# A Python interface to Diffpack-based classes and solvers

**by**

**Heidi Vikki Munthe-Kaas**

***Thesis***

*for the degree of*

***Master of Science***

*(Master i Anvendt matematikk og mekanikk)*


*Faculty of Mathematics and Natural Sciences*
*University of Oslo*

*September 2013*

*Det matematisk- naturvitenskapelige fakultet*
*Universitetet i Oslo*

# Preface

This report is written for my Master of Science degree at the Department of Mathematics, University of Oslo. The thesis was written at Simula Research Laboratory with Joakim Sundnes as my supervisor. The work done in this thesis is based on work done by Westlie [15], Thorsen [14], chapter 8 in Langtangen and Tveito [8], and the SWIG documentation ([12]).

All files, folders and scripts described in this thesis can be found in the PulseWrap directory at

`http://folk.uio.no/heidivm/PulseWrap.tar.gz`

To make use of the PulseWrap directory, it is expected that the following programs are installed (versions used in this thesis in parenthesis): SWIG (version: 2.0.9), Python (version: 2.6.8), Diffpack (version: 4.2.00), Gcc (version: 4.7.2) and Pulse (version: 0.9.0).

This report constitutes the end of a very long year filled with both joy and frustration. Joy when things finally were moving along, and problems worked on for a long time suddenly were fixed. Frustration when things that *should* work didn't, and small problems made hindrances for the rest of the work.

I would like to thank my supervisor Joakim Sundnes for giving me this incredibly interesting project, being supportive and helping me with pointers and guidance when I have needed it.

I would like to send a very big "Thank you" to everyone who have beared[1] with me for the past months when I have been frustrated beyond belief, or had low blood sugar and were cranky. There have been ups and downs, and you have been there for me. Those owed a particular thank you are my friends Ingvild Skartveit, Simen Tennøe and Aila Aspås, who have helped me through rough patches, and all of the people at Realistforeningen for being supportive and awesome!

I also want to especially say thank you to Eirik B. Sundmark, who have been the best boyfriend anyone could want, and who have supported me and listened to me during the whole period.

[1] figure 1 on page iv

Figure 1: Artist: Jo Tryti

## Abstract

Python is a programming language that has gained a lot of popularity during the last 15 years, and as a very easy-to-learn and flexible scripting language it is very well suited for computational science, both in mathematics and in physics. Diffpack is a PDE library written in C++, made for easier implementation of both smaller PDE solvers and for larger libraries of simulators. It contains large class hierarchies for different solvers, grids, arrays, parallel computing and almost everything needed to solve PDEs. Pulse, a Diffpack based PDE heart simulator, is made for simulating electrical and mechanical behavior in the heart.

Making an interface to Pulse in Python will hopefully merge the positive qualities from both C++ and Python. Taking advantage of the fact that the original code is written in C++ and therefore effective, and the flexibility of Python, can make for both a highly effective solver, and one it is easy to test and debug.

SWIG was used to wrap the original C++ code into Python, as it had been tested with success on Diffpack classes in an earlier thesis. This thesis consist of two parts. The first explains how to wrap the Heat1 class into a Python module, and how to implement the `timeLoop` and `solveAtThisTimestep` functions in Python, focusing on what problems may occur during the procedure. The second part explains the wrapping procedure used when wrapping Pulse, and how to implement Python versions of the `Circulation::timeLoop`, `Heart::simpleTimeLoop`, `Heart::solveCellsAndDiffusion` and `Heart::initTimeLoop` functions.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Python for computational science

Scripting languages are high level programming languages that are interpreted (not compiled) on the fly. They are often used for making small programs ("scripts") that can automate tasks/-computations. They can be very useful when it comes to "gluing" different programs together, and providing a program with an easy-to-use user interface, as well as being easy to debug and fast to learn. The programmer also doesn't have to worry about memory usage or anything else going on underneath, but concentrating on the task at hand. Scripting languages like Python have become very popular thanks to their flexibility and ease of use. When the user doesn't have to decide what type a variable should have, the program gains a kind of flexibility that it is harder to reproduce with more traditional statically typed languages. One function can then take one argument, but depending on the type of input, perform a variety of different tasks. The function could also have 30 different arguments, but some (or all) with default arguments, which would mean that the user could still use the function without having to set every variable if it wasn't needed. Example:

```
def plot(func, xaxis, yaxis, xlabel='x', ylabel='y', label="Plot")
```

Many people with background in mathematics are often used to programming in Matlab or Maple, but when it comes to scientific computing in general, Python is a more powerful tool. It is first of all free of cost, the environment is open and usable on a lot of different platforms, and you can contain a module with lots of functions in a single file (in contrast to Matlabs huge amount of M files). Python is also object oriented, and includes a lot of fast working libraries (many programmed in C/C++).

## 1.2 Wrapping of compiled code using SWIG

The *Simplified Wrapper and Interface Generator* program (SWIG)

> [1] [...] is a software development tool that simplifies the task of interfacing different languages to C and C++ programs. In a nutshell, SWIG is a compiler that takes

[1] From page 7 in [12]

> C/C++ declarations and creates the wrappers needed to access those declarations from other languages including Perl, Python, Tcl, Ruby, Guile, and Java.

Wrapping code written in C and C++ can be a very difficult and time consuming task, but it has a lot of advantages. While a lot of scripting languages like Python may be easier to use, makes the programs smaller, easier to read and also easier to debug, the problem with languages like this is that they are much slower than e.g. C++. Very often there are certain parts of a program that would be beneficial to have implemented in C++ (the bottlenecks in the program), or there exist a library (or a program) already written in C++ that you would like to use, but your main program would benefit from being programmed in Python. The solution to this is to wrap the parts of your program written in C++ (or the function/class that you would like to use from another library) into Python usable code. This is where SWIG comes in. It makes the wrapping procedure much easier, and does most of the work for you. For small programs/short snippets of code, it takes almost no time to do. For larger libraries or programs, like Diffpack and Pulse, the process may take som time, and is no trivial task, but the final result makes it worth it.

Using SWIG gives a programmer the option of combining the good qualities from both high level languages like Python with the ones from compiled languages like C++. Combining flexibility, easy debugging and good user interfaces with high performance and number crunching can suddenly be achieved, and you don't have to be an expert programmer to accomplish it.

SWIG can generate most of the interface needed to wrap C++ code directly from the C++ code, and the interfacefile generated can be adjusted and added to by the user when needed. There are a few things regarding statically typed languages like C++ that makes them more difficult to wrap into usable Python code, but they can often be solved by using functionality already implemented in SWIG. Problems often arise because C++ declarations are used directly when wrapped, which can cause conflicts with already existing keywords of functions in Python (or whichever scripting language you want to wrap the code to). The solution is to use the `%rename` directive[2], and give the functions or declarations new names.

There exists of course several other programs that can be used for wrapping C++ into Python (Boost.Python[1] and PyCXX[10]), but they were tried and found wanting in [15] compared to SWIG, and since the work done in [15] and [14] were already accomplished with SWIG it would make the work in this thesis easier to build on their work and continue with SWIG.

## 1.3 Diffpack

Diffpack[4] is a software library written in C++ for solving partial differential equations. It is quite extensive, and is often used for developing new simulators, as it contains a lot of the functionality needed when programming PDEs.

> [3][...] the library contains class hierarchies for arrays, linear systems, linear system solvers and preconditioners, grids and corresponding fields for finite difference, element, and volume methods, as well as utilities for data storage, adaptivity, multilevel methods, parallel computing, etc.

[2] See page 39 in [12]
[3] from page 223 [7]

An aspect of Diffpack is that it is written in C++, a compiled language. It has the advantage that it is object oriented, and is one of the best languages to program with when you need your code to be fast, but the code needs to be recompiled whenever you make a change (which can be time consuming), and if (*when*) you get a bug, it may be difficult to locate and fix. Because of the memory management in C++, new users may have a difficult time just getting a simple program to work without memory leaks[4]. When programming PDEs, the time element is very important, as they can be very time consuming to compute. Programming Diffpack in C++ makes therefore much sense, but may at the same time make it more difficult to use for mathematicians and physicists for whom Python and other scripting languages are more natural to program with.

Using Python to make an interface between the Diffpack based code and the users will make both testing the program and using it easier. There already exist a couple of Master thesis' that have wrapped Diffpack code into scripting languages, one with Python ([15]) and one with Ruby ([14]). The work in this thesis will be based on some of the work that were done in them.

## 1.4 Goal and procedure for the project

### 1.4.1 Heart modelling and Pulse

Pulse is a Diffpack-based heart simulator (written at Simula Research Laboratory), containing a large number of classes simulating electrical and mechanical behavior in the heart. Many classes are called in succession, and a Python interface to each class will give a very flexible system that are easy to use.

Simulating the human body is a very complex and difficult task, of which simulating the heart and the brain may be the most difficult because of their complexity. Describing the processes both on a cellular level and on a larger scale, and then having a model that are both complex enough to give a detailed picture, as well as being simple enough to be computable, is a task that aren't easily accomplished.

Pulse makes use of the large and flexible PDE solver library Diffpack, which gives easy access to a range of different methods for PDE solvers, both finite difference, finite element and finite volume methods.

### 1.4.2 Outline of thesis

Python has in the past (as seen in [15]) been proven to be a very flexible programming language for wrapping code written in more traditional compiled programming languages (in this case, C++). Based on the master thesis written by Magne Westlie on construction of a Python interface for the Diffpack library ([15]), the goal for this master thesis was to wrap the entire Pulse library into Python usable code, and equip the simulator with an interface written in Python that does all the communication between the C++ classes from the Python layer. The project plan was to do this in 3 steps:

[4] Segmentation faults, caused when a program tries to access an invalid section of memory, makes the program halt ungracefully, and may not be easy to fix.

1. Wrap a simple Diffpack based application into Python and implement a timeLoop function in Python (to be used in a testfile), to outline the potential technical difficulties.

2. Wrap the entire Pulse library, and make testscripts that compare the results from the original C++ based Pulse code with the Python implemented test cases.

3. Design an interface to Pulse where the communication between the relevant C++ classes are done through the Python layer.

# Chapter 2

# Python interface for a simple simulator, Heat1

## 2.1 Wrapping with Swig

### 2.1.1 First attempt at wrapping Heat1

When starting working on the thesis, it was clear that in order to be able to wrap a large simulator (such as Pulse), it would be useful to know which problems might arise during the work, and how to solve them. It is conceptually easy to wrap small C++ programs with SWIG, but because of heavy integration with Diffpack (which is a large and complicated library), it can become quite difficult to wrap even a small solver class. With that in mind, the first part of the work was to be able to successfully wrap (and use) a smaller example-solver also based on Diffpack. To that end, the diffusion class Heat1 (which can be found in the standard Diffpack library in the directory $NOR/doc/Book/src/fem/Heat1)[1] was used.

All files in the directory (including the hidden files) were copied into a local directory, so as not to make any changes in the original directory by mistake. When wrapping Heat1, the procedure described in chapter 8 in [8] was used, heavily relying on the two scripts `MkDpSWIGMakefile` and `MkDpSWIGInterface`. The `MkDpSWIGMakefile` script creates a Makefile that is used to compile the wrapper code and links it correctly with the Python and Diffpack libraries. The `MkDpSWIGInterface` script generates an interface file Heat1.i which is used for more easy wrapping, and which can be adjusted afterwards if needed. First test, when residing in the Heat1 directory, can be seen in figure 2.1 on the next page.

I first compiled the original C++ program, giving the Makefile the option `MODE` (which can be either "opt" or "nopt")[2]. I then made a new directory in Heat1 (for the wrapper code) called swig, and moved into it, before running the `MkDpSWIGMakefile` and `MkDpSWIGInterface` scripts.

When using the `MkDpSWIGMakefile` script, the option DPDIR is the directory where the Diffpack code and compiled program resides, in this case the parent directory. The option `MODE` tells the compiler which MODE was used when compiling the original Diffpack-program ("opt" or "nopt"). The option `CLASSES` in `MkDpSWIGInterface` tells the script which classes are

[1] $NOR is an environment variable set in .bashrc when you install Diffpack, and is the path to /.../Diffpack/NO

[2] Make must be used with a capital M, since it invokes a special Diffpack make command.

```
~/Heat1$ Make MODE=opt
~/Heat1$ mkdir swig
~/Heat1$ cd swig
~/Heat1/swig$ MkDpSWIGMakefile MODULE=Heat1 DPDIR=.. MODE=opt
Generated DpMakefile.defs and Makefile
~/Heat1/swig$ MkDpSWIGInterface MODULE=Heat1 CLASSES=Heat1 FILES=Heat1.h
Adding Handle_GridFE to interface!
Adding Handle_DegFreeFE to interface!
Adding Handle_FieldFE to interface!
Adding Handle_FieldFE to interface!
Adding Handle_LinEqAdmFE to interface!
Adding Handle_TimePrm to interface!
Adding Handle_FieldsFE to interface!
Adding Handle_SaveSimRes to interface!
Adding Handle_FieldFunc to interface!
Adding Handle_FieldFE to interface!
SWIG interface file written to Heat1.i
~/Heat1/swig$ Make
```

Figure 2.1: The first compilation of Heat1

in the Heat1 files. Usage: `CLASSES="ClassA ClassB ClassC ..."`. When you compile the programs, you need to be careful when using the `MODE` option, so that you give the same option to both the C++ Makefile and the wrapper code. If you use `MODE=opt` for the compilation of the original program, and use `MODE=nopt` for the wrapper code, you will not get any problems right away, but when trying to import the module, you might get an error saying that there are undefined symbols/functions[3]. To see what the undefined symbol actually looks like, you can use the command `c++filt hasClassType__C8HandleId` with the result (in this case) `HandleId::hasClassType(void) const`. The error appears because the program has linked with the wrong library version when the MODE option was mixed up. Of course, you may get that error without having done anything wrong, which then happens because there are functions defined in the header file that haven't been implemented in the C++ file. The easy way to fix this is to simply comment out the unimplemented definitions in the header file and recompile the program. When you have edited either the original code, or the interface file, you will need to recompile both of them. In the `swig` directory, there exist a `configure.sh` script that is used to run the `MkDpSWIGMakefile` script again with the options you gave when you originally ran the command. This can be used both when you have made adjustments, and if you want to move your code to another directory or computer.

The result after running the commands in figure 2.1 was an error message, see figure 2.2 on the next page. When I was unable to locate the checkDpMode.py script, I commented out that part in the Makefile, and recompiled. I then got another error message instead (see figure 2.3 on the facing page), which was resolved when I removed the -c option. Yet another error message occurred, after recompilation, see figure 2.4 on the next page.

When I was unable to find any swigpy library (and it later turned out that it was from an old version of SWIG[4]), I removed the part of the Makefile that used the swigpy library. To be able

[3] see page 334 in [8]

[4] from page 189 in [11] "[...]swigpy is a special purpose library that contains the SWIG pointer type checker and other support code"

```
  ~/Heat1/swig $ Make
2 checkDpMode.py MODE= DPDIR=..
  make: checkDpMode.py: Command not found
4 make: *** [python] Error 127
```

Figure 2.2: Errormessage nr.1 from Makefile when wrapping Heat1

```
  swig -python -c++ -c -shadow -I. -I..
2 ...
  swig error : Unrecognized option -c
4 Use 'swig -help' for available options.
  make[1]: *** [python\_cpp] Error 1
```

Figure 2.3: The second error message from Makefile when wrapping Heat1

to make the compilation process easier, without having to comment out or remove the same lines or options for every recompilation, I made a script (called `fixSwigMakefile.py`, residing in `$PULSEWRAP/src`) which does the necessary augmentations in the Makefile[5]. After adjusting the Makefile, the program compiled (although a lot of warnings were generated, see figure 2.5 on the next page), but using the program would not work properly before the issues in the warnings were solved.

### 2.1.2 Warnings

Of the warnings generated by the compiler (as seen in figure 2.5 on the following page), the 362 and the 509 warning are the easiest to fix. Looking at page 78 in [12], it is apparent what the problem causing the 509-error is. When you have several functions in C++ with the same name, but with different inputs, it causes a problem when wrapping it into Python. With statically typed languages, you have to make several different versions of the same function if you want it to be able to take an argument with varying types. Since Python isn't a statically typed language, you cannot have several functions with the same name. In this case, one of the warnings is caused by the `Handle_GridFE::Handle_GridFE()` function. There are two equally named functions, one taking a `GridFE *` object, the other taking a `GridFE const &` object. This is of course perfectly legal in C++, but in a scripting language such as Python this causes problems, since in Python there would only excist one type of GridFE object, and it can't differentiate between the two. Since SWIG doesn't know how to handle it, it causes a warning. The way to solve it is by giving one of the functions a new name to distinguish it from the other, which means that for each of the 509-warnings given, I needed to insert a di-

[5] One more adjustment was done in fixSwigMakefile.py later on, and are described in section A.3 on page 53

```
1  /usr/bin/ld: cannot find -lswigpy
  collect2: ld returned 1 exit status
3 make[1]: *** [python\_cpp] Error 1
```

Figure 2.4: The third error message from Makefile when wrapping Heat1

```
Heat1.i:89: Warning 362: operator= ignored
Heat1.i:90: Warning 362: operator= ignored
Heat1.i:91: Warning 362: operator= ignored
Heat1.i:211: Warning 302: Identifier 'Handle_FieldFE' redefined (ignored),
Heat1.i:165: Warning 302: previous definition of 'Handle_FieldFE'.
Heat1.i:17: Warning 401: Nothing known about base  class 'FEM'. Ignored.
Heat1.i:81: Warning 509: Overloaded method Handle_GridFE::Handle_GridFE(GridFE *)
    effectively ignored,
Heat1.i:80: Warning 509: as it is shadowed by Handle_GridFE::Handle_GridFE(GridFE
    const &).
```

Figure 2.5: Examples of warnings from compilation of the Heat1 interface

```
%rename(H_GFE__assign__) Handle_GridFE::operator = (const Handle_GridFE& );
%rename(H_GFE__assign__And) Handle_GridFE::operator = (const GridFE& );
%rename(H_GFE__assign__Star) Handle_GridFE::operator = (const GridFE* );
```

Figure 2.6: Example of renaming operator functions

rective in the Heat1.i interface file called `%rename`. Example: `%rename(H_GFE_H_GFE) Handle_GridFE::Handle_GridFE(GridFE *);` As long as the names given are distinguishable from each other, it doesn't really matter which names you give, but it will help for later usage of the program to give names that explains which function it is. For instance the Handle_GridFE could also be renamed `%renameHandle_GridFE_Handle_GridFE_Star`

The 362-warning is handled in much the same way as 509-warnings. It appears when there are operator functions in the program, e.g. `operator +`, `operator []` etc. The operator functions are handled by SWIG just like any other function, but the problem here is that the name they get in scripting languages are strings like `"operator +"`, which is an illegal identifier. The solution is the same as with the 509 warnings, simply renaming the functions[6]. In the Heat1 class (as in many other Diffpack classes), there may be more than one `operator =` function, which means that each of them will need to be renamed to have an unique identifier. For naming conventions regarding other `operator` functions, see [9].

The 401-warning is issued because the Heat1-class inherits the FEM-class from Diffpack, but SWIG hasn't been given any information about it. This is solved by including the FEM interface file in the Heat1 interface by adding this line to Heat1.i: `%include "FEM.i"`[7]. Since SWIG must see the information about the base classes before they are actually inherited, it is good custom to add the `%include` line before any class documentation. As an example, see figure 2.8 on page 10. Of course, to be able to include `FEM.i`, I would need to create an interface for the FEM class first. But I will come back to that process in section 2.1.3 on page 10.

It is important to not ignore the warnings issued by SWIG, because they *will* cause problems later on. If you haven't included FEM.i in Heat1.i (or have ignored warnings), you will get problems when trying to implement a timeLoop function in Python (see section 2.2.1 on page 20) later on. The 401 warning, if ignored, will see to it that you don't have access to the

[6] As seen in figure 2.6

[7] For explanation of how SWIG will find the correct interface file, see figure A.3 on page 53

```
#Example without %include FEM.i
>>> from Heat1 import *
-------------------------------------------------------------------------------
***    Diffpack Version 4.2.00 - Development Edition (internal use only)     ***
-------------------------------------------------------------------------------
>>> h=Heat1()
>>> h.u
<Swig Object of type 'Handle_FieldFE *' at 0x7f6c9d4a5b10>
>>> dir(h.u)
['__class__', '__cmp__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__
    ', '__getattribute__', '__gt__', '__hash__', '__hex__', '__init__', '__int__',
    '__le__', '__long__', '__lt__', '__ne__', '__new__', '__oct__', '__reduce__', '
    __reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '
    __subclasshook__', 'acquire', 'append', 'disown', 'next', 'own']

#Example with %include FEM.i
>>> from Heat1 import *
-------------------------------------------------------------------------------
***    Diffpack Version 4.2.00 - Development Edition (internal use only)     ***
-------------------------------------------------------------------------------
>>> h = Heat1()
>>> h.u
<Heat1.Handle_FieldFE; proxy of <Swig Object of type 'Handle_FieldFE *' at 0
    x7f0f233e85a0> >
>>> dir(h.u)
['FFEAnd__assign__', 'FFEStar__assign__', 'FFE_rebind', 'HFFE__assign__', '
    __call__', '__class__', '__del__', '__delattr__', '__deref__', '__dict__', '
    __doc__', '__eq__', '__format__', '__getattr__', '__getattribute__', '__gt__',
    '__hash__', '__init__', '__lt__', '__module__', '__ne__', '__new__', '
    __reduce__', '__reduce_ex__', '__ref__', '__repr__', '__setattr__', '__sizeof__
    ', '__str__', '__subclasshook__', '__swig_destroy__', '__swig_getmethods__', '
    __swig_setmethods__', '__weakref__', 'detach', 'getPtr', 'getPtrAdr', 'getRef',
     'ok', 'rebind', 'this']
```

Figure 2.7: Problems when ignoring 401 warning

inherited variables when you're attempting to use the python program later on. An example can be seen in figure 2.7, where you can see that it is not possible to get access to the functions belonging to the variable `u` when FEM.i is not included. Lesson learned: warnings are there for a reason!

As seen in figure 2.5 on the preceding page, you can also get a warning called 302, looking like this:

**Listing 2.1: 302 warning**

```
Heat1.i:211: Warning 302: Identifier 'Handle_FieldFE' redefined (ignored),
Heat1.i:165: Warning 302: previous definition of  'Handle_FieldFE'.
```

The warning states that the class `Handle_FieldFE` is already defined, and is solved by removing the redundant definition from the interface. In many cases when wrapping Diffpack classes, there will be multiple superfluous definitions of different Handle-classes. Removing the class definitions that triggers the 302-warnings from the interface file will remove the warnings.

```
%module Heat1
%{
/* necessary header files to compile the wrapper code: */
#include </home/.../Heat1.h>
%}

%rename(H_GFE__assign__) Handle_GridFE::operator = (const Handle_GridFE& );
%rename(H_GFE__assign__And) Handle_GridFE::operator = (const GridFE );
%rename(H_GFE__assign__Star) Handle_GridFE::operator = (const GridFE* );
%rename(H_GFE_H_GFE) Handle_GridFE::Handle_GridFE(GridFE *);
%rename(H_GFE_rebind) Handle_GridFE::rebind(GridFE const &);

%init%{
  const char* p[] =  { "Calling Diffpack from Python" };
  initDiffpack(1, p);
%}

%include "/home/.../FEM/swig/FEM.i"

class Heat1 : public FEM
{
public:
...
}
...
```

Figure 2.8: Heat1.i example

After compiling an interface for the first time, the best order to solve the warnings in is to follow this recipe:

1. fix 401 warnings (%include “FEM.i”)
2. Make (recompile)
3. fix 302 warnings (remove redundant classes)
4. Make (recompile)
5. fix 509 and 362 warnings (%rename() the operators and the overloaded functions.)

### 2.1.3 Wrapping Diffpack classes

Often when wrapping Diffpack classes, you will need to wrap more than just the solver you are currently working on (or use already wrapped versions if they exist). This will either be because your solver has a base class that it needs access to, or because you in your program will need to make objects of (or access) classes that are defined in other files. They may in turn also be based on other classes, so there may be quite a lot of classes that needs to be wrapped for one program. This is the case with the Heat1 solver.

Heat1 inherits the FEM class, which made it necessary to generate an interface for the class. As with Heat1, I copied the FEM files from the Diffpack library, and then repeated the procedure

```
Make MODE=opt
mkdir swig
cd swig
MkDpSWIGMakefile MODULE=Heat1 DPDIR=.. MODE=opt
python fixSwigMakefile.py
MkDpSWIGInterface MODULE=Heat1 CLASSES="A B C" FILES=Heat1.h
Make
%fix warnings
Make
Check that "python -c 'from Heat1 import *'" works
```

Figure 2.9: Short summary of wrapping Diffpack files

```
$ ls -a
.   ..  FEM.h  FEM.cpp
$ Mkdir FEM
The directory FEM is created.
Here are all the files in FEM:
total 32
drwxrwxr-x 2 heidi heidi  4096 Aug  8 15:16 ./
drwxr-xr-x 3 heidi heidi  4096 Aug  8 15:16 ../
-rw-rw-r-- 1 heidi heidi   385 Aug  8 15:16 .cmake1
-rw-rw-r-- 1 heidi heidi   652 Aug  8 15:16 .cmake2
-rwxr-xr-x 1 heidi heidi 16165 Aug  8 15:16 Makefile*
$ mv FEM.h FEM.cpp FEM
$ cd FEM
/FEM $ ls -a
.  ..  .cmake1  .cmake2  FEM.cpp FEM.h  Makefile
```

Figure 2.10: How to use the Diffpack Mkdir command

from Heat1, see figure 2.9. In this case, there weren't an already existing Makefile I could use to compile the C++ file FEM.cpp, so I would have to generate one. Luckily, there exist a command called `Mkdir`[8] from Diffpack that is used to generate a Makefile that compiles and links the Diffpack program correctly. As an example on usage, see figure 2.10. The Makefile generated by Mkdir is not to be adjusted, but in some cases it needs more information on which libraries to link with when the program is based on more libraries than Diffpack. This is done by adding a few lines to the hidden file `.cmake2`, which I will come back to in section 3.1 on page 25.

The SWIG wrapping is done in the same way as with Heat1, with one variance. The Heat1 file consist of only óne class, in the FEM file there are several classes. This gives a slightly different input to the `MkDpSWIGInterface` script. The `CLASSES` then looks like this: `CLASSES="FEM IntegrandCalc ElmMatVecCalc ElmMatVecCalcStd MassMatIntg SmoothDerivField SmoothField"`.

One thing that it may be useful to be aware of: if you're not careful when writing the `CLASSES` option, e.g. if you write an extra space between two classes (`CLASSES="FEM IntegrandCalc"`), or misspell a class name, you will get an output which includes warnings like this:

[8] See page 6 in [6]

```
  "Warning 401: Base class 'A' undefined.
2 Warning 401: 'A' must be defined before it is used as a base class."
  instead of
4 "Warning 401: Nothing known about base class 'A'. Ignored."
```

Figure 2.11: Warning given when wrong ordering of the classes in the CLASSES option

**Listing 2.2: Warning when running `MkDpSWIGInterface`**

```
1 Use of uninitialized value $classdeclaration in pattern match (m//) at /home/heidi
    /Programs/diffpack/NO/bin/MkDpSWIGInterface line 60.
  Use of uninitialized value $classdeclaration in concatenation (.) or string at /
    home/heidi/Programs/diffpack/NO/bin/MkDpSWIGInterface line 67.
```

, and the interface file will be generated without the necessary definitions. Also remember to give the class names in the right order, since there will be problems if a class that inherits another class in that file is given to the `CLASSES` option before the parent class. If class B inherits class A, and you give the class names like this: `CLASSES="B A ..."`, you will get a 401 warning like earlier, except with slightly different wording[9].

Since the FEM class inherits the SimCase class (which in turn inherits the HandleId class), I also needed to generate interfaces for them. The SimCase class is a part of the MenuSystem directory in Diffpack, and because it would be necessary to make use of the MenuSystem when working with the wrapped Heat1 class, I wrapped the whole of the MenuSystem directory (all of the files)[10].

When recompiling the FEM interface after having included the HandleId.i file, I got an error saying that the variable `const char* p []` had already been declared, see figure 2.12 on the next page. This was because the 'init' function containing the `initDiffpack` command (that always must be declared when using a Diffpack program) was defined several places, both in the HandleId-, the SimCase- and the FEM-interface. This could be solved by commenting out (or removing completely) the whole `%init` part in the HandleId and the SimCase file, and would need to be done for every file that was wrapped afterwards that was inherited by another class.

So as to not get any more problems regarding the `initDiffpack` function, I removed it from the interface of every class/file I wrapped, except in the main program Heat1 and other classes I would need to make an object of in my program[11]. If the `initDiffpack` command is left out completely, you will get an error message when trying to run the program (see 2.13 on the facing page). The best thing to do may be to remove it from all files that you will need to access (but not necessarily make an object of) in Python, and let it remain in the main file and the classes you need to make an object of in the Python program (like the MenuSystem class). I then wrapped the SimCase file similarly, and recompiled FEM with HandleId and SimCase included. Recompiling Heat1 then gave no error messages.

[9] see figure 2.11

[10] see chapter 2.1.3 on the facing page

[11] For instance MenuSystem

```
FEM_wrap.cxx: In function â void init_FEM() â :
FEM_wrap.cxx:16339: error: redeclaration of â const char* p [] â
FEM_wrap.cxx:16333: error: â const char* p [1] â  previously declared here
make[1]: *** [] Error 1
```

Figure 2.12: Problems with initDiffpack

```
>>> import Heat1
>>> dir(Heat1)
['Heat1', 'Heat1_swigregister', '_Heat1', '__builtins__',\
'__doc__', '__file__', '__name__', '__package__', '_newclass',\
'_object', '_swig_getattr', '_swig_property', '_swig_repr', \
'_swig_setattr', '_swig_setattr_nondynamic']
>>> h=Heat1.Heat1()
>>>>> Handling a fatal exception:  getCommandLineOption reports:
You must call initDiffpack at the beginning of main!
```

Figure 2.13: When missing initDiffpack-command

### MenuSystem

Diffpack programs make use of an implemented menu system, which can be found in

`%NOR/bt/src/libs/bt2/menu/MenuSystem/`

This makes it easier to give input, and as said on page 5 in [5]:

> This is a hierarchical data management system used at run time to define initial values of all entities used in a Diffpack program. Such entities may be simple numerical parameters, but may as well be more abstract quantities such as matrix formats, algebraic solvers, convergence criteria, numerical integration schemes, element types, etc. Thus, the menu system gives any application the ability to select, at run time, all program entities, from simple constants to the numerical algorithms that will be used.

To be able to use the Heat1 solver, I would need to be able to make a MenuSystem-object in the Python program. I copied the whole of the MenuSystem directory from Diffpack, and wrapped each of the individual files. It mostly worked as expected, except for the MenuItems file. When trying to compile the SWIG interface (after running the usual Mkdir, `MkDpSWIGMakefile` and `MkDpSWIGInterface` commands), I got an error message saying that it couldn't find a matching function to a call to `TypeInfo()`, because the arguments were wrong (see figure 2.15 on the next page). After having tried many different things (of which none worked), I included the btTypeInfo.i-file[12] in MenuItems.i, which seemed to work. I got one warning, `MenuItems_wrap.cxx:7045:24: warning: variable 'arg1' set but not used [-Wunused-but-set-variable]`, which didn't cause any problems.

[12] btTypeInfo.i was generated by the commands seen in figure 2.14 on the following page, following a recipe given on page 25 in [15]

```
>>> swigm MODULE=btTypeInfo DPDIR=.. MODE=opt
Generated DpMakefile.defs and Makefile
>>> fixSwigMakefile
>>> swigi MODULE=btTypeInfo CLASSES=TypeInfo FILES=btTypeInfo.h
(then remove the %init% part from btTypeInfo.i, and compile)
```

Figure 2.14: How to wrap btTypeInfo

```
MenuItems_wrap.cxx: In function â PyObject*
    _wrap_Handle_MenuItemBase_type_info_get(PyObject*, PyObject*) â :
MenuItems_wrap.cxx:6912:12: error: no matching function for call to â TypeInfo::
    TypeInfo() â
MenuItems_wrap.cxx:6912:12: note: candidates are:
In file included from /home/heidi/Programs/diffpack/NO/bt/include/genclass.h:21:0,
                 from /home/heidi/Programs/diffpack/NO/bt/include/IsOs.h:36,
                 from /home/heidi/Programs/diffpack/NO/bt/include/MenuItems.h:17,
                 from MenuItems_wrap.cxx:3078:
/home/heidi/Programs/diffpack/NO/bt/include/btTypeInfo.h:82:8: note: TypeInfo::
    TypeInfo(const char*)
/home/heidi/Programs/diffpack/NO/bt/include/btTypeInfo.h:82:8: note:   candidate
    expects 1 argument, 0 provided
/home/heidi/Programs/diffpack/NO/bt/include/btTypeInfo.h:29:7: note: TypeInfo::
    TypeInfo(const TypeInfo&)
/home/heidi/Programs/diffpack/NO/bt/include/btTypeInfo.h:29:7: note:   candidate
    expects 1 argument, 0 provided
make[1]: *** [python_cpp] Error 1
make[1]: Leaving directory `/home/heidi/PulseWrap/Diffpack/Menu/MenuItems/swig'
make: *** [python] Error 2
```

Figure 2.15: TypeInfo error message

```
>>> from Heat1 import *
--------------------------------------------------------------------------------
***    Diffpack Version 4.2.00 - Development Edition (internal use only)     ***
--------------------------------------------------------------------------------
>>> from Menu import *
--------------------------------------------------------------------------------
***    Diffpack Version 4.2.00 - Development Edition (internal use only)     ***
--------------------------------------------------------------------------------
>>> h = Heat1()
>>> m = MenuSystem.MenuSystem()
>>> a = "Heat1"
>>> b = "Heat1menu"
>>> type(a)
<type 'str'>
>>> m.init(a,b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/heidi/PulseWrap/src/Menu/MenuSystem.py", line 112, in init
    def init(self, *args): return _MenuSystem.MenuSystem_init(self, *args)
```

Figure 2.16: Attempting to test Heat1

## 2.2 Testprogram

After successfully wrapping and compiling the Heat1 class into a Python module (and wrapping the MenuSystem module, along with any inherited classes) the next step was to make sure that the Heat1 class could be used in a Python program, and see if it would be possible to implement a few methods in the Heat1 class purely in Python. Testing the Heat1 module, and thereby finding out what other problems that may arise when working with it, would also be paramount to be able to wrap the Pulse module successfully, and make sure that the interface used by the programmer could be implemented in Python.

Before actually making a test program, the first part that needed to be checked was if it (with the work already done) was possible to import Heat1 and the Menu module into a Python script, and then be to be able to make an object of each. As seen in figure 2.16, the import[13] of the modules worked, as did making objects of each, but there was a problem when I was trying to call the MenuSystem.init()-function with string arguments. The init-function takes two arguments in the form of `String const &`, while the arguments I sent in were of type `str`. Diffpack has its own String class that it uses. To be able to send in arguments of the correct type, I would have to wrap the DpString.cpp file (containing Diffpacks `String` function), import the DpString module and make variables with a call to `DpString.String("Heat1")`.

Operator renaming in DpString was a little different than with the Heat1 class (SWIG warnings for DpString can be seen in figure 2.17 on the next page). The 362 warning was accompanied by a 389, 314 and a 503 warning, of which some were solved differently than the earlier warnings. The 389 warning was solved in the same way as the 362, and the 314 was renamed according to the proposal given in the warning, see figure 2.18 on the following page.

[13] To be able to import the Heat1 module and the Menu module anywhere in your file system, you will need to make it visible to Python, see section A.2 on page 52.

```
1  DpString.i:72: Warning 362: operator= ignored
   DpString.i:123: Warning 314: 'del' is a python keyword, renaming to '_del'
3  DpString.i:143: Warning 314: 'from' is a python keyword, renaming to '_from'
   DpString.i:89: Warning 389: operator[] ignored (consider using %extend)
5  DpString.i:150: Warning 503: Can't wrap 'operator ==' unless renamed to a valid
       identifier.
   DpString.i:152: Warning 503: Can't wrap 'operator !=' unless renamed to a valid
       identifier.
7  DpString.i:155: Warning 503: Can't wrap 'operator <' unless renamed to a valid
       identifier.
```

Figure 2.17: Warnings from DpString.i

```
1  %rename(_del) del;
   %rename(_from) from;
3  %rename(String__aref__)  String::operator[](int);
   %rename(String__assign__) String::operator=(const char*);
5  %rename(String__neq__) operator!=(const String& , const char* );
```

Figure 2.18: Renaming in DpString.i

```
1  DpString_wrap.cxx: In function â void init_DpString() â :
   DpString_wrap.cxx:7953:20: error: â initDiffpackâ was not declared in this
       scope
3  make[1]: *** [python_cpp] Error 1
   make[1]: Leaving directory `/home/heidi/PulseWrap/Diffpack/DpString/swig'
5  make: *** [python] Error 2
```

Figure 2.19: Error message if initDiffpack was not removed from DpString.i

The 503 warning needed to be solved a little differently. Having tried the usual way[14] without any notable differences in the warnings, I checked the source code for DpString. The `operator == ()` and the other functions with 503 warnings were (according to DpString.h) "friend functions"[15] of the `String` class. For a definition of what exactly a "friend function" is, I found the explanation at [16] very helpful:

> [...]private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not affect friends. If we want to declare an external function as friend of a class, thus allowing this function to have access to the private and protected members of this class, we do it by declaring a prototype of this external function within the class, and preceding it with the keyword friend.

The solution for the 503 warning was simply to *not* give the String class as the class `operator=` belonged to, but rename without the class name (since the friend function isn't really a member of that class). Using `%rename(...) operator=(...)` instead of `%rename(...) String::operator=(...)` solved the problem. As mentioned earlier, the naming conventions for operator functions can be found on [9].

When DpString was wrapped, the testing could resume. I copied some of a test program from page 325 in [8], which can be seen in figure 2.20 on the next page. The first hiccup in the program happened when the program tried to call `Heat1.scan()`. The following error message says that the program hadn't called the `SimCase::attach()` function:

**Listing 2.3: SimCase::attach error**

```
1 Handling a fatal exception:  SimCase::getMenuSystem() reports:
  This function cannot return a MenuSystem& reference
3 because the SimCase::attach(MenuSystem&) function has not
  been called. You should call that function from your adm
5 routine in your simulator. This will bind the menu system
  to internal data in SimCase such that you can access the
7 menu system through the getMenuSystem() function later.

9 -> TERMINATION due to fatal error.
```

In the Heat1 class, this is done in the `adm()` routine. The problem is that the `adm()` routine starts an interactive session, which is what I wanted to avoid. I wanted to be able to make a Python script that can put in the parameters needed without running the adm() routine, so that everything could be done in one Python script. Having to use an interactive routine would negate most of the point with this thesis. I tried several ways to make this work, but none did. Trying to call the Simcase.attach(menu) in the python function before heat.scan() still gave the error that the attach() function wasn't called. The workaround that solved the problem was in this case to add `SimCase::attach(menu)` in the `Heat1::define()` function in the original C++ code. If you actually want to use the adm() routine, the `attach` function is called twice, but that doesn't seem to cause any problems. I also found after having solved it that this

[14] %rename(String__eq__) String::operator==(const String&,const char);
[15] friend /inline/ bool operator==(const String& x, const char s);

```
from Heat1 import *
from Menu import *
from DpString import *
import math

def run(dt, T, dx, a):
    xnodes=int(1./dx)
    grid_str="P=PreproBox | d=2 [0,1]x[0,1] |\
d=2 e=ElmB4n2D div=[%d,%d] grading=[1,1]" % (xnodes, xnodes)
    grid_str = String(grid_str)
    a.menu.set(String("gridfile"), grid_str)
    time_str = "dt = %e t in [0,%e]" % (dt, T)
    time_str= String(time_str)
    a.menu.set(String("time parameters"), time_str)
    a.solver.scan()
    a.solver.solveProblem()
    a.solver.resultReport()
    return

class A:
    def __init__(self):
        self.menu = MenuSystem.MenuSystem()
        self.solver = Heat1()
        self.menu.init(String("Heat1 menu"), String("Heat1Menu"))
        self.solver.define(self.menu)

if __name__=='__main__':
    a = A()
    T = 2.0
    dt =pow(2,-6)
    dx = pow(2, -2)
    run(dt,T,dx,a)
    a.menu.thisown = 0
    a.solver.thisown = 0
```

Figure 2.20: The testfil.py script for Heat1

"trick" was tried before with success, on page 34 in [14], which would indicate that I was on the right track. After having adjusted the `Heat1::define()` method, the program worked and printed out the correct numbers (see figure 4.2 on page 45).

Another problem arose when the program terminated. The garbage collector was trying to delete an object that had already been deleted, and this triggered a segmentation fault (see figure 2.21 on the next page). Python and C++ doesn't cooperate well when it comes to garbage collection, because they don't know who is in charge of the object and therefore should delete it. From page 46 in [15] you can see that this can be solved by making the MenuSystem- and the Heat1-object class variables, and then setting the object variable `thisown=0`[16], as seen in figure 2.22 on the facing page. This makes the garbage collector work correctly.

[16]From page 46 in[15]: When thisown=1, the object is owned by Python. When it is set to 0, it is owned by Diffpacks Handler.

```
  bt1/HandleId.cpp:121
>>>>> Handling an exception:  ~HandleId reports:
  Trying to delete this object (dynamically allocated),
but there are 1 references! (the number may be nonsense).
  Compile in non-optimized mode and re-run to get more
information. This is a serious error (twice delete is
  possible). Ignore this message if the program was aborted.

  Segmentation fault
```

Figure 2.21: Error message after program was done

```
class A:
    def __init__(self):
        self.menu = MenuSystem.MenuSystem()
        self.solver = Heat1()
        self.menu.init(String("Heat1 menu"), String("Heat1Menu"))
        self.solver.define(self.menu)


if __name__=='__main__':
    a = A()
    T = 2.0
    dt =pow(2,-6)
    dx = pow(2, -2)
    run(dt,T,dx,a)
    a.menu.thisown = 0
    a.solver.thisown = 0
```

Figure 2.22: How to avoid problems with garbage collector

### 2.2.1 TimeLoop

The next step in the testing of Heat1 was to see if it would be possible to implement the `Heat1::timeLoop` function in Python, and then try to get `Heat1::solveAtThisTimestep` to work when implemented purely in Python. If it is possible to reconstruct a C++ function from Heat1 purely in Python, using functionality from the original Heat1 class, it will give a lot more control over the program. Being able to implement the `solveAtThisTimestep` function in the Python script, and within it control the calls to the different functions residing in other classes will make it possible to at a later stage make an interface for the Pulse module which makes all of the communication between the classes in Python.

The program used is the same as in figure 2.20 on page 18, with the difference that the call to `Heat1:solveProblem()` in the run method is replaced by a call to a locally defined `solveProblem` method implemented in Python, that in turn calls the local `timeLoop` function, see figure 2.23 on the facing page. When running the program, the process halted already in the first line in the `timeLoop()` method, with an error that the TimePrm variable `tip` didn't have any attribute called initTimeLoop. To be able to see what was accessible of the `tip` variable's functions and objects in the program, I printed out `dir(solver.tip())`. As seen in figure 2.25 on the next page, it was not possible to access the `initTimeLoop` function just yet.

Python needs to be able to access the functions and variables contained in the TimePrm object, but cannot do this directly from Heat1. For the script to be able to do anything with functions and objects belonging to Heat1s variables, it would be necessary to wrap the classes those variables belonged to. Since `tip` was of type TimePrm, I needed to wrap that class, and import the TimePrm module into the timeLoop script. Before I wrapped TimePrm, I made all variables and functions public, so that I would be able to access the ones I needed later on. In `solveAtThisTimeStep`, the function prints the time at each time step, and to be able to do this from the Python version of the function, I needed to be able to access the time variable. The wrapping didn't cause any problems, and importing the module into the program went without a hitch[17].

A few other classes would need to be wrapped to access the functions and objects needed to implement the timeLoop function. The classes that needed to be wrapped was the SaveSimRes class (since the `database` variable was of this type, and it would be necessary to access SaveSimRes.dump() function) and the TimePrm class (because of already mentioned reasons).

A couple of new problems arose when I was wrapping SaveSimRes that hadn't appeared before. One error message (see figure 2.26 on page 22) was saying that the `SaveSimRes::dump()` function was undefined. As mentioned in section 2.1.1 on page 6, the first thing to check was that the program was compiled with `MODE=opt` both when compiling the C++ program and the wrapper code, but recompiling with `MODE=opt` in both cases didn't fix the problem. I then checked the header file and the `.cpp` file belonging to SaveSimRes, and it turned out that there were several functions defined in the header file that weren't implemented in the `.cpp` file. After commenting out the function definitions that weren't implemented[18] and recompil-

[17] Add `from TimePrm import *` to the Python program

[18] dump(FieldsWithPtValues,...), dump(FieldWithPtValues...), dump(FieldFV...), dump(FieldsFV...), lineCurve(FieldFV...)

```
    def timeLoop(self):
        tip=self.solver.tip()
        tip.initTimeLoop()
        self.solver.setIC()
        self.solver.database().dump(self.solver.u(),tip,"initial condition")
        while tip.finished()==False:
            tip.increaseTime()
            #local solveAtThisTimeStep function
            self.solveAtThisTimeStep()
            self.solver.u_prev = self.solver.u
        return
```

Figure 2.23: The local timeLoop function

```
  File "timeloop.py", line 97, in <module>
    run(dt,T,dx)
  File "timeloop.py", line 77, in run
    solveProblem(menu,heat)
  File "timeloop.py", line 84, in solveProblem
    timeLoop(menu, solver)
  File "timeloop.py", line 32, in timeLoop
    solver.tip.initTimeLoop()
  File "/home/heidi/PulseWrap/Diffpack/Heat1/swig/Heat1.py", line 748, in <lambda>
    __getattr__ = lambda self, name: _swig_getattr(self, Handle_TimePrm, name)
  File "/home/heidi/PulseWrap/Diffpack/Heat1/swig/Heat1.py", line 55, in
     _swig_getattr
    raise AttributeError(name)
AttributeError: initTimeLoop
```

Figure 2.24: Error when first trying to call TimePrm tip from Heat1

```
['H_TP__assign__', 'H_TP__assign__And', 'H_TP__assign__Star', \
'H_TP_rebind', '__call__', '__class__', '__del__', '__delattr__', \
'__deref__', '__dict__', '__doc__', '__eq__', '__format__', '__getattr__', \
'__getattribute__', '__gt__', '__hash__', '__init__', '__lt__', '__module__', \
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__ref__', '__repr__', \
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__swig_destroy__',\
'__swig_getmethods__', '__swig_setmethods__', '__weakref__', 'detach', 'getPtr',
    \
'getPtrAdr', 'getRef', 'ok', 'rebind', 'this']
```

Figure 2.25: Outprint of dir(solver.tip()) before wrapping of TimePrm

```
>>> import SaveSimRes
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "SaveSimRes.py", line 26, in <module>
    _SaveSimRes = swig_import_helper()
  File "SaveSimRes.py", line 22, in swig_import_helper
    _mod = imp.load_module('_SaveSimRes', fp, pathname, description)
ImportError: ./_SaveSimRes.so: undefined symbol:
    _ZN10SaveSimRes4dumpER18FieldsWithPtValuesPK7TimePrmPKcS6_
```

Figure 2.26: Undefined symbol error when importing

```
#Original timeLoop function copied from Heat1.cpp
void Heat1:: timeLoop()
{
  tip->initTimeLoop();
  setIC();
  database->dump (*u, tip.getPtr(), "initial condition");

  while(!tip->finished())
    {
      tip->increaseTime();
      solveAtThisTimeStep();
      *u_prev = *u;
    }
}
```

Figure 2.27: The original timeLoop function in Heat1.cpp

ing, the import statement worked as it should. Another problem was that a segmentation fault was triggered if the %init command was removed from SaveSimRes' interface file, so that was put back into the interface file. After having fixed the errors and imported TimePrm and SaveSimRes, there didn't seem to be any more problems when running the program. The output from timeloop.py (the error field) matched the output when using the C++ version of the timeLoop function, and can be seen in figure 4.2 on page 45.

### 2.2.2 solveAtThisTimeStep

Implementation of the solveAtThisTimeStep function in Python was done in much the same way as the timeLoop function[19]. A few variables used in the solveAtThisTimeStep were of types that weren't already wrapped and imported: the `dof` variable of type DegFreeFE, the `u_summary` of type FieldSummary and `lineq` of type LinEqAdmFE. These classes needed to be wrapped and imported.

After having wrapped and imported the classes into the program, a segmentation fault was triggered, and the error message when using the gdb debugger can be seen in figure 2.28 on the facing page. The LinEqAdmFE inherits LinEqAdm, where the error occurs. It turned out that removing the %init command from LinEqAdm caused the segfault (or at least that keeping it

[19]For the original code of this function, see figure 2.29 on the next page

```
  swig/python detected a memory leak of type 'Vec_real *', no destructor found.
2 Program received signal SIGSEGV, Segmentation fault.
  LinEqAdm::cputime (this=this@entry=0xa35fc0) at linsol/gen/LinEqAdm.cpp:115
4 115   if (proc_manager->master()) {
```

Figure 2.28: segfault from LinEqAdm

```
   void Heat1:: solveAtThisTimeStep ()
2 {
    fillEssBC ();                // incorporate time-dep. ess. b.c.
4
    System (*dof,*lineq);    // FEM's assembly algorithm
6
    dof->field2vec (*u, linsol); // use most recent u as start vector
8   lineq->solve ();             // solve linear system
    s_o << "t=" << tip->time();
10  int niterations; bool c;     // for iterative solver statistics
    if (lineq->getStatistics(niterations,c))  // iterative solver?
12    s_o << oform("  solver%sconverged in %3d iterations\n",
                   c ? " " : " not ",niterations);
14  s_o << '\n'; s_o.flush();    // flush() forces output _now_

16  dof->vec2field (linsol, *u); // load linsol into the field u
    database->dump (*u, tip.getPtr(), "some comment if desired...");
18  u_summary.update (tip->time()); // keep track of max/min u etc

20  // compute smooth flux -k*grad(u);
    FEM::makeFlux (*flux, *u);
22  database->dump(*flux, tip.getPtr(), "smooth flux -k*grad(u)");
  }
```

Figure 2.29: The original code used in Heat1::solveAtThisTimestep()

makes it not happen), as was the case with the SaveSimRes interface.

In the solveAtThisTimestep function, the time in each iteration is printed out, but the `TimePrm::time()` function returns a `Swig Object of type 'real *'`. To be able to print out the time in each iteration, the `TimePrm::t` variable was used directly: `print "t=%f" % tip.t`. This was not an optimal solution, but for me to be able to continue with the Pulse part of the project, it would have to do for the time being. After having imported the modules needed and correcting the other errors, the program worked and got the same results as the testfil.py and the timeLoop.py script (see figure 4.2 on page 45).

The test scripts for Heat1 can be found in `$PULSEWRAPDIR/testing/testHeat1/`[20].

[20] `$PULSEWRAPDIR` is the directory where the PulseWrap directory is installed

```
    def solveAtThisTimeStep(self):
        self.solver.fillEssBC()
        self.solver.makeSystem(self.solver.dof(),self.solver.lineq())
        u = self.solver.u()
        linsol=self.solver.linsol
        tip = self.solver.tip()
        flux = self.solver.flux()
        self.solver.dof().field2vec(u, linsol)
        self.solver.lineq().solve()
        print "t=%f" % tip.t
        self.solver.dof().vec2field(linsol, u)
        print
        self.solver.database().dump(u, tip, "some comment if desired...")
        self.solver.u_summary.update(tip.time())
        self.solver.makeFlux(flux, u)
        self.solver.database().dump(flux, tip, "smooth flux -k*grad(u)")
        return
```

Figure 2.30: The local solveAtThisTimestep function

# Chapter 3

# Python interface for Pulse

With the Heat1 module wrapped and tested, the biggest part of the project still remained: getting the Pulse library wrapped correctly, and testing it. After having wrapped the files belonging to the library, the test program should implement a solver (preferably one already implemented with the original Pulse program so that the solvers could be compared) and a timeLoop function that calls the solveAtThisTimestep from different classes.

Before wrapping the files, I copied the `pulse-cmake/pulse/src` directory to a local one (called Pulse) with the same goal in mind as with the Heat1 directory: avoid changing anything in the original program. The Pulse directory consists of 5 directories, `cellmodel`, `diffusion`, `heart`, `mechanics` and `ode`. Each of these directories consists of several files, of which some consists of more than one class. For each file, a new directory containing a Makefile needed to be generated, and the `Mkdir` command was used as it was in figure 2.10 on page 11.

## 3.1 Wrapping a single class in Pulse

As an example of how a Pulse file was wrapped, the `CellModel` file was chosen. In contrast with the Heat1 program, the compilation (or rather the linking of the libraries) of the original C++ code did not go smoothly. As seen in figure 3.1 on the next page, the compiler generated a lot of `"undefined reference to"` error messages during the linking part of the compilation. The reason for the errors was that the compiler didn't find the libraries it should link with, which was solved (in this case) by adding a few lines to the hidden file `.cmake2`[1] consisting of which library to link with, and where it resided, thereby giving the Makefile the information it needed to link correctly. I then had to copy the Pulse library libpulse.so into the `/usr/lib` directory[2]. After adding the required lines to the .cmake2 file, the program compiled, and the wrapping could begin.

`CellModel` inherits several other classes (`ExplicitODEeqUDC`, `ODE`, `SimCase`), which needed to be wrapped, so that they could be included in the interface file.`ExplicitODEeqUDC` and `ODE` are from the `ode` part of Pulse, while `SimCase` is from the Menu module from Diff-

[1] as seen in figure 3.2 on the following page

[2] Remember when you build (using ccmake) the Pulse library, that you specify that it should be built with shared libraries, so that libpulse.so is generated

```
>>> Make MODE=opt

...

---------- linking ./CellModel.o ----------

g++   -L. -L/usr/X11R6/lib -L/home/heidi/Programs/diffpack/NO/dp/lib/linux-gcc-4/
    opt -L/home/heidi/Programs/diffpack/NO/la/lib/linux-gcc-4/opt -L/home/heidi/
    Programs/diffpack/NO/bt/lib/linux-gcc-4/opt -L/home/heidi/Programs/diffpack/NO/
    ext/linux-gcc-4/lib -o app ./CellModel.o -ldpU -ldpK -llineq -larr3 -larr2 -
    lbt2 -larr1   -lf2c -lm -ldl
./CellModel.o: In function 'CellModel::defineStatic(MenuSystem&, int)':
/home/heidi/PulseWrap/Pulse/cellmodel/CellModel/CellModel.cpp:9: undefined
    reference to 'ODE::defineStatic(MenuSystem&, int)'
/home/heidi/PulseWrap/Pulse/cellmodel/CellModel/CellModel.cpp:14: undefined
    reference to 'hierODEsolver()'
./CellModel.o: In function 'CellModel::scan(MenuSystem&)':
/home/heidi/PulseWrap/Pulse/cellmodel/CellModel/CellModel.cpp:27: undefined
    reference to 'ODE::scan(MenuSystem&)'
...
./CellModel.o:(.data.rel.ro._ZTC16ExplicitODEeqUDC8_6DynSys[_ZTV16ExplicitODEeqUDC
    ]+0x60): undefined reference to 'DynSys::handleRoot(DynamicState&, int)'
./CellModel.o:(.data.rel.ro._ZTC16ExplicitODEeqUDC8_6DynSys[_ZTV16ExplicitODEeqUDC
    ]+0x90): undefined reference to 'typeinfo for DynSys'
collect2: error: ld returned 1 exit status
make: *** [app] Error 1
```

Figure 3.1: Error messages when compiling the C++ code of CellModel

```
LDPATH += -L/home/heidi/Programs/pulse-cmake/local/lib
LIBS += -lpulse
INCLUDEDIRS += -I/home/heidi/Programs/pulse-cmake/local/include
```

Figure 3.2: Lines added to .cmake2 in CellModel

pack. SimCase was already wrapped during the work on the Heat1 problem, so I only needed to do the same to ODE and ExplicitODEeqUDC.

ExplicitODEeqUDC also had problems when compiling the C++ code, but the error message was different from the one in CellModel. This time, the problem was with the f2c library. The error messages is listed in figure 3.3 on the next page. After a lot of googling, and not being able to find any explanation, I tried rearranging the order the libraries were linked in. The solution was to switch the order f2c and Pulse was linked, by adding the lines seen in figure 3.4 on the facing page to the .cmake2 file belonging to `ExplicitODEeqUDC`. `ExplicitODEeqUDC` compiled, and I could proceed with wrapping it. Both `ExplicitODEeqUDC` and `ODE` were wrapped without problems. I then included the interface files for `ExplicitODEeqUDC`, `ODE` and `SimCase` in the `CellModel` interface. This caused a TypeInfo-error (as seen in figure 2.15 on page 14), which was solved in the same way as it was in Heat1 (see section 2.1.3 on page 13) by including btTypeInfo.i in the interface file. The last remaining warning was a 302 warning that the `Handle_DynamicState` class was defined both in ODE.i and in DynSys.i (which was included from `ExplicitODEeqUDC`), and was solved by adding an `%ignore`

```
---------- linking ./ExplicitODEeqUDC.o ----------

 g++   -L. -L/usr/X11R6/lib -L/home/heidi/Programs/pulse-cmake/local/lib/pulse -L/
    usr/lib/ \
  -L/home/heidi/Programs/diffpack/NO/dp/lib/linux-gcc-4/nopt -L/home/heidi/Programs
    /diffpack/NO/la/lib/linux-gcc-4/nopt \
 -L/home/heidi/Programs/diffpack/NO/bt/lib/linux-gcc-4/nopt -L/home/heidi/Programs
    /diffpack/NO/ext/linux-gcc-4/lib -o app \
  ./ExplicitODEeqUDC.o -ldpU -ldpK -llineq -larr3 -larr2 -lbt2 -larr1   -lpulse -
    lf2c -lm -ldl
/usr/lib//libf2c.so(main.o): In function `main':
 /build/buildd/libf2c2-20090411/main.c:138: undefined reference to `MAIN__'
collect2: error: ld returned 1 exit status
```

Figure 3.3: Error message when compiling ExplicitODEeqUDC

```
INCLUDEDIRS += -I/usr/local/include
 LDPATH += -L/usr/lib/
LIBS += -lf2c
 INCLUDEDIRS += -I/home/heidi/Programs/pulse-cmake/local/include
LDPATH += -L/home/heidi/Programs/pulse-cmake/local/lib/pulse
 LIBS += -lpulse
```

Figure 3.4: Switching the order of linking with the f2c and the pulse library (in .cmake2)

`Handle_DynamicState` between the inclusion of `ODE` and `ExplicitODEeqUDC`. The compilation then continued without any problems[3]

## 3.2 Compilation and build of all modules

When I started compiling and wrapping all of the files in Pulse, a problem was early discovered in almost all of the .cpp and .h files. The header files that were included in the .cpp and .h files were included from the local directory using `#include "File.h"` instead of including the global files using `#include <File.h>`. This would cause problems when I created new local directories for all of the classes in Pulse (using Mkdir), and moved the .cpp and the .h files into them, since the compiler wouldn't be able to find the files when they no longer resided in the same directory.

At the beginning of the project, I could not know if I would need to make any alterations

[3]Code snippet from CellModel can be seen in figure 3.5

```
 %include "btTypeInfo.i"
%include "ExplicitODEeqUDC.i"
 %ignore Handle_DynamicState;
%include "ODE.i"
 %include "SimCase.i"
```

Figure 3.5: Include statements in CellModel.i

```
1  ---------- linking ./SuperLU.o ----------

3
  g++   -L. -L/usr/X11R6/lib -L/home/heidi/Programs/pulse-cmake/local/lib -L/home/
      heidi/Programs/diffpack/NO/dp/lib/linux-gcc-4/opt -L/home/heidi/Programs/
      diffpack/NO/la/lib/linux-gcc-4/opt -L/home/heidi/Programs/diffpack/NO/bt/lib/
      linux-gcc-4/opt -L/home/heidi/Programs/diffpack/NO/ext/linux-gcc-4/lib -o app
      ./SuperLU.o -ldpU -ldpK -llineq -larr3 -larr2 -lbt2 -larr1   -lpulse -lf2c -lm
      -ldl
5 /usr/bin/ld: ./SuperLU.o: undefined reference to symbol 'Destroy_CompCol_Matrix'
  /usr/bin/ld: note: 'Destroy_CompCol_Matrix' is defined in DSO /usr/lib/libsuperlu.
      so.3 so try adding it to the linker command line
7 /usr/lib/libsuperlu.so.3: could not read symbols: Invalid operation
  collect2: error: ld returned 1 exit status
9 make: *** [app] Error 1
```

Figure 3.6: Error after adding Pulse to the linking problem

in the C++ files, and therefore would need to use a local version of the header files and not those residing in the `pulse-cmake` directory. I therefore made a few changes to the include statements in the Pulse header files whenever a header file from Pulse was used. I replaced the `#include "File.h"` (or, if it was using a file in another directory, `#include <pulse/heart/File.h>`) with `#include <Pulse/cellmodel/File/File.h>`[4], thereby making sure that the files used were residing in my `PulseWrap/Pulse` directory and not in `pulse-cmake/pulse`. All of the include statements in the Pulse files were then altered.

Most of the wrapping of Pulse went without a hitch, but there were a few files that had different problems than had been seen in Heat1 or CellModel. In most of the .cmake2 files, I would need to add Pulse to the libraries to be linked with, as was done in figure 3.2 on page 26 to make the compilation of the C++ files work correctly. From here on out, I will only mention the files that had new problems, or where it could be useful to remember how to fix the more usual errors.

### 3.2.1 heart

The heart directory was the first to be wrapped, with few hitches, except for the SuperLU and the SuperLU_MT. There were problems with the linking in SuperLU, even after having added Pulse to the libraries. The error message said that it had problems with libsuperlu, as seen in figure 3.6. Adding the SuperLU library to the .cmake2 file made the linking go smoothly (see figure 3.7 on the facing page).

SuperLU_MT needed to be linked with both the Pulse and the f2c library (as was done in 3.4 on the previous page), but since I hadn't been able to install the SuperLU_MT library on my computer, it wasn't possible to wrap it either. Since I didn't have any need for the SuperLU_MT library, this was of no big concern.

[4]Note: the directory containing the local version of the Pulse library needs to be added to the path, so that Python and C++ can find the files).

```
LDPATH += -L/usr/lib/
LIBS += -lsuperlu
INCLUDEDIRS += -I/usr/include/superlu
```

Figure 3.7: Add SuperLU to the libraries Makefile should link with

```
%module create
%{
/* necessary header files to compile the wrapper code: */
#include </home/heidi/PulseWrap/Pulse/cellmodel/create/create.h>
%}

%rename(createODE_nthreads) createODE (const String&,int);

/*
%init%{
  const char* p[] =  { "Calling Diffpack from Python" };
  initDiffpack(1, p);
%}
*/

// Some global variables
extern const char**     hierODE();
extern CellModel*    createODE (const String& problem);
extern CellModel**    createODE (const String& problem,int nthreads);
```

Figure 3.8: The interfacefile create.i

### 3.2.2 cellmodel

In the cellmodel directory, there was a type of C++ file that I hadn't wrapped before, consisting only of functions and no classes. It consisted of the `extern` functions `hierODE` and `createODE`. When I used the `MkDpSWIGInterface` script, it only generated the %module and %init part of the interface, since there weren't any classes to be given to the script. It was therefore necessary to add code to the interface that defined the functions, if I wanted to be able to access the functions later (if nothing was done, the `dir()` command couldn't find the functions when the module was imported into Python). The code added to the interface file can be seen in figure 3.8. I also needed to add `#include <DpString.h>` to create.h, because of an error message when compiling the interface (see figure 3.9 on the next page). The rest of the cellmodel directory went smoothly.

### 3.2.3 diffusion

In diffusion, the two difficult files to wrap were the `VecSimplestGridCollector_Handle` and the `VecSimplestDiffusion_Handle`, the problem there being how the classes were named. Which name should be given to the option `CLASSES`? After a few tries and fails, I found a pattern that worked with both files, which can be seen in figure 3.10 on the following page.

```
...
 In file included from create_wrap.cxx:3067:0:
/home/heidi/PulseWrap/Pulse/cellmodel/create/create.h:9:39: error: â Stringâ
   does not name a type
/home/heidi/PulseWrap/Pulse/cellmodel/create/create.h:9:47: error: ISO C++ forbids
    declaration of â problemâ with no type [-fpermissive]
/home/heidi/PulseWrap/Pulse/cellmodel/create/create.h:10:40: error: â Stringâ
   does not name a type
/home/heidi/PulseWrap/Pulse/cellmodel/create/create.h:10:48: error: ISO C++
   forbids declaration of â problemâ with no type [-fpermissive]
create_wrap.cxx: In function â PyObject* _wrap_createODE(PyObject*, PyObject*) â
   :
create_wrap.cxx:3232:3: error: â Stringâ was not declared in this scope
create_wrap.cxx:3232:11: error: â arg1â was not declared in this scope
...
```

Figure 3.9: Error in create because of DpString.h not included

```
 swigi MODULE=VecSimplestGridCollector_Handle CLASSES=
    VecSimplest_Handle_GridCollector FILES=VecSimplestGridCollector_Handle.h
swigi MODULE=VecSimplestDiffusion_Handle CLASSES=VecSimplest_Handle_DiffusionMG
    FILES=VecSimplestDiffusion_Handle.h

/*
#define Type Handle(DiffusionMG)
#include <VecSimplest_Type.h>
#undef Type

-> CLASSES=VecSimplest_Handle_DiffusionMG
*/
```

Figure 3.10: How to wrap VecSimplestGridCollector_Handle and VecSimplestDiffusion_Handle

```
Materials_wrap.cxx: In function â PyObject* _wrap_delete_Material(PyObject*,
    PyObject*) â :
Materials_wrap.cxx:3353:10: warning: deleting object of polymorphic class type
    â Materialâ which has non-virtual destructor might cause undefined behavior
    [-Wdelete-non-virtual-dtor]
Materials_wrap.cxx: In function â PyObject* _wrap_delete_TransverseIso(PyObject*,
     PyObject*) â :
Materials_wrap.cxx:4993:10: warning: deleting object of polymorphic class type
    â TransverseIsoâ which has non-virtual destructor might cause undefined
    behavior [-Wdelete-non-virtual-dtor]
Materials_wrap.cxx: In function â PyObject* _wrap_delete_ModSVK(PyObject*,
    PyObject*) â :
Materials_wrap.cxx:6239:10: warning: deleting object of polymorphic class type
    â ModSVKâ which has non-virtual destructor might cause undefined behavior [-
    Wdelete-non-virtual-dtor]
Materials_wrap.cxx: In function â PyObject* _wrap_delete_Holzapfel(PyObject*,
    PyObject*) â :
Materials_wrap.cxx:6488:10: warning: deleting object of polymorphic class type
    â Holzapfelâ which has non-virtual destructor might cause undefined behavior
     [-Wdelete-non-virtual-dtor]
Materials_wrap.cxx: In function â PyObject* _wrap_delete_Holzapfel2(PyObject*,
    PyObject*) â :
Materials_wrap.cxx:6851:10: warning: deleting object of polymorphic class type
    â Holzapfel2â which has non-virtual destructor might cause undefined
    behavior [-Wdelete-non-virtual-dtor]
```

Figure 3.11: Warnings when wrapping Materials

### 3.2.4 mechanics

The mechanics directory contained the fewest files and classes - eig3, Myocardium and Materials. The eig3 file needed to have both Pulse and f2c in the .cmake2 file. In the interface file, I did the same as in figure 3.8 on page 29, to be able to access the function. The wrapping of Materials didn't cause any errors, but I got a lot of warnings I hadn't seen before (see figure 3.11). As far as I could tell, there weren't any problems regarding the warnings.

### 3.2.5 ode

In ode, the following files needed both f2c and Pulse added as libraries to link with in .cmake2: `CombinedDynamicState`, `CombinedDynSys`, `ExplicitODEeqUDC`, `createODEsolver` and `ODEsolver_prm`. In `CombinedDynamicState`, `CombinedDynSys` and `EmsODEsolver`, I needed to include btTypeInfo.i in the interface files. Both `createODEsolver` and `createRootFinder` needed extra lines added to the interface for Python to be able to access the functions defined in them, see figure 3.12 on the next page.

## 3.3 Example of use

After the wrapping of the files were completed, the next step was to implement a couple of test programs. The chosen programs were the isotopic_all.i and isometric_all.i in the `pulse-cmake/tests/thinslab/` directory, which meant that it would be possible to compare results from

```
  //createODESolver
2 extern ODEsolver* createODEsolver (ODEsolver\_prm& pm);
  extern const char** hierODEsolver ();
4 //createRootFinder
  extern RootFinder* createRootFinder (String name);
6 extern const char** hierRootFinder ();
```

Figure 3.12: The lines added to the interface files for createODEsolver and createRootFinder

```
      def main(self):
2         self.menu.init(String("Isometric"), String("isometric test"))
          casename = "testbox"
4         cellmodel = "WinslowRice"
          self.infile = "isometric_all.i"
6         cmd = String(cellmodel)
          mech_model = "TransverseIso"
8         self.simulator = Circulation(cmd, String("Something"), String(mech_model)
              )
          self.simulator.thisown = 0
10        self.simulator.define(self.menu)
          menu = self.menu
12        menu.initFromCommandLineArg("--cmd", cmd, cellmodel)
          menu.initFromCommandLineArg("--casename", String(casename), casename)
14        return
```

Figure 3.13: The main() function in the Isometric class implemented in Python

the original version with the Python implementation. In the `thinslab` directory there were already implemented programs that could be used to run the `isometric` and `isotopic` test cases, and the input variables given in those programs were used in the Python version also[5].

The solver class in Pulse is the `Circulation` class residing in `heart`, which in the original test programs is called from main, and therefore also used in the Python implementation of the test programs. It is a simple model for the circulation system, and is necessary to set the correct boundary conditions for the heart model. The `Circulation` class initializes the `Heart` class, and calls functions to model the electrical activity and mechanics in the heart.

The test programs were implemented in three scripts. The main program resides in test.py, where the Python implementations of the functions that were needed reside, as well as the run method and readFile function. In testisometric.py and testisotonic.py the `Isometric` and `Isotonic` classes were implemented, and were then imported into the test.py script. The testisometric.py script initializes the `Circulation` class, makes a `MenuSystem` object, and sets the correct infile and initial cell model and mechanics model in the class. The same is done in testisotonic.py with the isotonic case. The main function in the `Isometric` class can be seen in figure 3.13, and the main function in the `Isotonic` class can be seen in figure 3.14 on the next page

The constructor in the `Circulation` class needed 3 arguments, the cellmodel (for instance WinslowRice), the heart solver (SingleCell, Mono or Heart) and mechanics model (for

[5]The test programs from thinslab were called `run_isotopic_all.verify` and `run_isometric_all.verify`

```
    def main(self):
        casename = "testbox"
        cellmodel = "WinslowHMT"
        self.infile = "isotonic_all.i"
        self.menu.init(String("Isotonic"), String("isotonic test"))
        cmd = String(cellmodel)
        mech_model = "TransverseIso"
        self.simulator = Circulation(cmd, String("Something"), String(mech_model)
            )
        self.simulator.thisown = 0
        self.simulator.define(self.menu)
        menu = self.menu
        menu.initFromCommandLineArg("--cmd", cmd, cellmodel)
        menu.initFromCommandLineArg("--casename", String(casename), casename)
        return
```

Figure 3.14: The main() function in the Isotonic class implemented in Python

instance TransverseIso). To be able to insert the commands from isometric_all.i and isotonic_all.i directly into the `MenuSystem` object (and thereby avoiding the `adm()` function), a `readFile()` function was made. The `forceAnswer` function from `MenuSystem` was used to give the input directly into the menu. Just inserting the commands directly into the `forceAnswer` function wasn't received very well from `MenuSystem`, at least not the `set` command[6], which resulted in the following error message:

**Listing 3.1: Error when using the set prefix in readFile()**

```
>>>>> Handling a fatal exception:  MenuSystem::get reports:
When setting menu answers with the aid of the answers list (StringList)
in the MenuSystem class, the command must not be prefixed by "set".
Here we have encountered "set heart time integration parameters" which is illegal.
Check the functions where statements on the form
     menu.answers.append (aform("set ..."))
appear, and remove the "set" prefix.
```

The `readFile()` function was then adjusted to give the command without the prefix `set`. It also ignores any line starting with "!", since they are comments or lines not used, and any blank lines. The `readFile()` function can be seen in figure 3.15 on the following page.

The init-files (isometric_all.i and isotonic_all.i) were adjusted a little bit before testing to make the program run smoother, and to avoid having to make a more complicated `readFile` function. The commented out lines were removed (for the most part), the last part of any line where there were comments on the end (or other possible values for the input)[7] were removed, and also any curly brackets ("{")[8] since they would add to the complexity of the `readFile` function.

During the run of the test.py program, there was an error message from `MenuSystem`,

[6]For instance "set heart time integration parameters ..."

[7]Example of line before and after editing:`set mech #1:  residual type = ORIGINAL_RES ! PSEUDO_RES ! LEFTPREC_RES !`, `set mech #1:  residual type = ORIGINAL_RES`

[8]example of line where curly brackets were removed: "set udc bodygridfile = {input/thinslab_dir.grid}"

```
    readfile = open(Aobject.infile, "r")
    for line in readfile:
        splitline = line.split()
        if (len(splitline)!=0) and (splitline[0][0]!="!"):
            if splitline[0] =="set":
                Aobject.menu.forceAnswer(String(line[3:]))
            elif splitline[0] =="ok":
                Aobject.menu.forceAnswer(String(line[0]))
    readfile.close()
    return
```

Figure 3.15: A readFile() function to insert commands into MenuSystem

which resulted in adjustments of the init-files. The `MenuSystem` class didn't recognize the `PrecML` preconditioning type given by `set preconditioning type PrecML`, which resulted in the following error:

**Listing 3.2: Error when reading from input file**

```
>>>>> Handling a fatal exception:  Precond_prm::create reports:
Classname "PrecML" is not in Precond hierarchy.
Legal names are:
S/PrecNone/PrecUserDefLU/PrecUserDefInv/PrecUserDefMat/PrecUserDefProc/PrecRILU/
    PrecJacobi/PrecSOR/PrecSSOR/PrecJacobiIter/PrecSORIter/PrecSSORIter/

-> TERMINATION due to fatal error.
```

The preconditioning type was then changed to `PrecNone`[9] in both isometric_all.i and isotonic_all.i, which didn't seem to make any difference in the result. After having adjusted the init-files, the `readFile` function functioned as expected.

As with the Heat1 solver, the `SimCase::attach` function (used in `Circulation::adm()`) needed to be added to the `Circulation::define()` function to get the program to work without using the interactive interface. Also, to have access to different variables in the Heart-, Myocardium- (the mechanics solver), DiffusionMG- and Circulation-classes from Python, the header files (and also the interfacefile) belonging to the classes were adjusted to make all variables public.

After having adjusted the files satisfactorily, the implementation of the test script could resume. For the test script to have the wanted control over the solver, the `timeLoop` function from the `Circulation` class and the `solveCellsAndDiffusion` function from `Heart` (used in `timeLoop`) would have to be implemented. To gain access to the variables and functions needed in the script, the modules in figure 3.16 on the next page were imported in the test.py file: The `Circulation`, `MenuSystem` and `DpString` modules were imported directly into the testisometric.py and testisotonic.py scripts.

Because the `diffusion` variable used in `Heart::solveCellsAndDiffusion` were of type `VecSimplest(Handle(DiffusionMG))` (and not just a Handle function of Dif-

[9]The input was adjusted from the original init-file:`set preconditioning type = PrecML ! PrecNone !` to `set preconditioning type = PrecNone`

```
from testisotonic import *
from testisometric import *
from pulse_heart.Heart import *
from pulse_heart.Cells import *
from pulse_mechanics.Myocardium import *
#for access to the diffusion variable in solveCellsAndDiffusion()
from pulse_diffusion.DiffusionMG import *
from TimePrm import *
```

Figure 3.16: Module imports in test.py

fusionMG), it weren't enough to just import the `DiffusionMG` module to gain access to the functions and objects belonging to the variable. I got this error when I tried to run the program:

**Listing 3.3: VecSimplest(Handle_DiffusionMG) error**

```
/home/heidi/Programs/diffpack/NO/bt/include/VecSimplest_Type.cpp:264
>>>>> Handling an exception:  VecSimplest(Handle_DiffusionMG)::operator= reports:
No assignment operator can be available for this class
since operator= is not required for class Handle_DiffusionMG. Use
class VecSimple instead. When you get this message,
you can take two weeks off, that is about the time it
took to discover that some core dump was due to a
VecSimplest=VecSimplest, where the operator= was undefined
and C++ made its own, wrong version.
```

Importing VecSimplestDiffusion_Handle as well into the script didn't solve it either. After having tried using the VecSimple version (as was suggested in the error message), and not getting any results, it turned out that the solution of the problem was to also add `%include "VecSimplestDiffusion_Handle.i"` to the Heart interfacefile as well as importing `DiffusionMG` into the script. After this, the `diffusion` variable was called and used without fault.

In timeLoop, the `TimePrm` variable `tip` is used a lot. When the program just needs to access the tip.time() function to insert the result into another function, nothing more than importing the TimePrm module is needed, as SWIG just sends a pointer to the returned variable and not the variable itself. Trying to print out tip.time() gives this output: `<Swig Object of type 'real *' at 0x2954fc0>`. If you want to actually print the real value of tip.time(), or need to access it for other reasons (for instance as in `solveCellsAndDiffusion`: `while tip_diff.time() < tip.time()`), this gives more problems. In my program I temporarily solved this issue by instead of using the `time()` function to return the value, I accessed the `tip.t` value directly when I wanted to print it. This is not a good permanent solution, but it made it possible to continue with the program without working too long on that issue. Example of use: `print "Global time is now:  " + str(tip.t)` instead of `print "Global time is now:  " + str(tip.time())`. The `timeLoop` function can be seen in figure 3.17 on the following page, and the `solveCellsAndDiffusion` function can be seen in figure 3.18 on the next page.

It turned out during the testing that the isometric and isotonic test, in the timeLoop function,

```
def timeLoop(Aobject):
    """
    Python implementation of Circulation.timeLoop
    """
    solver = Aobject.simulator
    tip = solver.tip()
    heart = solver.heart
    tip.initTimeLoop()
    solver.dt = tip.Delta()
    save_dt = 0.0
    if (heart.dynamicBC()):
        heart.initTimeLoop(solver.p_cav, solver.restart_time)
        solver.V_heart = heart.getVheart()
        while (not tip.finished()):
            print "Global time is now: " + str(tip.t)
            print "----------- NEW STEP ---------------"

            tip.increaseTime()
            solveCellsAndDiffusion(Aobject)
            try:
                solver.solveMechDynamic()
            except:
                print "Convergence failure at t = " + str(tip.t) + ", exiting
                    program!"
                sys.exit(1)

            solver.dt = tip.Delta()
            heart.updateAndSaveMechVars(solver.dt, save_dt)
            solver.writePVfiles(tip.time(), solver.p_cav ,solver.V_heart)
        print"Succesfully completed Circulation:: timeLoop()"

    else:
        #simplified BCs, no circulation model:
        #solver.heart.simpleTimeLoop()
        simpleTimeLoop(Aobject)
    return
```

Figure 3.17: The Circulation::timeLoop function implemented in Python

```
def solveCellsAndDiffusion(Aobject):
    heart = Aobject.simulator.heart
    tip_diff = heart.tip_diff()
    tip = heart.tip()

    while tip_diff.t < tip.t:
        dt_diff = tip_diff.Delta()
        t0_diff = tip_diff.time()
        print "Solving cell model ODEs..."
        heart.cells.solveAtThisTimeStep(t0_diff,dt_diff)
        print "...done with cell step."
        print "Solving diffusion equations..."
        heart.diffusion(1)().solveAtThisTimeStep()
        print "...done with diffusion step."
    return
```

Figure 3.18: The Heart::solveCellsAndDiffusion function implemented in Python

```
def simpleTimeLoop(Aobject):
    save_dt = 0.0
    solver = Aobject.simulator
    tip = solver.tip()
    heart = solver.heart
    dummy2 = 0.0
    dummy = 1
    #heart.initTimeLoop(dummy, dummy2)
    heartInitTimeLoop(Aobject)
    while (not tip.finished()):
        print "Global time is now: %d " % tip.t
        print "----------- NEW STEP ---------------"
        tip.increaseTime()
        solveCellsAndDiffusion(Aobject)
        try:
            heart.solveMechSimple();
        except:
            print "Convergence failure at t = %d, exiting program!" % tip.t
        heart.updateAndSaveMechVars(tip.Delta(), save_dt)
    print "Successfully completed Heart::timeLoop."
    return
```

Figure 3.19: The Heart::simpleTimeLoop function implemented in Python

jumped to the `Heart:simpleTimeLoop` function, since the call to `heart.dynamicBC()` returned `False`. Since the timeLoop function therefore couldn't be tested thoroughly with the test cases chosen, the `simpleTimeLoop` function was also implemented in test.py. The next problem that appeared (and the one I worked most on solving during the testing of the program) was with the `Ptv(real)` class, used in the `simpleTimeLoop` function when calling `Heart::initTimeLoop(const Ptv_real& cavity_p, real restart_t)`. In the wrapper, the Python name of the class was `Ptv_real`. According to [2], the `Ptv(real)` class resided in the Ptv_real.h file, and should be possible to wrap. The hope was that it would be possible to make Ptv_real objects in Python and give them as arguments to functions that required Ptv(real) as arguments, just as it was done with the DpString class in the Heat1 chapter. The problem was that to wrap the classes in the file, I would need to know how SWIG would rename them to give the class names as an input to the `MkDpSWIGInterface` script. This was easier said than done, for of course the class wasn't really named Ptv_real in Ptv_real.h. Another class type had replaced it, and was defined by

**Listing 3.4: New version of Ptv(real)**

```
#define PtvGenDIM 0
#include <PtvGen_Type.h>
typedef PtvGen(Type,PtvGenDIM) Ptv(Type);
#undef PtvGenDIM
```

where PtvGenDIM could be any integer from 0 to 3. I found out that it was possible to wrap header files when I attempted to wrap a file involving the Ptv class, VecSimplest_Ptv_int, where the class definitions looked like this:

**Listing 3.5: VecSimplest_Ptv_int class**

```
#define Type Ptv(int)
#include <VecSimplest_Type.h>
#include <VecSimple_Type.h>
#undef Type
```

The class definitions in Python was VecSimplest_Ptv_int and VecSimple_Ptv_int, as I would expect. But with the Ptv_real.h class, I couldn't find any version of the class name that `MkDpSWIGInterface` would accept. When working with another problem in the test script (which I will come back to later in this section), I added these lines to Circulation.i:

**Listing 3.6: Added to the Circulation interface**

```
%include "typemaps.i"

%typemap(in) real& {
  real *d =  new real(PyFloat_AsDouble($input));
   $1 = d;
  printf("Received a float: %d\n", $1);
 }
%typemap(in) double& {
   double *d = PyFloat_AsDouble($input);
  $1 = d;
 }
```

Compiling the Circulation wrapper, the following warning appeared, which gave me the solution to what class names should be used in Python:

**Listing 3.7: Error that gave the name to use for classes in Ptv_real.h**

```
Circulation_wrap.cxx: In function â PyObject* _wrap_Circulation_writePVfiles(
    PyObject*, PyObject*) â :
Circulation_wrap.cxx:6867:41: warning: format â %d â expects argument of type
    â int â , but argument 2 has type â Ptv_real* {aka Ptv0dreal*} â [-Wformat]
Circulation_wrap.cxx:6872:41: warning: format â %d â expects argument of type
    â int â , but argument 2 has type â Ptv_real* {aka Ptv0dreal*} â [-Wformat]
```

After running `MkDpSWIGInterface MODULE=Ptv_real CLASSES="Ptv0dreal Ptv1dreal Ptv2dreal Ptv3dreal" FILES=Ptv_real.h`, the interface for Ptv_real was generated and the file could be wrapped.

Unfortunately, making Ptv0dreal objects in the test.py script wasn't the solution to the original problem (which was to make a Ptv_real object and passing it as an argument to one of the C++ functions). It was not recognized as a Ptv_real (and as such not accepted), even if the wrapper obviously (as can be seen from the previous warning) knew that Ptv_real and Ptv0dreal were the same class type. The only solution I could find that worked was to insert a few typemaps into the various interface files where I would need to input Ptv_real variables from the Python script.

A typemap[10] can be used to specify what an input or output to or from a function wrapped with SWIG should give, and how to convert it between C++ types and (in this case) Python types. It can either be a general conversion for all inputs and outputs of that type, or it can be specified to one certain variable. The latter is the one I chose to use, as I found out how to solve my problem with Ptv_real at a late date and therefore didn't have time to make it more general, and I this way at least avoided to automatically convert any integer input into Ptv_real objects (which could make later use much more problematic). The typemap I used that solved the Ptv_real problem looked like this:

**Listing 3.8: Typemap for the const Ptv_real & cavity_p variable**

```
%typemap(in) const Ptv_real& cavity_p {
  $1 = new Ptv_real(PyInt_AsLong($input));
}
```

and were put into the Heart class in the Heart.i file. This little code snippet made sure that all inputs of the variable `cavity_p` into any Heart function where the input was of type int would be converted to `const Ptv_real&`, and would not do anything about other arguments or variables.

It was also necessary to make a typemap for a few variables of type `real`, which weren't converted correctly by SWIG. Two were the variables sent as arguments into updateAndSaveMechVars(const real& dt, real& save_dt) in the simpleTimeLoop function, and the third variable was the second argument into the `Heart::initTimeLoop` function called in the `timeLoop` function. The variables were converted in these typemaps set in Heart.i:

**Listing 3.9: Typemap added to Heart.i**

```
%typemap(in) real restart_t {
  real *a =  new real(PyFloat_AsDouble($input));
  $1 = *a;
 }
%typemap(in) const real& dt {
  $1 = new real(PyFloat_AsDouble($input));
}
 %typemap(in) real& save_dt {
  $1 = new real(PyFloat_AsDouble($input));
 }
```

To make sure that I had encountered most of the problems that could arise in an implementation of a solver using Pulse, `Heart::initTimeLoop` function was also implemented in test.py and can be seen in figure 3.20 on the following page.

As was shown earlier, there will arise problems if you try to input values to a function that takes arguments of Ptv_real without having a typemap that converts the type correctly. The same is true if you assign a value to a variable that expects an Ptv_real object. In the assignment

[10]From chapter 10, page 139, in [12]: "'typemaps' are an advanced customization feature that provide direct access to SWIG's low-level code generator."

```
def heartInitTimeLoop(Aobject):
    cavity_pressure = 1
    restart_time = 0.0
    solver = Aobject.simulator
    heart = solver.heart
    mech = heart.mechanics()
    tip = heart.tip()
    tip.initTimeLoop()
    tip_diff = heart.tip_diff()
    tip_diff.initTimeLoop()
    heart.p_cav = cavity_pressure
    while (tip.t < restart_time):
        tip.increaseTime()
        while (tip_diff.t < tip.t):
            tip_diff.increaseTime()

    if heart.diffusion.size() == 1:
        heart.diffusion(1)().attachIntegrands()
    else:
        print "Heart:: initTimeLoop Diffusion has wrong size - should be 1"
        sys.exit(1)

    heart.diffusion(1)().initBlocks()
    heart.diffusion(1)().makePreconditioner()
    a = heart.diffusion(1)().getNoOfGrids()
    heart.diffusion(1)().getDataStructureOnGrid(a)

    if heart.dynamic_mech_BC:
        try:
            heart.V_heart = mech.computeNewVolume(heart.p_cav, 0.0)
            mech.dumpDeformedGridAndFibers()
            #print "..Initial Conditions, p = %d  V =%d"  %(heart.p_cav, heart.
                V_heart)
        except:
            print "Convergence failed for initial cavity pressure, exiting."
            sys.exit(1)

    else:
        heart.solveMechSimple()
    mech.saveResults()
    mech.computeLambda()
    mech.updateActivationFields()
    mech.dumpTrackPoints()
    if (heart.moving_bidomain):
        heart.diffusion(1)().updateDeformedConds(mech.getInverseCField())
    return
```

Figure 3.20: The Heart::initTimeLoop() function implemented in Python

`heart.p_cav = cavity_pressure` in heartInitTimeLoop(), that is what happens. To avoid errors, a typemap in Heart.i looking like this will correct it:

**Listing 3.10: Typemap in Heart.i**

```
%typemap(in) Ptv_real p_cav {
  Ptv_real *a = new Ptv_real(PyInt_AsLong($input));
 $1 = *a;
 }
```

The call to `Myocardium::computeNewVolume` in heartInitTimeLoop also needed additional typemaps included in Myocardium.i.

**Listing 3.11: Typemaps added to Myocardium.i**

```
 %typemap(in) const Ptv_real& p_cav {
       $1 = new Ptv_real(PyInt_AsLong($input));
 }
%typemap(in) const real& lin_stress {
   $1  =  new real(PyFloat_AsDouble($input));
}
```

One new problem arose that I hadn't seen before, but which was kind of similar to other encounters: I was trying to call

```
a = heart.diffusion(1)().getNoOfGrids()
heart.diffusion(1)().getDataStructureOnGrid(a)
```

and got this error:

**Listing 3.12: Error when trying to get the heart.diffusion(1)().getNoOfGrids()**

```
 File "/home/heidi/PulseWrap/src/pulse_diffusion/DiffusionMG.py", line 879, in
    getDataStructureOnGrid
    def getDataStructureOnGrid(self, *args): return _DiffusionMG.
       DiffusionMG_getDataStructureOnGrid(self, *args)
TypeError: in method 'DiffusionMG_getDataStructureOnGrid', argument 2 of type '
    SpaceId'
```

The SpaceId type was defined in `MLSolver_enum.h`. When I added `%include "MLSolver_enum.h"` in the DiffusionMG.i file so that the wrapper could identify the type used, the problem was solved.

After having wrapped, imported and included what was needed to get the test.py script to work, the results were promising. The cells.track files generated by the test scripts were plotted and compared to the original cells.track files generated from the C++ version, and they turned out to be the same. Both with the isometric and the isotonic case, the scripts were tested with both `t=20` and `t=500`, and plots from both cases can be seen in section 4.1.1 on page 46.

# Chapter 4

# Summary and discussion

## 4.1 Challenges and results

Not all challenges encountered when testing the wrapped Pulse library were solved, due to too little time left at the end of the project. Memory handling and deletion of variables and objects are a constant source of trouble when working with a combination of C++ and Python code. The most obvious error was triggered when C++ and Python were quarreling about who was the owner of the objects made in Python of C++ classes, as seen in figure 2.21 on page 19, and was solved by setting the objects `thisown` variable equal to 0. Even if this problem was solved, other warnings were constantly generated throughout the testing of both Heat1 and Pulse which I didn't have time to resolve:

**Listing 4.1: Memory warning**

```
1 swig/python detected a memory leak of type 'Vec_real *', no destructor found.
  swig/python detected a memory leak of type 'real *', no destructor found.
3 swig/python detected a memory leak of type 'Ptv_real *', no destructor found.
```

From what I have understood, the warning is triggered because `Vec_real`, `Ptv_real` and `real` are types of which SWIG has to little information available, but they don't cause any problems that I can see. Another problem has been that I didn't have time to properly learn the typemap functionality, so I hadn't time to get this print statement in `heartInitTimeLoop` to work:`print "..Initial Conditions, p = %d V =%d" , %(heart.p_cav, heart.V_heart)` Since `heart.p_cav` is of type `Ptv_real`, it would not print out the value, only something like this: `<Swig Object of type 'Ptv_real *' at 0xc55d80>`

A more serious error occur at the end of program, see figure 4.1 on the next page. This is a problem that occur if I do something in the program *after* the computation is done, like when I use the `subprocess` module's functionality to move the files generated into another directory (used inside the python program). If a print statement is added between the call to the `run` function and the call to `subprocess` , the program runs as it should, and no error occur. Or, that is not exactly true. At the end of all run-throughs of testscripts, both with Heat1 and Pulse, a strange error was triggered:

```
*** glibc detected *** python: double free or \
```

```
corruption (!prev): 0x00000000012e0960 ***@
```

But the program generates the correct files, and since the `glibc` error is triggered after the program is done running, it doesn't interfere with the process.

Plots from the cells.track file for both the isometric case and the isotonic case can be seen in 4.1.1 on page 46 compared with the original test programs output. The results gained from both the original C++ program and the Python implementation are the same, and the project can therefore be viewed as a success. I tested with both t=20 and t=500 in both the isometric and the isotopic case. All plots and files can be found in the $PULSEWRAPDIR/testing/testPulse directory. The Heat1 test case generated the same output of error field both in the original and Python version, and the output from the program can be found in figure 4.2 on the next page.

```
1 Traceback (most recent call last):
   File "test.py", line 164, in <module>
3    subprocess.Popen(args, shell=True)
   File "/usr/lib/python2.6/subprocess.py", line 569, in __init__
5    _cleanup()
   File "/usr/lib/python2.6/subprocess.py", line 438, in _cleanup
7    for inst in _active[:]:
 TypeError: a float is required
9 Exception AttributeError: "'Popen' object has no attribute '_child_created'" in <
    bound method Popen.__del__ of <subprocess.Popen object at 0x7f0c737be190>>
    ignored
```

Figure 4.1: Error when no print statement

```
 heidi@baffel ~/Master/Swig/Heat1/swig $ python testfil.py
 ------------------------------------------------------------------------------
***    Diffpack Version 4.2.00 - Development Edition (internal use only)     ***
 ------------------------------------------------------------------------------
 ------------------------------------------------------------------------------
 ***    Diffpack Version 4.2.00 - Development Edition (internal use only)     ***
 ------------------------------------------------------------------------------

Grid: P=PreproBox | d=2 [0,1]x[0,1] |d=2 e=ElmB4n2D div=[4,4] grading=[1,1]


 L1-error= 8.32617e-17,  L2-error= 1.03140e-16,  max-error= 2.01193e-16


[25]  nodal values of the error field $
 (1,1)= 0.000000e+00  (2,1)= 0.000000e+00  (3,1)= 0.000000e+00
(4,1)= 0.000000e+00  (5,1)= 0.000000e+00  (1,2)= 0.000000e+00
 (2,2)=-1.146857e-16  (3,2)=-1.621901e-16  (4,2)=-1.146857e-16
(5,2)= 6.197791e-34  (1,3)= 0.000000e+00  (2,3)=-1.621901e-16
 (3,3)=-2.293715e-16  (4,3)=-1.621901e-16  (5,3)= 8.765000e-34
(1,4)= 0.000000e+00  (2,4)=-1.146857e-16  (3,4)=-1.621901e-16
 (4,4)=-1.146857e-16  (5,4)= 6.197791e-34  (1,5)= 0.000000e+00
(2,5)= 6.197791e-34  (3,5)= 8.765000e-34  (4,5)= 6.197791e-34
 (5,5)= 1.073403e-49

 L1-error= 8.32617e-17,  L2-error= 1.03140e-16,  max-error= 2.01193e-16
```

Figure 4.2: Results from the scripts in Heat1

### 4.1.1 Plots from cells.track in Pulse

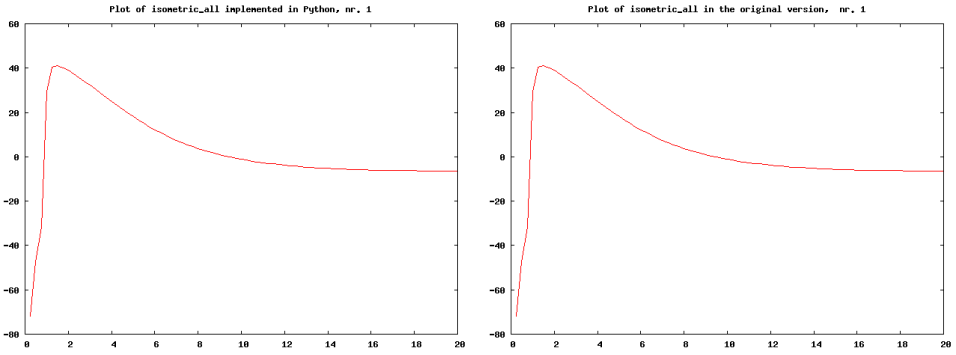



Figure 4.3: Plot of isometric_all nr 0, left, python, right: C++

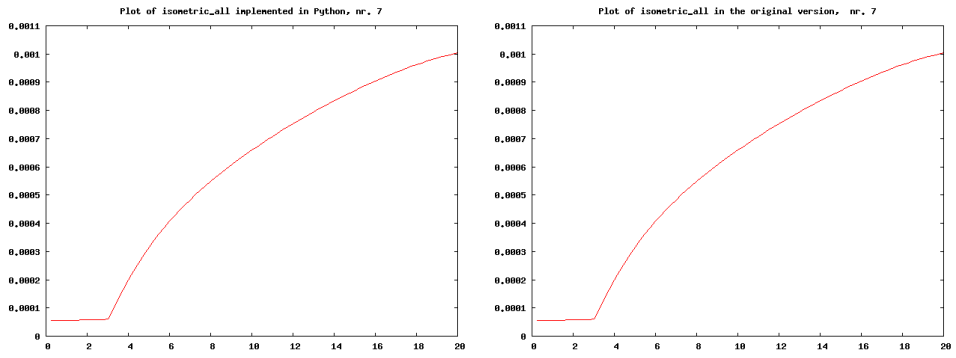



Figure 4.4: Plot of isometric_all nr 6, left: python, right: C++

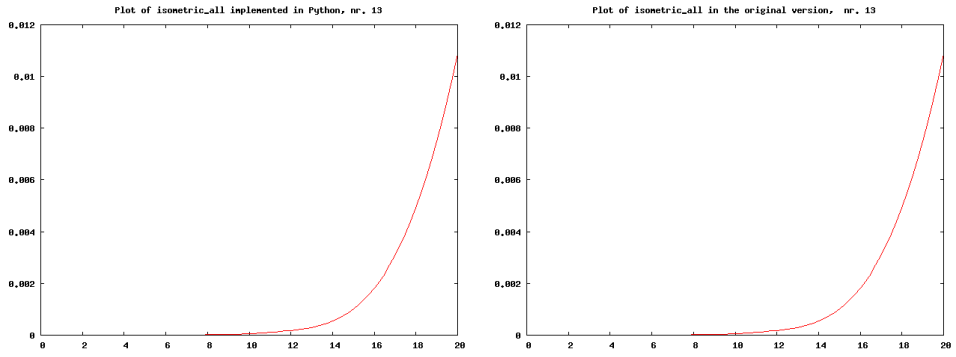



Figure 4.5: Plot of isometric_all nr 12, left, python, right: C++

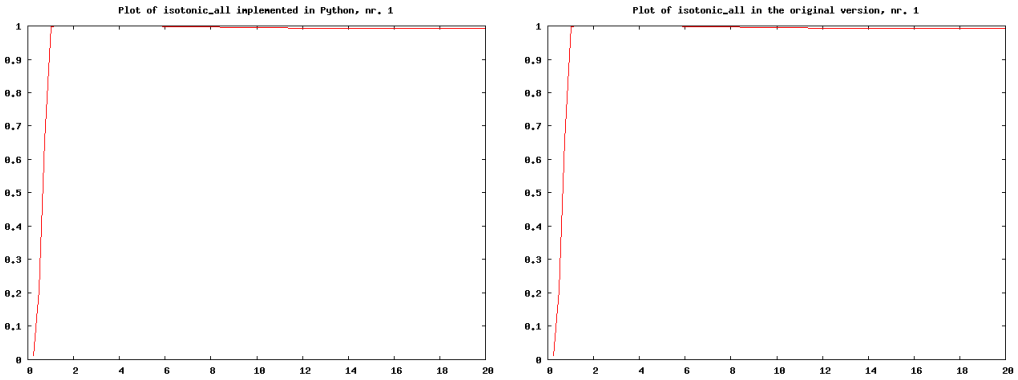



Figure 4.6: Plot of isotonic_all nr 0, left: python, right: C++

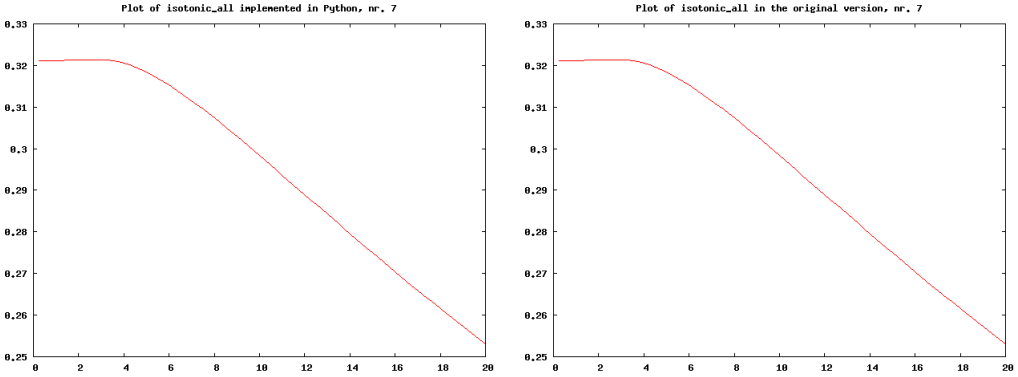



Figure 4.7: Plot of isotonic_all nr 6, left: python, right: C++

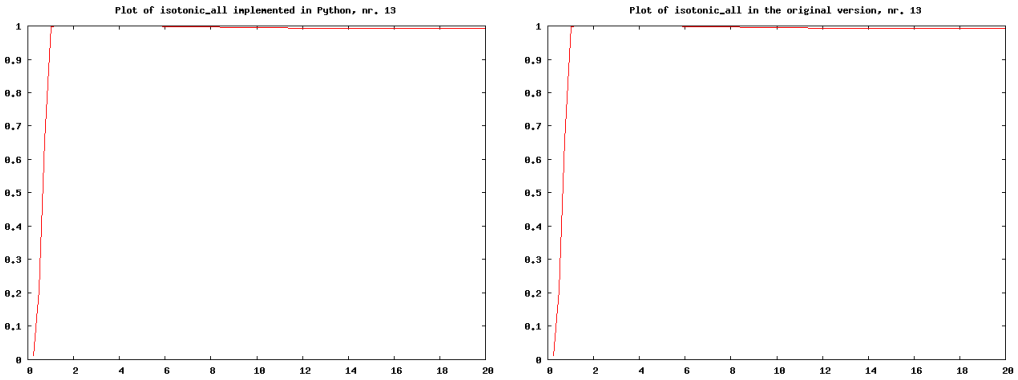



Figure 4.8: Plot of isotonic_all nr 12, left: python, right: C++

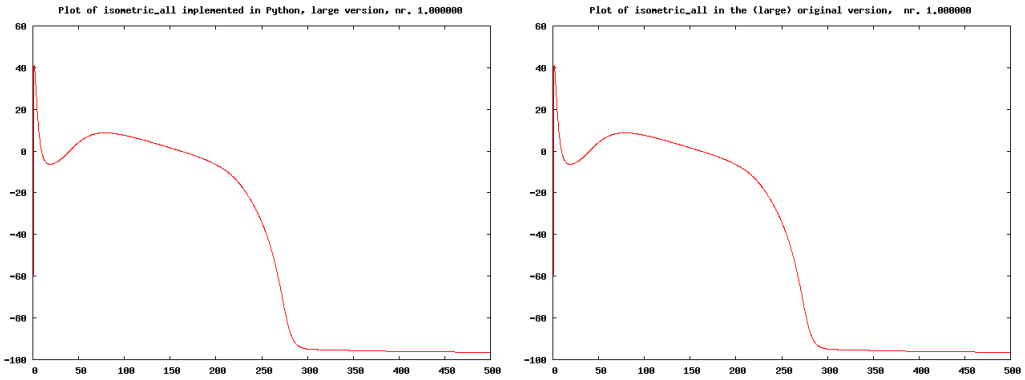



Figure 4.9: Plot of isometric_all (with dt=0.25, t=500) nr 0, left: python, right: C++

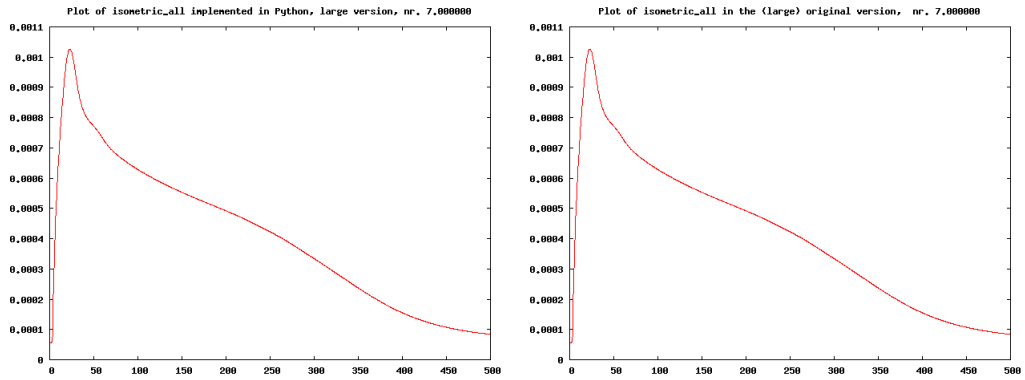



Figure 4.10: Plot of isometric_all (with dt=0.25, t=500) nr 6, left: python, right: C++

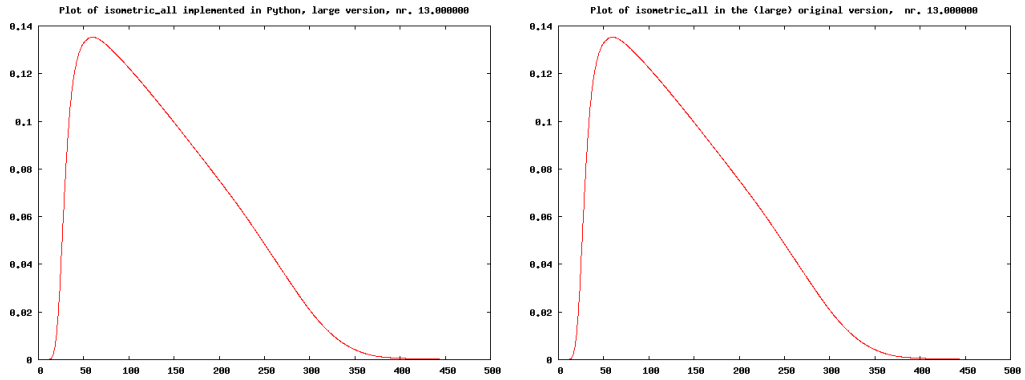



Figure 4.11: Plot of isometric_all (with dt=0.25, t=500) nr 12, left: python, right: C++

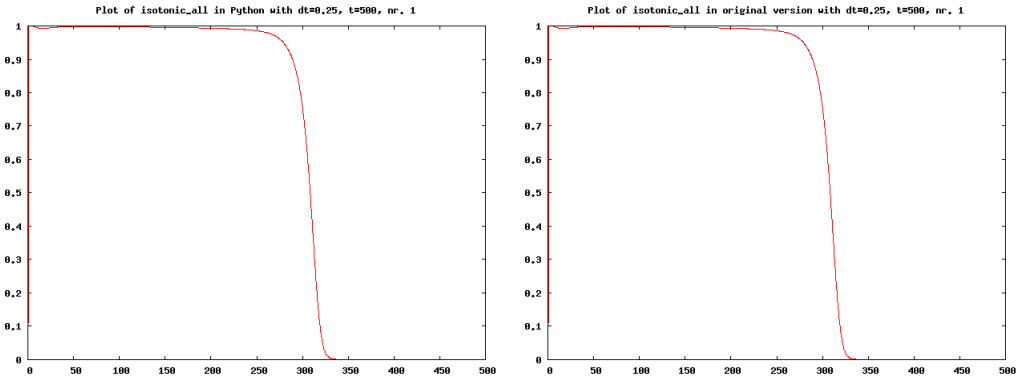



Figure 4.12: Plot of isotonic_all (with dt=0.25, t=500) nr 0, left: python, right: C++

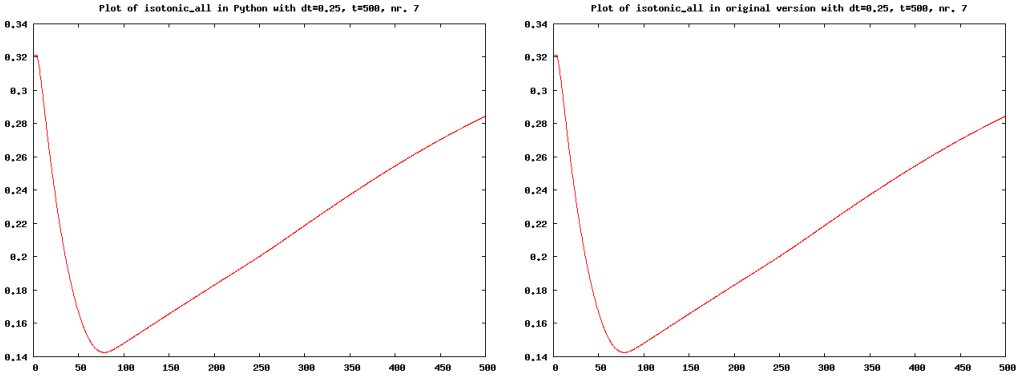



Figure 4.13: Plot of isotonic_all (with dt=0.25, t=500) nr 6, left: python, right: C++

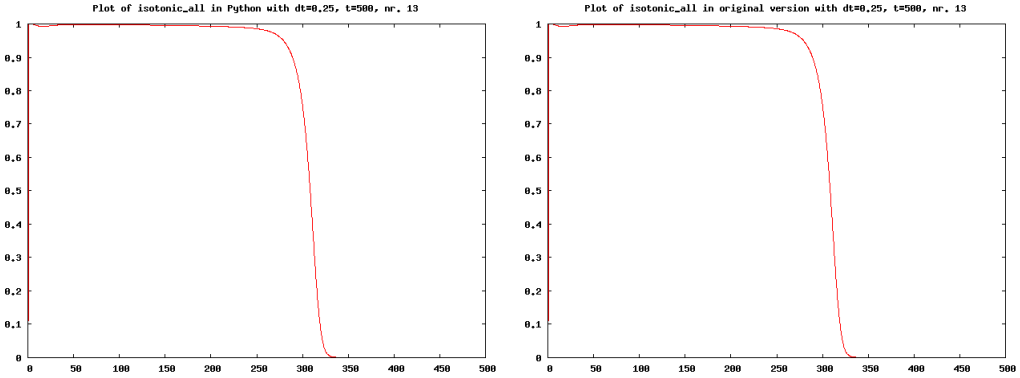



Figure 4.14: Plot of isotonic_all (with dt=0.25, t=500) nr 12, left: python, right: C++

## 4.2 Remaining work

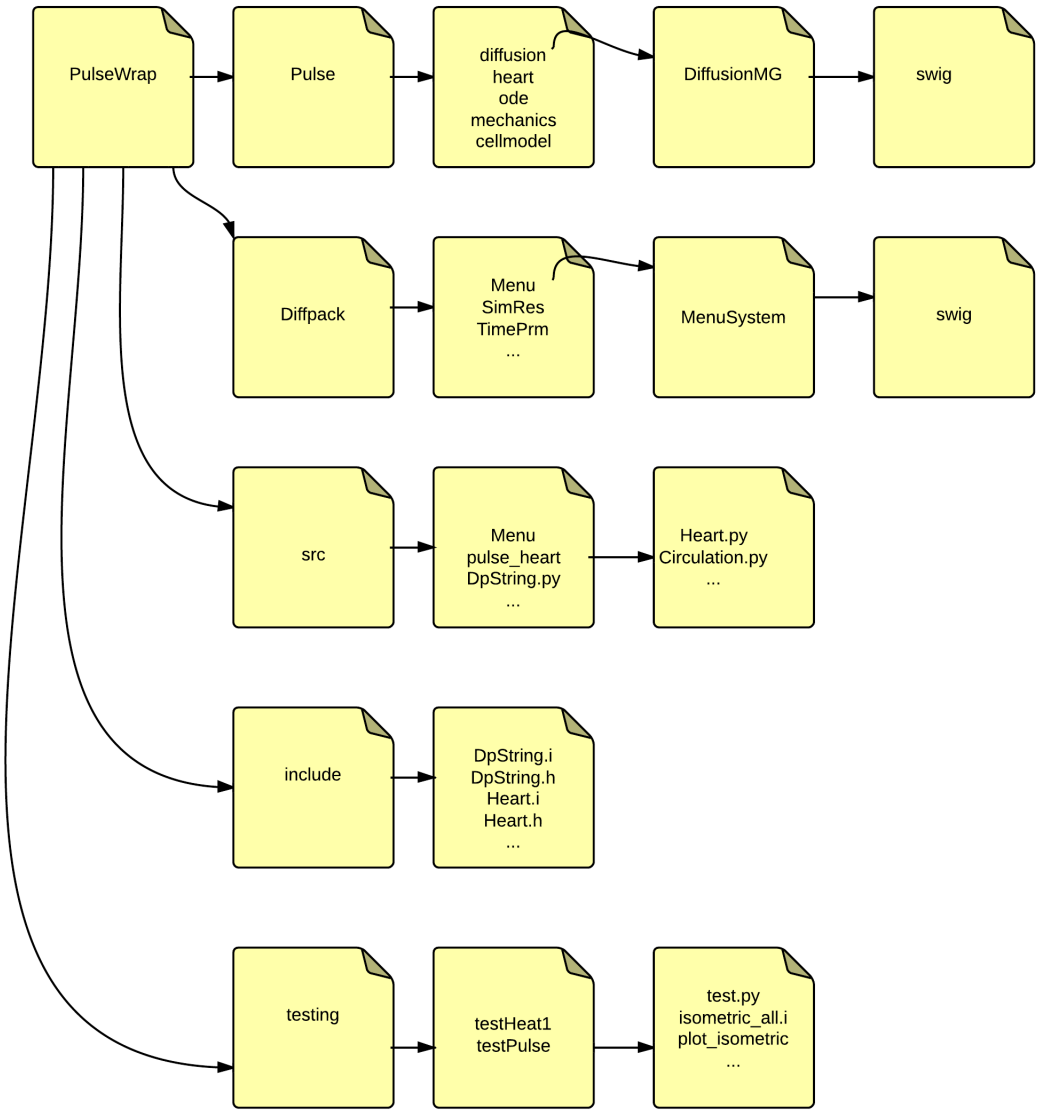
A lot of time went into just getting the wrapping of the Heat1 test case to work, and even more time into getting the test programs to work. The same was the case with Pulse, and the test programs in the testPulse directory weren't actually working until the last week before deadline. The last part of the project, design and possible implementation of a Python interface, was therefore not achieved. With more time it would be possible to make a user interface that does all of the communication between the classes in Python, and also evaluate the efficiency and flexibility of the Python wrapped simulator compared to the original C++ version.

Because of too little time remaining to understand the typemap functionality of SWIG, the solution to a few of the problems when using the Pulse library in Python was solved for only the specific variables and functions needed in the test program (test.py). More general solutions that could be implemented in the other classes to set and get variables, could be solved if more time had been possible. The readFile() function could easily be made more general, and use the MenuSystem functionality more, so that the user didn't have to adjust the input files for already implemented Pulse solvers.

# Appendix A

# Setup of PulseWrap

## A.1 Directory structure of PulseWrap



## A.2 Python module

When you work with scripting or with programming in general, you often make programs that you want to reuse later, either small scripts that make everyday life simpler, or larger programs that may be useful to be able to reuse in later projects. There are two different ways to reuse code that it is good to know of. One is to be able to run the script from command line from anywhere in your directory tree, the other is to import the script as a module into another Python script.

To be able to run a script from anywhere in your home directory, you will first need to add a shebang line[1] to the top of your script: `#!usr/bin/env python`. This makes your system understand that it is a script, and to run it using Python. The next step is to make a directory where you want to store your scripts, and to make the script executable using `chmod +x script.py`. Then you need to add the path to your directory to the $PATH variable so the scripts can be found[2].

If you want to use previously defined functions and classes in a new program/script, you will have to make a Python module/package of it. Make a directory where you want to store your Python modules, then add the path to that directory to $PYTHONPATH in `.bashrc`[3]. When you import a module into your program, it will be searched for in this order: first the directory where the script you work on resides, then PYTHONPATH, and lastly the installation dependent default.

**Listing A.1: Example of directory structure for python modules**

```
Scriptdir/
          __init__.py
    Pythonmodule1/
                  __init__.py
     script1.py
     script2.py
    Pythonmodule2/
                  __init__.py
                  script3.py
                  script4.py

    script5.py
    script6.py
```

A Python module can be either a single python file, or a directory containing several Python scripts. Figure A.1 is an example of a directory containing several python modules, two consisting of several files, and two single python scripts (which may contain several classes and functions). Every directory with Python modules needs to contain an `__init__.py` file, which can be either empty or with initialization code. The `__init__.py` file is necessary for Python to understand that the directory contains packages, and if you have a subdirectory with

[1] From [13]"In computing, a shebang [...], when it occurs as the initial two characters on the initial line of a script, is the character sequence consisting of the characters number sign and exclamation mark (that is, "#!")."

[2] Add `export PATH=/path/to/dir:$PATH` to `.bashrc`

[3] `export PYTHONPATH=/path/to/dir:$PYTHONPATH`

several python files you want to import as a single module, the `__init__.py` file inside that directory needs to contain an `__all__` variable. The `__init__.py` file in Pythonmodule1 would e.g. contain `__all__=["script1", "script2"]`. To import the module into your script, you could either use `import Pythonmodule1` or `from Pythonmodule1 import *`. The `__all__` is necessary if you want to use the `from module import *`. For further explanation, see [3].

## A.3 Include files in Diffpack

As mentioned earlier, you will need to add an `%include file.h"` statement to your interfacefile if one of your classes inherits a parent class residing in another file. To avoid having to give the whole path to the file (and as a result making your program less usable by other people), you want to let Diffpack find the directories where your include files/interface files resides. This is done by adjusting the Makefile generated by MkDpSWIGMakfile, by adding `-I$(PULSEWRAP)/include` to the SWIGOPT parameter. I adjusted my fixSwigMakfile.py script, so that in addition to removing the things that were redundant and hindered the compilation of the SWIG wrapper, it also adds the `PulseWrap/include` directory to the directories Diffpack should look through.

I have also added a configuration file to the PulseWrap directory called `PW.conf`[4] that contains all the necessary environment variables needed to use PulseWrap. Instructions and explanations on how to compile and install the PulseWrap directory can be found in the README file in the PulseWrap directory.

[4] You will need to make a few adjustments in PW.conf, explanations on how can be found as comments (#) in the file.

# Appendix B

# Code

## B.1 The fixSwigMakefile.py script

**Listing B.1: fixSwigMakefile.py**

```
#!/usr/bin/env python
"""
This script is used to fix the Makefile made by the MkDpSWIGMakefile-command.
It removes the "-c", comments out the checkDpMode.py-line, and removes the
-lswigpy, since it belongs to an older version.
"""
import sys

M=open("Makefile", "r")

#lines=M.readlines()
lines=M.readlines()

c=lines[10]
check=lines[21]
swigpy=lines[25]
if c[14]=="c":
    c=c[0:12]+c[15:]
if c[30:38]=="-I$(NOR)":
    c = c[0:29]+ " -I$(PULSEWRAP)/include " + c[30:]
if check.strip()[0]!="#":
    check="#"+check
if swigpy[73:80]=="lswigpy":
    swigpy=swigpy[0:72]+swigpy[80:]
lines[10]=c
lines[21]=check
lines[25]=swigpy
lines="".join(lines)
M.close()
M=open("Makefile", "w")
M.write(lines)
M.close()
```

## B.2 Heat1 testscripts

**Listing B.2: testfil.py**

```
#!/usr/bin/env python
"""
File to test if the Heat1 problem is functional after being wrapped with Swig,
and to check if the MenuSystem is functional.
Code mostly copied from "Advanced Topics in Computational Partial Differential
    Equtions: Numerical Simulations,...diffpack programming" by H.P. Langtangen,
    Aslak Tveito
"""
from Heat1 import *
from Menu import *
from DpString import *
import math

def run(dt, T, dx, a):
    xnodes=int(1./dx)
    grid_str="P=PreproBox | d=2 [0,1]x[0,1] |\
d=2 e=ElmB4n2D div=[%d,%d] grading=[1,1]" % (xnodes, xnodes)
    grid_str = String(grid_str)
    a.menu.set(String("gridfile"), grid_str)
    time_str = "dt = %e t in [0,%e]" % (dt, T)
    time_str= String(time_str)
    a.menu.set(String("time parameters"), time_str)
    a.solver.scan()
    a.solver.solveProblem()
    a.solver.resultReport()
    return

class A:
    def __init__(self):
        self.menu = MenuSystem.MenuSystem()
        self.solver = Heat1()
        self.menu.init(String("Heat1 menu"), String("Heat1Menu"))
        self.solver.define(self.menu)

if __name__=='__main__':
    a = A()
    T = 2.0
    dt =pow(2,-6)
    dx = pow(2, -2)
    run(dt,T,dx,a)
    a.menu.thisown = 0
    a.solver.thisown = 0
```

**Listing B.3: timeloop.py**

```
#!/usr/bin/env python
"""
```

```
"""
Program to check if I can implement the timeLoop function belonging to Heat1 in
    Python
"""
from TimePrm import *
from SaveSimRes import *
from Menu import *
from DpString import *
from Heat1 import *

#database SaveSimRes
#tip TimePrm

class A():
    def __init__(self):
        self.solver = Heat1()
        self.menu = MenuSystem.MenuSystem()
        self.menu.init(String("Heat1 menu"), String("Heat1Menu"))
        self.solver.define(self.menu)

    def run(self,dt, T, dx):
        xnodes=int(1./dx)
        grid_str="P=PreproBox | d=2 [0,1]x[0,1] | d=2 e=ElmB4n2D div=[%d,%d]
            grading=[1,1]" % (xnodes, xnodes)
        grid_str = String(grid_str)
        self.menu.set(String("gridfile"), grid_str)
        time_str = "dt = %e t in [0,%e]" % (dt, T)
        time_str= String(time_str)
        self.menu.set(String("time parameters"), time_str)
        self.solver.scan()
        #Local solveProblem function
        self.solveProblem()
        self.solver.resultReport()
        return

    def timeLoop(self):
        tip=self.solver.tip()
        tip.initTimeLoop()
        self.solver.setIC()
        self.solver.database().dump(self.solver.u(),tip,"initial condition")
        while tip.finished()==False:
            tip.increaseTime()
            self.solver.solveAtThisTimeStep()
            self.solver.u_prev = self.solver.u
        return

    def solveProblem(self):
        self.timeLoop()
        return


if __name__=='__main__':
    a = A()
    T = 2.0
```

```
    dt = pow(2,-6)
    dx = pow(2, -2)
    a.run(dt,T,dx)
    a.menu.thisown=0
    a.solver.thisown=0
```

**Listing B.4: solveAtThisTimestep**

```
#!/usr/bin/env python
"""
Program to check if I can implement the solveAtThisTimeStep function belonging to
   Heat1 in Python
"""
from Menu import *
from Heat1 import *
from DpString import *
from TimePrm import *
from SaveSimRes import *
from DegFreeFE import *
from LinEqAdmFE import *
from FieldSummary import *
from ErrorNorms import *
from FieldFE import *
#database SaveSimRes
#tip TimePrm
#u FieldFE
#u_summary FieldSummary
#dof DegFreeFE
#lineq LinEqAdmFE

#u FieldFE
#flux FieldsFE

class A():
    def __init__(self):
        self.solver = Heat1()
        self.solver.thisown=0
        self.menu = MenuSystem.MenuSystem()
        self.menu.thisown=0
        self.menu.init(String("Heat1 menu"), String("Heat1Menu"))
        self.solver.define(self.menu)
        return

    def run(self,dt, T, dx):
        xnodes=int(1./dx)
        grid_str="P=PreproBox | d=2 [0,1]x[0,1] | d=2 e=ElmB4n2D div=[%d,%d]
           grading=[1,1]" % (xnodes, xnodes)
        grid_str = String(grid_str)
        self.menu.set(String("gridfile"), grid_str)
        time_str = "dt = %e t in [0,%e]" % (dt, T)
        time_str= String(time_str)
        self.menu.set(String("time parameters"), time_str)
```

```
        self.solver.scan()
        #Local solveProblem function
        self.solveProblem()
        self.solver.resultReport()
        return self.solver.L2_error

    def timeLoop(self):
        tip=self.solver.tip()
        tip.initTimeLoop()
        self.solver.setIC()
        self.solver.database().dump(self.solver.u(),tip,"initial condition")
        while tip.finished()==False:
            tip.increaseTime()
            #local solveAtThisTimeStep function
            self.solveAtThisTimeStep()
            self.solver.u_prev = self.solver.u
        return


    def solveAtThisTimeStep(self):
        self.solver.fillEssBC()
        self.solver.makeSystem(self.solver.dof(),self.solver.lineq())
        u = self.solver.u()
        linsol=self.solver.linsol
        tip = self.solver.tip()
        flux = self.solver.flux()
        self.solver.dof().field2vec(u, linsol)
        self.solver.lineq().solve()
        print "t=%f" % tip.t
        self.solver.dof().vec2field(linsol, u)
        print
        self.solver.database().dump(u, tip, "some comment if desired...")
        self.solver.u_summary.update(tip.time())
        self.solver.makeFlux(flux, u)
        self.solver.database().dump(flux, tip, "smooth flux -k*grad(u)")
        return


    def solveProblem(self):
        self.timeLoop()
        return


if __name__=='__main__':
    a = A()
    T = 2.0
    dt = pow(2,-6)
    dx = pow(2, -2)
    L2 = a.run(dt,T,dx)
```

## B.3 Pulse testscripts

**Listing B.5: test.py**

```
#!/usr/bin/env python
from testisotonic import *
from testisometric import *
from pulse_heart.Heart import *
from pulse_heart.Cells import *
from pulse_mechanics.Myocardium import *
#for access to the diffusion variable in solveCellsAndDiffusion()
from pulse_diffusion.DiffusionMG import *
from TimePrm import *
import subprocess
import sys
import datetime
import shutil


def readFile(Aobject):
    readfile = open(Aobject.infile, "r")
    for line in readfile:
        splitline = line.split()
        if (len(splitline)!=0) and (splitline[0][0]!="!"):
            if splitline[0] =="set":
                Aobject.menu.forceAnswer(String(line[3:]))
            elif splitline[0] =="ok":
                Aobject.menu.forceAnswer(String(line[0]))
    readfile.close()
    return

def solveCellsAndDiffusion(Aobject):
    heart = Aobject.simulator.heart
    tip_diff = heart.tip_diff()
    tip = heart.tip()

    while tip_diff.t < tip.t:
        dt_diff = tip_diff.Delta()
        t0_diff = tip_diff.time()
        print "Solving cell model ODEs..."
        heart.cells.solveAtThisTimeStep(t0_diff,dt_diff)
        print "...done with cell step."
        print "Solving diffusion equations..."
        heart.diffusion(1)().solveAtThisTimeStep()
        print "...done with diffusion step."
    return

def timeLoop(Aobject):
    """
    Python implementation of Circulation.timeLoop
    """
    solver = Aobject.simulator
    tip = solver.tip()
    heart = solver.heart
```

```
    tip.initTimeLoop()
52  solver.dt = tip.Delta()
    save_dt = 0.0
54  if (heart.dynamicBC()):
        heart.initTimeLoop(solver.p_cav, solver.restart_time)
56      solver.V_heart = heart.getVheart()
        while (not tip.finished()):
58          print "Global time is now: " + str(tip.t)
            print "----------- NEW STEP ---------------"
60
            tip.increaseTime()
62          solveCellsAndDiffusion(Aobject)
            try:
64              solver.solveMechDynamic()
            except:
66              print "Convergence failure at t = " + str(tip.t) + ", exiting
                    program!"
                sys.exit(1)
68
            solver.dt = tip.Delta()
70          heart.updateAndSaveMechVars(solver.dt, save_dt)
            solver.writePVfiles(tip.time(), solver.p_cav ,solver.V_heart)
72      print"Succesfully completed Circulation:: timeLoop()"

74  else:
        #simplified BCs, no circulation model:
76      #solver.heart.simpleTimeLoop()
        simpleTimeLoop(Aobject)
78  return

80 def simpleTimeLoop(Aobject):
    save_dt = 0.0
82  solver = Aobject.simulator
    tip = solver.tip()
84  heart = solver.heart
    dummy2 = 0.0
86  dummy = 1
    #heart.initTimeLoop(dummy, dummy2)
88  heartInitTimeLoop(Aobject)
    while (not tip.finished()):
90      print "Global time is now: %d " % tip.t
        print "----------- NEW STEP ---------------"
92      tip.increaseTime()
        solveCellsAndDiffusion(Aobject)
94      try:
            heart.solveMechSimple();
96      except:
            print "Convergence failure at t = %d, exiting program!" % tip.t
98      heart.updateAndSaveMechVars(tip.Delta(), save_dt)
    print "Successfully completed Heart::timeLoop."
100 return

102
```

```
def heartInitTimeLoop(Aobject):
    cavity_pressure = 1
    restart_time = 0.0
    solver = Aobject.simulator
    heart = solver.heart
    mech = heart.mechanics()
    tip = heart.tip()
    tip.initTimeLoop()
    tip_diff = heart.tip_diff()
    tip_diff.initTimeLoop()
    heart.p_cav = cavity_pressure
    while (tip.t < restart_time):
        tip.increaseTime()
        while (tip_diff.t < tip.t):
            tip_diff.increaseTime()

    if heart.diffusion.size() == 1:
        heart.diffusion(1)().attachIntegrands()
    else:
        print "Heart:: initTimeLoop Diffusion has wrong size - should be 1"
        sys.exit(1)

    heart.diffusion(1)().initBlocks()
    heart.diffusion(1)().makePreconditioner()
    a = heart.diffusion(1)().getNoOfGrids()
    heart.diffusion(1)().getDataStructureOnGrid(a)

    if heart.dynamic_mech_BC:
        try:
            heart.V_heart = mech.computeNewVolume(heart.p_cav, 0.0)
            mech.dumpDeformedGridAndFibers()
            #print "..Initial Conditions, p = %d  V =%d"  %(heart.p_cav, heart.
                V_heart)
        except:
            print "Convergence failed for initial cavity pressure, exiting."
            sys.exit(1)

    else:
        heart.solveMechSimple()
    mech.saveResults()
    mech.computeLambda()
    mech.updateActivationFields()
    mech.dumpTrackPoints()
    if (heart.moving_bidomain):
        heart.diffusion(1)().updateDeformedConds(mech.getInverseCField())
    return

def run(Aobject):
    readFile(Aobject)
    Aobject.simulator.scan()
    timeLoop(Aobject)
    return

```

```
if __name__=="__main__":
    try:
        if sys.argv[1]=="isometric":
            t = Isometric()
            run(t)
            print dir(t)
            args = "mv *.grid *.track Nodemap isometric"
            subprocess.Popen(args, shell=True)

        elif sys.argv[1]=="isotonic":
            t = Isotonic()
            run(t)
            print dir(t)
            args = "mv *.grid *.track Nodemap isotonic_large"
            subprocess.Popen(args, shell=True)
    except:
        print "Missing input, isometric,isotonic or canine"
```

**Listing B.6: testisometric.py**

```
#!/usr/bin/env python
from pulse_heart.Circulation import *
from Menu import *
from DpString import *

class Isometric():
    def __init__(self):
        self.menu = MenuSystem.MenuSystem()
        self.menu.thisown = 0
        self.main()
        return

    def __del__(self):
        del self.infile
        return


    def main(self):
        self.menu.init(String("Isometric"), String("isometric test"))
        casename = "testbox"
        cellmodel = "WinslowRice"
        self.infile = "isometric_all.i"
        cmd = String(cellmodel)
        mech_model = "TransverseIso"
        self.simulator = Circulation(cmd, String("Something"), String(mech_model)
            )
        self.simulator.thisown = 0
        self.simulator.define(self.menu)
        menu = self.menu
        menu.initFromCommandLineArg("--cmd", cmd, cellmodel)
        menu.initFromCommandLineArg("--casename", String(casename), casename)
        return
```

**Listing B.7: testisotonic.py**

```
#!/usr/bin/env python
from pulse_heart.Circulation import *
from Menu import *
from DpString import *

class Isotonic():
    def __init__(self):
        self.menu = MenuSystem.MenuSystem()
        self.menu.thisown = 0
        self.main()
        return
    def __del__(self):
        del self.infile
        return

    def main(self):
        casename = "testbox"
        cellmodel = "WinslowHMT"
        self.infile = "isotonic_all.i"
        self.menu.init(String("Isotonic"), String("isotonic test"))
        cmd = String(cellmodel)
        mech_model = "TransverseIso"
        self.simulator = Circulation(cmd, String("Something"), String(mech_model)
            )
        self.simulator.thisown = 0
        self.simulator.define(self.menu)
        menu = self.menu
        menu.initFromCommandLineArg("--cmd", cmd, cellmodel)
        menu.initFromCommandLineArg("--casename", String(casename), casename)
        return
```

## B.4 Compilation and setup scripts

**Listing B.8: PW.conf: setup for the Pulsewrap package**

```
#Source this file in your .bashrc file to get the correct configuration of
    PulseWrap. Read through so you are sure they are set correctly.

#Make sure that you have installed Diffpack, SWIG and Pulse correctly first,

#PulseWrap: insert path to the directory where you have saved the PulseWrap
    directory.
export PULSEWRAPDIR=$HOME/.../PulseWrap
#Input the correct path to your local installation of Pulse in PULSEDIR, and make
    sure that pulse.conf is correct.
export PULSEDIR=$HOME/.../pulse-cmake
```

```
# For SuperLU files in Pulse
#Make sure that your SuperLU directory is called superlu, and give the path to
#the directory where you have installed it.
export SUPERLUDIR=/path/to/dir/superlu
#For SWIG
#Insert path to directory containing SWIG.
export SWIGSRC=$HOME/.../swig-2.0.9

#Python: You may have to export the path to your PYTHON directory.
#export PYTHONPATH=/usr/lib/python2.6:$PYTHONPATH
#export C_INCLUDE_PATH=/usr/lib/python2.6:$C_INCLUDE_PATH
#export CPLUS_INCLUDE_PATH=/usr/lib/python2.6:$CPlUS_INCLUDE_PATH


#NOR is a Diffpack environment constant.
export CPLUS_INCLUDE_PATH=$NOR/md/include:$CPLUS_INCLUDE_PATH
export C_INCLUDE_PATH=$NOR/md/include:$C_INCLUDE_PATH
export PYTHONPATH=$NOR/md/include:$PYTHONPATH

export CPLUS_INCLUDE_PATH=$NOR/dp/include:$CPLUS_INCLUDE_PATH
export C_INCLUDE_PATH=$NOR/dp/include:$C_INCLUDE_PATH
export PYTHONPATH=$NOR/dp/include:$PYTHONPATH

export CPLUS_INCLUDE_PATH=$NOR/md/src/libs/multilevel/hidden:$CPLUS_INCLUDE_PATH
export C_INCLUDE_PATH=$NOR/md/src/libs/multilevel/hidden:$C_INCLUDE_PATH
export PYTHONPATH=$NOR/md/src/libs/multilevel/hidden:$PYTHONPATH


#For Pulse
source $PULSEDIR/build/pulse.conf
export CPLUS_INCLUDE_PATH=$PULSEDIR/local/include/:$CPLUS_INCLUDE_PATH
export C_INCLUDE_PATH=$PULSEDIR/local/include/:$C_INCLUDE_PATH
export PYTHONPATH=$PULSEDIR/local/include/:$PYTHONPATH
export CPLUS_INCLUDE_PATH=:$SUPERLUDIR:$CPLUS_INCLUDE_PATH
export C_INCLUDE_PATH=:$SUPERLUDIR:$C_INCLUDE_PATH
export PYTHONPATH=:$SUPERLUDIR:$PYTHONPATH

#For PulseWrap
export PYTHONPATH=$PULSEWRAPDIR/src:$PYTHONPATH
export PATH=$PULSWRAPDIR/src:$PULSEWRAPDIR

export PYTHONPATH=$PULSEWRAPDIR:$PYTHONPATH
export CPLUS_INCLUDE_PATH=$PULSEWRAPDIR:$CPLUS_INCLUDE_PATH
export C_INCLUDE_PATH=$PULSEWRAPDIR:$C_INCLUDE_PATH
export CPLUS_INCLUDE_PATH=$PULSEWRAPDIR/include:$CPLUS_INCLUDE_PATH
export C_INCLUDE_PATH=$PULSEWRAPDIR/include:$C_INCLUDE_PATH
export PYTHONPATH=$PULSEWRAPDIR/include:$PYTHONPATH
```

**Listing B.9: install.py: compile the PulseWrap package**

```
#!/usr/bin/env python
```

```
import os, subprocess, shutil
"""
The installationscript for PulseWrap.
Do what the README file says, then run this script with
>>> python install.py
"""


def install(pathname):
    """
    root: current path
    folders: directories in root
    filenames: files in root of type other than dir.
    """
    paths = [pathname+"/Diffpack", pathname+"/Pulse"]
    for path in paths:
        print path
        for root, folders, filenames in os.walk(path):
            filepath = root + "/"
            if "Makefile" in filenames and ".cmake2" in filenames:
                #Compile the C++ files
                subprocess.call(["Make", "MODE=opt"], cwd=filepath)
            elif "Makefile" in filenames:
                #Compile the Python/Swig files
                subprocess.call(["./configure.sh"], cwd=filepath)
                fix = "fixSwigMakefile.py"
                subprocess.call([fix], cwd=filepath)
                subprocess.call(["Make"], cwd=filepath)
            #Copy files into include and src
            for fil in filenames:
                if fil.endswith("py") or fil.endswith("so"):
                    if (root.split("/")[-3])=="SimRes":
                        shutil.copy(filepath + fil, pathname + "/src/SimRes")
                    elif (root.split("/")[-3])=="Menu":
                        shutil.copy(filepath + fil, pathname + "/src/Menu")
                    elif (root.split("/")[-3])=="heart":
                        shutil.copy(filepath + fil, pathname + "/src/pulse_heart")
                    elif (root.split("/")[-3])=="cellmodel":
                        shutil.copy(filepath + fil, pathname + "/src/
                            pulse_cellmodel")
                    elif (root.split("/")[-3])=="diffusion":
                        shutil.copy(filepath + fil, pathname + "/src/
                            pulse_diffusion")
                    elif (root.split("/")[-3])=="ode":
                        shutil.copy(filepath + fil, pathname + "/src/pulse_ode")
                    elif (root.split("/")[-3])=="mechanics":
                        shutil.copy(filepath + fil, pathname + "/src/
                            pulse_mechanics")
                    else:
                        shutil.copy(filepath + fil, pathname + "/src")
                if fil.endswith("i") or fil.endswith("h"):
                    shutil.copy(filepath + fil, pathname + "/include")
```

```
52
        return
54
56 if __name__=='__main__':
        pathname = os.getcwd() #current working directory
58      install(pathname)
```

# Bibliography


[1] *Boost.Python*. URL: http://www.boost.org/doc/libs/1_54_0/libs/python/doc/index.html (visited on 09/04/2013).

[2] *Diffpack Kernel and Toolboxes Documentation 4.0.00*. URL: http://diffpack.com/diffpack/refmanuals/current/index.html (visited on 09/11/2013).

[3] Python Software Foundation. *Modules*. 2013-08-29. URL: http://docs.python.org/2/tutorial/modules.html (visited on 08/29/2013).

[4] inuTech GmbH. *Diffpack 4.0.11 Kernel and Toolboxes Documentation*. 2012. URL: http://inutech.de/diffpack_reference/ (visited on 08/06/2013).

[5] inuTech GmbH. *Diffpack Technical Summary*. URL: http://www.diffpack.com/products/what_is_dp.html (visited on 08/09/2013).

[6] Hans Petter Langtangen. *Computational Partial Differential Equations: Numerical Methods and Diffpack Programming*. 1st ed. Springer, 2003.

[7] Hans Petter Langtangen. *Python Scripting for Computational Science*. 3rd ed. Springer, 2009.

[8] Hans Petter Langtangen and Aslak Tveito. *Advanced Topics in Computational Partial Differential Eqution: Numerical Methods and Diffpack Programming*. Springer, 2003.

[9] *Operator Renaming for Swig*. 2011. URL: http://scion.duhs.duke.edu/vespa/gamma/wiki/SwigOperatorRenaming (visited on 08/06/2013).

[10] *PyCXX*. URL: http://cxx.sourceforge.net/ (visited on 09/04/2013).

[11] *SWIG Users Manual*. 1997. URL: http://www.swig.org/Doc1.1/PDF/SWIGManual.pdf (visited on 08/07/2013).

[12] SWIG. *Swig-2.0 Documentation*. 2013. URL: http://www.swig.org/Doc2.0/SWIGDocumentation.pdf (visited on 08/06/2013).

[13] *Shebang (Unix)*. 2013. URL: http://en.wikipedia.org/wiki/Shebang_(Unix) (visited on 08/29/2013).

[14] AsbjÃ¸rn Reglund Thorsen. "Ruby-grensesnitt til Diffpack og analyse av kompilator-opsjoner ved bruk av modifisert ACOVEA." MA thesis. The University of Oslo, 2005.

[15] Magne Westlie. "Utvikling av et Python grensesnitt til Diffpacks C++ biblioteker." MA thesis. The University of Oslo, 2004.

[16] cplusplus.com. *C++ Language Tutorial: Friendship and inheritance*. 2000-2013. URL: http://www.cplusplus.com/doc/tutorial/inheritance/ (visited on 08/09/2013).