A Python Framework for Verifying Codes for Numerical Solutions of Partial Differential Equations

Ola Skavhaug[*] Kent-Andre Mardal[†]
Hans Petter Langtangen[‡]

Abstract

We present a Python framework for applying the method of manufactured solutions (MMS) to verify software for solving partial differential equations. The user can, interactively or in a script, specify mathematical expressions for the solution and the PDE, and the framework modifies the simulator according to the new analytical solution. The verification framework applies to C, C++, and Fortran simulators, and effort has been made to make it both general and easy to use. We show three examples of code verification for PDE simulators written in C++. In these examples, the true errors of the numerical solutions are computed. We also discuss the constructs needed to use the Python verification framework with existing PDE simulators.

1 Introduction

Numerical solutions of partial differential equations (PDEs) modelling physical phenomena are increasingly being used in science and engineering. This puts high demands on the reliability and accuracy of the computed solutions. Bugs in the software, or the use of numerical algorithms outside their well defined scope, may be critical and cost huge amounts of money. Roache has for many years advocated the need for extensive verification and validation, see e.g. [11]. In his experience, the Method of Manufactured Solutions (MMS) is a simple and powerful technique to debug a simulation code. Salari and Knupp [6] have systematically used the MMS to investigate a series of common bugs. They found that most of the bugs are detected because their presence violates the order accuracy of the manufactured solution for the numerical methods in question.

[*]Dept. of Scientific Computing, Simula Research Laboratory and Dept. of Informatics, University of Oslo. Email: skavhaug@simula.no.
[†]Dept. of Scientific Computing, Simula Research Laboratory and Dept. of Informatics, University of Oslo. Email: kent-and@simula.no.
[‡]Dept. of Scientific Computing, Simula Research Laboratory and Dept. of Informatics, University of Oslo. Email: hpl@simula.no.

The scope of this paper is to present a high–level software platform for verifying[1] numerical simulators for PDE problems. Furthermore, particular emphasis is put on the verification of legacy code by integrating them in such a framework, with minimal modification of the existing code. To our knowledge most verification tools developed are closely linked to an application or library. We advocate the need for a general framework for comparing and verifying simulation codes in a wide range of application areas. The framework must be sufficiently easy to use such that the MMS is applied regularly; although the MMS is widely known, its use is limited according to our experience, because it requires too tedious modifications of the simulator software.

We have chosen to implement the MMS framework in the Python scripting language[2]. We emphasize that existing simulators written in Fortran, C or C++ can be integrated into this Python framework. This is often easy thanks to interfacing tools such as SWIG [3] and F2PY [10]. In this project, we have combined the two C++ libraries Diffpack [5] and GiNaC [2], as well as Fortran and C code, with Python. In fact, we find it simpler to combine Fortran and C++ simulators in Python (with SWIG and F2PY) than directly in the native languages.

This paper is outlined as follows. We start in Section 2 by explaining the basics of the Method of Manufactured Solutions as described by Roache [11]. Then, in Section 3, we present Famms (Fully automatic method of manufactured solutions) [12]; our interactive Python framework for automatic code verification. In Section 4, we apply Famms to three PDE simulators. The module `Symbolic`, used to express analytical solutions and PDEs symbolically is discussed in Section 5, and we make some concluding remarks in Section 6. Four examples of how to implement the constructs needed to evaluate Python functions from compiled PDE simulators are discussed in Appendix A–D.

2 The Method of Manufactured Solutions

We start with a brief review of the Method of Manufactured Solutions, which is a simple, yet powerful tool to construct known solutions to PDE problems. The knowledge of the analytical solution can be used to check that the error in the numerical solution satisfies the properties posed by the numerical methods used.

In the MMS, an additional source term is inserted into the model description, such that an analytical solution of the problem is known. Solving the new PDE problem, both the numerical and analytical solution are available, and the error can easily be computed and analyzed.

Consider a PDE problem on the form,

$$F(u) = 0, \tag{1}$$

[1]In computational science and engineering, program verification refers to empirical testing of software, for instance by the MMS. We remark that this meaning of program verification is different from the meaning of this term in computer science.

[2]The reasons for this decision will be explained later.

with appropriate boundary and initial conditions. Here, F may be a system of non–linear differential operators, and u an unknown system of scalar and vector functions. For example, we can easily rewrite the heat equation,

$$\frac{\partial u}{\partial t} = \nabla \cdot (k \nabla u) + f, \tag{2}$$

to fit into this formulation by defining:

$$F(u) = \frac{\partial u}{\partial t} - \nabla \cdot (k \nabla u) - f\,. \tag{3}$$

The MMS is based on choosing an analytical solution, v, such that $u = v$ is the solution of

$$F(u) = F(v). \tag{4}$$

The expression $F(v)$ can be computed *apriori* and used as an additional source term in the original problem specification. For example, the modified heat equation reads:

$$\frac{\partial u}{\partial t} = \nabla \cdot (k \nabla u) + f + F(v)\,. \tag{5}$$

Hence, instead of solving (1), we add a source term $F(v)$ to the PDE and solve (4), since we know the solution of the latter problem. Note that the source term $F(v)$ is a function of x and t since v is known.

From a computational point of view, we calculate $F(v)$ analytically and add the result as extra source terms in the equations. The boundary and initial conditions must be adjusted accordingly. For example, we may apply Dirichlet conditions $u = v$ on the whole boundary when solving (4), and use v at $t = 0$ as the initial condition. Using the MMS in the heat equation example and choosing $v = \exp(-t)\sin(xy)$, $k = 1$, and $f = 0$, we get the following source term:

$$F(v) = \exp(-t)\sin(yx)(x^2 + y^2 - 1)\,. \tag{6}$$

The new problem definition then reads

$$\frac{\partial u}{\partial t} = \nabla^2 u) + \exp(-t)\sin(yx)(x^2 + y^2 - 1)\,. \tag{7}$$

With a known solution it is possible to check if the simulator is *order–accurate*. For many numerical methods for PDEs we expect the following error estimate to be valid within the asymptotic regime,

$$\|v - v_h\| \leq C h^{\alpha} + D(\Delta t)^{\beta}. \tag{8}$$

Here, the parameters h and Δt are characteristic for the resolution of the discretization in space and time, respectively, and α and β depend on the discretization methods. The order accuracy is checked by running a sequence of decreasing values of h and Δt, computing the error in each run, fitting α and β in (8), and comparing the fitted values with the theoretically expected estimate for the numerical method in question.

The heat equation above is a simple toy model used to illustrate how the MMS works. A software framework to handle this problem may seem like overkill. In fact, cutting and pasting source code from Maple or Mathematica would probably be both easier and faster. However, later in Section 4.3 we will consider a coupled heat and fluid flow problem that models the extrusion of metal or the flow of highly viscous polymers. In this problem there are more than 10 variables depending on each other. Moreover, the models for these variables are uncertain and are typically varied during the investigation of a concrete application. In such situations it is desirable to have an interactive Python interface with MMS in the debugging–verification–validation cycle.

3 Famms

An automatic code verification framework must be easy to use, with high–level syntax and semantics. Also, such a framework must be able to tackle quite general PDE models; scalar and vector equations, as well as coupled systems. We have developed Famms, a Python framework for applying the MMS to PDE solvers, with these design goals in mind.

Code verification using the MMS is usually conducted in several, individual steps. Typically, the PDE model and the analytical solution are specified in a language dedicated to symbolic manipulation, such as Maple or Mathematica. After computing the source term, it is saved to file together with the solution as either Fortran or C code. The source code must in turn be compiled and linked with the PDE simulator manually. Each time a new analytical solution is chosen, these steps are repeated. The resulting MMS setup, if used in the first place, may soon become static, with only a few choices for analytical solutions. Another drawback with the traditional way of applying the MMS is that the syntax and semantics of modern symbolic manipulation tools are quite complicated. Especially, we find them less clear and intuitive than the Python language. (Having said this, Maple and Mathematica are powerful tools, not limited to computing source terms needed for code verification.)

In Famms, we glue all the parts of the MMS into one framework. The main components in the framework are: (i) a symbolic manipulation module for specifying solutions, PDE models, and source terms using the Python language, (ii) a PDE simulator accessible from Python as an *extension module*, and (iii) constructs for swapping functions written in the compiled language with Python functions. In Figure 1, we show how the components in a Famms session work together. These components are described in more detail in the following sections. In the technical note, we state the demands to the simulator to be verified.

In a typical Famms session, we specify the PDE and the analytical solution symbolically in a Python script. Also, the simulator, which we assume is equipped with a Python interface, is created and initialized from the script. The simulator and the symbolic representations of the PDE model and solution are then assigned to a Famms object. In this assignment, the new source term is

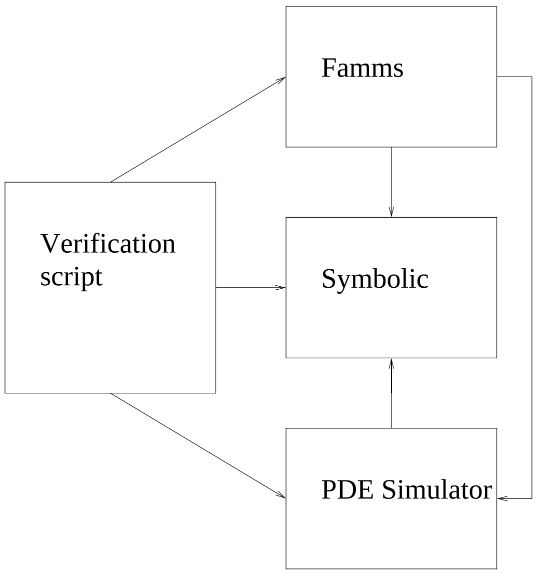

Figure 1: An overview of the various components involved when using Famms to verify a PDE simulator.

computed analytically and inserted as a callback in the simulator together with the solution. Later, when simulations are run from the script, the solution is known and the error in the numerical solution can be computed and processed further.

We have chosen Python as the high–level coding environment for several reasons. First, Python has a clean and very readable syntax close to that of Matlab, a popular scripting language for scientific programming. Therefore, learning Python is fairly easy. Being platform independent and having an extensive collection of both general purpose and scientific modules, Python serves as a well suited environment for high–level scientific programming in general. A third argument for choosing Python is that it has a mature C API (Application Programming Interface), used when extending Python with modules written in compiled languages. Also, the Python/C API can be used for creating interfaces to existing PDE solvers written in C/C++ and Fortran. We refer to [8] for a discussion of scripting languages as a complement to compiled languages.

3.1 Specifying the PDE Model in Python

In the MMS, we choose an analytical solution, insert this into the PDE, and fit an additional source term such that the chosen solution fulfills the modified PDE. Famms therefore relies on a Python module able to do symbolic manipulation. We have made the module `Symbolic` for this task. Here, we can specify PDE models in a convenient manner, with high–level mathematical operators

Technical Note

Famms Assumptions

In order to apply Famms to a PDE simulator, the implementation of the simulator must follow some rules. Especially, the implementation of source terms, boundary conditions, and initial conditions must follow some conventions. We have made the following assumptions:

- Source terms, boundary conditions, and initial conditions must be placed in functions (or methods). If these terms are hard–coded into the discretization schemes, Famms can not be applied.
- At the run–time of a simulator, it must be possible to change which function to call, e.g., by accessing functions through function pointers or by using function objects. Functions accessed directly can not easily be replaced at run–time, and Famms needs to do this.
- The PDE simulator must be split into functions for initializing and running a simulation. If the simulator is implemented as a single function performing both the initialization and the simulation, Famms may not be able to alter the PDE model specification according to the new analytical solution.

Famms has been applied to C++ simulators using the Diffpack library. These simulators fulfill the above assumptions.

such as divergences and gradients working on scalar and vector functions. The `Symbolic` module is only a thin layer on top of a full–fledged symbolic engine: GiNaC. This engine is implemented in C++ and available as open source code.

The setup for computing new source terms in `Symbolic` is simple: We define a Python function `F(u)`, representing the PDE model $F(u) = 0$. This function returns the new source term corresponding to the chosen solution v. For example, if we consider the heat equation (2),

$$F(u) = \frac{\partial u}{\partial t} - \nabla \cdot (k \nabla u) - f = 0,$$

the Python function may take the form:

```
def F(u):
  return u.diff(t,1) - div(k*grad(u)) - f
```

Here, `u.diff(t,1)` computes the derivative of `u` with respect to time, and `div` and `grad` computes the divergence and gradient, respectively. Constructing a solution `v`, e.g., `v = exp(-t)*sin(x*y)`, the new source term, `rhs`, is obtained by passing the solution to the function `F`, `rhs = F(v)`. The variable `rhs` now holds the auxiliary source term $F(v)$ as a symbolic expression.

3.2 PDE Simulators as Python Extension Modules

In Famms, the analytical solution and corresponding source term exist as Python objects only. Since a PDE simulator needs to access these objects to solve the modified PDEs, the application has to communicate with Python. The way to solve this problem is to extend the simulator with scripting capabilities, turning the PDE simulator into a Python extension module. The Python/C API, offering access to most of the Python run–time system, has been made to facilitate the job of mixing compiled applications and Python.

In most cases, we do not interface the entire PDE simulator from Python. The functions we need to reach typically initialize the simulator and run the computations. Variables holding important data such as the solution, the model parameters, and numbers related to the numerical methods used are usually interfaced as well.

Having chosen the parts of the simulator to interface from Python, the wrapper code is written using the Python/C API. When the interface is small, this job can be done by hand. However, for huge PDE simulators, writing the wrapper code manually can be both time–consuming and error–prone. We can instead utilize tools that automatically does this job, e.g., SWIG [3], Boost [1], and SIP [14] for C/C++, and F2PY [10] for Fortran. Describing in detail how these tools work is beyond the scope of this text. However, most of these tools are well documented and easy to learn.

The last stage is to compile and link the wrapper code with the simulator, forming a shared library file that Python can import as a module.

Having access to the main functionality of a PDE simulator from Python, we easily set up and run simulations in simple Python scripts. For instance, we can run numerical experiments where important model parameters are varied, information is collected, and results are saved in tabular form for later use.

3.3 Calling Python Functions from Low–Level Code

To apply Famms to a PDE simulator, Python functions for the symbolic expressions of the new source term and analytical solution must be callable from the application. In many cases this can be done without changing the source code of the simulator.

Using the Python/C API, we can write Python wrapper functions, having the same signature as the functions in the simulator. The job of these wrappers is to replace the native functions and call Python functions when invoked. More specifically, the arguments to a wrapper function are first converted to data types in Python and then passed on to a function defined in the scripting language. Replacing the native functions with the wrappers is done in Python scripts.

One of the assumptions Famms makes regarding a PDE simulator, is that source terms, boundary conditions, and initial conditions are placed in functions representing fields, or *field functions*. A field function takes a point in time and space as arguments and returns a scalar or vector value. Writing wrapper code for field functions is usually straight forward, because the input arguments are simple and therefore easy to convert to Python data types. However, the concrete implementation of a field function varies from one simulator to another. Therefore, we have only outlined how such wrappers can be constructed, and given some examples in Appendix A–D.

How do we replace a native function in a simulator with a Python function? If the application uses pointers to address functions, replacing a function is just a matter of moving a pointer. In C++, functions can be placed in objects, and we can replace the native function object with a version calling a function in Python. When a function is called directly, it can not be replaced. In this case, we have to modify the source code to replace the function. For PDE simulators written in Fortran, interfaced with F2PY, calling Python functions is very simple, because F2PY allows Python functions to replace subroutines automatically.

The Python field function wrappers usually must be written manually. The job is typically done once and the construct then provides a powerful extension of the Python interface to a PDE simulator: Whenever a simulator uses a native field function, it can be replaced by a Python field function.

3.4 Famms Overview

In Famms, the PDE model and desired solution are specified in Python by using the module `Symbolic`. Recall from Section 3.1 that the specification is given in terms of a function defining the PDE problem, and a symbolic expression for the solution. Suppose that a PDE simulator has been converted to a Python extension module. Then, what Famms basically does is to make the Python functions callable from the simulator.

Famms is a lightweight Python module, implementing a single class, `Famms`. A `Famms` object takes two arguments at initialization. The first is the number of space dimensions for the PDE, and the second is a boolean for specifying time dependent problems:

```
f = Famms(nsd=2, time=True)
```

The method `assign` is used to specify the setup of a MMS, and it takes the PDE model, solution, and the simulator as arguments:

```
f.assign(equation=F, solution=v, simulator=heat)
```

When assigning these objects to `Famms`, the new right–hand side and analytical solution of the PDE problem are automatically inserted into the simulator.

When applying MMS to a system of PDEs, the module `SystemFamms` can be used. It is built on top of `Famms`, and administers coupling of the various equations, solutions, and simulators. When constructing a `SystemFamms` instance, we must at least specify the number of equations in the coupled system and the maximum number of space dimensions:

```
f = SystemFamms(nproblems=2, max_nsd=2, simtype="DP")
```

Here, we specify a coupled system of two PDEs in two dimensions.

A method `assign` is again used to set up the MMS for the system. The arguments are three lists of simulators, solutions, and equations:

```
f.assign(sim_list=[momentum, energy], sol_list=[w, T], PDE_list=[F1, F2])
```

In this example, the two simulators `momentum` and `energy` solve the momentum equation and energy equation for a coupled fluid flow and heat problem having the analytical solutions `w` and `T`, respectively. `F1` and `F2` are Python functions specifying the PDE model, i.e., the momentum and energy equations. This simulator is treated in more detail in Section 4.3.

4 Examples

In this section, we show three examples of how to use Famms to verify PDE simulators. The simulators have first been turned into Python extension modules using SWIG. Starting with the heat equation used in the previous sections, we apply Famms to a scalar, time–dependent finite element solver, and show how the MMS can be used to investigate the true errors of the numerical solutions. In the following example, a simulator for a linear elasticity problem, utilizing a multigrid solver, is treated. The example shows an example of how Famms can be used to detect bugs. Especially, we use Famms to reveal a bug in the multigrid algorithm. The most challenging example is a solver for a coupled heat and fluid flow problem. This is a system of two non–linear PDEs. As an example, we test the iterative convergence of the non–linear solver as the degree of non–linearity is increased.

4.1 The Heat Equation

We consider a fully specified problem similar to the heat equation from Section 2:

$$\begin{aligned} \frac{\partial u}{\partial t} &= \nabla\cdot(k\nabla u) + f, \quad \text{in } \Omega, \\ u &= g, \quad \text{on } \partial\Omega, \\ u &= u_0 \quad \text{in } \Omega \text{ at } t = 0. \end{aligned}$$

Here u is the temperature in the domain Ω, k is the conductivity, f is a source term, and g is a known temperature at the boundary. One way of discretizing this problem is to use a finite difference for the time derivative and finite elements for the spatial derivatives.

The simulator for solving this problem is implemented as a Diffpack application and equipped with a Python interface extended with methods for inserting Python functions in place of the native functions for evaluating the source term, initial conditions, and boundary conditions. In Famms, a default naming convention is used for these extensions, to ease the verification of simulators following this standard. The convention is that the source term is attached to the simulator by calling the method `set_b_func`, and the analytical solution is inserted by calling `set_v_func`. The simulator is responsible for redirecting the function calls for setting boundary and initial conditions to the symbolic expression for the analytical solution.

We present a simple Famms script in detail below, containing less than thirty lines of code. The first part of the script creates and initializes of the heat simulator:

```
from Heat1 import Heat1 # Import simulator
from dputils import *   # Import Diffpack utilities
from Symbolic import exp, sin, cos, div, grad
from Famms import Famms

heat = Heat1();     # Create simulator
initSimulator(heat) # Initialize simulator
```

Above, the initialization of the simulator takes place in the function `initSimulator`, which is specific for Diffpack simulators. The next step is to create a Famms object, and symbolically specify the PDE model and analytical solution.

```
f = Famms(nsd=2, time=True) # Time dependent problem in 2D
[x, y] = f.x; t = f.t # Define some aliases
v = exp(-t)*(sin(2*x) + cos(2*y)) # Analytical solution
k = 1

def F(u):
    return u.diff(t,1) - div(k*grad(u))
```

To insert the analytical solution and source term into the simulator, we assign `F`, `v`, and `Heat1` to the Famms object. We then run a simulation and print the numerical error to screen:

```
f.assign(equation=F, solution=v, simulator=heat)
heat.solveProblem() # FE assebly, solve linear system

error = Error.Error() # Error computing utilities
print error.L_norm(heat.uanal(), heat.u(), time=heat.tip().time())
```

In the function `L_norm`, we compute the L_1, L_2, and L_∞ norms of the error of the numerical solution. The input arguments to this function are the analytical and numerical solution, and the time step.

Numerical Experiment

We want to inspect the error of the numerical solution from the heat equation simulator for different choices of finite elements, and use both piecewise bilinear and biquadratic finite elements is space. In time, we use a backward Euler finite difference scheme.

We use the menu system in the heat simulator to specify the experimental setup from Python. Especially, we set the solution domain, Ω, to the unit square in two dimensions, the conductivity is set to unity, and the source term, f, is dropped. The time interval is $(0, 1]$. We use a conjugate gradient method to solve the linear systems, with the convergence criterion $||r_k||_{L_2} \leq \epsilon = 10^{-10}$, where r_k is the residual after iteration k, is used for all simulations. The manufactured solution is $v = \exp(-t)(\sin(2x) + \cos(2y))$.

In Table 1 and Table 2 we show the L_2 error of the numerical solutions for two different choices of finite elements. In Figure 2 and Figure 3, we plot the lower row and right column of Table 1 and Table 2. For bilinear elements, we see quadratic convergence in space, and for biquadratic elements, the convergence rate is time is linear. To reveal the convergence rate in space for biquadratic elements, finer time stepping would be required, and similarly, we would need a finer grid to see the linear convergence rate in time for bilinear elements. Such behaviour is common: the discretization must be sufficiently fine to see the expected asymptotic convergence.

Table 1: L_2 errors of the numerical solution of the heat equation, using bilinear finite elements.

$\Delta t/h$	2^{-2}	2^{-3}	2^{-4}	2^{-5}
2^{-5}	$9.88 \cdot 10^{-3}$	$2.28 \cdot 10^{-3}$	$3.94 \cdot 10^{-4}$	$1.98 \cdot 10^{-4}$
2^{-6}	$1.00 \cdot 10^{-2}$	$2.41 \cdot 10^{-3}$	$5.05 \cdot 10^{-4}$	$8.16 \cdot 10^{-5}$
2^{-7}	$1.01 \cdot 10^{-2}$	$2.47 \cdot 10^{-3}$	$5.70 \cdot 10^{-4}$	$9.86 \cdot 10^{-5}$
2^{-8}	$1.01 \cdot 10^{-2}$	$2.51 \cdot 10^{-3}$	$6.03 \cdot 10^{-4}$	$1.26 \cdot 10^{-4}$

Table 2: L_2 errors of the numerical solution of the heat equation, using biquadratic finite elements.

$\Delta t/h$	2^{-2}	2^{-3}	2^{-4}	2^{-5}
2^{-5}	$1.71 \cdot 10^{-3}$	$3.63 \cdot 10^{-4}$	$3.19 \cdot 10^{-4}$	$3.20 \cdot 10^{-4}$
2^{-6}	$1.72 \cdot 10^{-3}$	$2.54 \cdot 10^{-4}$	$1.59 \cdot 10^{-4}$	$1.59 \cdot 10^{-4}$
2^{-7}	$1.73 \cdot 10^{-3}$	$2.24 \cdot 10^{-4}$	$8.20 \cdot 10^{-5}$	$7.93 \cdot 10^{-5}$
2^{-8}	$1.74 \cdot 10^{-3}$	$2.19 \cdot 10^{-4}$	$4.65 \cdot 10^{-5}$	$3.96 \cdot 10^{-5}$

4.2 Linear Elasticity

One commonly applied PDE model is the Navier's equation governing small deformations of an elastic structure. The equation reads

$$(\lambda + \mu)\nabla[\nabla \cdot \mathbf{u}] + \mu\nabla \cdot [\nabla \mathbf{u}] = 0, \tag{9}$$

where $\mathbf{u}$ is the displacement field, and λ and μ are parameters describing the elastic properties of the structure. For simplicity, we have omitted the inertia and body forces in Equation (9). The PDE is a vector equation, with the number of components coinciding with the number of space dimensions.

A simple Famms script for this problem is quite similar to that for the heat equation in the previous section, and starts with initializing the simulator:

```
from Elasticity1MG import Elasticity1MG
```

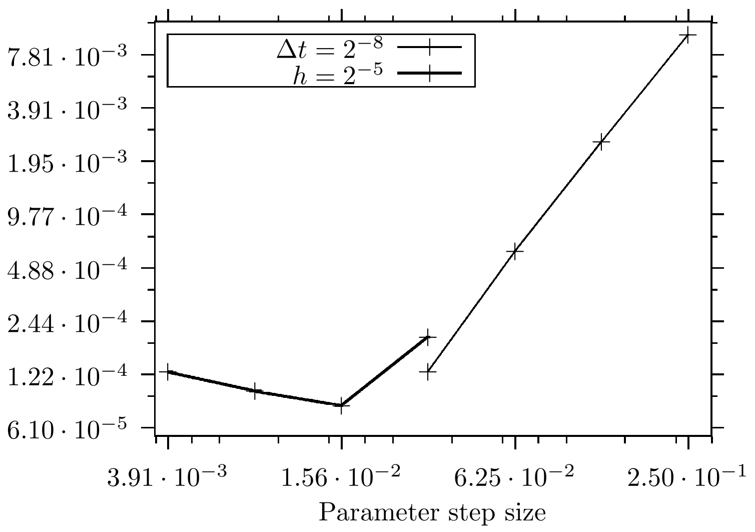

Figure 2: Logarithmic plot of the numerical errors of the heat simulations, when bilinear finite elements are used.

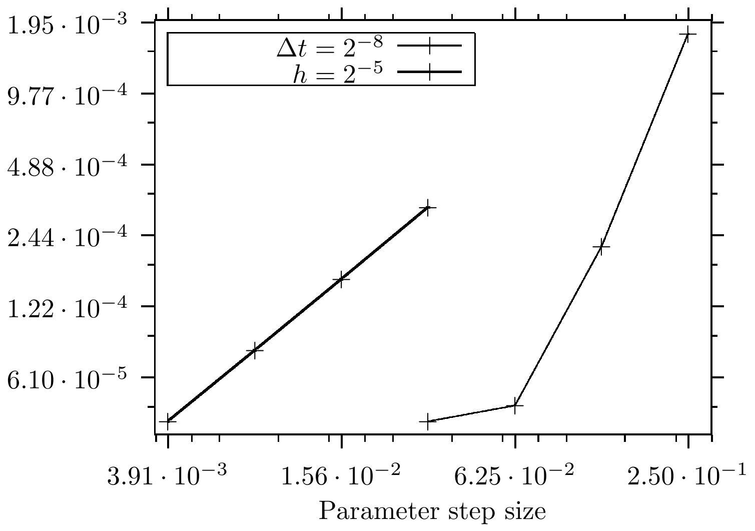

Figure 3: Logarithmic plot of the numerical errors of the heat simulations, when biquadratic finite elements are used.

```
from dputils.DP import MenuSystem
from Famms import Famms
from Symbolic import Vector, sin, cos, grad, div
from dputils import Error
```

```
m = MenuSystem()
m.init("Elasticity1MG", "Python interface"); m.thisown = 0

el = Elasticity1MG()
el.define(m)
el.scan()
```

The model depends on the parameters λ and μ, which we get from the simulator before specifying the model. We choose an analytical solution, and symbolically express the PDE using `Symbolic`. We also create a Famms object, and assign the model and simulator to it:

```
lambda_ = el.get_lambda(nu,E) # Get model parameters from
mu = el.get_mu(nu,E)          # the simulator

f = Famms(nsd=2)  # Problem in 2 space dimensions
[x, y] = f.x      # Define some aliases
v = Vector([sin(x), cos(y)]) # Analytical solution

def F(u):
   return -grad((lambda_ + mu)*div(u)) - div(mu*grad(u))

f.assign(equation=F, solution=v, simulator=el)
```

Now the MMS setup is complete, and we run a simulation and print the error:

```
el.solveProblem()

err = Error.Error()
print err.L_norm(el.uanal(), el.u())
```

The above Famms script is complete. When used for real simulations, however, the initialization of the simulator, involving specifying various model parameters, can be quite comprehensive.

Numerical Experiment

The simulator for the PDE model above uses a finite element method for discretization and a multigrid solver for the linear system of equations. The multigrid method is known to be robust. In fact, it often works even when implemented wrong. For instance, restrictions and interpolation operators can be made without references to boundary conditions, giving a multigrid method that in many cases works well. However, boundary conditions must be incorporated at all multigrid levels, according to theory. See [4, Chapter 4] for a general treatment of this topic. In the experiment below, we show an example where the multigrid interpolation and restriction operators are made both with and without references to the boundary conditions[3].

When the interpolation and restriction operators are without references to the boundary conditions, it results in a too small reduction of the error in each multigrid cycle. To debug the multigrid solver, we compute the L_2 error of the solution after each multigrid V–cycle sweep. Instead of changing the code in

[3]In fact, the bug uncovered in this example was present for several years in the original code.

the multigrid solver to compute this error, we add a Python callback function that is called after each multigrid sweep. This way, we can take care of the debugging process in Python.

We have used the manufactured solution, $v = (\sin(x+y), \cos(x-y))$, of the PDE problem, and the solution domain is the unit square in two dimensions.

The element basis functions are bilinear with two degrees of freedom at each node, and we use box elements. Starting with the coarsest mesh with nine nodes, each refinement is made by dividing the elements in four. We use six grid levels, which gives 4096 elements at the finest grid. The smoother in the multigrid solver is the symmetric successive over–relaxation (SSOR) method, and we carry out two pre– and post–smoothing sweeps at each grid level. As coarse grid solver we use Gaussian elimination. The convergence criterion is $\frac{||r_k||_{L_2}}{||r_0||_{L_2}} \leq \epsilon = 10^{-8}$, where r_k is the residual at the k-th multigrid iteration and r_0 is the initial residual.

Table 3 shows the L_2 error of the numerical solution for the simulator with the bug at the boundary after each multigrid sweep. In Table 4, we show the same error after the bug is fixed. The improvement of the multigrid algorithm by removing the bug in multigrid is significant; the number of iterations needed to meet the convergence criterion is reduced from 13 to 4. In Figure 4, we compare the convergence of the two multigrid algorithms.

Table 3: L_2 errors after each multigrid V–cycle iteration for the simulator with a bug.

Iteration	1	2	3	4
L_2 error	$2.62 \cdot 10^{-1}$	$8.04 \cdot 10^{-2}$	$2.69 \cdot 10^{-2}$	$9.37 \cdot 10^{-3}$
Iteration	5	6	7	8
L_2 error	$3.26 \cdot 10^{-3}$	$1.21 \cdot 10^{-3}$	$3.87 \cdot 10^{-4}$	$1.90 \cdot 10^{-4}$
Iteration	9	10	11	12
L_2 error	$3.99 \cdot 10^{-5}$	$6.35 \cdot 10^{-5}$	$4.35 \cdot 10^{-5}$	$4.98 \cdot 10^{-5}$
Iteration	13			
L_2 error	$4.74 \cdot 10^{-5}$			

Table 4: L_2 errors after each multigrid V–cycle iteration for the debugged simulator.

Iteration	1	2	3	4
L_2 error	$2.54 \cdot 10^{-2}$	$8.17 \cdot 10^{-4}$	$6.95 \cdot 10^{-5}$	$4.83 \cdot 10^{-5}$

4.3 Coupled Heat and Fluid Flow

Our next problem concerns coupled heat and fluid flow in a straight pipe with constant cross section Ω, see [7, pp. 611–614]. Because the velocity field can be taken as uni–directional, the general governing equations (the incompressible

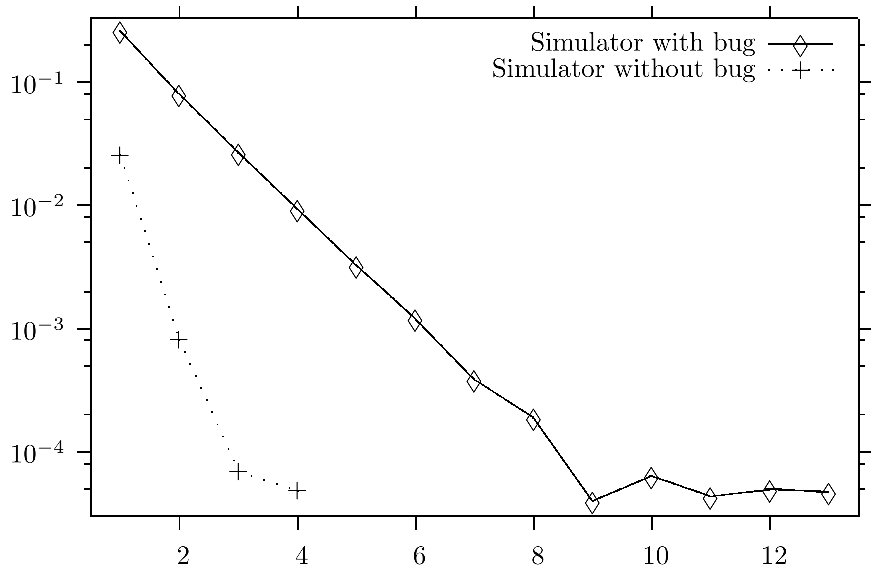

Figure 4: The L_2 error of the numercial solution after each multigrid V–cycle iteration.

Navier–Stokes equations and the heat transfer equation) can be simplified to the following system:

$$\nabla \cdot [\mu \nabla w] = -\beta, \tag{10}$$

$$\nabla^2 T = -\kappa^{-1} \mu \dot{\gamma}^2, \tag{11}$$

$$\mu = \mu_T(T) \mu_w(\dot{\gamma}), \tag{12}$$

$$\dot{\gamma} = \sqrt{(w_{,x})^2 + (w_{,y})^2}. \tag{13}$$

Here, w is the velocity along the pipe, T is the temperature, κ is the heat conductivity, and $\dot{\gamma}$ is a measure of the deformation rate of the flow.

The model (10)–(13) depends on some constitutive relations, i.e., μ_T and μ_w must be defined. Here we apply a power–law fluid model,

$$\mu_w(\dot{\gamma}) = \mu_\infty + \mu_0 \dot{\gamma}^{n-1}, \tag{14}$$

where μ_∞ is the viscosity at high shear rates, μ_0 is a reference viscosity, and n is a real number. The other factor in (12) is taken as,

$$\mu_T(T) = e^{\alpha(T - T_0)}, \tag{15}$$

where T_0 is a given reference temperature and α is a constant.

Experimenting with the parameter values in the models for, e.g., μ_T, μ_w and κ is hard when cutting and pasting from Maple or Mathematica. In the following, we will see that this is much easier and safer done in Python, with Famms.

The C++ simulator for solving the problem (10)–(13) uses an operator splitting technique, where two simulator classes, `EnergySPy` and `MomentumSPy` are coupled. These two classes have been turned into Python extension modules using SWIG. In a Python script, we create one instance of each simulator class:

```
from Pipeflow import *
mgr = Manager()
energy = EnergySPy(mgr)
momentum = MomentumSPy(mgr)
```

Here, `Manager` is a C++ class, wrapped as a Python extension module, that is used by the C++ application to couple the two simulators.

Because we have two equations, we use the `SystemFamms` module in Python instead of the `Famms` module previously used. The `SystemFamms` module administrates several `Famms` instances, one for each equation in the system, and makes sure that the coupling of the two equations is done correctly. We specify the model symbolically in the Python script:

```
def dgamma(w):
    return (w.diff(x, 1)**2 + w.diff(y, 1)**2)**0.5

def mu_w(dgamma):
    return mu_inf + mu0*dgamma**(n - 1)

def mu_T(T):
    return exp(-alpha*(T - T0))

def mu(T, dgamma):
    return mu_T(T)*mu_w(dgamma)

def F1(w, T):
    print "SS"
    print grad(w).spatial_symbs
    return div(mu(T, dgamma(w))*grad(w)) + beta

def F2(w, T):
    return laplace(T) + kappa**(-1)*mu(T, dgamma(w))*dgamma(w)**2
```

Here, `F1` and `F2` are the Python versions of (10) and (11), respectively. We choose analytical solutions and assign the problem specification to the `SystemFamms` object.

```
w = sin(x*y)
T = cos(y*x)
f.assign(sim_list=[momentum, energy], sol_list=[w, T], PDE_list=[F1, F2])
```

At this point, the simulators are up–to–date with the new problem specification.

Numerical Experiment

In this example, we investigate the non–linearity of the model (10)–(15), i.e., we vary the parameter n in (14). When n decreases towards zero, the model becomes more non–linear, and harder to solve.

The solver for this problem is quite complicated. Being a non–trivial PDE model itself, the possible lack of convergence of the solver may be grounded in either coding mistakes, or problems with the numerical methods used. With Famms, we may be able to rule out some sources of bugs, as we can closely investigate the true error of the numerical solution.

The simulator in this experiment utilizes the finite element method for discretization, a Gauss–Seidel–like operator splitting technique for the coupled system, and either Newton–Raphson or successive substitution (Picard iteration) as non–linear solver for each PDE, where the linearized systems are solved with a conjugate gradient squared solver. The convergence criteria used for the two non–linear solvers are $\epsilon_{NL} = \frac{||\Delta x_l||_{L_2}}{||x_l||_{L_2}} = 10^{-6}$, where x_k is the solution of either the momentum or the energy equation at iteration k, and Δx is the change from the previous iteration. In the Gauss–Seidel method, we use the stopping criterion $\max(||\Delta T_l||, ||\Delta w_l||) \leq \epsilon_{GS} = 10^{-6}$, where l denotes the iteration number, and ΔT_l and Δw_l are the change of residuals from the previous to the current Gauss–Seidel iteration.

The solution domain is the unit square in 2D, and the mesh size, h, is $h = 0.1$. The manufactured solutions are $w = \sin(xy)$ and $T = \cos(xy)$. The (constant) model parameters are $\beta = 0.2$ in (10), $\kappa = 0.5$ in (11), $\mu_0 = 1$ and $\mu_\infty = 0$ in (14), and $T_0 = 0$ and $\alpha = 1$ in (15).

In Table 5, we show the number of iterations needed to fulfill the convergence criterion for the Gauss–Seidel method when the power–law exponent, n, is decreased, as well as the final error measured in L_2 norm. When n is small, $n < 0.15$, the number of iterations needed to fulfill the convergence criterion suddenly drops, and the errors increase. The reason for this is that an absolute convergence criterion is used, and because the change in the solutions for highly non–linear choices of n is small in the beginning, this criterion is fulfilled before the method has converged.

Table 5: Number of Gauss–Seidel iterations and L_2 errors for different power–law exponents. The analytical solutions are $w = \sin(xy)$ and $T = \cos(xy)$.

n	0.6	0.3	0.15	0.10
# Iter.	6	9	16	2
$\|\|e_T\|\|_{L_2}$	$1.15 \cdot 10^{-4}$	$1.10 \cdot 10^{-4}$	$1.07 \cdot 10^{-4}$	$1.30 \cdot 10^{-2}$
$\|\|e_w\|\|_{L_2}$	$4.83 \cdot 10^{-5}$	$2.93 \cdot 10^{-5}$	$2.16 \cdot 10^{-5}$	$2.47 \cdot 10^{-1}$

4.4 A Note on the Simulators

The simulators presented here are all using the Diffpack PDE library. Diffpack is a comprehensive C++ library, with a focus on solving PDEs using finite element methods, see [5, 7]. The Famms framework does not, however, depend on Diffpack, as it has been developed as a tool for verifying PDE solvers in general. However, the authors have to this point only used it for real problems in combination with the Diffpack library.

5 Symbolic Manipulation in Python

In Section 3, the module `Symbolic` was used to express analytical solutions and PDE models symbolically in Python. Below, we describe this module in more detail.

GiNaC (GiNaC is Not a CAS (Computer Algebra System)), see [2], is a C++ library designed for applications that need symbolic mathematics. It offers, among other things, symbols and expressions, and differentiation of expression with respect to the symbols involved. Therefore, it is ideal for the purpose of constructing manufactured solutions and differentiating these in accordance to the governing PDE problem. There already exists a Python interface to GiNaC, named PyGiNaC, see [9]. Instead of building on this module, we have chosen to build a new interface using SWIG. Thereby we can tailor the behavior of the interface to fit our needs.

The interface to GiNaC, named Swiginac [13], is basically made by running SWIG on simplified versions of the GiNaC header files. This does not, however, give the wanted high–level mathematical module for expressing PDEs symbolically. Instead, a pure Python module, `Symbolic` [13], is used as high–level front–end to Swiginac. Here, we have access to symbolic expressions, vectors, matrices and various functions working on these. The module does not offer a general interface to all the functionality of GiNaC, but provides what is needed in combination with the MMS.

In `Symbolic` we differ between symbols and symbolic expressions. The latter are mathematical expression built with symbols. A simple example reads:

```
>>> from Symbolic import *
>>> x = Symbol('x')
>>> y = Symbol('y')
>>> t = Symbol('t')
>>> u = exp(-t)*sin(x+y)**2
>>> u.setSymbols([x, y], time=t)
>>> f = (u.diff(t, 1) - laplace(u)).simplify()
>>> print f
(-4*cos(y+x)**2+(3.0)*sin(y+x)**2)*exp(-t)
```

Here, `f` is a symbolic expression for the source term corresponding to the solution `u=exp(-t)*sin(x+y)**2` of the heat equation in two space dimensions. The functions `grad` and `div` above compute the gradient and divergence of expressions, respectively. To do this, we first have to specify the dimension of the expressions. The method `setSymbols` of expressions is used for this. It takes two arguments: a list of symbols defining the space and an optional symbol for time.

Vectors are lists of expressions and are equipped with arithmetic operations, as well as high–level mathematical operations such as `grad` and `div`.

```
>>> u = sin(x)
>>> v = cos(y)
>>> w = Vector([u, v], [x, y])
>>> f = grad(div(w))
>>> print f
[-sin(x),-cos(y)]
```

Both expressions and vectors have methods for evaluation in terms of Python floating point numbers.

6 Summary and Concluding Remarks

In this paper, we have introduced the Python framework Famms for verifying PDE simulators written in C, C++, and Fortran. Famms uses the method of manufactured solutions, and applies it to simulators equipped with a Python interface. We have tried to make Famms simple to use, yet powerful enough to tackle general PDE problems. Famms is freely available, and can be downloaded from `http://software.simula.no/sc/famms`.

We have presented a series of examples, ranging from simple to complicated, and showed how Famms can be used to investigate the properties of numerical solutions and algorithms. Especially, the true numerical error has been computed and compared with the expected theoretical behavior with the aim of uncovering bugs.

We have also presented the Python module `Symbolic` for symbolic calculations. This module is built on top of a low–level SWIG interface to GiNaC, and offers high–level abstractions convenient for expressing analytical solutions and PDE models symbolically.

Appendix

In this appendix we show some examples of Python callbacks, which must be implemented to fit the user's simulation code in oder to utilize Famms. However, implementing the callbacks for a given software library is a *one time job*. For instance, in Diffpack the class `FieldFunc` is used to implement functions of space and (optionally) time. Therefore, we made the class `FieldFuncPython` for Python field functions. Because `FieldFuncPython` is a subclass of `FieldFunc`, a typical Diffpack simulator does not distinguish between field function defined in Python or C/C++.

In the following we list three implementations of Python callbacks to give the reader a starting point for writing her own.

A Python Callbacks and Fortran

One of the advantages of the wrapper tool F2PY [10] is that it allows Python functions to replace Fortran functions. We therefore do not need to implement special callback functions for evaluating source terms and analytical solutions when applying Famms to Fortran simulators.

To create a Python interface to a Fortran PDE simulator, we use the command line tool `f2py`. It can read Fortran source code and generate wrapper code, which in turn must be compiled and linked with the PDE simulator. To exemplify the use of F2PY, we show how Famms can be used to set the initial condition in a Fortran heat simulator.

The function `bell` below can be used to point–wise assign an initial condition to a Fortran array. The arguments are the spatial coordinates and the return value is a scalar floating point number. The Fortran subroutine `setIC` performs a loop over all grid nodes and calls a function, for instance `bell`, to set the initial condition in the array `u` given as argument to the subroutine:

```
      REAL*8 FUNCTION bell(x, y)
      REAL*8 x, y
      bell = exp(- x*x - y*y)
      RETURN
      END

      SUBROUTINE setIC(u, n, initFunc)
      INTEGER n
      REAL*8 u(n, n), initFunc
Cf2py intent(in, out) u
      EXTERNAL initFunc
      INTEGER i, j
      REAL*8 x, y, delta
      delta = 1.0/(n-1)
      DO 20 j=1, n
         DO 10 i=1, n
            x=(i-1)*delta
            y=(j-1)*delta
            u(i, j) = initFunc(x, y)
 10      CONTINUE
 20   CONTINUE
      RETURN
      END
```

After using F2PY to generate a Python interface to this Fortran code, a Python function can be used in place of `bell` directly. The requirement is that the Python and Fortran functions have compatible arguments.

To control the way F2PY builds a Python interface, special comment lines starting with `Cf2py` can be used. Above, `Cf2py intent(in, out) u` is used to specify that the array `u` is both an input argument and return value of `setIC`.

In a Python script for this Fortran simulator, we use Numerical Python arrays and initialize them in Python. We also specify the analytical solution and the PDE model symbolically and use Famms to create Python callbacks for the analytical solution and the source term:

```
import HeatF # Fortran Heat simulator
from Numeric import *
from Symbolic import *
from Famms import Famms

n = 51 # Mesh size in each space direction
u = zeros([n, n], 'f') # Solution
up = zeros([n, n], 'f') # Solution at previous time step

f = Famms(nsd=2, time=True)
[x, y] = f.x; t = f.t # Define aliases

v = exp(-t)*exp(-x*x - y*y) # Manufactured solution

def F(u): # PDE definition
    return u.diff(t, 1) - laplace(u)
```

```
# Compute the source term and return callback functions
[source_cb, sol_cb] = f.createCallbacks(solution=v, equation=F)

# Set initial condition to analytical solution.
def ic(x, y):
    return sol_cb(x, y, 0)

up  = HeatF.setic(up, ic)
```

The initial condition is set by sending the callback to the Fortran subroutine `setic`[4].

B Python Callbacks and C

Contrary to F2PY in Appendix A, SWIG and SIP do not automatically allow a Python function to replace a function written in C/C++. Therefore, when using SWIG or SIP to interface simulators, the wrapper code must be implemented manually, by using the Python/C API directly.

We start by assuming that the PDE solver uses function pointers, e.g., that the source term is evaluated through a function pointer similar to the following:

```
double(*source)(const double pt[], int n, double t);
```

The idea is to implement a wrapper function with a signature the above function pointer can address. The wrapper function then invokes a Python function object defined in the scripting layer. We first convert the arguments to the wrapper function to their Python equivalents and then call a Python function object with the converted values. The latter is done by the Python/C API function `PyEval_CallObject`, which takes two arguments: the function object to call and a Python tuple holding the arguments to the Python function. The function returns a Python object containing the return value. This value is converted to a native data type, in our case a `double`, and returned. The source code for this function reads:

```
double pycb(const double point[], int n, double time) {
   PyObject* pt;
   PyObject* arglist;
   int i;

   pt = PyTuple_New(n);
   for (i=0; i<n; i++) {
      PyTuple_SetItem(pt, i, PyFloat_FromDouble((double)point[i]));
   }

   arglist = PyTuple_New(2);
   PyTuple_SetItem(arglist, 0, pt);
   PyTuple_SetItem(arglist, 1, PyFloat_FromDouble((double)time));

   double dres = 0.0;
   PyObject* result;
   result = PyEval_CallObject(pyobj, arglist);
```

[4]Note that F2PY by default lowercases all subroutine names. In Fortran, the name of this subroutine is `setIC`.

```
   Py_DECREF(arglist);
   if ( PyFloat_Check(result)) {
           dres = PyFloat_AsDouble(result);
   }
   else {
           printf("FieldFuncPython::valuePt: PyObject does not return double\n");
           PyErr_SetString(PyExc_TypeError, "Could not evaluate Python callback\n");
           dres = 0.0;
   }
   Py_XDECREF(result);
   return dres;
}
```

When several function pointers are used in a simulator, we have to implement one wrapper function for each function pointer we want to replace.

C Using Function Objects to Implement Python Callbacks

When a PDE simulator is written in an object–oriented language like C++, we can implement Python callbacks using function objects. Basically, a function object is a function wrapped as a method of a class. Such methods are typically redefined in subclasses, for example to implement different source terms in a PDE simulator.

The code presented below should provide a good starting point for implementing function objects for many PDE simulators in C++. We declare the base class as follows:

```
class evalPt {
public:
   virtual double operator()(const double* point, int n, double t=0.0);
   virtual void operator()(const double* point,double* rets,int n,double t=0.0);
};
```

A simulator that utilizes this class can use the overloaded method `operator()` for both scalar and vector field functions. The input arguments are typically a point in space and a time step.

We declare a Python callback subclass of `evalPt`, named `pyevalPt`:

```
class pyevalPt: public evalPt {
public:
   int nsd; // Number of space dimensions
   pyevalPt (int _nsd){ nsd = _nsd; }
   PyObject* pycb; // Python function to be called
   virtual double operator()(const double* point, double t=0.0);
   virtual void operator()(const double* point, double* rets, double t=0.0);
   void attach(PyObject* pycb_);
};
```

The method `attach` is used from Python to assign a Python function.

The implementation of the scalar field function in `operator()` is similar to the C implementation in Appendix B. The main differences are that the pointer to the Python function is now a class variable, and, because the number of space dimensions are specified in the constructor, we avoid sending the length of the coordinate to the method:

```
double pyevalPt:: operator()(const double* point, double t){
   PyObject* pt;
   PyObject* arglist;

   pt = PyTuple_New(nsd);
   for (int i=0; i<nsd; i++)
      PyTuple_SetItem(pt, i, PyFloat_FromDouble((double)point[i]));

   arglist = PyTuple_New(2);
   PyTuple_SetItem(arglist, 0, pt);
   PyTuple_SetItem(arglist, 1, PyFloat_FromDouble((double)t));

   double dres = 0.0;
   PyObject* result;
   result = PyEval_CallObject(pycb, arglist);
   Py_DECREF(arglist);
   if (result) dres = PyFloat_AsDouble(result);
   Py_XDECREF(result);
   return dres;
}
```

The great advantage of class `pyevalPt` is that we need to implement the wrapper only once.

D Python Callbacks for Diffpack

Diffpack has the classes `FieldFunc` and `FieldsFunc` for representing scalar and vector valued functions (resp.) of space and time. These classes define a virtual method, `valuePt`, which takes points in time and space as arguments and returns the implemented functions evaluated at these points.

For the base class `FieldFunc` we define a subclass named `FieldFuncPython`:

```
class FieldFuncPython: public FieldFunc {
   public:
   PyObject* pyobj_value;
   PyObject* pyobj_grad;
   PyObject* pyobj_hessian;

   FieldFuncPython(const char* name = __null);
   ~FieldFuncPython() {}

   bool ok();
   void attach(PyObject* value);
   void attach(PyObject* value, PyObject* grad);
   void attach(PyObject* value, PyObject* grad, PyObject* hessian);

   virtual double valuePt(const Ptv(double)& p, double t);
   virtual double valueFEM(const FiniteElement& fe, double t=DUMMY);
```

We omit the definition of `valuePt`, because it is similar to the definition of of the method `operator()` in `pyevalPt` in Appendix C. The vector version of `FieldFuncPython`, named `FieldsFuncPython`, is similar, where the main difference is that the returned value from the Python function object is a list that has to be converted to a C++ array type.

References

[1] D. Abrahams. Boost.Python homepage.
http://www.boost.org/libs/python/doc/, 2004.

[2] Christian Bauer, Alexander Frink, and Richard Kreckel. GiNaC homepage.
http://www.ginac.de, 2004.

[3] D. Beazley. SWIG homepage. http://www.swig.org, 2004.

[4] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A Multigrid Tutorial*. SIAM Books, 2nd edition, 2000.

[5] Diffpack. See http://www.diffpack.com/ for further information.

[6] P. M. Knupp, S. G. Krantz, and K. Salari. *Verification of Computer Codes in Computational Science and Engineering*. CRC Press, 2002.

[7] H. P. Langtangen. *Computational Partial Differential Equations – Numerical Methods and Diffpack Programming*. Textbook in Computational Science and Engineering. Springer, 2nd edition, 2003.

[8] J.K. Ousterhout. Scripting: Higher Level Programming for the 21st Century. *IEEE Computer*, March, 1998.

[9] Pearu Peterson. PyGiNaC homepage.
http://cens.ioc.ee/projects/pyginac/, 2001.

[10] Pearu Peterson. F2PY homepage.
http://cens.ioc.ee/projects/f2py2e/, 2004.

[11] P. J. Roache. *Verification and Validation in Computational Science and Engineering*. Hermosa Publishers, Albuquerque, 1998.

[12] Ola Skavhaug. Famms homepage.
http://famms.berlios.de, 2005.

[13] Ola Skavhaug and Ondrej Certik. Swiginac homepage.
http://swiginac.berlios.de, 2005.

[14] P. Thompson. SIP homepage.
http://www.riverbankcomputing.co.uk/sip/index.php, 2004.