke3 = spring1e(ep3);  Ke4 = spring1e(ep4)
>>> Ke5 = spring1e(ep5)
>>>
>>> assem(Edof[0, :], K, Ke1);
matrix([[ 25., -25.,   0.,   0.,   0.,   0.],
        [-25.,  25.,   0.,   0.,   0.,   0.],
        [  0.,   0.,   0.,   0.,   0.,   0.],
        [  0.,   0.,   0.,   0.,   0.,   0.],
        [  0.,   0.,   0.,   0.,   0.,   0.],
        [  0.,   0.,   0.,   0.,   0.,   0.]])
>>> assem(Edof[1], K, Ke2)
```

```
matrix([[ 25. , -25. ,   0. ,   0. ,   0. ,   0. ],
        [-25. ,  49.3, -24.3,   0. ,   0. ,   0. ],
        [  0. , -24.3,  24.3,   0. ,   0. ,   0. ],
        [  0. ,   0. ,   0. ,   0. ,   0. ,   0. ],
        [  0. ,   0. ,   0. ,   0. ,   0. ,   0. ],
        [  0. ,   0. ,   0. ,   0. ,   0. ,   0. ]])
>>> assem(Edof[2, :], K, Ke3);
matrix([[ 25. , -25. ,   0. ,   0. ,   0. ,   0. ],
        [-25. ,  49.3, -24.3,   0. ,   0. ,   0. ],
        [  0. , -24.3,  24.7,  -0.4,   0. ,   0. ],
        [  0. ,   0. ,  -0.4,   0.4,   0. ,   0. ],
        [  0. ,   0. ,   0. ,   0. ,   0. ,   0. ],
        [  0. ,   0. ,   0. ,   0. ,   0. ,   0. ]])
>>> assem(Edof[3], K, Ke4)
matrix([[ 25. , -25. ,   0. ,   0. ,   0. ,   0. ],
        [-25. ,  49.3, -24.3,   0. ,   0. ,   0. ],
        [  0. , -24.3,  24.7,  -0.4,   0. ,   0. ],
        [  0. ,   0. ,  -0.4,  17.4, -17. ,   0. ],
        [  0. ,   0. ,   0. , -17. ,  17. ,   0. ],
        [  0. ,   0. ,   0. ,   0. ,   0. ,   0. ]])
>>> assem(Edof[4, :], K, Ke5)
matrix([[ 25. , -25. ,   0. ,   0. ,   0. ,   0. ],
        [-25. ,  49.3, -24.3,   0. ,   0. ,   0. ],
        [  0. , -24.3,  24.7,  -0.4,   0. ,   0. ],
        [  0. ,   0. ,  -0.4,  17.4, -17. ,   0. ],
        [  0. ,   0. ,   0. , -17. ,  24.7,  -7.7],
        [  0. ,   0. ,   0. ,   0. ,  -7.7,   7.7]])
>>>
>>> bc = array([1, 6]);  bcVal = array([-17.0, 20.0])
>>>
>>> a, r = solveq(K, f, bc, bcVal)
>>> print a
[[-17.        ]
 [-16.43842455]
 [-15.86067203]
 [ 19.23779344]
 [ 19.47540439]
 [ 20.        ]]
>>> print r
[[ -1.40393862e+01]
 [ -5.68434189e-14]
 [ -1.15463195e-14]
 [  0.00000000e+00]
 [  5.68434189e-14]
 [  4.03938619e+00]]
```

The temperature values T_i in the node points are given in the vector a and the boundary flows in the vector r.

After solving the system of equations, the heat flow through the wall is computed using extrac and spring1s.

```
>>> ed1 = extract(Edof[0, :], a)
>>> ed2 = extract(Edof[1, :], a)
>>> ed3 = extract(Edof[2, :], a)
>>> ed4 = extract(Edof[3, :], a)
>>> ed5 = extract(Edof[4, :], a)
>>>
>>> q1 = spring1s(ep1, ed1)
>>> print q1
14.0393861892
>>> q2 = spring1s(ep2, ed2)
>>> print q2
14.0393861892
>>> q3 = spring1s(ep3, ed3)
>>> print q3
14.0393861892
>>> q4 = spring1s(ep4, ed4)
>>> print q4
4.03938618922
>>> q5 = spring1s(ep5, ed5)
>>> print q5
4.03938618922
```

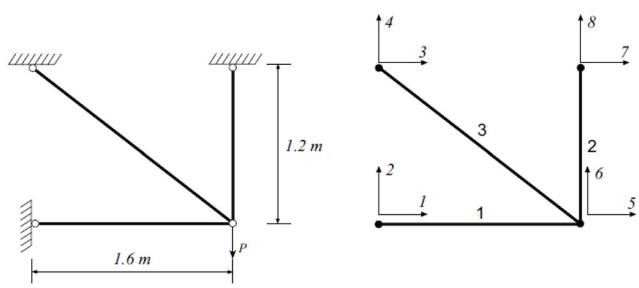
The heat flow through the wall is $q = 14.0$ W/m^2 in the part of the wall to the left of the heat source, and $q = 4.0$ W/m^2 in the part to the right of the heat source.

Appendix C

Example exs3

This example is from the CALFEM manual [1].
Purpose:
Analysis of a plane truss.

Description:
Consider a plane truss consisting of three bars with the properties $E = 200$ GPa, $A_1 = 6.0 \cdot 10^{-4}$ m^2, $A_2 = 3.0 \cdot 10^{-4}$ m^2, $A_3 = 10.0 \cdot 10^{-4}$ m^2, and loaded by a single force $P = 80$ kN. The corresponding finite element model consists of three elements and eight degrees of freedom.



Necessary modules are first imported.

```
>>> from numpy import *
>>> from pycalfem import *
```

The topology is defined by the matrix

```
>>> Edof = array([
... [1, 2, 5, 6],
... [5, 6, 7, 8],
... [3, 4, 5, 6]
```

```
... ])
```

The stiffness matrix K and the load vector f, are defined by

```
>>> K = matrix(zeros((8, 8)))
>>> f = matrix(zeros((8, 1)));  f[5] = -80e3
```

The element property vectors ep1, ep2 and ep3 are defined by

```
>>> E = 2.0e11
>>> A1 = 6.0e-4;  A2 = 3.0e-4;  A3 = 10.0e-4
>>> ep1 = [E, A1];  ep2 = [E, A2];  ep3 = [E, A3]
```

and the element coordinates vectors ex1, ex2, ex3, ey1, ey2 and ey3 by

```
>>> ex1 = [0., 1.6];  ex2 = [1.6, 1.6];  ex3 = [0., 1.6]
>>> ey1 = [0., 0.];  ey2 = [0., 1.2];  ey3 = [1.2, 0.]
```

The element stiffness matrices Ke1, Ke2 and Ke3 are computed using bar2e.

```
>>> Ke1 = bar2e(ex1, ey1, ep1)
>>> print Ke1
[[ 74999999.99999999     0.     -74999999.99999999     0.        ]
 [        0.              0.             0.             0.        ]
 [-74999999.99999999     0.      74999999.99999999     0.        ]
 [        0.              0.             0.             0.        ]]
>>> Ke2 = bar2e(ex2, ey2, ep2)
>>> print Ke2
[[        0.              0.             0.             0.        ]
 [        0.      49999999.99999999     0.     -49999999.99999999]
 [        0.              0.             0.             0.        ]
 [        0.     -49999999.99999999     0.      49999999.99999999]]
>>> Ke3 = bar2e(ex3, ey3, ep3)
>>> print Ke3
[[ 64000000. -48000000. -64000000.  48000000.]
 [-48000000.  36000000.  48000000. -36000000.]
 [-64000000.  48000000.  64000000. -48000000.]
 [ 48000000. -36000000. -48000000.  36000000.]]
```

Based on the topology information, the global stiffness matrix can be generated by assembling the element stiffness matrices

```
>>> K = assem(Edof[0, :], K, Ke1)
>>> K = assem(Edof[1, :], K, Ke2)
>>> K = assem(Edof[2, :], K, Ke3)
>>> print K =
[[  7.50000000e+07,   0.00000000e+00,   0.00000000e+00,
```

```
    0.00000000e+00,  -7.50000000e+07,   0.00000000e+00,
    0.00000000e+00,   0.00000000e+00],
 [  0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
    0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
    0.00000000e+00,   0.00000000e+00],
 [  0.00000000e+00,   0.00000000e+00,   6.40000000e+07,
   -4.80000000e+07,  -6.40000000e+07,   4.80000000e+07,
    0.00000000e+00,   0.00000000e+00],
 [  0.00000000e+00,   0.00000000e+00,  -4.80000000e+07,
    3.60000000e+07,   4.80000000e+07,  -3.60000000e+07,
    0.00000000e+00,   0.00000000e+00],
 [ -7.50000000e+07,   0.00000000e+00,  -6.40000000e+07,
    4.80000000e+07,   1.39000000e+08,  -4.80000000e+07,
    0.00000000e+00,   0.00000000e+00],
 [  0.00000000e+00,   0.00000000e+00,   4.80000000e+07,
   -3.60000000e+07,  -4.80000000e+07,   8.60000000e+07,
    0.00000000e+00,  -5.00000000e+07],
 [  0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
    0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
    0.00000000e+00,   0.00000000e+00],
 [  0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
    0.00000000e+00,   0.00000000e+00,  -5.00000000e+07,
    0.00000000e+00,   5.00000000e+07]])
```

Considering the prescribed displacements in bc, the system of equations is solved using the functions solveq, yielding displacements a and support force r.

```
>>> bc = array([1, 2, 3, 4, 7, 8])
>>> a, r = solveq(K, f, bc)
>>> print a
[[ 0.        ]
 [ 0.        ]
 [ 0.        ]
 [ 0.        ]
 [-0.00039793]
 [-0.00115233]
 [ 0.        ]
 [ 0.        ]]
>>> print r
[[ 29844.55958549]
 [     0.        ]
 [-29844.55958549]
 [ 22383.41968912]
 [     0.        ]
 [     0.        ]
 [     0.        ]
 [ 57616.58031088]]
```

The vertical displacement at the point of loading is 1.15 mm. The section forces es1, es2 and es3 are calculated using bar2s from element displacements ed1, ed2 and ed3 obtained using extract.

```
>>> ed1 = extract(Edof[0, :], a)
>>> N1 = bar2s(ex1, ey1, ep1, ed1)
>>> print N1
-29844.5595855
>>>
>>> ed2 = extract(Edof[1, :], a)
>>> N2 = bar2s(ex2, ey2, ep2, ed2)
>>> print N2
57616.5803109
>>>
>>> ed3 = extract(Edof[2, :], a)
>>> N3 = bar2s(ex3, ey3, ep3, ed3)
>>> print N3
37305.6994819
```

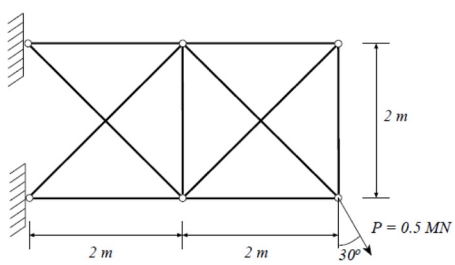
i.e., the normal forces are $N_1 = -29.84$ kN, $N_2 = 57.62$ kN and $N_3 = 37.31$ kN.

Appendix D

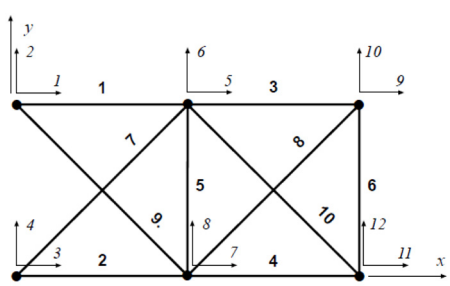
Example exs4

This example is from the CALFEM manual [1].
Purpose:
Analysis of a plane truss.

Description:
Consider a plane truss, loaded by a single force $P = 0.5$ MN.
$A = 25.0 \cdot 10^{-4}$ m^2
$E = 2.10 \cdot 10^5$ MPa



The corresponding finite element model consists of ten elements and twelve degrees of freedom.



Necessary modules are first imported.

```
>>> from numpy import *
>>> from pycalfem import *
```

The topology is defined by the matrix

```
>>> Edof = array([
... [1, 2, 4, 5],
... [3, 4, 7, 8],
... [5, 6, 9, 10],
... [7, 8, 11, 12],
... [11, 12, 9, 10],
... [3, 4, 5, 6],
... [7, 8, 9, 10],
... [1, 2, 7, 8],
... [5, 6, 11, 12]
... ])
```

A global stiffness matrix K and a load vector f are defined. The load P is divided into x and y components and inserted in the load vector f

```
>>> K = zeros([12, 12])
>>> f = zeros([12, 1]);
>>> f[10] = 0.5e6*sin(pi/6);  f[11] = -0.5e6*cos(pi/6)
```

The element matrices Ke are computed by the function bar2e. These matrices are then assembled in the global matrix using the functions assem.

```
>>> A = 25.0e-4;  E = 2.1e11;  ep = [E, A]
>>>
>>> Ex = array([
... [0., 2.],
... [0., 2.],
... [2., 4.],
... [2., 4.],
... [2., 2.],
... [4., 4.],
... [0., 2.],
... [2., 4.],
... [0., 2.],
... [2., 4.]
... ])
>>> Ey = array([
... [2., 2.],
... [0., 0.],
... [2., 2.],
```

```
... [0., 0.],
... [0., 2.],
... [0., 2.],
... [0., 2.],
... [0., 2.],
... [2., 0.],
... [2., 0.]
... ])
```

All the element matrices are computed and assembled in the loop

```
>>> for elx, ely, eltopo in zip(Ex, Ey, Edof):
...     Ke = bar2e(elx, ely, ep)
...     K = assem(eltopo, K, Ke)
```

The displacements in `a` and the support forces in `r` are computed by solving the system of equations considering the boundary conditions in `bc`.

```
>>> bc = array([1, 2, 3, 4])
>>> a, r = solveq(K, f, bc)
>>> print a
[[  0.00000000e+00]
 [  0.00000000e+00]
 [  0.00000000e+00]
 [  0.00000000e+00]
 [ -1.81618691e+12]
 [ -3.46420333e-04]
 [  7.87183401e-05]
 [ -4.77904381e-04]
 [ -1.81618691e+12]
 [  1.81618691e+12]
 [  1.61962931e-04]
 [  1.81618691e+12]]
>>> print r
[[  74095.12857481]
 [  74095.12857481]
 [ -41327.12857481]
 [ 181870.67495855]
 [      0.        ]
 [ -65536.        ]
 [ -19494.53897242]
 [ -65536.        ]
 [  65536.        ]
 [      0.        ]
 [ -53392.        ]
 [ -91275.29810778]]
```

The displacement at the point of loading is $-1.7 \cdot 10^{-3}$ m in the x-direction and $-11.3 \cdot 10^{-3}$ m in the y-direction. At the upper support the horizontal force is -0.866 MN and the vertical 0.240 MN. At the lower support the force are 0.616 MN and 0.193 MN, respectively.

Normal forces are evaluated from element displacements. These are obtained from the global displacements a using the function extract. The normal forces are evaluated using the function bar2s.

```
>>> ed = extract(Edof, a)
>>>
>>> N = zeros([Edof.shape[0]])
>>> i = 0
>>> for elx, ely, eld, in zip(Ex, Ey, ed):
...     N[i] = bar2s(elx, ely, ep, eld)
...     print("N%d = %g" % (i + 1, N[i]))
...     i += 1
...
N1 = 0
N2 = 20663.6
N3 = 0
N4 = 21851.7
N5 = 65536
N6 = -90935.3
N7 = -32768
N8 = -52393.2
N9 = 119562
```

The largest normal force $N = 0.626$ MN is obtained in element 1 and is equivalent to a normal stress $\sigma = 250$ MPa.

To reduce the quantity of input data, the element coordinates matrices Ex and Ey can alternatively be created from a global coordinate matrix Coord and a global topology matrix Dof using the function coordxtr, i.e.

```
>>> Coord = array([
... [0, 2],
... [0, 0],
... [2, 2],
... [2, 0],
... [4, 2],
... [4, 0]
... ])
>>> Dof = array([
... [1, 2],
... [3, 4],
... [5, 6],
... [7, 8],
... [9, 10],
```

```
... [11, 12]
... ])
>>> ex, ey = coordxtr(Edof, Coord, Dof, 2)
```

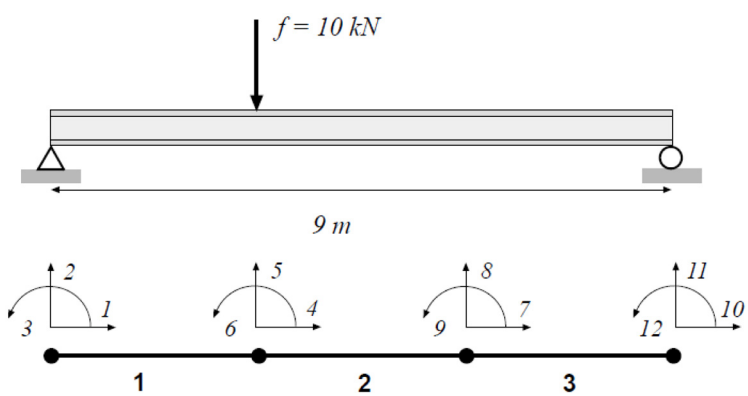
Appendix E

Example exs5

This example is from the CALFEM manual.
Purpose:
Analysis of a simply supported beam.

Description:
Consider the simply supported beam loaded by a single load $f = 10000$ N, applied at a point 1 meter from the left support. The corresponding finite element mesh is also shown. The following data apply to the beam

Young's modulus	E	$=$	$2.10 \cdot 10^{1}$	*Pa*
Cross section area	A	$=$	$45.3 \cdot 10^{-4}$	m^2
Moment of inertia	I	$=$	$2510 \cdot 10^{-8}$	m^4



Necessary modules are first imported.

```
>>> from numpy import *
>>> from pycalfem import *
```

The element topology is defined by the topology matrix

```
>>> Edof = array([
... [1, 2, 3, 4, 5, 6],
... [4, 5, 6, 7, 8, 9],
... [7, 8, 9, 10, 11, 12]
... ])
```

The system matrices, i.e. the stiffness matrix K and the load vector f, are defined by

```
>>> K = mat(zeros((12,12)))
>>> f = mat(zeros((12,1)));  f[4] = -10000
```

The element property vector ep, the element coordinate vectors ex and ey, and the element stiffness matrix Ke, are generated. Note that the same coordinate vectors are applicable for all elements because they are identical.

```
>>> E = 2.1e11;  A = 45.3e-4;  I = 2510e-8;  ep = array([E, A, I])
>>> ex = array([0, 3])
>>> ey = array([0, 0])
>>>
>>> Ke = beam2e(ex, ey, ep)
>>> print Ke
[[  3.17100000e+08   0.00000000e+00   0.00000000e+00
   -3.17100000e+08   0.00000000e+00   0.00000000e+00]
 [  0.00000000e+00   2.34266667e+06   3.51400000e+06
    0.00000000e+00  -2.34266667e+06   3.51400000e+06]
 [  0.00000000e+00   3.51400000e+06   7.02800000e+06
    0.00000000e+00  -3.51400000e+06   3.51400000e+06]
 [ -3.17100000e+08   0.00000000e+00   0.00000000e+00
    3.17100000e+08   0.00000000e+00   0.00000000e+00]
 [  0.00000000e+00  -2.34266667e+06  -3.51400000e+06
    0.00000000e+00   2.34266667e+06  -3.51400000e+06]
 [  0.00000000e+00   3.51400000e+06   3.51400000e+06
    0.00000000e+00  -3.51400000e+06   7.02800000e+06]]
```

Based on the topology information, the global stiffness matrix can be generated by assembling the element stiffness matrices

```
>>> K = assem(Edof, K, Ke)
```

Finally, the solution can be calculated by defining the boundary conditions in bc and solving the system of equations. Displacements a and support forces r are computed by the function solveq.

```
>>> bc = array([1, 2, 11])
>>> a,r = solveq(K,f,bc)
```

The section forces es are calculated from element displacements Ed

```
>>> Ed = extract(Edof, a)
>>> es1, ed1, ec1 = beam2s(ex, ey, ep, Ed[0,:])
>>> es2, ed2, ec2 = beam2s(ex, ey, ep, Ed[1,:])
>>> es3, ed3, ec3 = beam2s(ex, ey, ep, Ed[2,:])
```

Results

```
>>> print "a = ",a
a =  [[ 0.        ]
 [ 0.        ]
 [-0.00948587]
 [ 0.        ]
 [-0.02276608]
 [-0.00379435]
 [ 0.        ]
 [-0.01992032]
 [ 0.00474293]
 [ 0.        ]
 [ 0.        ]
 [ 0.00758869]]
>>> print "r = ", r
r =  [[  0.00000000e+00]
 [  6.66666667e+03]
 [  3.63797881e-12]
 [  0.00000000e+00]
 [  1.45519152e-11]
 [  3.63797881e-12]
 [  0.00000000e+00]
 [ -3.63797881e-12]
 [  0.00000000e+00]
 [  0.00000000e+00]
 [  3.33333333e+03]
 [  7.27595761e-12]]
>>>
>>> print "es1 = ",es1
es1 = [[  0.00000000e+00  -6.66666667e+03   9.14372744e-12]
 [  0.00000000e+00  -6.66666667e+03   2.00000000e+04]]
>>> print "es2 = ",es2
es2 = [[     0.           3333.33333333  20000.        ]
 [     0.           3333.33333333  10000.        ]]
>>> print "es3 = ",es3
es3 = [[  0.00000000e+00   3.33333333e+03   1.00000000e+04]
 [  0.00000000e+00   3.33333333e+03   2.17163527e-11]]
```

Appendix F

Status of existing CALFEM functions

Summary status of existing element functions. Functions replaced by "-" are not yet implemented. Functions with an "*" will most likely be removed from the next release of CALFEM. An implementation has been started for functions with a "†" but they are not completed yet.

Python	MATLAB
spring1e(ep)	spring1e(ep)
spring1s(ep, ed)	spring1s(ep, ed)
bar1e(ep)	bar1e(ep)
bar1s(ep, ed)	bar1s(ep, ed)
bar2e(ex, ey, ep)	bar2e(ex, ey, ep)
bar2g(ex, ey, ep, N)	bar2g(ex, ey, ep, N)
bar2s(ex, ey, ep, ed)	bar2s(ex, ey, ep, ed)
bar3e(ex, ey, ez, ep)	bar3e(ex, ey, ez, ep)
bar3s(ex, ey, ez, ep, ed)	bar3s(ex, ey, ez, ep, ed)
flw2te(ex, ey, ep, D, eq)	flw2te(ex, ey, ep, D, eq)
flw2ts(ex, ey, D, ed)	flw2ts(ex, ey, D, ed)
flw2qe(ex, ey, ep, D, eq)	flw2qe(ex, ey, ep, D, eq)
flw2qs(ex, ey, ep, D, ed, eq)	flw2qs(ex, ey, ep, D, ed, eq)
flw2i4e(ex, ey, ep, D, eq)	flw2i4e(ex, ey, ep, D, eq)
flw2i4s(ex, ey, ep, D, ed)	flw2i4s(ex, ey, ep, D, ed)
flw2i8e(ex, ey, ep, D, eq)	flw2i8(ex, ey, ep, D, eq)
flw2i8s(ex, ey, ep, D, ed)	flw2i8s(ex, ey, ep, D, ed)
flw3i8e(ex, ey, ez, ep, D, eq)	flw3i8e(ex, ey, ez, ep, D, eq)
flw3i8s(ex, ey, ez, ep, D, ed)	flw3i8s(ex, ey, ez, ep, D, ed)
plante(ex, ey, ep, D, eq)	plante(ex, ey, ep, D, eq)
plants(ex, ey, ep, D, ed)	plants(ex, ey, ep, D, ed)
plantf(ex, ey, ep, es)	plantf(ex, ey, ep, es)
-	planqe(ex, ey, ep, D, eq) *

Python	MATLAB
-	planqs(ex, ey, ep, D, ed, eq) *
-	planre(ex, ey, ep, D, eq)
-	planrs(ex, ey, ep, D, ed)
-	plantce(ex, ey, ep, eq) *
-	plantcs(ex, ey, ep, ed) *
-	plani4e(ex, ey, ep, D, eq)
-	plani4s(ex, ey, ep, D, ed)
-	plani4f(ex, ey, ep, es)
-	plani8e(ex, ey, ep, D, eq) *
-	plani8s(ex, ey, ep, D, ed) *
-	plani8f(ex, ey, ep, es) *
-	soli8e(ex, ey, ez, ep, D, eq) *
-	soli8s(ex, ey, ez, ep, D, ed) *
-	soli8f(ex, ey, ez, ep, es) *
beam2e(ex, ey, ep, eq)	beam2e(ex, ey, ep, eq)
beam2s(ex, ey, ep ,ed, eq, np)	beam2s(ex, ey, ep, ed, eq, n)
beam2t(ex, ey, ep, eq)	beam2t(ex, ey, ep, eq)
beam2ts(ex, ey, ep, ed, eq, np)	beam2ts(ex, ey, ep, ed, eq, n)
beam2w(ex, ey, ep, eq)	beam2w(ex, ey, ep, eq)
beam2ws(ex, ey, ep, ed, eq)	beam2ws(ex, ey, ep, ed, eq)
beam2g(ex, ey, ep, N, eq)	beam2g(ex, ey, ep, N, eq)
beam2gs(ex, ey, ep, ed, N, eq)	beam2gs(ex, ey, ep, ed, N, eq)
beam2d(ex, ey, ep)	beam2d(ex, ey, ep)
-	beam2ds(ex, ey, ep, ed, ev, ea)
beam3e(ex, ey, ez, eo, ep, eq)	beam3e(ex, ey, ez, eo, ep, eq)
beam3s(ex, ey, ez, eo, ep, ed, eq, n)	beam3s(ex, ey, ez, eo, ep, ed, eq, n)
platre(ex, ey, ep, D, eq)	platre(ex, ey, ep, D, eq)
-	platrs(ex, ey, ep, D, ed)
-	red(A, b)
hooke(ptype, E, v)	hooke(ptype, E, v)
-	mises(ptype, mp, est, st)
-	dmises(ptype, mp, es, st)
assem(edof, K, Ke, f, fe)	assem(edof, K, Ke, f, fe)
coordxtr(edof, coords, dofs)	cordxtr(Edof, Coord, Dof, nen)
-	eigen(K, M, b)
extract(edof, a)	extract(edof, a)
-	insert(edof, f, ef)
solveq(K, f, bcPresc, bcVal)	solveq(K, f, bc)
statcon(K, f, cd)	statcon(K, f, b)
-	dyna2(w2, xi, f, g, dt)
-	dyna2f(w2, xi, f, p, dt)
-	freqresp(D, dt)
-	gfunc(G, dt)
-	ritz(K, M, f, m, b)
-	spectra(a, xi, dt, f)
-	step1(K, C, d0, ip, f, pbound)
-	step2(K, C, d0, v0, ip, f, pdisp)

Python	MATLAB
-	sweep(K, C, M, p, w)
-	eldia2(ex, ey, es, plotpar, sfac, eci)
eldisp2(ex, ey, ed, magnfac, showmesh) †	eldisp2(Ex, Ey, Ed, plotpar, sfac)
eldraw2(ex, ey) †	eldraw2(Ex, Ey, plotpar, elnum)
-	elflux2(Ex, Ey, Es, plotpar, sfac)
eliso2(ex, ey, ed, showmesh) †	eliso2(Ex, Ey, Ed, isov, plotpar)
-	elprinc2(Ex, Ey, Es, plotpar, sfac)
-	pltscalb2(sfac, magnitude, plotpar)
scalfact2(ex, ey, ed, rat) †	scalfact2(ex, ey, ed, rat)

Bibliography

[1] P-E Austrell, O Dahlblom, J Lindemann, A Olsson, K-G Olsson, K Persson, H Petersson, M Ristinmaa, G Sandberg, P-A Wernberg
CALFEM - A Finite Element Toolbox version 3.4, Lund University, The Division of Structural Mechanics
2004

[2] Mark Lutz, Programming Python, Second Edition,
OReilly & Associates Inc,
2001

[3] Interview with Guido van Rossum,
http://www.amk.ca/python/writing/gvr-interview,
2010-05-03

[4] Hans Petter Langtangen, Introduction to Computer Programming,
University of Oslo,
2008

[5] Getting Started,
http://new.scipy.org/getting-started.html,
2010-05-17

[6] NumPy for MATLAB Users,
http://www.scipy.org/NumPy_for_Matlab_Users,
2010-05-18

[7] MATLAB,
http://en.wikipedia.org/wiki/Matlab,
2010-05-24

[8] About The MathWorks AB,
http://www.mathworks.se/company/aboutus/,
2010-05-24

[9] Python (programming langugage),
http://en.wikipedia.org/wiki/Python_(programming_language),
2010-05-25

[10] Integrated development environment,
http://en.wikipedia.org/wiki/Integrated_development_environment,
2010-05-27

[11] Evaluation strategy,
http://en.wikipedia.org/wiki/Evaluation_strategy,
2010-05-28

[12] IDLE,
http://docs.python.org/library/idle.html,
2010-05-29

[13] Enthought Python Distribution,
http://www.enthought.com/products/epd.php,
2010-02-17

[14] IPython frontpage,
http://ipython.scipy.org/moin/,
2010-05-29

[15] IPython,
http://wiki.python.org/moin/IPython,
2010-05-30

[16] Spyder documentation,
http://packages.python.org/spyder/,
2010-05-30

[17] ActiveState Komodo IDE,
http://www.activestate.com/komodo-ide,
2010-05-30

[18] MATLAB - The Language Of Technical Computing,
http://www.mathworks.com/products/matlab/,
2010-05-31