Proceedings of the International MultiConference of Engineers and Computer Scientists 2015 Vol I,
IMECS 2015, March 18 - 20, 2015, Hong Kong

# Python Classes for Numerical Solution of PDE's

Asif Mushtaq, *Member, IAENG,* Trond Kvamsdal, Kåre Olaussen, *Member, IAENG,*

***Abstract*—We announce some Python classes for numerical solution of partial differential equations, or boundary value problems of ordinary differential equations. These classes are built on routines in `numpy` and `scipy.sparse.linalg` (or `scipy.linalg` for smaller problems).**

***Index Terms*—Boundary value problems, partial differential equations, sparse scipy routines.**

## I. Introduction

THE Python computer language has gained increasing popularity in recent years. For good reasons: It is fast and easy to code and use for small "prototyping" tasks, since there is no need for explicit declaration of variables or a separate compilation cycle. It is freely available for most computer platforms, and comes with a huge repository of packages covering a large area of applications. Python also have features which facilitates development and encourages documentation of large well-structured program systems.

Obviously, as an interpreted language native Python is not suitable for performing extended numerical computations. But very often the code for such computations reduces to calls to precompiled library routines. The **`numpy`** [1] and **`scipy`** [2], [3] packages make a large number of such routines directly available from Python. These packages are freely available for most operating systems, including Linux, OSX, and MSWindows.

We here describe a process of making some of these routines even simpler to use for a field of applications, the numerical solution of partial differential equations discretized on a rectangular grid (or a subdomain of such a grid). As a simple reference problem one may consider the solution of the wave equation in the frequency domain,

$$\left(-\Delta + \omega^2\right) \varphi(\boldsymbol{x}) = f(\boldsymbol{x}), \tag{1}$$

f.i. in a space with periodic boundary conditions. Our work is to a considerable extent motivated by a goal to solve the 3D acoustic wave equation with position dependent material properties, and its related inverse problem [4], [5], to interesting accuracy in acceptable time on current (2015) high-end laptops.

However, the classes used to solve this problem are designed with additional topologies, geometries, and applications in mind. These classes are **`Lattice`**, **`LatticeFunction`**, and **`LatticeOperator`**. A specific application from Quantum Mechanics [6] has been refactored to extend these classes.

Manuscript received January 22, 2015.

Asif Mushtaq is with the Department of Mathematical Sciences, Norwegian University of Science and Technology (NTNU), Trondheim. e-mail: asif.mushtaq@math.ntnu.no

Trond Kvamsdal is with the Department of Mathematical Sciences, NTNU, Trondheim, N-7491, Norway. e-mail: trond.kvamsdal@math.ntnu.no

Kåre Olaussen is with the Department of Physics, NTNU, Trondheim, N-7491, Norway. e-mail: Kare.Olaussen@ntnu.no

## II. The Lattice class

This class is intended to handle the most basic properties and operations of a discretized model. We divide them into topological and geometrical aspects of the model. The most basic properties of a dicrete model are the dimensionality of space, and how we approximate a continuous space with a number of sites in each direction (referred to as its **`shape`**). The code snippet

```
L1 = Lattice(shape=(2**13, ))
L2 = Lattice(bC=('P', 'A'))
L3 = Lattice(shape=(2**8, 2**8, 2**7))
```

demonstrate how three **`Lattice`** instances can be defined, `L1` with a one-dimensional lattice of $2^{13} = 8\,192$ sites, `L2` with (by default) a two-dimensional lattice of $2^7 \times 2^7$ sites, and `L3` with a three-dimensional lattice of $2^8 \times 2^8 \times 2^7 = 8\,388\,608$ sites. In this process the instance properties **`shape`**, **`dim`**, and **`size`** are specified or given default values.

### A. Boundary conditions

One additional property, **`bC`**, specifies the default boundary conditions in all directions. These conditions specify how functions defined on a finite lattice is extended beyond its edges, as is required when applying discrete differential operators or operations like Fast Fourier Transforms (FFT).

Each specific case of **`bC`** is a property of each function defined on the lattice. Hence it belongs to the class **`LatticeFunction`**, to be used and set by methods of **`LatticeOperator`**. However, since **`bC`** is often the same for all functions and operators in a given lattice model, it is convenient to provide a default property, which may be inherited by instances of **`LatticeFunction`** and **`LatticeOperator`**.

The default value of **`bC`** is **`'allP'`**, for periodic boundary conditions in all directions. Otherwise, **`bC`** must be a list with possible entries **`'P'`** (for periodic extension), **`'S'`** (for symmetric extension), **`'A'`** (for antisymmetric extension), **`'F'`** (for extension with fixed provided values), and **`'Z'`** (for extension with zero values).

For **`'S'`** and **`'A'`** the symmetry point is midway between two lattice points. The boundary condition can be specified differently in different directions, and (unless periodic **`'P'`**) differently at the two edges of a given direction (in which case the corresponding entry in **`bC`** must be a two-component list). Internally **`bC`** is either stored as `[['allP'], ]`, or as a **`dim`**-component list of two-component lists. The **`Lattice`** class is equipped with a method, **`set_bC(bC='allP')`**, which returns the internal representation from a variety of possible inputs.

### B. Subdomains and slices

Assume that $\phi(\boldsymbol{n})$ and $\phi_{\mathrm{O}}(\boldsymbol{n})$ are two arrays defined on a 3-dimensional lattice, with $s_{\mathrm{O}}$ a constant, and that we want

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

to perform the operation

$$\phi_O(\boldsymbol{n}) = \phi_O(\boldsymbol{n}) + s_O\,\phi(\boldsymbol{n}). \tag{2}$$

Python code for this operation could be the snippet

```
for nx in range(phi.shape[0]):
    for ny in range(phi.shape[1]):
        for nz in range(phi.shape[2]):
            phiO[nx, ny, nz] = \
                phiO[nx, ny, xz] + \
                sO*phi[nx, ny, nz]
```

This code is lengthy (hence error-prone) and runs slowly, because all **for**-loops are executed in native Python. The **numpy** code for the same operation is simply

```
phiO += sO*phi
```

wherein all loop operations are delegated to **numpy** (and maybe further translated to optimized **BLAS** operations).[1] The similar operation corresponding to

$$\phi_O(\boldsymbol{n}) = \phi_O(\boldsymbol{n}) + \sum_{\boldsymbol{b}} s_O(\boldsymbol{b})\,\phi(\boldsymbol{n}-\boldsymbol{b}), \tag{3}$$

where $\boldsymbol{b}$ is a non-zero integer vector, requires more care and coding, since there will be values of $\boldsymbol{n}$ for which $\boldsymbol{n}-\boldsymbol{b}$ falls outside the lattice. In such cases the expression $\phi(\boldsymbol{n}-\boldsymbol{b})$ must be related to known values of $\phi$ by use of the boundary conditions. Assume a case where `b = (3,0,-2)`, that the lattice have (much) more than 3 sites in all directions, and that the boundary conditions is given by

```
bC = [['P','P'],['S','A'],['A','S']]
```

We may first treat the sites $\boldsymbol{n}$ where also $\boldsymbol{n}-\boldsymbol{b}$ fall inside the lattice:

```
phiO[3:,:,-2] += sO[3,0,-2]*phi[:-3,:,2:]
```

Here the *slice*-notation defines a rectangular subdomain of the lattice. For instance, the slice `[3:,:,-2]` specifies the intersection of (i) all planes in the $x$-direction except the first 3, (ii) all planes in the $y$-direction, and (iii) all planes in the $z$-direction except the last 2.

Note that array positions are counted from zero, with negative numbers referring to distances from the end. For a large lattice the above operation would cover most of the cases, and everything if the boundary conditions were `'Z'` in all directions.

In our example there are three more regions to be included:

$$0 \le n_x < 3, \text{ and } 0 \le n_z < -2, \tag{4a}$$

$$3 \le n_x \le -1, \text{ and } -2 \le n_z \le -1, \tag{4b}$$

$$0 \le n_x < 3, \text{ and } -2 \le n_z \le -1. \tag{4c}$$

The case (4a) can be handled by the code

```
phiO[:3,:,:-2] += \
        sO[3,0,-2]*phi[-3:,:,2:]
```

using the periodic boundary condition in the $x$-direction. For the cases (4b) and (4c) two planes in the $z$-direction fall outside the lattice on the upper side. Due to the symmetric `'S'` boundary condition at this edge of the lattice, the function values on these planes are related to their values on the last two planes inside the lattice (counted in opposite order). This can be handled by the code

```
phiO[3:,:,-2:] += \
        sO[3,0,-2]*phi[:-3,:,:-3:-1]
phiO[:3,:,-2:] += \
        sO[3,0,-2]*phi[-3:,:,:-3:-1]
```

For detailed information about indexing and slicing in **numpy**, consult the *Indexing* section of the Numpy reference manual [7]. However, gory details like the above are best handled by computers. The **Lattice** class provides a method, **targetNsource**`(b, bC=None)`, which yields all the source and target slices required for a given vector $\boldsymbol{b}$. Using this, the code snippet

```
for cf, dT, dS in L3.targetNsource(b):
    phiO[dT] += cf*phi[dS]
```

replaces all operations above. Here the coefficient `cf` is $-1$ if an odd number of antisymmetric boundary conditions are employed (otherwise $+1$).

**Lattice** also provides a related method, **domain**`(shape, shift)`. This returns a slice `dI` pointing to a rectangular subdomain of the lattice, of shape **shape**, shifted from the origin by an integer vector **shift**.

### C. Index arrays and broadcasting

Each site of a **dim**-dimensional lattice is labeled by a **dim**-dimensional integer index vector $\boldsymbol{n}$. To construct an array `A` defined on all points of a 3-dimensional lattice, one could write a code snippet similar to the following

```
defA = lambda n: \
            numpy.exp(-numpy.dot(n,n))
shape = (2**8, 2**8, 2**8)
A = numpy.zeros(shape)
for n in numpy.ndindex(shape):
    A[n] = defA(numpy.array(n))
```

Although this code is brief and general with respect to dimensionality, it is *not a good way to do it*. Since the **for**-loop will be executed in native Python, the code will run too slow. A better way is to define three arrays `n0`, `n1`, `n2`, all of shape $(2^8, 2^8, 2^8)$, once and for all. We may then replace the code above with the snippet

```
defA = lambda n0, n1, n2: \
    exp(-n0*n0)*exp(-n1*n1)*exp(-n2*n2)
A = defA(n0, n1, n2)
```

All loops are now implicit, and will be executed by compiled **numpy** functions.

Further, the memory cost of permanently storing three large arrays can be avoided by use of the *broadcasting* facility of **numpy**. Since the index array `n0` is constant in the $y$- and $z$-directions, it only contains a one-dimensional amount of information, stored in an array of shape $(2^8, 1, 1)$. Likewise, `n1` can be stored in an array of shape $(1, 2^8, 1)$, and `n2` in an array of shape $(1, 1, 2^8)$. All these arrays contain the same amount of data ($2^8$ linearely stored entries). But, due to their different *shape* they will act differently under f.i. algebraic operations: `n0*n0` will still produce an array of shape $(2^8, 1, 1)$, and similary `n1*n1` an array of shape $(1, 2^8, 1)$. However, the addition of these two results produces an array of shape $(2^8, 2^8, 1)$.

[1] Note that the codeline `phiO = phiO + s0*phi` is *not* equivalent to `phiO += s0*phi`. In the former a new copy of `phiO` is made; this requires more memory.

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Finally, adding `n2*n2` generates an array of the final shape $(2^8, 2^8, 2^8)$. Hence, the cost of computing and storing index arrays are modest. We have chosen *not* to include them as properties, but provide a method **`narr`**`()` which computes them when needed. This method returns a list `n` of arrays, `[n[0], n[1],..]`.

### D. Geometric properties

The discussion above maily concerns topological properties of the lattice. For most application we also need some geometric properties. In general these may be implemented by defining a **`dim`**-dimensional vector of arrays, $\boldsymbol{r}(\boldsymbol{n})$, specifying the position coordinates of all sites. These coordinates (which should depend monotoneously on $\boldsymbol{n}$) could also be dynamical, i.e. part of the equation system to be solved.

The wide range of possibilities indicate that several versions of $\boldsymbol{r}(\boldsymbol{n})$ should be implemented, with the appropriate version chosen when a **`Lattice`** instance is defined. We have introduced a property **`geometry`**, which specifies the version to be used. So far, **`geometry`** can only take the value **`'fixedRect'`**, wherein rectangular regions of space, aligned with the lattice directions, are modelled. Such regions can be specified by a **`dim`**-dimensional vector $\boldsymbol{r}_E$ of edge-lengths, plus a vector $\boldsymbol{r}_0$ specifying the position of the "lower left" corner of the spatial region. For a given lattice **`shape`** parameter, this defines a lattice cell with a vector of sidelengths $\boldsymbol{dr}$, such that

```
dr[d] = rE[d]/shape[d].
```
(5)

The position coordinate $\boldsymbol{r}(\boldsymbol{n})$ is then defined such that its component in the `d`-direction is

```
r[d] = r0[d] + dr[d]*(n[d]+1/2).
```
(6)

This implementation introduces three new properties: **`r0`**, by default a **`dim`**-dimensional tuple with entries 0, **`rE`**, by default a **`dim`**-dimensional tuple with entries 1, and **`dr`**, calculated from equation (5). The method **`rvec`**`()` returns a list `r` of arrays, `[r[0], r[1],...]`, calculated from equation (6).

### E. Lattice initialization, methods, and properties

All currently available keyword arguments and default values for initialization of a **`Lattice`** instance is specified by the code snippet below:

```
def __init__(shape=(128, 128),
    bC='allP', geometry='fixedRect',
    rE=None, r0=None):
```

A value of **`None`** will invoke a default initialization process, following the rules discussed above. The current list of **`Lattice`** methods, with arguments, is as follows:

```
          set_bC  (bC='allP')
          domain  (shape, shift)
    targetNsource (b, bC=None)
            narr  ()
            rvec  ()
```

A summary of all **`Lattice`** properties, with example values, is as provided in the table below.

```
   shape  L1.shape = (8192, )
          L3.shape = (256, 256, 128)
     dim  L1.dim = 1
          L3.dim = 3
    size  L1.size = 8192
          L3.size = 8388608
      bC  L1.bC = [['allP']]
          L2.bC = [['P','P'],['A','A']]
geometry  L1.geometry = 'fixedRect'
      r0  L1.r0 = (0, )
          L3.r0 = (0, 0, 0)
      rE  L1.rE = (1, )
          L3.rE = (1, 1, 1)
      dr  L1.dr = [1.19209290e-07]
          L3.dr = [0.00390625,
                   0.00390625, 0.0078125]
```

## III. The LatticeFunction class

Space does not allow us to continue with an equally detailed discussion of all components in the **`LatticeFunction`** and **`LatticeOperator`** classes. We will instead provide examples of uses, augmented with general comments.

```
L = Lattice(shape=(2**15, 2**15),
rE=(18, 18), r0=(-9, -9))
defF = lambda r: \
    numpy.exp(-r[0]**2/2)* \
    numpy.exp(-r[1]**2/2)
F = LatticeFunction(L, def_F=defF)
t0 = time.time()
F.evalFr()
print (L.size,
(time.time()-t0)/L.size)
```

Here we first define a $2^{15} \times 2^{15}$ lattice model with periodic boundary conditions, and next a gaussian function centered in the middle of this lattice. The parameter **`rE`** is chosen large enough to make the periodic extension of this function smooth: It acquires a discontinuity in the first derivative of magnitude $18\exp(-9^2/2) \approx 0.5 \cdot 10^{-16}$ or smaller (i.e., below double precision accuracy).

The gaussian function is *not* evaluated when the instance `F` is defined, only when we execute the method `F.`**`evalFr`**`()`. This method evaluates the function, and stores the result in the array `F.`**`values`**.

The wall-clock time used to perform this computation on a 2013 MacBook Pro with 16 Gb of memory was measured to 3.26 ns per point. Note that this time is mostly spent multiplying double precision numbers; only $2 \times 2^{15}$ exponential function evaluations are performed. However, if the the code for **`def_F`** is changed to

```
def_F = lambda r: \
    numpy.exp(-(r[0]**2 + r[1]**2)/2)
```

the execution time increases to 118 ns per point. This increase is partly due to the fact that the exponential function is now evaluated $10^{30}$ times, but also because the system now has to deal with *two* very large arrays (one for the argument of the exponential function, and one for final result), and is operating very close to the limit of available memory. A more detailed analysis, for lattices of various sizes (total number of lattice points), is shown in Fig 1.

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

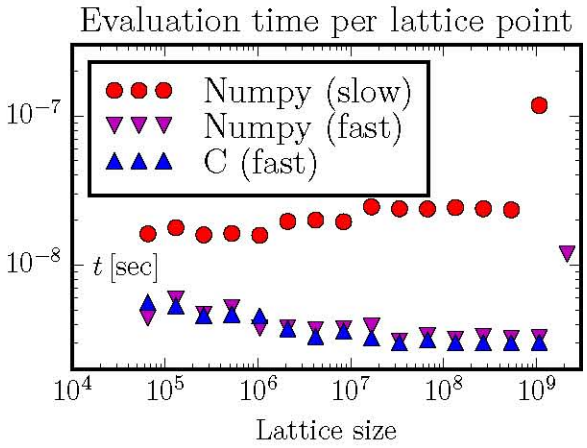


Fig. 1. Comparison of NumPy and C evaluation times for a gaussian defined on lattices of various sizes. The fast evaluation occur when writing the gaussian as $\exp(-x^2/2) \times \exp(-y^2/2)$, the slow evaluation when writing it as $\exp[-(x^2+y^2)/2]$. For these cases the wall-clock and CPU times are essentially the same. As can be seen, there is little to gain in evaluation time by writing the code in a fast, compiled language like C (and a lot to lose in coding time).

### A. FFT and related discrete transforms

We may apply a discrete Fourier transformation to the data stored in `F.values`. This is done by the function call `F.FFT()`. The transformed data is stored in the array `F.fftvalues`. The inverse transform is performed by the function call `F.iFFT()`, with the transformed data being stored in the array `F.values` (overwriting any previous data).

Acctually, the method `FFT()` (or `iFFT()`) do not necessarily perform a regular (multidimensional) discrete Fourier transform `fftn` (or its inverse `ifftn`). This is but one of several related discrete transforms available in `scipy.fftpack`. Other such transforms are the discrete cosine transform `dct` (suitable for functions with symmetric boundary conditions on both sides), the discrete sine transform `dst` (suitable for functions with antisymmetric boundary conditions on both sides), the fast fourier transform `rfft` of real data, and their inverses (`idct`, `idst`, `irfft`). The rules are

1) If `bC[0][0] == 'allP'` the transform `fftn` (or `ifftn`) is used. Complex data is allowed. Otherwise, the data is assumed to be real, and an iterated sequence of transforms over all axes is executed.
2) For directions such that `bC[d][0] == 'S'` the transform `dct` (or `idct`) is performed.
3) For directions such that `bC[d][0] == 'A'` the transform `dst` (or `idst`) is performed.
4) In all other cases the transform `rfft` (or `irfft`) is performed.

Note that the `bC` used here is a property of `LatticeFunction`. This may be different from the corresponding property of its lattice instance. By default they are equal.

The discrete transforms above are useful because they allow (i) differential operators to be implemented as multiplication operators on the transformed functions, and (ii) accurate interpolation of lattice functions outside the lattice sites. The latter is useful for implementation of prolongations in multigrid methods. To assess to which extent this is a practical approach, we have investigated the accuracy and the time requirements of these transforms. The code snippet below illustrate how this can be done:

```
shape = (2**14, 2**14)
L = Lattice(shape=shape, bC=('P','P'))
F = LatticeFunction(L)
F.values = numpy.random.rand(*shape)
values = numpy.copy(myF.values)
t0 = time.time(); F.FFT()
t1 = time.time(); F.iFFT()
t2 = time.time()
err=numpy.max(numpy.abs(F.values-values))
print((t1-t0)/L.size,(t2-t1)/L.size,err)
```

The output of this code shows that the forward transform takes about 68 ns per lattice point, the inverse transform about 57 ns, and that the maximum difference between the original and backtransformed values is $1.7 \times 10^{-15}$. I.e., the cost of a one-way transform is roughly the same as 20 multiplications. The time per site increases by almost an order of magnitude for a lattice of `shape = (2**14, 2**15)`, since this is close to the limit of available memory.

We have investigated the behavior above in more detail, for different choices of the `shape` and `bC` parameters, with similar results. See Fig. 2. The crude conclusion is that the transformation times grow roughly linearly with lattice size, with a prefactor which depends only slightly on transformation type and lattice dimensionality.

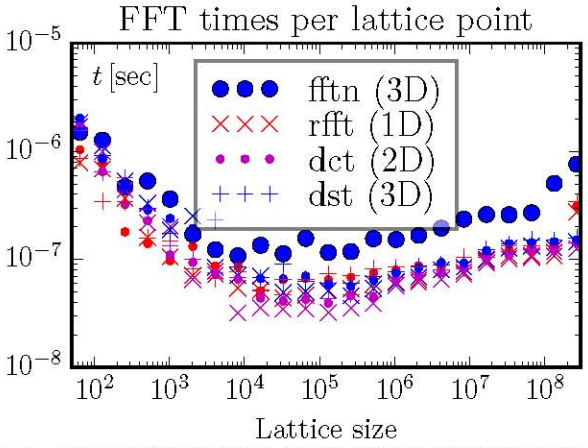


Fig. 2. Time used to perform a discrete lattice transformations of various types. Each time plotted is the *sum* of the forward and inverse transformation time. 1D lattices are plotted in red, 2D lattices in magenta, and 3D lattices in blue. Within the range of lattice sizes allowed by available memory, the theoretically expected logarithmic growth of transformation time with size is not a very distinct feature.

### B. LatticeFunction initialization, methods, and properties

All currently available keyword arguments for initialization of a `LatticeFunction` instance is specified by the argument list below:

```
def __init__(self, lattice, def_f=None,
    def_F=None, def_g=None, def_G=None,
    bC='allP', evalf=False, evalF=False,
    evalg=False, evalG=False):
```

As can be inferred from the above, there are several ways to specify a function: (i) As a function of the index arrays, $f(\boldsymbol{n})$, or as a function of the position vectors, $F(\boldsymbol{r})$. The discrete transform of the function also lives on a lattice, the

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

*dual lattice*, whose sites can be labelled by a list of index arrays `q = [q[0], q[1],..]`. We denote the geometric version of this lattice as *reciprocal space*, wherein each site $q$ has a reciprocal position vector $k(q)$. Hence, the function can also be specified from its discrete transformation, as the function (iii) $g(q)$ or (iv) $G(k)$.

The current list of **LatticeFunction** methods is as follows:

| | |
|---|---|
| **qarr**() | List index vectors for the dual lattice. |
| **kvec**() | List resiprocal position vectors. |
| **evalfn**() | Compute **values** from **def_f**. |
| **evalFr**() | Compute **values** from **def_F**. |
| **evalgq**() | Compute **fftvalues** from **def_g**. |
| **evalGk**() | Compute **fftvalues** from **def_G**. |
| **FFT**() | Discrete transformation of **values**. |
| **iFFT**() | Inverse transformation of **fftvalues**. |
| **shift**(frac) | Return the function translated by frac. |
| **restrict**() | Return the function restricted to a cruder lattice. |
| **prolong**() | Return the function prolonged to a finer lattice. |

The current list of **LatticeFunction** properties is as follows:

| | |
|---|---|
| **lattice** | Related **Lattice** instance. |
| **def_f** | Possible function definition (default **None**). |
| **def_F** | Possible function definition (default **None**). |
| **def_g** | Possible function definition (default **None**). |
| **def_G** | Possible function definition (default **None**). |
| **bC** | Boundary conditions (**lattice.bC**). |
| **values** | Array of function values. |
| **fftvalues** | Array of transformed function values. |

## IV. The LatticeOperator class

Many routines in **scipy.sparse.linalg** do not require an explicit matrix representation of the operator under analysis. Only some algorithm which returns the result of applying the operator to a given vector is needed. Such algorithms can be assigned to a **LinearOperator** instance, after which it functions essentially as an explicit matrix representation. Such algorithms should not demand too much memory or computation time, but do not require any explicitly known sparse representation of the operator. F.i., any computational process involving a fixed number of multiplication, additions and fast fourier transformations will have a memory requirement which scales linearly with the lattice size, and a time requirement which (for large systems) also scales roughly linearly with lattice size.

The **LinearOperator** class requires an input vector of shape `(M, )` or `(M,1)`, and an output vector of shape `(N, )`. For higher-dimensional lattices this does not match the natural construction of lattice operators, which we do not want to interfere with. We have therefore implemented a general **linOp**`(phi0)` method, to be used as a universal `matvec` parameter for **LinearOperator**. The currently implemented code for this is

```
phi = phi0.reshape(self.lattice.shape)
return numpy.ravel(self.varOp(phi))
```

This code assumes `phi0` to represent a scalar function. It will be extended to more general (vector, spinor, tensor,...) objects. The **reshape** and **ravel** operations above do not modify or move any data; they only change how the data is interpreted (the *view* of the data).

The code above also call a specific method, **varOp**`(phi)`. However, this is just a handle which should be assigned to the operator under analysis. The latter may either be an appropriate predefined method in the **LatticeOperator** class, or a method provided from outside.

### A. Explicit matrix representations

It may be useful to inspect an explicit matrix representation of a given operator on a small lattice. The method **matrix**`(operator)` provides such a representation:

```
L = Lattice(shape=(4, ), rE=(4,))
O = LatticeOperator(L)
laplace = O.matrix(O.laplace)
print (laplace)
```

The output from this code is

```
[[-2.  1.  0.  1.]
 [ 1. -2.  1.  0.]
 [ 0.  1. -2.  1.]
 [ 1.  0.  1. -2.]]
```

which is easily verified to have the correct form for a 3-stensil one-dimensional lattice Laplacian with periodic boundary conditions. We may redefine the lattice to have the **'Z'** boundary condition:

```
L = Lattice(shape=(4,), bC='Z', rE=(4,))
```

The output now becomes:

```
[[-2.  1.  0.  0.]
 [ 1. -2.  1.  0.]
 [ 0.  1. -2.  1.]
 [ 0.  0.  1. -2.]]
```

When applied to a small two-dimensional lattice

```
L = Lattice(shape=(2,3),bC='Z',rE=(2,3))
```

the output for the correponding 5-stensil becomes

```
[[-4.  2.  0.  0.  0.  0.]
 [ 2. -4.  2.  0.  0.  0.]
 [ 0.  2. -4.  0.  0.  0.]
 [ 0.  0.  0. -4.  2.  0.]
 [ 0.  0.  0.  2. -4.  2.]
 [ 0.  0.  0.  0.  2. -4.]]
```

We have found such applications of the **matrix**`()` method to be quite educating, and very useful for debugging purposes.

The output matrix can also be used directly as input to all the standard (dense matrix) linear algebra routines in **scipy**. Lattice sizes up to about $10^4$ can be handled in this way, sufficient for most one-dimensional systems (and useful when comparing dense and iterative methods on small higher-dimensional systems).

Methods for generating sparse matrix representations will also be implemented.

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

### B. Example of use

An example illustrating the discussion above is provided by the code snippet:

```
L = Lattice(shape=(2**8, ),
        rE=(18, ), r0=(-9, ))
defF = lambda r: numpy.exp(-r[0]**2/2)
F = LatticeFunction(L, def_F=defF,
        evalF=True)
O = LatticeOperator(L)
O.varOp = O.laplace
F2values = O.varOp(F.values)
```

In this simple case it does not matter if `F2values` is computed by use of `O.linOp`, `O.varOp` or `O.laplace`. The result of evaluating $\Delta_L \exp(-r^2/2)$ can be compared with the exact result, $(r^2 - 1)\exp(-r^2/2)$. A good way to assess the discretization error is to compute $\max_r |\Delta_L F(\boldsymbol{r}) - \Delta F(\boldsymbol{r})|$. This is plotted in Fig. 3 for a range of square lattices.

### C. The lattice Laplace operator

We have used a simple implementation of the lattice Laplacian in the examples above. This is the common $(2d+1)$-stensil approximation. For periodic boundary conditions the implemention is very simple, as indicated by the code snippet below:

```
def laplace(self, phi):
    Lphi = numpy.zeros_like(phi)
    for d in range(self.dim):
        Lphi += numpy.roll(phi, 1,
            axis=d)
        Lphi += numpy.roll(phi,-1,
            axis=d)
        Lphi -= 2*phi
    return Lphi/self.dr**2
```

Here the `roll`-function rotates the entries of the `phi`-array in the `d`-direction by the specified amount ($\pm 1$ for the code above). We have investigated how fast this implementation is.. The results is plotted in Fig. 4. As expected, the evaluation times scales (essentially) linearly will lattice size, with a prefactor which increases with the complexity of the stensil. But, somewhat surprisingly, the evaluation times are not very different from the time to make back-and-forth fast fourier fourier transformations. This suggests an alternative approach, based on fast fourier transforms.

The roll-process is fast, with all loop operations done in NumPy, but requires new memory for the rolled data. To avoid this we have implemented a general method, `stensOp(phi)`. The essential algorithm of this is illustrated by the snippet below:

```
for b in numpy.ndindex(stensil.shape):
    cf, dT, dS = lattice.targetNsource(b)
    phiO[dT] += cf*stensil[b]*phi[dS]
```

Here `stensil` is a (small) `dim`-dimensional array defining the operator in question.

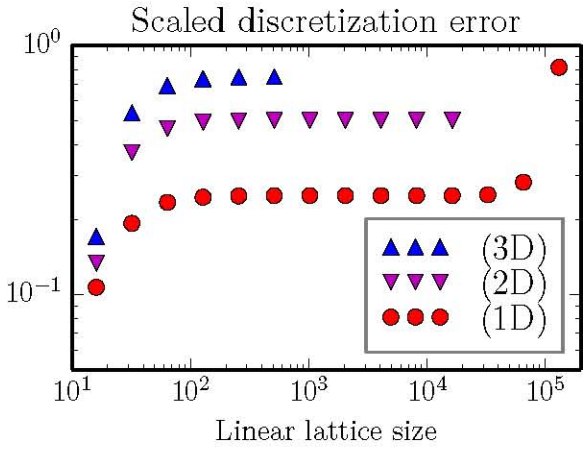


Fig. 3. The maximum absolute difference between the numerical and exact evaluation of the Laplace operator, divided by $dr^2$, as function of the linear lattice size. This shows that the error scales like $dr^2$, as expected for these stensils. The increase in error for large linear size is probably due to numerical roundoff (because $dr^2$ becomes very small), the decrease for small linears size due to incomplete sampling of errors (too few lattice points to compare the functions where the error is maximum).

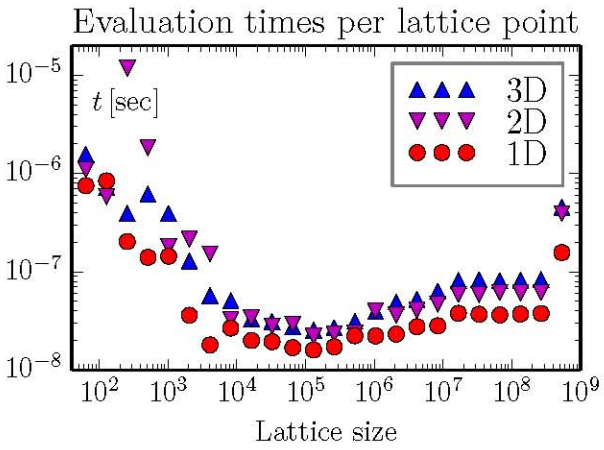


Fig. 4. The times to evaluate $-\Delta_L \phi$, for the (standard) $(2D+1)$-sensil approximation of the Laplace operator, are plotted for various lattice sizes and dimensionalites. As expected, the times increases with the complexity of the stensil. Somewhat surprisingly, the times are not significantly different from the times to perform back-and-forth fast fourier transform (or its discrete analogs), c.f. Fig. 2.

## ACKNOWLEDGMENT

We thank dr. Peder Eliasson (Research manager, SINTEF petroleum Trondheim) for an informative discussion. This work has been partially supported by the UniCQue project.

## REFERENCES

[1] S. van der Walt, S.C. Colbert, and G. Varoquaux, *The NumPy Array: A Structure for Efficient Numerical Computation*, Computing in Science & Engineering **13**, 22–30 (2011)

[2] Eric Jones, Travis Oliphant, Pearu Peterson and others, *SciPy: Open Source Scientific Tools for Python*, `http://www.scipy.org/` (2001)

[3] Travis E. Oliphant, *Python for Scientific Computing*, Computing in Science & Engineering **9**, 90 (2007)

[4] S. Operto, J. Virieux, P. Amestoy, J-Y. L´Excellent, L. Giraud, and H.B.H. Ali, *3D finite-difference frequency-domain modeling of visco-acoustic wave propagation using a massivly parallel direct solver: A feasibility study*, Geophysics **72**, SM195–SM211 (2007)

[5] Peder Eliasson, *3D Full Waveform Modelling in Frequency Domain*, SINTEF Petroleum Research Presentation 2010–09–08.

[6] Amna Noreen and Kåre Olaussen, *A Python Class for Higher-Dimensional Schrödinger Equations*, accepted contribution to ICCS'15, Hong Kong 18–20 March (2015)

[7] Travis E. Oliphant and others, *Numpy reference manual, indexing section*, `http://docs.scipy.org/doc/numpy/reference/arrays.indexing.html`