PySE; Python Stencil Environment Solving Partial Differential Equations with Python

Åsmund Ødegård

November 15, 2005

Abstract

The purpose of the present paper is to discuss a new framework, Python Stencil Environment, PySE, [1], implemented in Python, for solving Partial Differential Equations (PDEs) using the Finite Difference Method (FDM). By implementing powerful building blocks, or abstractions, the implementation makes a clean user interface possible. This allows the user to focus more on the problem specification and the numerical methods he wants to explore. All parts of PySE are inherently parallel, for effortless deployment on parallel computers, thus making large-scale simulations possible.

PySE is aimed at scientists who need an environment for implementing solvers for PDEs in a simple and clean syntax. The strengths of PySE are flexibility and the possibility of developing PDE solvers with immediate feedback, suitable for experimenting with numerical methods or the rapid prototyping of PDE solvers.

1 Introduction

Creating solvers for Partial Differential Equations (PDEs) is often associated with complicated and error-prone coding in low-level languages. Maximum efficiency is usually the guiding rule when the environment is selected, and for production code that should perform large-scale simulations for longer periods, this is an excellent and even necessary guideline. However, for exploratory work in scientific computing it may be worthwhile to consider other options.

Traditionally, the low-level languages C, C++ and Fortran have been the most popular among computational scientists. However, in recent years, environments such as Matlab, Maple, Octave, S-Plus and R have become popular. In particular, Matlab has been the preferred environment for many computational scientists and engineers. Some of the reasons for the success of these environments are these: simple and clean syntax, interactive execution with immediate feedback, integration of visualization, and good documentation and online help. In addition, a rich standard library of numerical functionality is available in the environments, together with lots of built-in functions that operate efficiently on arrays in compiled code. Although the performance from low-level compiled languages is better, many scientists simply feel more productive in, for instance, Matlab.

The high-level programming language Python has emerged as a viable alternative to Matlab, Octave and other similar environments. When extended with numerical and visualization modules, Python shares many of the advantages with the other interpreted environments mentioned above. Further, Python has

an additional advantage in that the language is very rich and powerful, having been designed from the beginning as an object-oriented language. See [5] for more complete coverage of the advantages of Python in scientific computing.

Using Python, we have implemented the framework PySE (Python Stencil Environment) for solving PDEs using the Finite Difference Method (FDM). By creating a set of high-level abstractions, PySE allows the user to both specify the problem and implement numerical solutions on a high level. In order to move the focus of the user away from the details of the implementation towards the problems and the solution methods, it has been a goal for our design that code should be as close to pseudo code as possible. Our target audience are researchers who need to prototype FDM solvers, and want to play with the FDM or the numerical methods used to solve the resulting discrete problem.

Due to its design, numerical computations carried out in Python will usually be much slower than the same computations implemented in low-level languages such as C and Fortran [5]. As discussed in e.g., [16], a large part of the code for PDE solvers deals with problem specification and various bookkeeping tasks. For these parts of the code, efficiency is not important, and a high-level language such as Python is very well-suited. However, there is a part of the code where efficiency is important; the part of the solver where the numerical computations are carried out. Better performance can be achieved by moving the time-critical part of the code into extension modules for Python, implemented in a compiled language. The extension modules can be either general purpose, like Numerical Python (NumPy) [27], or custom-made for a particular application or library function. In addition, parallel computing can be used to solve a problem faster by spreading the workload over more than one computer. Both approaches are used in PySE.

Ideally, the framework itself should take care of parallelization in such a way that the user code does not need to change at all when moving from a sequential to a parallel computer. If parallelization is performed manually, details of both the problem and the environment in which the application will be run can be taken into account in order to create an optimal parallel solver for the problem. If the parallelization is handled automatically by the framework, some flexibility (and perhaps performance) must be sacrificed. We believe, however, that the benefits of automatic parallelization are more important for the experimental researcher. Hence, we implement parallelization in PySE, in such a way that the user code is left uncluttered.

We begin with a discussion on what kind of user interface a framework for solving PDEs with FDM should have. Then we provide a few examples of PySE in use. Then, we discuss the implementation in more detail to see how the goals are achieved, before presenting some more involved case studies with PySE. Finally, we offer concluding remarks and suggestions for the further development of PySE.

2 Design of PySE

We want to design the user interface of our framework for solving PDEs such that it is possible to work with FDM on a high level, without sacrificing flexibility. That is, it should be possible to describe a problem in as much detail as is necessary, and the scientist should have full control over discretization and solution algorithms. When working with mathematics, the operators and symbols of mathematics have a powerful "built-in" semantics. For example, an expression such as $r = b - Ax$ can result in quite different calculations, depending on what the symbols represent. Therefore, it is fair to say that this is a high-level

```
from pyse import *

# source function:
def f(x,y):
    return math.sin(2*math.pi*x)

# create a Grid, store the division (Δx):
g = Grid(domain=([0,1], [0,1]), division=(40,40))
h = g.dx

# build a FDM stencil:
fdm_stencil = Stencil(nsd=2, nodes = {        \
                  (0,1):  1.0,                \
    (-1,0):  1.0, (0,0):  -4.0, (1,0):  1.0,  \
                  (0,-1):  1.0 } )

# create FDM operator (a StencilSet), and add the stencil:
A = StencilSet(g)
A.addStencil(fdm_stencil,g.innerPoints())

# use a "zero-flux" Neumann boundary condition:
A += createNeumannBoundary(fdm_stencil, g, 0.0, g.range())

# create fields for the source and the solution,
# fill the source:
b = Field(g)
b.fill(f)
b *= -(h**2)
u = Field(g)

# solve the problem, and plot the result:
solution = conjgrad(A, u, b, tolerance=0.005)
plot(solution)
```

Figure 1: Code for solving problem (1).

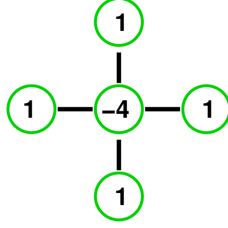

Figure 2: Stencil for the equation in inner nodes, (2)

representation. We seek the same behaviour for the user interface of PySE.

interface will be a set of classes and functions. The user should be able to utilize the framework from both Python programs and interactive Python sessions.

We consider that the important aspects of a utility or framework suitable for high-level work with FDM solvers for PDEs can be summarized as follows:

- The problem can be specified in an intuitive and flexible way.
- The solution method can be implemented clearly and easily.
- Results and runtime statistics should be easily accessible for postprocessing and analysis.

To address these aspects and discuss possible solutions for our framework, we consider the following problem, which models stationary heat conduction, as an illustrative test case:

$$\begin{aligned} -\nabla^2 u &= f, \qquad && \mathbf{x} \in \Omega \\ \frac{\partial u}{\partial n} &= 0, \qquad && \mathbf{x} \in \partial\Omega \end{aligned} \tag{1}$$

where $\Omega = [0,1] \times [0,1]$, the unit square in 2D, and $\partial\Omega$ its boundary, and f is a given function. Given the number n, a lattice grid of nodes (x_i, y_j) for the domain Ω is created, where $i = 0, 1, \ldots, n$, $j = 0, 1, \ldots, n$. The cell length $h = x_i - x_{i-1} = y_i - y_{i-1} = 1/n$ is assumed to be constant.

Using the standard centred differences for the second-order derivative in (1) yields

$$u_{i,j-1} + u_{i-1,j} - 4u_{i,j} + u_{i+1.j} + u_{i,j+1} = -h^2 f_{i,j}, \quad i,j = 1,2,\ldots,n-1 \tag{2}$$

for the inner nodes in the domain, where $u_{i,j} = u(x_i, y_j)$ and $f_{i,j} = f(x_i, y_j)$.

Along the boundaries, the Neumann boundary condition will be used to eliminate values in the schema (2) falling outside the domain. For instance, along the y_n boundary, a second-order approximation of the Neumann condition yields

$$2u_{i,n-1} + u_{i-1,n} - 4u_{i,n} + u_{i+1,n} = -h^2 f_{i,n}, \quad i = 1,2,\ldots,n-1. \tag{3}$$

To give an impression of what a solver implemented with the PySE framework looks like, we show the code for solving problem (1) when $f = \sin(2\pi x)$, in Figure 1.

The equations (2) and (3) relates a point to a few of its neighbours. It is common to name these relations computational molecules or stencils [23, 9, 6]. A stencil can be visualized as in Figure 2, and can be interpreted as the action of

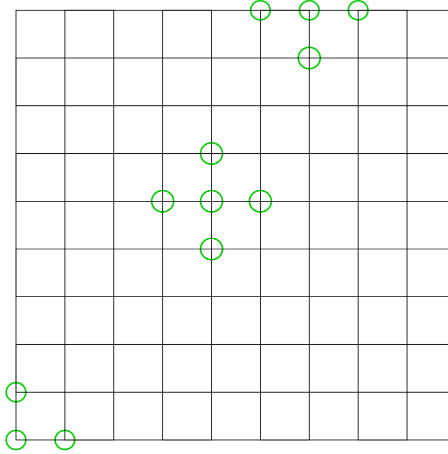

Figure 3: Grid for problem (1). The green circles indicates nodes in stencils.

the PDE at a point. Usually, only a few neighbours are included in each stencil, and the number of different stencils needed to specify the problem for the whole domain is limited. Therefore, a convenient way to specify the FDM for the PDE on a given grid is to specify the different stencils to be used, together with the nodes in the grid where each stencil should be applied.

In Figure 3 we show two of the boundary stencils for the Neumann boundary condition, as well as the shape of the stencil for the inner nodes.

Considering example (1) and the FDM form given in (2,3), the specification of the problem involves describing the lattice grid for the computational domain and the discrete form of the PDE on that geometry. As outlined above, one way of describing the discrete problem is in terms of the stencils, but there are other ways. For instance, there are several tools [22, 17, 10, 11] that use specialized languages, or a graphical interface, to specify the PDE, boundary conditions, computational domain, and so on. Based on user input, a discrete problem is generated in the tool, and solved. This offers a very high-level interface for problem specification, and hides the details of discretization and solution algorithms. We have chosen to use the stencil approach.

The set of stencils for a problem defines a linear operator for scalar fields over the lattice grid for the problem. In the case that an explicit scheme is used, an update of the solution can be computed by applying the operator to the solution from the previous iteration. If the scheme is implicit, the stencils formally define a generator of a linear system. However, by iterative methods, the solution can be approximated using the operator, without forming the linear system. Krylow methods [32, 2] are examples of such methods.

The set of stencils for a problem, defining the linear operator, should therefore be encapsulated in a single object. Each stencil should be paired with the set of nodes in the grid, for which the stencil defines the action of the PDE. This object can be used in both explicit and implicit solvers, as explained above.

The final step, postprocessing, will usually only involve analysis of the unknowns after the solver is finished, as well as statistics about error estimates and the number of iterations stored during the solution process. As the unknowns will be collected in some data structure, and other kinds of information may conveniently be stored in some ready-made objects as well, this last part may also be separated from the rest of the tasks through a clean interface. We have not paid much attention to the matter of postprocessing in the early stages of

this framework. Only simple plotting capabilities are implemented.

2.1 Abstractions

From the example above, it is evident that the design of a set of abstractions is crucial for achieving the user interface we want. Some of the abstractions we have decided to implement may be common to other frameworks for solving PDEs [7, 20, 18, 3, 28, 30, 23], in some cases with different names, while some abstractions are unique to this framework.

We will focus on the following abstractions: Grid, Field, Stencil, and StencilSet.

2.1.1 Grid

Before we can apply the FDM to a given problem, we need to define the computational domain, as well as a lattice grid covering the domain. This will be the purpose of the *Grid* abstraction. Information about geometry, division parameters for the lattice grid, and so on, will be contained in the grid.

Our *Grid* abstraction will only handle lattice grids, which can always be described functionally, based on the geometry and the division parameters. Hence, coordinates for all nodes in the grid will not be stored, thus reducing the memory requirement of a grid significantly.

2.1.2 Field

We need to work with discrete approximations of the unknowns in the PDE. These can be represented as fields over the grid. The current implementation of *Field* is scalar, i.e., it stores one value for each node in the corresponding grid. In the future, we may consider vector fields with more than one degree of freedom in each node.

We are using a separate abstraction for scalar fields because we want to be able to index the field in different ways, as well as perform certain operations on fields. This will be discussed in Section 5.4.

2.1.3 Stencil

The *Stencil* is, together with *StencilSet*, the most important abstraction in our framework. As explained above, the stencil defines the discrete approximation of the PDE at one point of the grid. As the stencil will be the key component used to specify a problem, the abstraction must have the following properties:

- A stencil should be easy to build by specifying the individual stencil nodes and their coefficients.
- In addition to the offset nodes, the stencil should also contain an optional source term.
- Both constant and variable, or functional, coefficients should be supported. Functional coefficients should be evaluated by giving the coordinates of a point as argument.
- It should be possible to add and subtract stencils, as well as multiply a stencil with a scalar, in order to scale the stencil.

The first three points are important to give the user sufficient power to build the stencils for the PDE. The last point enables the user to create complex stencils

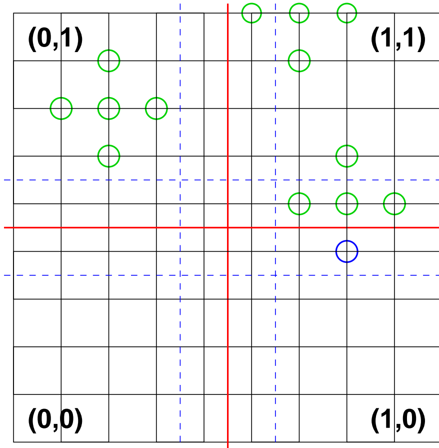

Figure 4: Grid for problem (1). The red lines mark the partition for four processors, while the dashed blue lines mark extent of ghost nodes. The green circles indicate stencils (the blue circle is a ghost node), while the numbers in brackets indicate the processor ID.

by first building simpler ones and later joining them together. It also gives the flexibility of building a library of frequently used stencils, which can be adjusted and combined to form the complete stencil for a particular problem. In the PySE framework, we have a number of such predefined stencils; see section 5.6.1.

2.1.4 StencilSet

While the stencil describes the discrete approximation of the PDE for one node, the operator defined by all stencils for the problem, describes the action of the PDE problem, including the boundary conditions, for all nodes in the computational domain. We use the term *StencilSet* for this abstraction.

The stencilset is a container in which multiple stencils can be stored. Together with each stencil, we store a description of a set of nodes in the grid, where the stencil should be applied. Key features of the stencilset are as follows:

- Built by adding stencils and a specifications of set of nodes to the stencilset.
- Operates on a field as a linear operator.
- Two stencilsets can be combined into one, in order to build one linear operator from multiple stencilsets.

This design makes our framework very flexible. For instance, it is easy to implement a solver for a problem with different conditions on different parts of the boundary. It is also possible to implement a solver for a problem where different PDEs govern different parts of the computational domain.

3 Automatic Parallelization

The process of creating a parallel solver by partitioning the problem is well known in the literature, cf. [6, ch. 1.5]. Partitioning a problem can be a challenging task, and user interaction is usually required in the process. Our approach is, however, to implement the framework in such a way that a parallel

solver can be created from a sequential solver without any user interaction but a call on the method that initializes parallelization. Because partitioning is done automatically, we have to ensure that it is as optimal as possible. That is, each subgrid should have approximately the same number of nodes and the communication cost should be minimized. Working with structured lattice grids, it is relatively easy to fulfil the first requirement.

In Figure 4 we show a grid partitioned for four processors, with some stencils for a problem. In order to avoid communication each time a stencil is applied at an internal boundary, it is common to enlarge each subpart of the grid with a layer of "ghost nodes", such that all required updates from a neighbour can be made with one vector communication. The width of the ghost node layer will be determined by the shape of the stencils. By inspecting the stencilset, we can acquire information about the communication cost in each direction. Assume that all stencils for a problem are combined into a single stencil - an aggregated stencil. For each space dimension i we define the communication cost value c_i as the width of the aggregated stencil in direction i. For a given number of space dimensions k, the communication cost functional for one process will be

$$C(P;k) = \sum_{i=0}^{k} \frac{c_i * n_i}{p_i} \tag{4}$$

where n_i is the number of nodes in direction i, and $P = \{p_i\}_{i=1}^{k}$ is the number of subparts into which each direction should be split, such that $P_T = \prod_i p_i$ is the total number of processes available. To obtain the optimal partition, we minimize the functional (4).

Before the presented algorithm can be used, the stencilset must be known. Hence, there will be a sequential part of the code that must be executed before the problem can be parallelized. The user should try to avoid creating fields in this sequential part of the code, as such fields will be global and hence may require a large amount of memory.

4 Experiments

Through examples, will now show how the framework can be used and study the performance of implemented solvers. The presented code examples will include details yet to be discussed, as details of the implementation will be addressed in Section 5.

Example 1: Consider the heat flow problem

$$u_t(x,t) = u_{xx}(x,t), \qquad x \in [0,1], \qquad t \in \mathbb{R}^+, \tag{5a}$$

$$\frac{\partial u(x,t)}{\partial n} = f(x), \qquad x = 0 \wedge x = 1, \qquad t \in \mathbb{R}^+, \tag{5b}$$

$$u(x,0) = g(x), \qquad x \in [0,1]. \tag{5c}$$

We apply centred differences for the space derivative in (5a), and a forward difference approximation for time, which yields the schema

$$u_i^{n+1} = u_i^n + \frac{\Delta t}{\Delta x^2}(u_{i-1}^n - 2u_i^n + u_{i+1}^n) \tag{6}$$

for the inner nodes. We show the Python code of a solver for this problem in Figure 5. A few import statements for the PySE framework, and code for the initial condition and the Neumann boundary condition functions are omitted.

```
g = Grid(domain=[0,1],division=n)
lap = Laplace(g)
id = Identity(g.nsd)
innerstencil = id + dt*lap
A = StencilSet(g)
innerindex = A.addStencil(innerstencil,g.innerPoints())
A += createNeumanBoundary(A[innerindex],g,nc)
g.partition(A) # partition if more than 1 cpu
u = Field(g)
u.fill(initc)
tc = 0
while tc < T:
    u = A(u)
    tc += dt
    u.plot(movie='on',title='Heat transfer in 1D, t = %e' % (tc))
plot(field=u,title='Finish, Heat transfer in 1D')
```

Figure 5: A simple example. The functions `nc` and `initc` are defined, `T` is the end time, and `dt` is the timestep.

This code make use of two stencils that are predefined in the framework, the `Identity` and `Laplace` stencils (see section 5.6.1). The concepts of adding stencils, and multiplying a stencil with a scalar, are used to form the complete stencil for the problem. Note that the Δx appearing in (6) is incorporated in the `Laplace` stencil, so we need only multiply by the time-step size Δt when we build the complete stencil.

The expression `u = A(u)` in the code is the explicit time stepping, where the stencilset is used as a linear operator on the solution from the previous iteration.

Example 2: Next, we consider the following problem:

$$u_t = \nabla \cdot (k(x)\nabla u) + f, \qquad x \in \Omega, \qquad t \in \mathbb{R}^+, \tag{7a}$$
$$u(x,t) = h(x,t), \qquad x \in \partial\Omega, \qquad t \in \mathbb{R}^+, \tag{7b}$$
$$u(x,0) = g(x), \qquad x \in \Omega. \tag{7c}$$

Assume that $\Omega = [0,1] \times [0,1]$, the unit square in 2D, is the computational domain, and $x = (x_1, x_2)$ is a point in 2D. We define the coefficient function $k(x)$, the source function $f(x,t)$, the Dirichlet boundary condition $h(x,t)$ and the initial condition $g(x)$ such that the analytical solution for the problem is

$$u(x,t) = e^{-t} \sin(\pi x_1) \cos(\pi x_2) \tag{8}$$

If we use a forward difference for the time derivative and centred difference for the derivative in space in (7a), we get

$$\begin{aligned} u_{i,j}^{l+1} = H[&k_{i,j-1/2} u_{i,j-1}^l + k_{i-1/2,j} u_{i-1,j}^l \\ &+ (1/H - k_{i,j-1/2} - k_{i-1/2,j} - k_{i+1/2,j} - k_{i,j+1/2}) u_{i,j}^l \\ &+ k_{i+1/2,j} u_{i+1,j}^l + k_{i,j+1/2} u_{i,j+1}^l] + \Delta t f_{i,j}^l, \end{aligned} \tag{9}$$

where $H = \Delta t / \Delta x^2$. Expressions such as $k_{i-1/2,j}$ indicate that the function $k(x)$ should be evaluated at the midpoint between $x_{i-1,j}$ and $x_{i,j}$. If $k(x)$ is given as a scalar field over the grid, an average of k over the same segment can

```
# build the stencil
s = Stencil(nsd=2,varcoeff=True,source=dt_f)
s.addNode((0,-1),[lambda x:  H*k_p(x)])
s.addNode((-1,0),[lambda x:  H*k_p(x)])
s.addNode((0,0),[lambda x:  1.0 - 2.0*H*(k_m(x) + k_p(x))])
s.addNode((1,0),[lambda x:  H*k_m(x)])
s.addNode((0,1),[lambda x:  H*k_m(x)])

bs = DirichletBoundary(2,bf)

A = StencilSet(g)
A.addStencil(s,g.innerPoints())
A.addStencil(bs,g.boundary())
```

Figure 6: Code for building the stencils for problem (7a).

# cpus:	1	2	4	8	16	24	32
1000×1000, 160 step:	7984	3969	1998	996.6	498.5	332.0	249.3
speed–up:	1	2.01	3.99	8.01	16.0	24.0	32.0
1500×1500, 240 step:	26820	13460	6728	3373	1681	1125	838.8
speed–up:	1	1.99	3.98	7.95	15.9	23.8	31.9

Table 1: CPU time in seconds and corresponding speed-up numbers for the heat conduction solver.

be used instead. The complete code for this example is included in Appendix A. In Figure 6, we show the part of the code that builds the stencils. In this case, we cannot use predefined stencils from the framework, but have to build them manually. The function `k_p(x)` in the code, is $k(x) + \Delta x/2$, while `k_m(x)` is $k(x) - \Delta x/2$.

In Table 1, we list the CPU time for the solver when run on an Itanium-II based linux cluster[1], for different numbers of processors and two different grid sizes. We also list the speed-up numbers. We have only measured the time of the main computational loop over all time steps (the timeloop), excluding startup and initialization costs. Parallelization works quite well in that we can see close to linear speed up. From previous experiences with Python, we expect the solver to run quite slowly. For comparison, we have therefore implemented a scalar solver for the same problem in C. This solver has almost no flexibility at all, can only solve this particular problem, and therefore is supposed to give close to optimal efficiency. In Table 2, we show the CPU time for the solver implemented in C, as well as speed-up numbers relative to the Python solver. The C-solver running on one processor performs about 75 times faster than the Python solver running on one processor, and also performs about 2.3 times faster than the Python solver running on 32 processors.

While 75 times slower than C is a substantial performance hit for the Python solver, the gain in flexibility is significant. In addition, the Python performance may be suitable for a range of applications, such as experiments with stencils, prototyping of solvers and smaller problems. However, to solve large scale

[1]Linux cluster: 24 dual HP RX2600 1.3GHz Itanium-II computers, 4GB memory each, Gigabit ethernet network. All CPU time measurements in this paper are performed on this system.

Problem size	runtime	1-cpu Python / C	32-cpu Python / C
1000×1000, 160 steps:	107.3	74.4	2.32
1500×1500, 240 steps:	362.4	74.0	2.31

Table 2: CPU time in seconds for the solver implemented in C, as well as speed–up relative to the Python solver running on one and 32 processors.

Problem size	C	Python
1000×1000, 160 time steps:	34.5	30.5

Table 3: Timing of the computational loop for an experiment with only space-dependent source and Dirichlet boundary condition functions. The numbers are CPU time in seconds.

problems the user needs better performance. After discussing PySE in greater detail, we will return to this experiment and show how better performance can be achieved.

In the example discussed here, both the source function and the Dirichlet boundary condition are time-dependent. These functions are implemented as standard Python functions, and must to be executed for all relevant nodes at each timestep. This is a major performance bottleneck of the solver. Consider instead the problem where the Dirichlet and source functions are space-dependent only. If the CPU time of the timeloop is measured, we get the numbers given in Table 3 In this case, the Python simulator is actually faster than the authors' hand-implemented C simulator. It should be noted that the initialization phase, which is omitted in this timing, is significantly longer for the Python simulator than the C simulator, but this is of less importance when the simulator runs for a long time. We conclude that the need to call the source and Dirichlet functions inside the time loop for all relevant nodes constitutes the main bottleneck of the current implementation. This issue will be addressed in Section 6.

5 PySE overview

In the following sections, we provide an overview of the classes and functions implemented in PySE. We constrain ourself to discussing the basic public user interface of the framework. In Sections 5.6.4 and 6.3 we discuss some of the more advanced parts of PySE. A discussion of tools that are mainly for internal use are beyond the scope of this text.

Above, we discussed the key abstractions in PySE. These abstractions are realized as classes in Python. Consequently, there are classes named `Grid`, `Field`, `Stencil` and `StencilSet`. In addition, PySE consists of a set of utility classes and functions. For parts of this discussion, a general understanding of "special methods" and operator overloading in Python, will be beneficial. For an overview of this topic, please refer to, for instance, [31, sec. 3.3] or [21, p. 90-99].

5.1 Grid

The class `Grid` contains a description of the computational domain, as well as how this should be covered with a lattice grid. Currently, "box-shaped" domains

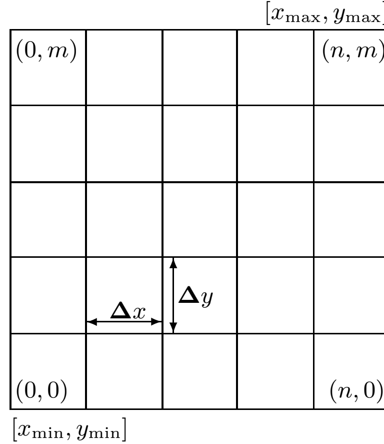

Figure 7: A 2D lattice grid

in any number of space dimensions are supported. An example in 2D is given in Figure 7. To describe the grid fully, $[x_{\min}, y_{\min}]$, $[x_{\max}, y_{\max}]$, and either $(\Delta x,\ \Delta y)$ or (n, m) must be known. The constructor, `__init__`, of the `Grid` class takes the geometry information and the number of divisions as arguments, together with the number of space dimensions:

```
g = Grid(domain=([0,1],[0,2]), division=[50,100])
```

The geometry information is given as $([x_{\min}, x_{\max}], [y_{\min}, y_{\max}])$. The division parameter can also be given as a single integer:

```
g = Grid(domain=([0,1],[0,2]),division=100).
```

In that case, each dimension will be divided by the same number. The generalization to n dimensions is straightforward.

The most important members and methods of `Grid` are listed in Figure 8. Note that we store the $\Delta x, \Delta y, \ldots$ values as a list in the member `division`. In addition, the division parameters are directly available as `dx`, `dy`, and `dz` for grids in one, two and three space dimensions, as this is often needed in the numerical algorithms.

As discussed in Section 2.1.4, each stencil in a stencilset needs a description of a set of nodes in the grid where the stencil should be applied. The purpose of the methods `corners`, `boundary`, and `innerPoints` in `Grid` is to provide such descriptions. For instance,

```
g.innerPoints()
```

will give an *iterator* [21, p. 55] for the indices of all inner nodes in the grid. Similarly, `boundary` and `corners` give the indices for boundary and corner nodes, respectively. It should be noted that only nodes that are on

Grid

Members:

nsd: number of space dimensions
division: list, length=nsd
geometry: list, length=nsd
fields: list of field−references

Methods:

__init__(div,nsd,geo) : constructor
corners(): returns iterator
boundary(): returns iterator
innerPoints(): returns iterator
range(): returns index−range
partition(stencillist): parallelize grid and attached fields.

Figure 8: Outline of the `Grid` class.

the physical boundary are considered boundary nodes and that interior nodes, as returned by innerPoints, are the complement to the boundary. Support for higher-order methods is something that will be considered for further development. The methods innerPoints and boundary accept an argument region that can be specified to select a subset of the domain. The argument should be given in the same form as the geometry, and if it is not present, the whole grid is assumed. The boundary method can, in addition, create an iterator for nodes inside or outside a circle with a given centre and radius. More examples of the use of these methods will be given in the case studies in Section 6, but to indicate the syntax we provide one example here, a circle on the x–y face of a 3D domain:

```
g.boundary(region=([-1,1],[-1,1],[0,0]), type='circle',\
   center=(0,0,0), radius=0.5, direction='in')
```

The method range gives the index range for nodes in the grid. For the grid in Figure 7, for instance, g.range() returns the list [(0,0),(n,m)].

The last method in Figure 8 is partition. The line

```
g.partition(stencilset)
```

in a solver partitions the grid for parallel computations, based on the instance of stencilset given as argument. In addition, all fields defined over the grid, which are referenced in the member fields, will be parallelized by the method. Finally, the stencilset given as argument will be modified according to the parallelization of the grid. In effect, this single method invocation is the only user interface to parallel computing in PySE.

5.2 Stencil

The members and methods in the class Stencil are outlined in Figure 9. The basic version of the constructor for a stencil object accepts three arguments:

```
s_f = Stencil(nsd=2, varcoeff=False, source=0.0)
```

The arguments given here are also the default arguments for the constructor, and can therefore be omitted.

Here, a two-dimensional stencil with fixed coefficient, varcoeff=False, and a source term of 0.0 is created. If variable coefficients are requested, the source term can be given as a function as well:

```
def sourcefunc(x,y):
   return sin(x)*cos(y)

s_v = Stencil(2,True,sourcefunc)
```

Nodes can be added to a stencil in two ways, directly on instantiation using the nodes keyword argument, or with the addNode method. The node is specified as an offset tuple, and the coefficient is given as either a scalar value or a function:

```
s_f = Stencil(nsd=2, nodes = {\
             (0,1): 1.0,\
```

Stencil

Members:
- nsd: number of space dimensions
- varcoeff: True/False – var. coefficients
- source: Source value/function
- stencil_coeff: dictionary

Methods:
- __init__(nsd,varcoeff,source, nodes)
- addNode(index,value): add node
- addSource(value): add source
- __add__(stencil): add operator
- __sub__(stencil): subtract operator
- __iadd__(stencil): inplace–add
- __rmul__(val): multiply with scalar
- convert_to_varcoeff()

Figure 9: Outline of the Stencil class.

```
(-1,0): 1.0, (0,0): -4.0, (1,0): 1.0\
             (0,-1): 1.0 })
s_f.addNode((1,1),1.0)

# variable coefficients:
s_v.addNode((0,1),some_function)
```

The `nodes` argument should be specified using the syntax of Python dictionaries.

As shown in Figure 9, `Stencil` supports addition, subtraction and in–place addition operators with another stencil, implemented with the special methods [21, p. 90] `__add__`, `__sub__`, and `__iadd__`. Also, a stencil can be scaled by multiplication with a scalar, implemented with `__rmul__`.

Finally, an instance of `Stencil` with fixed coefficients may be converted to a variable coefficients stencil at any time,using the method `convert_to_varcoeff`, but the converse is not possible.

5.3 StencilSet

Although `StencilSet` probably is the most important abstraction in our framework, the implementation of the class is rather short and straightforward. From the point of view of implementation, it is basically a placeholder for stencils equipped with a limited set of operators. An outline of the class is shown in Figure 10.

The constructor of `StencilSet` takes one argument, a grid:

```
g = Grid((n,m),[(0,1),(0,1)])
A = StencilSet(g)
```

The method `addStencil` is used to add stencils and the accompanying iterators to an instance of `StencilSet`. Suitable iterators can be obtained from the methods on the grid instance:

```
is = A.addStencil(i_s,g.innerPoints())
bs = A.addStencil(b_s,g.boundary())
```

The value returned by `addStencil` is an integer index value, which can be used later on to retrieve a reference to the added stencil from the stencilset. For this purpose, `StencilSet` implementes the special methods `__getitem__` and `__setitem__` [21, p. 96], which allows square bracket operators to be used with instances of `StencilSet`:

```
old_stencil = A[is]
A[is] = new_stencil
```

The most important operator supported by `StencilSet` is the "call" operator. The "call" operator implemented with the special `__call__` method is a feature of object orientation in Python which makes an instance of the class behave like a function. This is used to implement the linear operator behaviour of the stencilset. Let `A` be an instance of `StencilSet`, and `u` be an instance of `Field`. The user can use the syntax `A(u)` to call

StencilList

Members:
grid: reference to a grid
collection: list of stencils
where: list of iterators

Methods:
__init__(grid) : constructor
addStencil(st,it): add stencil and iterator
__call__(field): apply to field
__iadd__(stencillist): inplace–add
__mul__(field): alias for call
__getitem__(index): get stencil for index
__setitem__(index,st): replace stencil for index.

`__call__` with `u` as argument. This operator walks through the list of stencils and, for each of them, applies the stencil at each node given by the corresponding iterator. When there are variable coefficients, and when the coefficient for a stencil node contains more than one function, all functions are evaluated with the coordinates for the current grid node as argument. The results are then added together and multiplied by the value of the field in the given node. The implemented "call" operator supports only `Field` as operand.

To be more exact, the set of stencils and iterators in the stencilset is only traversed on the first invocation. For subsequent calls, the information is assembled in a sparse matrix structure, which can be applied directly to fields, thus yielding fast operations. If any coefficient or source function needs to be updated, for instance due to time dependency, the matrix structure needs to be reassembled. The methods `updateDataStructures` and `updateSourceDataStructures` set a flag in the stencilset instance that triggers reassembling of either all data or data for source functions only, on the next invocation of the call operator.

The arithmetic multiplication operator `*`, implemented with the special method `__mul__` is an alias for the call operator. The reason for this is that in the context of, e.g., iterative methods such as Krylow solvers, it is common to write `A*u`, thus keeping the matrix-vector analogy in the syntax.

The last method mentioned in Figure 10 is `__iadd__`. This implements the in-place arithmetic addition, `+=`, operator, which is used to combine two instances of `StencilSet` into one object:

```
A = StencilSet(g)
# add stencils to A
...
B = StencilSet(g)
# add stencils to B
...
# Add B into A:
A += B
```

A common usage of this operator will be seen in Section 5.6.2.

5.4 Field

As described above, `Field` is used to hold scalar fields over grids. To hold the numbers, `Field` uses a one-dimensional NumPy array, stored in the member `data`. The instance of `Grid` that the field is defined for is stored in `grid`. In the case that the dimension of the grid is larger than one, the reshape functionality in NumPy is used to create an auxiliary data structure `shapedata` that references the data array, but can be indexed with multidimensional indices according to the grid. Arithmetic operations can be carried out more efficiently on the one-dimensional data, while a multidimensional structure is convenient when working with the stencils. The members and methods of `Field` are outlined in Figure 11.

The constructor in `Field` takes an instance of `grid` as argument:

```
g = Grid((n,m),[(0,1),(0,1)])
```

```
f = Field(g)
```

`Field` supports a collection of operators commonly used for vectors. Vector operations on fields are carried out directly on the underlying NumPy array, using efficient operators implemented in NumPy. These operators include addition, subtraction, and pointwise multiplication with another `Field`, and multiplication with a scalar. All these operations are implemented with the corresponding special methods, as seen in Figure 11, and hence called using the common arithmetic operators:

```
f1 = Field(g)
f2 = Field(g)
f = f1+f2; f = f1-f2; f = f1*f2
f = 5.6*f1
```

Square bracket notation can be used to acquire and set values for individual nodes.

There are three more methods in Figure 11. First, there is `fill`, which fills the field with values based on a function:

```
def initfunc(x,y):
   return sin(x)*cos(y)
f.fill(initfunc)
```

Field

Members:
data: numeric array, 1D
shapedata: reference to data
grid: reference to grid

Methods:
__init__(grid) : constructor
fill(func): fill with values from func.
inner(field): innerproduct
plot(): create 2D/3D plot
__setitem__(index,val): set value
__getitem__(index): get value
__add__(field): add operator
__sub__(field): subtract operator
__mul__(val): multiply operator
__rmul__(val): multiply w. scalar as left operand.

Figure 11: Outline of the `Field` class.

The argument to the supplied function will be coordinates for each node in the grid, given as a list. Note that the coordinates will also be given as a list for one-dimensional grids. Further, instances of `Field` have the method `inner`, which computes the inner product of two fields, and `plot`, which creates a graphical plot for one- or two-dimensional fields.

5.5 Parallelization

A parallel infrastructure is necessary in order to utilize a parallel computer. There are several different approaches to implementing parallelism; see e.g. [12, 33, 4, 16]. In PySE we have used the de facto standard for parallel computing, MPI [13, 29, 25], which is widely available and used. There are several Python modules that provides wrappers to MPI; some of them are pyMPI [24], mpi4py [8], Scientific.MPI [15], and Pypar [26]. Based on its simplicity and its ability to communicate numerical arrays in a very efficient manner, we have chosen to use Pypar. Pypar only implements a limited, but important, subset of MPI. For instance, nonblocking send and receive is not implemented. Also, it is not possible to use MPI communicators with Pypar. However, for our project, it is well-suited.

As the actual implementation of the automatic parallelization is quite involved, and also never exposed to the user directly, we do not here include a discussion of this topic.

5.6 Implemented functions and tools

We will now discuss some of the other tools that are implemented in the framework. This includes some implemented stencils, some iterative solvers, and a

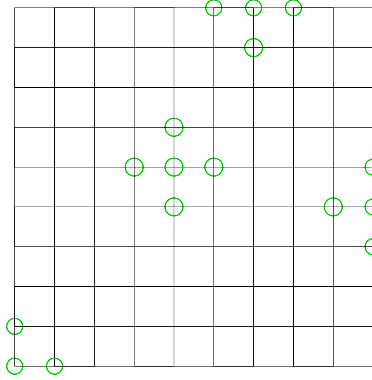

Figure 12: A stencil modified for a Neumann boundary condition.

tool for implementing Neumann boundary conditions. These tools are both often used, and can serve as examples of how PySE can be used. Finally, we also discuss the iterators to be used in stencilsets.

5.6.1 Stencils

Four stencils are implemented: an identity stencil, a stencil for the Laplace operator, a modified Laplace stencil, and a stencil for Dirichlet boundary conditions. All stencils are implemented as subclasses of the `Stencil` class.

The class `Identity` implements a stencil with the coefficient 1.0 in the centre as the only node. The constructor takes a single parameter, the number of space dimensions.

The discrete Laplace operator, which results from using centred differences on the Laplace operator ∇^2, is often encountered in applications. The discrete form depends on the division parameter of the grid as well as the number of space dimensions. We have implemented a `Laplace` class that builds the stencil for the Laplace operator in any number of space dimensions, where the division parameter in the grid may be different for each dimension. An instance of `grid` is the required argument for the constructor.

The class `LaplaceJ` implements a version of the discrete Laplace operator where the coefficient for the centre node is zero, and the other coefficients are multiplied with the inverse of the coefficient for the centre node in the regular discrete Laplace operator. This is, for instance, the stencil that will appear when a Jacobi iteration is applied to the scheme for the Laplace equation $\nabla^2 u = f$, as given in (2).

The last stencil we have implemented is a Dirichlet boundary condition, in the class `DirichletBoundary`. This is a stencil that does not have nodes, but does have a nonzero source term that represents the essential boundary condition. Hence, the arguments to the constructor are the number of space dimensions and the boundary condition, as a function.

5.6.2 Neumann Boundary Conditions

While in principle easy, the implementation of a Neumann boundary condition can be challenging, at least for multiple dimensions. The Neumann condition is interpreted as an auxiliary condition used to eliminate nodes that fall outside the computational domain, from stencils at the boundary, as discussed in Section 2. Which nodes, and even the number of nodes, to be eliminated, varies for the

```
def jacobi(A, x, b, tolerance=1.0E-05, relativeconv=False):
    r = b - A*x
    xn = x + r
    r0 = inner(r,r)
    if relativeconv:
        tolerance *= sqrt(inner(b,b))
    while sqrt(r0) > tolerance:
        xp = xn
        r = b - A*xp
        xn = xp + r
        r0 = inner(r,r)
    return xn
```

Figure 13: Code for the Jacobi iteration

different parts of the boundary, and so it is a complex task to implement the stencils for the Neumann condition and the iterators that are needed for each of them. Figure 12 shows three different example boundary stencils, for the upper boundary, the right boundary, and the lower left corner. As we see, one node is removed from the upper boundary stencil, another one from the right boundary stencil, while two nodes are removed from the corner stencil.

To simplify this task, a function with the name `createNeumannBoundary` is implemented. This function creates a set of stencils together with the accompanying iterators, which implements the Neumann condition on a part, or the whole boundary, of a grid. The stencil that should be modified at the boundary must be given as an argument to the create function. Further, it needs a grid, the Neumann condition as a constant or a function, and a specification of the boundary. The boundary should be specified as for the `boundary` method in `Grid`. The create function returns a new stencilset with all the modified stencils and their iterators. The boundary specification can be used to select a part of the boundary of the grid where the condition should be used. In this way, different boundary conditions can be implemented on different parts of the boundary. Examples will be provided in the case studies in Section 6.

The stencilset produced by `createNeumannBoundary` can either be added to another stencilset, which contains stencils for the rest of the domain, or stencils and iterators for other parts of the domain can be added to the produced stencilset. The in-place addition operator on stencilset was created for the purpose of adding this created stencilset to another existing stencilset in theuser code.

5.6.3 Implemented Solvers

Regarding direct or explicit solvers, there is really nothing to implement, because a step or an iteration in an explicit method will simply apply the stencilset to some field of unknowns. An example of a direct solver was given in Section 4.

There are two iterative solvers implemented in the framework: the Jacobi iteration and the Conjugate Gradient iteration, available as the functions `Jacobi.jacobi` and `ConjGrad.conjgrad`. Note that these solvers are provided as examples of how linear solvers can be implemented in Python in general and PySE in particular. Most real-world application will require preconditioning of the linear system, which is not implemented in these examples.

In Figure 13 we show the code for the Jacobi iteration. Given a stencilset

```
def conjgrad(A, x, b, tolerance=1.0E-05, relativeconv=False):
    r = b - A*x
    p = r.copy()
    r0 = inner(r,r)
    if relativeconv:
        tolerance *= sqrt(inner(b,b))
    while sqrt(r0) > tolerance:
        w = A*p
        a = r0/inner(p,w)
        x = x + a*p
        r = r - a*w
        r1 = inner(r,r)
        p = r + (r1/r0)*p
        r0 = r1
    return x
```

Figure 14: Code for the Conjugate Gradient iteration

`A`, a field for the initial guess of the solution, `x`, and a field for the right-hand side in the problem, `b`, the Jacobi iteration is performed in order to find an approximate solution for `x`. A convergence criterion can be specified as a fourth argument or with the keyword `tolerance`. There is also a choice of using a relative convergence criterion, by setting the flag `relativeconv` to `True`. In that case, the norm of the residual is divided by the norm of the right-hand side before being checked against the tolerance. The default values for the tolerance and the relative convergence flag are $1.0e^{-5}$ and `False`, respectively.

Remark that even though we are working with classes implemented in Python, most of the arithmetic operations are carried out directly on the underlying data arrays, where these operations are implemented in a low-level compiled language, usually C. In this way, decent numerical efficiency can be achieved. Also, note how uncluttered the implementation is. This is achieved by utilizing the operator overloading concept that is common to object-oriented languages to define the action of common arithmetic operators for any object. The dynamic typing in Python is also an important factor.

In Figure 14 we show the code for the Conjugate gradient solver, as implemented in `ConjGrad.conjgrad`. The signature of the function is exactly the same as for the Jacobi solver, such that the solvers can be interchanged.

There are two striking facts that should be mentioned regarding the codes in Figure 13 and 14. First, it is worthwhile to note the resemblance of the code to pseudo code. Although what we show here is real, working code as implemented in the framework, it looks very much like pseudo code for the same algorithms in any textbook. The importance of this is that the code is easy to understand, and that it is quite easy to implement other solvers if you have pseudo code for the algorithm available, as an almost line by line translation to Python code is possible. The second point is that there is no reference to parallelism in the code. As the components used are inherently parallel, there is no need to consider parallelism explicitly in the solvers.

Finally, we would like to remark that although the object `A` was supposed to be a stencilset and the objects `x` and `b` fields, there is nothing in the implemented solvers that require this. Any objects that support the arithmetic operations used, as well as the inner product and the copy operations, can be used with the solvers. Hence, if `A` is a matrix and `x` and `b` are vectors, the same solvers can be

```
# Create an iterator for the set (0,0) - (4,4):
ti = tupleIterator(2,(0,0),(4,4))

# Usage:

for i in ti:
    print i

# Generates (0,0), (1,0),..., (0,1), (1,1),...,(4,4)

# Create an iterator for the plane (1,1,0) - (3,3,0):
ti = tupleIterator(3,(1,1,0),(3,3,0))
```

Figure 15: Create and use a `tupleIterator`

used directly in a scalar context. In a parallel context, only the parallel versions of `A*x` and the inner product must be implemented before the solvers can be used with matrix and vectors in this case. This is a feature made possible by using a dynamically-typed language such as Python.

5.6.4 Implementation of Iterators

In the `Utils` module of PySE we have implemented a few more useful tools. The most important are the iterators we use in stencilsets to describe sets of indices in grids where a stencil should be applied.

Flexibility regarding how stencilsets can be created is one of the main strengths of our framework, because it offers a powerful tool for specifying and solving a large class of problems. It can be a difficult task to select a subset of indices from the lattice grid for a stencil, at least in the multidimensional case, when a nontrivial subset is called for. Hence, we have implemented a set of iterator classes that should give the required flexibility, while maintaining a relatively simple user interface. In addition, the iterators we have implemented handle the requirements of parallelization. The implemented iterators are the basis for the methods in `Grid` that return sets of indices (see Section 5.1), and for the `createNeumannBoundary` function. For a large range of cases, these methods should be sufficient, but for even more flexibility, the classes presented here can be used directly. For the examples presented in this paper, we will adhere to the interface provided by `Grid` and the create function for Neumann conditions.

There are four different iterators implemented in the `Utils` module. First, there is a general class named `tupleIterator`, which is used by the other more specialized iterators, but can also be used directly. The name reflects the fact that the elements produced are indices given as tuples. Figure 15 shows how a `tupleIterator` is created, as well as sample usage. Arguments are the number of space dimensions and the range of the tuples generated, specified by the minimum and maximum corners. Using `tupleIterator`, the user can generate tuples for any "box-shaped" region in a grid, in order to assign a particular stencil to this region. The dimension of the "box" does not need to be the same as that of the grid; a 2D region in a 3D grid can be selected by letting any number of values in the minimum and maximum corners specified be the same. An example is shown in Figure 15, where a region in the x-y plane (z=0) is selected in terms of 3D indices, which could be used to select a region from a 3D grid.

In addition to `tupleIterator`, the following iterator classed are imple-

mented: `innerTupleIterator` for inner nodes, `boundaryTupleIterator` for boundary nodes and `cornerTupleIterator` for corner nodes. The signature for the constructor is the same for all these iterators; the number of space dimensions, the minimum tuple, and the maximum tuple.

Finally, there is a special iterator implemented for circular regions. This iterator is special in that it wraps any of the other iterators, and restricts the indices to those either outside or inside a circle or sphere with given radius and centre. While all the other iterators only produce sets of indices, the circle iterator needs to relate to some grid, in order to compute the distance between a given point and the centre of the circle. The class is named `circleIterator`, and the constructor requires four arguments: an iterator providing the indices, a grid, the center, and the radius. In addition, the direction can be specified as either `'in'` or `'out'`, `'in'` is the default.

6 Case studies

We now present three cases that provide a better understanding of how PySE can be used. In the first case, we investigate the convergence order of different finite difference schemes for a wave equation in two space dimensions. The second case concerns another application of the wave equation in the simulation of ultrasonic acoustical waves, commonly known as ultrasound. In this case, the wave equation is solved in three space dimensions. In the last case, we revisit the heat conduction problem in equations (7a-7c) with more focus on performance.

6.1 Wave equation in 2D - a convergence study

Consider the wave equation:

$$\begin{aligned} u_{tt} &= \nabla^2 u, \quad (x,t) \in \Omega \times \mathbb{R}^+, \\ u(x,0) &= f(x), \quad x \in \Omega, \\ u_t(x,0) &= 0, \quad x \in \Omega. \end{aligned} \tag{10}$$

Boundary conditions in our experiments will be either the homogeneous Dirichlet condition,

$$u(x,t) = 0, \quad (x,t) \in \partial\Omega \times \mathbb{R}^+, \tag{11}$$

or the homogeneous Neumann condition,

$$\frac{\partial u(x,t)}{\partial n} = 0, \quad (x,t) \in \partial\Omega \times \mathbb{R}^+. \tag{12}$$

Further, we assume that $\Omega = [-1,1] \times [-1,1]$. Using a second-order difference in time, we obtain the semi-discrete form:

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} = \nabla^2 u^n, \tag{13}$$

which gives the explicit scheme

$$u^{n+1} = 2u^n - u^{n-1} + \Delta t^2 L_\Delta u^n. \tag{14}$$

Here, $L_\Delta u$ represents the discretization of the Laplace operator.

If we apply the centred difference for the Laplace operator, as we did for the heat conduction discussed in Section 2 (see e.g., equation (2)), the error should theoretically be of second order in time and space, $e = \mathcal{O}(\Delta t^2, h^2)$. Here, and in the rest of this section, we assume $h = \Delta x = \Delta y$. By instead using a 9-point

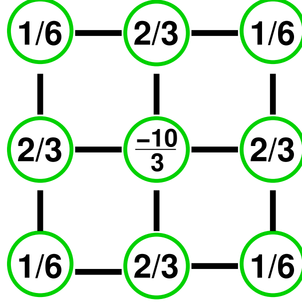

Figure 16: 9-point stencil of the compact fourth-order scheme for the Laplace operator in 2D.

scheme for the Laplace operator (see [19, ch. 7.3]), we obtain theoretically a scheme with error $e = \mathcal{O}(\Delta t^2, h^4)$. The coefficients in the 9-point stencil will be as in Figure 16.

Convergence for the problem can be improved even further by using a higher-order method for the time derivative. We will use a fourth-order Nyström method [14, ch. II.14] for the time derivative in order to get an error estimate $e = \mathcal{O}(\Delta t^4, h^4)$.

For the problem with a Dirichlet boundary condition we expect to see close to optimal convergence order in the experiments. For the problem with a Neumann boundary condition, we will use the `createNeumannBoundary` function in PySE, which implements a second-order approximation of the boundary condition. Hence, theoretically, we obtain a second-order scheme for the whole domain, even if a fourth-order 9-point scheme is used for the inner nodes in the domain.

For both the Dirichlet and Neumann boundary condition cases, we can choose the analytical solution of (10), and derive the initial function $f(x)$ and the boundary condition. The error can then be computed as the sum over all time steps of the discrete L_2 norm of the difference between the analytical and approximative solutions, multiplied with Δt,

$$e = (\Delta t \sum_n h^2 \sum_{i,j} (u_{i,j}^n - a_{i,j}^n)^2)^{1/2}.$$

Here, $u_{i,j}$ is the approximation of the solution and $a_{i,j}$ is the analytical solution in the nodes of the grid. The error estimate for our problem can be written in the form

$$e = A\Delta t^\alpha + Bh^\beta.$$

If we assume $\alpha = 2$ and $\beta = 4$, and choose $\Delta t \leq h^2$, we get

$$e = A\Delta t^2 + Bh^4 = A(h^2)^2 + Bh^4 = Ch^4.$$

Hence, if Δt is chosen such that $\Delta t \leq h^2$, we expect to observe fourth-order convergence when a fourth-order approximation is used in space, even if a second-order scheme is used for the time derivative. For second-order approximation in both space and time, we get

$$e = Ch^2$$

by choosing $\Delta t \leq h$, and hence we expect to observe second-order convergence in this case. In the same way, fourth-order convergence is expected for the

h	5-pt	9-pt	9-pt + N
1.6e-01	1.085e-08	1.073e-09	1.073e-09
8.0e-02	3.211e-09	1.218e-10	1.218e-10
4.0e-02	8.232e-10	8.068e-12	8.067e-12
2.0e-02	2.071e-10	5.110e-13	5.103e-13
Least square fit, C:	4.22e-07	1.41e-06	1.41e-06
Least square fit, α:	1.9	3.8	3.8

Table 4: Error and convergence rate for the wave equation with Dirichlet boundary condition, with $\Delta t = 2.5e-05$, such that $\Delta t \leq h^2$ for the smallest h. Eight time steps performed.

h	5-pt	9-pt	9-pt + N
1.6e-01	1.068e-03	1.063e-04	1.062e-04
8.0e-02	3.153e-04	1.273e-05	1.220e-05
4.0e-02	8.032e-05	1.541e-06	8.504e-07
2.0e-02	1.973e-05	7.980e-07	6.461e-08
Least square fit, C:	4.30e-02	7.63e-03	1.08e-01
Least square fit, α:	2.0	2.5	3.7

Table 5: Error and convergence rate for the wave equation with Dirichlet boundary condition, with $\Delta t = 2.5e-03$, such that $\Delta t \leq h$ for the smallest h. Eight time steps performed.

$\mathcal{O}(\Delta t^4, h^4)$ scheme, which is obtained by combining the 9-point scheme in space with a fourth-order Nyström method in time.

Based on the discussion above, we we assume an error model of the form

$$e = Ch^{\alpha}$$

for our experiments, and use a linear least square fit to compute C and α from the experimental data. We use the following labels for the experiments: *5-pt* for the scheme based on the centered difference in time and the 5-point stencil for Laplace, *9-pt* for the scheme based on the 9-point stencil for Laplace and the same difference in time, and *9-pt + N* denotes that the fourth-order Nyström scheme in time and the 9-point stencil for Laplace is used. The results of our experiments are given in Tables 4-7. As expected for the Dirichlet boundary condition problem, we get close to second-order convergence for the *5-pt* scheme when $\Delta t \leq h$, and fourth-order convergence for *9-pt* and *9-pt + N* when $\Delta t \leq h^2$. For *9-pt + N* we get close to fourth-order convergence also when $\Delta t \leq h$ for the Dirichlet problem, while the convergence rate of *9-pt* is close to 2 in this case.

The convergence rate for the Neumann boundary condition problem degrades for the schemes using fourth-order approximation in space, because the Neumann condition is implemented with only a second-order approximation. However, it is interesting to observe that we get third-order convergence for both *9-pt* and *9-pt + N* in this case whenever $\Delta t \leq h$. This indicates that the second-order approximation of the Neumann boundary condition does not degrade the convergence order as much as the theory suggests.

h	5-pt	9-pt	9-pt + N
1.6e-01	1.252e-08	2.008e-09	2.008e-09
8.0e-02	3.468e-09	3.307e-10	3.307e-10
4.0e-02	8.562e-10	4.217e-11	4.218e-11
2.0e-02	2.113e-10	5.309e-12	5.309e-12
Least square fit, C:	5.38e-07	4.94e-07	4.94e-07
Least square fit, α:	2.0	2.9	2.9

Table 6: Error and convergence rate for the wave equation with Neumann boundary condition, with $\Delta t = 2.5e-05$, such that $\Delta t \leq h^2$ for the smallest h. Eight time steps performed.

h	5-pt	9-pt	9-pt + N
1.6e-01	1.233e-03	1.981e-04	1.980e-04
8.0e-02	3.405e-04	3.234e-05	3.204e-05
4.0e-02	8.354e-05	3.400e-06	3.724e-06
2.0e-02	2.012e-05	8.874e-07	3.330e-07
Least square fit, C:	5.47e-02	2.83e-02	7.80e-02
Least square fit, α:	2.0	2.7	3.1

Table 7: Error and convergence rate for the wave equation with Neumann boundary condition, with $\Delta t = 2.5e-03$, such that $\Delta t \leq h$ for the smallest h. Eight time steps performed.

6.1.1 On implementation

Consider the discrete form given in (14). This scheme can be rewritten as

$$u^{n+1} = (2 + \Delta t^2 L_\Delta)u^n - u^{n-1} \tag{15}$$

In the examples we have seen so far, the stencilset has always operated on a single field of unknown values, u^n, in order to compute an update, using either explicit methods or an iterative solver. From (15) it is clear that we need to work with two different versions of u, the current and the previous time step, to compute values for the next time step. As a stencilset is only able to operate on a single field, we need to use two stencilsets, one for the term $2+\Delta t^2 L_\Delta$ and one for -1. The following excerpt from the code shows the basis of the wave solver:

```
g = Grid(domain=([-1,1],[-1,1]), division=[m.m])
A = StencilSet(g)
B = StencilSet(g)

lap_5pt = Stencil(nsd=2,                               \
      nodes={            (0,1):  1.,                   \
            (-1,0): 1., (0,0):  -4., (1,0): 1., \
                         (0,-1): 1.})

lap_9pt = Stencil(nsd=2,                                      \
      nodes={(-1,1):  1./6, (0,1):  2./3,   (1,1):  1./6, \
            (-1,0):  2./3, (0,0):  -10./3, (1,0):  2./3, \
```

```
              (-1,-1): 1./6, (0,-1): 2./3,   (1,-1): 1./6})

# The -1*uprev term:
uprev_stencil = Stencil(nsd=2, nodes={(0,0): -1.0})
A.addStencil(uprev_stencil,g.innerPoints())

# The (2 + dt^2L) term:
u_stencil = Stencil(nsd=2, nodes={(0,0): 2.0})
if use_5pt:
   fullu_stencil = u_stencil + (dt**2/g.dx**2)*lap_5pt
elif use_9pt:
   fullu_stencil = u_stencil + (dt**2/g.dx**2)*lap_9pt
B.addStencil(fullu_stencil, g.innerPoints())

# Go parallel on parallel computers: g.partition(B)
A.doInitParallel()

# Assume u and uprev initialized according to # initial
conditions. rt = dt while rt < T:
   unew = B(u) + A(uprev)
   # Increment time, and switch fields:
   rt += dt
   uprev = u
   u = unew
```

In order to use a parallel computer, the call to `partition` on the `Grid` instance is inserted as usual, with the stencilset `B` as argument. However, as there is another stencilset in this problem, `A`, an explicit call to `doInitParallel` on this stencilset is inserted as well. This is required, because there is no relation between the stencilsets. This may change in the future.

The complete code for the simulator is more involved, because it contains the option of selecting either Dirichlet or Neumann boundary conditions in the simulation, as well as computation of the error norm. In addition, we discovered during the process that the 9-point scheme for Laplace does not automatically yield fourth-order convergence. In order to achieve fourth-order convergence, the scheme must be applied to the modified equation

$$\mathcal{M}u_{tt} = \nabla^2 u \tag{16}$$

where $\mathcal{M}$ is defined by the stencil shown in Figure 17. According to [19], the difference between the solution of (10) and (16) is of order $\mathcal{O}(h^4)$, so the solution of (16) should still give an error of fourth order when compared to analytical solutions of (10). By writing (16) as

$$u_{tt} = \mathcal{M}^{-1}\nabla^2 u, \tag{17}$$

we can proceed as before to obtain the discrete scheme

$$u^{n+1} = 2u^n - u^{n-1} + \Delta t^2 \mathcal{M}^{-1} L_\triangle u^n. \tag{18}$$

As $\mathcal{M}^{-1}$ is not known, and we generally avoid inverting a matrix directly, we write the problem as

$$\mathcal{M}u^{n+1} = (2\mathcal{M}u^n + \Delta t^2 L_\triangle)u^n - \mathcal{M}u^{n-1}. \tag{19}$$

Due to the $\mathcal{M}$ term on the left-hand side, this is no longer an explicit scheme. Fortunately, the condition number of $\mathcal{M}$ is small, so the problem can be solved effectively with both the Jacobi and Conjugate gradient solvers discussed in Section 5.6.3.

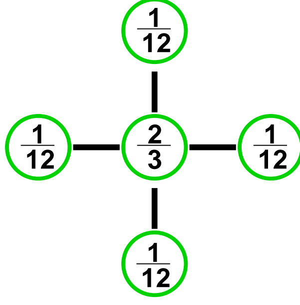

Figure 17: The averaging mass–stencil $\mathcal{M}$ that must be used together with the 9-point scheme for Laplace

6.2 Ultrasound

In [34], finite element simulation of ultrasound is discussed. For ultrasound simulation in the human body, the finite element method can be beneficial due to complex geometry of the body. However, for most of the examples presented in [34], use of the finite element method is not a requirement. Therefore, we want to implement a finite difference simulator for ultrasound in linear media, using PySE.

The linear model for ultrasound is similar to the wave equation discussed in the previous section:

$$\begin{aligned}
u_{tt}(x,t) &= c^2\nabla^2 u(x,t), && (x,t)\in\Omega\times\mathbb{R}^+,\\
u(x,0) &= 0, && x\in\Omega,\\
u_t(x,0) &= 0, && x\in\Omega,\\
u(x,t) &= T(x,t), && (x,t)\in\partial\Omega_T\times\mathbb{R}^+,\\
\frac{\partial u}{\partial n} &= -\frac{1}{c}\frac{\partial u}{\partial t}, && (x,t)\in\partial\Omega_{T^c}\times\mathbb{R}^+.
\end{aligned} \tag{20}$$

Here, c is the speed of sound, $\partial\Omega_T$ is the region on the boundary for the transducer that generates the ultrasound waves, and $T(x)$ is the transducer function. On the rest of the boundary, $\partial\Omega_{T^c}$, an absorbing boundary condition is used, in the form of a Neumann condition.

The solver for the ultrasound problem is implemented along the same lines as the wave equation discussed above, using the centered differences for the Laplace operator, resulting in a 7-point stencil in three-dimensional space. For the time derivative, the same centred difference as in (14) is used. New in this problem is the mix of Dirichlet and Neumann conditions on different parts of the boundary. In addition, the absorbing boundary condition includes the field u, which means that we need some mechanism to include instances of `Field` in the function supplied to `createNeumannBoundary`.

The transducer in our simulation is specified as a circle on the x-y face of the domain, with centre in origin, and a given radius. We use a bessel function as the generating function for the pressure distribution on the transducer, and generate a given number of pulses on the transducer:

```
def transducerGenerator(a0, w, a, centre, radius, pulses, init_pressure):
   def transducer(x,y,t):
      if t < pulses*(1./w):
         r = distance(centre, (x,y,0))
         return init_pressure + a0*sin(2*pi*w*t)*j0(a*r)
```

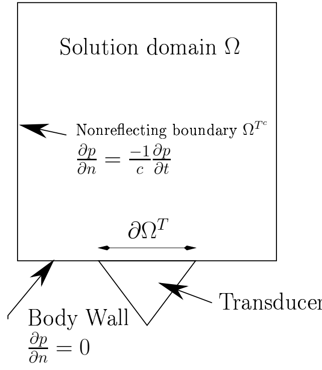

Figure 18: Domain and boundary conditions for the Ultrasound experiment. We show the x-z plane in this figure.

```
        else:
            return init_pressure
    return transducer

rt = 0.0
transducer = transducerGenerator(tr_scale, w, tr_a, tr_center, tr_radius, \
    tr_pulses, init_pressure)
transducer_now = lambda x,y,z: transducer(x,y,rt)
transducer_bc = DirichletBoundary(nsd=3, transducer_now)
```

We have implemented the transducer with a function that generates the transducer function, based on the required parameters. When working in 3D, the source function in the Dirichlet condition will be called with `x`, `y`, and `z` as arguments. Our transducer is two-dimensional, and only need `x` and `y`, but in addition the time must be given. Therefore, we wrap the generated transducer function using a *lambda*[2] function, which throws away the `z` argument, and instead supplies the time. For this to work, it is crucial that the time variable used, `rt`, is updated in the time loop later on.

The absorbing boundary condition that we will use as the Neumann condition for the rest of the boundary includes the time derivative of the unknown field:

$$\frac{\partial u}{\partial n} = -\frac{1}{c}\frac{\partial u}{\partial t}, \quad x \in \partial\Omega_{T^c}. \tag{21}$$

By using a first-order difference for the time derivative, we obtain

$$\frac{\partial u}{\partial n} = -\frac{1}{c\Delta t}(u^n - u^{n-1}). \tag{22}$$

We implement the boundary condition as shown in the following code:

```
# Initialize a Field for the absorbing boundary
# To be filled with u - uprev later on.
absorbing_bc_field = Field(g)
call_absbc = lambda x,y,z: (-1./(c*dt))* \
    absorbing_bc_field.getValByPoint(x,y,z)
```

The method `getValByPoint` returns an approximate value for the field at the given point. If the point matches an index in the field, the corresponding nodal value is returned.

[2] *lambda* is a keyword in Python, which creates an anonymous function. Such anonymous functions can be used exactly as other functions.

Assume that the stencilsets A and B are initialized with stencils for the inner nodes as in Section 6.1.1. We can then add boundary conditions as follows:

```
ez = 0.5*g.dz
front = ((-xmax,xmax),(-ymax,ymax),(0,0))
sides = ((-xmax,xmax),(-ymax,ymax),(0+ez,zmax))

# Set the transducer Dirichlet b.c.
B.addStencil(transducer_bc, g.boundary(region=front, type='circle',\
   center=tr_center, radius=tr_radius, direction='in'))

# Set the Neumann b.c.
B += createNeumannBoundary(fullu_stencil, g, call_absbc, \
   region=front, type='circle', center=tr_center, radius=tr_radius,\
   direction='out')
B += createNeumannBoundary(fullu_stencil, g, call_absbc, \
   region=sides)
```

First, we set the transducer condition inside the circle that defines the transducer; then we set the absorbing boundary condition outside the same circle. Finally, we set the absorbing boundary condition on the other sides of the domain. The main loop of the solver will be similar to the wave simulator in the previous section:

```
# Initialize fields:
uprev = Field(g)
uprev.fill(init_pressure)
u = 0.5*B(uprev)

# Update the absorbing b.c. field:
absorbing_bc_field = u - uprev
B.updateDataStructures()
rt = dt
while rt < T:
   unew = B(u) + A(uprev)
   uprev = u
   u = unew
   # update the absorbing b.c. field:
   absorbing_bc_field = u - uprev
   B.updateDataStructures()
   rt += dt
```

The main difference between this solver, and the wave solver we have seen before, is the call to updateDataStructures on the stencilset. As mentioned in Section 5.3, we need to update data structures in the stencilset if any of the coefficient or source functions are time-dependent. Here, both the transducer and the absorbing boundary condition are time-dependent. Note that there is no need to call this method on the other stencilset, A, because there are no time-dependent coefficient or source function in the stencils in that stencilset.

6.3 Heat conduction revisited

In Example 2 in Section 4 we studied the performance of a solver for heat conduction. In Table 2 we saw that the C solver was significantly faster than the Python solver, but we also saw in Table 3, where a problem without a

time-dependent source function was solved, that the C and Python solvers can perform comparably.

We will now show how the original solver for the heat conduction problem can be optimized. As the main bottleneck seems to be how the time-dependent source function is handled, we focus on that first. In the original solver, we included the source directly in the main stencil. Instead we can use a field for the source. By modifying the source function such that it works with coordinate vectors instead of points, we can use the functionality of NumPy to enter data into the field efficiently. This is implemented in the `fill_vec` method of `Field`.

The same trick can be used for the time-dependent Dirichlet boundary condition. To improve the initialization time, we can also do this with the initial function. After implementing these changes, the stencilsets must also be changed accordingly. But before we discuss those changes, we address another potential performance bottleneck. The call operator of a stencilset, as in the statement `v = A(u)`, is basically a matrix-vector product, and also a vector-vector addition, if any stencil in the set has a source. While the call operator yields very nice syntax, there is some overhead in the creation of a new instance of `Field` for the result, as well as in the copying of numerical arrays. In addition, given that the data is stored in a numerical array inside fields, we should be able to use a more direct approach. For this purpose, `Field` implements a method `direct_matvec`, which takes a numerical array as data, and returns a numerical array. However, the use of this method does not ensure that the stencilset operator works correctly in a parallel context. Therefore, `Field` also implements the method `updateField`, which can be used to ensure consistency manually on a parallel computer.

As before, the stencilset `A` will hold the main stencil for the problem for all inner nodes, but the source function f is now removed from the stencil. We want to create the stencilset `S` and `B` such that `S` can be applied to the source field `F` and `B` to the boundary field `bF`, and the results added together with `A(u)` to provide an update (with the original call syntax):

```
u = A(u) + S(F) + B(bF)
```

We can achieve this in the following way: `S` is the identity for all inner nodes, and zero for boundary nodes, `B` is the identity for boundary nodes, and zero for inner nodes, and `A` uses the problem stencil for inner nodes and zero for the boundary:

```
A = StencilSet(g)
# Assume that s is built as before
A.addStencil(s,g.innerPoints())

# identity:
idstencil = Stencil(nsd=2,nodes={(0,0): 1.0})

# sources:
S = StencilSet(g)
S.addStencil(idstencil,g.innerPoints())

# boundary
B = StencilSet(g)
B.addStencil(idstencil,g.boundary())
```

Next, we need to change source, boundary, and initial functions such that a vectorized fill algorithm can be used for fields. Consider the boundary condition function:

```
bf = lambda x,y: exp(-rt)*sin(pi*x)*cos(pi*y)
```

If we use the trigonometric functions from NumPy instead, they will work directly with vector arguments:

```
bf = lambda x,y: exp(-rt)*Numeric.sin(pi*x)*Numeric.cos(pi*y)
```

The source and initial functions are changed accordingly. In order to perform matrix-vector operations directly using `direct_matvec`, we need to access the array that contains the data inside instances of `Field`. This array is named `data`. We can now solve the problem as shown in the following code; note that the complete revised code is included in Appendix B:

```
# the unknowns, filled with initial condition
u = Field(g)
u.fill_vec(initf)
d = Numeric.array(u.data)

# Source and boundary fields:
F = Field(g)
bF = Field(g)
dF = Numeric.array(F.data)
dbF = Numeric.array(bF.data)

# build the datastructures in StencilSets:
A.buildMatrixOperator(u)
S.buildMatrixOperator(u)
B.buildMatrixOperator(u)

rt = dt while rt < T:
   # Vectorized computation of source and boundary
   F.fill_vec(dt_f)
   bF.fill_vec(bf)
   # Get the computed values
   dF[:] = F.data
   dbF[:] = bF.data

   # compute an explicit step:
   d[:] = A.direct_matvec(d) + S.direct_matvec(dF) + B.direct_matvec(dbF)

   if u.isParallel:
      # copy data back in field, and update:
      u.data[:] = d
      u.updateField()
      # and copy back into numeric array:
      d[:] = u.data[:]
   rt += dt
```

The only new method introduced here is `buildMatrixOperator`. When using the regular user interface, the call operator of a stencilset, e.g., `A(u)`, causes the sparse matrix structure and accompanying vectors, to be built. As we now use the `direct_matvec` method instead, we need to build these structures upfront. This is the purpose of the `buildMatrixOperator` method. The slice notation `d[:]` is used to copy values in numeric arrays into existing data structures, rather than creating new storage.

# cpus:	1	2	4	8	16	24	32
1000 × 1000, 160 time step:	365.0	185.4	93.90	46.99	23.51	16.05	12.52
speed–up:	1	1.97	3.89	7.77	15.5	22.7	29.2
1500 × 1500, 240 time step:	1226	620.0	315.7	160.9	78.42	52.33	40.73
speed–up:	1	1.98	3.88	7.62	15.6	23.4	30.1

Table 8: CPU time in seconds and corresponding speed-up numbers

Problem size	runtime	rel. 1-cpu Python	rel. 32-cpu Python
1000 × 1000, 160 time steps:	107.3	3.40	0.12
1500 × 1500, 240 time steps:	362.4	3.38	0.11

Table 9: CPU time in seconds for the time loop in the solver implemented in C, as well as speed up relative to the Python solver running on one and 32 processors.

In Table 8, we list the CPU time for the main computational loop of the solver, excluding initialization. Compared to the numbers for the same problem in Section 4, there is a significant performance gain. Regarding parallel efficiency, we again observe very good speed up. When compared against the C program that solves the same problem, the performance gain from the rewrite is even more visible. Now, the Python solver running on one CPU is just 3.4 times slower than the C solver, while the performance of the original solver was 75 times slower than C. When more processors are used, the Python solver can solve the problem significantly faster than the scalar C solver. To obtain the performance increase, we had to implement code that is more difficult to read and use some less intuitive constructs. In addition, we had to fill the source and boundary fields in a vectorized manner, which is less flexible than assigning values point by point. Nevertheless, the approach used is general and can be applied to many solvers prototyped with PySE, to achieve better performance.

7 Concluding remarks and future development

We have implemented a framework for creating FDM-based solvers for PDEs. Centred around four abstractions (grid, field, stencil and stencilset), we created a user interface that yields clean, uncluttered and flexible solvers. The simple user interface both allows the rapid prototyping of solvers for PDEs and gives the user a flexible tool for experimenting with different FDMs for the problem at hand, even interactively if that is desired.

While numerical efficiency not have been our primary focus, we saw in the case studies how the advanced user interface can be used to achieve quite good performance. A natural process will be to prototype solvers using the basic syntax, which will give code that is very close to pseudo code. If the prototyped solver eventually evolves into a simulator that needs to be run for larger problems, or for many time steps, the more advanced functionality can be used for increased performance. In addition, parallel computers can be put into use without any extra effort.

There are several issues that should be addressed for the development of future versions of PySE. The most important of these are the following: support for higher-order methods in the methods that yield boundary and interior

nodes for grids; support for non-linear problems; and support for preconditioned linear solvers. In addition, support for more flexible boundary conditions and more complex geometries will be important. The performance issues should be addressed further, to improve the performance achieved with the basic user interface, such that the more advanced operations presented in the case studies in Section 6 become unnecessary.

References

[1] The PyFDM webpage. http://pyfdm.sourceforge.net, 2005.

[2] A. M. Bruaset. *A Survey of Preconditioned Iterative Methods.* Addison-Wesley Pitman, 1995.

[3] Federico Bassetti, David Brown, Kei Davis, William Henshaw, and Dan Quinlan. OVERTURE: An object-oriented framework for high-performance scientific computing. In *Proceedings of Supercomputing'98 (CD-ROM).* ACM SIGARCH and IEEE, 1998.

[4] Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI.* Oxford University Press, 2004.

[5] X. Cai, H. P. Langtangen, and H. Moe. On the performance of the python programming language for serial and parallel scientific computations. *Scientific Programming*, 13:31–56, 2005.

[6] Xing Cai, Elizabeth Acklam, Hans Petter Langtangen, and Aslak Tveito. Parallel Computing. In H. P. Langtangen and A. Tveito, editors, *Advanced Topics in Computational Partial Differential Equations - Numerical Methods and Diffpack Programming*, LNCSE, pages 1–56. Springer, 2003.

[7] The Diffpack website. http://www.diffpack.com.

[8] Lisandro Dalcin et al. Mpi for python. http://sourceforge.net/projects/mpi4py, 2005.

[9] R.D. Falgout, J.E. Jones, and U.M. Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. In A. M. Bruaset, P. Bjørstad, and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers.* Springer, To appear.

[10] Femlab. http://www.femlab.com.

[11] freeFEM. http://www.freefem.org.

[12] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing, (2nd. ed.).* Addison-Wesley, 2003.

[13] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI - Portable Parallel Programming with the Message-Passing Interface, (2nd ed.).* The MIT Press, 1999.

[14] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I, Nonstiff Problems.* Springer, 2000. Second revised edition.

[15] Konrad Hinsen. ScientificPython. http://dirac.cnrs-orleans.fr/ScientificPython/.

[16] Konrad Hinsen, Hans Petter Langtangen, Ola Skavhaug, and Åsmund Ødegård. Using BSP and Python to Simplify Parallel Programming. *Future Generation Computer Systems*, 2004.

[17] E. N. Houstis, J. R. Rice, S. Weerawarana, A. C. Catlin, P. Papachiou, K.-Y. Wang, and M. Gaitatzes. PELLPACK: A problem solving environment for PDE based applications on multicomputer platforms. *ACM Transactions on Mathematical Software*, 24(1):30–73, 1998.

[18] Scalable Linear Solvers and hypre. http://www.llnl.gov/CASC/linear_solvers.

[19] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge University Press, 1996.

[20] H. P. Langtangen. *Python Scripting for Computational Science*, volume 3. Springer-Verlag, 2004. 725 pages.

[21] Alex Martelli. *Python in A Nutshell*. O'Reilly & Associates, Inc, 2003.

[22] Partial Differential Equation Toolbox for Matlab. http://www.mathworks.com/products/pde/.

[23] Michael Thuné, Eva Mossberg, Peter Olsson, Jarmo Rantakokko, Krister Åhlander, and Kurt Otto. Object–Oriented Construction of Parallel PDE Solvers. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 203–226. Birkhäuser, 1997.

[24] Patrick Miller. The pyMPI project. http://pympi.sourceforge.net.

[25] The Message Passing Interface (MPI) Forum. http://www.mpi-forum.org.

[26] Ole Nielsen. The Pypar website. http://datamining.anu.edu.au/ ole/pypar/.

[27] Travis Oliphant. Numerical python web page. http://numeric.scipy.org, 2005.

[28] Overture. http://www.llnl.gov/CASC/Overture/.

[29] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, 1997.

[30] The PETSc website. http://www-fp.mcs.anl.gov/petsc.

[31] G. van Rossum and F. L. Drake. Python reference manual. http://docs.python.org/ref/ref.html, 2005.

[32] W. Hackbusch. *Iterative Solution of Large Sparse Systems of Equations*. Springer-Verlag, 1994.

[33] B. Wilkinson and M. Allen. *Parallel Programming - Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 1999.

[34] Åsmund Ødegård, Paul D. Fox, Sverre Holm, and Aslak Tveito. Finite Element Modeling of Pulsed Bessel Beams and X-Waves using Diffpack. In *Proc. of 25th Int. Acoustical Imaging Symp*, pages 59–64, Bristol, UK, 2000.

A Complete code for problem (7a-7c)

```
#!/usr/bin/env python
# -*- coding: iso8859-1 -*-
#
# (c) Copyright 2005
#     Author: Åsmund Ødegård
#     Simula Research Laboratory AS

from pyse import *

from math import exp,sin,cos,pi,sqrt

def runstandalone():
    T=0.0025
    n=20
    m=40

    parasize = pypar.size()
    myrank = pypar.rank()

    # If we are parallel, and are using mpich MPI, we need to distribute the values
    # for the options, because they are only read on the first cpu when using mpich.

    if myrank == 0:
        opts = sys.argv[1:]
        numopts = len(opts)
        i = 0
        while i < numopts:
            opt = opts[i]
            if opt == "-t":
                T = float(opts[i+1])
                i += 1
            if opt == "-n":
                n = int(opts[i+1])
                i += 1
            if opt == "-m":
                m = int(opts[i+1])
                i += 1
            i += 1

    if parasize > 1:
        allopts = Numeric.zeros(3,typecode='d')
        if myrank == 0:
            allopts = Numeric.array((T,n,m))
        pypar.broadcast(allopts,0)
        if myrank != 0:
            T = float(allopts[0])
            n = int(allopts[1])
            m = int(allopts[2])

    # Solve, return the solution and a reference to the grid
    solution,g = solve(T,n,m)

    # The analytical solution
    analytical = lambda x,y: exp(-T)*sin(pi*x)*cos(pi*y)

    u_a = Field(g)
    u_a.fill(analytical)

    # The error
    error = u_a - solution
    error.plot(title='Error')
    solution.plot(title='Solution')
    u_a.plot(title='Analytical solution')

    errornorm = sqrt(error.inner(error))
    if g.myrank == 0 or g.myrank == -1:
        print "The norm of the error is: ",errornorm

    if solution.isParallel:
        pypar.finalize()

def usage(T,n,m):
    print "-t T: solve from 0 to T (default %s)" % (T)
    print "-n steps: Split time in so many steps (default %s)" % (n)
    print "-m div: Split each direction in so many nods (default %s)" % (m)
```

```
def solve(T=1.0, n=1000, m=100):
    # solve the problem from 0 to T, dt = 1/1000, dx = 1/100
    print "Solve the problem from 0 to %s in %s steps, %s grid-nodes" % (T,n,m)

    # Running time:
    rt = 0.0

    # compute dt:
    dt = T/(1.0*n)

    # Create a grid
    g = Grid((m,m),2,([0.0,1.0],[0.0,1.0]))

    dx = g.dx
    h = dt/(g.dx*2)

    # Functions I need (use rt, dt, and dx)
    dt_f = lambda x,y: dt*(-exp(-rt)*sin(pi*x)*cos(pi*y) - \
           pi*exp(-rt)*cos(pi*x)*cos(pi*y) + \
           pi*exp(-rt)*sin(pi*x)*sin(pi*y) + \
           2*(x + y)*pi*pi*exp(-rt)*sin(pi*x)*cos(pi*y))
    # need only two k-functions, as dx is the same in both directions!
    k_p = lambda x,y: x + y + 0.5*dx
    k_m = lambda x: x + y - 0.5*dx
    # boundary fu:
    bf = lambda x,y: exp(-rt)*sin(pi*x)*cos(pi*y)
    # initical condition:
    initf = lambda x,y: sin(pi*x)*cos(pi*y)

    # build the stencil
    s = Stencil(nsd=2,varcoeff=True,source=dt_f)
    s.addNode((0,-1),[lambda *x: h*k_p(*x)])
    s.addNode((-1,0),[lambda *x: h*k_p(*x)])
    s.addNode((0,0),[lambda *x: 1.0 - 2.0*h*(k_m(*x) + k_p(*x))])
    s.addNode((1,0),[lambda *x: h*k_m(*x)])
    s.addNode((0,1),[lambda *x: h*k_m(*x)])

    bs = DirichletBoundary(2,bf)

    A = StencilSet(g)
    A.addStencil(s,g.innerPoints())
    A.addStencil(bs,g.boundary())

    # Go parallel if we are on parallel stuff
    g.partition(A)

    u = Field(g)
    u.fill(initf)

    #u.plot(title='Initical condition')

    while rt < T:
        u = A(u)
        rt += dt
        A.updateSourceDataStructures()

    #u.plot(title='Example')

    return u,g

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

B Complete code for problem (7a-7c) Fast version

Here, we give the complete code for the faster solver for heat conduction discussed in Section 6.3.

```
#!/usr/bin/env python
# -*- coding: iso8859-1 -*-
#
# (c) Copyright 2005
#     Author: Åsmund Ødegård
#     Simula Research Laboratory AS
```

```
from pyse import *

import sys
from math import exp,sin,cos,pi,sqrt
import time

def runstandalone():
    T=0.0025
    n=20
    m=40

    parasize = pypar.size()
    myrank = pypar.rank()

    # If we are parallel, and are using mpich MPI, we need to distribute the values
    # for the options, because they are only read on the first cpu when using mpich.

    if myrank == 0:
        opts = sys.argv[1:]
        numopts = len(opts)
        i = 0
        while i < numopts:
            opt = opts[i]
            if opt == "-t":
                T = float(opts[i+1])
                i += 1
            if opt == "-n":
                n = int(opts[i+1])
                i += 1
            if opt == "-m":
                m = int(opts[i+1])
                i += 1
            i += 1

    if parasize > 1:
        allopts = Numeric.zeros(3,typecode='d')
        if myrank == 0:
            allopts = Numeric.array((T,n,m))
        pypar.broadcast(allopts,0)
        if myrank != 0:
            T = float(allopts[0])
            n = int(allopts[1])
            m = int(allopts[2])

    # Solve, return the solution and a reference to the grid
    solution,g,T = solve(T,n,m)

    # The analytical solution
    analytical = lambda x,y: exp(-T)*numarray.sin(pi*x)*numarray.cos(pi*y)

    u_a = Field(g)
    u_a.fill_vec(analytical)

    # The error
    error = u_a - solution
    #error.plot(title='Error')
    #solution.plot(title='Solution')
    #u_a.plot(title='Analytical solution')

    errornorm = sqrt(g.dx*g.dy*error.inner(error))
    if g.myrank == 0 or g.myrank == -1:
        print "The norm of the error is: ",errornorm

    if solution.isParallel:
        pypar.finalize()

def usage(T,n,m):
    print "-t T: solve from 0 to T (default %s)" % (T)
    print "-n steps: Split time in so many steps (default %s)" % (n)
    print "-m div: Split each direction in so many nods (default %s)" % (m)


def solve(T=1.0, n=1000, m=100):
    # solve the problem from 0 to T, dt = 1/1000, dx = 1/100
    myrank = pypar.rank()

    if myrank <= 0:
        print "Solve the problem from 0 to %s in %s steps, %s grid-nodes" % (T,n,m)
```

```
# Running time:
rt = 0.0

# compute dt:
dt = T/(1.0*n)

# Create a grid
g = Grid((m,m),([0.0,1.0],[0.0,1.0]))

dx = g.dx
h = dt/(dx*dx)

# Functions for source, coefficients, boundary, initial condition
dt_f = lambda x,y: dt*exp(-rt)*(-pi*numarray.cos(pi*x)*numarray.cos(pi*y) \
         + pi*numarray.sin(pi*x)*numarray.sin(pi*y) \
         + (2.0*(x + y)*pi*pi - 1.0)*numarray.sin(pi*x)*numarray.cos(pi*y))
# need only two k-functions, as dx is the same in both directions!
k_p = lambda x,y: x + y + 0.5*dx
k_m = lambda x,y: x + y - 0.5*dx
# boundary fu:
bf = lambda x,y: exp(-rt)*Numeric.sin(pi*x)*Numeric.cos(pi*y)
# initical condition:
initf = lambda x,y: numarray.sin(pi*x)*numarray.cos(pi*y)


# build the stencil
s = Stencil(nsd=2,varcoeff=True)
s.addNode((0,-1),[lambda *x: h*k_p(*x)])
s.addNode((-1,0),[lambda *x: h*k_p(*x)])
s.addNode((0,0),[lambda *x: 1.0 - 2.0*h*(k_m(*x) + k_p(*x))])
s.addNode((1,0),[lambda *x: h*k_m(*x)])
s.addNode((0,1),[lambda *x: h*k_m(*x)])


idstencil = Stencil(nsd=2,nodes={(0,0): 1.0})
A = StencilList(g)
A.addStencil(s,g.innerPoints())

B = StencilSet(g)
B.addStencil(idstencil,g.boundary())

S = StencilSet(g)
S.addStencil(idstencil,g.innerPoints())

# Go parallel if we are on parallel stuff
g.partition(A)
B.doInitParallel()
S.doInitParallel()

u = Field(g)
u.fill_vec(initf)
d = Numeric.array(u.data)

# Source and boundary fields
F = Field(g)
dF = Numeric.array(F.data)
bF = Field(g)
dbF = Numeric.array(bF.data)

# build data structures in StencilSets
A.buildMatrixOperator(u)
B.buildMatrixOperator(u)
S.buildMatrixOperator(u)

# reset to start-time again
rt = dt

if myrank <= 0:
    print "start..."
ct = -time.clock()
laps = -time.time()
while rt < T:
    # update source field
    F.fill_vec(dt_f)
    bF.fill_vec(bf)

    dF[:] = F.data
    dbF[:] = bF.data
```

```
        d[:] = A.direct_matvec(d) + S.direct_matvec(dF) + B.direct_matvec(dbF)

        if u.isParallel:
            u.data[:] = d
            u.updateField()
            d[:] = u.data[:]

        rt += dt

    ct += time.clock()
    laps += time.time()
    if myrank <= 0:
        print "stop..."
        print "Used cputime in comp.loop: ",ct
        print "Elapsed time in comp.loop: ",laps

    if not u.isParallel:
        u.data[:] = d
    return u,g,rt

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