UNIVERSITY OF OSLO
Department of Informatics

A Python Library for Solving Partial Differential Equations

Master thesis

Johannes Hofaker Ring

May 2, 2007

Preface

The aim with this thesis is to investigate how we can create unified interfaces to some key software components that are needed when solving partial differential equations. Two particular components are addressed here: sparse matrices and visualization. We want the interfaces to be simple to use, preferably with a Matlab-like syntax. We also want the interfaces to be "thin" in the sense that the interface code is small and provides access to core functionality only, not all nice-to-have options that one can think of. The interfaces are written in Python [29], a scripting language with a simple, clean and easy-to-use syntax, great software development flexibility, rapidly growing popularity, and rich libraries for both numerical and administrative tasks. The idea is to use Python to write the main algorithm for solving PDEs and thereby steer underlying numerical software.

Chapter 1 presents a matrix library for storage, factorization, and "solve" operations. The goal is to have a unified interface to many different types of matrix formats, mainly sparse matrix formats, where the various formats typically have the core numerics in widely different packages (BLAS, LAPACK, PySparse, for instance). PDE solvers written in Python can then work with one API for creating matrices and solving linear systems. This idea is not new and has been explored in many C++ libraries, e.g., Diffpack [3], DOLFIN [5] and GLAS [10]. The new contribution in this thesis is to have such an interface in Python and explore some of Python's flexibility. We also have a more heterogeneous collection of underlying matrix libraries than what the cited C++ packages aim at. Python is slow for number crunching so it is crucial to perform the factorization and solve operations in compiled Fortran, C or C++ libraries. The PyACTS project [26] has a goal quite similar to ours, but is aimed at the tools in the ACTS collection only (ScaLAPACK, SuperLU, Hyper, to mention some). A natural future extension of our matrix library will be to incorporate PyACTS. This will also provide support for parallel computing. The present thesis is limited to serial computing only.

At the end of Chapter 1 we explore the matrix library for solving diffusion equations. In particular, we compare the the efficiency of the traditional ADI methods with sparse matrix factorization-based solution techniques.

Chapter 2 deals with a unified interface, called Easyviz, to visualization packages, both for curve plotting and for 2D/3D visualization of scalar and vector fields. The interface calls up various plotting packages, some simple and easy-to-install like Gnuplot and some more comprehensive like VTK and Matplotlib. The Easyviz syntax almost coincides with that of Matlab, thus making it easy for students and researchers trained in Matlab to start plotting with a Python-based platform. Or in other words, one can use Matlab-like syntax for accessing a wide range of visualization tools. In this way, PDE solvers can visualize the solutions with a code independent of the underlying plotting package that actually produces the plots. Chapter 2 acts as a tutorial for Easyviz and has been written together with my supervisor Hans Petter Langtangen.

The focus of this thesis has been on developing reusable and stable software and to document this software such that it can readily be applied by students and scientists.

Acknowledgments. I would like to thank my supervisor Professor Hans Petter Langtangen for his most valuable help in writing this thesis. Without his help this thesis would never exist.

I would also like to thank my family who have supported and encouraged me over the years. A special thanks goes to my father Øivind Ring for his valuable help throughout the studies.

Table of Contents

Chapter 1

A Matrix Library

1.1 Introduction

When solving partial differential equations (PDEs) numerically one normally needs to solve a system of linear equations. Solving this linear system is often the computationally most demanding operation in a simulation program. Therefore we need to carefully select the algorithm to be used for solving linear systems. However, the choice of algorithm depends on the PDE problem being solved and the size of the problem (i.e., the number of grid points). The purpose of the present chapter is to create a library of various algorithms and data structures which give the programmer (or the user) of PDE applications great flexibility in choosing an appropriate solution method for linear systems, given the PDEs and the problem size.

Some of the most standard methods for solving PDEs is the Finite Difference, Finite Element and Finite Volume methods. These methods lead to large *sparse* linear systems, or more precisely, the coefficient matrix is large and sparse. Taking advantage of the sparsity structure of the coefficient matrix is of great importance for constructing fast solution methods. This complicates the algorithms and their implementations significantly, compared to straight Gaussian elimination on a dense matrix. The different methods need different implementations, but we would like to hide these differences and how the coefficient matrices are stored in memory. We would also like to hide many of the details of the algorithms and be able to work with the *primary conceptual mathematical quantities and operations*. That is, given a matrix and a right-hand side, we want to perform a "solve" operation to solve the associated linear system.

To enable the application code to be written at such a high level, we need to design a layered library where different layers have different levels of abstraction. The bottom layer operates directly on the numbers and must face, e.g., the storage structure of a matrix, while the top layer provides a convenient programming interface to a PDE application programmer.

The following text describes a matrix library in Python particularly suited for the needs met when solving PDEs. The matrix library provides many different matrix formats, such as dense, banded, diagonal, tridiagonal, sparse, and structured sparse. We will also look at some of the difficulties with respect to storage structure and operations on these matrix formats.

1.2 Key Design Issues

Arrays constitute a fundamental data structure in numerical applications. One applies arrays for representing diverse mathematical structures such as vectors, matrices, sets, grids, and fields. From the machine's point of view, an array is actually a consecutive set of bytes storing numbers

in a fixed sequence. In a scientific computing code we would like to use higher-level abstractions inspired by the mathematics instead of just manipulating the low-level arrays directly, since a close relationship between computer code and the mathematical formulation of the problem is desirable. We would also like to hide unnecessary details such as the length of the arrays and how abstractions are stored in memory. We will now study an object-oriented implementation of the matrix abstraction.

Linear Systems Solution by Classical Software. Let us say we want to solve the linear system $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A} \in \mathbb{R}^{n,n}$ is the coefficient matrix, $\mathbf{x} \in \mathbb{R}^n$ is the unknown vector, i.e. the numerical solution, and $\mathbf{b} \in \mathbb{R}^n$ is a given right-hand side vector. Such linear systems, can be solved by direct or iterative methods. Here we use direct methods, which normally means some type of Gaussian elimination. In Gaussian elimination, the solution procedure consists first of an LU factorization of the coefficient matrix and then solve using the factorized matrix. If, e.g., the coefficient matrix is a dense matrix, we could express this (conceptually) in Fortran 77 as

```
call fact_densem(A,n)
call solve_densem(A,n,b,x)
```

This is easy and straightforward, but we are required to supply details on the array lengths. We want to hide these details since they are not needed in the mathematical expression.

If the coefficient matrix, $\mathbf{A}$, is a banded matrix with k_ℓ sub-diagonals and k_u super-diagonals, we could use Banded Gaussian elimination which operates on the matrix elements inside the band only and saves considerably work. The Fortran 77 call would in this case be something like:

```
call fact_bandm(A,n,kl,ku)
call solve_bandm(A,n,kl,ku,b,x)
```

For a sparse matrix representation of $\mathbf{A}$, several arrays and integers constitute the matrix data structure, and all of these variables are explicitly visible in the calls to the solve operations.

We want the application programmer to code without bother whether the matrix is dense or sparse. Anyway it should be easy to switch from a dense matrix representation (for debugging) to a computationally efficient sparse matrix representation in the application program. Such a switch is non-trivial, and may easily introduce new bugs, when programming in F77 with long and complicated parameter lists. What we need is a "solve call" with the same syntax regardless of the matrix format the user of the program has chosen in this particular run. The mechanism for reaching this goal is to pack data structures in objects and hide the details of arrays, reals, and integers making up the data structure. In the application program we have some matrix `A` and perform the operations

```
A.factorize()
x = A.solve(b)
```

to carry out the solution process. In case `A` is a banded matrix, the object `A` holds all necessary information about how the data are stored and what type of underlying software (e.g., LAPACK) that is used in `A.factorize()` and `A.solve()`. Switching to a sparse matrix just makes the inner details of `A` more complicated, but the factorize and solve statements above look the same.

Object-Oriented Programming. During the 1990s it became apparent that object-oriented programming could provide the technical means to realize the coding example in the previous paragraph. Especially C++ has received much attention for its support for object-oriented programming combined with its speed for numerical computations. In the present work we adopt the Python language, which has full support for object-oriented programming, but which is often very slow for intensive number crunching. The CPU-intensive parts must therefore be migrated to compiled code, typically Fortran, C, or C++. The easiest way of doing this is to use the Numerical Python (NumPy) library and break up an algorithm into basic array operations each of which are implemented efficiently by NumPy functions in C. If the speed-up by vectorization is not sufficient, one can easily migrate slow Python loops or compound NumPy operations to hand-written, special-purpose functions in Fortran, C, or C++. The idea is that only a fraction of a large simulation program needs to be implemented in low-level compiled languages, i.e., most of the code can be written in a convenient high-level language like Python without sacrificing the overall computational efficiency.

The principal idea of the software design for matrices and associated solve operations is to implement the matrix as a class. The class may contain data and functions operating on the data. For a dense matrix class, the required data are the entries of the matrix and the size (number of rows and columns). The entries could be represented as an `array` object (in Numerical Python) which might be of real or complex kind, either with single or double precision. The number of rows and columns is represented as integer objects.

Each matrix format is represented by a class, and all matrix classes are collected in a class hierarchy. On top of the hierarchy we have a base class `MatrixBase`. This class offers a generic interface to all matrix formats. Subclasses of `MatrixBase` implement specific matrix formats, e.g., class `DenseMatrix` for dense matrices, class `BandMatrix` for banded matrices, class `TriDiagMatrix` for tridiagonal matrices, and so forth. Each class holds suitable array structures for storing the matrix data, plus methods for common matrix operations related to solving linear systems. In particular we may make use of Python's special methods `__getitem__` and `__setitem__` to enable subscripting the matrix. If $\mathbf{A}$ is an $m \times n$ matrix with entries $a_{i,j}$ for $i = 0, 1, \ldots, m-1$ and $j = 0, 1, \ldots, n-1$, we can access entry (i, j) in the matrix with the (familiar) syntax `A[i,j]`. Similarly we can assign a scalar to entry (i, j) in matrix $\mathbf{A}$ by `A[i,j]=1.57`. We note that assignment to `A[i,j]` might not always be possible. This is because in a sparse matrix only some of the index pairs (i, j) exist. The important point to be made here, however, is that the programmer is in charge of defining what is meant by `A[i,j]`, not a language constructor.

One of the most fundamental operations when solving PDEs is matrix factorization, matrix solve (using the factorization), and matrix-vector products. Every subclass of `MatrixBase` needs their own implementation of these operations. For example, computing a matrix-vector product using a dense matrix, is implemented in a straightforward loop. A sparse matrix has a much more efficient matrix-vector product function which utilizes the sparsity structure of the matrix. These operations are the time consuming parts when solving PDEs and must be implemented as efficiently as possible.

New Python Classes and Old Fortran Software. As mentioned, computationally intensive parts of a Python code must often be carried out in compiled code. Fortunately, the standardized LAPACK (Linear Algebra Package) and BLAS (Basic Linear Algebra Subprograms) libraries [18, 2], written in Fortran 77, contains very efficient compiled code for many of the most common linear algebra operations. These libraries provide, for example, a family of factorization and solve routines for dense, banded, and tridiagonal matrices, with or with-

out symmetry, with real or complex entries, in single or double precision format. Calling the routines from Python is quite straightforward, but SciPy [31], a major package for scientific computing with Python, already provides a unified framework for calling up LAPACK and BLAS. Unfortunately, not all the LAPACK routines we need in a PDE context are yet integrated into the SciPy framework so we need to extend SciPy in this respect. Our Python class will then be able to get their functionality through SciPy rather than partly through SciPy and partly through some home-made wrapping of parts of LAPACK.

Now, let us take a closer look at the `DenseMatrix` and `TriDiagMatrix` classes. As mentioned above, one of the most fundamental functions needed for solving linear systems using Gaussian elimination is to factorize a matrix and then solve using the factorization. There is also a need for a matrix-vector product, which is used in iterative methods. For a dense matrix, the proper routines in LAPACK for factorizing and solving is `xGETRF` and `xGETRS` respectively. For multiplying a dense matrix with a vector, the proper BLAS routine is `xGEMV`. For a tridiagonal matrix the LAPACK routines for factorizing and solving are `xGTTRF`, `xGTTRS` and for matrix-vector product `xLAGTM`. Note that the prefix `x` in the notation `xNAME` specifies the data type of that particular LAPACK or BLAS routine. For instance, `DGETRF` works for arrays with double precision floating point numbers, while `CGETRF` works for arrays with single precision complex numbers. We refer to the LAPACK Users' Guide [1] for a complete explanation of the naming conventions in LAPACK.

As an example, we take a closer look at the factorize and solve methods in the `DenseMatrix` class. First we look at the factorize method:

```
def factorize(self):
    if self._check_if_already_factorized():
        return

    fact, = get_lapack_funcs(('getrf',), (self.m,))

    self.m[:,:], self.ipiv[:], self.info = fact(self.m)
    if self.info < 0:
        raise ValueError, \
            "illegal value in %d-th argument of LAPACK's xGETRF" \
            % (-self.info)
    if self.info > 0:
        warn("diagonal number %d is exactly zero. Singular matrix." \
             % self.info, RuntimeWarning)
    self.factorized = True
```

To save CPU-time, we first check if the matrix is already factorized. If it is, we do not need to factorize it again, so we can just return from the method. If we do want to factorize the matrix again, we can simply set the class variable `factorize` to `False`. Next, we fetch the proper interfaced LAPACK routine with a call to `get_lapack_funcs` from the module `scipy.linalg.lapack` and then compute the factorization. Before factorizing, the matrix is stored in the class variable `m`, which is a 2D NumPy array. After the call to `fact`, `m` is updated with the factorized matrix. From the calls to the different LAPACK routines, we also receives some info on whether the operation was successful or not. If the operation was unsuccessful, the user is notified by either a warning or an exception. At the end, we set the global class variable `factorized` to `True`. This way we can later find out if the matrix is factorized or not. Finally, we should note that the array `ipiv` is used as an help array for pivoting in the LAPACK routines `xGETRF` and `xGETRS`.

Now that the matrix is factorized, we are ready to solve the linear system. To this end, we use the `solve` method in class `DenseMatrix`, which takes the following form:

```
def solve(self, b, solution=None, transpose=0):
    self._check_if_not_factorized()
```

```
    _check_size(b, 'rhs', self.n, self.skip, self.raise_exception)

    if solution is None:
        solution = zeros(len(b), self.elm_tp)
    else:
        _check_size(solution, 'solution', self.n,
                    self.skip, self.raise_exception)

    solve, = get_lapack_funcs(('getrs',), (self.m,))

    solution[:], self.info = solve(self.m, self.ipiv, b,
                                   trans=transpose)
    if self.info < 0:
        raise ValueError, \
            "illegal value in %d-th argument of LAPACK's xGETRS" \
            % (-self.info)
    return solution
```

First, we check if the matrix is factorized. If it is not, we raise an exception. Otherwise, we check the size of the right-hand side vector and allocate storage for the solution if not already given. Then we solve the system with the proper LAPACK routine and return the solution if the operation was successful.

Let us look at an interactive Python session for demonstrating the usage of the `DenseMatrix` class:

```
>>> A = DenseMatrix(5, 5)      # create a 5x5 dense matrix
>>> # insert some data:
>>> for i in range(A.nrows):
...     for j in range(A.ncolumns):
...         A[i,j] = ...       # assign value to entry
>>> A.factorize()              # factorize matrix
>>> x = A.solve(b)             # solve linear system
```

Here we have assumed that the right-hand side vector `b` is a given NumPy array with length 5.

The default data type in a matrix instance is `float` (double precision real numbers); however, switching to another data type is straightforward:

```
>>> A = DenseMatrix(5, 5, element_type=complex)
```

This will initialize a 5×5 `DenseMatrix` instance with double precision complex entries.

For the `TriDiagMatrix` class, there is not much changes needed compared with the `factorize` and `solve` methods in `DenseMatrix`, however, the storage structure is somewhat different. In this class we only store the main diagonal and the sub- and super-diagonals. They are stored in three different 1D NumPy array-objects; here called `d`, `dl`, and `du`, respectively. The `factorize` and `solve` methods for class `TriDiagMatrix` are listed next.

```
def factorize(self):
    if self._check_if_already_factorized():
        return

    fact, = get_lapack_funcs(('gttrf',), (self.d,))

    self.dl[:], self.d[:], self.du[:], \
                self.du2[:], self.ipiv[:], self.info = \
                fact(self.dl, self.d, self.du)
    if self.info < 0:
        raise ValueError, \
              "illegal value in %d-th argument of LAPACK's xGTTRF" \
              % (-self.info)
    if self.info > 0:
        warn("diagonal number %d is exactly zero. Singular matrix." \
```

```
            % self.info, RuntimeWarning)
        self.factorized = True

    def solve(self, b, solution=None, transpose=0):
        self._check_if_not_factorized()
        _check_size(b, 'rhs', self.n, self.skip, self.raise_exception)

        if solution is None:  # allocate ?
            solution = zeros(self.n, self.elm_tp)
        else:
            _check_size(solution, 'solution', self.n,
                        self.skip, self.raise_exception)

        solve, = get_lapack_funcs(('gttrs',), (self.d,))

        solution[:], self.info = \
                  solve(self.dl, self.d, self.du, self.du2, self.ipiv, b,
                        trans=transpose)
        if self.info < 0:
            raise ValueError, \
                  "illegal value in %d-th argument of LAPACK's xGTTRS" \
                  % (-self.info)
        return solution
```

As we can see, the only difference from class `DenseMatrix`, is the parameters to the function `get_lapack_funcs` and the usage of the returned functions.

The solution of the linear system $\mathbf{Ax} = \mathbf{b}$ is $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, where $\mathbf{A}^{-1}$ is the inverse of the matrix $\mathbf{A}$. So instead of using the notation

```
A.factorize()
x = A.solve(b)
```

for solving the linear system, it might be more user-friendly if it was more similar to the mathematical notation, like

```
x = A**(-1)*b
```

This can easily be achieved by using operator overloading in Python. To this end, we need to implement the special methods `__pow__` (operator `**`) and `__mul__` (operator `*`) in the base class `MatrixBase`. The `__pow__` method takes the following form:

```
def __pow__(self, other):
    if other == -1:
        if not self.factorized:
            self.factorize()
        self.inverse = True
        return self
    raise NotImplementedError
```

If the `other` argument is -1, we factorize the matrix (if needed) and set a flag `inverse` to `True` before returning the matrix object itself. This invokes a product of the matrix object and the right-hand side vector (NumPy array), which again requires the `__mul__` method to be implemented:

```
def __mul__(self, other):
    if self.inverse:
        self.inverse = False
        return self.solve(other)
    return self.prod(other)
```

If the flag `inverse` is `True`, we set it back to `False` and return the solution of the linear system with the `solve` method in the matrix class. Otherwise, we return the standard matrix-vector product by calling the `prod` method. Note that the expression `x = A**(-1)*b` is equivalent to

```
x = MatrixBase.__mul__(MatrixBase.__pow__(A, -1), b)
```

1.3 Matrix Formats

From the discretization of partial differential equations we get different sparsity structures for the coefficient matrix. Some of the most common matrix formats are dense matrices, banded matrices, tridiagonal matrices, general sparse matrices, and structured sparse matrices. The efficiency of numerical algorithms depends strongly on the matrix storage scheme. The goal is to offer all these matrix formats and hide the details of the storage schemes. To this end, we have constructed a matrix class hierarchy as described in the previous section.

In this section we will give a description of the different matrix formats that are implemented in this library. Let $\mathbf{A}$ be an $m \times n$ matrix with entries $a_{i,j}$ for $i = 0, 1, \ldots, m-1$ and $j = 0, 1, \ldots, n-1$ and we will look at the different matrix formats like dense, band, tridiagonal, etc. Most of the matrix formats uses a class variable `m` (a NumPy array) in the subclass to store the matrix unless explicitly noted.

Class `DenseMatrix`: The most general matrix format is the dense matrix format. It is implemented in a subclass of `MatrixBase` called `DenseMatrix`. In the case of a dense matrix, all $a_{i,j}$ entries might be nonzero, so all entries of the matrix must be stored. Here is a typical dense matrix with dimension 5×5:

$$\mathbf{A} = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix}$$

The indexing of a dense matrix behaves as normal. That is, the entries are addressed as `A[i,j]` where $i = 0, \ldots, m-1$ and $j = 0, \ldots, n-1$. As already mentioned, to allow subscripting of a matrix object, we need to implement the special methods `__getitem__` and `__setitem__` in the matrix class. For the `DenseMatrix` class, these methods can be straightforwardly implemented as follows:

```
def __getitem__(self, (i, j)):
    if i >= 0 and i < self.n and j >= 0 and j < self.ncolumns:
        return self.m[i, j]
    else:
        raise IndexError, \
              '(%d,%d) outside matrix dimensions [%d,%d]' % \
              (i,j,self.n,self.ncolumns)

def __setitem__(self, (i, j), value):
    if i >= 0 and i < self.n and j >= 0 and j < self.ncolumns:
        self.m[i, j] = value
    else:
        raise IndexError, \
              '(%d,%d) outside matrix dimensions [%d,%d]' % \
              (i,j,self.n,self.ncolumns)
```

Class `BandMatrix`: In banded $m \times n$ matrices we have in addition to the main diagonal, k_ℓ sub-diagonals and k_u super-diagonals. The total bandwidth is then $k_\ell + k_u + 1$. An example of a band matrix with dimensions 5×5 and with $k_\ell = 2$ and $k_u = 1$ is

$$\mathbf{A} = \begin{pmatrix} a_{0,0} & a_{0,1} & 0 & 0 & 0 \\ a_{1,0} & a_{1,1} & a_{1,2} & 0 & 0 \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & 0 \\ 0 & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ 0 & 0 & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix}.$$

The LAPACK routines `xGBTRF` (factorize) and `xGBTRS` (solve) for banded matrices requires a matrix with dimensions of at least $(2k_\ell + k_u + 1) \times n$. The super-diagonals are stored in rows k_ℓ to $k_\ell + k_u - 1$, the main diagonal is stored in row $k_\ell + k_u$, and the sub-diagonals are stored in rows $k_\ell + k_u + 1$ to $2k_\ell + k_u$. Then, the band storage scheme in LAPACK for the matrix above becomes the following 6×5 matrix:

$$\texttt{A.m} = \begin{pmatrix} * & * & * & + & + \\ * & * & + & + & + \\ * & a_{0,1} & a_{1,2} & a_{2,3} & a_{3,4} \\ a_{0,0} & a_{1,1} & a_{2,2} & a_{3,3} & a_{4,4} \\ a_{1,0} & a_{2,1} & a_{3,2} & a_{4,3} & * \\ a_{2,0} & a_{3,1} & a_{4,2} & * & * \end{pmatrix}$$

The array elements marked with a $*$ are not used by LAPACK and the elements marked with a $+$ do not need to be set, but are required by the `xGBTRF` and `xGBTRS` routines.

For indexing the matrix, we need to map the logical index to the physical index where the entry is stored. In the case of banded matrices, the mapping will be

$$a_{i,j} = \texttt{A.m}[k_\ell + k_u + i - j, j], \quad \text{for } \max(0, j - k_u) \le i \le \min(m, j + k_\ell)$$

This mapping can be inserted into the `__getitem__` and `__setitem__` methods to give us a nice syntax for accessing the entries of the matrix. The `__setitem__` method takes the following form:

```
def __setitem__(self, (i, j), value):
    if i >= max(0,j-self.ku) and i <= min(self.n,j+self.kl) \
           and j >= 0 and j < self.ncolumns:
        self.m[self.kl+self.ku+i-j, j] = value
    else:
        raise IndexError, '(%d,%d) not inside band [%d,%d]' % \
              (i,j,max(i-self.kl+self.ku,0),\
               min(i+self.kl+self.ku,self.n))
```

The `__getitem__` method is similar and therefore not listed here. Trying to assign a value to an entry outside the band, e.g.

```
>>> A[1,3] = 1.57
```

will result in an exception, while it is possible to access the entry:

```
>>> print A[1,3]
0.0
>>>
```

This natural behavior when indexing the matrix outside its sparsity pattern is common to all matrix classes.

Class `SymmBandMatrix`: This class provides support for the symmetric case of the `BandMatrix` class. By utilizing the symmetry of a banded matrix, we only need roughly half the storage space compared with `BandMatrix`. For example, an 5×5 symmetric band matrix with $k = 2$ (total bandwidth is $2k + 1$), will look like

$$\mathbf{A} = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & 0 & 0 \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & 0 \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ 0 & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ 0 & 0 & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix}.$$

Here, $a_{i,i+1} = a_{i+1,i}$ for $i = 0, 1, 2, 3$ and $a_{i,i+2} = a_{i+2,i}$ for $i = 0, 1, 2$.

In LAPACK we only need to store the main diagonal and the k super-diagonals (or sub-diagonals). The matrix storage scheme therefore becomes a $(k + 1) \times n$ matrix where the super-diagonals are stored in rows 0 to $k - 1$ and the main diagonal is stored in row k. The storage structure of the matrix in the example above is listed next.

$$\texttt{A.m} = \begin{pmatrix} * & * & a_{0,2} & a_{1,3} & a_{2,4} \\ * & a_{0,1} & a_{1,2} & a_{2,3} & a_{3,4} \\ a_{0,0} & a_{1,1} & a_{2,2} & a_{3,3} & a_{4,4} \end{pmatrix}$$

For indexing the symmetric band matrix, we use the following mapping in the `__getitem__` and `__setitem__` methods:

$$a_{i,j} = \texttt{A.m}[k + i - j, j] \quad \text{for } \max(0, j - k) \leq i \leq j.$$

If we try to access an entry on one of the sub-diagonals of the matrix, we only map this to the corresponding super-diagonal entry.

Class `TriDiagMatrix`: A tridiagonal matrix is a $n \times n$ matrix with nonzero elements only on the diagonal and entries adjacent to the diagonal, i.e., along the sub- and super-diagonal. Here is an example of a 5×5 tridiagonal matrix:

$$\mathbf{A} = \begin{pmatrix} a_{0,0} & a_{0,1} & 0 & 0 & 0 \\ a_{1,0} & a_{1,1} & a_{1,2} & 0 & 0 \\ 0 & a_{2,1} & a_{2,2} & a_{2,3} & 0 \\ 0 & 0 & a_{3,2} & a_{3,3} & a_{3,4} \\ 0 & 0 & 0 & a_{4,3} & a_{4,4} \end{pmatrix}$$

Storing the matrix in terms of the LAPACK storage scheme, we use three 1D NumPy arrays, one of length n for the main diagonal and two of length $n - 1$ for the sub- and super-diagonals. For the matrix above, we have

$$\begin{aligned} \texttt{A.dl} &= \begin{pmatrix} a_{1,0} & a_{2,1} & a_{3,2} & a_{4,3} \end{pmatrix}, \\ \texttt{A.d} &= \begin{pmatrix} a_{0,0} & a_{1,1} & a_{2,2} & a_{3,3} & a_{4,4} \end{pmatrix}, \text{ and} \\ \texttt{A.du} &= \begin{pmatrix} a_{0,1} & a_{1,2} & a_{2,3} & a_{3,4} \end{pmatrix}. \end{aligned}$$

In addition to these three arrays, we need a fourth array of length $n-2$, `A.du2`. The entries in this array do not need to be set, but the array is required by the LAPACK routines `xGTTRF` and `xGTTRS`.

Indexing the tridiagonal matrix is easy. Trying to access entry (i,j) in the matrix, we only need to check for three different possibilities: $j = i-1$, $j = i$, and $j = i+1$ (sub-, main- and super-diagonals, respectively). For example, if we try to assign a value to the entry `A[i,i-1]` for a given i, the value is assigned to `A.dl[i-1]`. Assigning a value to entry `A[i,i]`, the value is assigned to `A.d[i]`. And last, assigning a value to entry `A[i,i+1]`, the value is assigned to `A.du[i]`. This is implemented in the `__setitem__` method in the `TriDiagMatrix` class and with a similar definition for the `__getitem__` method.

Class `SymmTriDiagMatrix`: A symmetric tridiagonal matrix is a special case of tridiagonal matrices. Here, entry $(i, i-1)$ is equal to entry $(i, i+1)$ for $i = 1, \ldots, n-2$. A 5×5 symmetric tridiagonal matrix will look like

$$\mathbf{A} = \begin{pmatrix} a_{0,0} & a_{0,1} & 0 & 0 & 0 \\ a_{1,0} & a_{1,1} & a_{1,2} & 0 & 0 \\ 0 & a_{2,1} & a_{2,2} & a_{2,3} & 0 \\ 0 & 0 & a_{3,2} & a_{3,3} & a_{3,4} \\ 0 & 0 & 0 & a_{4,3} & a_{4,4} \end{pmatrix},$$

where $a_{i,i+1} = a_{i+1,i}$ for $i = 0, 1, 2, 3$.

In the case of a symmetric tridiagonal matrix, LAPACK requires us to store only the main diagonal and the sub-diagonal. Thus, we end up with two arrays, both of length n (the sub-diagonal array actually needs only to be of length $n-1$, but the LAPACK eigenvalue routine (`xSTEV`), requires a vector of length n also for the sub-diagonal)[1]. For the matrix above, this results in the following storage scheme

$$\begin{aligned} \texttt{A.dl} &= \begin{pmatrix} a_{1,0} & a_{2,1} & a_{3,2} & a_{4,3} & + \end{pmatrix} \\ \texttt{A.d} &= \begin{pmatrix} a_{0,0} & a_{1,1} & a_{2,2} & a_{3,3} & a_{4,4} \end{pmatrix} \end{aligned}$$

The indexing of symmetric tridiagonal matrices follows basically the same scheme as for general tridiagonal matrices. The only difference is that when we try to access or assign a value in an entry from the super-diagonal, we simply map it to the corresponding entry in the sub-diagonal.

Class `StructSparseMatrix`: A structured sparse matrix is (as the name indicates) a matrix that follows some special regular sparsity pattern. For instance, a structured sparse 5×5 matrix might look like the following matrix:

$$\mathbf{A} = \begin{pmatrix} a_{0,0} & 0 & 0 & a_{0,3} & 0 \\ 0 & a_{1,1} & a_{1,2} & 0 & a_{1,4} \\ 0 & a_{2,1} & a_{2,2} & 0 & 0 \\ a_{3,0} & 0 & 0 & a_{3,3} & a_{3,4} \\ 0 & a_{4,1} & 0 & a_{4,3} & a_{4,4} \end{pmatrix}$$

When storing a structured sparse matrix, we only need to store the diagonals that has nonzero entries. These diagonals are stored as columns in a rectangular array structure

[1] Support for eigenvalues and eigenvectors are also implemented in the matrix library for some of the matrix formats. However, these are neither explained nor used in the present thesis.

with $n \times ndiags$ entries. There are five nonzero diagonals in the matrix above and the storage structure then becomes:

$$\texttt{A.m} = \begin{pmatrix} 0 & 0 & a_{0,0} & 0 & a_{0,3} \\ 0 & 0 & a_{1,1} & a_{1,2} & a_{1,4} \\ 0 & a_{2,1} & a_{2,2} & 0 & 0 \\ a_{3,0} & 0 & a_{3,3} & a_{3,4} & 0 \\ a_{4,1} & a_{4,3} & a_{4,4} & 0 & 0 \end{pmatrix}$$

As we can see, the row index matches the row index of the logical matrix index. To be able to carry out operations on this data structure, we need an index vector `offset` (with length *ndiags*) that holds information on how each stored diagonal is placed relative to the main diagonal. In the current example, the `offset` vector would be:

$$\texttt{A.offset} = \begin{pmatrix} -3 & -1 & 0 & 1 & 3 \end{pmatrix}$$

Indexing this particular storage structure requires some special treatment. Let us look at the `__setitem__` method (`__getitem__` is similar):

```
def __setitem__(self, (i,j), value):
    idx = self._offset2index(j-i)
    if idx >= 0:
        self.m[i,idx] = value
    else:
        raise IndexError, "(%d,%d) outside diags" % (i,j)
```

Here we use the method `_offset2index` in the `StructSparseMatrix` class to locate the column number of the internal storage array. The `_offset2index` method takes the following form:

```
def _offset2index(self, d):
    for k in range(self.ndiags):
        if self.offset[k] == d:
            return k
    return -1
```

Given a offset value `d`, this method returns the corresponding column number of the internal storage array. If the requested diagonal is stored, the returned value is a number in the range $0, \ldots, ndiags - 1$, otherwise it returns -1.

Structured sparse matrices arises frequently when PDEs are solved by finite difference methods on regular grids.

Class `SparseMatrix`: In a general sparse matrix, there may be only a few nonzero entries in each row, but there is no regular structure with respect to where in a row the nonzeros appear. One possible structure of a 5×5 general sparse matrix might be

$$\mathbf{A} = \begin{pmatrix} a_{0,0} & 0 & 0 & a_{0,3} & 0 \\ 0 & a_{1,1} & a_{1,2} & 0 & a_{1,4} \\ 0 & a_{2,1} & a_{2,2} & 0 & 0 \\ a_{3,0} & 0 & 0 & a_{3,3} & a_{3,4} \\ 0 & a_{4,1} & 0 & a_{4,3} & a_{4,4} \end{pmatrix}.$$

This type of general sparse matrices arises when PDEs are solved by finite element methods, especially when the grid is irregular.

The implementation of the `SparseMatrix` class is based on a Python package called PySparse [8]. This package provides a set of matrix types holding real double precision numbers. Unfortunately, there is no support for real single precision or complex numbers.

In PySparse, there is a module called `spmatrix` containing three types named `ll_mat`, `csr_mat`, and `sss_mat`. These types represent sparse matrices in the LL, CSR and SSS formats, respectively. The common way to use the `spmatrix` module is to first build a matrix in the LL format, manipulate it until it has its final shape and content, and then convert it to either CSR or SSS format. The two latter formats are faster and requires less memory. For a thoroughly explanation of the three formats, see Appendix A in [9].

The indexing of a general sparse $m \times n$ matrix is implemented in PySparse. The entries are addressed as `A[i,j]` where $i = 0, 1, \ldots, m-1$ and $j = 0, 1, \ldots, n-1$. This is the same behavior as for the dense matrix format and the `__getitem__` and `__setitem__` methods in the `SparseMatrix` class is therefore the same as the ones on page 7.

1.4 How to Interface LAPACK and BLAS Routines via SciPy Tools

In SciPy there are already some LAPACK routines interfaced for use in Python. Some of them includes LU factorization (`xGETRF`) and solve using the factorization (`xGETRS`) for dense matrices. For other matrix formats, there were no support in SciPy for these routines when the software for this thesis was developed[2]. To extend SciPy with more LAPACK routines, we need to edit the file called `generic_flapack.pyf` in the subdirectory `Lib/linalg` of the SciPy source code. This file is a F2PY signature file, which is based mostly on Fortran 90 syntax. In addition there is some F2PY specific commands. The structure of the `generic_flapack.pyf` file is listed next.

```
python module generic_flapack
  interface
    subroutine name_of_subroutine(param1, param2, ...)
      ... body (initializations and call to LAPACK routine)
    end name_of_subroutine
    ... more subroutines
  end interface
end python module generic_flapack
```

So, we have to create one subroutine for each routine in LAPACK we wish to interface.

Let us take a closer look at one specific example. In this example, we create an interface to the LAPACK routine for factorizing a dense matrix. This is, as mentioned above, already interfaced in SciPy, but it will be useful for illustration purposes. The LAPACK routine for factorizing a dense matrix is called `xGETRF`. In Fortran, a call to the LAPACK routine `DGETRF` (double precision real numbers) would be

```
call DGETRF(m, n, a, lda, ipiv, info)
```

Here, `a` (the matrix we want to factorize), is an array with dimension `(lda,n)`. The integers `m` and `n`, are the number of rows and columns, respectively, and `lda=max(1,m)` is the leading dimension of the array `a`. On output, the factorized matrix is stored in the array `a`. `ipiv` is an output parameter, which is filled with the pivot indices used during factorization. `info` is another output parameter, which is equal to zero if the factorization was successful.

[2]Recently, SciPy has been extended with several more LAPACK routines, including `xGBTRF` (factorize band matrix) and `xGBTRS` (solve band matrix).

In Python we will like to hide the parameters for the dimensions of the matrix. These data is already included in the data structure of a NumPy array, so it can easily be extracted. We also want the `ipiv` and `info` parameters to be return values only and not input parameters to the function. The `a` array should be both an input parameter and a return value. Now, let us see how we interface this LAPACK routine in `generic_flapack.pyf`:

```
subroutine <tchar=s,d,c,z>getrf(m,n,a,piv,info)

! lu,piv,info = getrf(a,overwrite_a=0)
! Compute an LU factorization of a  general  M-by-N  matrix  A.
! A = P * L * U
  threadsafe
  callstatement {
    int i;
    (*f2py_func)(&m,&n,a,&m,piv,&info);
    for(i=0,n=MIN(m,n);i<n;--piv[i++]);
  }
  callprotoargument int*,int*,<type_in_c>*,int*,int*,int*

  integer depend(a),intent(hide):: m = shape(a,0)
  integer depend(a),intent(hide):: n = shape(a,1)
  <type_in> dimension(m,n),intent(in,out,copy,out=lu) :: a
  integer dimension(MIN(m,n)),depend(m,n),intent(out) :: piv
  integer intent(out):: info

end subroutine <tchar=s,d,c,z>getrf
```

Here, the first line specifies the name of the function as we will see it from Python. The `<tchar=s,d,c,z>` tag means that this function needs to be built for the specified four types (`s`=float, `d`=double, `c`=complex and `z`=double complex). We will then end up with four versions of the same function (except that the type differs), namely `sgetrf`, `dgetrf`, `cgetrf`, and `zgetrf`. The next three lines are comments explaining how we want to call the function in Python and what it does. The `threadsafe` statement is used to indicate that the wrapped function is thread-safe, i.e., it will insert `Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS` around the function call. We refer to Chapter 8.1 in the Python/C API Reference Manual [34] for more information on this topic. In the `callstatement` statement we do the actual call to the LAPACK routine. The wrapped function is available as `(*f2py_func)` and the proper arguments are entered. We also decrease the `piv` array since in Fortran arrays has base index 1, while in Python it is 0. We should note that the `callstatement` block is supposed to be on a single line and the line is only broken because of page width limitations. In the `callprotoargument` statement, we specify the prototype of the arguments to the LAPACK routine. The rest of the statements are declarations of the variables used as parameters to the LAPACK routine. As an example, the two lines

```
integer depend(a),intent(hide):: m = shape(a,0)
integer depend(a),intent(hide):: n = shape(a,1)
```

means that we are declaring integers `m` and `n`, which depends on the shape of the input array `a`. Here, `shape(a,0)` gives the number of rows in the matrix, while `shape(a,1)` gives the number of columns. We also use the attribute `intent(hide)`, such that the argument is removed from the required or optional arguments to the Python function. As a result, we do not have to supply the number of rows and columns when calling the function in Python. Another example:

```
<type_in> dimension(m,n),intent(in,out,copy,out=lu) :: a
```

Here we declare an $m \times n$ array `a` of type `<type_in>`, i.e., the same type as the input array. The attribute `intent(in,out,copy,out=lu)` might need some explanation. The `in` keyword, specifies

that the array `a` should be given as an input argument to the Python function, while the `out` keyword specifies that it also should be returned from the function. The `copy` keyword ensures that the original contents of the input array is not altered. When the `copy` keyword is specified, F2PY creates an optional argument `overwrite_a` with default value 0. If this argument is set to 1, the array entries will be overwritten. The F2PY application actually creates doc strings for the functions automatically and the `out=lu` keyword replaces the default return name with `lu` in the functions `__doc__` string.

Now, let us see how we can use this function in Python. First we need to import it into Python. The function lies in a module named `flapack` in the `scipy.linalg` module. From another module in `scipy.linalg`, called only `lapack`, there is a function `get_lapack_funcs`, which can be used to extract the interfaced LAPACK functions:

```
>>> from scipy.linalg.lapack import get_lapack_funcs
>>> # let m be a NumPy array with dtype=float
>>> factorize, = get_lapack_funcs(('getrf',),(m,))
```

The first argument to `get_lapack_funcs` (given as a sequence of strings), are the names (without the type prefix) of the LAPACK routines we want to use. The second argument is used to determine the type of the function, i.e., if `m` is of type `float`, the returned function is `dgetrf`, if `m` is of type `complex64` (complex number composed of to single precision floats), the returned function is `cgetrf`, and so on. To see the usage of the `factorize` function, we can simply print out the functions doc string:

```
>>> print factorize.__doc__
dgetrf - Function signature:
  lu,piv,info = dgetrf(a,[overwrite_a])
Required arguments:
  a : input rank-2 array('d') with bounds (m,n)
Optional arguments:
  overwrite_a := 0 input int
Return objects:
  lu : rank-2 array('d') with bounds (m,n) and a storage
  piv : rank-1 array('i') with bounds (MIN(m,n))
  info : int

>>>
```

As noted earlier, the doc string is automatically generated by the F2PY tool. The matrix `m` can now be factorized by the following command:

```
>>> lu, ipiv, info = factorize(m)
```

Now, `lu` contains the LU factorization of the matrix `m` and `piv` contains the pivot indices used during factorization. If `info` is equal to zero, the matrix was successfully factorized.

Interfacing the solve routine `xGETRS` for dense matrices (and LAPACK routines for other matrix formats) can be done in a similar manner as described above. The BLAS routines for computing a matrix-vector product are also interfaced similarly; however, these interfaces are placed in the file `generic_fblas2.pyf` in the subdirectory `Lib/linalg` of the Scipy source code.

1.5 Building a Python Module for Structured Sparse Matrices

In LAPACK there is unfortunately no support for structured sparse matrices and there is currently no Python modules that have support for this particular matrix format. We must

therefore create a Python module from scratch for structured sparse matrices. The CPU-intensive parts should be taken care of by a compiled language like Fortran, C, or C++. Here we will use Fortran 77 as the number crunching language and use F2PY for the connection with Python. The Python module should have functionality both for solving a linear system and for computing a matrix-vector product. The solve functionality will be separated into two steps: Relaxed Incomplete LU (RILU) factorizing of the coefficient matrix and then solve using forward and backward substitution in connection with the RILU factorization. The Fortran code must then contain the following three subroutines:

- `xfactRILU` - Performs the RILU factorization of a structured sparse matrix.
- `xforwBackRILU` - Solves the linear system $\mathbf{Ax} = \mathbf{b}$ by forward and backward substitution given that the matrix is factorized by `xfactRILU`.
- `xprod` - Performs the matrix-vector product $\mathbf{y} = \mathbf{Ax}$.

The prefix `x` is to be replaced by the type of the subroutine and indicates that the subroutine should be implemented for both single and double precision floating point numbers and single and double precision complex numbers. For consistency with LAPACK, we will use the same syntax for the type prefix. That is, `s` for single precision floating point numbers, `d` for double precision floating point numbers, `c` for single precision complex numbers, and `z` for double precision complex numbers. We will in the following look at the three subroutines mentioned above, but restricts us to the ones with double precision floating point numbers. We start by looking at the `dfactRILU` subroutine:

```
      subroutine dfactRILU(omega, A, n, ndiags, offset)
      integer            n, ndiags
      double precision   omega
      double precision   A(0:n-1, 0:ndiags-1)
      integer            offset(0:ndiags-1)

      integer            di, dj, i, j, k, r, s, maindiag
      double precision   mm
      integer            offset2index
      external           offset2index
      intrinsic          abs

      maindiag = offset2index(ndiags, offset, 0)

      do 50 r=0, n-2
         do 40 di=maindiag-1, 0, -1
            if (r.le.(n-1)-abs(offset(di))) then
               i = abs(offset(di)) + r
               A(i,di) = A(i,di) / A(r,maindiag)
               mm = A(i,di)
               do 30 dj=maindiag+1, ndiags-1
                  if (A(r,dj).ne.0) then
                     k = 1
                     if (r.le.(n-1)-offset(dj)) then
                        j = offset(dj) + r
                        s = 0
                        do 20 k=k, ndiags-1
                           if (offset(k).eq.j-i) then
                              s = k
                              goto 10
                           end if
20                      continue
10                      if (s.ne.0) then
                           A(i,s) = A(i,s) - mm * A(r,dj)
                        else
                           A(i,maindiag) = A(i,maindiag) -
```

```
     &                                        omega * mm * A(r,dj)
                        end if
                     end if
                  end if
 30            continue
            end if
 40       continue
 50    continue
       return
```

The `dfactRILU` subroutine takes five arguments as input, where `omega` is the relaxation parameter (floating point number between 0 and 1), `A` is an $n \times ndiags$ matrix, and `offset` is an index vector with length $ndiags$, holding information on how each stored diagonal is placed relatively to the main diagonal. We should note that the algorithms used in the Fortran code presented here are taken from the file `MatStructSparse_Type.cpp` in the Diffpack library, which again is based on a C code by Are Magnus Bruaset. The algorithm is rather complex, however, we will not go into detail about the code here.

After a call to `dfactRILU`, the array `A` contains the factorized matrix and we are ready to solve the linear system. To this end, we use the subroutine `dforwBackRILU`, which takes the following form:

```
      subroutine dforwBackRILU(A, n, ndiags, offset, b, x)
      integer            n, ndiags
      double precision   A(0:n-1, 0:ndiags-1), b(0:n-1), x(0:n-1)
      integer            offset(0:ndiags-1)

      integer            i, k, l, maindiag
      double precision   sum
      integer            offset2index
      external           offset2index

      maindiag = offset2index(ndiags, offset, 0)
*
*     forward :
*
      do 20 i=0, n-1
         sum = b(i)
         do 10 k=0, maindiag-1
*
*     offset(k) always negativ for k < maindiag
*
            l = i + offset(k)
            if (l.ge.0) then
               sum = sum - A(i,k) * x(l)
            end if
 10      continue
         x(i) = sum
 20   continue
*
*     backward:
*
      do 40 i=n-1, 0, -1
         sum = x(i)
         do 30 k=ndiags-1, maindiag+1, -1
*
*     offset(k) always be positive for k > maindiag
*
            l = i + offset(k)
            if (l.le.n-1) then
               sum = sum - A(i,k) * x(l)
            end if
 30      continue
         x(i) = sum / A(i,maindiag)
 40   continue
      return
```

The subroutine `dforwBackRILU` has six input parameters, where `A` is a matrix (factorized by `dfactRILU`) with $n \times ndiags$ entries, `offset` is the index vector, `b` is the right-hand side vector with length n, and `x` is a vector with length n for the solution. The solution is obtained by first doing forward substitution and then a backward substitution. The backward substitution is essentially the same as forward substitution, but we are solving from the opposite end of the matrix. The solution of the linear system is stored in the vector `x` when the subroutine returns.

Now we move over to the subroutine for performing a matrix-vector product, that is, `dprod`. The subroutine takes the following form:

```
      subroutine dprod(A, n, ndiags, offset, x, y, trans)
      integer            n, ndiags, trans
      double precision   A(0:n-1, 0:ndiags-1), x(0:n-1), y(0:n-1)
      integer            offset(0:ndiags-1)

      integer            i, j, k, maindiag
      integer            offset2index
      external           offset2index

      maindiag = offset2index(ndiags, offset, 0)

      if (trans.eq.0) then
*
*        i and j refer to row and col of original matrix
*
         do 20 i=0, n-1
            do 10 k=0, ndiags-1
               j = i + offset(k)
               if ((j.ge.0) .and. (j.le.n-1)) then
                  y(i) = y(i) + A(i, k)*x(j)
               end if
 10         continue
 20      continue
      else
*
*     transposed matrix, loops organized to traverse storage row by row
*
*     i and j refer to row and col of transposed matrix
*
         do 40 j=0, n-1
            do 30 k=0, ndiags-1
               i = j + offset(k)
               if ((i.ge.0) .and. (i.le.n-1)) then
                  y(i) = y(i) + A(j, k)*x(j)
               end if
 30         continue
 40      continue
      end if
      return
```

The `dprod` subroutine takes seven parameters as input, where `A` is the $n \times ndiags$ matrix, `offset` is the index vector, `x` is the vector in the matrix-vector product, `y` is the vector where the solution will be stored, and `trans` is an integer (0 or 1) determining whether we should compute the matrix-vector product $\mathbf{y} = \mathbf{Ax}$ or the transposed matrix-vector product $\mathbf{y} = \mathbf{A}^T\mathbf{x}$. The code is pretty much straightforward.

The subroutines for the other types is more or less identical to the ones presented here, except for the type of course. They are all located in the file `pypdelib/Lib/matrix/src/sspmatrix.f`.

The next step is to make Python interfaces for all of these subroutines. This is easy with aid from F2PY, however, it is unnecessary to interface every subroutine manually. This would also be hard to maintain. Instead we can follow the same ideas as in SciPy for interfacing LAPACK and BLAS subroutines. What SciPy does, is to create a generic F2PY signature

file with only one interface for all types of a specific subroutine. In SciPy, these files are called `generic_flapack.pyf` and `generic_fblas.pyf`. Then, when setting up SciPy, a script named `interface_gen.py` ensures that each function in the signature file are interfaced with the requested type. By using the same method on the subroutines located in `sspmatrix.f`, we only need to interface manually the three generic functions `xfactRILU`, `xforwBackRILU`, and `xprod`. These three interfaces are placed together in a file called `generic_sspmatrix.pyf` in the directory `pypdelib/Lib/matrix`. As an example, we list here the interface for `xfactRILU`:

```
subroutine <tchar=s,d,c,z>factrilu(omega,a,n,ndiags,offset)

    threadsafe
    callstatement {(*f2py_func)(&omega,a,&n,&ndiags,offset);}
    callprotoargument <type_in_c>*,<type_in_c>*,int*,int*,int*

    <type_in> optional,intent(in) :: omega = <type_convert=1.0>
    <type_in> dimension(n,ndiags),intent(in,out,copy) :: a
    integer intent(hide),depend(a) :: n = shape(a,0)
    integer intent(hide),depend(a) :: ndiags = shape(a,1)
    integer dimension(ndiags),intent(in),depend(ndiags) :: offset

end subroutine <tchar=s,d,c,z>factrilu
```

The syntax is the same as in `generic_flapack.pyf`, which was thoroughly explained in the previous section. The other subroutines, `xforwBackRILU` and `xprod`, are interfaced in a similar manner. We can now run the script `interface_gen.py` from the SciPy package on the `generic_sspmatrix.pyf`-file to created Python interfaces for all the different types of the three functions. The Python module for structured sparse matrices, called `sspmatrix`, is then complete and ready to be used in the matrix library.

1.6 The Build Process

The matrix library described in the previous sections, is implemented in a Python module called `matrix`. This module is part of a bigger package, which we have called PyPDELib; that is, a Python library for solving PDEs. So, before we can use the functionality provided by the matrix library, we first need to install the PyPDELib package.

Since the PyPDELib package depends heavily on the SciPy package, we will need to install that package prior to installing PyPDELib. For a guide on how to install SciPy and required libraries, see Chapter A.1.5 in [17]. The SciPy source code needs to be updated with several files, including the files `generic_flapack.pyf` and `generic_fblas2.py`, which we edited in Section 1.4. These files must be placed in the `Lib/linalg` subdirectory of the SciPy source code. The easiest way of doing this, is by copying the contents of the directory `pypdelib/scipy_ext` (including subdirectories) to the root of the SciPy source code tree. In some Unix shell, the following command preforms the necessary copying:

```
cp -rf scipy_ext/* /path/to/scipy_source
```

The SciPy package should then be built again.

Next, to enable the use of the `SparseMatrix` class, one should also install the PySparse package. This package is available in the directory `Lib/sandbox` in the SciPy source.

After installing SciPy and PySparse, we are ready to install the PyPDELib package. The standard way for installing a Python package, is to use Python's Distutils (Distribution Utilities) tool, which comes with the standard Python distribution. Here we have used NumPy's Distutils

instead which works similarly, but has additional features. See the NumPy Distutils Users Guide [32] for more information. Using this tool, we need to create a script `setup.py`, which calls various Distutils functionality. Then, to install the package, we only need to run the command

```
python setup.py install
```

This will install the package into a subdirectory in the "official" Python installation directory, which usually is

```
sys.prefix + '/lib/pythonX/site-packages'
```

where `X` reflects the version of Python. This requires the user to have write permissions in subdirectories of `sys.prefix`. If not, the user might use the `--home` option when running `setup.py`:

```
python setup.py install --home=$HOME/some/path
```

This will install the package into `$HOME/some/path/lib/python` and the user then needs to make sure that this path is specified in the `PYTHONPATH` environment variable.

Now that we have installed the PyPDELib package, we should run some tests to make sure that everything works as intended. First we try to import the module into Python:

```
python -c 'import pypdelib'
```

If this command results in no output, the installation of PyPDELib was successful. The next step is then to test all the different features provided by the PyPDELib package. To this end, we have created a series of tests based on a verification strategy referred to as unit testing. For an explanation of unit testing in Python one can consult Chapter 23.3 in the Python Library Reference [33]. The following commands invokes the tests for the PyPDELib package:

```
>>> import pypdelib
>>> pypdelib.test()
  Found 208 tests for pypdelib.matrix
  Found 0 tests for __main__
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.............
  ----------------------------------------------------------------------
Ran 366 tests in 0.669s

OK
<unittest.TextTestRunner object at 0xb4c0fb4c>
>>>
```

If a similar output appears on your screen, then the PyPDELib package should be ready for use.

We will in the following three sections look at several examples on how to use the different functionality that the matrix library in the PyPDELib package provides. We start by looking at matrix-vector products, then we solve a simple 1D stationary problem, and finally we test the matrix library extensively in both 1D and 2D time-dependent diffusion problems. Most of the code used in these examples are available in the directory `pypdelib/examples`.

1.7 Example: Matrix-Vector Products

Suppose we want to compute the product of a given matrix and a vector. A matrix-vector product would in the mathematical language be expressed as: Given $\mathbf{A} \in \mathbb{R}^{m,n}$ and $\mathbf{x} \in \mathbb{R}^n$, compute $\mathbf{y} = \mathbf{A}\mathbf{x}$, $\mathbf{y} \in \mathbb{R}^m$.

Let us first see how a matrix-vector product would be expressed in a computing language like Fortran 77. First, we define some relevant data structures, which in this case would be

```
integer m, n
double precision A(m,n)
double precision x(n), y(m)
```

Given these data items, we may simply call a routine `prodv_densem` for the matrix-vector product:

```
call prodv_densem(A, m, n, x, y)
```

This is simple and straightforward, however, the call to `prodv_densem` involves details on the array sizes that are not explicitly needed in the mathematical formulation $\mathbf{y} = \mathbf{A}\mathbf{x}$. In addition, if the matrix is some sort of a sparse matrix, we would need to call a different matrix-vector product routine than `prodv_densem`, which is specialized for the sparse matrix in question. For example, if the matrix is a n-by-n tridiagonal matrix, we would probably call a routine `prodv_tridiagm` for the matrix-vector product:

```
call prodv_tridiagm(dl, d, du, n, x, y)
```

Here, `dl`, `d`, and `du` are respectively the lower-, main-, and upper-diagonals of the tridiagonal matrix, represented as vectors.

Now let us see how we can compute a matrix-vector product using our matrix library. We start by defining the necessary data structures:

```
A = DenseMatrix(m, n)
x = zeros(n)
```

This will setup a $m \times n$ `DenseMatrix` instance `A` and a NumPy array `x` with length n (both initially filled with zeros). After filling in the matrix and vector entries, we can compute the matrix-vector product by calling

```
y = A*x
```

which is the exact same syntax as in the mathematical formulation. In case of a tridiagonal matrix, we would instead of a `DenseMatrix` instance use a `TriDiagMatrix` instance:

```
A = TriDiagMatrix(n)
```

This initializes a `TriDiagMatrix` instance with $n \times n$ entries. Again, after filling in the matrix entries, we can compute the matrix-vector product by the same syntax as for a matrix-vector product with a dense matrix, i.e., `y = A*x`. This is also the case for all the other matrix formats available in the matrix library.

Note that the reason we can use the expression `y = A*x` to compute the matrix-vector product, is that we have implemented the special method `__mul__` in the base class `MatrixBase`. This allows the use of the `*` operator between a matrix object (subclass of `MatrixBase`) and a NumPy array to represent a matrix-vector product. What the `__mul__` method actually does,

is to call the `prod` method in the matrix class. This means that the expression `y = A*x` is equivalent to the expression `y = A.prod(x)`.

To get more control over the matrix-vector product, we can use the `prod` method in the matrix class directly. For example, if we need to compute a transposed matrix-vector product, we must call the `prod` method directly. A transposed matrix-vector product is defined as: Given $\mathbf{A} \in \mathbb{R}^{m,n}$ and $\mathbf{x} \in \mathbb{R}^m$, compute $\mathbf{y} = \mathbf{A}^T\mathbf{x}$, $\mathbf{y} \in \mathbb{R}^n$. Here is how we compute a transposed matrix-vector product:

```
y = A.prod(x, transpose=1)
```

So, by setting the keyword argument `transpose` to 1, we get the transposed matrix-vector product. The default value for `transpose` is 0, i.e., no transpose. If instead the matrix is an Hermitian matrix and we want to compute the matrix-vector product $\mathbf{y} = \mathbf{A}^H\mathbf{x}$, we do this by calling the `prod` method in the matrix instance with `transpose=2`.

In many iterative methods for solving linear systems, like the Conjugate Gradient Method, one or more matrix-vector products are performed in each iteration. The expression `y = A*x` will in such cases not be optimal since the result array `y` needs to be allocated each time. Instead we can use the following expression:

```
y = A.prod(x, result=y)
```

This ensures that no extra storage needs to be allocated for the result. The overhead can be reduced by as much as a factor of 1.4 depending on the size of the problem and the number of iterations.

1.8 Example: Solving a System of Linear Algebraic Equations

In this example, which is based heavily on the text in Chapter 1.3 in [16], we will create a simulation script for solving the simple one-dimensional boundary value problem

$$-u''(x) = f(x), \quad x \in (0,1), \tag{1.1}$$
$$u(0) = 0, \tag{1.2}$$
$$u'(1) = 1. \tag{1.3}$$

Discretization. By using a standard finite difference method on these equations, we will end up with the following algebraic equations:

$$u_0 = 0, \tag{1.4}$$
$$u_{i+1} - 2u_i + u_{i-1} = -h^2 f(x_i), \quad i = 1, \ldots, n-1, \tag{1.5}$$
$$2u_{n-1} - 2u_n = -2h - h^2 f(x_n). \tag{1.6}$$

Here, we have partitioned the domain $(0,1)$ into n cells $[x_i, x_{i+1}]$, $i = 0, ...n-1$, with $x_0 = 0$ and $x_n = 1$. The cell length $h = x_{i+1} - x_i$ is assumed to be constant and u_i represents the numerical approximation to the exact solution $u(x_i)$ for $i = 0, \ldots, n$.

The equations in (1.4)-(1.6) can be written on matrix form $\mathbf{Au} = \mathbf{b}$, where $\mathbf{A}$ is an $(n+1) \times (n+1)$ matrix, $\mathbf{u}$ is the unknown vector, and $\mathbf{b}$ is the right-hand side vector. The matrix

entries $A_{i,j}$ are easily identified from the scheme (1.4)-(1.6):

$$A_{0,0} = 1, \tag{1.7}$$
$$A_{i,i-1} = 1, \quad i = 1, \ldots, n-1, \tag{1.8}$$
$$A_{i,i} = -2, \quad i = 1, \ldots, n, \tag{1.9}$$
$$A_{i,i+1} = 1, \quad i = 1, \ldots, n-1, \tag{1.10}$$
$$A_{n,n-1} = 2. \tag{1.11}$$

The rest of the entries in the matrix $\mathbf{A}$ are filled with zeros. The unknown vector is $\mathbf{u} = (u_0, \ldots, u_n)^T$, while the entries in the right-hand side vector $\mathbf{b} = (b_0, \ldots, b_n)^T$ are given by

$$b_0 = 0, \quad b_i = -h^2 f(x_i),\ i = 1, \ldots, n-1, \quad b_n = -2h - h^2 f(x_n). \tag{1.12}$$

Implementation. We now want to use the functionality provided by the matrix library to solve the linear system $\mathbf{Au} = \mathbf{b}$. To this end, we will create a simulator script in Python that generates and solves the discrete equations in (1.7)-(1.12). To verify that our implementation is correct, we need a test problem where an exact solution is known. One can show that if we let the right hand side in (1.1) be given as $f(x) = \gamma \exp(-\beta x)$, we end up with the following solution:

$$u(x) = \frac{\gamma}{\beta^2}\left(1 - e^{-\beta x}\right) + \left(1 - \frac{\gamma}{\beta}e^{-\beta}\right)x, \quad \beta \neq 0, \tag{1.13}$$
$$u(x) = x\left(1 + \gamma\left(1 - \frac{1}{2}x\right)\right), \quad \beta = 0. \tag{1.14}$$

A very attractive feature of the problem (1.1)-(1.3) with $f(x) = \gamma \exp(-\beta x)$ is that the numerical solution is exact for $\beta = 0$, regardless of the values of n and γ. So, running the code with $\beta = 0$ should result in a zero error (within machine precision). This result is useful for partially verifying the implementation.

The Code. We are now ready to make a simple Python script that solves (1.1)-(1.3) numerically. The script should first build the linear system $\mathbf{Au} = \mathbf{b}$ according to the formulas in (1.7)-(1.12), then solve the linear system, and at the end write the computed solution and the error to the screen in addition to some CPU time measurements.

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

from pypdelib import *
from scitools.numpytools import *
from math import *
import time

n = int(raw_input("Give number of solution points: "))
h = 1/float(n)
A = DenseMatrix(n+1, n+1)
b = zeros(n+1, float)
u = zeros(n+1, float)
beta = float(raw_input("Give beta: "))
gamma = float(raw_input("Give gamma: "))

t0 = time.clock()  # measure CPU time
# upper boundary:
A[0,0] = 1.0
b[0] = 0.0
```

```
# inner grid points:
for i in iseq(1, n-1):
    x = i*h
    A[i,i] = -2.0
    A[i,i-1] = A[i,i+1] = 1.0
    b[i] = -h**2*gamma*exp(-beta*x)

# lower boundary:
x = n*h
A[n,n-1] = 2.0;  A[n,n] = -2.0
b[n] = -2*h - h**2*gamma*exp(-beta*x)
t_init = time.clock() - t0

if n <= 8:
    print "A matrix\n", A
    print "right-hand side\n", b
# factorize matrix:
t0 = time.clock()
A.factorize()
t_fact = time.clock() - t0
# solve linear system:
t0 = time.clock()
u = A.solve(b)
t_solve = time.clock() - t0

# write out the solution and the error
if n <= 100:
    print "\n\n x           numerical       error:\n"
    for i in iseq(0, n):
        x = i*h
        if float_eq(beta, 0):  # is beta zero?
            u_exact = x*(1 + gamma*(1 - 0.5*x))
        else:
            u_exact = gamma/(beta**2)*(1 - exp(-beta*x)) + \
                      (1 - gamma/beta*exp(-beta))*x
        print "%4.3f        %8.5f       %12.5e" % \
              (x, u[i], u_exact-u[i])
print "\nn: %d, cpu init: %s, cpu factorize: %s, cpu solve: %s" \
      % (n, t_init, t_fact, t_solve)
```

Let us look closer at our simulator script. The first thing we need to do, is to import necessary data structures and functions. This is done in the first four lines below the Python heading. In the next code block, we do all initializations of data structures that we are going to need later on. The most important part here, however, is the initialization of the matrix and the storage allocations for the unknown and the right-hand side vector. The matrix is represented by a `DenseMatrix` instance:

```
A = DenseMatrix(n+1, n+1)
```

This will initialize a `DenseMatrix` instance with $(n+1)\times(n+1)$ entries where all the entries are automatically filled with zeros. The unknown and the right-hand side vectors are represented by two NumPy arrays with length $n+1$, also filled with zeros:

```
b = zeros(n+1, float)
u = zeros(n+1, float)
```

After the initialization phase, we can build the linear system by filling in the entries in the matrix `A` and the right-hand side vector `b` with values according to the formulas in (1.7)-(1.12). When this has completed, we are ready to solve the linear system. This is done in two steps: (i) factorizing the coefficient matrix and (ii) solve using the factorized matrix:

```
A.factorize()
u = A.solve(b)
```

As we saw on page 6, these two steps can also be carried out by the more compound statement `u = A**(-1)*b`.

Now that we have the solution of the linear system stored in the array `u`, the next step is to verify that the implementation is correct. To this end, we compute the exact solution from the formulas in (1.13)-(1.14). The numerical solution and the error are then written to the screen and at the end we also write out some CPU times for the different parts of the script.

Running the Script. The Python script described above is located in the file `poisson1d.py` in the directory `pypdelib/examples/poisson1d` and it can be started by giving the command

```
python poisson1d.py
```

that is, if the current working directory is `pypdelib/examples/poisson1d`. Running this command, will start the script, which in turn will ask questions and prompt the user for answers. For example, we can give values like $n = 3$, $\beta = 0$, and an arbitrary γ as input, to verify that the numerical solution is exact also for small grids. Doing so, results in the following output:

```
 x           numerical        error:

0.000         0.00000        0.00000e+00
0.333       278.11111         0.00000e+00
0.667       445.11111         5.68434e-14
1.000       501.00000         5.68434e-14
```

We can try different values of n, β, and γ and observe the behavior of the numerical error.

Optimization. Our simulator script has two serious drawbacks. First of all, explicit loops over array entries in Python are known to be slow. We should therefore try to avoid loops and instead express our mathematical algorithm in terms of (NumPy) array operations, a technique known as vectorization. The loop that we want to vectorize, takes the following form:

```
for i in iseq(1, n-1):
    x = i*h
    A[i,i] = -2.0
    A[i,i-1] =  A[i,i+1] = 1.0
    b[i] = - h**2*gamma*exp(-beta*x)
```

The right-hand side vector `b` is easy to vectorize. First we need to create an array of x-values:

```
x = seq(0, 1, h)
```

The vectorized expression can then be written as

```
b[1:-1] = -h**2*gamma*exp(-beta*x[1:-1])
```

To be able to vectorize the loop over the matrix entries, we need to have knowledge of how the data are stored in the `DenseMatrix` class. By looking at the `DenseMatrix` class, we can see that the matrix entries are stored in a NumPy array `m` with shape (`nrows`, `ncolumns`) or in the current problem (`n`+1, `n`+1). By working on the class variable `A.m` directly, we can then replace the code in the `for` loop with the following vectorized expression:

```
ind0 = iseq(1,n-1)
ind1 = iseq(0,n-2)
ind2 = iseq(2,n)
A.m[ind0,ind0] = -2.0
A.m[ind0,ind1] = A.m[ind0,ind2] = 1.0
```

This should speed up the code significantly as we will see later.

The second drawback in our script, is that we are not taking advantage of the fact that the matrix $\mathbf{A}$ is tridiagonal, as can be seen from (1.7)-(1.11). This means that each row in $\mathbf{A}$ has at most three nonzero entries and by utilizing this fact, we can increase the computational efficiency considerably in addition to reducing the memory requirements from $\mathcal{O}(n^2)$ to $\mathcal{O}(3n)$.

To change the script to use the tridiagonal matrix structure instead of the current dense structure, we only need to do some minor modifications. The first is in the line where the matrix is initialized, that is:

```
A = DenseMatrix(n+1, n+1)
```

This allocates memory for a full $(n+1) \times (n+1)$ matrix. To specify a tridiagonal matrix instead, we simply replace the line with the following statement:

```
A = TriDiagMatrix(n+1)
```

This is actually all that is needed to use the tridiagonal matrix structure instead of the dense structure, but only if we use the `for` loop to set the inner grid points as we did in the original script. However, as noted above, we should avoid explicit loops over array entries if possible and instead use vectorized expressions. To this end, we need to know the underlying data structure of the `TriDiagMatrix` class. By looking at the `TriDiagMatrix` class, we can see that it uses three 1D NumPy arrays, one for each of the three diagonals in the matrix. They are named `dl`, `d`, and `du` for the lower-, main-, and upper-diagonal respectively. The array `d` has length n while `dl` and `du` has length $n-1$. This means that the `for` loop over the matrix entries can be vectorized using the following expression:

```
A.d[1:-1] = -2.0
A.dl[:-1] = A.du[1:] = 1.0
b[1:-1] = - h**2*gamma*exp(-beta*x[1:-1])
```

The complete optimized script now takes the following form:

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

from pypdelib import *
from scitools.numpytools import *
import time

n = int(raw_input("Give number of solution points: "))
h = 1/float(n)
x = seq(0, 1, h)
A = TriDiagMatrix(n+1)
b = zeros(n+1, float)
u = zeros(n+1, float)
beta = float(raw_input("Give beta: "))
gamma = float(raw_input("Give gamma: "))

t0 = time.clock()  # measure CPU time
# upper boundary:
A[0,0] = 1.0
b[0] = 0.0
```

```
# inner grid points:
A.d[1:-1] = -2.0
A.dl[:-1] = A.du[1:] = 1.0
b[1:-1] = - h**2*gamma*exp(-beta*x[1:-1])

# lower boundary:
A[n,n-1] = 2.0;  A[n,n] = -2.0
b[n] = -2*h - h**2*gamma*math.exp(-beta*x[n])
t_init = time.clock() - t0

if n <= 8:
    print "A matrix\n", A
    print "right-hand side\n", b
# factorize matrix:
t0 = time.clock()
A.factorize()
t_fact = time.clock() - t0
# solve linear system:
t0 = time.clock()
u = A.solve(b)
t_solve = time.clock() - t0

# write out the solution and the error
if n <= 100:
    print "\n\n x           numerical       error:\n"
    for i in iseq(0, n):
        x = i*h
        if float_eq(beta, 0):  # is beta zero?
            u_exact = x*(1 + gamma*(1 - 0.5*x))
        else:
            u_exact = gamma/(beta**2)*(1 - exp(-beta*x)) + \
                      (1 - gamma/beta*exp(-beta))*x
        print "%4.3f        %8.5f       %12.5e" % \
              (x, u[i], u_exact-u[i])
print "\nn: %d, cpu init: %s, cpu factorize: %s, cpu solve: %s" \
      % (n, t_init, t_fact, t_solve)
```

To measure the efficiency gain we get by using `TriDiagMatrix` instead of `DenseMatrix` and by vectorized expressions instead of scalar Python loops, we have compared the four versions of the script described above:

1. `DenseMatrix` and plain loops,
2. `DenseMatrix` and vectorized implementations of loops,
3. `TriDiagMatrix` and plain loops, and
4. `TriDiagMatrix` and vectorized implementations of loops.

Running the script with a grid with $n = 7000$ grid points, results in the following table:

matrix format	loops	set-up	factorize	solve
`DenseMatrix`	scalar	0.09	185.92	0.55
`DenseMatrix`	vectorized	0.01	186.14	0.52
`TriDiagMatrix`	scalar	0.07	0.0008	0.0006
`TriDiagMatrix`	vectorized	0.0013	0.0008	0.0005

We can see that the first version required a total of 186.6 s while the second version ran at 186.7 s, implying a negligible difference in speed. This is because over 99% of the CPU time

is used for factorizing the matrix as can be seen in the table. The third version ran at 0.07 s and the fourth version at 0.003 s, but for such small numbers, the results are not reliable and hard to compare. Therefore, we increase the grid to $n = 5000000$ grid points, resulting in the following table:

matrix format	loops	set-up	factorize	solve
`TriDiagMatrix`	scalar	50.64	1.1	0.56
`TriDiagMatrix`	vectorized	1.05	1.1	0.56

For such a large grid we have no result for the first and second version of the script as this would have taken to long time (if we had enough memory). However, the third version required a total of 52.3 s, while the fourth version ran at only 2.7 s. This is a factor of 19 in favor of the fourth version and in this case it definitely pays off to vectorize the loops. What we also should note about the results from the fourth version of the script, is the fact that about 63% of the time is used to factorize and solve the linear system. This means that most of the work is being done in the underlying Fortran routines in the LAPACK library.

While the speed-up by vectorization was very good, we should note that migrating the loops to a compiled language like C or Fortran 77 may speed things up even more. This will be one of the topics in the next section.

1.9 Example: 1D and 2D Diffusion Equations

In this example, we will test the efficiency of our matrix solve library with implicit finite difference methods for a given linear problem. We will use the following time-dependent diffusion problem as our test problem:

$$\frac{\partial u}{\partial t} = k\nabla^2 u + f(\mathbf{x}, t), \quad \mathbf{x} \in \Omega \subset \mathbb{R}^d,\ t > 0, \tag{1.15}$$

$$u(\mathbf{x}, t) = g(\mathbf{x}, t), \quad \mathbf{x} \in \partial\Omega_E,\ t > 0, \tag{1.16}$$

$$u(\mathbf{x}, 0) = I(\mathbf{x}), \quad \mathbf{x} \in \Omega,\ t = 0. \tag{1.17}$$

Here, $u(\mathbf{x}, t)$ is the primary unknown, k is a constant diffusion coefficient, $f(\mathbf{x}, t)$ is a diffusion source term, $I(\mathbf{x})$ is a given initial condition, $g(\mathbf{x}, t)$ is the Dirichlet boundary values for u, and $\partial\Omega_E$ is the boundary of the domain Ω. The subscript E indicates that $\partial\Omega_E$ has essential boundary conditions.

We start by looking at the one-dimensional case and then we will look at the two-dimensional case later.

1.9.1 1D Diffusion Equation

In one dimension, the mathematical problem in (1.15)-(1.17) translates to the following problem:

$$\frac{\partial u}{\partial t} = k\frac{\partial^2 u}{\partial x^2} + f(x, t), \quad x \in (0, L) \quad t > 0, \tag{1.18}$$

$$u = g(x, t), \quad x = 0, L, \tag{1.19}$$

$$u(x, t) = I(x), \quad x \in [0, L]. \tag{1.20}$$

Here, we have selected the domain to be $\Omega = [0, L]$. A second order centered finite difference method in space, combined with a θ-rule in time, is suitable for solving (1.18) numerically. By using the compact notation explained in Appendix A3 in [16], the scheme for the current problem can be written as

$$[\delta_t u]_i^{\ell-\frac{1}{2}} = \theta k[\delta_x\delta_x u]_i^{\ell} + (1-\theta)k[\delta_x\delta_x u]_i^{\ell-1} + [f]_i^{\ell} + (1-\theta)[f]_i^{\ell-1}. \tag{1.21}$$

Writing out this in detail, we get

$$\frac{u_i^{\ell} - u_i^{\ell-1}}{\Delta t} = \theta k\frac{u_{i-1}^{\ell} - 2u_i^{\ell} + u_{i+1}^{\ell}}{\Delta x^2} + (1-\theta)k\frac{u_{i-1}^{\ell-1} - u_i^{\ell-1} + u_{i+1}^{\ell-1}}{\Delta x^2} + \theta f_i^{\ell} + (1-\theta)f_i^{\ell-1} \tag{1.22}$$

where $f_i^{\ell} = f(x_i, t_\ell)$. The parameter Δt is the time step: $\Delta t = t_\ell - t_{\ell-1}$ and $t_\ell = \ell\Delta t$. The grid increment is assumed to be constant, $\Delta x = L/n$.

By collecting the unknown new values at time level ℓ on the left-hand side and the previously computed values on the right-hand side, we can see that we are dealing with a linear tridiagonal system:

$$-\theta k\frac{\Delta t}{\Delta x^2}u_{i-1}^{\ell} + \left(1 + 2\theta k\frac{\Delta t}{\Delta x^2}\right)u_i^{\ell} - \theta k\frac{\Delta t}{\Delta x^2} = u_i^{\ell-1} + (1-\theta)k\frac{\Delta t}{\Delta x^2}\left(u_{i-1}^{\ell-1} - 2u_i^{\ell-1} + u_{i+1}^{\ell-1}\right) + \theta\Delta t f_i^{\ell} + (1-\theta)\Delta t f_i^{\ell-1}. \tag{1.23}$$

When $\theta = 0$, we get an explicit scheme where the new values u_i^{ℓ} are computed by an explicit formula involving only known values $u_i^{\ell-1}$. The explicit scheme is stable only when the time step Δt is small, or more precisely, it is only stable for $\Delta t \leq \Delta x^2/2$. However, when $\theta > 0$, we get an implicit scheme; that is, the new u_i^{ℓ} values are coupled in a system of linear algebraic equations: $\mathbf{Au} = \mathbf{b}$, where $\mathbf{A}$ is a tridiagonal $(n+1)\times(n+1)$ matrix, $\mathbf{u}$ is the vector of new values u_i^{ℓ}, and $\mathbf{b}$ is the right-hand side vector. We can easily identify the matrix entries as

$$A_{i,i-1} = -\theta k\frac{\Delta t}{\Delta x^2}, \quad A_{i,i} = 1 + 2\theta k\frac{\Delta t}{\Delta x^2}, \quad A_{i,i+1} = A_{i,i-1},$$

and the right-hand side values as

$$b_i = u_i^{\ell-1} + (1-\theta)k\frac{\Delta t}{\Delta x^2}\left(u_{i-1}^{\ell-1} - 2u_i^{\ell-1} + u_{i+1}^{\ell-1}\right) + \theta\Delta t f_i^{\ell} + (1-\theta)\Delta t f_i^{\ell-1}.$$

The scheme (1.23) is unconditionally stable when $\theta \geq 1/2$ and the accuracy is of second order in both time and space when $\theta = 1/2$. When $\theta \neq 1/2$ the order in time is reduced to Δt for the accuracy. The choice $\theta = 1/2$, known as the Crank-Nicolson scheme, is therefore popular, since it appears to be the optimal combination of stability and accuracy.

Handling the initial condition and the boundary condition is straightforward. The initial condition, $u(x, 0) = I(x)$, is set by $u_i^0 = I_i$, and then we use (1.23) for all time levels $\ell > 0$. The boundary values are implemented by setting $A_{i,i} = 1$ and $b_i = g_i^{\ell}$ at the end points, i.e. $i = 0, n$, and using (1.23) for the inner points. The complete numerical method is summarized in the following algorithm:

define u_i and u_i^- to represent u_i^{ℓ} and $u_i^{\ell-1}$ respectively
SET THE INITIAL CONDITION:

$u_i = I_i, \quad \text{for } i = 0, \ldots, n$
$t = 0$
while time $t \leq t_{\text{stop}}$
$t \leftarrow \Delta t$
DEFINE TRIDIAGONAL SYSTEM:
$\mathbf{Au} = \mathbf{b}$, where
$\mathbf{u} = (u_0, \ldots, u_n)^T$,
$A_{i,i-1} = -\theta k \frac{\Delta t}{\Delta x^2},\ A_{i,i} = 1 + 2\theta k \frac{\Delta t}{\Delta x^2},\ A_{i,i+1} = -\theta k \frac{\Delta t}{\Delta x^2},$
$b_i = u_i^- + (1-\theta) k \frac{\Delta t}{\Delta x^2} \left(u_{i-1}^- - 2u_i^- + u_{i+1}^-\right) + \theta f_i^\ell + (1-\theta) f_i^{\ell-1}$
for $i = 1, \ldots, n-1$, and
$A_{0,0} = 1,\ A_{n,n} = 1,\ b_0 = g_0^\ell, b_n = g_n^\ell$
(All other $A_{i,j}$ values are zero)
SOLVE THE SYSTEM $\mathbf{Au} = \mathbf{b}$
INITIALIZE FOR NEXT STEP:
$u_i^- = u_i, \quad \text{for } i = 0, \ldots, n$

A First Implementation in Python. We are now ready to create a implementation in Python for solving the model problem (1.18)-(1.20). The algorithm above can be straightforwardly implemented in Python using the functionality provided by the matrix library. It is implemented in a function called `solver0` that can be found in the module `diffusion1d` in the directory `pypdelib/examples/diffusion1d`. The `solver0` function takes the following form:

```
def solver0(I, f, k, bc, L, n, dt, tstop, theta,
            user_action=None):
    # f and bc are functions of x and t, I is a function of x
    dx = L/float(n)
    x = sequence(0, L, dx)       # grid points in x dir

    C1 = theta*dt*k/dx**2        # help variable
    C2 = (1-theta)*dt*k/dx**2    # help variable

    u  = zeros(n+1, float)       # solution array

    # set initial condition:
    t = 0.0
    for i in iseq(0,n):
        u[i] = I(x[i])
    up = u.copy()                # solution at previous time step

    if user_action is not None:
        user_action(u, x, t)     # allow user to plot etc.

    while t <= tstop:
        t_old = t;  t += dt

        A = TriDiagMatrix(n+1)   # coefficient matrix
        b = zeros(n+1, float)    # right-hand side vector

        # lower boundary:
        i = 0;  A[i,i] = 1;  b[i] = bc(x[i], t);
        # update all inner points:
        for i in iseq(1, n-1):
            A[i,i-1] = -C1
            A[i,i] = 1 + 2*C1
            A[i,i+1] = -C1
            b[i] = up[i] + C2*(up[i-1] - 2*up[i] + up[i+1]) + \
                   theta*dt*f(x[i], t) + \
                   (1-theta)*dt*f(x[i], t_old)
```

```
        # upper boundary:
        i = n;  A[i,i] = 1;  b[i] = bc(x[i], t)

        u = A**(-1)*b  # solve linear system

        if user_action is not None:
            user_action(u, x, t)

        # update data structures for next step:
        up, u = u, up
```

One should note the following points about the `solver0` function:

1. The loop over the inner points are implemented using a straight Python `for` loop. This is fine for small problems, but as the grid gets larger it will pay off to vectorize the loop or migrate the loop to a compiled language like Fortran or C/C++.

2. The coefficient matrix $\mathbf{A}$ is equal for all time steps $t > 0$. This means that it is only necessary to build it once; before the start of the time loop. It also means that the matrix only need to be factorized once. This will reduce the CPU time considerably since factorizing the matrix is one of the most demanding operations wrt. CPU time.

3. The callback function `user_action(u, x, t)` can be used to process the solution during a simulation. The `user_action` function can for instance be used for visualizing the solution or computing errors from an analytical solution.

We can illustrate the usage of the `user_action` function through a couple of examples. First we will use the `user_action` function to verify that the computed solution is correct. We will then look at how we can use the `user_action` function to visualize the solution by a curve plot.

Computing Errors. To verify that the implementation is correct, we need to construct a test problem where an analytical solution is known. To this end, the particular choice of $I(x) = 0$, $f(x,t) = \pi^2 \sin \pi x$, and $g(x,t) = 0$ corresponds to the analytical solution

$$u(x,t) = \left(1 - e^{-\pi^2 t}\right) \sin \pi x. \tag{1.24}$$

The following function computes the error at every time steps and constitutes a verification of the `solver0` function:

```
def test_error0():
    L = 1;  k = 1

    def exact(x, t):
        return (1-exp(-pow(pi,2)*t))*sin(pi*x)

    def I(x):      return 0.0
    def f(x, t):   return pi**2*sin(pi*x)
    def bc(x, t):  return 0.0

    error = []
    def action(u, x, t):
        e = exact(x, t) - u
        error.append((t, sqrt(dot(e,e))))

    n = 100;  dt = 0.1;  tstop = 2;  theta = 0.5
    solver0(I, f, k, bc, L, n, dt, tstop, theta,
            user_action=action)
    for t, e in error:
        print 't=%10.2E  error=%10.2E' % (t, e)
```

Here we use $n = 100$ grid points, $\Delta t = 0.1$, and $\theta = 1/2$. Running the `test_error0` function will run the test problem and write out an error measure at each time level. When $t = 1$, the error is about $8 \cdot 10^{-4}$. By halving Δx and Δt, the error should be reduced by a factor of 4 since the scheme is of order Δx^2 and Δt^2 for $\theta = 1/2$. Changing the parameters in the `test_error0` function to `n=200` and `dt=0.05` results in an error of about $3 \cdot 10^{-4}$ when $t = 1$.

Visualization. The `user_action` function can also easily be used to visualize the solution u at each time level, for instance as a curve plot. In the following function we start with an initial Gaussian bell in the middle of the domain and we let the source term function and the boundary values be zero for all time levels $t > 0$:

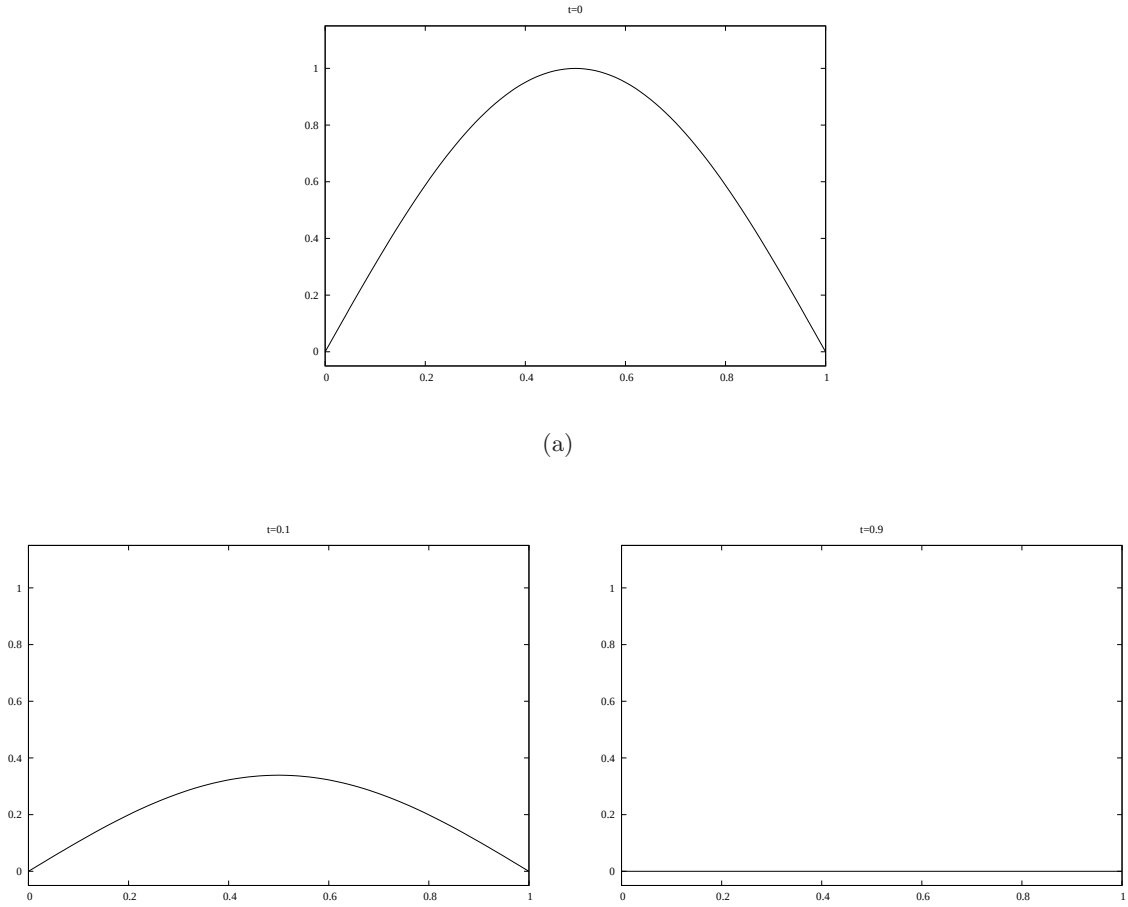
```
def test_plot0():
    L = 1;  k = 1

    def I(x):      return sin(pi*x/L)
    def f(x, t):   return 0.0
    def bc(x, t):  return 0.0

    def action(u, x, t):
        easyviz.plot(x, u, title="t=%g" % t,
                     ymin=-0.05, ymax=1.15)
        time.sleep(0.2) # pause between frames

    n = 100;  dt = 0.1;  tstop = 2;  theta = 0.5
    solver0(I, f, k, bc, L, n, dt, tstop, theta,
            user_action=action)
```

Here we have used the `plot` command from the Easyviz (from SciTools) package to plot the solution at each time level t. One should notice how we use the `ymin` and `ymax` keyword arguments to keep the y-axis fixed at every time step.

The `test_plot0` function can easily be extended to create a movie file from each of the plots in the simulation. At each time level we use the `hardcopy` command in Easyviz to store a copy of the plot to a file. We can then create a movie by using the `movie` command, also available in Easyviz, by providing a list of file names:

```
def test_movie0():
    L = 1;  k = 1

    def I(x):      return sin(pi*x/L)
    def f(x, t):   return 0.0
    def bc(x, t):  return 0.0

    files = []
    def action(u, x, t):
        easyviz.plot(x, u, title="t=%g" % t,
                     ymin=-0.05, ymax=1.15)
        filename = 'tmp_%020f.png' % t
        easyviz.hardcopy(filename=filename, color=True,
                         fontname='Times-Roman', fontsize=14)
        files.append(filename)
        time.sleep(0.2) # pause between frames

    n = 100;  dt = 0.1;  tstop = 2;  theta = 0.5
    solver0(I, f, k, bc, L, n, dt, tstop, theta,
            user_action=action)
    easyviz.movie(files)
```

The name of the resulting movie will be `movie.avi`, which is the default output name given by the `movie` command in Easyviz. Figure 1.1 provides some snapshots of the movie.



(a)

(b)

(c)

Figure 1.1: Plots of an initial bell in the middle of the domain at (a) $t = 0$, (b) $t = 0.1$, and (c) $t = 0.9$.

Vectorizing Loops. A major point regarding optimization in the `solver0` function, is the loop over the finite difference scheme which is implemented using a plain Python loop. Even for one-dimensional problems, plain Python loops can be too slow for large grids over long time spans; however, the speed can easily be increased by vectorizing the loops. The vectorized version of the inner loop over the matrix entries takes the following form:

```
A.dl[0:n-1] = -C1
A.d[1:n] = 1 + 2*C1
A.du[1:n] = -C1
```

while the loop over the inner entries in the right-hand side vector can be computed by the following vectorized expression:

```
b[1:n] = up[1:n] + C2*(up[0:n-1] - 2*up[1:n] + up[2:n+1]) + \
         theta*dt*f(x[1:n], t) + (1-theta)*dt*f(x[1:n], t_old)
```

For large grids over shorter time spans, we will also benefit from vectorizing the loop over the initial condition:

```
u[:] = I(x)
```

The vectorized expression of the loops speeds up the code considerably. For a grid with $n = 40000$ grid points with 100 time steps the solver with scalar implementation of loops required 68.2 s while the solver with the vectorized expressions ran at 1.05 s, implying a factor of 65 in favor of the latter. The speed-up by vectorization is great, however, migrating the loops to compiled code will speed things up even further as we will see later.

Supporting All Matrix Formats. One of our focuses in the current example will be on measuring the efficiency of all the different matrix formats that are available in the matrix library. We should therefore add the option to let the user select the matrix format for the coefficient matrix when calling the solver function. Since the vectorized expression over the matrix entries is dependent on the storage structure of the matrix, we need to make specialized versions for each of the matrix formats. For clarity, we will take the code for the different formats out into separate functions. The vectorized expressions for the different matrix formats are listed below.

```
def scheme_dense_vec(A, C1):
    n = A.n-1
    ind0 = iseq(1,n-1)
    ind1 = iseq(0,n-2)
    ind2 = iseq(2,n)
    i = 0;  A.m[i,i] = 1
    A.m[ind0,ind1] = -C1
    A.m[ind0,ind0] = 1 + 2*C1
    A.m[ind0,ind2] = -C1
    i = n;  A.m[i,i] = 1
    return A

def scheme_band_vec(A, C1):
    n = A.n-1
    i = 0;  A.m[2,i] = 1
    A.m[1,2:n+1] = -C1
    A.m[2,1:n] = 1 + 2*C1
    A.m[3,0:n-1] = -C1
    i = n;  A.m[2,i] = 1
    return A

def scheme_symmband_vec(A, C1):
    n = A.n-1
    i = 0;  A.m[1,i] = 1
    A.m[0,2:n] = -C1
    A.m[1,1:n] = 1 + 2*C1
    i = n;  A.m[1,i] = 1
    return A

def scheme_tridiag_vec(A, C1):
    n = A.n-1
    i = 0;  A.d[i] = 1
    A.dl[0:n-1] = -C1
    A.d[1:n] = 1 + 2*C1
    A.du[1:n] = -C1
    i = n;  A.d[i] = 1
    return A

def scheme_symmtridiag_vec(A, C1):
    n = A.n-1
    i = 0;  A.d[i] = 1
    A.dl[1:n-1] = -C1
    A.d[1:n] = 1 + 2*C1
    i = n;  A.d[i] = 1
    return A
```

```
def scheme_structsparse_vec(A, C1):
    n = A.n-1
    i = 0;  A.m[i,1] = 1
    A.m[1:n,0] = -C1
    A.m[1:n,1] = 1 + 2*C1
    A.m[1:n,2] = -C1
    i = n;  A.m[i,1] = 1
    return A

def scheme_sparse_vec(A, C1):
    return scheme_sparse_scalar(A, C1)
```

Note that the sparse matrix format is not vectorized. This is because it is based on the PySparse package, which does not support vectorized insertion of matrix entries. The coefficient matrix can now be built by the following expression:

```
func = 'scheme_'+matrix_format+'_'+implementation['scheme']
A = eval(func)(A, C1)
```

The string `matrix_format` can be either `'dense'`, `'band'`, `'symmband'`, `'tridiag'`, `'symmtridiag'`, `'structsparse'`, or `'sparse'`. The dictionary `implementation` specifies the particular implementations to be used for the loop over the initial conditions (`'ic'`) and the loop over the finite difference scheme (`'scheme'`). The values can be either `'scalar'` or `'vec'` for plain Python loops (scalar implementation) or vectorized expressions respectively.

Following the same idea, we also take out the initial condition and the updating of the right-hand side into separate functions. These two functions takes the following form:

```
def ic_vec(u, I, x):
    u[:] = I(x)  # works for scalar I too...
    return u

def scheme_rhs_vec(b, up, f, bc, x, t, C2, dt, t_old, theta):
    n = len(x)-1
    i = 0;  b[i] = bc(x[i], t)
    b[1:n] = up[1:n] + C2*(up[0:n-1] - 2*up[1:n] + up[2:n+1]) + \
             theta*dt*f(x[1:n], t) + (1-theta)*dt*f(x[1:n], t_old)
    i = 0;  b[i] = bc(x[i], t)
    return b
```

Next, we will add support for Fortran 77 and C/C++ implementation of loops.

Migrating Loops To Compiled Code. Another approach for optimizing the slow Python loops is by migrating the loops to compiled code. To this end, there are several tools available; however, we will here concentrate on Weave (a subpackage of Scipy) and F2PY. We will first look at Weave, which allows us to embed C or C++ code directly into the Python code. Below follows the code for filling in the matrix entries for a `TriDiagMatrix` object using `weave.inline`:

```
def scheme_tridiag_weave(A, C1):
    n = A.n-1;  dl = A.dl;  d = A.d;  du = A.du
    code = """
int i;
i = 0;  d(i) = 1;
for (i=1; i<=n-1; i++) {
  dl(i-1) = -C1;
  d(i) = 1 + 2*C1;
  du(i) = -C1;
}
i = n;  d(i) = 1;
"""
```

```
    args = ['dl', 'd', 'du', 'n', 'C1']
    weave.inline(code, args,
                 type_converters=weave.converters.blitz,
                 compiler='gcc')
    return A
```

The C++ code is more or less a wrapping of the corresponding Python implementation with scalar loops except that we must work directly on the underlying data structure of the matrix instance. The first time this function is called, it will take a long while doing some magic behind the scene, but the next time it is called, it will run immediately. We refer to the Weave documentation [14] for more information on this.

We also need to define similar functions for the other matrix formats. These functions follows closely the code in the `scheme_tridiag_weave` function except for the sparse matrix format. Since the underlying data structure in class `SparseMatrix` is based on PySparse, there is no easy way to fill in the matrix entries using Weave. Remember that in PySparse one normally first creates an `ll_mat` object, manipulates it, and then convert it to either `csr_mat` or `sss_mat` before doing operations on it. The `ll_mat` object can not be manipulated in Weave so we have extended the PySparse package with functionality for creating `csr_mat` objects directly by providing the three arrays needed in the CSR-format. That is, an array holding the nonzero values (`nonzeros`), an array holding the column index for each stored entry (`jcol`), and an array holding references to the first stored value for each row (`irow`). By using this extension, the Weave function for a sparse matrix takes the following form

```
def scheme_sparse_weave(nonzeros, irow, jcol, C1):
    n = len(irow)-2;  nnz = 3*n-2
    code = """
int i, entry;
entry = -1;
i = 0;
irow(i) = entry+1;
nonzeros(++entry) = 1;  jcol(entry) = i;
for (i=1; i<=n-1; i++) {
  irow(i) = entry+1;
  nonzeros(++entry) = -C1;       jcol(entry) = i-1;
  nonzeros(++entry) = 1 + 2*C1;  jcol(entry) = i;
  nonzeros(++entry) = -C1;       jcol(entry) = i+1;
}
i = n;
irow(i) = entry+1;
nonzeros(++entry) = 1;  jcol(entry) = i;
irow(n+1) = nnz+1;
"""
    args = ['nonzeros', 'irow', 'jcol', 'n', 'nnz', 'C1']
    err = weave.inline(code, args,
                       type_converters=weave.converters.blitz,
                       compiler='gcc')
    return nonzeros, irow, jcol
```

Calling `scheme_sparse_weave` also needs special treatment:

```
func = 'scheme_'+matrix_format+'_'+implementation['scheme']
if func == 'scheme_sparse_weave':
    nnz = 3*n-2
    irow = zeros(n+2, int)
    jcol = zeros(nnz+1, int)
    nonzeros = zeros(nnz+1, float)
    nonzeros, irow, jcol = \
              eval(func)(nonzeros, irow, jcol, C1)
    A.m = spmatrix.csr_mat(n+1, n+1, nonzeros, jcol, irow)
elif ...
```

When factorizing the matrix we must call `superlu.factorize` (from PySparse) explicitly and set the variable `factorized` in the matrix instance to `True`:

```
A.lu = superlu.factorize(A.m)
A.factorized = True
```

Before we move over to Fortran and the F2PY tool, we also list here the Weave based function for updating the right-hand side vector:

```
def scheme_rhs_weave(b, up, f, bc, x, t, C2, dt, t_old, theta):
    n = len(x)-1
    extra_code = f.C_code('_f', inline=True) + \
                 bc.C_code('_bc', inline=True)

    code = """
int i;
i = 0;  b(i) = _bc(x(i), t);
for (i=1; i<=n-1; i++) {
  b(i) = up(i) + C2*(up(i-1) - 2*up(i) + up(i+1)) + \
         theta*dt*_f(x(i), t) + (1-theta)*dt*_f(x(i), t_old);
}
i = n;  b(i) = _bc(x(i), t);
"""
    args = ['b', 'up', 'n', 'x', 't', 'C2', 'dt', 't_old', 'theta']
    weave.inline(code, args,
                 type_converters=weave.converters.blitz,
                 support_code=extra_code, compiler='gcc')
    return b
```

Note that the functions `f` and `bc` for the diffusion source term and the boundary conditions, respectively, are assumed to be specified as a `StringFunction` objects (see Chapter 8.6.10 and Chapter 12.2.1 in [17] for more information on `StringFunction`). We can then extract the C code as strings from the `StringFunction` objects by using the methods `f.C_code` and `bc.C_code` and add them to the `support_code` argument in `weave.inline`.

Now, we turn our focus over to the F2PY tool. With aid from F2PY we can easily call Fortran routines from Python. First we present the F77 subroutine for inserting the matrix entries into a tridiagonal matrix:

```
      subroutine scheme_tridiag_f77(dl, d, du, n, C1)
      integer n
      real*8 dl(0:n-1), d(0:n), du(0:n-1)
      real*8 C1
Cf2py intent(in, out) dl, d, du
Cf2py intent(in) C1
Cf2py intent(hide) n

      i = 0;  d(i) = 1
      do i = 1, n-1
         dl(i-1) = -C1
         d(i) = 1 + 2*C1
         du(i) = -C1
      end do
      i = n;  d(i) = 1
      return
      end
```

Because F2PY and Fortran does not have any knowledge of a `TriDiagMatrix` object, we must work directly on the underlying data structure, in this case the three arrays for the sub-, main-, and super-diagonals in the matrix. The typical call of `scheme_tridiag_f77` goes like

```
A.dl, A.d, A.du = f77.scheme_tridiag_f77(A.dl, A.d, A.du, C1)
```

if `f77` is the name of the extension module. For the other matrix formats we define similar subroutines. The sparse matrix format follows the same ideas as in the Weave code above.

Before these Fortran routines can be used in a script we need to build an extension module with the F77 code. This is taken care of by the following function:

```
def make_f77(f, bc, I):
    code = """

<definitions of subroutines for matrices>

      subroutine scheme_rhs_f77(b, up, x, t, n, C2,
     &                          dt, t_old, theta)
      integer n
      real*8 up(0:n)
      real*8 x(0:n)
      real*8 b(0:n)
      real*8 C2, dt, t, t_old, theta
Cf2py intent(in, out) b
      real*8 f
      external f
      real*8 bc
      external bc

      i = 0;  b(i) = bc(x(i), t)
      do i = 1, n-1
         b(i) = up(i) + C2*(up(i-1) - 2*up(i) + up(i+1)) +
     &          theta*dt*f(x(i), t) +
     &          (1-theta)*dt*f(x(i), t_old)
      end do
      i = n;  b(i) = bc(x(i), t)
      return
      end

      subroutine ic_f77(u, x, n)
      integer n
      real*8 u(0:n)
      real*8 x(0:n)
Cf2py intent(in, out) u
      real*8 ic
      external ic

      do i = 0, n
         u(i) = ic(x(i))
      end do
      return
      end

%s

%s

%s

%s
""" % (f.F77_code('f'), bc.F77_code('bc'), I.F77_code('ic'), I.F77_pow())

    f = open('_tmp.f', 'w')
    f.write(code)
    f.close()

    cmd = "f2py -m f77 -c --fcompiler='Gnu' --build-dir tmp2"\
          " -DF2PY_REPORT_ON_ARRAY_COPY=1 _tmp.f"
    print cmd
    os.system('rm -rf tmp2')
    failure, output = commands.getstatusoutput(cmd)
    if failure:
        print 'unsuccessful F77 extension module compilation'
```

```
        print output
        sys.exit(1)
```

Note that we have assumed that the functions `f`, `bc`, and `I` are given as `StringFunction` objects. This allows us to extract the F77 subroutines as string expressions via the `F77_code` method in the `StringFunction` object. In this way we will avoid expensive callbacks to Python.

For a grid with $n = 40000$ points and 100 time steps, the code with Weave implementation of loops required 0.67 s, that is, about 102 times faster than the scalar Python loops, but only 1.6 times faster than the vectorized loops. The solver with F77 implementation of loops ran at 0.77 s, slightly slower than the Weave code.

Putting It All Together. An extension of the `solver0` function with the improvements described above is listed below.

```
def solver(I, f, k, bc, L, n, dt, tstop, theta,
           user_action=None,
           implementation={'ic': 'vec',  # or 'scalar', 'f77', 'weave'
                           'scheme': 'vec'},
           matrix_format='symmtridiag'):
    dx = L/float(n)
    x = sequence(0, L, dx)  # grid points in x dir

    gamma = dt*k/dx**2
    C1 = theta*gamma;  C2 = (1-theta)*gamma  # help variables

    u  = zeros(n+1, float)  # solution array

    # use scalar implementation mode if no info from user:
    if 'ic' not in implementation:
        implementation['ic'] = 'scalar'
    if 'scheme' not in implementation:
        implementation['scheme'] = 'scalar'

    if 'weave' in implementation.itervalues() or \
       'f77' in implementation.itervalues():
        # we avoid callback to Python and require f, bc, and I to be
        # string formulas:
        print f, bc, I
        if not isinstance(f,  StringFunction) or \
           not isinstance(bc, StringFunction) or \
           not isinstance(I,  StringFunction):
            raise TypeError, \
                  'with Weave or F77, f, bc, and I must be StringFunction'

    if 'f77' in implementation.itervalues():
        make_f77(f, bc, I) # build F77 module
        import f77
        # unified names with py versions:
        ic_f77 = f77.ic_f77
        scheme_dense_f77 = f77.scheme_dense_f77
        scheme_band_f77 = f77.scheme_band_f77
        scheme_symmband_f77 = f77.scheme_symmband_f77
        scheme_tridiag_f77 = f77.scheme_tridiag_f77
        scheme_symmtridiag_f77 = f77.scheme_symmtridiag_f77
        scheme_structsparse_f77 = f77.scheme_structsparse_f77
        scheme_sparse_f77 = f77.scheme_sparse_f77
        scheme_rhs_f77 = f77.scheme_rhs_f77

    # set initial condition:
    t0 = time.clock()
    t = 0.0
    print '***', implementation['ic']
    func = 'ic_'+implementation['ic']
    if func == 'ic_f77':
```

```
    u = eval(func)(u, x)
else:
    u = eval(func)(u, I, x)
t_ic = time.clock() - t0
up = u.copy()  # solution at previous time step

if user_action is not None:
    user_action(u, x, t)  # allow user to plot etc.

# allocate storage for coefficient matrix and rhs:
if matrix_format == 'dense':
    A = DenseMatrix(n+1,n+1)
elif matrix_format == 'band':
    A = BandMatrix(n+1, n+1, 1, 1)
elif matrix_format == 'symmband':
    A = SymmBandMatrix(n+1, 1)
elif matrix_format == 'tridiag':
    A = TriDiagMatrix(n+1)
elif matrix_format == 'symmtridiag':
    A = SymmTriDiagMatrix(n+1)
elif matrix_format == 'structsparse':
    offset = array([-1,0,1], int)
    A = StructSparseMatrix(n+1, 3, offset)
elif matrix_format == 'sparse':
    A = SparseMatrix(n+1, n+1)
else:
    print "unknown matrix format", matrix_format
b = zeros(n+1, float)

t_scheme = 0                    # CPU time scheme
t_solve = 0                     # CPU time solve linear system

# compute coefficient matrix:
t0 = time.clock()
func = 'scheme_'+matrix_format+'_'+implementation['scheme']
if func == 'scheme_sparse_weave' or func == 'scheme_sparse_f77':
    nnz = 3*n-2
    irow = zeros(n+2, int)
    jcol = zeros(nnz+1, int)
    nonzeros = zeros(nnz+1, float)
    nonzeros, irow, jcol = \
              eval(func)(nonzeros, irow, jcol, C1)
    A.m = spmatrix.csr_mat(n+1, n+1, nonzeros, jcol, irow)
elif 'f77' in func:
    if matrix_format == 'tridiag':
        A.dl, A.d, A.du = eval(func)(A.dl, A.d, A.du, C1)
    elif matrix_format == 'symmtridiag':
        A.d, A.dl = eval(func)(A.d, A.dl, C1)
    else:
        A.m = eval(func)(A.m, C1)
else:
    A = eval(func)(A, C1)
t_scheme += time.clock() - t0

while t <= tstop:
    t_old = t;  t += dt

    # update right-hand side:
    t0 = time.clock()
    func = 'scheme_rhs_'+implementation['scheme']
    if func == 'scheme_rhs_f77':
        b = eval(func)(b, up, x, t, C2, dt, t_old, theta)
    else:
        b = eval(func)(b, up, f, bc, x, t, C2, dt, t_old, theta)
    t_scheme += time.clock() - t0

    # solve linear system:
    t0 = time.clock()
    if isinstance(A, SparseMatrix) and not A.factorized and \
```

```
            implementation['scheme'] in ('weave', 'f77'):
            A.lu = superlu.factorize(A.m)
            A.factorized = True
        u = A**(-1)*b
        t_solve += time.clock() - t0

        if user_action is not None:
            user_action(u, x, t)

        # update data structures for next step:
        up, u = u, up

    return t_ic, t_scheme, t_solve
```

This `solver` function is located in the file `pypdelib/examples/diffusion1d/diffusion1d.py`.

Efficiency Comparison of Matrix Formats. We have seen that the Weave based solver results in the fastest CPU time with a slight advantage over the F77 based solver. We will therefore let Weave handle the loops when testing and comparing the efficiency of the different matrix formats. To this end, we have created a function `benchmark_matrices` in `diffusion1d.py`. Running the test with $n = 400000$ grid points over 100 time steps on a IBM Thinkpad X41 running Ubuntu Linux with 1.5 GB of memory and the CPU frequency fixed at 1.5 GHz, results in the following table:

matrix format	time
`SymmTriDiagMatrix`	1.0
`StructSparseMatrix`	1.11
`TriDiagMatrix`	1.27
`SymmBandMatrix`	1.63
`SparseMatrix`	3.31
`BandMatrix`	4.19
`DenseMatrix`	n/a

It comes as no surprise that the fastest implementation of the current problem is by using `SymmTriDiagMatrix`, slightly ahead of `StructSparseMatrix`. The slowest is of course `DenseMatrix`, which is nearly 2000 times slower than `SymmTriDiagMatrix` for a grid with $n = 5000$ points. For $n = 400000$, the implementation with a dense matrix would have used over two hours. We should note that the times in the table above have been scaled by the CPU time of the fastest implementation.

Looking closer at the results from the test reveals that about 70% of the CPU time is used for solving the linear system for the optimal choice `SymmTriDiagMatrix`. This result is rather important, because it means that most of the work is done in the underlying Fortran routines from the LAPACK library.

One might think that the efficiency gain from migrating to compiled code was a little disappointing compared with the vectorized expressions. However, when we now move over to two-dimensional grids we may benefit significantly from migrating to compiled code.

1.9.2 2D Diffusion Equation

In two dimensions, the mathematical problem in (1.15)-(1.17) translates to the following problem:

$$\frac{\partial u}{\partial t} = k\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + f(x,y,t), \quad (x,y) \in (0,L)\times(0,L),\ t > 0, \tag{1.25}$$

$$u = g(x,y,t), \quad x = 0, L,\ y = 0, L \tag{1.26}$$

$$u(x,y,t) = I(x,y), \quad x \in [0,L]\times[0,L]. \tag{1.27}$$

We start by introducing a uniform grid on the square $\Omega = [0,L]\times[0,L]$, with grid points (x_i, y_j), $i,j = 0,\ldots,m$. The grid increment $h = \Delta x = \Delta y$ then becomes L/m. The problem can be discretized by a centered difference in space, combined with a θ-rule in time. The discrete problem can then be compactly written as

$$[\delta_t u]_{i,j}^{\ell-\frac{1}{2}} = \theta k[\delta_x\delta_x u + \delta_y\delta_y u]_{i,j}^{\ell} + (1-\theta)k[\delta_x\delta_x u + \delta_y\delta_y u]_{i,j}^{\ell-1} + \theta[f]_{i,j}^{\ell} + (1-\theta)[f]_{i,j}^{\ell-1}. \tag{1.28}$$

As in the one-dimensional case, we end up with an explicit scheme when $\theta = 0$ and an implicit scheme when $\theta > 0$. The choice $\theta = 1/2$ appears to be the optimal combination of stability and accuracy, both of second order. However, this means that we for each time step $t > 0$, need to solve a $n \times n$ linear system $\mathbf{Au} = \mathbf{b}$ where $n = (m+1)^2$. The matrix $\mathbf{A}$ is sparse with only five nonzero diagonals as shown in Figure 1.2. For the inner grid points, the matrix entries can be

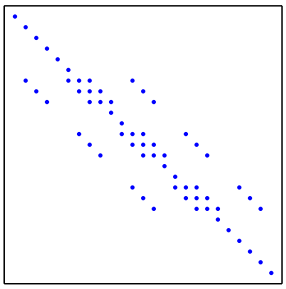


Figure 1.2: Sparsity pattern for the matrix in the linear system arising from (1.28) when $\theta > 0$ and using a 5×5 uniform grid. Each nonzero entry in the matrix are here represented by a dot.

identified as

$$\begin{aligned} A_{\text{row,row}-(m+1)} &= -\theta\gamma, \\ A_{\text{row,row}-1} &= -\theta\gamma, \\ A_{\text{row,row}} &= 1 + 4\theta\gamma, \\ A_{\text{row,row}+1} &= A_{\text{row,row}-1}, \\ A_{\text{row,row}+(m+1)} &= A_{\text{row,row}-(m+1)}, \end{aligned}$$

where $\gamma = k\Delta t/h^2$ and $\text{row} = i(m+1)+j$. Here we assume that the grid points are numbered according to a double loop over i and j, with the fastest variation over j. The right-hand side

vector is given by $\mathbf{b} = (b_{0,0}, b_{0,1}, \ldots, b_{0,m}, b_{1,0}, \ldots, b_{m,m})^T$, where the entries for the inner points are

$$b_{\text{row}} = u_{i,j}^{\ell-1} + (1-\theta)\gamma\left(u_{i,j-1}^{\ell-1} + u_{i-1,j}^{\ell-1} - 4u_{i,j}^{\ell-1} + u_{i+1,j}^{\ell-1} + u_{i,j+1}^{\ell-1}\right) + \theta\Delta t f_{i,j}^{\ell} + (1-\theta)\Delta t f_{i,j}^{\ell-1}.$$

The initial condition is handled normally, i.e. $u_{i,j}^0 = I_{i,j}$, while the setting of boundary conditions at each side of the domain requires four index sets: $i = 0$ and $j = 0, \ldots, m$; $j = 0$ and $i = 0, \ldots, m$; $i = m$ and $j = 0, \ldots, m$; and $j = m$ and $i = 0, \ldots, m$. At these four index sets we have $A_{\text{row,row}} = 1$ and $b_{\text{row}} = g_{i,j}^{\ell}$.

The extension of the 1D algorithm in the previous subsection to the 2D case is straightforward. The biggest difference is the range of the indices. The following algorithm summarizes the complete numerical method:

define $u_{i,j}$ and $u_{i,j}^-$ to represent $u_{i,j}^{\ell}$ and $u_{i,j}^{\ell-1}$ respectively
SET THE INITIAL CONDITION:
$u_{i,j} = I(x_i, y_j)$, for $i, j = 0, \ldots, m$
$t = 0$
while time $t \leq t_{\text{stop}}$
  $t \leftarrow \Delta t$
  DEFINE SPARSE SYSTEM:
  $\mathbf{Au} = \mathbf{b}$, where
  $\mathbf{u} = (u_{0,0}, u_{0,1}, \ldots, u_{0,m}, u_{1,0}, \ldots, u_{m,m})^T$,
  $A_{\text{row,row}-(m+1)} = -\theta k\frac{\Delta t}{h^2}$,
  $A_{\text{row,row}-1} = -\theta k\frac{\Delta t}{h^2}$,
  $A_{\text{row,row}} = 1 + 4\theta\frac{\Delta t}{h^2}$,
  $A_{\text{row,row}+1} = -\theta k\frac{\Delta t}{h^2}$,
  $A_{\text{row,row}+(m+1)} = -\theta k\frac{\Delta t}{h^2}$,
  $b_{\text{row}} = u_{i,j}^- + (1-\theta)\gamma\left(u_{i,j-1}^- + u_{i-1,j}^- - 4u_{i,j}^- + u_{i+1,j}^- + u_{i,j+1}^-\right) + \theta\Delta t f_{i,j}^{\ell} + \theta\Delta t f_{i,j}^{\ell-1}$,
  for $i, j = 1, \ldots, m-1$, and
  $A_{\text{row,row}} = 1$, $b_{\text{row}} = g_{i,j}^{\ell}$ at each side of the domain
  (All other $A_{i,j}$ values are zero)
  SOLVE THE SYSTEM $\mathbf{Au} = \mathbf{b}$
  INITIALIZE FOR NEXT STEP:
  $u_{i,j}^- = u_{i,j}$, for $i, j = 0, \ldots, m$

A First Implementation in Python. A simple Python implementation following closely the 2D algorithm above can be found in `pypdelib/examples/diffusion2d/diffusion2d.py`. The function is called `solver0` and takes the following form:

```
def solver0(I, f, k, bc, L, m, dt, tstop, theta,
            user_action=None):
    h = L/float(m)
    x = sequence(0, L, h)  # grid points in x dir
    y = x.copy()           # grid points in y dir
    xv = x[:,NewAxis]      # for vectorized function evaluations
    yv = y[NewAxis,:]
```

```
gamma = k*dt/h**2
C1 = theta*gamma;  C2 = (1-theta)*gamma  # help variables

u = zeros((m+1,m+1), float)  # solution array

# set initial condition:
t = 0.0
for i in iseq(0, m):
    for j in iseq(0, m):
        u[i,j] = I(x[i], y[j])
up = u.copy()                   # solution at previous time step

if user_action is not None:
    user_action(u, xv, yv, t)  # allow user to plot etc.

while t <= tstop:
    t_old = t;  t += dt

    n = (m+1)*(m+1)
    A = DenseMatrix(n, n)       # coefficient matrix
    b = zeros(n, float)         # right-hand side vector

    i = 0  # lower boundary
    for j in iseq(0, m):
        row = i*(m+1)+j;  A[row,row] = 1;  b[row] = bc(x[i], y[j], t)
    j = 0  # left boundary
    for i in iseq(0, m):
        row = i*(m+1)+j;  A[row,row] = 1;  b[row] = bc(x[i], y[j], t)
    # inner points:
    for i in iseq(1, m-1):
        for j in iseq(1, m-1):
            row = i*(m+1)+j    # treat next row
            A[row,row-(m+1)] = -C1
            A[row,row-1] = -C1
            A[row,row] = 1 + 4*C1
            A[row,row+1] = -C1
            A[row,row+(m+1)] = -C1
            b[row] = up[i,j] + \
                     C2*(up[i,j-1] + up[i-1,j] - 4*up[i,j] + \
                         up[i+1,j] + up[i,j+1]) + \
                     theta*dt*f(x[i], y[j], t) + \
                     (1-theta)*dt*f(x[i], y[j], t_old)
    i = m  # upper boundary
    for j in iseq(0, m):
        row = i*(m+1)+j;  A[row,row] = 1;  b[row] = bc(x[i], y[j], t)
    j = m  # right boundary
    for i in iseq(0, m):
        row = i*(m+1)+j;  A[row,row] = 1;  b[row] = bc(x[i], y[j], t)

    u = reshape(u, n)          # u must be 1D vector
    # solve linear system:
    u = A**(-1)*b
    u = reshape(u, (m+1,m+1))  # switch back to two indices in u

    if user_action is not None:
        user_action(u, xv, yv, t)

    # update data structures for next step:
    up, u = u, up
```

Some points are worth noticing about the `solver0` function:

1. The coefficient matrix is stored as a `DenseMatrix` instance while the optimal choice for the current problem would be a `StructSparseMatrix` instance. To allow the use of a `StructSparseMatrix` instance instead, we only need to take out this part

   ```
   A = DenseMatrix(n, n)
   ```

and replace it with the following code segment:

```
ndiags = 5
offset = zeros(ndiags, int)
offset[0] = -(m+1)
offset[1] = -1
offset[2] = 0
offset[3] = 1
offset[4] = m+1
A = StructSparseMatrix(n, ndiags, offset)
```

The rest of the `solver0` function remains the same. This modification should reduce the CPU time significantly and the memory requirements for storing the matrix is reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(5n)$. Note that the array `offset` tells the `StructSparseMatrix` instance where the five nonzero diagonals are located relative to the main diagonal.

2. As in the implementation of the 1D solver, the nonzero entries in the coefficient matrix are independent of time and we should therefore set-up the matrix before the start of the time loop to save CPU time.

3. The double loop over the inner points in the scheme are implemented using a plain Python `for` loop. While this is fine for small grids, we should benefit greatly from vectorizing the loops or by migrating the loops to a compiled language for larger grids. This will be even more significant in 2D than in 1D.

4. The expression `u = A**(-1)*b` may lead to some overhead for large grids, because the result array needs to be allocated in each call. Instead we can use the storage that is already allocated in the array `u` by switching to the following more complex statement:

```
if not A.factorized:
    A.factorize()
u = A.solve(b, solution=u)
```

5. The numerical solution $\mathbf{u}$ is stored in a NumPy array `u` with $m+1$ rows and $m+1$ columns. Since the expression `A.solve(b, solution=u)` requires that `u` is a 1D vector of length $n = (m+1)^2$, we need to flatten the array before solving the linear system:

```
u = reshape(u, n)
u = A.solve(b, solution=u)
u = reshape(u, (m+1,m+1))
```

At the end, we switch back to two indices in `u`, making it ready for the next time step.

Computing Errors. We should now verify that the code in the `solver0` function is implemented correctly. To this end, the following analytical solution can be used:

$$u(x, y, t) = e^{-2\pi^2 t} \sin \pi x \sin \pi y$$

We let $I(x, y) = u(x, y, 0)$, $g(x, y, t) = u(x, y, t)$, and $f(x, y, t) = 0$ in the following function:

```
def test_error0():
    L = 1.0;  k = 1.0

    def exact(x, y, t):
        return exp(-2*pow(pi/L,2)*k*t)*sin(pi*x)*sin(pi*y)
```

```
def I(x, y):     return exact(x, y, 0)
def bc(x, y, t): return exact(x, y, t)
def f(x, y, t):  return 0.0

error = []
def action(u, xv, yv, t):
    e = exact(xv, yv, t) - u
    error.append((t, sqrt(dot(e.flat,e.flat))))

m = 30;  dt = 0.01;  tstop = 0.5;  theta = 0.5
solver0(I, f, k, bc, L, m, dt, tstop, theta, user_action=action)
for t, e in error:
    print 't=%10.2E  error=%10.2E' % (t, e)
```

As already mentioned, the accuracy should be of second order in both time and space when $\theta = 0.5$. We let $\Delta t = 0.01$ and use a grid with 30×30 grid points. The computed error is then around $5.5 \cdot 10^{-4}$ when $t = 0.3$. Increasing the number of grid points to 60×60 and halving Δt should reduce the error by a factor of 4. For $t = 0.3$ the error is then about $2.8 \cdot 10^{-4}$, while it should have been about $1.4 \cdot 10^{-4}$. The difference is probably due to some roundoff errors.

Visualization. Let us visualize the following plug:

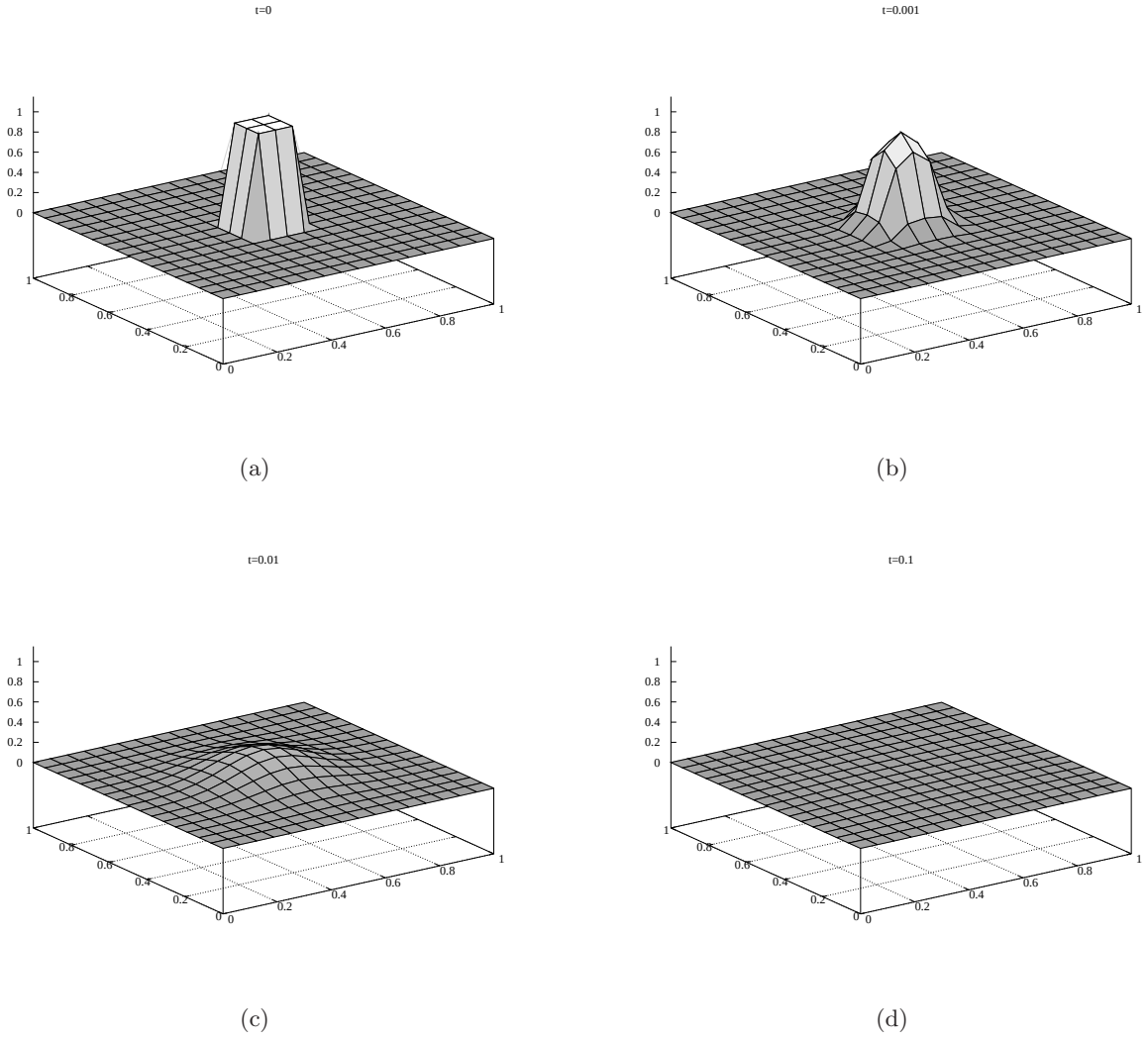
$$I(x, y) = \begin{cases} 1 & \text{if } 0.4 < x, y < 0.6 \\ 0 & \text{otherwise} \end{cases}$$

We let the boundary conditions and the source term function be given as $g(x, y, t) = 0$ and $f(x, y, t) = 0$, respectively. We can then plot the plug using the following function:

```
def test_plot0():
    L = 1.0;  k = 1.0

    def I(x, y):
        if x > 0.4 and x < 0.6 and y > 0.4 and y < 0.6:
            return 1.0
        else:
            return 0.0
    def bc(x, y, t): return 0.0
    def f(x, y, t):  return 0.0

    def action(u, xv, yv, t):
        easyviz.surf(xv, yv, u, zmin=-0.05, zmax=1.15,
                     title="t=%g" % t,
                     caxis=(0,1),
                     memoryorder='xyz')
        time.sleep(0.1) # pause between frames
        if t == 0:
            time.sleep(2)

    m = 16;  dt = 0.001;  tstop = 0.1;  theta = 0.5
    solver0(I, f, k, bc, L, m, dt, tstop, theta, user_action=action)
```

Here we use the `surf` command in Easyviz to draw an elevated surface of the solution. See Figure 1.3 for some snapshots of the plug at different time steps.

Vectorizing Loops. For small problems, the pure Python loops are fine, even in two dimensions. However, as the grid gets larger, the Python loops will be too slow. By vectorizing the loops, the CPU time should be reduced considerably. In 2D, where we have a double loop over the inner points in the finite difference scheme, vectorization is even more important than in 1D. Let us first recapitulate the double loop over the inner points in the scheme:



(a) (b)

(c) (d)

Figure 1.3: Plots of an initial plug in the middle of the domain.

```
for i in iseq(1, m-1):
    for j in iseq(1, m-1):
        row = i*(m+1)+j  # treat next row
        A[row,row-(m+1)] = -C1
        A[row,row-1] = -C1
        A[row,row] = 1 + 4*C1
        A[row,row+1] = -C1
        A[row,row+(m+1)] = -C1
        b[row] = up[i,j] + \
                 C2*(up[i,j-1] + up[i-1,j] - 4*up[i,j] + \
                     up[i+1,j] + up[i,j+1]) + \
                 theta*dt*f(x[i], y[j], t) + \
                 (1-theta)*dt*f(x[i], y[j], t_old)
```

As already mentioned, the matrix should be set-up before the start of the time loop. To this end, we can split the above code segment into two parts, i.e., a double loop over the matrix entries

```
for i in iseq(1, m-1):
    for j in iseq(1, m-1):
        row = i*(m+1)+j  # treat next row
        A[row,row-(m+1)] = -C1
        A[row,row-1] = -C1
        A[row,row] = 1 + 4*C1
        A[row,row+1] = -C1
        A[row,row+(m+1)] = -C1
```

and a double loop over the right-hand side entries

```
for i in iseq(1, m-1):
    for j in iseq(1, m-1):
        row = i*(m+1)+j  # treat next row
        b[row] = up[i,j] + \
                 C2*(up[i,j-1] + up[i-1,j] - 4*up[i,j] + \
                     up[i+1,j] + up[i,j+1]) + \
                 theta*dt*f(x[i], y[j], t) + \
                 (1-theta)*dt*f(x[i], y[j], t_old)
```

Vectorizing the latter code segment is straightforward:

```
b = reshape(b, (m+1, m+1))
b[1:m,1:m] = up[1:m,1:m] + \
             C2*(up[1:m,0:m-1] + up[0:m-1,1:m] - 4*up[1:m,1:m] + \
                 up[2:m+1,1:m] + up[1:m,2:m+1]) + \
             theta*dt*f(xv[1:m,0], yv[0,1:m], t) + \
             (1-theta)*dt*f(xv[1:m,0], yv[0,1:m], t_old)
b = reshape(b, n)
```

Note that we for convenience modify the shape of the array `b` before and after the vectorized expression. Also note that in the call to the source term function `f`, we use the arrays `xv` and `yv` rather than the arrays `x` and `y`. These are defined as

```
xv = x[:,NewAxis]
yv = y[NewAxis,:]
```

and should be used since we are working on a 2D grid.

Vectorizing the double loop over the matrix entries is a bit more complex since it depends on the underlying storage structure of the matrix. The optimal matrix format for the current problem is the structured sparse matrix format and the `StructSparseMatrix` class stores the five diagonals as columns in an $n \times 5$ NumPy array `m` with the main diagonal in the third column. First we create an index set over the inner grid points:

```
ind = seq(n-1, type=int)
ind = reshape(ind, (m+1,m+1))
i_ind = ravel(ind[1:m,1:m])
```

The vectorized expression over the inner points in the matrix then takes the following form:

```
A.m[i_ind,0] = -C1
A.m[i_ind,1] = -C1
A.m[i_ind,2] = 1 + 4*C1
A.m[i_ind,3] = -C1
A.m[i_ind,4] = -C1
```

Following the same ideas as in the 1D case, we want to support all matrix formats that are suitable for the current problem. To this end, we have created vectorized expressions for `DenseMatrix`, `BandMatrix`, `StructSparseMatrix`, and `SparseMatrix`. Only the code for a structured sparse matrix is presented here; however, the others are located in the file `diffusion2d.py` in the directory `pypdelib/examples/diffusion2d`.

Also the loops over the boundary conditions should be vectorized. The pure Python loops goes like this:

```
i = 0
for j in iseq(0, m):
    row = i*(m+1)+j;  A[row,row] = 1;  b[row] = bc(x[i], y[j], t)
j = 0
for i in iseq(0, m):
    row = i*(m+1)+j;  A[row,row] = 1;  b[row] = bc(x[i], y[j], t)
i = m
for j in iseq(0, m):
    row = i*(m+1)+j;  A[row,row] = 1;  b[row] = bc(x[i], y[j], t)
j = m
for i in iseq(0, m):
    row = i*(m+1)+j;  A[row,row] = 1;  b[row] = bc(x[i], y[j], t)
```

Splitting this into two parts, i.e., one for the matrix and one for the right-hand side vector, the vectorized expression for a `StructSparseMatrix` instance `A` then becomes

```
ind = seq(n-1, type=int)
ind = reshape(ind, (m+1,m+1))
i = 0;  ind0 = ind[i,:];    A.m[ind0,2] = 1
j = 0;  ind1 = ind[1:m,j];  A.m[ind1,2] = 1
i = m;  ind2 = ind[m,:];    A.m[ind2,2] = 1
j = m;  ind3 = ind[1:m,j];  A.m[ind3,2] = 1
```

while for the right-hand side vector `b` we get

```
b = reshape(b, (m+1, m+1))
i = 0;  b[i,:] = bc(x[i], y[:], t)
j = 0;  b[:,j] = bc(x[:], y[j], t)
i = m;  b[i,:] = bc(x[i], y[:], t)
j = m;  b[:,j] = bc(x[:], y[j], t)
b = reshape(b, n)
```

For completeness we also list here the vectorized expression for the initial condition:

```
u[:,:] = I(xv, yv)
```

Also here we use the 2D arrays `xv` and `yv`.

For a 600×600 grid with 20 time steps, the solver using only scalar Python loops required 144 s while the solver based on vectorized loops ran at 2.6 s. This gives us a speed up by a factor of 55, which is a very nice performance boost.

Migrating Loops to Compiled Code. While the speed up by vectorization was very good, we should be able to reduce the CPU time even further by migrate the loops to compiled code. Following the same ideas as in the solver for the 1D diffusion equation, we have implemented all the loops in both C++ and F77 versions using the tools Weave and F2PY, respectively. The Weave based code is very similar to the one presented in the previous section and the code is therefore not listed here. Also the F77 based code is very similar, however, one should note that multi-dimensional arrays are stored differently in C and Fortran. A two-dimensional array in C is stored row by row, while in Fortran it is stored column by column. This means that the first index runs faster than the second index in Fortran, while in C the second index is the fastest. This affects the double loop over the entries in the right-hand side vector. In Python, the double loop over i and j, has the fastest variation over j, i.e., the innermost loop runs over j:

```
for i in iseq(1, m-1):
    for j in iseq(1, m-1):
        row = i*(m+1)+j
        b[row] = up[i,j] +
```

```
                C2*(up[i,j-1] + up[i-1,j] - 4*up[i,j] + \
                    up[i+1,j] + up[i,j+1]) + \
                theta*dt*f(x[i], y[j], t) + \
                (1-theta)*dt*f(x[i], y[j], t_old)
```

In Fortran, however, the fastest variation is over i, i.e., the innermost loop runs over i:

```
 do j = 1, m-1
    do i = 1, m-1
       row = j*(m+1)+i
       b(row) = up(i,j) +
&               C2*(up(i,j-1) + up(i-1,j) - 4*up(i,j) +
&                   up(i+1,j) + up(i,j+1)) +
&               theta*dt*f(x(i), y(j), t) +
&               (1-theta)*dt*f(x(i), y(j), t_old)
    end do
 end do
```

Note that the single index `row` for the right-hand side vector in the F77 code is $j(m+1)+i$ rather than $i(m+1)+j$) as in the Python code.

Another important aspect regarding the different storage structure for multi-dimensional arrays in C and Fortran, is array copying. When a 2D NumPy array is created in Python, the default is to use row major storage (as in C). When this array is sent to Fortran with the aid of F2PY, the array will be copied to a new array with column major storage. For large arrays, this copying results in some overhead. If the array already has column major storage, no copying will be made. Therefore, to avoid overhead, we should ensure that the storage structure of the arrays `up` and `u` are compatible with Fortran storage:

```
u = numpy.asarray(u, order='FORTRAN')
up = numpy.asarray(up, order='FORTRAN')
```

This is actually not necessary since we switch references (`u,up=up,u`) in the time loop and after a few time steps, both arrays will be brought to column major storage.

Compiling the module with the flag `-DF2PY_REPORT_ON_ARRAY_COPY=1`, shows that an array is copied at every time step. After some testing, we see that the array `u` is turned back to row major storage by the `reshape` function, which is used before and after solving the linear system in the time loop:

```
u = reshape(u, n)
u = A.solve(b, u)
u = reshape(u, (m+1,m+1))
```

To avoid this overhead, we should explicitly set the storage to column major storage in the `reshape` function:

```
u = reshape(u, n, order='FORTRAN')
u = A.solve(b, u)
u = reshape(u, (m+1,m+1), order='FORTRAN')
```

Now there are no longer reports of arrays being copied.

For the 600×600 grid over 20 time steps, the solver with loops implemented in Weave required 1.2 s, that is, about 120 times faster than the solver with pure Python loops and 2.2 times faster than the vectorized version of the solver. The solver based on F77 implementation of loops is again slightly solver, requiring 1.3 s for the same grid over the same time span. These results shows that it really pays off to migrate the loops to compiled code.

Efficiency Comparison of Matrix Formats. An extension of the function `solver0`, called `solver`, is available in the file `pypdelib/examples/diffusion2d/diffusion2d.py`. The `solver` function supports all the different implementations of loops described in the past two paragraphs, which can be specified by the `implementation` keyword argument. In addition, it supports several different matrix formats, including dense, band, structured sparse, and general sparse. The matrix format can be set with the `matrix_format` keyword argument in the `solver` function using one of the following strings: `'dense'`, `'band'`, `'structsparse'`, or `'sparse'`.

Comparing the matrix formats is a bit hard since they have totally different memory requirements. For instance, the use of a `DenseMatrix` instance makes the computer (with 1.5 GB memory) start swapping to the hard drive for grids larger than 80×80. For such small grids, the CPU time using a `StructSparseMatrix` is only 0.02 s, which is too small to be reliable as a measurement. Increasing the grid to 250×250 results in the following table[3]:

matrix format	time
`StructSparseMatrix`	1.0
`SparseMatrix`	11.0
`BandMatrix`	96.0
`DenseMatrix`	n/a

No results are available for a `DenseMatrix` instance, however, we can estimate the CPU time to be more than two hours, i.e., if we had enough memory. More important is the fact that about 60% of the CPU time is used for solving the linear system $\mathbf{Au} = \mathbf{b}$ in the optimal solver, i.e., using a structured sparse matrix combined with loops running in C++ with aid from Weave. This means that most of the CPU time is spent in the Fortran subroutines `dfactRILU` and `dforwBackRILU` from the `sspmatrix` module.

1.9.3 ADI Method for 2D Diffusion Equation

The finite difference scheme presented in the previous section is quite flexible, but not an optimal solution procedure for the two-dimensional diffusion problem. In this subsection we will use the alternating direction implicit (ADI) method [13, 7] as an alternative for solving the problem given in (1.25). The ADI method was first introduced by Peaceman and Rachford in 1955 and the fundamental idea is to replace a two-dimensional problem with a series of one-dimensional problems to generate a computationally efficient algorithm. For linear problems, the ADI method is unconditionally stable and the accuracy is of second order in both space and time.

First, we introduce a rectangular grid on $\Omega = [0, L_x] \times [0, L_y]$, with grid points (x_i, y_j), where

$$x_i = i\Delta x \quad \text{and} \quad y_j = j\Delta y, \quad \text{for } i = 0, \ldots, n_x,\ j = 0, \ldots, n_y.$$

The grid increments are assumed to be constant, $\Delta x = L_x/n_x$ in the x-direction and $\Delta y = L_y/n_y$ in the y-direction. We then divide each time step into two steps of size $\Delta t/2$ and in each substep the solution is computed in one dimension:

$$\frac{u_{i,j}^{\ell+\frac{1}{2}} - u_{i,j}^{\ell}}{\Delta t/2} = k\frac{u_{i-1,j}^{\ell+\frac{1}{2}} - 2u_{i,j}^{\ell+\frac{1}{2}} + u_{i+1,j}^{\ell+\frac{1}{2}}}{\Delta x^2} + k\frac{u_{i,j-1}^{\ell} - 2u_{i,j}^{\ell} + u_{i,j+1}^{\ell}}{\Delta y^2} + f_{i,j}^{\ell+\frac{1}{2}}, \tag{1.29}$$

[3]The solver with a `BandMatrix` instance starts to swap for larger grids.

respectively

$$\frac{u_{i,j}^{\ell+1} - u_{i,j}^{\ell+\frac{1}{2}}}{\Delta t/2} = k\frac{u_{i-1,j}^{\ell+\frac{1}{2}} - 2u_{i,j}^{\ell+\frac{1}{2}} + u_{i+1,j}^{\ell+\frac{1}{2}}}{\Delta x^2} + k\frac{u_{i,j-1}^{\ell+1} - 2u_{i,j}^{\ell+1} + u_{i,j+1}^{\ell+1}}{\Delta y^2} + f_{i,j}^{\ell+\frac{1}{2}} \tag{1.30}$$

where first diffusion along the i dimension is solved implicitly and then over the j dimension. Note that the intermediate value $u_{i,j}^{\ell+1/2}$ has no physical significance. By rewriting (1.29) and (1.30) we see that we deal with two linear tridiagonal systems, which we can solve very efficiently:

$$-\gamma_x u_{i-1,j}^{\ell+\frac{1}{2}} + 2\left(1+\gamma_x\right) u_{i,j}^{\ell+\frac{1}{2}} - \gamma_x u_{i+1,j}^{\ell+\frac{1}{2}} = 2u_{i,j}^{\ell} + \gamma_y\left(u_{i,j-1}^{\ell} - 2u_{i,j}^{\ell} + u_{i,j+1}^{\ell}\right) + \Delta t f_{i,j}^{\ell+\frac{1}{2}} \tag{1.31}$$

and

$$-\gamma_y u_{i,j-1}^{\ell+1} + 2\left(1+\gamma_y\right) u_{i,j}^{\ell+1} - \gamma_y u_{i,j+1}^{\ell+1} = 2u_{i,j}^{\ell+\frac{1}{2}} + \gamma_x\left(u_{i-1,j}^{\ell+\frac{1}{2}} - 2u_{i,j}^{\ell+\frac{1}{2}} + u_{i+1,j}^{\ell+\frac{1}{2}}\right) + \Delta t f_{i,j}^{\ell+\frac{1}{2}}, \tag{1.32}$$

where $\gamma_x = k\Delta t/\Delta x^2$ and $\gamma_y = k\Delta t/\Delta y^2$. By writing (1.31) on matrix form, one obtains, for the j-th column in the grid, the tridiagonal system

$$\begin{bmatrix} 1 & & & & & \\ -\gamma_x & 2+2\gamma_x & -\gamma_x & & & \\ & -\gamma_x & 2+2\gamma_x & -\gamma_x & & \\ & & \ddots & \ddots & \ddots & \\ & & & -\gamma_x & 2+2\gamma_x & -\gamma_x \\ & & & & & 1 \end{bmatrix} \begin{bmatrix} u_{0,j}^{\ell+\frac{1}{2}} \\ u_{1,j}^{\ell+\frac{1}{2}} \\ u_{2,j}^{\ell+\frac{1}{2}} \\ \vdots \\ u_{n_x-1,j}^{\ell+\frac{1}{2}} \\ u_{n_x,j}^{\ell+\frac{1}{2}} \end{bmatrix} = \begin{bmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ \vdots \\ b_{n_x-1,j} \\ b_{n_x,j} \end{bmatrix}, \tag{1.33}$$

where

$$\begin{aligned} b_{0,j} &= g_{0,j}^{\ell+\frac{1}{2}}, \\ b_{i,j} &= 2u_{i,j}^{\ell} + \gamma_y\left(u_{i,j-1}^{\ell} - 2u_{i,j}^{\ell} + u_{i,j+1}^{\ell}\right) + \Delta t f_{i,j}^{\ell+\frac{1}{2}}, \\ &\qquad \text{for } i = 1, \ldots, n_x - 1, \quad \text{and} \\ b_{n_x,j} &= g_{n_x,j}^{\ell+\frac{1}{2}}. \end{aligned}$$

This can be abbreviated as $\mathbf{A}\mathbf{u}^{\ell+\frac{1}{2}} = \mathbf{b}$ and constitutes the *x-direction sweep*. We can also express (1.32) on matrix form. For the i-th row, we get

$$\begin{bmatrix} 1 & & & & & \\ -\gamma_y & 2+2\gamma_y & -\gamma_y & & & \\ & -\gamma_y & 2+2\gamma_y & -\gamma_y & & \\ & & \ddots & \ddots & \ddots & \\ & & & -\gamma_y & 2+2\gamma_y & -\gamma_y \\ & & & & & 1 \end{bmatrix} \begin{bmatrix} u_{i,0}^{\ell+1} \\ u_{i,1}^{\ell+1} \\ u_{i,2}^{\ell+1} \\ \vdots \\ u_{i,n_y-1}^{\ell+1} \\ u_{i,n_y}^{\ell+1} \end{bmatrix} = \begin{bmatrix} c_{i,0} \\ c_{i,1} \\ c_{i,2} \\ \vdots \\ c_{i,n_y-1} \\ c_{i,n_y} \end{bmatrix}, \tag{1.34}$$

where

$$
\begin{aligned}
c_{i,0} &= g_{i,0}^{\ell+1}, \\
c_{i,j} &= 2u_{i,j}^{\ell+\frac{1}{2}} + \gamma_x \left(u_{i-1,j}^{\ell+\frac{1}{2}} - 2u_{i,j}^{\ell+\frac{1}{2}} + u_{i+1,j}^{\ell+\frac{1}{2}} \right) + \Delta t f_{i,j}^{\ell+\frac{1}{2}}, \\
&\quad \text{for } j = 1, \ldots, n_y - 1, \quad \text{and} \\
c_{i,n_y} &= g_{i,n_y}^{\ell+1},
\end{aligned}
$$

or $\mathbf{B}\mathbf{u}^{\ell+1} = \mathbf{c}$. This is the *y-direction sweep*. The procedure then becomes to first solve $n_y - 2$ systems of equations on the form (1.33) to obtain the values for the intermediate solution $u_{i,j}^{\ell+1/2}$. Then one solves $n_x - 2$ systems of equations on the form (1.34) to compute the values $u_{i,j}^{\ell+1}$, given the values $u_{i,j}^{\ell+1/2}$ from the intermediate time step.

In the computational algorithm, we need to have full control of the initial condition and boundary conditions. The initial condition $u(x, y, 0) = I(x, y)$ is set by $u_{i,j}^0 = I_{i,j}$, and then (1.33) and (1.34) can be used for all time levels $\ell > 0$. At the intermediate time step, the upper and lower boundary values, i.e. at $x = 0, n_x$, are incorporated directly into the matrix and the right-hand side in (1.33), while the left and right boundary values, i.e. $y = 0, n_y$ must be set explicitly: $u_{i,j}^{\ell+1/2} = g_{i,j}^{\ell+1/2}$ for $i = 0, \ldots, n_x$ and $j = 0, n_y$. However, at the complete time step, the left and right boundary values are incorporated into (1.34) and the lower and upper boundary values must be set explicitly: $u_{i,j}^{\ell+1} = g_{i,j}^{\ell+1}$ for $i = 0, n_x$ and $j = 0, \ldots, n_y$. The complete numerical algorithm follows next.

define $u_{i,j}$, $u_{i,j}^*$, and $u_{i,j}^+$ to represent $u_{i,j}^\ell$, $u_{i,j}^{\ell+\frac{1}{2}}$, and $u_{i,j}^{\ell+1}$, respectively
SET THE INITIAL CONDITION:
$u_{i,j} = I_{i,j}, \quad \text{for } i = 0, \ldots, n_x,\ j = 0, \ldots, n_y$
$t = 0$
while time $t \leq t_{\text{stop}}$
  $t \leftarrow \frac{\Delta t}{2}$
  PERFORM X-DIRECTION SWEEP:
  Equation (1.33) for $j = 1, \ldots, n_y - 1$
  UPDATE LEFT AND RIGHT BOUNDARY VALUES:
  $u_{i,0}^* = g_{i,0}^{\ell+1/2}$ and $u_{i,n_y}^* = g_{i,n_y}^{\ell+1/2}$ for $i = 0, \ldots, n_x$
  $t \leftarrow \frac{\Delta t}{2}$
  PERFORM Y-DIRECTION SWEEP:
  Equation (1.34) for $i = 1, \ldots, n_x - 1$
  UPDATE LOWER AND UPPER BOUNDARY VALUES:
  $u_{0,j}^+ = g_{0,j}^{\ell+1}$ and $u_{n_x,j}^+ = g_{n_x,j}^{\ell+1}$ for $j = 0, \ldots, n_y$
  INITIALIZE FOR NEXT STEP:
  $u_{i,j} = u_{i,j}^+, \quad \text{for } i = 0, \ldots, n_x,\ j = 0, \ldots, n_y$

A First Implementation In Python. The algorithm presented above can straightforwardly be implemented in a Python function:

```
def solver0(I, f, k, bc, Lx, Ly, nx, ny, dt, tstop,
            user_action=None):
    dx = Lx/float(nx)
```

```
dy = Ly/float(ny)
x = sequence(0, Lx, dx)        # grid points in x dir
y = sequence(0, Ly, dy)        # grid points in y dir

u = zeros((nx+1,ny+1), float)  # solution array
ui = u.copy()                  # intermediate solution at t+dt/2
up = u.copy()                  # solution at t+dt

Cx = k*dt/dx**2;  Cy = k*dt/dy**2  # help variables

# set initial condition:
t = 0.0
for i in iseq(0,nx):
    for j in iseq(0,ny):
        u[i,j] = I(x[i], y[j])

if user_action is not None:
    user_action(u, x, y, t)  # allow user to plot etc.

while t <= tstop:
    t_old = t;  t += 0.5*dt

    # x-direction sweep:
    j = 0  # left boundary
    for i in iseq(0,nx):
        ui[i,j] = bc(x[i], y[j], t)

    # solve linear tridiagonal system for all internal columns j:
    for j in iseq(1,ny-1):
        A = TriDiagMatrix(nx+1);  b = zeros(nx+1, float)

        # first treat lower boundary for column j:
        i = 0;  A[i,i] = 1;  b[i] = bc(x[i], y[j], t)

        # run through all inner points for column j:
        for i in iseq(1,nx-1):
            A[i,i-1] = -Cx
            A[i,i] = 2 + 2*Cx
            A[i,i+1] = -Cx
            b[i] = 2*u[i,j] + \
                   Cy*(u[i,j-1] - 2*u[i,j] + u[i,j+1]) + \
                   dt*f(x[i], y[j], t)

        # treat upper boundary:
        i = nx;  A[i,i] = 1;  b[i] = bc(x[i], y[j], t)

        # solve linear system:
        tmp = A**(-1)*b
        # insert solution into column j:
        for i in iseq(0,nx):
            ui[i,j] = tmp[i]

    j = ny  # right boundary
    for i in iseq(0,nx):
        ui[i,j] = bc(x[i], y[j], t)

    t_old = t;  t += 0.5*dt
    # y-direction sweep:
    i = 0  # lower boundary
    for j in iseq(0,ny):
        up[i,j] = bc(x[i], y[j], t)

    # solve linear tridiagonal system for all internal rows i:
    for i in iseq(1,nx-1):
        B = TriDiagMatrix(ny+1);  c = zeros(ny+1, float)

        # first treat left boundary for row i:
        j = 0;  B[j,j] = 1;  c[j] = bc(x[i], y[j], t)
```

```
            # then run through all inner points for row i:
            for j in iseq(1,ny-1):
                B[j,j-1] = -Cy
                B[j,j] = 2 + 2*Cy
                B[j,j+1] = -Cy
                c[j] = 2*ui[i,j] + \
                       Cx*(ui[i-1,j] - 2*ui[i,j] + ui[i+1,j]) + \
                       dt*f(x[i], y[j], t_old)

            # treat right boundary
            j = ny;  B[j,j] = 1;  c[j] = bc(x[i], y[j], t)

            # solve linear system:
            tmp = B**(-1)*c
            # insert solution into row i:
            for j in iseq(0,ny):
                up[i,j] = tmp[j]

        i = nx  # upper boundary
        for j in iseq(0,ny):
            up[i,j] = bc(x[i], y[j], t)

        if user_action is not None:
            user_action(up, x, y, t)

        # update data structures for next step:
        u, up = up, u
```

We should note that this code is written for clarity and is not optimal wrt. CPU time. The `solver0` function may be found in the file `pypdelib/examples/diffusion2d_adi.py`.

Computing Errors. For verification of the `solver0` function, we can use the `test_solver0` function from the previous subsection (with some slight modifications). As already mentioned, the ADI method is second order accurate in space and time. We use a grid with 30×30 grid points and let $\Delta t = 0.01$. When $t = 0.3$, the computed error is about $2.5 \cdot 10^{-2}$. By doubling the number of grid points and halving Δt, the error should be reduced by a factor of 4. However, the error is instead increased to $4.8 \cdot 10^{-2}$ when $t = 0.3$. The implementation of ADI methods is known to be difficult to get as accurate as in theory. We refer to [6] for more on ADI methods and accuracy.

Vectorizing Loops. Vectorizing the loops in the ADI implementation is quite easy. The vectorized set-up of the coefficient matrix in the x-direction sweep is straightforward:

```
i = 0;  A[i,i] = 1
A.dl[0:nx-1] = -Cx
A.d[1:nx] = 2 + 2*Cx
A.du[1:nx] = -Cx
i = nx;  A[i,i] = 1
```

The matrix is independent of time and this code segment should therefore be executed before the start of the time loop.

In the time loop we have the following loops for the x-direction sweep:

```
for j in iseq(1,ny-1):
    i = 0;  b[i] = bc(x[i], y[j], t)
    for i in iseq(1,nx-1):
        b[i] = 2*u[i,j] + \
               Cy*(u[i,j-1] - 2*u[i,j] + u[i,j+1]) + \
               dt*f(x[i], y[j], t)
    i = nx;  b[i] = bc(x[i], y[j], t)
```

```
        # solve linear system:
        tmp = A**(-1)*b
        # insert solution into column j:
        for i in iseq(0,nx):
            ui[i,j] = tmp[i]
```

This can be replaced by the following vectorized expression:

```
    for j in iseq(1,ny-1):
        i = 0;  b[i] = bc(xv[i,0], yv[0,j], t)
        b[1:nx] = 2*u[1:nx,j] + \
                  Cy*(u[1:nx,j-1] - 2*u[1:nx,j] + u[1:nx,j+1]) + \
                  dt*f(xv[1:nx,0], yv[0,j], t)
        i = nx;  b[i] = bc(xv[i,0], yv[0,j], t)
        # solve linear system:
        ui[:,j] = A**(-1)*b
```

Note how we insert the solution into column j in the last line.

Finally, we need to update the intermediate solution array `ui` with the boundary conditions for the left and right boundary. The scalar Python loops looks as follows

```
    j = 0;  for i in iseq(0,nx): ui[i,j] = bc(x[i], y[j], t)
    j = ny; for i in iseq(0,nx): ui[i,j] = bc(x[i], y[j], t)
```

and the vectorized expression is then:

```
    j = 0;  ui[:,j] = bc(x, y[j], t)
    j = ny; ui[:,j] = bc(x, y[j], t)
```

This concludes the x-direction sweep and the intermediate solution is now stored in `ui`. For the y-direction sweep, the vectorized expressions is very similar to the ones described above and they are therefore not listed here.

The efficiency gain from vectorizing the loops are quite good. For a 600×600 grid with 20 time steps, the ADI method with scalar Python loops required 181 s while the version with vectorized loops ran at 6.7 s. This is a factor of 27 in favor of the vectorized version.

Migrating Loops to Compiled Code. Migrating the loops in the ADI implementation to C and Fortran is straightforward, following the same ideas as in the previous two subsections. The only code segment we will look at here is the one for the inner j-columns in the x-direction sweep, that is

```
    for j in iseq(1,ny-1):
        i = 0;  b[i] = bc(x[i], y[j], t)
        for i in iseq(1,nx-1):
            b[i] = 2*u[i,j] + \
                   Cy*(u[i,j-1] - 2*u[i,j] + u[i,j+1]) + \
                   dt*f(x[i], y[j], t)
        i = nx;  b[i] = bc(x[i], y[j], t)
        # solve linear system:
        ui[:,j] = A**(-1)*b
```

Here we will let the compiled language handle the innermost loop for updating the right-hand side vector (including the lower and upper boundary conditions):

```
    for j in iseq(1,ny-1):
        func = 'scheme_rhs_xsweep_'+implementation['scheme']
        if func == 'scheme_rhs_xsweep_vec':
            b = eval(func)(b, u, f, bc, xv, yv, j, t, Cy, dt, t_old)
        elif func == 'scheme_rhs_xsweep_f77':
```

```
        b = eval(func)(b, u, x, y, j, t, Cy, dt, t_old)
    else:
        b = eval(func)(b, u, f, bc, x, y, j, t, Cy, dt, t_old)
    # solve linear system:
    ui[:,j] = A**(-1)*b
```

Notice that we pass the integer for the j-th column as a parameter to the `scheme_rhs_xsweep_*` function. The code for the inner i-rows in y-direction sweep follows the same ideas.

For the same 600×600 grid over 20 time steps, the results are 5.2 s for the ADI solver with loops migrated to C and 4.1 s for the one with the loops migrated to Fortran. When comparing to the solver with scalar Python loops, this yields a factor of about 35 in favor of the first version and a factor of about 44 in favor of the latter. The results are quite disappointing compared with the results obtained for the direct (sparse) Gaussian elimination solver in the previous subsection. As one might remember, the optimal implementation there required only 1.2 s for the same grid over the same time span, that is a factor of 3.4 compared with the fastest ADI implementation. However, by increasing the grid to 2600×2600, the factor is reduced to 2. Increasing the grid even further results in the direct solver starting to use swap space on my computer. While the memory requirement for storing the coefficient matrix in a $m \times m$ grid is $\mathcal{O}(5m^2)$ for the direct solver, it is only $\mathcal{O}(6m)$ for the ADI method. This is a big advantage for the ADI method.

The ADI implementation with the different implementation of loops is available in the function `solver` in the file `pypdelib/examples/diffusion2d/diffusion2d_adi.py`.

Optimizing the ADI Implementation. As we have seen above, the results for the ADI implementation was a little disappointing. However, there are a couple of drawbacks in the code, so we might be able optimize further:

1. We can use `SymmTriDiagMatrix` rather than `TriDiagMatrix` to store the matrices in (1.33) and (1.34) by zero out the entries in the second row in the first column and the second to last row in the last column. So, we set

   ```
   A[1,0] = A[nx-1,nx] = 0
   ```

   in the x-direction sweep and

   ```
   B[1,0] = B[ny-1,ny] = 0
   ```

   in the y-direction sweep. This should reduce the CPU time in addition to reducing the memory requirements for the matrices from $\mathcal{O}(6m)$ to $\mathcal{O}(4m)$ for a $m \times m$ grid.

2. We can migrate the complete x-direction sweep to Fortran 77. That is, we take the code segment (focusing only on F77)

   ```
   for j in iseq(1,ny-1):
       b = scheme_rhs_xsweep_f77(b, u, x, y, j, t, Cy, dt, t_old)
       # solve linear system:
       ui[:,j] = A**(-1)*b
   # insert left and right boundary values:
   ui = bc_xsweep_f77(ui, x, y, t)
   ```

   and replace it with the call

   ```
   ui = xsweep_f77(A.dl, A.d, b, u, ui, x, y, t, Cy, dt, t_old)
   ```

where the `xsweep_f77` subroutine takes the following form:

```
      subroutine xsweep_f77(dl, d, b, u, ui, x, y, t,
     &                      nx, ny, Cy, dt, t_old)
      integer nx, ny, i, j, info
      real*8 dl(0:nx), d(0:nx)
      real*8 b(0:nx)
      real*8 u(0:nx, 0:ny), ui(0:nx, 0:ny)
      real*8 x(0:nx), y(0:ny)
      real*8 Cy, dt, t, t_old
Cf2py intent(in, out) ui
      real*8 f, bc
      external f, bc

      j = 0
      do i = 0, nx
         ui(i,j) = bc(x(i), y(j),t)
      end do

      do j = 1, ny-1
         i = 0;  b(i) = bc(x(i), y(j), t)
         do i = 1, nx-1
            b(i) = 2*u(i,j) +
     &             Cy*(u(i,j-1) - 2*u(i,j) + u(i,j+1)) +
     &             dt*f(x(i), y(j), t)
         end do
         i = nx;  b(i) = bc(x(i), y(j), t)

         call dpttrs(nx, 1, d, dl, b, nx, info)
         do i = 0, nx
            ui(i,j) = b(i)
         end do
      end do

      j = ny
      do i = 0, nx
         ui(i,j) = bc(x(i), y(j),t)
      end do
      return
      end
```

Note that we call the LAPACK function `dpttrs` for solving a symmetric tridiagonal linear system. This function becomes available by using the flag `--link-lapack-opt` when compiling the extension module with F2PY. Also note that the matrix should be factorized before we call `xsweep_f77`. The y-direction sweep is treated similarly.

The switch from `TriDiagMatrix` to `SymmTriDiagMatrix` reduced the CPU time with a factor of 1.3 (from 44.6 s to 34.8 s) for the 2600×2600 grid. The factor in favor of the direct solver (22.1 s) is then reduced to 1.6. Implementing the loops as suggested in the second point has an even bigger impact on the CPU time. The clock stops at 23.4 s, that is, a reduction of the CPU time by another factor of 1.5. The total factor is nearly 1.9 compared with the original ADI implementation. Even more important, the factor in favor of the direct solver is now only 1.05 compared to the fastest ADI implementation. Adding the memory advantage of the ADI method and the fact that it is easier to implement, the ADI method can surely be recommended. Finally, we note that the optimized ADI implementation is available in the function `solverX` in the file `pypdelib/examples/diffusion2d/diffusion2d_adi.py`.

Chapter 2

Easyviz: A Matlab-like Plotting Interface

2.1 Introduction

Easyviz is a light-weight interface to various packages for scientific visualization and plotting. The Easyviz interface is written in Python with the purpose of making it very easy to visualize data in Python scripts. Both curve plots and more advanced 2D/3D visualization of scalar and vector fields are supported. The Easyviz interface was designed with three ideas in mind: 1) a simple, Matlab-like syntax; 2) a unified interface to lots of visualization engines (called backends later): Gnuplot, VTK, Matlab, Matplotlib, PyX, etc.; and 3) a minimalistic interface which offers only basic control of plots (fine-tuning is left to programming in the specific backend directly).

2.1.1 Guiding Principles

First principle. Array data can be plotted with a minimal set of keystrokes using a Matlab-like syntax. A simple

```
t = linspace(0, 3, 51)    # 51 points between 0 and 3
y = t**2*exp(-t**2)
plot(t, y)
```

plots the data in (the NumPy array) `t` versus the data in (the NumPy array) `y`. If you need legends, control of the axis, as well as additional curves, all this is obtained by the standard Matlab-style commands

```
y2 = t**4*exp(-t**2)
# pick out each 4 points and add random noise:
t3 = t[::4]
random.seed(11)
y3 = y2[::4] + random.normal(loc=0, scale=0.02, size=len(t3))

plot(t, y1, 'r-')
hold('on')
plot(t, y2, 'b-')
plot(t3, y3, 'bo')
legend('t^2*exp(-t^2)', 't^4*exp(-t^2)', 'data')
title('Simple Plot Demo')
axis([0, 3, -0.05, 0.6])
xlabel('t')
ylabel('y')
```

```
show()

hardcopy('tmp0.ps')  # this one can be included in latex
hardcopy('tmp0.png') # this one can be included in HTML
```

Easyviz also allows these additional function calls to be executed as a part of the `plot` call:

```
plot(t, y1, 'r-', t, y2, 'b-', t3, y3, 'bo',
     legend=('t^2*exp(-t^2)', 't^4*exp(-t^2)', 'data'),
     title='Simple Plot Demo',
     axis=(0, 3, -0.05, 0.6),
     xlabel='t', ylabel='y',
     hardcopy='tmp1.ps',
     show=True)

hardcopy('tmp0.png') # this one can be included in HTML
```

A scalar function $f(x, y)$ may be visualized as an elevated surface with colors using these commands:

```
x = seq(-2, 2, 0.1)      # -2 to 2 with steps of 0.1
xv, yv = meshgrid(x, x)  # define a 2D grid with points (xv,yv)
values = f(xv, yv)       # function values
surfc(xv, yv, values,
      shading='interp',
      clevels=15,
      clabels='on',
      hidden='on',
      show=True)
```

Second principle. Easyviz is just a unified interface to other plotting packages that can be called from Python. Such plotting packages are referred to as backends. Several backends are supported: Gnuplot, Matplotlib, Pmw.Blt.Graph, PyX, Matlab, VTK. In other words, scripts that use Easyviz commands only, can work with a variety of backends, depending on what you have installed on the machine in question and what quality of the plots you demand. For example, switching from Gnuplot to Matplotlib is trivial.

Scripts with Easyviz commands will most probably run anywhere since at least the Gnuplot package can always be installed right away on any platform. In practice this means that when you write a script to automate investigation of a scientific problem, you can always quickly plot your data with Easyviz (i.e., Matlab-like) commands and postpone to marry any specific plotting tool. Most likely, the choice of plotting backend can remain flexible. This will also allow old scripts to work with new fancy plotting packages in the future if Easyviz backends are written for those packages.

Third principle. The Easyviz interface is minimalistic, aimed at rapid prototyping of plots. This makes the Easyviz code easy to read and extend (e.g., with new backends). If you need more sophisticated plotting, like controlling tickmarks, inserting annotations, etc., you must grab the backend object and use the backend-specific syntax to fine-tune the plot. The idea is that you can get away with Easyviz and a plotting package-independent script "95%" of the time – only now and then there will be demand for package-dependent code for fine-tuning and customization of figures.

These three principles and the Easyviz implementation make simple things simple and unified, and complicated things are not more complicated than they would otherwise be. You can always start out with the simple commands – and jump to complicated fine-tuning only when strictly needed.

2.1.2 Controlling the Backend

The Easyviz backend can either be set in a config file (see Config File below) or by a command-line option

```
--SCITOOLS_easyviz_backend name
```

where `name` is the name of the backend: `gnuplot`, `vtk`, `matplotlib`, `blt`. Which backend you choose depends on what you have available on your computer system and what kind of plotting functionality you want.

2.1.3 Config File

Easyviz is a subpackage of SciTools, and the the SciTools configuration file, called `scitools.cfg` has a section `[easyviz]` where the `backend` in Easyviz can be set:

```
[easyviz]
backend = vtk
```

A `scitools.cfg` can be placed in the current working folder, thereby affecting plots made in this folder, or it can be located in the user's home folder, which will affect all plotting sessions for the user in question.

2.2 Tutorial

This tutorial starts with plotting a single curve with a simple `plot(x,y)` command. Then we add a legend, axis labels, a title, etc. Thereafter we show how multiple curves are plotted together. We also explain how line styles and axis range can be controlled. The next section deals with animations and making movie files. More advanced topics such as fine tuning of plots (using plotting package-specific commands) and working with Axis and Figure objects close the curve plotting part of the tutorial.

Various methods for visualization of scalar fields in 2D and 3D are treated next, before we show how 2D and 3D vector fields can be handled.

2.2.1 Plotting a Single Curve

Let us plot the curve $y = t^2 \exp(-t^2)$ for t values between 0 and 3. First we generate equally spaced coordinates for t, say 51 values (50 intervals). Then we compute the corresponding y values at these points, before we call the `plot(t,y)` command to make the curve plot. Here is the complete program:

```
from scitools.all import *

def f(t):
    return t**2*exp(-t**2)

t = linspace(0, 3, 51)    # 51 points between 0 and 3
y = zeros(len(t), 'd')    # 51 doubles ('d')
for i in xrange(len(t)):
    y[i] = f(t[i])

plot(t, y)
```

The first line imports all of SciTools and Easyviz that can be handy to have when doing scientific computations. In this program we pre-allocate the `y` array and fill it with values, element by element, in a (slow) Python loop. Alternatively, we may operate on the whole `t` at once, which yields faster and shorter code:

```
from scitools.all import *

def f(t):
    return t**2*exp(-t**2)

t = linspace(0, 3, 51)    # 51 points between 0 and 3
y = f(t)                  # compute all f values at once
plot(t, y)
```

The `f` function can also be skipped, if desired, so that we can write directly

```
y = t**2*exp(-t**2)
```

To include the plot in reports, we need a hardcopy of the figure in PostScript, PNG, or another image format. The `hardcopy` command produces files with images in various formats:

```
hardcopy('tmp1.ps')  # produce PostScript
hardcopy('tmp1.png') # produce PNG
```

The filename extension determines the format: `.ps` or `.eps` for PostScript, and `.png` for PNG. Figure 2.1 displays the resulting plot.

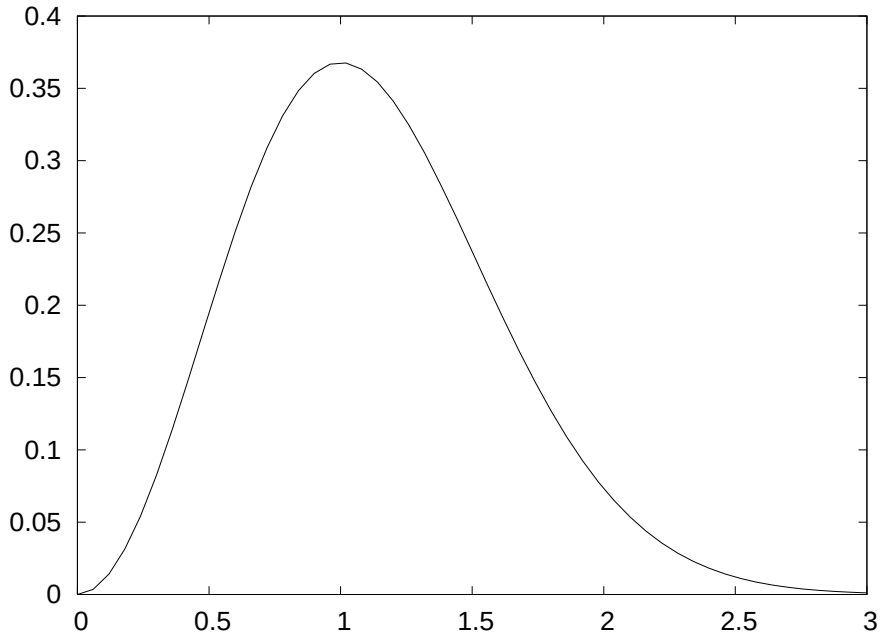


Figure 2.1: A simple plot in PostScript format.

2.2.2 Decorating the Plot

The x and y axis in curve plots should have labels, here t and y, respectively. Also, the curve should be identified with a label, or legend as it is often called. A title above the plot is also common. All such things are easily added after the `plot` command:

```
xlabel('t')
ylabel('y')
legend('t^2*exp(-t^2)')
axis([0, 3, -0.05, 0.6])   # t in [0,3], y in [-0.05,0.6]
title('My First Easyviz Demo')
```

This syntax is inspired by Matlab to make the switch between SciTools/Easyviz and Matlab almost trivial. Easyviz has also introduced a more "Pythonic" `plot` command where all the plot properties can be set at once:

```
plot(t, y,
     xlabel='t',
     ylabel='y',
     legend='t^2*exp(-t^2)',
     axis=[0, 3, -0.05, 0.6],
     title='My First Easyviz Demo',
     hardcopy='tmp1.ps',
     show=True)
```

With `show=False` one can avoid the plot window on the screen and just make the hardcopy. This feature is particularly useful if you generate a large number of plots in a loop.

Note that we in the curve legend write `t` square as `t^2` (LaTeX style) rather than `t**2` (program style). Whichever form you choose is up to you, but the LaTeX form sometimes looks better in some plotting programs (Gnuplot is one example). See Figure 2.2 for how the modified plot looks like and how `t^2` is typeset in Gnuplot.

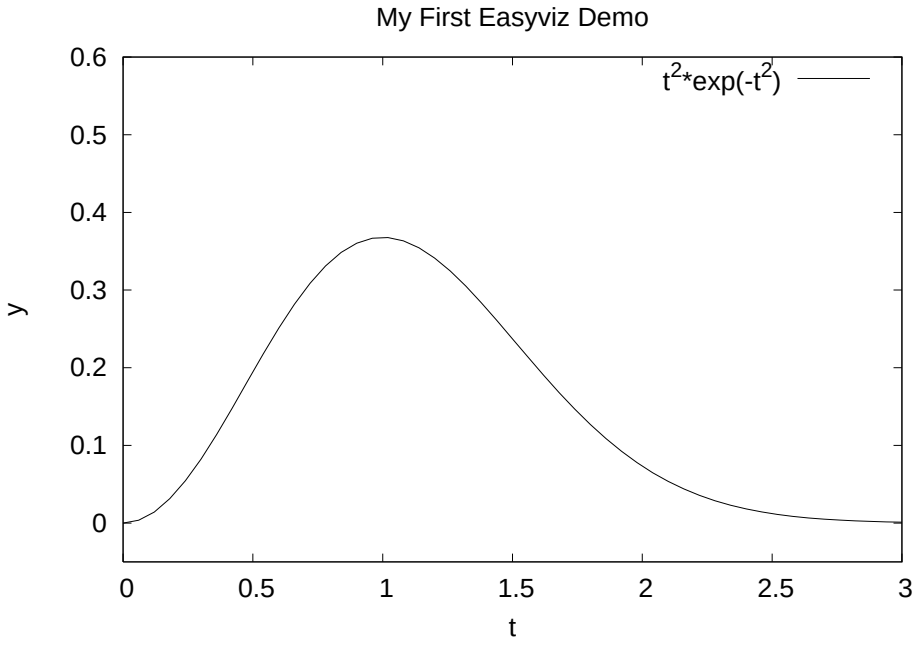


Figure 2.2: A single curve with label, title, and axis adjusted.

2.2.3 Plotting Multiple Curves

A common plotting task is to compare two or more curves, which requires multiple curves to be drawn in the same plot. Suppose we want to plot the two functions $f_1(t) = t^2 \exp(-t^2)$ and

$f_2(t) = t^4 \exp(-t^2)$. If we issue two `plot` commands after each other, two separate plots will be made. To make the second `plot` command draw the curve in the first plot, we need to issue a `hold('on')` command. Alternatively, we can provide all data in a single `plot` command. A complete program illustrates the different approaches:

```
from scitools.all import *   # for curve plotting

def f1(t):
    return t**2*exp(-t**2)

def f2(t):
    return t**2*f1(t)

t = linspace(0, 3, 51)
y1 = f1(t)
y2 = f2(t)

# Matlab-style syntax:
plot(t, y1)
hold('on')
plot(t, y2)

xlabel('t')
ylabel('y')
legend('t^2*exp(-t^2)', 't^4*exp(-t^2)')
title('Plotting two curves in the same plot')
hardcopy('tmp2.ps')

# alternative:
plot(t, y1, t, y2, xlabel='t', ylabel='y',
     legend=('t^2*exp(-t^2)', 't^4*exp(-t^2)'),
     title='Plotting two curves in the same plot',
     hardcopy='tmp2.ps')
```

The sequence of the multiple legends is such that the first legend corresponds to the first curve, the second legend to the second curve, and so on. The visual result appears in Figure 2.3.

2.2.4 Controlling Axis and Line Styles

A plotting program will normally compute sensible ranges of the axis. For example, the Gnuplot program has in our examples so far used an y axis from 0 to 0.6 while the x axis goes from 0 to 3. Sometimes it is desired to adjust the range of the axis. Say we want the x axis to go from 0 to 4 (although the data stops at $x = 3$), while y axis goes from -0.1 to 0.6. In the Matlab-like syntax new axis specifications are done by the `axis` command:

```
axis([0, 4, -0.1, 0.6])
```

With a single `plot` command we must use the `axis` keyword:

```
plot(t, y1, t, y2, ...
     axis=[0, 4, -0.1, 0.6],
     ...)
```

In both cases, the axis specification is a list of the $x_{\min}$, $x_{\max}$, $y_{\min}$, and $y_{\max}$ values.

The two curves get distinct default line styles, depending on the program that is used to produce the curve (and the settings for this program). It might well happen that you get a green and a red curve (which is bad for a significant portion of the male population). We may therefore often want to control the line style in detail. Say we want the first curve (`t` and `y1`) to be drawn as a red solid line and the second curve (`t` and `y2`) as blue circles at the discrete

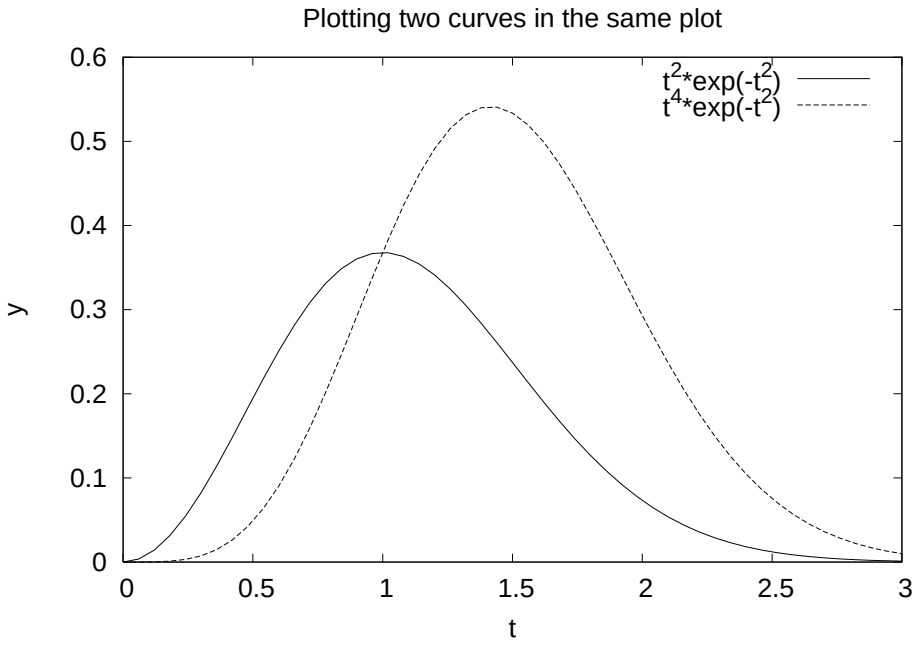


Figure 2.3: Two curves in the same plot.

data points. The Matlab-inspired syntax for specifying line types applies a letter for the color and a symbol from the keyboard for the line type. For example, `r-` represents a red (`r`) line (`-`), while `bo` means blue (`b`) circles (`o`). The line style specification is added as an argument after the x and y coordinate arrays of the curve:

```
plot(t, y1, 'r-')
hold('on')
plot(t, y2, 'bo')

# or
plot(t, y1, 'r-', t, y2, 'bo')
```

The effect of controlling the line styles can be seen in Figure 2.4.

Assume now that we want to plot the blue circles at only each 4 points. We can grab each 4 points out of the `t` array by using an appropriate slice: `t2 = t[::4]`. Note that the first colon means the range from the first to the last data point, while the second colon separates this range from the stride, i.e., how many points we should "jump over" when we pick out a set of values of the array.

In this plot we also adjust the size of the line and the circles by adding an integer: `r-6` means a red line with thickness 6 and `bo5` means red circles with size 5. The effect of the given line thickness and symbol size depends on the underlying plotting program. For the Gnuplot backend one can view the effect in Figure 2.5.

```
from scitools.all import *

def f1(t):
    return t**2*exp(-t**2)

def f2(t):
    return t**2*f1(t)
```

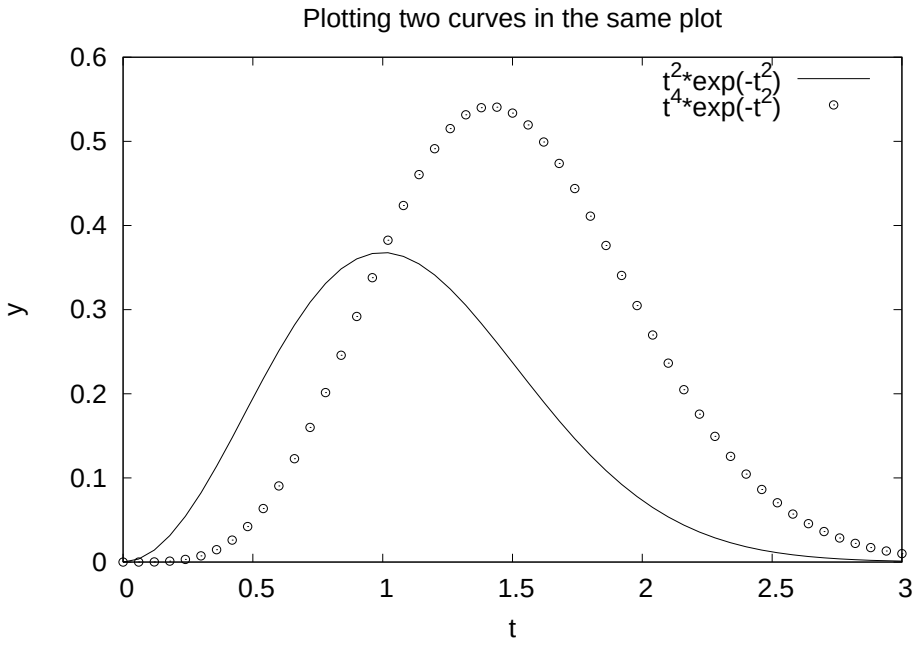


Figure 2.4: Two curves in the same plot, with controlled line styles.

```
t = linspace(0, 3, 51)
y1 = f1(t)
t2 = t[::4]
y2 = f2(t2)

plot(t, y1, 'r-6', t2, y2, 'bo3',
     xlabel='t', ylabel='y',
     axis=[0, 4, -0.1, 0.6],
     legend=('t^2*exp(-t^2)', 't^4*exp(-t^2)'),
     title='Plotting two curves in the same plot',
     hardcopy='tmp2.ps')
```

The different available line colors include

- yellow: `'y'`
- magenta: `'m'`
- cyan: `'c'`
- red: `'r'`
- green: `'g'`
- blue: `'b'`
- white: `'w'`
- black: `'k'`

The different available line types are

- solid line: `'-'`

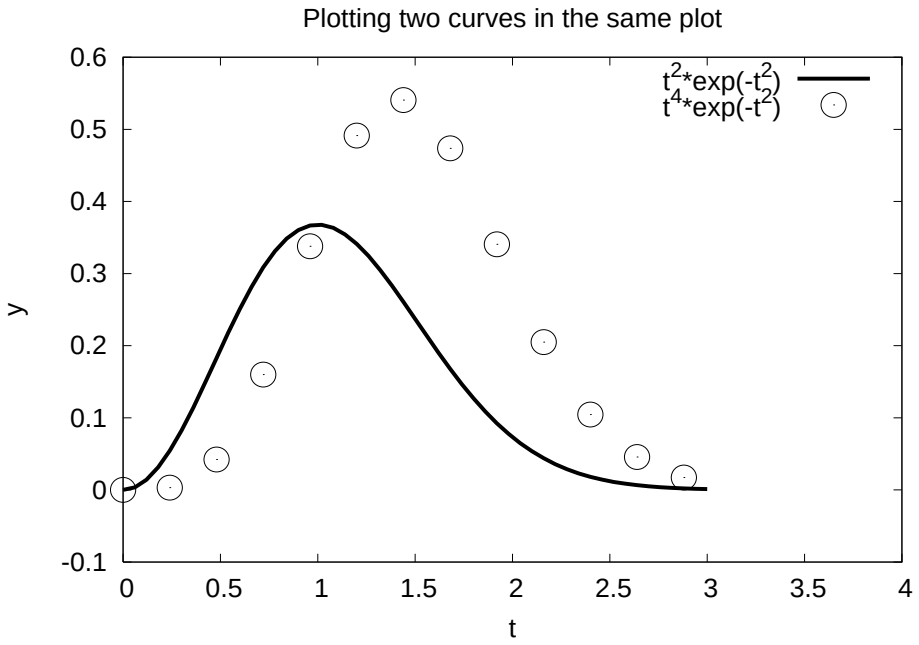


Figure 2.5: Circles at every 4 points and extended line thickness (6) and circle size (3).

- dashed line: `'--'`
- dotted line: `':'`
- dash-dot line: `'-.'`

We remark that in the Gnuplot backend all the different line types are drawn as solid lines on the screen. The hardcopy chooses automatically different line types (solid, dashed, etc.) and not in accordance with the line type specification.

Lots of markers at data points are available:

- plus sign: `'+'`
- circle: `'o'`
- asterisk: `'*'`
- point: `'.'`
- cross: `'x'`
- square: `'s'`
- diamond: `'d'`
- upward-pointing triangle: `'^'`
- downward-pointing triangle: `'v'`
- right-pointing triangle: `'>'`

- left-pointing triangle: `'<'`
- five-point star (pentagram): `'p'`
- six-point star (hexagram): `'h'`
- no marker (default): `None`

Symbols and line styles may be combined, for instance as in `'kx-'`, which means a black solid line with black crosses at the data points.

The line thickness can be added as a number in the line style specification string. For example, `'r-2'` means red solid line with thickness 2.

Another Example. Let us extend the previous example with a third curve where the data points are slightly randomly distributed around the $f_2(t)$ curve:

```
from scitools.all import *

def f1(t):
    return t**2*exp(-t**2)

def f2(t):
    return t**2*f1(t)

t = linspace(0, 3, 51)
y1 = f1(t)
y2 = f2(t)

# pick out each 4 points and add random noise:
t3 = t[::4]      # slice, stride 4
random.seed(11)  # fix random sequence
noise = random.normal(loc=0, scale=0.02, size=len(t3))
y3 = y2[::4] + noise

plot(t, y1, 'r-')
hold('on')
plot(t, y2, 'ks-')   # black solid line with squares at data points
plot(t3, y3, 'bo')

legend('t^2*exp(-t^2)', 't^4*exp(-t^2)', 'data')
title('Simple Plot Demo')
axis([0, 3, -0.05, 0.6])
xlabel('t')
ylabel('y')
show()
hardcopy('tmp3.ps')
hardcopy('tmp3.png')
```

The plot is shown in Figure 2.6.

Minimalistic Plotting. When exploring mathematics in the interactive Python shell, most of us are interested in the quickest possible commands. Here is an example on minimalistic syntax for comparing the two sample functions we have used in the previous examples:

```
t = linspace(0, 3, 51)
plot(t, t**2*exp(-t**2), t, t**4*exp(-t**2))
```

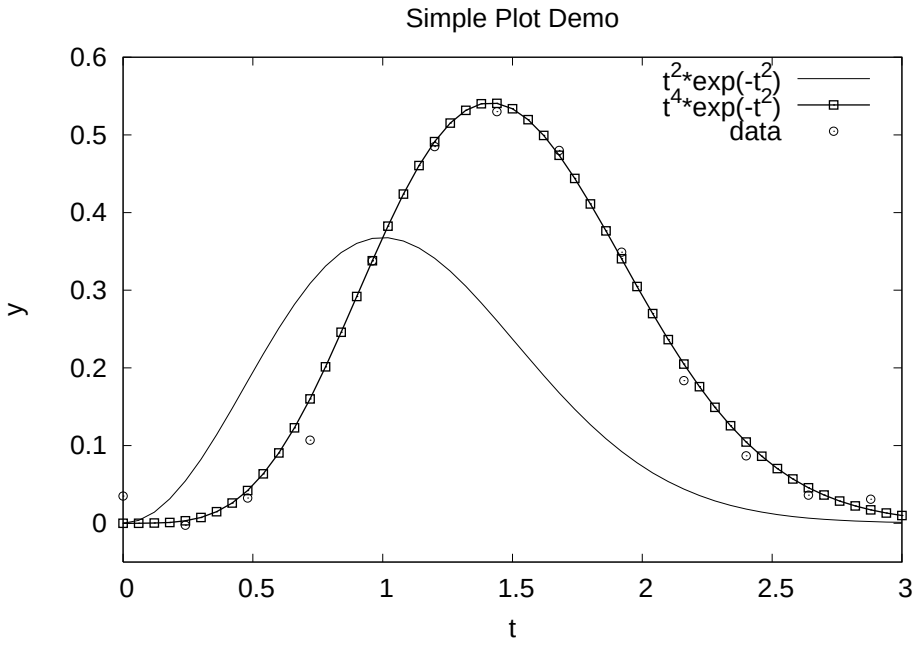


Figure 2.6: A plot with three curves.

2.2.5 Interactive Plotting Sessions

All the Easyviz commands can of course be issued in an interactive Python session. The only thing to comment is that the `plot` command returns an argument:

```
>>> t = linspace(0, 3, 51)
>>> plot(t, t**2*exp(-t**2))
[<scitools.easyviz.common.Line object at 0xb5727f6c>]
```

Most users will just ignore this output line.

All Easyviz commands that produce a plot return an object reflecting the particular type of plot. The `plot` command returns a list of `Line` objects, one for each curve in the plot. These `Line` objects can be invoked to see, for instance, the value of different parameters in the plot (`Line.get()`):

```
>>> lines = plot(x, y, 'b')
>>> pprint.pprint(lines[0].get())
{'description': '',
 'dims': (4, 1, 1),
 'legend': '',
 'linecolor': 'b',
 'pointsize': 1.0,
 ...
```

Such output is mostly of interest to advanced users.

2.2.6 Making Animations

A sequence of plots can be combined into an animation and stored in a movie file. First we need to generate a series of hardcopies, i.e., plots stored in files. Thereafter we must use a tool

to combine the individual plot files into a movie file. We shall illustrate the process with an example.

Consider the "Gaussian bell" function

$$f(x; m, s) = (2\pi)^{-1/2} s^{-1} \exp\left[-\frac{1}{2}\left(\frac{x-m}{s}\right)^2\right],$$

which is a "wide" function for large s and "peak-formed" for small s, see Figure 2.7, Our goal is to make an animation where we see how this function evolves as s is decreased. In Python we implement the formula above as a function `f(x, m, s)`.

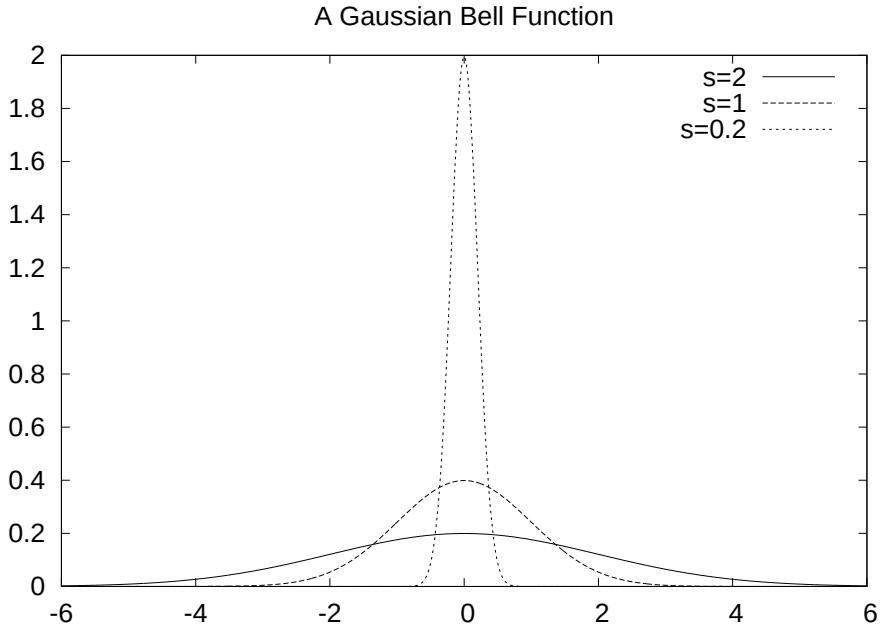


Figure 2.7: Different shapes of a Gaussian bell function.

The animation is created by varying s in a loop and for each s issue a `plot` command. A moving curve is then visible on the screen. One can also make a movie file that can be played as any other computer movie using a standard movie player. To this end, each plot is saved to a file, and all the files are combined together using some suitable tool, which is reached through the `movie` function in Easyviz. All necessary steps will be apparent in the complete program below, but before diving into the code we need to comment upon a couple of issues with setting up the `plot` command for animations.

The underlying plotting program will normally adjust the axis to the maximum and minimum values of the curve if we do not specify the axis ranges explicitly. For an animation such automatic axis adjustment is misleading – the axis ranges must be fixed to avoid a jumping axis. The relevant values for the axis range is the minimum and maximum value of f. The minimum value is zero, while the maximum value appears for $x = m$ and increases with decreasing s. The range of the y axis must therefore be $[0, f(m; m, \min s)]$.

The function f is defined for all $-\infty < x < \infty$, but the function value is very small already $3s$ away from $x = m$. We may therefore limit the x coordinates to $[m - 3s, m + 3s]$.

Now we are ready to take a look at the complete code for animating how the Gaussian bell curve evolves as the s parameter decreases from 2 to 0.2:

```
from scitools.all import *
import time

def f(x, m, s):
    return (1.0/(sqrt(2*pi)*s))*exp(-0.5*((x-m)/s)**2)

m = 0
s_start = 2
s_stop = 0.2
s_values = linspace(s_start, s_stop, 30)
x = linspace(m -3*s_start, m + 3*s_start, 1000)
# f is max for x=m; smaller s gives larger max value
max_f = f(m, m, s_stop)

# show the movie, and make hardcopies of frames simultaneously:
counter = 0
for s in s_values:
    y = f(x, m, s)
    plot(x, y, axis=[x[0], x[-1], -0.1, max_f],
         xlabel='x', ylabel='f', legend='s=%4.2f' % s,
         hardcopy='tmp_%04d.ps' % counter)
    counter += 1
    #time.sleep(0.2)  # can insert a pause to control movie speed

# make movie file:
movie('tmp_*.ps')
```

First note that the s values are decreasing (`linspace` handles this automatically if the start value is greater than the stop value). Also note that we, simply because we think it is visually more attractive, let the y axis go from -0.1 although the f function is always greater than zero.

For each frame (plot) in the movie we store the plot in a file. The different files need different names and an easy way of referring to the set of files in right order. We therefore suggest to use filenames of the form `stem0001.ext`, `stem0002.ext`, `stem0003.ext`, etc., since the expression `stem*.ext` then lists all files in the right order. In our example, `stem` is `tmp_`, and `.ext` is `.ps` (PostScript format in the hardcopy).

Having a set of `stem*.ext` files, one can simply generate a movie by a `movie('stem*.ext')` call. When a movie file is not wanted (it may take some time to generate it), one can simply skip the `hardcopy` argument and the call to `movie`.

2.2.7 Advanced Easyviz Topics

The information in the previous subsections aims at being sufficient for the daily work with plotting curves. Sometimes, however, one wants to fine-control the plot or how Easyviz behaves. First, we explain how to speed up the `from scitools.all import *` statement. Second, we show how to operate with the plotting program directly and using plotting program-specific advanced features. Third, we explain how the user can grab `Figure` and `Axis` objects that Easyviz produces "behind the curtain".

Importing Just Easyviz

The `from scitools.all import *` statement imports many modules and packages:

- Easyviz
- SciPy (if it exists)
- NumPy (if SciPy is not installed)

- the Python modules `sys`, `os`, `math`, `operator`
- the SciTools module `StringFunction` and the SciTools functions `watch` and `trace` for debugging

The `scipy` import can take some time and lead to slow start-up of plot scripts. A more minimalistic import for curve plotting is

```
from scitools.easyviz import *
from numpy import *
```

Alternatively, one can edit the `scitools.cfg` configure file or add one's own `.scitools.cfg` file with redefinition of selected options, such as `load` in the `scipy` section. The user `.scitools.cfg` must be placed in the folder where the plotting script in action resides, or in the user's home folder. Instead of editing a configuration file, one can just add the command-line argument `--SCITOOLS_scipy_load no` to the curve plotting script (all sections/options in the configuration file can also be set by such command-line arguments).

Working with the Plotting Program Directly

Easyviz supports just the most common plotting commands, typically the commands you use "95%" of the time when exploring curves. Various plotting packages have lots of additional commands for different advanced features. When Easyviz does not have a command that supports a particular feature, one can grab the Python object that communicates with the underlying plotting program and work with this object directly, using plotting program-specific command syntax. Let us illustrate this principle with an example where we add a text and an arrow in the plot, see Figure 2.8.

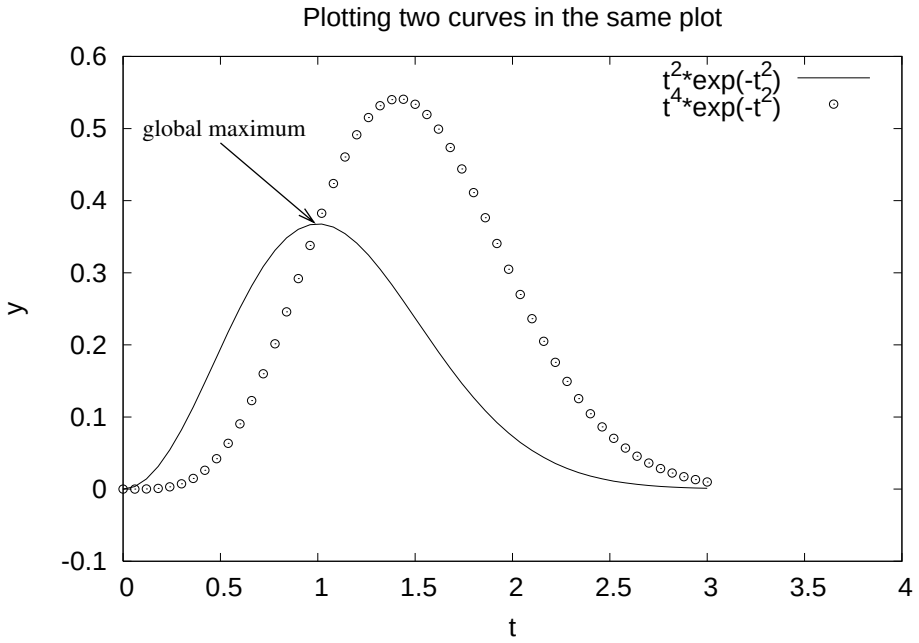


Figure 2.8: Illustration of a text and an arrow using Gnuplot-specific commands.

Easyviz does not support text and arrows at arbitrary places inside the plot, but Gnuplot does. If we use Gnuplot as backend, we may grab the `Gnuplot` object (a Python module) and issue Gnuplot commands to this object directly:

```
g = get_backend()
if g.__class__.__name__ == 'Gnuplot':
    # g is a Gnuplot object, work with Gnuplot commands directly:
    g('set label "global maximum" at 0.1,0.5 font "Times,18"')
    g('set arrow from 0.5,0.48 to 0.98,0.37 linewidth 2')
g.refresh()
g.hardcopy('tmp2.ps')  # make new hardcopy
```

We refer to the Gnuplot manual for the features of this package and the syntax of the commands. The idea is that you can quickly generate plots with Easyviz, using standard commands that are independent of the underlying plotting package. However, when you need advanced features, you must add plotting package-specific code as shown above. This principle makes Easyviz a light-weight interface, but without limiting the available functionality of various plotting programs.

Working with Axis and Figure Objects

Easyviz supports the concept of Axis objects, as in Matlab. The Axis object represent a set of axis, with curves drawn in the associated coordinate system. A figure is the complete physical plot. One may have several axis in one figure, each axis representing a subplot. One may also have several figures, represented by different windows on the screen or separate hardcopies.

Axis Objects. Users with Matlab experience may prefer to set axis labels, ranges, and the title using an Axis object instead of providing the information in separate commands or as part of a `plot` command. The `gca` (get current axis) command returns an `Axis` object, whose `set` method can be used to set axis properties:

```
plot(t, y1, 'r-', t, y2, 'bo',
     legend=('t^2*exp(-t^2)', 't^4*exp(-t^2)'),
     hardcopy='tmp2.ps')

ax = gca()   # get current Axis object
ax.set(xlabel='t', ylabel='y',
       axis=[0, 4, -0.1, 0.6],
       title='Plotting two curves in the same plot')
show()  # show the plot again after ax.set actions
```

Figure Objects. The `figure()` call makes a new figure, i.e., a new window with curve plots. Figures are numbered as 1, 2, and so on. The command `figure(3)` sets the current figure object to figure number 3.

Suppose we want to plot our `y1` and `y2` data in two separate windows. We need in this case to work with two `Figure` objects:

```
plot(t, y1, 'r-', xlabel='t', ylabel='y',
     axis=[0, 4, -0.1, 0.6])

figure()  # new figure

plot(t, y2, 'bo', xlabel='t', ylabel='y')
```

We may now go back to the first figure (with the `y1` data) and set a title and legends in this plot, show the plot, and make a PostScript version of the plot:

```
figure(1)  # go back to first figure
title('One curve')
legend('t^2*exp(-t^2)')
show()
hardcopy('tmp2_1.ps')
```

We can also adjust figure 2:

```
figure(2)  # go to second figure
title('Another curve')
hardcopy('tmp2_2.ps')
show()
```

The current `Figure` object is reached by `gcf` (get current figure), and the `dump` method dumps the internal parameters in the `Figure` object:

```
fig = gcf(); print fig.dump()
```

These parameters may be of interest for troubleshooting when Easyviz does not produce what you expect.

Let us then make a third figure with two plots, or more precisely, two axes: one with `y1` data and one with `y2` data. Easyviz has a command `subplot(r,c,a)` for creating `r` rows and `c` columns and set the current axis to axis number `a`. In the present case `subplot(2,1,1)` sets the current axis to the first set of axis in a "table" with two rows and one column. Here is the code for this third figure:

```
figure()  # new, third figure
# plot y1 and y2 as two axis in the same figure:
subplot(2, 1, 1)
plot(t, y1, xlabel='t', ylabel='y')
subplot(2, 1, 2)
plot(t, y2, xlabel='t', ylabel='y')
title('A figure with two plots')
show()
hardcopy('tmp2_3.ps')
```

We remark that the `hardcopy` command does not work with the Gnuplot backend in this case with multiple axes in a figure.

If we need to place an axis at an arbitrary position in the figure, we must use the command

```
ax = axes(viewport=[left, bottom, width, height])
```

The four parameters `left`, `bottom`, `width`, `height` are location values between 0 and 1 ((0,0) is the lower-left corner and (1,1) is the upper-right corner).

2.2.8 Visualization of Scalar Fields

A scalar field is a function from space or space-time to a real value. This real value typically reflects a scalar physical parameter at every point in space (or in space and time). One example is temperature, which is a scalar quantity defined everywhere in space and time. In a visualization context, we work with discrete scalar fields that are defined on a grid. Each point in the grid is then associated with a scalar value.

There are several ways to visualize a scalar field in Easyviz. Both two- and three-dimensional scalar fields are supported. In 2D we can create elevated surface plots, contour plots, and pseudocolor plots, while in 3D we can create isosurface plots, volumetric slice plots, and contour slice plots.

Elevated Surface Plots

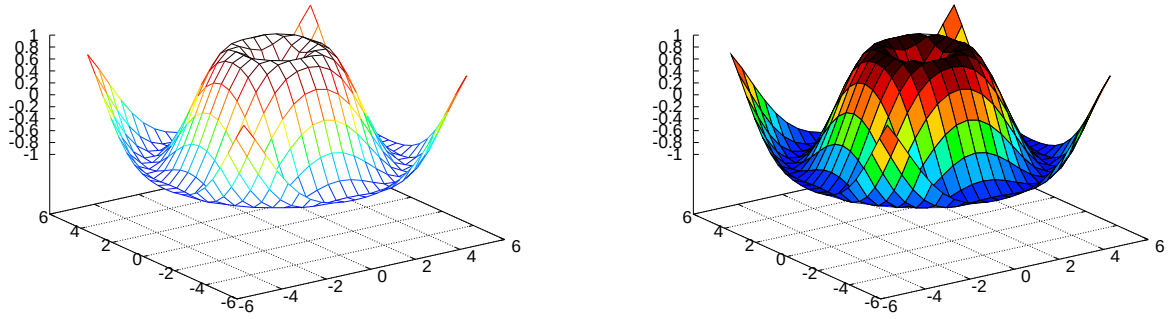
To create elevated surface plots we can use either the `surf` or the `mesh` command. Both commands have the same syntax, but the `mesh` command creates a wireframe mesh while the `surf` command creates a solid colored surface.

Our examples will make use of the scalar field $f(x, y) = \sin r$, where r is the distance in the plane from the origin, i.e., $r = \sqrt{x^2 + y^2}$. The x and y values in our 2D domain lie between -5 and 5.

The example first creates the necessary data arrays for 2D scalar field plotting: the coordinates in each direction, extensions of these arrays to form a *meshgrid*, and the function values. The latter array is computed in a vectorized operation which requires the extended coordinate arrays from the `meshgrid` function. The `mesh` command can then produce the plot with a syntax that mirrors the simplicity of the `plot` command for curves:

```
x = y = linspace(-5, 5, 21)
xv, yv = meshgrid(x, y)
values = sin(sqrt(xv**2 + yv**2))
h = mesh(xv, yv, values)
```

The `mesh` command returns a reference to a new `Surface` object, here stored in a variable h. This reference can be used to set or get properties in the object at a later stage if needed. The resulting plot can be seen in Figure 2.9(a).



(a) (b)

Figure 2.9: Results of plotting a 2D scalar field using (a) the `mesh` command and (b) the `surf` command (Gnuplot backend).

The `surf` command employs the same syntax, but results in a different plot (see Figure 2.9(b)):

```
surf(xv, yv, values)
```

There are many possibilities to adjust the resulting plot after a call to a plotting command (here `surf`):

```
set(interactive=False)
surf(xv, yv, values)
```

```
shading('flat')
colorbar()
colormap(hot())
axis([-6,6,-6,6,-1.5,1.5])
view(35,45)
show()
```

Here we have specified a flat shading model, added a color bar, changed the color map to `hot`, set some suitable axis values, and changed the view point. The same plot can also be accomplished with one single, compound statement (as Easyviz offers for the `plot` command):

```
surf(xv, yv, values,
     shading='interp',
     colorbar='on',
     colormap=jet(),
     axis=[-6,6,-6,6,-1.5,1.5],
     view=[-35,35])
```

Figure 2.10 displays the result.

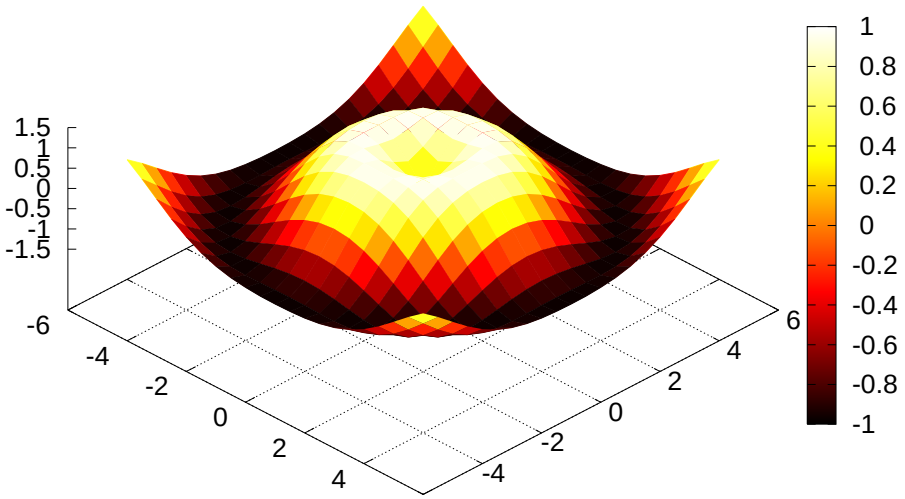


Figure 2.10: Result of an extended `surf` command (Gnuplot backend).

Contour Plots

A contour plot is another useful technique for visualizing scalar fields. The primary examples on contour plots from everyday life is the level curves on geographical maps, reflecting the height of the terrain. Mathematically, a contour line, also called isoline, is defined as the implicit curve $f(x, y) = c$. The contour levels c are normally uniformly distributed between the extreme values of the function f (this is the case in a map: the height difference between two contour lines is constant), but in scientific visualization it is sometimes useful to use a few carefully selected c values to illustrate particular features of a scalar field.

In Easyviz, there are several commands for creating different kinds of contour plots:

- `contour`: Draw a standard contour plot, i.e., lines in the plane.
- `contourf`: Draw a filled 2D contour plot, where the space between the contour lines is filled with colors.
- `contour3`: Same as `contour`, but the curves are drawn at their corresponding height levels in 3D space.
- `meshc`: Works in the same way as `mesh` except that a contour plot is drawn in the plane beneath the mesh.
- `surfc`: Same as `meshc` except that a solid surface is drawn instead of a wireframe mesh.

We start with illustrating the plain `contour` command, assuming that we already have computed the `xv`, `yv`, and `values` arrays as shown in our first example on scalar field plotting. The basic syntax follows `mesh` and `surf`:

```
contour(xv, yv, values)
```

By default, five uniformly spaced contour level curves are drawn, see Figure 2.11(a).
The number of levels in a contour plot can be specified with an additional argument:

```
n = 15   # number of desired contour levels
contour(xv, yv, values, n)
```

The result can be seen in Figure 2.11(b).

Sometimes one wants contour levels that are not equidistant or not distributed throughout the range of the scalar field. Individual contour levels to be drawn can easily be specified as a list:

```
levels = [-0.5, 0.1, 0.3, 0.9]
contour(xv, yv, values, levels, clabels='on')
```

Now, the `levels` list specify the values of the contour levels, and the `clabel` keyword allows labeling of the level values in the plot. Figure 2.11(c) shows the result. We remark that the Gnuplot backend colors the contour lines and places the contour values and corresponding colors beside the plot. Figures that are reproduced in black and white only can then be hard to analyze. Other backends may draw the contour lines in black and annotate each line with the corresponding contour level value. Such plots are better suited for being displayed in black and white.

The `contourf` command,

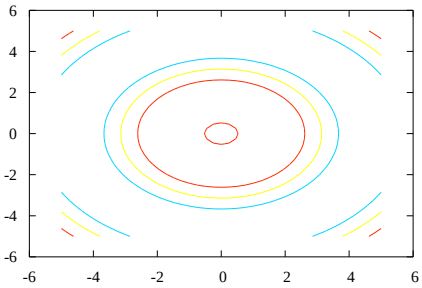
```
contourf(xv, yv, values)
```

gives a filled contour plot as shown in Figure 2.12. Only the Matplotlib and VTK backends currently supports filled contour plots.

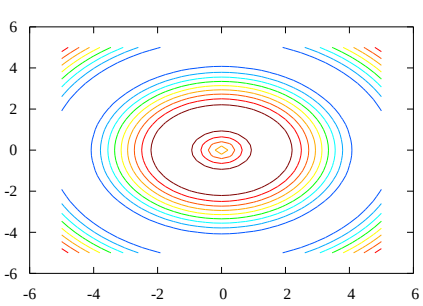
The contour lines can be "lifted up" in 3D space, as shown in Figure 2.13, using the `contour3` command:

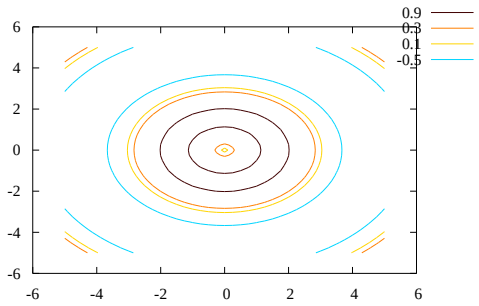
```
contour3(xv, yv, values, 15)
```

Finally, we show a simple example illustrating the `meshc` and `surfc` commands:



(a)





(b)

(c)

Figure 2.11: (a) Result of the simplest possible `contour` command (Gnuplot backend). (b) A contour plot with 15 contour levels (Gnuplot backend). (c) Four individually specified contour levels (Gnuplot backend).

```
meshc(xv, yv, values,
      clevels=10,
      colormap=hot(),
      grid='off')
figure()
surfc(xv, yv, values,
      clevels=15,
      colormap=hsv(),
      grid='off',
      view=(30,40))
```

Note that we set the number of contour levels with the `clevels` keyword. This keyword can also take a list specifying individual contour levels. The resulting plots are displayed in Figures 2.14(a) and 2.14(b).

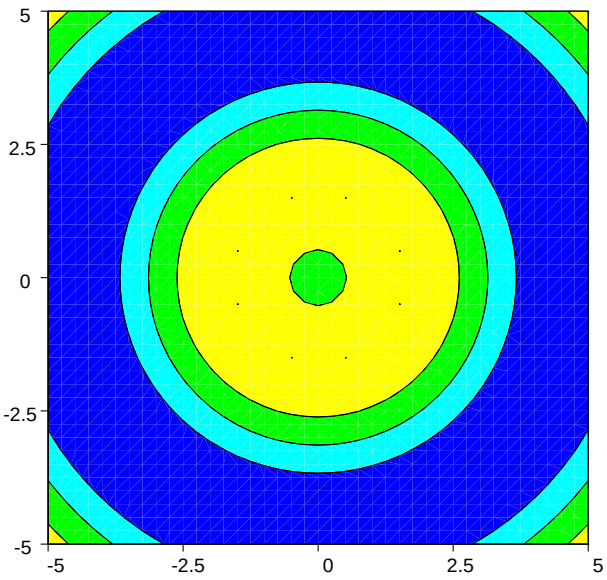


Figure 2.12: Filled contour plot created by the `contourf` command (VTK backend).

Pseudocolor Plots

Another way of visualizing a 2D scalar field in Easyviz is the `pcolor` command. This command creates a pseudocolor plot, which is a flat surface viewed from above. The simplest form of this command follows the syntax of the other commands:

```
pcolor(xv, yv, values)
```

We can set the color shading in a pseudocolor plot either by giving the `shading` keyword argument to `pcolor` or by calling the `shading` command. The color shading is specified by a string that can be either `'faceted'` (default), `'flat'`, or `'interp'` (interpolated). The Gnuplot and Matplotlib backends support `'faceted'` and `'flat'` only, while the VTK backend supports all of them.

Isosurface Plots

For 3D scalar fields, isosurfaces or contour surfaces constitute the counterpart to contour lines or isolines for 2D scalar fields. An isosurface connects points in a scalar field with (approximately) the same scalar value and is mathematically defined by the implicit equation $f(x, y, z) = c$. In Easyviz, isosurfaces are created with the `isosurface` command. We will demonstrate this command using 3D scalar field data from the `flow` function. This function, also found in Matlab, generates fluid flow data. Our first isosurface visualization example (taken from the Matlab documentation) then looks as follows:

```
x, y, z, v = flow()  # generate fluid-flow data
set(show=False)
h = isosurface(x,y,z,v,-3)
h.set(opacity=0.5)
```

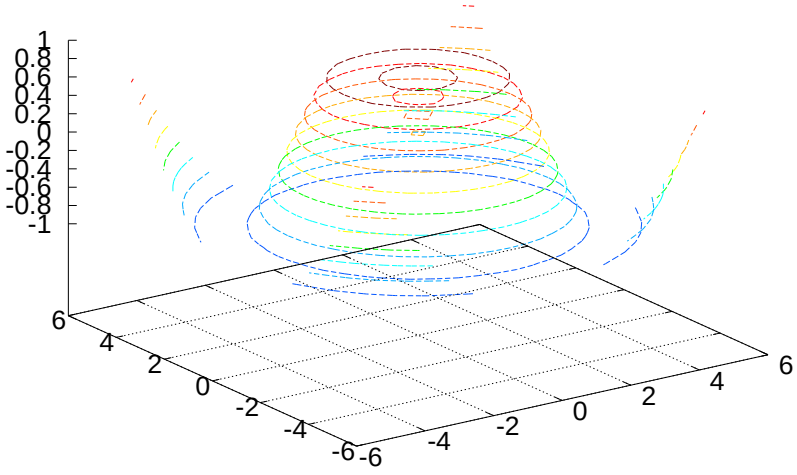
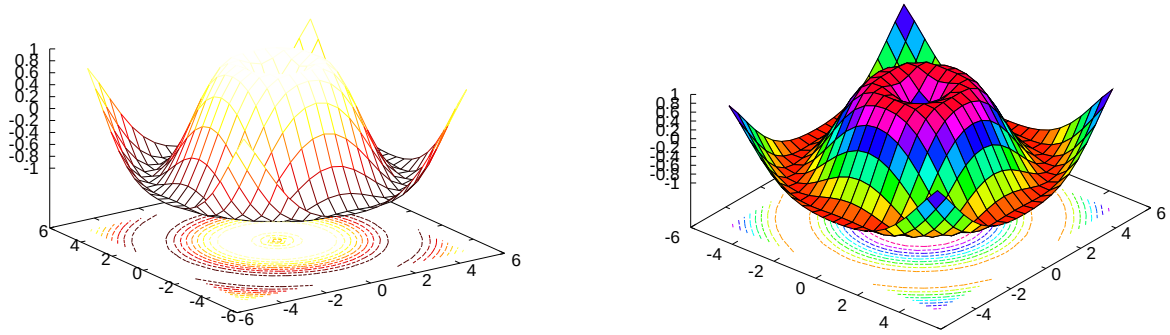


Figure 2.13: Example on the `contour3` command for elevated contour levels (Gnuplot backend).



(a) (b)

Figure 2.14: (a) Wireframe mesh with contours at the bottom (Gnuplot backend). (b) Surface plot with contours (Gnuplot backend).

```
shading('interp')
daspect([1,1,1])
view(3)
axis('tight')
set(show=True)
show()
```

After creating some scalar volume data with the `flow` function, we create an isosurface with the isovalue -3. The isosurface is then set a bit transparent (`opacity=0.5`) before we specify the

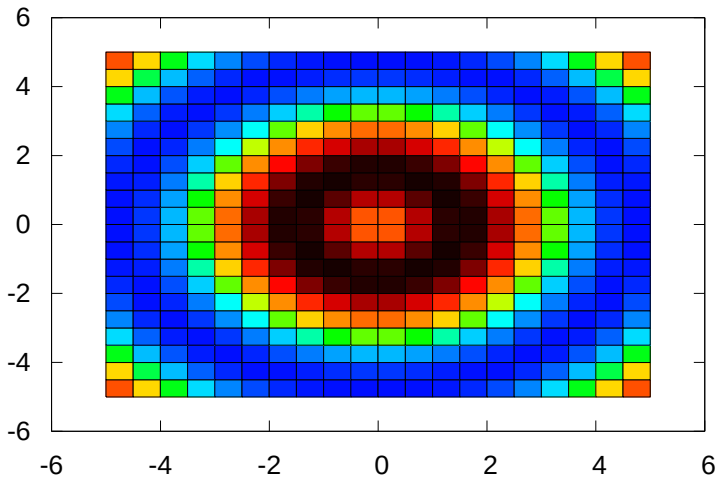
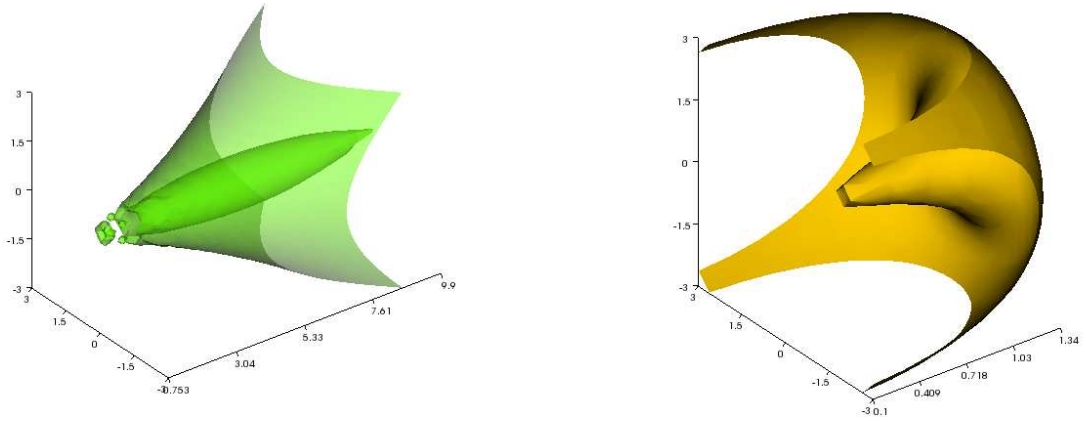


Figure 2.15: Pseudocolor plot (Gnuplot backend).

shading model and the view point. We also set the data aspect ratio to be equal in all directions with the `daspect` command. The resulting plot is shown in Figure 2.16(a). We remark that the Gnuplot backend does not support 3D scalar fields and hence not isosurfaces.



(a) (b)

Figure 2.16: (a) Isosurface plot (VTK backend). (b) Another isosurface plot (VTK backend).

Here is another, more compact example that demonstrates the `isosurface` command (again using the `flow` function):

```
x, y, z, v = flow()
isosurface(x,y,z,v,0,
```

```
shading='interp',
daspect=[1,4,4],
view=[-65,20],
axis='tight')
```

Figure 2.16(b) shows the resulting plot.

Volumetric Slice Plot

Another way of visualizing scalar volume data is by using the `slice_` command (since the name `slice` is already taken by a built-in function in Python for array slicing, we have followed the standard Python convention and added a trailing underscore to the name in Easyviz – `slice_` is thus the counterpart to the Matlab function `slice`.). This command draws orthogonal slice planes through a given volumetric data set. Here is an example on how to use the `slice_` command:

```
x, y, z = meshgrid(seq(-2,2,.2), seq(-2,2,.25), seq(-2,2,.16),
                   sparse=True)
v = x*exp(-x**2 - y**2 - z**2)
xslice = [-1.2, .8, 2]
yslice = 2
zslice = [-2, 0]
slice_(x, y, z, v, xslice, yslice, zslice,
       colormap=hsv(), grid='off')
```

Note that we here use the SciTools function `seq` for specifying a uniform partitioning of an interval – the `linspace` function from `numpy` could equally well be used. The first three arguments in the `slice_` call are the grid points in the x, y, and z directions. The fourth argument is the scalar field defined on-top of the grid. The next three arguments defines either slice planes in the three space directions or a surface plane (currently not working). In this example we have created 6 slice planes: Three at the x axis (at $x = -1.2$, $x = 0.8$, and $x = 2$), one at the y axis (at $y = 2$), and two at the z axis (at $z = -2$ and $z = 0.0$). The result is presented in Figure 2.17.

Contours in Slice Planes. With the `contourslice` command we can create contour plots in planes aligned with the coordinate axes. Here is an example using 3D scalar field data from the `flow` function:

```
x, y, z, v = flow()
set(show=False)
h = contourslice(x, y, z, v, seq(1,9), [], [0], linspace(-8,2,10))
axis([0, 10, -3, 3, -3, 3])
daspect([1, 1, 1])
ax = gca()
ax.set(fgcolor=(1,1,1), bgcolor=(0,0,0))
box('on')
view(3)
set(show=True)
show()
```

The first four arguments given to `contourslice` in this example are the extended coordinates of the grid (`x`, `y`, `z`) and the 3D scalar field values in the volume (`v`). The next three arguments defines the slice planes in which we want to draw contour lines. In this particular example we have specified two contour plots in the planes $x = 1, 2, \ldots, 9$, none in $y = \text{const}$ planes (empty list) , and one contour plot in the plane $z = 0$. The last argument to `contourslice` is optional, it can be either an integer specifying the number of contour lines (the default is five) or, as in

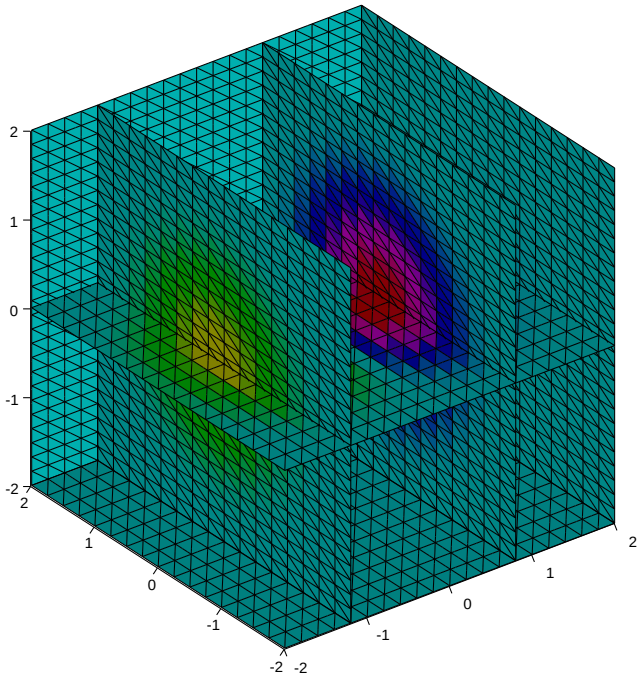


Figure 2.17: Slice plot where the x axis are sliced at -1.2, 0.8, and 2, the y axis is sliced at 2, and the z axis are sliced at -2 and 0.0 (VTK backend).

the current example, a list specifying the level curves. Running the set of commands results in the plot shown in Figure 2.18.

Here is another example where we draw contour slices from a three-dimensional MRI data set:

```
import scipy
mri = scipy.io.loadmat('mri_matlab_v6.mat')
D = mri['D']
image_num = 8

# Displaying a 2D Contour Slice:
contourslice(D, [], [], image_num, daspect=[1,1,1])
```

The MRI data set is loaded from the file `mri_matlab_v6.mat` with the aid from the `loadmat` function available in the `io` module in the SciPy package. We then create a 2D contour slice plot with one slice in the plane $z = 8$. Figure 2.19 displays the result.

2.2.9 Visualization of vector fields

A vector field is a function from space or space-time to a vector value, where the number of components in the vector corresponds to the number of space dimensions. Primary examples on vector fields are the gradient of a scalar field; or velocity, displacement, or force in continuum physics.

In Easyviz, a vector field can be visualized either by a quiver (arrow) plot or by various kinds of stream plots like stream lines, stream ribbons, and stream tubes. Below we will look closer at each of these visualization techniques.

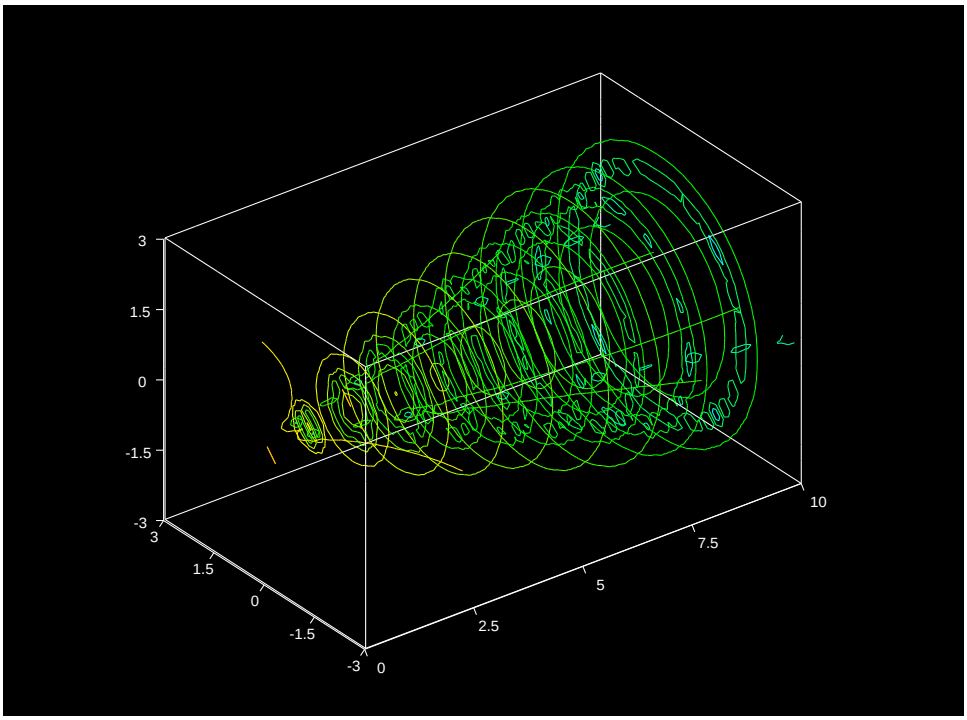


Figure 2.18: Contours in slice planes (VTK backend).

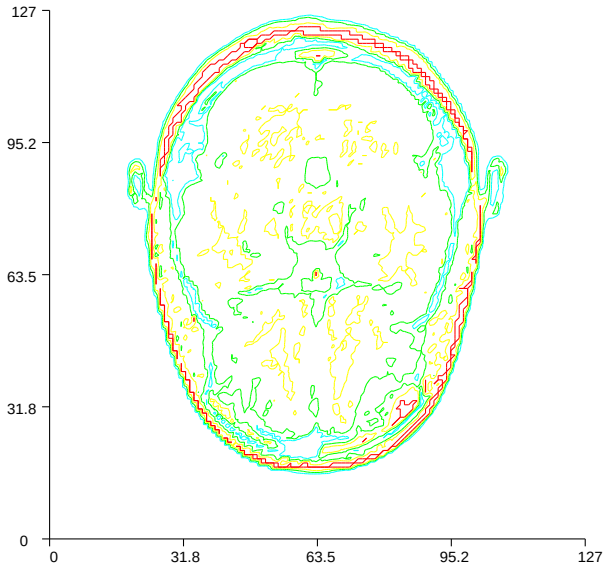


Figure 2.19: Contour slice plot of a 3D MRI data set (VTK backend).

Quiver Plots

The `quiver` and `quiver3` commands draw arrows to illustrate vector values (length and direction) at discrete points. As the names indicate, `quiver` is for 2D vector fields in the plane and `quiver3`

plots vectors in 3D space. The basic usage of the `quiver` command goes as follows:

```
x = y = linspace(-5, 5, 21)
xv, yv = meshgrid(x, y, sparse=False)
values = sin(sqrt(xv**2 + yv**2))
uv, vv = gradient(values)
quiver(xv, yv, uv, vv)
```

Our vector field in this example is simply the gradient of the scalar field used to illustrate the commands for 2D scalar field plotting. The `gradient` function computes the gradient using finite difference approximations. The result is a vector field with components `uv` and `vv` in the x and y directions, respectively. The grid points and the vector components are passed as arguments to `quiver`, which in turn produces the plot in Figure 2.20.

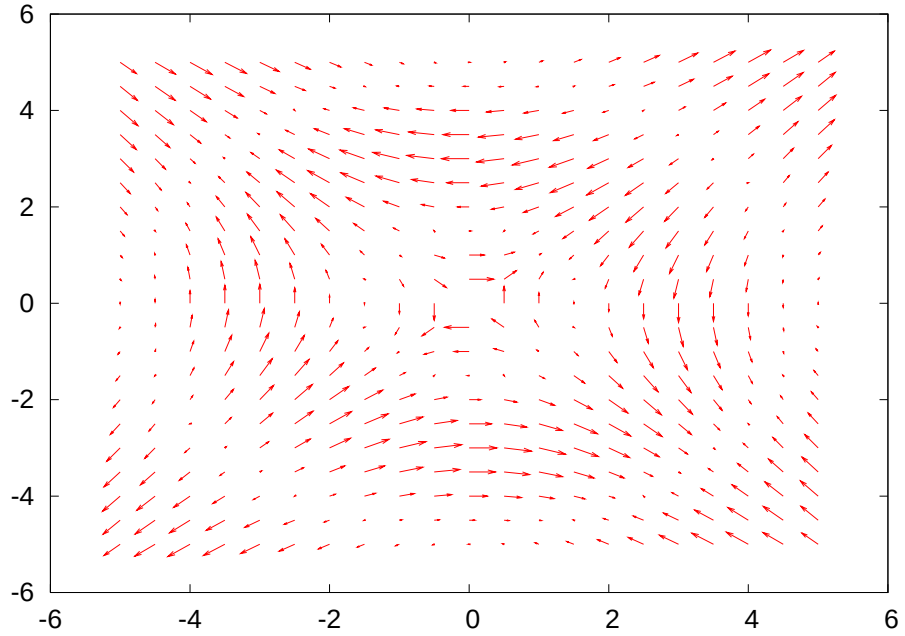


Figure 2.20: Velocity vector plot (Gnuplot backend).

The arrows in a quiver plot are automatically scaled to fit within the grid. If we want to control the length of the arrows, we can pass an additional argument to scale the default lengths:

```
scale = 2
quiver(xv, yv, uv, vv, scale)
```

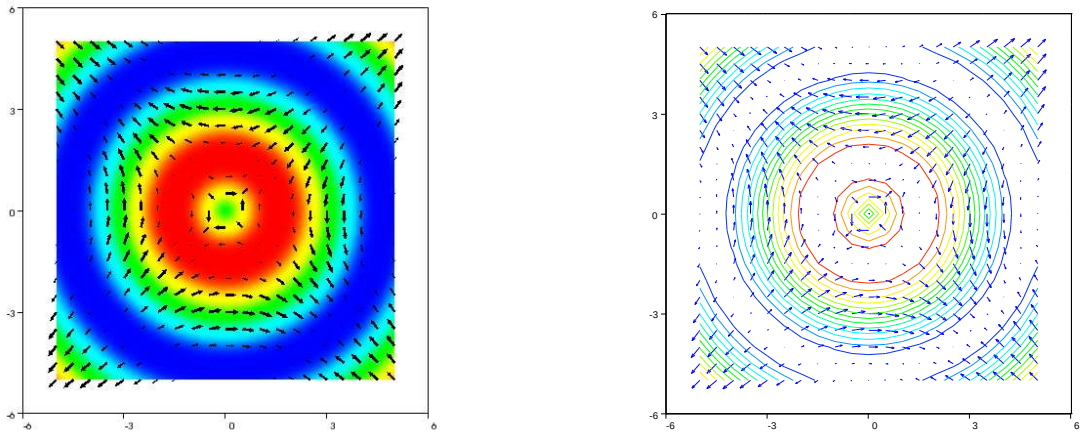
This value of `scale` will thus stretch the vectors to their double length. To turn off the automatic scaling, we can set the scale value to zero.

Quiver plots are often used in combination with other plotting commands such as pseudo-color plots or contour plots, since this may help to get a better perception of a given set of data. Here is an example demonstrating this principle for a simple scalar field, where we plot the field values as colors and add vectors to illustrate the associated gradient field:

```
xv, yv = meshgrid(seq(-5,5,0.1), seq(-5,5,0.1))
values = sin(sqrt(xv**2 + yv**2))
pcolor(xv, yv, values, shading='interp')
```

```
# create a coarser grid for the gradient field:
xv, yv = meshgrid(seq(-5,5,0.5), seq(-5,5,0.5))
values = sin(sqrt(xv**2 + yv**2))
uv, vv = gradient(values)
hold('on')
quiver(xv, yv, uv, vv, 'filled', 'k', axis=[-6,6,-6,6])
figure(2)
contour(xv, yv, values, 15)
hold('on')
quiver(xv, yv, uv, vv, axis=[-6,6,-6,6])
```

The resulting plots can be seen in Figure 2.21(a) and 2.21(b).



(a) (b)

Figure 2.21: (a) Combined quiver and pseudocolor plot (VTK backend). (b) Combined quiver and pseudocolor plot (VTK backend).

Visualization of 3D vector fields by arrows at grid points can be done with the `quiver3` command. At the time of this writing, only the VTK backend supports 3D quiver plots. A simple example of plotting the "radius vector field" $\vec{v} = (x, y, z)$ is given next:

```
x = y = z = seq(-3,3,2)
xv, yv, zv = meshgrid(x, y, z, sparse=False)
uv = xv
vv = yv
wv = zv
quiver3(xv, yv, zv, uv, vv, wv, 'filled', 'r',
        axis=[-7,7,-7,7,-7,7])
```

The strings `'filled'` and `'r'` are optional and makes the arrows become filled and red, respectively. The resulting plot is presented in Figure 2.22.

Stream Plots

Stream plots constitute an alternative to arrow plots for visualizing vector fields. The stream plot commands currently available in Easyviz are `streamline`, `streamtube`, and `streamribbon`. Stream lines are lines aligned with the vector field, i.e., the vectors are tangents to the stream-lines. Stream tubes are similar, but now the surfaces of thin tubes are aligned with the vectors.

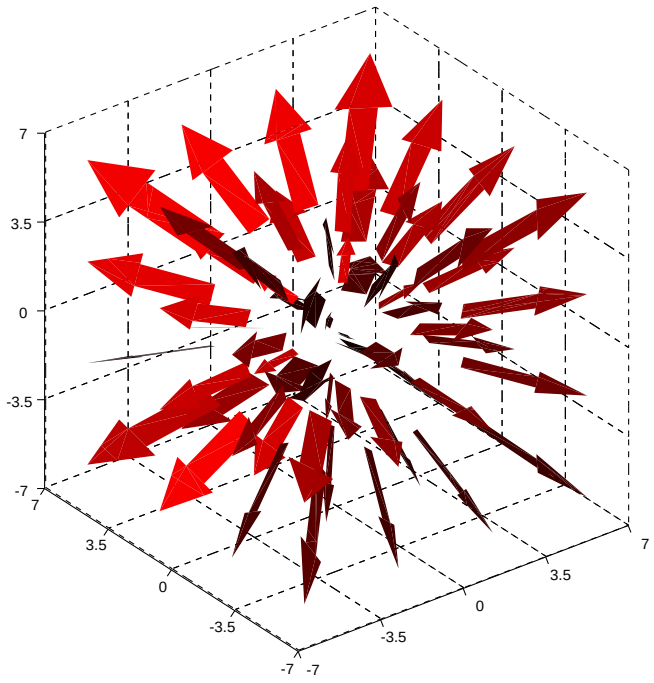


Figure 2.22: 3D quiver plot (VTK backend).

Stream ribbons are also similar: thin sheets are aligned with the vectors. The latter type of visualization is also known as stream or flow sheets. In the near future, Matlab commands such as `streamslice` and `streamparticles` might also be implemented.

We start with an example on how to use the `streamline` command. In this example (and in the following examples) we will use the `wind` data set that is included with Matlab. This data set represents air currents over a region of North America and is suitable for testing the different stream plot commands. The following commands will load the `wind` data set and then draw some stream lines from it:

```
import scipy  # need scipy to load binary .mat-files

# load the wind data set and create variables:
wind = scipy.io.loadmat('wind.mat')
x = wind['x']
y = wind['y']
z = wind['z']
u = wind['u']
v = wind['v']
w = wind['w']

# create starting points for the stream lines:
sx, sy, sz = meshgrid([80]*4, seq(20,50,10), seq(0,15,5),
                      sparse=False)

# draw stream lines:
streamline(x, y, z, u, v, w, sx, sy, sz,
           view=3, axis=[60,140,10,60,-5,20])
```

The `wind` data set is stored in a binary '.mat'-file called `wind.mat`. To load the data in this file into Python, we can use the `loadmat` function which is available through the `io` module in SciPy. Using the `loadmat` function on the 'wind.mat'-file returns a Python dictionary (called `wind` in

the current example) containing the NumPy arrays `x`, `y`, `z`, `u`, `v`, and `w`. The arrays `u`, `v`, and `w` are the 3D vector data, while the arrays `x`, `y`, and `z` defines the (3D extended) coordinates for the associated grid. The data arrays in the dictionary `wind` are then stored in separate variables for easier access later.

Before we call the `streamline` command we must set up some starting point coordinates for the stream lines. In this example, we have used the `meshgrid` command to define the starting points with the line:

```
sx, sy, sz = meshgrid([80]*4, seq(20,50,10), seq(0,15,5))
```

This command defines starting points which all lie on $x = 80$, $y = 20, 30, 40, 50$, and $z = 0, 5, 10, 15$. We now have all the data we need for calling the `streamline` command. The first six arguments to the `streamline` command are the grid coordinates `(x,y,z)` and the 3D vector data `(u,v,w)`, while the next three arguments are the starting points which we defined with the `meshgrid` command above. The resulting plot is presented in Figure 2.23(a).

The next example demonstrates the `streamtube` command applied to the same `wind` data set:

```
streamtube(x, y, z, u, v, w, sx, sy, sz,
           daspect=[1,1,1],
           view=3,
           axis='tight',
           shading='interp')
```

The arrays `sx`, `sy`, and `sz` are the same as in the previous example and defines the starting positions for the center lines of the tubes. The resulting plot is presented in Figure 2.23(b).

Finally, we illustrate the `streamribbon` command:

```
streamribbon(x, y, z, u, v, w, sx, sy, sz,
             ribbonwidth=5,
             daspect=[1,1,1],
             view=3,
             axis='tight',
             shading='interp')
```

Figure 2.23(c) shows the resulting stream ribbons.

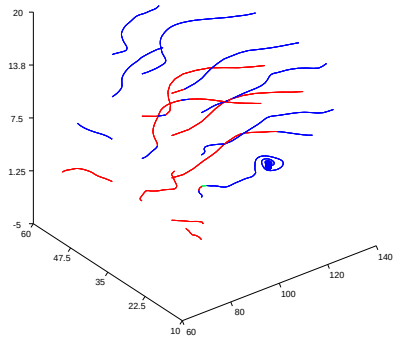
2.3 Design

2.3.1 Main Objects

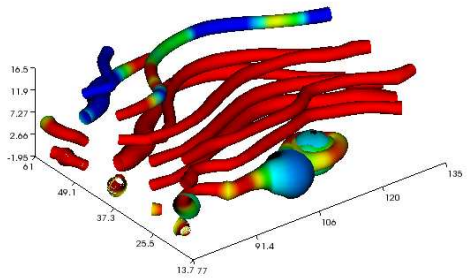
All code that is common to all backends is gathered together in a file called `common.py`. For each backend there is a separate file where the backend dependent code is stored. For example, code that are specific for the Gnuplot backend, are stored in a file called `gnuplot_.py` and code specific for the VTK backend are stored in `vtk_.py` (note the final underscore in the stem of the filename – all backend files have this underscore).

Each backend is a subclass of class `BaseClass`. The `BaseClass` code is found in `common.py` and contains all common code for the backends. Basically, a backend class extends `BaseClass` with rendering capabilities and backend-specific functionality.

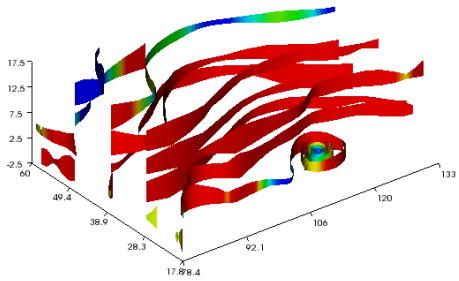
The most important method that needs to be implemented in the backend is the `_replot` method, which updates the backend and the plot after a change in the data. Another important method for the backend class is the `hardcopy` method, which stores an image of the data in the current figure to a file.



(a)



(b)



(c)

Figure 2.23: (a) Stream line plot (VTK backend). (b) Stream tubes (VTK backend). (c) Stream ribbons (VTK backend).

Inspired by Matlab, the Easyviz interface is organized around figures and axes. A figure contains an arbitrary number of axes, and the axes can be placed in arbitrary positions in the figure window. Each figure appears in a separate window on the screen. The current figure is accessed by the `gcf()` call. Similarly, the current axes are accessed by calling `gca()`.

It is natural to have one class for figures and one for axes. Class `Figure` contains a dictionary with one (default) or more `Axis` objects in addition to several properties such as figure width and height. Class `Axis` has another dictionary with the plot data as well as lots of parameters for colors, text fonts, labels on the axes, hidden surfaces, etc. For example, when adding an elevated surface to the current figure, this surface will be appended to a list in the current `Axis` object. Optionally one can add the surface to another `Axis` object by specifying the `Axis` instance as an argument.

All the objects that are to be plotted in a figure such as curves, surfaces, vectors, and so on, are stored in respectively classes. An elevated surface, for instance, is represented as an instance

of class `Surface`. All such classes are subclasses of `PlotProperties`. Besides being the base class of all objects that can be plotted in a figure (`Line`, `Surface`, `Contours`, `VelocityVectors`, `Streams`, `Volume`), class `PlotProperties` also stores various properties that are common to all objects in a figure. Examples include line properties, material properties, storage arrays for x and y values for `Line` objects, and x, y, and z values for 3D objects such as `Volume`.

The classes mentioned above, i.e., `BaseClass` with subclasses, class `PlotProperties` with subclasses, as well as class `Figure` and class `Axis` constitute the most important classes in the Easyviz interface. Less important classes are `Camera`, `Light`, `Colorbar`, and `MaterialProperties`.

All the classes in `common.py` follows a convention where class parameters are set by a `set` method and read by a `get` method. For example, we can set axis limits using the `set` methods in a `Axis` instance:

```
ax = gca()                  # get current axes
ax.set(xmin=-2, xmax=2)
```

To extract the values of these limits we can write

```
xmin = ax.get('xmin')
xmax = ax.get('xmax')
```

Normal use will seldom involve `set` and `get` functions, since most most users will apply the Matlab-inspired interface and set, e.g., axis limits by

```
axis([-2, 2, 0, 6])
```

Chapter 3

Concluding Remarks

The purpose with this thesis was to investigate how to create unified interfaces to important software components needed when solving partial differential equations. The interfaces should be clean and simple, using a familiar Matlab-style if possible, but implemented in Python. Applications of the interfaces to solve some simpler PDEs was also a topic.

3.1 Significance of Results

The main result is that we have implemented and documented reusable, general libraries for (sparse) matrices and for plotting that are ready for being used in PDE codes, as I have demonstrated. With these libraries, one can write simple "Matlab-like" code to solve PDEs. In particular, one can use standard Matlab syntax for plotting curves, scalar fields and vector fields, with a flexible choice of the underlying visualization package (which can be Matlab itself, Gnuplot, or other plotting software).

In Chapter 1 we successfully created a matrix library in Python with support for many different matrix formats that arise when solving PDEs with the Finite Difference, Finite Element, and Finite Volume methods. The matrix library was designed as a class hierarchy with the aim to be user-friendly and without loosing computational efficiency. This kind of software already exist, however, it is the first of its kind written in Python. The use of Python has its drawbacks when it comes to number crunching, but this can easily be avoided by letting the CPU intensive parts of the code be handled by a compiled languages such as Fortran or C/C++. To this end, we introduced the libraries LAPACK and BLAS via the SciPy package. We extended SciPy with interfaces for more LAPACK and BLAS routines, including factorizing, solve using factorization, and matrix-vector product for banded and tridiagonal matrices, both symmetric and unsymmetric versions. Also interfaces for LAPACK routines for eigenvalue problems for symmetric banded and symmetric tridiagonal matrices were added to SciPy; however, this did not become part of this thesis.

We applied the matrix library on a few simple PDEs and we touched subjects such as vectorization and migration of code to compiled languages – both very important techniques when working with Python in scientific computing. The results showed that most of the work are done in the underlying Fortran and C/C++ code, implying that the implementation should be nearly as fast as a pure Fortran or C/C++ implementation.

We also wanted to test the ADI method which was a very popular solution method during the 1960s, but was later "forgotten"; however, it is still mentioned in many books these days. The results compared to direct (sparse) Gaussian elimination in the 2D diffusion problem was

a little unclear. We expected the ADI implementation to be a faster solution method for that problem, but the speed was more or less the same in both methods.

In Chapter 2 we introduced Easyviz, a user-friendly visualization tool written in Python. Easyviz is designed as a unified interface to other plotting packages that can be called from Python. Scripts with Easyviz commands only can therefore work with a variety of plotting packages and as new fancy plotting packages become available and support for them are added to Easyviz, these scripts will still work great. The syntax is similar to what we find in Matlab and thus very easy to use for those already familiar with Matlab.

Easyviz is already a stable and useful package and has been taken in use by some of the scientific researchers at Simula Research Laboratory at Fornebu. Later this year, it will also be used in a beginners course in scientific computing at the University in Oslo.

PyPDELib is open source software and can be download from the World Wide Web via `http://folk.uio.no/johannr/thesis`. From this page you can also download a PDF version of this thesis. Easyviz will be available for download later this year.

3.2 Future Work

The matrix library is ready for use, but needs more extensive testing and more features should be added. As we saw, the accuracy was not quite what it should be according to theory and this is naturally to investigate further. It is also natural to compare the implementations presented in Section 1.9 to a pure C or Fortran implementation to make sure that the use of Python is not resulting in any great speed loss. The results from the ADI implementation was a little unclear and this is something that should be investigated further.

The Easyviz interface will be under heavy development for the next couple of months. We are planning to add interfaces for more backends, like VisIt, Veusz, Grace, OpenDX, and several others. We should therefore create a template file for backends, making it easier to add support for new plotting packages. We will also do more volume visualization in VTK. The code (especially in the backends) needs to be cleaned up and the documentation should be improved and extended.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 2nd edition, 1995.

[2] BLAS software package. *http://www.netlib.org/blas/*.

[3] A. M. Bruaset and H. P. Langtangen. Object-oriented design of preconditioned iterative methods in Diffpack. *Transactions on Mathematical Software*, 23:50–80, 1997.

[4] Diffpack software package. *http://www.diffpack.com*.

[5] DOLFIN software package. *http://www.fenics.org/wiki/DOLFIN*.

[6] J. Douglas and J. Kim. On accuracy of alternating direction implicit methods for parabolic equations, 1999.

[7] C. A. J. Fletcher. *Computational Techniques for Fluid Dynamics, Vol I and II*. Springer Series in Computational Physics. Springer, 1988.

[8] R. Geus. Calculating electro-magnetic waves in accelerator cavities using Python. *http://people.web.psi.ch/geus/pyfemax/index.html*.

[9] R. Geus. *The Jacobi-Davidson algorithm for solving large sparse symmetric eigenvalue problems with application to the design of accelerator cavities*. PhD thesis, Swiss Federal Institute of Technology Zurich, 2002.

[10] GLAS: Generic Linear Algebra Software. *http://glas.sourceforge.net/*.

[11] Gnuplot software package. *http://www.gnuplot.info/*.

[12] Grace software package. *http://plasma-gate.weizmann.ac.il/Grace/*.

[13] M. B. Allen III, I. Herrera, and G. F. Pinder. *Numerical Modeling in Science and Engineering*. Wiley, 1988.

[14] E. Jones. Weave Documentation. *http://projects.scipy.org/scipy/scipy/browser/trunk/Lib/weave/doc/tutorial.html?format=raw*.

[15] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[16] H. P. Langtangen. *Computational Partial Differential Equations – Numerical Methods and Diffpack Programming*. Text in Computational Science and Engineering, vol 1. Springer, 2nd edition, 2003.

[17] H. P. Langtangen. *Python Scripting for Computational Science*, volume 3 of *Texts in Computational Science and Engineering*. Springer, 2004.

[18] LAPACK software package. *http://www.netlib.org/lapack/*.

[19] Matlab software package. *http://www.mathworks.com*.

[20] Matplotlib software package. *http://matplotlib.sourceforge.net/*.

[21] Netlib repository of numerical software. *http://www.netlib.org*.

[22] Numerical Python software package. *http://sourceforge.net/projects/numpy*.

[23] OpenDX software package. *http://www.opendx.org/*.

[24] P. Peterson. F2PY software package. *http://cens.ioc.ee/projects/f2py2e*.

[25] Pmw software package. *http://pmw.sourceforge.net/*.

[26] PyACTS software package. *http://www.pyacts.org*.

[27] pygrace software package. *http://www.its.caltech.edu/ mmckerns/software.html*.

[28] Python-gnuplot interface. *http://gnuplot-py.sourceforge.net*.

[29] Python programming language. *http://www.python.org*.

[30] Pyx software package. *http://pyx.sourceforge.net/*.

[31] SciPy software package. *http://www.scipy.org*.

[32] G. van Rossum and F. L. Drake. NumPy Distutils - Users Guide. *http://scipy.org/Documentation/numpy_distutils*.

[33] G. van Rossum and F. L. Drake. Python Library Reference. *http://docs.python.org/lib/lib.html*.

[34] G. van Rossum and F. L. Drake. Python/C API Reference Manual. *http://www.python.org/doc/current/api/contents.html*.

[35] Veusz software package. *http://home.gna.org/veusz/*.

[36] VisIt software package. *http://www.llnl.gov/visit/*.

[37] VTK software package. *http://www.kitware.com*.

[38] T. Williams, C. Kelly, H. Böker, et al. Gnuplot - An Interactive Plotting Program, 2004.