Shenfun Documentation

Release 2.2.2

Mikael Mortensen

Jun 22, 2020

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Spectral Galerkin

The spectral Galerkin method solves partial differential equations through a special form of the method of weighted residuals (WRM). As a Galerkin method it is very similar to the finite element method (FEM). The most distinguishable feature is that it uses global shape functions, where FEM uses local. This feature leads to highly accurate results with very few shape functions, but the downside is much less flexibility when it comes to computational domain than FEM.

Consider the Poisson equation with a right hand side function $f(\boldsymbol{x})$

$$-\nabla^2 u(\boldsymbol{x}) = f(\boldsymbol{x}) \quad \text{for } \boldsymbol{x} \in \Omega. \tag{1.1}$$

To solve this equation, we will eventually need to supplement appropriate boundary conditions. However, for now just assume that any valid boundary conditions (Dirichlet, Neumann, periodic).

With the method of weighted residuals we attempt to find $u(\boldsymbol{x})$ using an approximation, u_N, to the solution

$$u(\boldsymbol{x}) \approx u_N(\boldsymbol{x}) = \sum_{k=0}^{N-1} \hat{u}_k \phi_k(\boldsymbol{x}). \tag{1.2}$$

Here the N expansion coefficients $\hat{u}_k$ are unknown and $\{\phi_k\}_{k\in\mathcal{I}^N}, \mathcal{I}^N = 0, 1, \ldots, N-1$ are *trial* functions. Inserting for u_N in Eq. (1.1) we get a residual

$$R_N(\boldsymbol{x}) = \nabla^2 u_N(\boldsymbol{x}) + f(\boldsymbol{x}) \neq 0. \tag{1.3}$$

With the WRM we now force this residual to zero in an average sense using *test* function $v(\boldsymbol{x})$ and *weight* function $w(\boldsymbol{x})$

$$(R_N, v)_w := \int_\Omega R_N(\boldsymbol{x})\, \overline{v}(\boldsymbol{x})\, w(\boldsymbol{x}) d\boldsymbol{x} = 0, \tag{1.4}$$

where $\overline{v}$ is the complex conjugate of v. If we now choose the test functions from the same space as the trial functions, i.e., $V_N = span\{\phi_k\}_{k\in\mathcal{I}^N}$, then the WRM becomes the Galerkin method, and we get N equations for N unknowns $\{\hat{u}_k\}_{k\in\mathcal{I}^N}$

$$\sum_{j\in\mathcal{I}^N} \underbrace{\left(-\nabla^2\phi_j, \phi_k\right)_w}_{A_{kj}} \hat{u}_j = \left(f, \phi_k\right)_w, \text{ for } k \in \mathcal{I}^N. \tag{1.5}$$

Note that this is a regular linear system of algebra equations

$$A_{kj}\hat{u}_j = \tilde{f}_k,$$

where the matrix $A \in \mathbb{R}^{N\times N}$.

The choice of basis functions $v(\boldsymbol{x})$ is highly central to the method. For the Galerkin method to be *spectral*, the basis is usually chosen as linear combinations of Chebyshev, Legendre, Laguerre, Hermite, Jacobi or trigonometric functions. In one spatial dimension typical choices for ϕ_k are

$$\begin{aligned}
\phi_k(x) &= T_k(x) \\
\phi_k(x) &= T_k(x) - T_{k+2}(x) \\
\phi_k(x) &= L_k(x) \\
\phi_k(x) &= L_k(x) - L_{k+2}(x) \\
\phi_k(x) &= \exp(\imath k x)
\end{aligned}$$

where T_k, L_k are the k'th Chebyshev polynomial of the first kind and the k'th Legendre polynomial, respectively. Note that the second and fourth functions above satisfy the homogeneous Dirichlet boundary conditions $\phi_k(\pm 1) = 0$, and as such these basis functions may be used to solve the Poisson equation (1.1) with homogeneous Dirichlet boundary conditions. Similarly, two basis functions that satisfy homogeneous Neumann boundary condition $u'(\pm 1) = 0$ are

$$\begin{aligned}
\phi_k &= T_k - \left(\frac{k}{k+2}\right)^2 T_{k+2} \\
\phi_k &= L_k - \frac{k(k+1)}{(k+2)(k+3)} L_{k+2}
\end{aligned}$$

Shenfun contains classes for working with several such bases, to be used for different equations and boundary conditions.

Complete demonstration programs that solves the Poisson equation (1.1), and some other problems can be found by following these links

- *Demo - 1D Poisson's equation*
- *Demo - 3D Poisson's equation*
- *Demo - Helmholtz equation in polar coordinates*
- Demo - Helmholtz equation on the unit sphere
- *Demo - Cubic nonlinear Klein-Gordon equation*
- *Demo - Kuramato-Sivashinsky equation*
- *Demo - Stokes equations*
- *Demo - Lid driven cavity*
- *Demo - Rayleigh Benard*

1.2 Tensor products

If the problem is two-dimensional, then we need two basis functions, one per dimension. If we call the basis function along x-direction $\mathcal{X}(x)$ and along y-direction $\mathcal{Y}(y)$, a test function can then be computed as

$$v(x, y) = \mathcal{X}(x)\mathcal{Y}(y).$$

If we now have a problem that has Dirichlet boundaries in the x-direction and periodic boundaries in the y-direction, then we can choose $\mathcal{X}_k(x) = T_k - T_{k+2}$, for $k \in \mathcal{I}^{N-2}$ (with $N-2$ because T_{k+2} then equals T_N for $k = N-2$), $\mathcal{Y}_l(y) = \exp(\imath l y)$ for $l \in \mathcal{I}^M$ and a tensor product test function is then

$$v_{kl}(x, y) = (T_k(x) - T_{k+2}(x)) \exp(\imath l y), \text{ for } (k, l) \in \mathcal{I}^{N-2} \times \mathcal{I}^M. \tag{1.6}$$

In other words, we choose one test function per spatial dimension and create global basis functions by taking the outer products (or tensor products) of these individual test functions. Since global basis functions simply are the tensor products of one-dimensional basis functions, it is trivial to move to even higher-dimensional spaces. The multi-dimensional basis functions then form a basis for a multi-dimensional tensor product space. The associated domains are similarily formed by taking Cartesian products of the one-dimensional domains. For example, if the one-dimensional domains in x- and y-directions are $[-1, 1]$ and $[0, 2\pi]$, then the two-dimensional domain formed from these two are $[-1, 1] \times [0, 2\pi]$, where $\times$ represents a Cartesian product.

The one-dimensional domains are discretized using the quadrature points of the chosen basis functions. If the meshes in x- and y-directions are $x = \{x_i\}_{i \in \mathcal{I}^N}$ and $y = \{y_j\}_{j \in \mathcal{I}^M}$, then a Cartesian product mesh is $x \times y$. With index and set builder notation it is given as

$$x \times y = \left\{ (x_i, y_j) \,|\, (i, j) \in \mathcal{I}^N \times \mathcal{I}^M \right\}. \tag{1.7}$$

With shenfun a user chooses the appropriate bases for each dimension of the problem, and may then combine these bases into tensor product spaces and Cartesian product domains. For example, to create the required spaces for the aforementioned domain, with Dirichlet in x- and periodic in y-direction, we need the following:

$$\begin{aligned} N, M &= (16, 16) \\ B^N(x) &= \text{span}\{T_k(x) - T_{k+2}(x)\}_{k \in \mathcal{I}^{N-2}} \\ B^M(y) &= \text{span}\{\exp(\imath l y)\}_{l \in \mathcal{I}^M} \\ V(x, y) &= B^N(x) \otimes B^M(y) \end{aligned}$$

where $\otimes$ represents a tensor product.

This can be implemented in *shenfun* as follows:

```
from shenfun import Basis, TensorProductSpace
from mpi4py import MPI
comm = MPI.COMM_WORLD
N, M = (16, 16)
BN = Basis(N, 'Chebyshev', bc=(0, 0))
BM = Basis(M, 'Fourier', dtype='d')
V = TensorProductSpace(comm, (BN, BM))
```

Note that the Chebyshev basis is created using N and not $N - 2$. The chosen boundary condition `bc=(0, 0)` ensures that only $N - 2$ bases will be used. The Fourier basis `BM` has been defined for real inputs to a forward transform, which is ensured by the `dtype` keyword being set to `d` for double. `dtype` specifies the data type that is input to the `forward` method, or the data type of the solution in physical space. Setting `dtype='D'` indicates that this datatype will be complex. Note that it will not trigger an error, or even lead to wrong results, if `dtype` is by mistake set to `D`. It is merely less efficient to work with complex data arrays where double precision is sufficient. See Sec *Getting started* for more information on getting started with using bases.

Shenfun is parallelized with MPI through the mpi4py-fft package. If we store the current example in `filename.py`, then it can be run with more than one processor, e.g., like:

```
mpirun -np 4 python filename.py
```

In this case the tensor product space `V` will be distributed with the *slab* method (since the problem is 2D) and it can here use a maximum of 9 CPUs. The maximum is 9 since the last dimension is transformed from 16 real numbers to 9 complex, using the Hermitian symmetry of real transforms, i.e., the shape of a transformed array in the V space will be (14, 9). You can read more about MPI in the later section *MPI*.

1.3 Tribute

Shenfun is named as a tribute to Prof. Jie Shen, as it contains many tools for working with his modified Chebyshev and Legendre bases, as described here:

- Jie Shen, SIAM Journal on Scientific Computing, 15 (6), 1489-1505 (1994) (JS1)
- Jie Shen, SIAM Journal on Scientific Computing, 16 (1), 74-87, (1995) (JS2)

Shenfun has implemented classes for the bases described in these papers, and within each class there are methods for fast transforms, inner products and for computing matrices arising from bilinear forms in the spectral Galerkin method.

CHAPTER

TWO

GETTING STARTED

2.1 Basic usage

Shenfun consists of classes and functions whoose purpose are to make it easier to implement PDE's with spectral methods in simple tensor product domains. The most important everyday tools are

- `TensorProductSpace`
- `MixedTensorProductSpace`
- `TrialFunction`
- `TestFunction`
- `Function`
- `Array`
- `inner()`
- `div()`
- `grad()`
- `project()`
- `Basis()`

A good place to get started is by creating a `Basis()`. There are six families of bases: Fourier, Chebyshev, Legendre, Laguerre, Hermite and Jacobi. All bases are defined on a one-dimensional domain, with their own basis functions and quadrature points. For example, we have the regular Chebyshev basis $\{T_k\}_{k=0}^{N-1}$, where T_k is the k'th Chebyshev polynomial of the first kind. To create such a basis with 8 quadrature points (i.e., $\{T_k\}_{k=0}^{7}$) do:

```
from shenfun import Basis
N = 8
T = Basis(N, 'Chebyshev', bc=None)
```

Here `bc=None` is used to indicate that there are no boundary conditions associated with this basis, which is the default, so it could just as well have been left out. To create a regular Legendre basis (i.e., $\{L_k\}_{k=0}^{N-1}$, where L_k is the k'th Legendre polynomial), just replace `Chebyshev` with `Legendre` above. And to create a Fourier basis, just use `Fourier`.

The basis $T = \{T_k\}_{k=0}^{N-1}$ has many useful methods associated with it, and we may experiment a little. A `Function` `u` using basis T has expansion

$$u(x) = \sum_{k=0}^{7} \hat{u}_k T_k(x) \tag{2.1}$$

and an instance of this function (initialized with $\{\hat{u}_k\}_{k=0}^{7} = 0$) is created in shenfun as:

```
from shenfun import Function
u = Function(T)
```

Consider now for exampel the polynomial $2x^2 - 1$, which happens to be exactly equal to $T_2(x)$. We can create this polynomial using sympy

```
import sympy as sp
x = sp.Symbol('x')
u = 2*x**2 - 1  # or simply u = sp.chebyshevt(2, x)
```

The Sympy function `u` can now be evaluated on the quadrature points of basis T:

```
from shenfun import Array
xj = T.mesh()
ue = Array(T)
ue[:] = [u.subs(x, xx) for xx in xj]
print(xj)
  [ 0.98078528  0.83146961  0.55557023  0.19509032 -0.19509032 -0.55557023
   -0.83146961 -0.98078528]
print(ue)
  [ 0.92387953  0.38268343 -0.38268343 -0.92387953 -0.92387953 -0.38268343
    0.38268343  0.92387953]
```

We see that `ue` is an `Array` on the basis `T`, and not a `Function`. The `Array` and `Function` classes are both subclasses of Numpy's ndarray, and represent the two arrays associated with the spectral Galerkin function, like (2.1). The `Function` represent the entire spectral Galerkin function, with array values corresponding to the expansion coefficients $\hat{u}$. The `Array` represent the spectral Galerkin function evaluated on the quadrature mesh of the basis `T`, i.e., here $u(x_i), \forall\, i \in 0, 1, \ldots, 7$.

We now want to find the `Function` `uh` corresponding to `Array` `ue`. Considering (2.1), this corresponds to finding $\hat{u}_k$ if the left hand side $u(x_j)$ is known for all quadrature points x_j.

Since we already know that `ue` is equal to the second Chebyshev polynomial, we should get an array of expansion coefficients equal to $\hat{u} = (0, 0, 1, 0, 0, 0, 0, 0)$. We can compute `uh` either by using `project()` or a forward transform:

```
from shenfun import project
uh = Function(T)
uh = T.forward(ue, uh)
# or
# uh = ue.forward(uh)
# or
# uh = project(ue, T)
print(uh)
  [-1.38777878e-17  6.72002101e-17  1.00000000e+00 -1.95146303e-16
    1.96261557e-17  1.15426347e-16 -1.11022302e-16  1.65163507e-16]
```

So we see that the projection works to machine precision.

The projection is mathematically: find $u_h \in T$, such that

$$(u_h - u, v)_w = 0 \quad \forall v \in T,$$

where v is a test function, u_h is a trial function and the notation $(\cdot, \cdot)_w$ was introduced in (1.4). Using now $v = T_k$ and

$u_h = \sum_{j=0}^{7} \hat{u}_j T_j$, we get

$$(\sum_{j=0}^{7} \hat{u}_j T_j, T_k)_w = (u, T_k)_w,$$
$$\sum_{j=0}^{7} (T_j, T_k)_w \hat{u}_j = (u, T_k)_w,$$

for all $k \in 0, 1, \ldots, 7$. This can be rewritten on matrix form as

$$B_{kj}\hat{u}_j = \tilde{u}_k,$$

where $B_{kj} = (T_j, T_k)_w$, $\tilde{u}_k = (u, T_k)_w$ and summation is implied by the repeating j indices. Since the Chebyshev polynomials are orthogonal the mass matrix B_{kj} is diagonal. We can assemble both B_{kj} and $\tilde{u}_j$ with shenfun, and at the same time introduce the `TestFunction`, `TrialFunction` classes and the `inner()` function:

```
from shenfun import TestFunction, TrialFunction, inner
u = TrialFunction(T)
v = TestFunction(T)
B = inner(u, v)
u_tilde = inner(ue, v)
print(B)
  {0: array([3.14159265, 1.57079633, 1.57079633, 1.57079633, 1.57079633,
   1.57079633, 1.57079633, 1.57079633])}
print(u_tilde)
  [-4.35983562e-17  1.05557843e-16  1.57079633e+00 -3.06535096e-16
    3.08286933e-17  1.81311282e-16 -1.74393425e-16  2.59438230e-16]
```

The `inner()` function represents the (weighted) inner product and it expects one test function, and possibly one trial function. If, as here, it also contains a trial function, then a matrix is returned. If `inner()` contains one test, but no trial function, then an array is returned. Finally, if `inner()` contains no test nor trial function, but instead a number and an `Array`, like:

```
a = Array(T, val=1)
print(inner(1, a))
  2.0
```

then `inner()` represents a non-weighted integral over the domain. Here it returns the length of the domain (2.0) since *a* is initialized to unity.

Note that the matrix B assembled above is stored using shenfun's `SpectralMatrix` class, which is a subclass of Python's dictionary, where the keys are the diagonals and the values are the diagonal entries. The matrix B is seen to have only one diagonal (the principal) $\{B_{ii}\}_{i=0}^{7}$.

With the matrix comes a *solve* method and we can solve for $\hat{u}$ through:

```
u_hat = Function(T)
u_hat = B.solve(u_tilde, u=u_hat)
print(u_hat)
  [-1.38777878e-17  6.72002101e-17  1.00000000e+00 -1.95146303e-16
    1.96261557e-17  1.15426347e-16 -1.11022302e-16  1.65163507e-16]
```

which obviously is exactly the same as we found using `project()` or the *T.forward* function.

Note that `Array` merely is a subclass of Numpy's `ndarray`, whereas `Function` is a subclass of both Numpy's `ndarray` *and* the `BasisFunction` class. The latter is used as a base class for arguments to bilinear and linear forms, and is as such a base class also for `TrialFunction` and `TestFunction`. An instance of the `Array` class cannot be used in forms, except from regular inner products of numbers or test function vs an `Array`. To illustrate, lets create some forms, where all except the last one is ok:

```
from shenfun import Dx
T = Basis(12, 'Legendre')
u = TrialFunction(T)
v = TestFunction(T)
uf = Function(T)
ua = Array(T)
A = inner(v, u)    # Mass matrix
c = inner(v, ua)   # ok, a scalar product
d = inner(v, uf)   # ok, a scalar product (slower than above)
e = inner(1, ua)   # ok, non-weighted integral of ua over domain
df = Dx(uf, 0, 1) # ok
da = Dx(ua, 0, 1) # Not ok

    AssertionError                            Traceback (most recent call last)
    <ipython-input-14-3b957937279f> in <module>
    ----> 1 da = inner(v, Dx(ua, 0, 1))

    ~/MySoftware/shenfun/shenfun/forms/operators.py in Dx(test, x, k)
         82         Number of derivatives
         83     """
    ---> 84     assert isinstance(test, (Expr, BasisFunction))
         85
         86     if isinstance(test, BasisFunction):

    AssertionError:
```

So it is not possible to perform operations that involve differentiation (*Dx* represents a partial derivative) on an `Array` instance. This is because the `ua` does not contain more information than its values and its TensorProductSpace. A `BasisFunction` instance, on the other hand, can be manipulated with operators like `div() grad()` in creating instances of the `Expr` class, see *Operators*.

Note that any rules for efficient use of Numpy `ndarrays`, like vectorization, also applies to `Function` and `Array` instances.

2.2 Operators

Operators act on any single instance of a `BasisFunction`, which can be `Function`, `TrialFunction` or `TestFunction`. The implemented operators are:

- `div()`
- `grad()`
- `curl()`
- `Dx()`

Operators are used in variational forms assembled using `inner()` or `project()`, like:

```
A = inner(grad(u), grad(v))
```

which assembles a stiffness matrix A. Note that the two expressions fed to inner must have consistent rank. Here, for example, both `grad(u)` and `grad(v)` have rank 1 of a vector.

2.3 Multidimensional problems

As described in the introduction, a multidimensional problem is handled using tensor product spaces, that have basis functions generated from taking the outer products of one-dimensional basis functions. We create tensor product spaces using the class `TensorProductSpace`:

```
N, M = (12, 16)
C0 = Basis(N, 'L', bc=(0, 0), scaled=True)
K0 = Basis(M, 'F', dtype='d')
T = TensorProductSpace(comm, (C0, K0))
```

Associated with this is a Cartesian mesh $[-1, 1] \times [0, 2\pi]$. We use classes `Function`, `TrialFunction` and `TestFunction` exactly as before:

```
u = TrialFunction(T)
v = TestFunction(T)
A = inner(grad(u), grad(v))
```

However, now `A` will be a tensor product matrix, or more correctly, the sum of two tensor product matrices. This can be seen if we look at the equations beyond the code. In this case we are using a composite Legendre basis for the first direction and Fourier exponentials for the second, and the tensor product basis function is

$$\begin{aligned} v_{kl}(x, y) &= \frac{1}{\sqrt{4k+6}}(L_k(x) - L_{k+2}(x)) \exp(\imath l y), \\ &= \Psi_k(x)\phi_l(y), \end{aligned}$$

where L_k is the k'th Legendre polynomial, $\psi_k = (L_k - L_{k+2})/\sqrt{4k+6}$ and $\phi_l = \exp(\imath l y)$ are used for simplicity in later derivations. The trial function becomes

$$u(x, y) = \sum_k \sum_l \hat{u}_{kl} v_{kl}$$

and the inner product is

$$\begin{aligned} (\nabla u, \nabla v)_w &= \int_{-1}^{1}\int_{0}^{2\pi} \nabla u \cdot \nabla v dx dy, \\ &= \int_{-1}^{1}\int_{0}^{2\pi} \frac{\partial u}{\partial x}\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\frac{\partial v}{\partial y} dx dy, \\ &= \int_{-1}^{1}\int_{0}^{2\pi} \frac{\partial u}{\partial x}\frac{\partial v}{\partial x} dx dy + \int_{-1}^{1}\int_{0}^{2\pi} \frac{\partial u}{\partial y}\frac{\partial v}{\partial y} dx dy, \end{aligned} \tag{2.2}$$

showing that it is the sum of two tensor product matrices. However, each one of these two terms contains the outer product of smaller matrices. To see this we need to insert for the trial and test functions (using v_{mn} for test):

$$\begin{aligned} \int_{-1}^{1}\int_{0}^{2\pi} \frac{\partial u}{\partial x}\frac{\partial v}{\partial x} dx dy &= \int_{-1}^{1}\int_{0}^{2\pi} \frac{\partial}{\partial x}\left(\sum_k \sum_l \hat{u}_{kl}\Psi_k(x)\phi_l(y)\right) \frac{\partial}{\partial x}\left(\Psi_m(x)\phi_n(y)\right) dx dy, \\ &= \sum_k \sum_l \underbrace{\int_{-1}^{1} \frac{\partial \Psi_k(x)}{\partial x}\frac{\partial \Psi_m(x)}{\partial x} dx}_{A_{mk}} \underbrace{\int_{0}^{2\pi} \phi_l(y)\phi_n(y) dy}_{B_{nl}} \hat{u}_{kl}, \end{aligned}$$

where $A \in \mathbb{R}^{N-2 \times N-2}$ and $B \in \mathbb{R}^{M \times M}$. The tensor product matrix $A_{mk}B_{nl}$ (or in matrix notation $A \otimes B$) is the first item of the two items in the list that is returned by `inner(grad(u), grad(v))`. The other item is of course

the second term in the last line of (2.2):

$$\int_{-1}^{1}\int_{0}^{2\pi}\frac{\partial u}{\partial y}\frac{\partial v}{\partial y}dxdy = \int_{-1}^{1}\int_{0}^{2\pi}\frac{\partial}{\partial y}\left(\sum_k\sum_l \hat{u}_{kl}\Psi_k(x)\phi_l(y)\right)\frac{\partial}{\partial y}\left(\Psi_m(x)\phi_n(y)\right)dxdy$$
$$= \sum_k\sum_l \underbrace{\int_{-1}^{1}\Psi_k(x)\Psi_m(x)dx}_{C_{mk}}\underbrace{\int_{0}^{2\pi}\frac{\partial\phi_l(y)}{\partial y}\frac{\phi_n(y)}{\partial y}dy}_{D_{nl}}\hat{u}_{kl}$$

The tensor product matrices $A_{mk}B_{nl}$ and $C_{mk}D_{nl}$ are both instances of the `TPMatrix` class. Together they lead to linear algebra systems like:

$$(A_{mk}B_{nl} + C_{mk}D_{nl})\hat{u}_{kl} = \tilde{f}_{mn}, \tag{2.3}$$

where

$$\tilde{f}_{mn} = (v_{mn}, f)_w,$$

for some right hand side f, see, e.g., (2.10). Note that an alternative formulation here is

$$A\hat{u}B^T + C\hat{u}D^T = \tilde{f}$$

where $\hat{u}$ and $\tilde{f}$ are treated as regular matrices ($\hat{u} \in \mathbb{R}^{N-2\times M}$ and $\tilde{f} \in \mathbb{R}^{N-2\times M}$). This formulation is utilized to derive efficient solvers for tensor product bases in multiple dimensions using the matrix decomposition method in [She94] and [She95].

Note that in our case the equation system (2.3) can be greatly simplified since three of the submatrices (A_{mk}, B_{nl} and D_{nl}) are diagonal. Even more, two of them equal the identity matrix

$$A_{mk} = \delta_{mk},$$
$$B_{nl} = \delta_{nl},$$

whereas the last one can be written in terms of the identity (no summation on repeating indices)

$$D_{nl} = -nl\delta_{nl}.$$

Inserting for this in (2.3) and simplifying by requiring that $l = n$ in the second step, we get

$$\begin{aligned}(\delta_{mk}\delta_{nl} - lnC_{mk}\delta_{nl})\hat{u}_{kl} &= \tilde{f}_{mn},\\ (\delta_{mk} - l^2C_{mk})\hat{u}_{kl} &= \tilde{f}_{ml}.\end{aligned} \tag{2.4}$$

Now if we keep l fixed this latter equation is simply a regular linear algebra problem to solve for $\hat{u}_{kl}$, for all k. Of course, this solve needs to be carried out for all l.

Note that there is a generic solver available for the system (2.3) in `SolverGeneric2NP` that makes no assumptions on diagonality. However, this solver will, naturally, be quite a bit slower than a tailored solver that takes advantage of diagonality. For the Poisson equation such solvers are available for both Legendre and Chebyshev bases, see the extended demo *Demo - 3D Poisson's equation* or the demo programs dirichlet_poisson2D.py and dirichlet_poisson3D.py.

2.4 Curvilinear coordinates

Shenfun can be used to solve equations using curvilinear coordinates, like polar, cylindrical and spherical coordinates. The feature was added April 2020, and is still rather experimental. The curvilinear coordinates are defined by the user, who needs to provide a map, i.e., the position vector, between new coordinates and the Cartesian coordinates. The

basis functions of the new coordinates need not be orthogonal, but non-orthogonal is not widely tested so use with care. In shenfun we use non-normalized natural (covariant) basis vectors. For this reason the equations may look a little bit different than usual. For example, in cylindrical coordinates we have the position vector

$$\mathbf{r} = r\cos\theta\,\mathbf{i} + r\sin\theta\,\mathbf{j} + z\,\mathbf{k}, \tag{2.5}$$

where $\mathbf{i}, \mathbf{j}, \mathbf{k}$ are the Cartesian unit vectors and r, θ, z are the new coordinates. The covariant basis vectors are then

$$\begin{aligned}\mathbf{b}_r &= \frac{\partial \mathbf{r}}{\partial r},\\ \mathbf{b}_\theta &= \frac{\partial \mathbf{r}}{\partial \theta},\\ \mathbf{b}_z &= \frac{\partial \mathbf{r}}{\partial z},\end{aligned} \tag{2.6}$$

leading to

$$\begin{aligned}\mathbf{b}_r &= \cos(\theta)\,\mathbf{i} + \sin(\theta)\,\mathbf{j},\\ \mathbf{b}_\theta &= -r\sin(\theta)\,\mathbf{i} + r\cos(\theta)\,\mathbf{j},\\ \mathbf{b}_z &= \mathbf{k}.\end{aligned} \tag{2.7}$$

We see that $|\mathbf{b}_\theta| = r$ and not unity.

A vector $\mathbf{u}$ in this basis is given as

$$\mathbf{u} = u^r\mathbf{b}_r + u^\theta\mathbf{b}_\theta + u^z\mathbf{b}_z, \tag{2.8}$$

and the vector Laplacian $\nabla^2\mathbf{u}$ is

$$\begin{aligned}&\left(\frac{\partial^2 u^r}{\partial^2 r} + \frac{1}{r}\frac{\partial u^r}{\partial r} + \frac{1}{r^2}\frac{\partial^2 u^r}{\partial^2\theta} - \frac{2}{r}\frac{\partial u^\theta}{\partial\theta} - \frac{1}{r^2}u^r + \frac{\partial^2 u^r}{\partial^2 z}\right)\mathbf{b}_r\\ &+\left(\frac{\partial^2 u^\theta}{\partial^2 r} + \frac{3}{r}\frac{\partial u^\theta}{\partial r} + \frac{2}{r^3}\frac{\partial u^r}{\partial\theta} + \frac{1}{r^2}\frac{\partial^2 u^\theta}{\partial^2\theta} + \frac{\partial^2 u^\theta}{\partial^2 z}\right)\mathbf{b}_\theta\\ &+\left(\frac{\partial^2 u^z}{\partial^2 r} + \frac{1}{r}\frac{\partial u^z}{\partial r} + \frac{1}{r^2}\frac{\partial^2 u^z}{\partial^2\theta} + \frac{\partial^2 u^z}{\partial^2 z}\right)\mathbf{b}_z.\end{aligned} \tag{2.9}$$

which is slightly different from what you see in most textbooks, which are using normalized basis vectors.

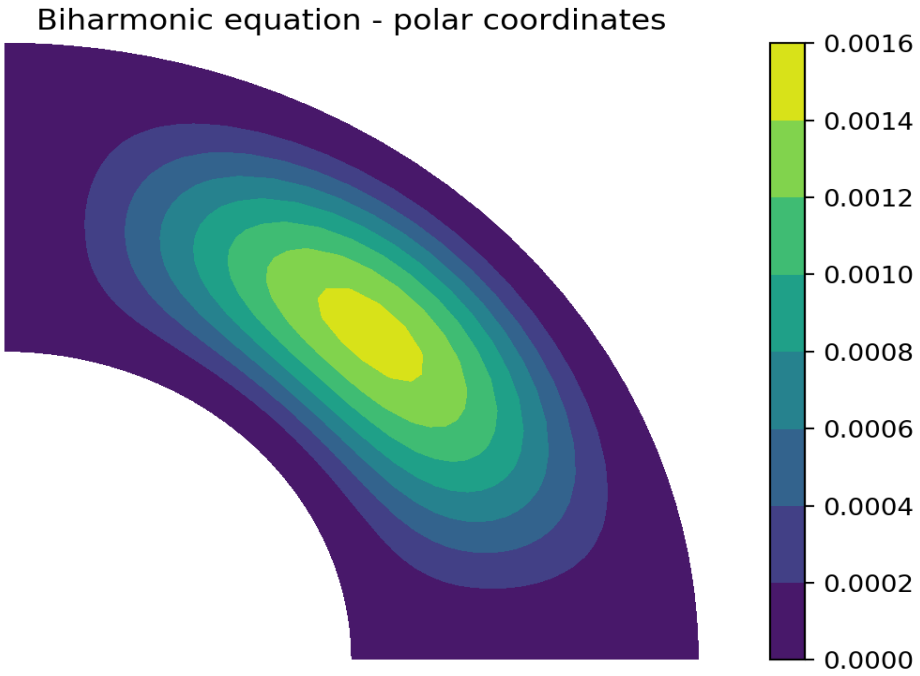
Note that once the curvilinear map has been created, shenfun's operators `div()`, `grad()` and `curl()` work out of the box with no additional effort. So you do not have to implement messy equations that look like (2.9) directly. Take the example with cylindrical coordinates. The vector Laplacian can be implemented as:

```
from shenfun import *
import sympy as sp

r, theta, z = psi = sp.symbols('x,y,z', real=True, positive=True)
rv = (r*sp.cos(theta), r*sp.sin(theta), z)

N = 10
F0 = Basis(N, 'F', dtype='d')
F1 = Basis(N, 'F', dtype='D')
L = Basis(N, 'L', domain=(0, 1))
T = TensorProductSpace(comm, (L, F1, F0), coordinates=(psi, rv))
V = VectorTensorProductSpace(T)
u = TrialFunction(V)
du = div(grad(u))
```

There are currently curvilinear demos for solving both Helmholtz's equation and the biharmonic equation on a circular disc, a solver for 3D Poisson equation in a pipe, and a solver for the biharmonic equation on a part of the disc. Also, the Helmholtz equation solved on the unit sphere using spherical coordinates is shown here, and on the torus here. A solution from solving the biharmonic equation with homogeneous Dirichlet boundary conditions on $(\theta, r) \in [0, \pi/2] \times [0.5, 1]$ is shown below.



2.5 Coupled problems

With Shenfun it is possible to solve equations coupled and implicit using the `MixedTensorProductSpace` class for multidimensional problems and `MixedBasis` for one-dimensional problems. As an example, lets consider a mixed formulation of the Poisson equation. The Poisson equation is given as always as

$$\nabla^2 u(\boldsymbol{x}) = f(\boldsymbol{x}), \quad \text{for} \quad \boldsymbol{x} \in \Omega, \tag{2.10}$$

but now we recast the problem into a mixed formulation

$$\begin{aligned} \sigma(\boldsymbol{x}) - \nabla u(\boldsymbol{x}) &= 0, \quad \text{for} \quad \boldsymbol{x} \in \Omega, \\ \nabla \cdot \sigma(\boldsymbol{x}) &= f(\boldsymbol{x}), \quad \text{for} \quad \boldsymbol{x} \in \Omega. \end{aligned}$$

where we solve for the vector σ and scalar u simultaneously. The domain Ω is taken as a multidimensional Cartesian product $\Omega = [-1, 1] \times [0, 2\pi]$, but the code is more or less identical for a 3D problem. For boundary conditions we use Dirichlet in the x-direction and periodicity in the y-direction:

$$\begin{aligned} u(\pm 1, y) &= 0 \\ u(x, 2\pi) &= u(x, 0) \end{aligned}$$

Note that there is no boundary condition on σ, only on u. For this reason we choose a Dirichlet basis SD for u and a regular Legendre or Chebyshev ST basis for σ. With $K0$ representing the function space in the periodic direction,

we get the relevant 2D tensor product spaces as $TD = SD \otimes K0$ and $TT = ST \otimes K0$. Since σ is a vector we use a `VectorTensorProductSpace` $VT = TT \times TT$ and finally a `MixedTensorProductSpace` $Q = VT \times TD$ for the coupled and implicit treatment of (σ, u):

```
from shenfun import VectorTensorProductSpace, MixedTensorProductSpace
N, M = (16, 24)
family = 'Legendre'
SD = Basis(N[0], family, bc=(0, 0))
ST = Basis(N[0], family)
K0 = Basis(N[1], 'Fourier', dtype='d')
TD = TensorProductSpace(comm, (SD, K0), axes=(0, 1))
TT = TensorProductSpace(comm, (ST, K0), axes=(0, 1))
VT = VectorTensorProductSpace(TT)
Q = MixedTensorProductSpace([VT, TD])
```

In variational form the problem reads: find $(\sigma, u) \in Q$ such that

$$\begin{aligned}(\sigma, \tau)_w - (\nabla u, \tau)_w &= 0, \quad \forall \tau \in VT, \\ (\nabla \cdot \sigma, v)_w &= (f, v)_w \quad \forall v \in TD\end{aligned} \tag{2.11}$$

To implement this we use code that is very similar to regular, uncoupled problems. We create test and trialfunction:

```
gu = TrialFunction(Q)
tv = TestFunction(Q)
sigma, u = gu
tau, v = tv
```

and use these to assemble all blocks of the variational form (2.11):

```
# Assemble equations
A00 = inner(sigma, tau)
if family.lower() == 'legendre':
    A01 = inner(u, div(tau))
else:
    A01 = inner(-grad(u), tau)
A10 = inner(div(sigma), v)
```

Note that we here can use integration by parts for Legendre, since the weight function is a constant, and as such get the term $(-\nabla u, \tau)_w = (u, \nabla \cdot \tau)_w$ (boundary term is zero due to homogeneous Dirichlet boundary conditions).

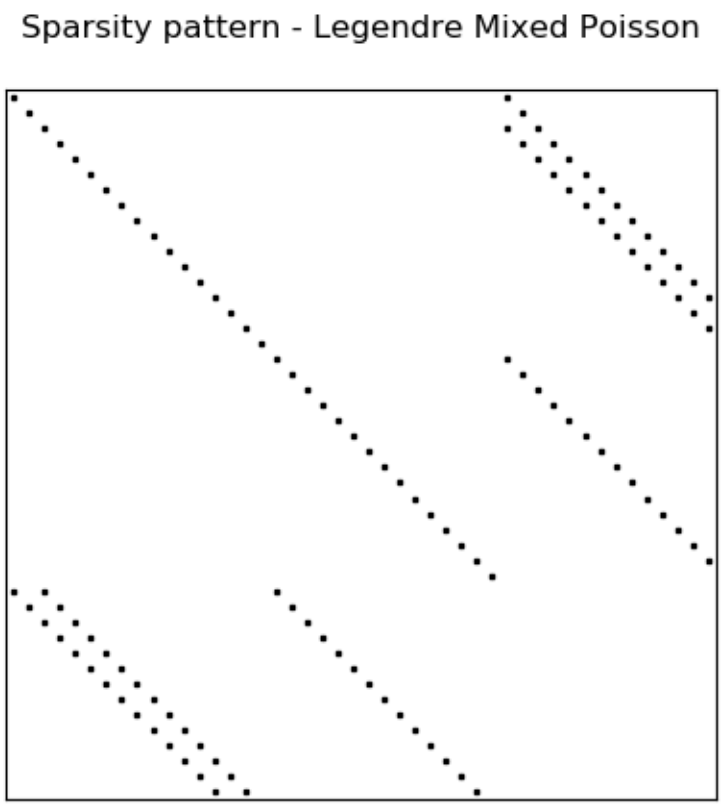
We collect all assembled terms in a `BlockMatrix`:

```
from shenfun import BlockMatrix
H = BlockMatrix(A00+A01+A10)
```

This block matrix `H` is then simply (for Legendre)

$$\begin{bmatrix} (\sigma, \tau)_w & (u, \nabla \cdot \tau)_w \\ (\nabla \cdot \sigma, v)_w & 0 \end{bmatrix} \tag{2.12}$$

Note that each item in (2.12) is a collection of instances of the `TPMatrix` class, and for similar reasons as given around (2.4), we get also here one regular block matrix for each Fourier wavenumber. The sparsity pattern is the same for all matrices except for wavenumber 0. The (highly sparse) sparsity pattern for block matrix H with wavenumber $\neq 0$ is shown in the image below



A complete demo for the coupled problem discussed here can be found in MixedPoisson.py and a 3D version is in MixedPoisson3D.py.

2.6 Integrators

The `integrators` module contains some interator classes that can be used to integrate a solution forward in time. However, for now these integrators are only implemented for purely Fourier tensor product spaces. There are currently 3 different integrator classes

- `RK4`: Runge-Kutta fourth order
- `ETD`: Exponential time differencing Euler method
- `ETDRK4`: Exponential time differencing Runge-Kutta fourth order

See, e.g., H. Montanelli and N. Bootland "Solving periodic semilinear PDEs in 1D, 2D and 3D with exponential integrators", https://arxiv.org/pdf/1604.08900.pdf

Integrators are set up to solve equations like

$$\frac{\partial u}{\partial t} = Lu + N(u) \tag{2.13}$$

where u is the solution, L is a linear operator and $N(u)$ is the nonlinear part of the right hand side.

To illustrate, we consider the time-dependent 1-dimensional Kortveeg-de Vries equation

$$\frac{\partial u}{\partial t} + \frac{\partial^3 u}{\partial x^3} + u\frac{\partial u}{\partial x} = 0$$

which can also be written as

$$\frac{\partial u}{\partial t} + \frac{\partial^3 u}{\partial x^3} + \frac{1}{2}\frac{\partial u^2}{\partial x} = 0$$

We neglect boundary issues and choose a periodic domain $[0, 2\pi]$ with Fourier exponentials as test functions. The initial condition is chosen as

$$u(x, t=0) = 3A^2/\cosh(0.5A(x-\pi+2))^2 + 3B^2/\cosh(0.5B(x-\pi+1))^2 \tag{2.14}$$

where A and B are constants. For discretization in space we use the basis $V_N = span\{exp(\imath kx)\}_{k=0}^{N}$ and formulate the variational problem: find $u \in V_N$ such that

$$\frac{\partial}{\partial t}\Big(u, v\Big) = -\Big(\frac{\partial^3 u}{\partial x^3}, v\Big) - \Big(\frac{1}{2}\frac{\partial u^2}{\partial x}, v\Big), \quad \forall v \in V_N$$

We see that the first term on the right hand side is linear in u, whereas the second term is nonlinear. To implement this problem in shenfun we start by creating the necessary basis and test and trial functions

```
import numpy as np
from shenfun import *

N = 256
T = Basis(N, 'F', dtype='d')
u = TrialFunction(T)
v = TestFunction(T)
u_ = Array(T)
u_hat = Function(T)
```

We then create two functions representing the linear and nonlinear part of (2.13):

```
def LinearRHS(**params):
    return -inner(Dx(u, 0, 3), v)

k = T.wavenumbers(scaled=True, eliminate_highest_freq=True)
def NonlinearRHS(u, u_hat, rhs, **params):
    rhs.fill(0)
    u_[:] = T.backward(u_hat, u_)
    rhs = T.forward(-0.5*u_**2, rhs)
    return rhs*1j*k    # return inner(grad(-0.5*Up**2), v)
```

Note that we differentiate in `NonlinearRHS` by using the wavenumbers `k` directly. Alternative notation, that is given in commented out text, is slightly slower, but the results are the same.

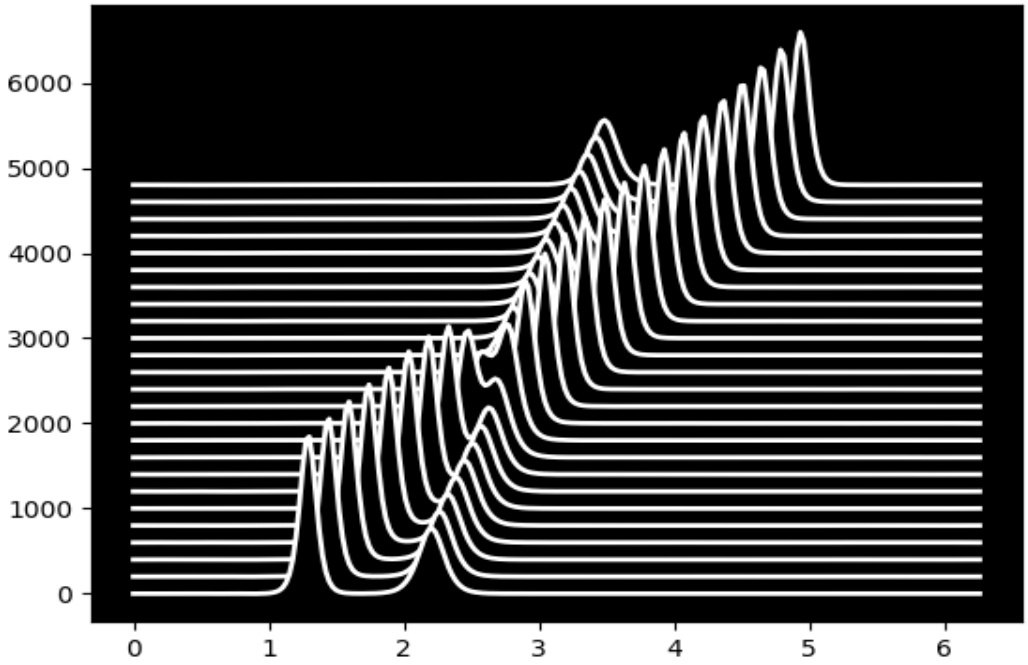
The solution vector `u_` needs also to be initialized according to (2.14)

```
A = 25.
B = 16.
x = T.points_and_weights()[0]
u_[:] = 3*A**2/np.cosh(0.5*A*(x-np.pi+2))**2 + 3*B**2/np.cosh(0.5*B*(x-np.pi+1))**2
u_hat = T.forward(u_, u_hat)
```

Finally we create an instance of the `ETDRK4` solver, and integrate forward with a given timestep

```
dt = 0.01/N**2
end_time = 0.006
integrator = ETDRK4(T, L=LinearRHS, N=NonlinearRHS)
integrator.setup(dt)
u_hat = integrator.solve(u_, u_hat, dt, (0, end_time))
```

The solution is two waves travelling through eachother, seemingly undisturbed.



2.7 MPI

Shenfun makes use of the Message Passing Interface (MPI) to solve problems on distributed memory architectures. OpenMP is also possible to enable for FFTs.

Dataarrays in Shenfun are distributed using a new and completely generic method, that allows for any index of a multidimensional array to be distributed. To illustrate, lets consider a `TensorProductSpace` of three dimensions, such that the arrays living in this space will be 3-dimensional. We create two spaces that are identical, except from the MPI decomposition, and we use 4 CPUs (`mpirun -np 4 python mpitest.py`, if we store the code in this section as `mpitest.py`):

```
from shenfun import *
from mpi4py import MPI
from mpi4py_fft import generate_xdmf
comm = MPI.COMM_WORLD
N = (20, 40, 60)
K0 = Basis(N[0], 'F', dtype='D', domain=(0, 1))
K1 = Basis(N[1], 'F', dtype='D', domain=(0, 2))
K2 = Basis(N[2], 'F', dtype='d', domain=(0, 3))
```

(continues on next page)

(continued from previous page)

```
T0 = TensorProductSpace(comm, (K0, K1, K2), axes=(0, 1, 2), slab=True)
T1 = TensorProductSpace(comm, (K0, K1, K2), axes=(1, 0, 2), slab=True)
```

Here the keyword `slab` determines that only *one* index set of the 3-dimensional arrays living in `T0` or `T1` should be distributed. The defaul is to use two, which corresponds to a so-called pencil decomposition. The `axes`-keyword determines the order of which transforms are conducted, starting from last to first in the given tuple. Note that `T0` now will give arrays in real physical space that are distributed in the first index, whereas `T1` will give arrays that are distributed in the second. This is because 0 and 1 are the first items in the tuples given to `axes`.

We can now create some Arrays on these spaces:

```
u0 = Array(T0, val=comm.Get_rank())
u1 = Array(T1, val=comm.Get_rank())
```

such that `u0` and `u1` have values corresponding to their communicating processors rank in the `COMM_WORLD` group (the group of all CPUs).

Note that both the TensorProductSpaces have functions with expansion

$$u(x, y, z) = \sum_{n=-N/2}^{N/2-1} \sum_{m=-N/2}^{N/2-1} \sum_{l=-N/2}^{N/2-1} \hat{u}_{l,m,n} e^{\imath(lx+my+nz)}. \tag{2.15}$$

where $u(x, y, z)$ is the continuous solution in real physical space, and $\hat{u}$ are the spectral expansion coefficients. If we evaluate expansion (2.15) on the real physical mesh, then we get

$$u(x_i, y_j, z_k) = \sum_{n=-N/2}^{N/2-1} \sum_{m=-N/2}^{N/2-1} \sum_{l=-N/2}^{N/2-1} \hat{u}_{l,m,n} e^{\imath(lx_i+my_j+nz_k)}. \tag{2.16}$$

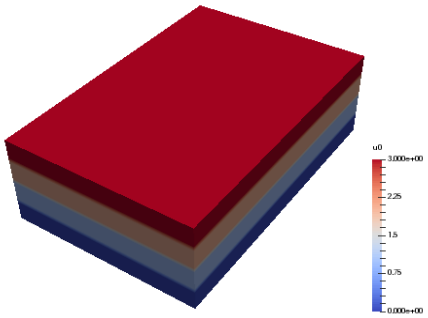
The function $u(x_i, y_j, z_k)$ corresponds to the arrays `u0, u1`, whereas we have not yet computed the array $\hat{u}$. We could get $\hat{u}$ as:

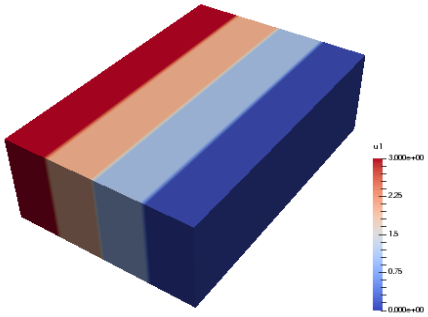
```
u0_hat = Function(T0)
u0_hat = T0.forward(u0, u0_hat)
```

Now, `u0` and `u1` have been created on the same mesh, which is a structured mesh of shape $(20, 40, 60)$. However, since they have different MPI decomposition, the values used to fill them on creation will differ. We can visualize the arrays in Paraview using some postprocessing tools, to be further described in Sec Postprocessing:

```
u0.write('myfile.h5', 'u0', 0, domain=T0.mesh())
u1.write('myfile.h5', 'u1', 0, domain=T1.mesh())
if comm.Get_rank() == 0:
    generate_xdmf('myfile.h5')
```

And when the generated `myfile.xdmf` is opened in Paraview, we can see the different distributions. The function `u0` is shown first, and we see that it has different values along the short first dimension. The second figure is evidently distributed along the second dimension. Both arrays are non-distributed in the third and final dimension, which is fortunate, because this axis will be the first to be transformed in, e.g., `u0_hat = T0.forward(u0, u0_hat)`.

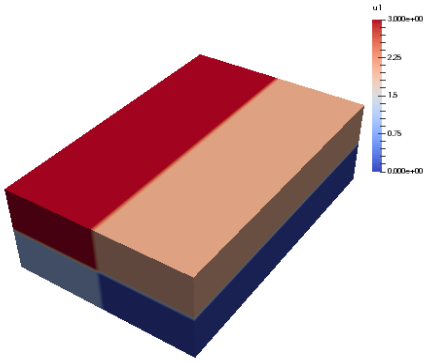




We can now decide to distribute not just one, but the first two axes using a pencil decomposition instead. This is achieved simply by dropping the slab keyword:

```
T2 = TensorProductSpace(comm, (K0, K1, K2), axes=(0, 1, 2))
u2 = Array(T2, val=comm.Get_rank())
u2.write('pencilfile.h5', 'u2', 0)
if comm.Get_rank() == 0:
    generate_xdmf('pencilfile.h5')
```

Running again with 4 CPUs the array `u2` will look like:



The local slices into the global array may be obtained through:

```
>>> print(comm.Get_rank(), T2.local_slice(False))
0 [slice(0, 10, None), slice(0, 20, None), slice(0, 60, None)]
```

(continues on next page)

(continued from previous page)

```
1 [slice(0, 10, None), slice(20, 40, None), slice(0, 60, None)]
2 [slice(10, 20, None), slice(0, 20, None), slice(0, 60, None)]
3 [slice(10, 20, None), slice(20, 40, None), slice(0, 60, None)]
```

In spectral space the distribution will be different. This is because the discrete Fourier transforms are performed one axis at the time, and for this to happen the dataarrays need to be realigned to get entire axis available for each processor. Naturally, for the array in the pencil example *(see image)*, we can only perform an FFT over the third and longest axis, because only this axis is locally available to all processors. To do the other directions, the dataarray must be realigned and this is done internally by the `TensorProductSpace` class. The shape of the datastructure in spectral space, that is the shape of $\hat{u}$, can be obtained as:

```
>>> print(comm.Get_rank(), T2.local_slice(True))
0 [slice(0, 20, None), slice(0, 20, None), slice(0, 16, None)]
1 [slice(0, 20, None), slice(0, 20, None), slice(16, 31, None)]
2 [slice(0, 20, None), slice(20, 40, None), slice(0, 16, None)]
3 [slice(0, 20, None), slice(20, 40, None), slice(16, 31, None)]
```

Evidently, the spectral space is distributed in the last two axes, whereas the first axis is locally avalable to all processors. Tha dataarray is said to be aligned in the first dimension.

CHAPTER

THREE

POST PROCESSING

MPI is great because it means that you can run Shenfun on pretty much as many CPUs as you can get your hands on. However, MPI makes it more challenging to do visualization, in particular with Python and Matplotlib. For this reason there is a `utilities` module with helper classes for dumping dataarrays to HDF5 or NetCDF

Most of the IO has already been implemented in mpi4py-fft. The classes `HDF5File` and `NCFile` are used exactly as they are implemented in mpi4py-fft. As a common interface we provide

- `ShenfunFile()`

where `ShenfunFile()` returns an instance of either `HDF5File` or `NCFile`, depending on choice of backend.

For example, to create an HDF5 writer for a 3D TensorProductSpace with Fourier bases in all directions:

```
from shenfun import *
from mpi4py import MPI
N = (24, 25, 26)
K0 = Basis(N[0], 'F', dtype='D')
K1 = Basis(N[1], 'F', dtype='D')
K2 = Basis(N[2], 'F', dtype='d')
T = TensorProductSpace(MPI.COMM_WORLD, (K0, K1, K2))
fl = ShenfunFile('myh5file', T, backend='hdf5', mode='w')
```

The file instance *fl* will now have two method that can be used to either `write` dataarrays to file, or `read` them back again.

- `fl.write`
- `fl.read`

With the `HDF5` backend we can write both arrays from physical space (`Array`), as well as spectral space (`Function`). However, the `NetCDF4` backend cannot handle complex dataarrays, and as such it can only be used for real physical dataarrays.

In addition to storing complete dataarrays, we can also store any slices of the arrays. To illustrate, this is how to store three snapshots of the `u` array, along with some *global* 2D and 1D slices:

```
u = Array(T)
u[:] = np.random.random(u.shape)
d = {'u': [u, (u, np.s_[4, :, :]), (u, np.s_[4, 4, :])]}
fl.write(0, d)
u[:] = 2
fl.write(1, d)
```

The `ShenfunFile` may also be used for the `MixedTensorProductSpace`, or `VectorTensorProductSpace`, that are collections of the scalar `TensorProductSpace`. We can create a `MixedTensorProductSpace` consisting of two TensorProductSpaces, and an accompanying writer class as:

```
TT = MixedTensorProductSpace([T, T])
fl_m = ShenfunFile('mixed', TT, backend='hdf5', mode='w')
```

Let's now consider a transient problem where we step a solution forward in time. We create a solution array from the `Array` class, and update the array inside a while loop:

```
TT = VectorTensorProductSpace(T)
fl_m = ShenfunFile('mixed', TT, backend='hdf5', mode='w')
u = Array(TT)
tstep = 0
du = {'uv': (u,
             (u, [4, slice(None), slice(None)]),
             (u, [slice(None), 10, 10]))}
while tstep < 3:
    fl_m.write(tstep, du, forward_output=False)
    tstep += 1
```

Note that on each time step the arrays `u`, `(u, [4, slice(None), slice(None)])` and `(u, [slice(None), 10, 10])` are vectors, and as such of global shape `(3, 24, 25, 26)`, `(3, 25, 26)` and `(3, 25)`, respectively. However, they are stored in the hdf5 file under their spatial dimensions `1D`, `2D` and `3D`, respectively.

Note that the slices in the above dictionaries are *global* views of the global arrays, that may or may not be distributed over any number of processors. Also note that these routines work with any number of CPUs, and the number of CPUs does not need to be the same when storing or retrieving the data.

After running the above, the different arrays will be found in groups stored in *myyfile.h5* with directory tree structure as:

```
myh5file.h5/
└─ u/
   ├─ 1D/
   |  └─ 4_4_slice/
   |     ├─ 0
   |     └─ 1
   ├─ 2D/
   |  └─ 4_slice_slice/
   |     ├─ 0
   |     └─ 1
   ├─ 3D/
   |  ├─ 0
   |  └─ 1
   └─ mesh/
      ├─ x0
      ├─ x1
      └─ x2
```

Likewise, the *mixed.h5* file will at the end of the loop look like:

```
mixed.h5/
└─ uv/
   ├─ 1D/
   |  └─ slice_10_10/
   |     ├─ 0
   |     ├─ 1
   |     └─ 3
   ├─ 2D/
```

(continues on next page)

(continued from previous page)

```
|   └─ 4_slice_slice/
|      ├─ 0
|      ├─ 1
|      └─ 3
├─ 3D/
|   ├─ 0
|   ├─ 1
|   └─ 3
└─ mesh/
    ├─ x0
    ├─ x1
    └─ x2
```

Note that the mesh is stored as well as the results. The three mesh arrays are all 1D arrays, representing the domain for each basis in the TensorProductSpace.

With NetCDF4 the layout is somewhat different. For `mixed` above, if we were using backend `netcdf` instead of `hdf5`, we would get a datafile where `ncdump -h mixed.nc` would result in:

```
netcdf mixed {
dimensions:
        time = UNLIMITED ; // (3 currently)
        i = 3 ;
        x = 24 ;
        y = 25 ;
        z = 26 ;
variables:
        double time(time) ;
        double i(i) ;
        double x(x) ;
        double y(y) ;
        double z(z) ;
        double uv(time, i, x, y, z) ;
        double uv_4_slice_slice(time, i, y, z) ;
        double uv_slice_10_10(time, i, x) ;
}
```

Note that it is also possible to store vector arrays as scalars. For NetCDF4 this is necessary for direct visualization using Visit. To store vectors as scalars, simply use:

```
fl_m.write(tstep, du, forward_output=False, as_scalar=True)
```

3.1 ParaView

The stored datafiles can be visualized in ParaView. However, ParaView cannot understand the content of these HDF5-files without a little bit of help. We have to explain that these data-files contain structured arrays of such and such shape. The way to do this is through the simple XML descriptor XDMF. To this end there is a function imported from mpi4py-fft called `generate_xdmf` that can be called with any one of the generated hdf5-files:

```
generate_xdmf('myh5file.h5')
generate_xdmf('mixed.h5')
```

This results in some light xdmf-files being generated for the 2D and 3D arrays in the hdf5-file:

- `myh5file.xdmf`

- `myh5file_4_slice_slice.xdmf`
- `mixed.xdmf`
- `mixed_4_slice_slice.xdmf`

These xdmf-files can be opened and inspected by ParaView. Note that 1D arrays are not wrapped, and neither are 4D.

An annoying feature of Paraview is that it views a three-dimensional array of shape (N_0, N_1, N_2) as transposed compared to shenfun. That is, for Paraview the *last* axis represents the x-axis, whereas shenfun (like most others) considers the first axis to be the x-axis. So when opening a three-dimensional array in Paraview one needs to be aware. Especially when plotting vectors. Assume that we are working with a Navier-Stokes solver and have a three-dimensional `VectorTensorProductSpace` to represent the fluid velocity:

```
from mpi4py import MPI
from shenfun import *

comm = MPI.COMM_WORLD
N = (32, 64, 128)
V0 = Basis(N[0], 'F', dtype='D')
V1 = Basis(N[1], 'F', dtype='D')
V2 = Basis(N[2], 'F', dtype='d')
T = TensorProductSpace(comm, (V0, V1, V2))
TV = VectorTensorProductSpace(T)
U = Array(TV)
U[0] = 0
U[1] = 1
U[2] = 2
```

To store the resulting `Array U` we can create an instance of the `HDF5File` class, and store using keyword `as_scalar=True`:

```
hdf5file = ShenfunFile("NS", TV, backend='hdf5', mode='w')
...
file.write(0, {'u': [U]}, as_scalar=True)
file.write(1, {'u': [U]}, as_scalar=True)
```

Alternatively, one may store the arrays directly as:

```
U.write('U.h5', 'u', 0, domain=T.mesh(), as_scalar=True)
U.write('U.h5', 'u', 1, domain=T.mesh(), as_scalar=True)
```

Generate an xdmf file through:

```
generate_xdmf('NS.h5')
```

and open the generated `NS.xdmf` file in Paraview. You will then see three scalar arrays `u0, u1, u2`, each one of shape `(32, 64, 128)`, for the vector component in what Paraview considers the z, y and x directions, respectively. Other than the swapped coordinate axes there is no difference. But be careful if creating vectors in Paraview with the Calculator. The vector should be created as:

```
u0*kHat+u1*jHat+u2*iHat
```

CHAPTER

FOUR

INSTALLATION

Shenfun has a few dependencies

- mpi4py
- FFTW
- mpi4py-fft
- cython
- numpy
- sympy
- scipy
- h5py

that are mostly straight-forward to install, or already installed in most Python environments. The first two are usually most troublesome. Basically, for mpi4py you need to have a working MPI installation, whereas FFTW is available on most high performance computer systems. If you are using conda, then all you need to install a fully functional shenfun, with all the above dependencies, is

```
conda install -c conda-forge shenfun
```

You probably want to install into a fresh environment, though, which can be achieved with

```
conda create --name shenfun -c conda-forge shenfun
conda activate shenfun
```

Note that this gives you shenfun with default settings. This means that you will probably get the openmpi backend. To make sure that shenfun is is installed with mpich instead do

```
conda create --name shenfun -c conda-forge shenfun mpich
```

If you do not use conda, then you need to make sure that MPI and FFTW are installed by some other means. You can then install any version of shenfun hosted on pypi using pip

```
pip install shenfun
```

whereas the following will install the latest version from github

```
pip install git+https://github.com/spectralDNS/shenfun.git@master
```

Note that a common approach is to install `shenfun` from `conda-forge` to get all the dependencies, and then build a local version by (after cloning or forking to a local folder) running from the top directory

```
pip install .
```

or

```
python setup.py build_ext -i
```

This is required to build all the Cython dependencies locally.

4.1 Optimization

Shenfun contains a few routines (essentially linear algebra solvers and matrix vector products) that are difficult to vectorize with numpy, and for this reason they have been implemented in either (or both of) Numba or Cython. The user may choose which implementation to use through the environment variable `SHENFUN_OPTIMIZATION`. The default is to use cython, but it is possible to enable either one by making the appropriate choice in the active terminal

```
export SHENFUN_OPTIMIZATION={CYTHON,NUMBA}
```

4.2 Additional dependencies

For storing and retrieving data you need either HDF5 or netCDF4, compiled with support for MPI (see Postprocessing). Both HDF5 and netCDF4 are already available with parallel support on conda-forge, and, if they were not installed at the same time as shenfun, they can be installed as

```
conda install -c conda-forge h5py=*=mpi* netcdf4=*=mpi*
```

Note that parallel HDF5 and NetCDF4 often are available as modules on supercomputers. Otherwise, see the respective packages for how to install with support for MPI.

Some of the plots in the Demos are created using the matplotlib library. Matplotlib is not a required dependency, but it may be easily installed from conda using

```
conda install matplotlib
```

4.3 Test installation

After installing (from source) it may be a good idea to run all the tests located in the tests folder. The tests are run with pytest from the main directory of the source code

```
python -m pytest tests/
```

However, note that for conda you need to install pytest into the correct environment as well. A common mistake is to run a version of pytest that has already been installed in a different conda environment, perhaps using a different Python version.

The tests are run automatically on every commit to github, see

CHAPTER

FIVE

HOW TO CITE?

Please cite shenfun using

```
@inproceedings{shenfun,
    author = {Mortensen, Mikael},
    booktitle = {MekIT'17 - Ninth national conference on Computational Mechanics},
    isbn = {978-84-947311-1-2},
    pages = {273--298},
    archivePrefix = "arXiv",
    eprint = {1708.03188},
    publisher = {International Center for Numerical Methods in Engineering (CIMNE)},
    title = {Shenfun - automating the spectral Galerkin method},
    editor = {Skallerud, Bjorn Helge and Andersson, Helge Ingolf},
    year = {2017}
}
@article{mortensen_joss,
    author = {Mortensen, Mikael},
    year = 2018,
    title = {Shenfun: High performance spectral Galerkin computing platform},
    journal = {Journal of Open Source Software},
    volume = 3,
    number = 31,
    pages = 1071,
    doi = https://doi.org/10.21105/joss.01071
}
```

CHAPTER

SIX

HOW TO CONTRIBUTE?

Shenfun is an open source project and anyone is welcome to contribute. An easy way to get started is by suggesting a new enhancement on the issue tracker. If you have found a bug, then either report this on the issue tracker, er even better, make a fork of the repository, fix the bug and then create a pull request to get the fix into the master branch.

If you have a particularily interesting application, then we are very interested in pull requests that are adding new demo programs. If you feel like adding an extended demo, then have a look at the examples in the docs/demos folder. Note that extended demos are written using doconce.

CHAPTER

SEVEN

DEMOS

7.1 Demo - 1D Poisson's equation

Authors Mikael Mortensen (mikaem at math.uio.no)

Date Jun 7, 2020

Summary. This is a demonstration of how the Python module shenfun can be used to solve Poisson's equation with Dirichlet boundary conditions in one dimension. Spectral convergence, as shown in Figure *Convergence of 1D Poisson solvers for both Legendre and Chebyshev modified basis function*, is demonstrated. The demo is implemented in a single Python file dirichlet_poisson1D.py, and the numerical method is is described in more detail by J. Shen [She94] and [She95].

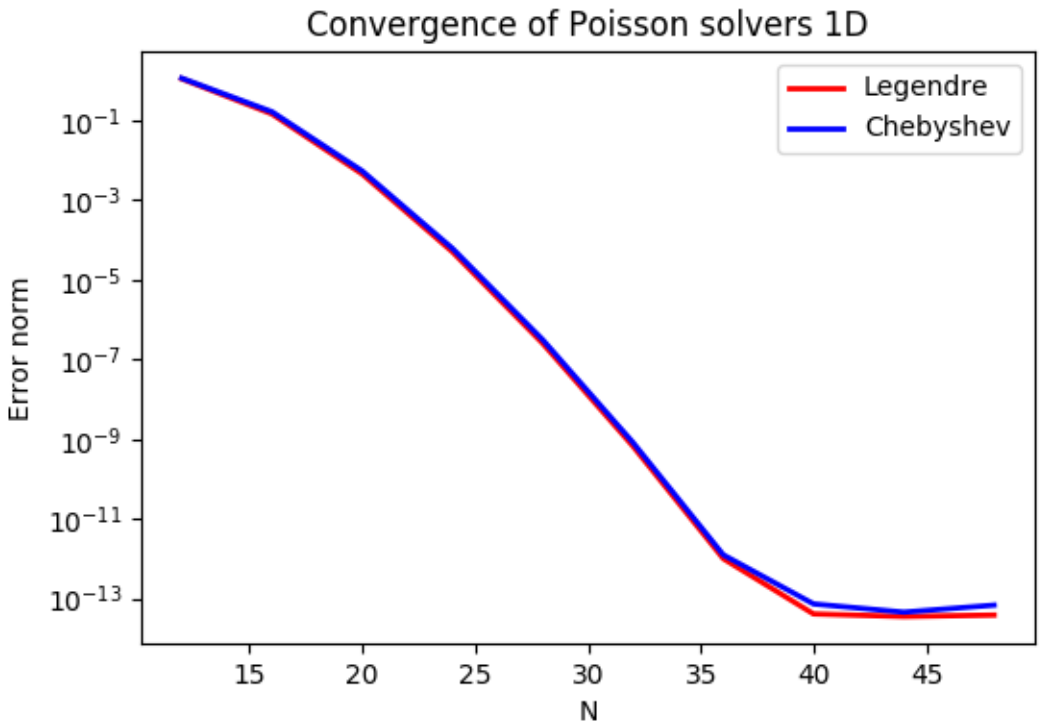


Fig. 1: *Convergence of 1D Poisson solvers for both Legendre and Chebyshev modified basis function*

7.1.1 Model problem

Poisson's equation

Poisson's equation is given as

$$\nabla^2 u(x) = f(x) \quad \text{for } x \in [-1, 1], \tag{7.1}$$

$$u(-1) = a, u(1) = b,$$

where $u(x)$ is the solution, $f(x)$ is a function and a, b are two possibly non-zero constants.

To solve Eq. (7.1) with the Galerkin method we need smooth continuously differentiable basis functions, v_k, that satisfy the given boundary conditions. And then we look for solutions like

$$u(x) = \sum_{k=0}^{N-1} \hat{u}_k v_k(x), \tag{7.2}$$

where N is the size of the discretized problem, $\hat{\mathbf{u}} = \{\hat{u}_k\}_{k=0}^{N-1}$ are the unknown expansion coefficients, and the basis is span$\{v_k\}_{k=0}^{N-1}$.

The basis functions can, for example, be constructed from Chebyshev, $T_k(x)$, or Legendre, $L_k(x)$, polynomials and we use the common notation $\phi_k(x)$ to represent either one of them. It turns out that it is easiest to use basis functions with homogeneous Dirichlet boundary conditions

$$v_k(x) = \phi_k(x) - \phi_{k+2}(x), \tag{7.3}$$

for $k = 0, 1, \ldots N-3$. This gives the basis $V_0^N = \text{span}\{v_k(x)\}_{k=0}^{N-3}$. We can then add two more linear basis functions (that belong to the kernel of Poisson's equation)

$$v_{N-2} = \frac{1}{2}(\phi_0 - \phi_1), \tag{7.4}$$

$$v_{N-1} = \frac{1}{2}(\phi_0 + \phi_1). \tag{7.5}$$

which gives the inhomogeneous basis $V^N = \text{span}\{v_k\}_{k=0}^{N-1}$. With the two linear basis functions it is easy to see that the last two degrees of freedom, $\hat{u}_{N-2}$ and $\hat{u}_{N-1}$, now are given as

$$u(-1) = \sum_{k=0}^{N-1} \hat{u}_k v_k(-1) = \hat{u}_{N-2} = a, \tag{7.6}$$

$$u(+1) = \sum_{k=0}^{N-1} \hat{u}_k v_k(+1) = \hat{u}_{N-1} = b, \tag{7.7}$$

and, as such, we only have to solve for $\{\hat{u}_k\}_{k=0}^{N-3}$, just like for a problem with homogeneous boundary conditions (for homogeneous boundary condition we simply have $\hat{u}_{N-2} = \hat{u}_{N-1} = 0$). We now formulate a variational problem using the Galerkin method: Find $u \in V^N$ such that

$$\int_{-1}^{1} \nabla^2 u \, v \, w \, dx = \int_{-1}^{1} f \, v \, w \, dx \quad \forall v \in V_0^N. \tag{7.8}$$

Note that since we only have $N-3$ unknowns we are only using the homogeneous test functions from V_0^N.

The weighted integrals, weighted by $w(x)$, are called inner products, and a common notation is

$$\int_{-1}^{1} u \, v \, w \, dx = (u, v)_w. \tag{7.9}$$

The integral can either be computed exactly, or with quadrature. The advantage of the latter is that it is generally faster, and that non-linear terms may be computed just as quickly as linear. For a linear problem, it does not make much of a difference, if any at all. Approximating the integral with quadrature, we obtain

$$\begin{aligned}\int_{-1}^{1} u\,v\,w\,dx &\approx (u,v)_w^N\,,\\ &\approx \sum_{j=0}^{N-1} u(x_j)v(x_j)w(x_j),\end{aligned}$$

where $\{w(x_j)\}_{j=0}^{N-1}$ are quadrature weights. The quadrature points $\{x_j\}_{j=0}^{N-1}$ are specific to the chosen basis, and even within basis there are two different choices based on which quadrature rule is selected, either Gauss or Gauss-Lobatto.

Inserting for test and trialfunctions, we get the following bilinear form and matrix $A \in \mathbb{R}^{N-3\times N-3}$ for the Laplacian (using the summation convention in step 2)

$$\begin{aligned}\left(\nabla^2 u, v\right)_w^N &= \left(\nabla^2 \sum_{k=0}^{N-3} \hat{u}_k v_k, v_j\right)_w^N, \quad j = 0,1,\ldots,N-3\\ &= \left(\nabla^2 v_k, v_j\right)_w^N \hat{u}_k,\\ &= a_{jk}\hat{u}_k.\end{aligned}$$

Note that the sum in $a_{jk}\hat{u}_k$ runs over $k = 0, 1, \ldots, N-3$ since the second derivatives of v_{N-1} and v_N are zero. The right hand side linear form and vector is computed as $\tilde{f}_j = (f, v_j)_w^N$, for $j = 0, 1, \ldots, N-3$, where a tilde is used because this is not a complete transform of the function f, but only an inner product.

The linear system of equations to solve for the expansion coefficients of $u(x)$ is given as

$$A\hat{\mathbf{u}} = \tilde{\mathbf{f}}. \tag{7.10}$$

Now, when the expansion coefficients $\hat{\mathbf{u}}$ are found by solving this linear system, they may be transformed to real space $u(x)$ using (7.160), and here the contributions from $\hat{u}_{N-2}$ and $\hat{u}_{N-1}$ must be accounted for. Note that the matrix A (different for Legendre or Chebyshev) has a very special structure that allows for a solution to be found very efficiently in order of $\mathcal{O}(N)$ operations, see [She94] and [She95]. These solvers are implemented in shenfun for both bases.

Method of manufactured solutions

In this demo we will use the method of manufactured solutions to demonstrate spectral accuracy of the `shenfun` Dirichlet bases. To this end we choose an analytical function that satisfies the given boundary conditions:

$$u_e(x) = \sin(k\pi x)(1-x^2) + a(1-x)/2 + b(1+x)/2, \tag{7.11}$$

where k is an integer and a and b are constants. Now, feeding u_e through the Laplace operator, we see that the last two linear terms disappear, whereas the first term results in

$$\nabla^2 u_e(x) = \frac{d^2 u_e}{dx^2}, \tag{7.12}$$

$$= -4k\pi x\cos(k\pi x) - 2\sin(k\pi x) - k^2\pi^2(1-x^2)\sin(k\pi x). \tag{7.13}$$

Now, setting $f_e(x) = \nabla^2 u_e(x)$ and solving for $\nabla^2 u(x) = f_e(x)$, we can compare the numerical solution $u(x)$ with the analytical solution $u_e(x)$ and compute error norms.

7.1.2 Implementation

Preamble

We will solve Poisson's equation using the shenfun Python module. The first thing needed is then to import some of this module's functionality plus some other helper modules, like Numpy and Sympy:

```
from shenfun import inner, div, grad, TestFunction, TrialFunction, Function, \
    project, Dx, Array, Basis
import numpy as np
from sympy import symbols, cos, sin, exp, lambdify
```

We use `Sympy` for the manufactured solution and `Numpy` for testing.

Manufactured solution

The exact solution $u_e(x)$ and the right hand side $f_e(x)$ are created using `Sympy` as follows

```
a = -1
b = 1
k = 4
x = symbols("x")
ue = sin(k*np.pi*x)*(1-x**2) + a*(1 - x)/2. + b*(1 + x)/2.
fe = ue.diff(x, 2)
```

These solutions are now valid for a continuous domain. The next step is thus to discretize, using a discrete mesh $\{x_j\}_{j=0}^{N-1}$ and a finite number of basis functions.

Note that it is not mandatory to use `Sympy` for the manufactured solution. Since the solution is known (7.13), we could just as well simply use `Numpy` to compute f_e at $\{x_j\}_{j=0}^{N-1}$. However, with `Sympy` it is much easier to experiment and quickly change the solution.

Discretization

We create a basis with a given number of basis functions, and extract the computational mesh from the basis itself

```
N = 32
SD = Basis(N, 'Chebyshev', bc=(a, b))
#SD = Basis(N, 'Legendre', bc=(a, b))
X = SD.mesh(N)
```

Note that we can either choose a Legendre or a Chebyshev basis.

Variational formulation

The variational problem (7.124) can be assembled using `shenfun`'s `TrialFunction`, `TestFunction` and `inner()` functions.

```
u = TrialFunction(SD)
v = TestFunction(SD)
# Assemble left hand side matrix
A = inner(v, div(grad(u)))
# Assemble right hand side
fj = Array(SD, buffer=fe)
```

(continues on next page)

(continued from previous page)

```
f_hat = Function(SD)
f_hat = inner(v, fj, output_array=f_hat)
```

Note that the `sympy` function `fe` can be used to initialize the `Array` `fj`. We wrap this Numpy array in an `Array` class (`fj = Array(SD, buffer=fe)`), because an Array is required as input to the `inner()` function.

Solve linear equations

Finally, solve linear equation system and transform solution from spectral $\{\hat{u}_k\}_{k=0}^{N-1}$ vector to the real space $\{u(x_j)\}_{j=0}^{N-1}$ and then check how the solution corresponds with the exact solution u_e. To this end we compute the L_2-errornorm using the `shenfun` function `dx()`

```
u_hat = A.solve(f_hat)
uj = SD.backward(u_hat)
ue = Array(SD, buffer=ue)

print("Error=%2.16e" %(np.sqrt(dx((uj-ua)**2))))
assert np.allclose(uj, ue)
```

Convergence test

A complete solver is given in Sec. *Complete solver*. This solver is created such that it takes in two commandline arguments and prints out the L_2-errornorm of the solution in the end. We can use this to write a short script that performs a convergence test. The solver is run like

```
>>> python dirichlet_poisson1D.py 32 legendre
Error=1.8132185245826562e-10
```

for a discretization of size $N = 32$ and for the Legendre basis. Alternatively, change `legendre` to `chebyshev` for the Chebyshev basis.

We set up the solver to run for a list of $N = [12, 16, \ldots, 48]$, and collect the errornorms in arrays to be plotted. Such a script can be easily created with the subprocess module

```
import subprocess

N = range(12, 50, 4)
error = {}
for basis in ('legendre', 'chebyshev'):
    error[basis] = []
    for i in range(len(N)):
        output = subprocess.check_output("python dirichlet_poisson1D.py {} {}".
↪format(N[i], basis), shell=True)
        exec(output) # Error is printed as "Error=%2.16e"%(np.linalg.norm(uj-ua))
        error[basis].append(Error)
```

The error can be plotted using matplotlib, and the generated figure is shown in the summary's Fig. *Convergence of 1D Poisson solvers for both Legendre and Chebyshev modified basis function*. The spectral convergence is evident and we can see that after $N = 40$ roundoff errors dominate as the errornorm trails off around 10^{-14}.

```
import matplotlib.pyplot as plt
plt.figure(figsize=(6, 4))
for basis, col in zip(('legendre', 'chebyshev'), ('r', 'b')):
```

(continues on next page)

(continued from previous page)

```
    plt.semilogy(N, error[basis], col, linewidth=2)
plt.title('Convergence of Poisson solvers 1D')
plt.xlabel('N')
plt.ylabel('Error norm')
plt.savefig('poisson1D_errornorm.png')
plt.legend(('Legendre', 'Chebyshev'))
plt.show()
```

Complete solver

A complete solver, that can use either Legendre or Chebyshev bases, chosen as a command-line argument, can be found here.

7.2 Demo - Cubic nonlinear Klein-Gordon equation

Authors Mikael Mortensen (mikaem at math.uio.no)

Date Jun 7, 2020

Summary. This is a demonstration of how the Python module shenfun can be used to solve the time-dependent, nonlinear Klein-Gordon equation, in a triply periodic domain. The demo is implemented in a single Python file KleinGordon.py, and it may be run in parallel using MPI. The Klein-Gordon equation is solved using a mixed formulation. The discretization, and some background on the spectral Galerkin method is given first, before we turn to the actual details of the `shenfun` implementation.

7.2.1 The nonlinear Klein-Gordon equation

Movie showing the evolution of the solution u from Eq. (7.14), in a slice through the center of the domain, computed with the code described in this demo.

Model equation

The cubic nonlinear Klein-Gordon equation is a wave equation important for many scientific applications such as solid state physics, nonlinear optics and quantum field theory [Waz08]. The equation is given as

$$\frac{\partial^2 u}{\partial t^2} = \nabla^2 u - \gamma(u - u|u|^2) \quad \text{for } u \in \Omega, \tag{7.14}$$

with initial conditions

$$u(\boldsymbol{x}, t = 0) = u^0 \quad \text{and} \quad \frac{\partial u(\boldsymbol{x}, t = 0)}{\partial t} = u_t^0. \tag{7.15}$$

The spatial coordinates are here denoted as $\boldsymbol{x} = (x, y, z)$, and t is time. The parameter $\gamma = \pm 1$ determines whether the equations are focusing $(+1)$ or defocusing (-1) (in the movie we have used $\gamma = 1$). The domain $\Omega = [-2\pi, 2\pi]^3$ is triply periodic and initial conditions will here be set as

$$u^0 = 0.1 \exp\left(-\boldsymbol{x} \cdot \boldsymbol{x}\right), \tag{7.16}$$

$$u_t^0 = 0. \tag{7.17}$$

We will solve these equations using a mixed formulation and a spectral Galerkin method. The mixed formulation reads

$$\frac{\partial f}{\partial t} = \nabla^2 u - \gamma(u - u|u|^2), \tag{7.18}$$

$$\frac{\partial u}{\partial t} = f. \tag{7.19}$$

The energy of the solution can be computed as

$$E(u) = \int_\Omega \left(\frac{1}{2} f^2 + \frac{1}{2}|\nabla u|^2 + \gamma(\frac{1}{2}u^2 - \frac{1}{4}u^4) \right) dx \tag{7.20}$$

and it is crucial that this energy remains constant in time.

The movie (*The nonlinear Klein-Gordon equation*) is showing the solution u, computed with the code shown in the bottom of Sec. *Complete solver*.

Spectral Galerkin formulation

The PDEs in (7.18) and (7.19) can be solved with many different numerical methods. We will here use the shenfun software and this software makes use of the spectral Galerkin method. Being a Galerkin method, we need to reshape the governing equations into proper variational forms, and this is done by multiplying (7.18) and (7.19) with the complex conjugate of proper test functions and then integrating over the domain. To this end we use testfunctions $g \in V(\Omega)$ with Eq. (7.18) and $v \in V(\Omega)$ with Eq. (7.19), where $V(\omega)$ is a suitable function space, and obtain

$$\frac{\partial}{\partial t} \int_\Omega f\, \overline{g}\, w\, dx = \int_\Omega \left(\nabla^2 u - \gamma(u - u|u|^2) \right) \overline{g}\, w\, dx, \tag{7.21}$$

$$\frac{\partial}{\partial t} \int_\Omega u\, \overline{v}\, w\, dx = \int_\Omega f\, \overline{v}\, w\, dx. \tag{7.22}$$

Note that the overline is used to indicate a complex conjugate, and w is a weight function associated with the test functions. The functions f and u are now to be considered as trial functions, and the integrals over the domain are often referred to as inner products. With inner product notation

$$(u, v) = \int_\Omega u\, \overline{v}\, w\, dx.$$

and an integration by parts on the Laplacian, the variational problem can be formulated as:

$$\frac{\partial}{\partial t}(f, g) = -(\nabla u, \nabla g) - \gamma \left(u - u|u|^2, g \right), \tag{7.23}$$

$$\frac{\partial}{\partial t}(u, v) = (f, v). \tag{7.24}$$

The time and space discretizations are still left open. There are numerous different approaches that one could take for discretizing in time, and the first two terms on the right hand side of (7.23) can easily be treated implicitly as well as explicitly. However, the approach we will follow in Sec. (*Runge-Kutta integrator*) is a fully explicit 4th order Runge-Kutta method.

Discretization

To find a numerical solution we need to discretize the continuous problem (7.23) and (7.24) in space as well as time. Since the problem is triply periodic, Fourier exponentials are normally the best choice for trial and test functions, and as such we use basis functions

$$\phi_l(x) = e^{\imath l x}, \quad -\infty < l < \infty, \tag{7.25}$$

where l is the wavenumber, and $\underline{l} = \frac{2\pi}{L} l$ is the scaled wavenumber, scaled with domain length L (here 4π). Since we want to solve these equations on a computer, we need to choose a finite number of test functions. A function space V^N can be defined as

$$V^N(x) = \text{span}\{\phi_l(x)\}_{l \in \boldsymbol{l}}, \tag{7.26}$$

where N is chosen as an even positive integer and $\boldsymbol{l} = (-N/2, -N/2+1, \ldots, N/2-1)$. And now, since Ω is a three-dimensional domain, we can create tensor products of such bases to get, e.g., for three dimensions

$$W^{\boldsymbol{N}}(x, y, z) = V^N(x) \otimes V^N(y) \otimes V^N(z), \tag{7.27}$$

where $\boldsymbol{N} = (N, N, N)$. Obviously, it is not necessary to use the same number (N) of basis functions for each direction, but it is done here for simplicity. A 3D tensor product basis function is now defined as

$$\Phi_{lmn}(x, y, z) = e^{\imath \underline{l} x} e^{\imath \underline{m} y} e^{\imath \underline{n} z} = e^{\imath (\underline{l} x + \underline{m} y + \underline{n} z)} \tag{7.28}$$

where the indices for y- and z-direction are $\underline{m} = \frac{2\pi}{L} m, \underline{n} = \frac{2\pi}{L} n$, and $\boldsymbol{m}$ and $\boldsymbol{n}$ are the same as $\boldsymbol{l}$ due to using the same number of basis functions for each direction. One distinction, though, is that for the z-direction expansion coefficients are only stored for $n = (0, 1, \ldots, N/2)$ due to Hermitian symmetry (real input data).

We now look for solutions of the form

$$u(x, y, z, t) = \sum_{n=-N/2}^{N/2-1} \sum_{m=-N/2}^{N/2-1} \sum_{l=-N/2}^{N/2-1} \hat{u}_{lmn}(t) \Phi_{lmn}(x, y, z). \tag{7.29}$$

The expansion coefficients $\hat{\boldsymbol{u}} = \{\hat{u}_{lmn}(t)\}_{(l,m,n) \in \boldsymbol{l} \times \boldsymbol{m} \times \boldsymbol{n}}$ can be related directly to the solution $u(x, y, z, t)$ using Fast Fourier Transforms (FFTs) if we are satisfied with obtaining the solution in quadrature points corresponding to

$$x_i = \frac{4\pi i}{N} - 2\pi \quad \forall\, i \in \boldsymbol{i}, \text{where } \boldsymbol{i} = (0, 1, \ldots, N-1), \tag{7.30}$$

$$y_j = \frac{4\pi j}{N} - 2\pi \quad \forall\, j \in \boldsymbol{j}, \text{where } \boldsymbol{j} = (0, 1, \ldots, N-1), \tag{7.31}$$

$$z_k = \frac{4\pi k}{N} - 2\pi \quad \forall\, k \in \boldsymbol{k}, \text{where } \boldsymbol{k} = (0, 1, \ldots, N-1). \tag{7.32}$$

Note that these points are different from the standard (like $2\pi j/N$) since the domain is set to $[-2\pi, 2\pi]^3$ and not the more common $[0, 2\pi]^3$. We have

$$\boldsymbol{u} = \mathcal{F}_k^{-1}\left(\mathcal{F}_j^{-1}\left(\mathcal{F}_i^{-1}\left(\hat{\boldsymbol{u}}\right)\right)\right) \tag{7.33}$$

with $\boldsymbol{u} = \{u(x_i, y_j, z_k)\}_{(i,j,k) \in \boldsymbol{i} \times \boldsymbol{j} \times \boldsymbol{k}}$ and where $\mathcal{F}_i^{-1}$ is the inverse Fourier transform along the direction of index i, for all $(j, k) \in \boldsymbol{j} \times \boldsymbol{k}$. Note that the three inverse FFTs are performed sequentially, one direction at the time, and that there is no scaling factor due to the definition used for the inverse Fourier transform

$$u(x_j) = \sum_{l=-N/2}^{N/2-1} \hat{u}_l e^{\imath \underline{l} x_j}, \quad \forall\, j \in \boldsymbol{j}. \tag{7.34}$$

Note that this differs from the definition used by, e.g., Numpy.

The inner products used in Eqs. (7.23), (7.24) may be computed using forward FFTs. However, there is a tiny detail that deserves a comment. The regular Fourier inner product is given as

$$\int_0^L e^{\imath \underline{k} x} e^{-\imath \underline{l} x} dx = L\, \delta_{kl}$$

where a weight function is chosen as $w(x) = 1$ and δ_{kl} equals unity for $k = l$ and zero otherwise. In Shenfun we choose instead to use a weight function $w(x) = 1/L$, such that the weighted inner product integrates to unity:

$$\int_0^L e^{\imath \underline{k} x} e^{-\imath \underline{l} x} \frac{1}{L} dx = \delta_{kl}.$$

With this weight function the scalar product and the forward transform are the same and we obtain:

$$(u, \Phi_{lmn}) = \hat{u}_{lmn} = \left(\frac{1}{N}\right)^3 \mathcal{F}_l\left(\mathcal{F}_m\left(\mathcal{F}_n\left(\boldsymbol{u}\right)\right)\right) \quad \forall (l, m, n) \in \boldsymbol{l} \times \boldsymbol{m} \times \boldsymbol{n}, \tag{7.35}$$

From this we see that the variational forms (7.23) and (7.24) may be written in terms of the Fourier transformed quantities $\hat{\boldsymbol{u}}$ and $\hat{\boldsymbol{f}}$. Expanding the exact derivatives of the nabla operator, we have

$$(\nabla u, \nabla v) = (\underline{l}^2 + \underline{m}^2 + \underline{n}^2)\hat{u}_{lmn}, \tag{7.36}$$

$$(u, v) = \hat{u}_{lmn}, \tag{7.37}$$

$$(u|u|^2, v) = \widehat{u|u|^2}_{lmn} \tag{7.38}$$

and as such the equations to be solved for each wavenumber can be found directly as

$$\frac{\partial \hat{f}_{lmn}}{\partial t} = \left(-(\underline{l}^2 + \underline{m}^2 + \underline{n}^2 + \gamma)\hat{u}_{lnm} + \gamma \widehat{u|u|^2}_{lnm}\right), \tag{7.39}$$

$$\frac{\partial \hat{u}_{lnm}}{\partial t} = \hat{f}_{lnm}. \tag{7.40}$$

There is more than one way to arrive at these equations. Taking the 3D Fourier transform of both equations (7.18) and (7.19) is one obvious way. With the Python module shenfun, one can work with the inner products as seen in (7.23) and (7.24), or the Fourier transforms directly. See for example Sec. *Runge-Kutta integrator* for how $(\nabla u, \nabla v)$ can be implemented. In short, `shenfun` contains all the tools required to work with the spectral Galerkin method, and we will now see how `shenfun` can be used to solve the Klein-Gordon equation.

For completion, we note that the discretized problem to solve can be formulated with the Galerkin method as: for all $t > 0$, find $(f, u) \in W^N \times W^N$ such that

$$\frac{\partial}{\partial t}(f, g) = -(\nabla u, \nabla g) - \gamma\left(u - u|u|^2, g\right), \tag{7.41}$$

$$\frac{\partial}{\partial t}(u, v) = (f, v) \quad \forall\, (g, v) \in W^N \times W^N. \tag{7.42}$$

where $u(x, y, z, 0)$ and $f(x, y, z, 0)$ are given as the initial conditions according to Eq. (7.15).

7.2.2 Implementation

To solve the Klein-Gordon equations we need to make use of the Fourier bases in `shenfun`, and these base are found in submodule `shenfun.fourier.bases`. The triply periodic domain allows for Fourier in all three directions, and we can as such create one instance of this base class using `Basis()` with family `Fourier` for each direction. However, since the initial data are real, we can take advantage of Hermitian symmetries and thus make use of a real to complex class for one (but only one) of the directions, by specifying `dtype='d'`. We can only make use of the real-to-complex class for the direction that we choose to transform first with the forward FFT, and the reason is obviously that the output from a forward transform of real data is now complex. We may start implementing the solver as follows

```
from shenfun import *
from mpi4py import MPI
import numpy as np

# Set size of discretization
N = (32, 32, 32)

# Create bases
K0 = Basis(N[0], 'F', domain=(-2*np.pi, 2*np.pi), dtype='D')
K1 = Basis(N[1], 'F', domain=(-2*np.pi, 2*np.pi), dtype='D')
K2 = Basis(N[2], 'F', domain=(-2*np.pi, 2*np.pi), dtype='d')
```

We now have three instances `K0`, `K1` and `K2`, corresponding to the space (7.26), that each can be used to solve one-dimensional problems. However, we want to solve a 3D problem, and for this we need a tensor product space, like (7.27), created as a tensor product of these three spaces

```
# Create communicator
comm = MPI.COMM_WORLD
T = TensorProductSpace(comm, (K0, K1, K2), **{'planner_effort':
                                              'FFTW_MEASURE'})
```

Here the `planner_effort`, which is a flag used by FFTW, is optional. Possibel choices are from the list (`FFTW_ESTIMATE`, `FFTW_MEASURE`, `FFTW_PATIENT`, `FFTW_EXHAUSTIVE`), and the flag determines how much effort FFTW puts in looking for an optimal algorithm for the current platform. Note that it is also possible to use FFTW wisdom with `shenfun`, and as such, for production, one may perform exhaustive planning once and then simply import the result of that planning later, as wisdom.

The `TensorProductSpace` instance `T` contains pretty much all we need for computing inner products or fast transforms between real and wavenumber space. However, since we are going to solve for a mixed system, it is convenient to also use the `MixedTensorProductSpace` class

```
TT = MixedTensorProductSpace([T, T])
```

We need containers for the solution as well as intermediate work arrays for, e.g., the Runge-Kutta method. Arrays are created as

```
uf = Array(TT)            # Solution array in physical space
u, f = uf                 # Split solution array by creating two views u and f
duf = Function(TT)        # Array for right hand sides
du, df = duf              # Split into views
uf_hat = Function(TT)     # Solution in spectral space
uf_hat0 = Function(TT)    # Work array 1
uf_hat1 = Function(TT)    # Work array 2
u_hat, f_hat = uf_hat     # Split into views
```

The `Array` class is a subclass of Numpy's ndarray, without much more functionality than constructors that return arrays of the correct shape according to the basis used in the construction. The `Array` represents the left hand side of (7.29), evaluated on the quadrature mesh. A different type of array is returned by the `Function` class, that subclasses both Nympy's ndarray as well as an internal `BasisFunction` class. An instance of the `Function` represents the entire spectral Galerkin function (7.29). As such, it can be used in complex variational linear forms. For example, if you want to compute the partial derivative $\partial u/\partial x$, then this may be achieved by projection, i.e., find $u_x \in V^N$ such that $(u_x - \partial u/\partial x, v) = 0$, for all $v \in V^N$. This projection may be easily computed in `shenfun` using

```
ux = project(Dx(u_hat, 0, 1), T)
```

The following code, on the other hand, will raise an error since you cannot take the derivative of an interpolated `Array u`, only a `Function`

```
try:
    project(Dx(u, 0, 1), T)
except AssertionError:
    print("AssertionError: Dx not for Arrays")
```

Initialization

The solution array `uf` and its transform `uf_hat` need to be initialized according to Eq. (7.15). To this end it is convenient (but not required, we could just as easily use Numpy for this as well) to use Sympy, which is a Python library for symbolic mathamatics.

```
from sympy import symbols, exp, lambdify

x, y, z = symbols("x,y,z")
ue = 0.1*exp(-(x**2 + y**2 + z**2))
ul = lambdify((x, y, z), ue, 'numpy')
X = T.local_mesh(True)
u[:] = Array(T, buffer=ul(*X))
u_hat = T.forward(u, u_hat)
```

Here `X` is a list of the three mesh coordinates `(x, y, z)` local to the current processor. Each processor has its own part of the computational mesh, and the distribution is handled during the creation of the `TensorProductSpace` class instance `T`. There is no need to do anything about the `f/f_hat` arrays since they are already initialized by default to zero. Note that calling the `ul` function with the argument `*X` is the same as calling with `X[0], X[1], X[2]`.

Runge-Kutta integrator

A fourth order explicit Runge-Kutta integrator requires only a function that returns the right hand sides of (7.39) and (7.40). Such a function can be implemented as

```
# focusing (+1) or defocusing (-1)
gamma = 1
uh = TrialFunction(T)
vh = TestFunction(T)
k2 = -(inner(grad(vh), grad(uh)).scale  + gamma)

def compute_rhs(duf_hat, uf_hat, up, Tp, w0):
    duf_hat.fill(0)
    u_hat, f_hat = uf_hat
    du_hat, df_hat = duf_hat
    df_hat[:] = k2*u_hat
    up = Tp.backward(u_hat, up)
    df_hat += Tp.forward(gamma*up**3, w0)
    du_hat[:] = f_hat
    return duf_hat
```

The code is fairly self-explanatory. `k2` represents the coefficients in front of the linear $\hat{u}$ in (7.39). The output array is `duf_hat`, and the input array is `uf_hat`, whereas `up` and `w0` are work arrays. The array `duf_hat` contains the right hand sides of both (7.39) and (7.40), where the linear and nonlinear terms are recognized in the code as comments `(1)` and `(2)`. The array `uf_hat` contains the solution at initial and intermediate Runge-Kutta steps.

With a function that returns the right hand side in place, the actual integrator can be implemented as

```
w0 = Function(T)
a = [1./6., 1./3., 1./3., 1./6.]          # Runge-Kutta parameter
b = [0.5, 0.5, 1.]                        # Runge-Kutta parameter
t = 0
dt = 0.01
end_time = 1.0
while t < end_time-1e-8:
    t += dt
    uf_hat1[:] = uf_hat0[:] = uf_hat
    for rk in range(4):
        duf = compute_rhs(duf, uf_hat, u, T, w0)
        if rk < 3:
            uf_hat[:] = uf_hat0 + b[rk]*dt*duf
        uf_hat1 += a[rk]*dt*duf
    uf_hat[:] = uf_hat1
```

Complete solver

A complete solver is given below, with intermediate plotting of the solution and intermediate computation of the total energy. Note that the total energy is unchanged to 8 decimal points at $t = 100$.

```
from sympy import symbols, exp, lambdify
import numpy as np
import matplotlib.pyplot as plt
from mpi4py import MPI
from time import time
from shenfun import *

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Use sympy to set up initial condition
x, y, z = symbols("x,y,z")
ue = 0.1*exp(-(x**2 + y**2 + z**2))
ul = lambdify((x, y, z), ue, 'numpy')

# Size of discretization
N = (64, 64, 64)

# Defocusing or focusing
gamma = 1

K0 = Basis(N[0], 'F', domain=(-2*np.pi, 2*np.pi), dtype='D')
K1 = Basis(N[1], 'F', domain=(-2*np.pi, 2*np.pi), dtype='D')
K2 = Basis(N[2], 'F', domain=(-2*np.pi, 2*np.pi), dtype='d')
T = TensorProductSpace(comm, (K0, K1, K2), slab=False,
                       **{'planner_effort': 'FFTW_MEASURE'})

TT = MixedTensorProductSpace([T, T])

X = T.local_mesh(True)
uf = Array(TT)
u, f = uf[:]
up = Array(T)
duf = Function(TT)
du, df = duf[:]
```

(continues on next page)

(continued from previous page)

```
uf_hat = Function(TT)
uf_hat0 = Function(TT)
uf_hat1 = Function(TT)
w0 = Function(T)
u_hat, f_hat = uf_hat[:]

# initialize (f initialized to zero, so all set)
u[:] = ul(*X)
u_hat = T.forward(u, u_hat)

uh = TrialFunction(T)
vh = TestFunction(T)
k2 = -inner(grad(vh), grad(uh)).scale - gamma

count = 0
def compute_rhs(duf_hat, uf_hat, up, T, w0):
    global count
    count += 1
    duf_hat.fill(0)
    u_hat, f_hat = uf_hat[:]
    du_hat, df_hat = duf_hat[:]
    df_hat[:] = k2*u_hat
    up = T.backward(u_hat, up)
    df_hat += T.forward(gamma*up**3, w0)
    du_hat[:] = f_hat
    return duf_hat

def energy_fourier(comm, a):
    result = 2*np.sum(abs(a[..., 1:-1])**2) + np.sum(abs(a[..., 0])**2) + np.
↪sum(abs(a[..., -1])**2)
    result =  comm.allreduce(result)
    return result

# Integrate using a 4th order Rung-Kutta method
a = [1./6., 1./3., 1./3., 1./6.]          # Runge-Kutta parameter
b = [0.5, 0.5, 1.]                        # Runge-Kutta parameter
t = 0.0
dt = 0.005
end_time = 1.
tstep = 0
if rank == 0:
    plt.figure()
    image = plt.contourf(X[1][..., 0], X[0][..., 0], u[..., 16], 100)
    plt.draw()
    plt.pause(1e-4)
t0 = time()
K = np.array(T.local_wavenumbers(True, True, True))
TV = VectorTensorProductSpace([T, T, T])
gradu = Array(TV)
while t < end_time-1e-8:
    t += dt
    tstep += 1
    uf_hat1[:] = uf_hat0[:] = uf_hat
    for rk in range(4):
        duf = compute_rhs(duf, uf_hat, up, T, w0)
        if rk < 3:
```

(continues on next page)

(continued from previous page)

```
            uf_hat[:] = uf_hat0 + b[rk]*dt*duf
        uf_hat1 += a[rk]*dt*duf
    uf_hat[:] = uf_hat1

    if tstep % 100 == 0:
        uf = TT.backward(uf_hat, uf)
        ekin = 0.5*energy_fourier(T.comm, f_hat)
        es = 0.5*energy_fourier(T.comm, 1j*K*u_hat)
        eg = gamma*np.sum(0.5*u**2 - 0.25*u**4)/np.prod(np.array(N))
        eg =  comm.allreduce(eg)
        gradu = TV.backward(1j*K*u_hat, gradu)
        ep = comm.allreduce(np.sum(f*gradu)/np.prod(np.array(N)))
        ea = comm.allreduce(np.sum(np.array(X)*(0.5*f**2 + 0.5*gradu**2
                            - (0.5*u**2 - 0.25*u**4)*f))/np.prod(np.array(N)))
        if rank == 0:
            image.ax.clear()
            image.ax.contourf(X[1][..., 0], X[0][..., 0], u[..., 16], 100)
            plt.pause(1e-6)
            plt.savefig('Klein_Gordon_{}_real_{}.png'.format(N[0], tstep))
            print("Time = %2.2f Total energy = %2.8e Linear momentum %2.8e Angular␣
↪momentum %2.8e" %(t, ekin+es+eg, ep, ea))
        comm.barrier()

print("Time ", time()-t0)
```

7.3 Demo - 3D Poisson's equation

Authors Mikael Mortensen (mikaem at math.uio.no)

Date Jun 7, 2020

Summary. This is a demonstration of how the Python module shenfun can be used to solve a 3D Poisson equation in a 3D tensor product domain that has homogeneous Dirichlet boundary conditions in one direction and periodicity in the remaining two. The solver described runs with MPI without any further considerations required from the user. Spectral convergence, as shown in Figure *Convergence of 3D Poisson solvers for both Legendre and Chebyshev modified basis function*, is demonstrated. The demo is implemented in a single Python file dirichlet_poisson3D.py, and the numerical method is is described in more detail by J. Shen [She94] and [She95].

7.3.1 Model problem

Poisson equation

The Poisson equation is given as

$$\nabla^2 u(\boldsymbol{x}) = f(\boldsymbol{x}) \quad \text{for } \boldsymbol{x} = (x, y, z) \in \Omega, \tag{7.43}$$

$$u(\pm 1, y, z) = 0, \tag{7.44}$$

$$u(x, 2\pi, z) = u(x, 0, z), \tag{7.45}$$

$$u(x, y, 2\pi) = u(x, y, 0), \tag{7.46}$$

where $u(\boldsymbol{x})$ is the solution and $f(\boldsymbol{x})$ is a function. The domain $\Omega = [-1, 1] \times [0, 2\pi]^2$.

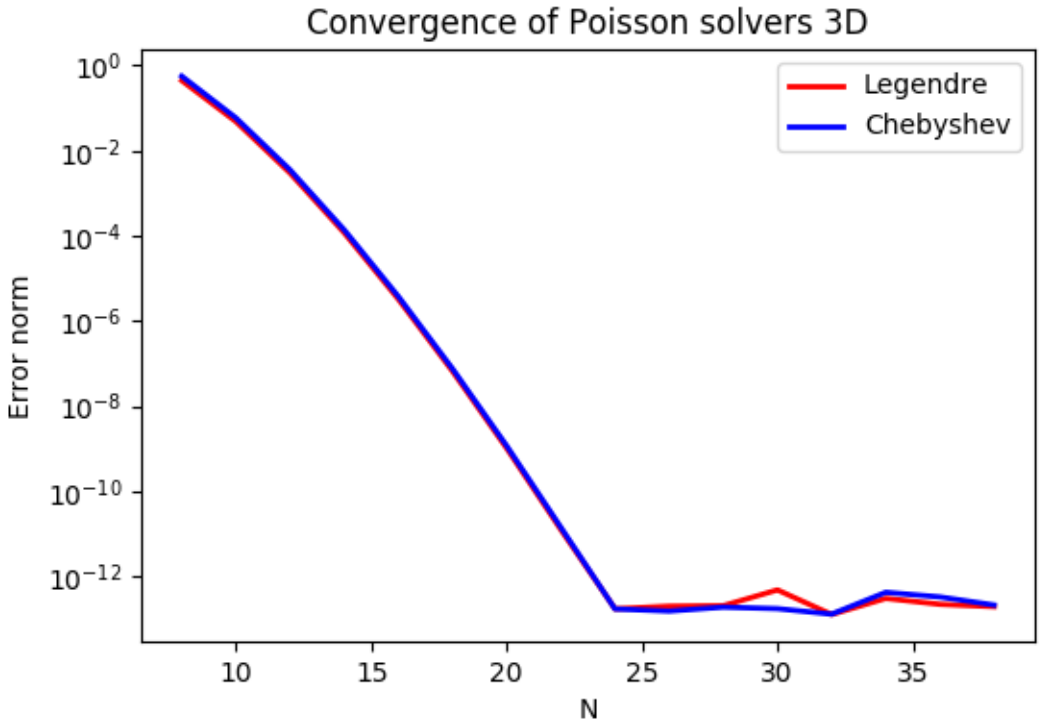


Fig. 2: *Convergence of 3D Poisson solvers for both Legendre and Chebyshev modified basis function*

To solve Eq. (7.43) with the Galerkin method we need smooth basis functions, $v(\boldsymbol{x})$, that live in the Hilbert space $H^1(\Omega)$ and that satisfy the given boundary conditions. To this end we will use one basis function for the x-direction, $\mathcal{X}(x)$, one for the y-direction, $\mathcal{Y}(y)$, and one for the z-direction, $\mathcal{Z}(z)$. And then we create three-dimensional basis functions like

$$v(x, y, z) = \mathcal{X}(x)\mathcal{Y}(y)\mathcal{Z}(z).$$

The basis functions $\mathcal{Y}(y)$ and $\mathcal{Z}(z)$ are chosen as Fourier exponentials, since these functions are periodic. Likewise, the basis functions $\mathcal{X}(x)$ are chosen as modified Legendre or Chebyshev polynomials, using $\phi_l(x)$ to refer to either one

$$\mathcal{X}_l(x) = \phi_l(x) - \phi_{l+2}(x), \forall\, l \in \boldsymbol{l}^{N_0}, \tag{7.47}$$

$$\mathcal{Y}_m(y) = e^{\imath m y}, \forall\, m \in \boldsymbol{m}^{N_1}, \tag{7.48}$$

$$\mathcal{Z}_n(z) = e^{\imath n z}, \forall\, n \in \boldsymbol{n}^{N_2}, \tag{7.49}$$

where the size of the discretized problem is $\boldsymbol{N} = (N_0, N_1, N_2)$, $\boldsymbol{l}^{N_0} = (0, 1, \ldots, N_0 - 3)$, $\boldsymbol{m}^{N_1} = (-N_1/2, -N_1/2 + 1, \ldots, N_1/2 - 1)$ and $\boldsymbol{n}^{N_2} = (-N_2/2, -N_2/2 + 1, \ldots, N_2/2 - 1)$. However, due to Hermitian symmetry, we only store $N_2/2 + 1$ wavenumbers in the z-direction, such that $\boldsymbol{n}^{N_2} = (0, 1, \ldots, N_2/2)$. We refer to the Cartesian wavenumber mesh on vector form as $\boldsymbol{k}$:

$$\boldsymbol{k} = \{(l, m, n)\,|\,(l, m, n) \in \boldsymbol{l}^{N_0} \times \boldsymbol{m}^{N_1} \times \boldsymbol{n}^{N_2}\}.$$

We have the one-dimensional spaces

$$V^{N_0} = \text{span}\{\mathcal{X}_l\}_{l \in \boldsymbol{l}^{N_0}}, \tag{7.50}$$

$$V^{N_1} = \text{span}\{\mathcal{Y}_m\}_{m\in\boldsymbol{m}^{N_1}}, \tag{7.51}$$

$$V^{N_2} = \text{span}\{\mathcal{Z}_n\}_{n\in\boldsymbol{n}^{N_2}}, \tag{7.52}$$

and from these we create a tensor product space $W^{\boldsymbol{N}}(\boldsymbol{x})$

$$W^{\boldsymbol{N}}(\boldsymbol{x}) = V^{N_0}(x) \otimes V^{N_1}(y) \otimes V^{N_2}(z). \tag{7.53}$$

And then we look for discrete solutions $u \in W^{\boldsymbol{N}}$ like

$$u(\boldsymbol{x}) = \sum_{l\in\boldsymbol{l}^{N_0}} \sum_{m\in\boldsymbol{m}^{N_1}} \sum_{n\in\boldsymbol{n}^{N_2}} \hat{u}_{lmn}\mathcal{X}_l(x)\mathcal{Y}_m(y)\mathcal{Z}_n(z), \tag{7.54}$$

$$= \sum_{\mathsf{k}\in\boldsymbol{k}} \hat{u}_{\mathsf{k}} v_{\mathsf{k}}(\boldsymbol{x}), \tag{7.55}$$

where $\hat{u}_{lmn}$ are components of the expansion coefficients for u and the second form, $\{\hat{u}_{\mathsf{k}}\}_{\mathsf{k}\in\boldsymbol{k}}$, is a shorter, simplified notation, with sans-serif $\mathsf{k} = (l, m, n)$. The expansion coefficients are the unknowns in the spectral Galerkin method.

We now formulate a variational problem using the Galerkin method: Find $u \in W^{\boldsymbol{N}}$ such that

$$\int_\Omega \nabla^2 u\, \overline{v}\, w\, \boldsymbol{dx} = \int_\Omega f\, \overline{v}\, w\, \boldsymbol{dx} \quad \forall v \in W^{\boldsymbol{N}}. \tag{7.56}$$

Here $\boldsymbol{dx} = dxdydz$, and the overline represents a complex conjugate, which is needed here because the Fourier exponentials are complex functions. The weighted integrals, weighted by $w(\boldsymbol{x})$, are called inner products, and a common notation is

$$\int_\Omega u\, \overline{v}\, w\, \boldsymbol{dx} = \langle u, v\rangle_w. \tag{7.57}$$

The integral can either be computed exactly, or with quadrature. The advantage of the latter is that it is generally faster, and that non-linear terms may be computed just as quickly as linear. For a linear problem, it does not make much of a difference, if any at all. Approximating the integral with quadrature, we obtain

$$\int_\Omega u\, \overline{v}\, w\, \boldsymbol{dx} \approx \langle u, v\rangle_w^{\boldsymbol{N}}, \tag{7.58}$$

$$\approx \sum_{i=0}^{N_0-1}\sum_{j=0}^{N_1-1}\sum_{k=0}^{N_2-1} u(x_i, y_j, z_k)\overline{v}(x_i, y_j, z_k)w(x_i, y_j, z_k), \tag{7.59}$$

where $w(\boldsymbol{x})$ now are the quadrature weights. The quadrature points $\{x_i\}_{i=0}^{N_0-1}$ are specific to the chosen basis, and even within basis there are two different choices based on which quadrature rule is selected, either Gauss or Gauss-Lobatto. The quadrature points for the Fourier bases are the uniform $\{y_j\}_{j=0}^{N_1-1} = 2\pi j/N_1$ and $\{z_k\}_{k=0}^{N_2-1} = 2\pi k/N_2$.

Inserting for test function (7.54) and trialfunction $v_{pqr} = \mathcal{X}_p\mathcal{Y}_q\mathcal{Z}_r$ on the left hand side of (7.56), we get

$$\begin{aligned}
\langle\nabla^2 u, v\rangle_w^{\boldsymbol{N}} &= \left\langle \nabla^2 \sum_{l\in\boldsymbol{l}^{N_0}} \sum_{m\in\boldsymbol{m}^{N_1}} \sum_{n\in\boldsymbol{n}^{N_2}} \hat{u}_{lmn}\mathcal{X}_l\mathcal{Y}_m\mathcal{Z}_n, \mathcal{X}_p\mathcal{Y}_q\mathcal{Z}_r \right\rangle_w^{\boldsymbol{N}}, \\
&= \left[\left(\mathcal{X}_l^{''}, \mathcal{X}_p\right)_w^N - (m^2+n^2)\left(\mathcal{X}_l, \mathcal{X}_p\right)_w^N\right]\delta_{mq}\delta_{nr}\hat{u}_{lmn}, \\
&= \left(a_{pl} - (m^2+n^2)b_{pl}\right)\hat{u}_{lqr},
\end{aligned}$$

where the notation $(\cdot,\cdot)_w^{N_0}$

$$b_{pl} = (\mathcal{X}_l, \mathcal{X}_p)_w^{N_0} = \sum_{i=0}^{N_0-1} \mathcal{X}_l(x_i)\mathcal{X}_p(x_i)w(x_i), \tag{7.60}$$

is used to represent an L_2 inner product along only the first, nonperiodic, direction. The delta functions above come from integrating over the two periodic directions, where we use constant weight functions $w = 1/(2\pi)$ in the inner products

$$\int_0^{2\pi} \mathcal{Y}_m(y)\overline{\mathcal{Y}}_q(y)\frac{1}{2\pi}dy = \delta_{mq}, \tag{7.61}$$

$$\int_0^{2\pi} \mathcal{Z}_n(z)\overline{\mathcal{Z}}_r(z)\frac{1}{2\pi}dz = \delta_{nr}, \tag{7.62}$$

The Kronecker delta-function δ_{ij} is one for $i = j$ and zero otherwise.

The right hand side of Eq. (7.56) is computed as

$$\tilde{f}_{pqr} = \langle f, \mathcal{X}_p\mathcal{Y}_q\mathcal{Z}_r\rangle_w^{\boldsymbol{N}}, \tag{7.63}$$

where a tilde is used because this is not a complete transform of the function f, but only an inner product.

The linear system of equations to solve for the expansion coefficients can now be found as follows

$$\left(a_{lj} - (m^2+n^2)b_{lj}\right)\hat{u}_{jmn} = \tilde{f}_{lmn} \quad \forall\,(l,m,n) \in \boldsymbol{k}. \tag{7.64}$$

Now, when $\hat{\boldsymbol{u}} = \{\hat{u}_{\mathsf{k}}\}_{\mathsf{k}\in\boldsymbol{k}}$ is found by solving this linear system over the entire computational mesh, it may be transformed to real space $u(\boldsymbol{x})$ using (7.54). Note that the matrices $A \in \mathbb{R}^{N_0-3\times N_0-3}$ and $B \in \mathbb{R}^{N_0-3\times N_0-3}$ differ for Legendre or Chebyshev bases, but for either case they have a special structure that allows for a solution to be found very efficiently in the order of $\mathcal{O}(N_0-3)$ operations given m and n, see [She94] and [She95]. Fast solvers for (7.64) are implemented in `shenfun` for both bases.

Method of manufactured solutions

In this demo we will use the method of manufactured solutions to demonstrate spectral accuracy of the `shenfun` bases. To this end we choose a smooth analytical function that satisfies the given boundary conditions:

$$u_e(x,y,z) = (\cos(4x) + \sin(2y) + \sin(4z))\,(1-x^2). \tag{7.65}$$

Sending u_e through the Laplace operator, we obtain the right hand side

$$\nabla^2 u_e(x,y,z) = -16(1-x^2)\cos(4x) + 16x\sin(4x) - 2\cos(4x) - (1-x^2)(4\sin(2y) + 16\sin(4z)). \tag{7.66}$$

Now, setting $f_e(\boldsymbol{x}) = \nabla^2 u_e(\boldsymbol{x})$ and solving for $\nabla^2 u(\boldsymbol{x}) = f_e(\boldsymbol{x})$, we can compare the numerical solution $u(\boldsymbol{x})$ with the analytical solution $u_e(\boldsymbol{x})$ and compute error norms.

7.3.2 Implementation

Preamble

We will solve the Poisson problem using the shenfun Python module. The first thing needed is then to import some of this module's functionality plus some other helper modules, like Numpy and Sympy:

```
from sympy import symbols, cos, sin, exp, lambdify
import numpy as np
from shenfun.tensorproductspace import TensorProductSpace
from shenfun import inner, div, grad, TestFunction, TrialFunction, Function, \
    project, Dx, Basis
from mpi4py import MPI
```

We use `Sympy` for the manufactured solution and `Numpy` for testing. MPI for Python (`mpi4py`) is required for running the solver with MPI.

Manufactured solution

The exact solution $u_e(x, y, z)$ and the right hand side $f_e(x, y, z)$ are created using `Sympy` as follows

```
x, y, z = symbols("x,y,z")
ue = (cos(4*x) + sin(2*y) + sin(4*z))*(1-x**2)
fe = ue.diff(x, 2) + ue.diff(y, 2) + ue.diff(z, 2)

# Lambdify for faster evaluation
ul = lambdify((x, y, z), ue, 'numpy')
fl = lambdify((x, y, z), fe, 'numpy')
```

These solutions are now valid for a continuous domain. The next step is thus to discretize, using the computational mesh

$$(x_i, y_j, z_k) \forall (i, j, k) \in [0, 1, \ldots, N_0 - 1] \times [0, 1, \ldots, N_1 - 1] \times [0, 1, \ldots, N_2 - 1]$$

and a finite number of basis functions.

Note that it is not mandatory to use `Sympy` for the manufactured solution. Since the solution is known (7.66), we could just as well simply use `Numpy` to compute f_e. However, with `Sympy` it is much easier to experiment and quickly change the solution.

Discretization and MPI

We create three bases with given size, one for each dimension of the problem. From these three bases a `TensorProductSpace` is created.

```
# Size of discretization
N = [14, 15, 16]

SD = Basis(N[0], 'Chebyshev', bc=(0, 0))
#SD = Basis(N[0], 'Legendre', bc=(0, 0))
K1 = Basis(N[1], 'Fourier', dtype='D')
K2 = Basis(N[2], 'Fourier', dtype='d')
T = TensorProductSpace(comm, (SD, K1, K2), axes=(0, 1, 2))
X = T.local_mesh()
```

Note that we can either choose a Legendre or a Chebyshev basis for the nonperiodic direction. The `TensorProductSpace` class takes an MPI communicator as first argument and the computational mesh is distributed internally using the `pencil` method. The `T.local_mesh` method returns the mesh local to each processor. The `axes` keyword determines the order of transforms going back and forth between real and spectral space. With `axes=(0, 1, 2)` and a forward transform (from real space to spectral, i.e., from u to $\hat{u}$) axis 2 is transformed first and then 1 and 0, respectively.

The manufactured solution is created with Dirichlet boundary conditions in the x-direction, and for this reason `SD` is the first basis in `T`. We could just as well have put the nonperiodic direction along either y- or z-direction, though, but this would then require that the order of the transformed axes be changed as well. For example, putting the Dirichlet direction along y, we would need to create the tensorproductspace as

```
T = TensorProductSpace(comm, (K1, SD, K2), axes=(1, 0, 2))
```

such that the Dirichlet direction is the last to be transformed. The reason for this is that only the Dirichlet direction leads to matrices that need to be inverted (or solved). And for this we need the entire data array along the Dirichlet direction to be local to the processor. If the `SD` basis is the last to be transformed, then the data will be aligned in this direction, whereas the other two directions may both, or just one of them, be distributed.

Note that `X` is a list containing local values of the arrays $\{x_i\}_{i=0}^{N_0-1}$, $\{y_j\}_{j=0}^{N_1-0}$ and $\{z_k\}_{k=0}^{N_2-1}$. For example, using 4 procesors and a processor mesh of shape 2×2, then the local slices for each processor in spectral space are

```
>>> print(comm.Get_rank(), T.local_slice())
3 [slice(0, 14, None), slice(8, 15, None), slice(5, 9, None)]
1 [slice(0, 14, None), slice(0, 8, None), slice(5, 9, None)]
2 [slice(0, 14, None), slice(8, 15, None), slice(0, 5, None)]
0 [slice(0, 14, None), slice(0, 8, None), slice(0, 5, None)]
```

where the global shape is $\boldsymbol{N} = (14, 15, 9)$ after taking advantage of Hermitian symmetry in the z-direction. So, all processors have the complete first dimension available locally, as they should. Furthermore, processor three owns the slices from $8:15$ and $5:9$ along axes y and z, respectively. Processor 2 owns slices $0:8$ and $0:5$ etc. In real space the mesh is distributed differently. First of all the global mesh shape is $\boldsymbol{N} = (14, 15, 16)$, and it is distributed along the first two dimensions. The local slices can be inspected as

```
>>> print(comm.Get_rank(), T.local_slice(False))
0 [slice(0, 7, None), slice(0, 8, None), slice(0, 16, None)]
1 [slice(0, 7, None), slice(8, 15, None), slice(0, 16, None)]
2 [slice(7, 14, None), slice(0, 8, None), slice(0, 16, None)]
3 [slice(7, 14, None), slice(8, 15, None), slice(0, 16, None)]
```

Since two directions are distributed, both in spectral and real space, we say that we have a two-dimensional decomposition (here a 2×2 shaped processor mesh) and the MPI distribution is of type *pencil*. It is also possible to choose a *slab* decomposition, where only one dimension of the array is distributed. This choice needs to be made when creating the tensorproductspace as

```
T = TensorProductSpace(comm, (SD, K1, K2), axes=(0, 1, 2), slab=True)
```

which will lead to a mesh that is distributed along x-direction in real space and y-direction in spectral space. The local slices are

```
>>> print(comm.Get_rank(), T.local_slice()) # spectral space
1 [slice(0, 14, None), slice(4, 8, None), slice(0, 9, None)]
2 [slice(0, 14, None), slice(8, 12, None), slice(0, 9, None)]
0 [slice(0, 14, None), slice(0, 4, None), slice(0, 9, None)]
3 [slice(0, 14, None), slice(12, 15, None), slice(0, 9, None)]
>>> print(comm.Get_rank(), T.local_slice(False)) # real space
3 [slice(11, 14, None), slice(0, 15, None), slice(0, 16, None)]
0 [slice(0, 4, None), slice(0, 15, None), slice(0, 16, None)]
2 [slice(8, 11, None), slice(0, 15, None), slice(0, 16, None)]
1 [slice(4, 8, None), slice(0, 15, None), slice(0, 16, None)]
```

Note that the *slab* decomposition is usually the fastest choice. However, the maximum number of processors with *slab* is $\min\{N_0, N_1\}$, whereas a *pencil* approach can be used with up to $\min\{N_1(N_2/2+1), N_0 N_1\}$ processors.

Variational formulation

The variational problem (7.56) can be assembled using `shenfun`'s form language, which is perhaps surprisingly similar to FEniCS.

```
u = TrialFunction(T)
v = TestFunction(T)
K = T.local_wavenumbers()
# Get f on quad points
fj = Array(T, buffer=fl(*X))
# Compute right hand side of Poisson equation
```

(continues on next page)

(continued from previous page)

```
f_hat = inner(v, fj)
# Get left hand side of Poisson equation
matrices = inner(v, div(grad(u)))
```

The Laplacian operator is recognized as `div(grad)`. The `matrices` object is a dictionary representing the left hand side of (7.64), and there are two keys: (`ADDmat`, `BDDmat`). The value of `matrices["ADDmat"]` is an object of type `SpectralMatrix`, which is `shenfun`'s type for a matrix. This matrix represents A_{lj}, see (7.64), and it has an attribute `scale` that is equal to $(2\pi)^2$ (also see (7.64)). The other key in matrices is `BDDmat`, and the value here is a `SpectralMatrix` representing B_{lj} from (7.64). This matrix has an attribute `scale` that is equal to $m^2 + n^2$. This `scale` is stored as a numpy array of shape $(1, 15, 9)$, representing the set $\{m^2 + n^2 : (m, n) \in \boldsymbol{m}^{N_1} \times \boldsymbol{n}^{N_2}\}$. Note that $\boldsymbol{n}^{N_2}$ is stored simply as an array of length $N_2/2 + 1$ (here 9), since the transform in direction z takes a real signal and transforms it taking advantage of Hermitian symmetry, see rfft.

Solve linear equations

Finally, solve linear equation system and transform solution from spectral $\hat{u}_k$ vector to the real space $u(\boldsymbol{x})$ and then check how the solution corresponds with the exact solution u_e.

```
# Create Helmholtz linear algebra solver
H = Solver(*matrices)

# Solve and transform to real space
u_hat = Function(T)           # Solution spectral space
u_hat = H(u_hat, f_hat)       # Solve
uq = T.backward(u_hat)

# Compare with analytical solution
uj = ul(*X)
error = comm.reduce(np.linalg.norm(uj-uq)**2)
if comm.Get_rank() == 0:
    print("Error=%2.16e" %(np.sqrt(error)))
```

Convergence test

A complete solver is given in Sec. *Complete solver*. This solver is created such that it takes in two commandline arguments and prints out the L_2-errornorm of the solution in the end. We can use this to write a short script that performs a convergence test. The solver is run like

```
>>> python dirichlet_poisson3D.py 32 legendre
Error=6.5955040031498912e-10
```

for a discretization of size $\boldsymbol{N} = N^3 = 32^3$ and for the Legendre basis. Alternatively, change `legendre` to `chebyshev` for the Chebyshev basis.

We set up the solver to run for a list of $N = [8, 10, \ldots, 38]$, and collect the errornorms in arrays to be plotted. Such a script can be easily created with the subprocess module

```
import subprocess
from numpy import log, array
from matplotlib import pyplot as plt

N = range(8, 40, 2)
error = {}
```

(continues on next page)

(continued from previous page)

```
for basis in ('legendre', 'chebyshev'):
    error[basis] = []
    for i in range(len(N)):
        output = subprocess.check_output("python dirichlet_poisson3D.py {} {}".
↪format(N[i], basis), shell=True)
        exec(output) # Error is printed as "Error=%2.16e"%(np.linalg.norm(uj-ua))
        error[basis].append(Error)
        if i == 0:
            print("Error          hmin           r       ")
            print("%2.8e %2.8e %2.8f"%(error[basis][-1], 1./N[i], 0))
        if i > 0:
            print("%2.8e %2.8e %2.8f"%(error[basis][-1], 1./N[i], log(error[basis][-
↪1]/error[basis][-2])/log(N[i-1]/N[i])))
```

The error can be plotted using matplotlib, and the generated figure is shown in the summary's Fig. *Convergence of 3D Poisson solvers for both Legendre and Chebyshev modified basis function*. The spectral convergence is evident and we can see that after $N = 25$ roundoff errors dominate as the errornorm trails off around 10^{-13}.

```
plt.figure(figsize=(6, 4))
for basis, col in zip(('legendre', 'chebyshev'), ('r', 'b')):
    plt.semilogy(N, error[basis], col, linewidth=2)
plt.title('Convergence of Poisson solvers 3D')
plt.xlabel('N')
plt.ylabel('Error norm')
plt.legend(('Legendre', 'Chebyshev'))
plt.savefig('poisson3D_errornorm.png')
plt.show()
```

Complete solver

A complete solver, that can use either Legendre or Chebyshev bases, and any quadrature size chosen as a command-line argument, is shown below.

```
>>> python dirichlet_poisson3D.py 36 legendre
```

or similarly with `chebyshev` instead of `legendre`.

```
import sys, os
import importlib
from sympy import symbols, cos, sin, lambdify
import numpy as np
from shenfun import inner, div, grad, TestFunction, TrialFunction, Array, \
    Function, Basis, TensorProductSpace
import time
from mpi4py import MPI
try:
    import matplotlib.pyplot as plt
except ImportError:
    plt = None

comm = MPI.COMM_WORLD

assert len(sys.argv) == 3
assert sys.argv[-1].lower() in ('legendre', 'chebyshev')
assert isinstance(int(sys.argv[-2]), int)
```

(continues on next page)

(continued from previous page)

```
# Collect basis and solver from either Chebyshev or Legendre submodules
family = sys.argv[-1].lower()
base = importlib.import_module('.'.join(('shenfun', family)))
Solver = base.la.Helmholtz

# Use sympy to compute a rhs, given an analytical solution
a = -0
b = 0
x, y, z = symbols("x,y,z")
ue = (cos(4*x) + sin(2*y) + sin(4*z))*(1-z**2) + a*(1 + z)/2. + b*(1 - z)/2.
fe = ue.diff(x, 2) + ue.diff(y, 2) + ue.diff(z, 2)

# Lambdify for faster evaluation
ul = lambdify((x, y, z), ue, 'numpy')
fl = lambdify((x, y, z), fe, 'numpy')

# Size of discretization
N = int(sys.argv[-2])
N = [N, N, N]

SD = Basis(N[0], family=family, bc=(a, b))
K1 = Basis(N[1], family='F', dtype='D')
K2 = Basis(N[2], family='F', dtype='d')
T = TensorProductSpace(comm, (K1, K2, SD), axes=(0, 1, 2), slab=True)
X = T.local_mesh()
u = TrialFunction(T)
v = TestFunction(T)

K = T.local_wavenumbers()

# Get f on quad points
fj = Array(T, buffer=fl(*X))

# Compute right hand side of Poisson equation
f_hat = inner(v, fj)
if family == 'legendre':
    f_hat *= -1.

# Get left hand side of Poisson equation
if family == 'chebyshev':
    matrices = inner(v, div(grad(u)))
else:
    matrices = inner(grad(v), grad(u))

# Create Helmholtz linear algebra solver
H = Solver(*matrices)

# Solve and transform to real space
u_hat = Function(T)           # Solution spectral space
t0 = time.time()
u_hat = H(u_hat, f_hat)       # Solve
uq = T.backward(u_hat, fast_transform=False)

# Compare with analytical solution
uj = ul(*X)
error = comm.reduce(np.linalg.norm(uj-uq)**2)
```

(continues on next page)

(continued from previous page)

```
if comm.Get_rank() == 0:
    print("Error=%2.16e" %(np.sqrt(error)))
```

7.4 Demo - Helmholtz equation in polar coordinates

Authors Mikael Mortensen (mikaem at math.uio.no)

Date Jun 7, 2020

Summary. This is a demonstration of how the Python module shenfun can be used to solve the Helmholtz equation on a circular disc, using polar coordinates. This demo is implemented in a single Python file unitdisc_helmholtz.py, and the numerical method is described in more detail by J. Shen [She97].

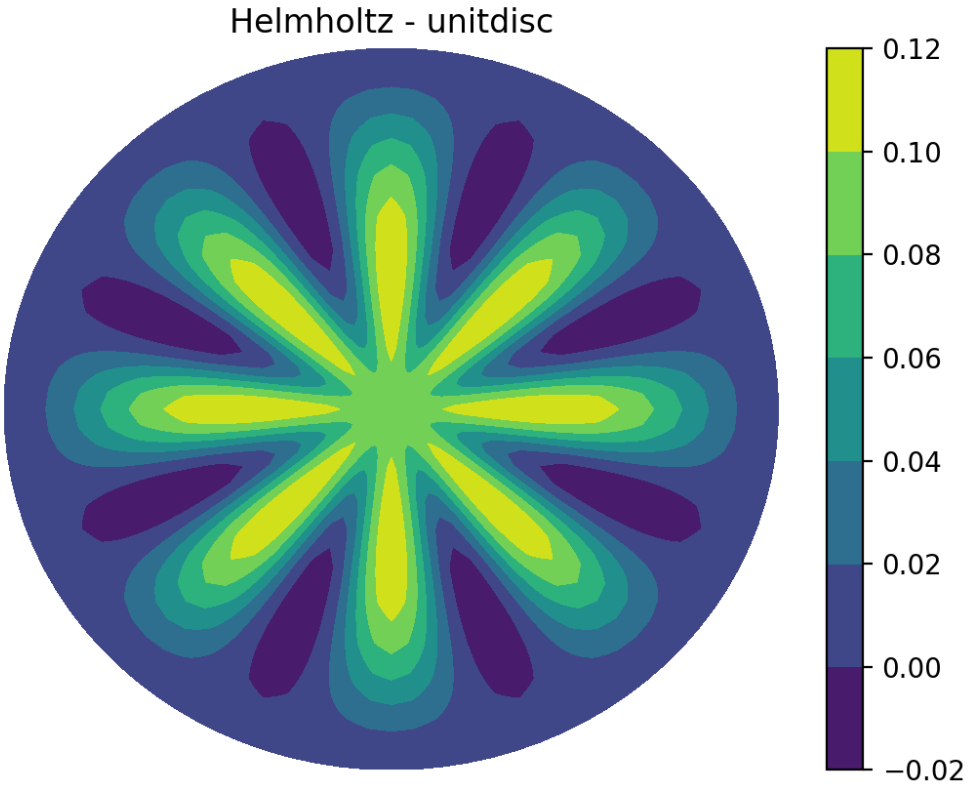


Fig. 3: *Helmholtz on the unit disc*

7.4.1 Helmholtz equation

The Helmholtz equation is given as

$$-\nabla^2 u(\boldsymbol{x}) + \alpha u(\boldsymbol{x}) = f(\boldsymbol{x}) \quad \text{for } \boldsymbol{x} = (x, y) \in \Omega, \tag{7.67}$$

$$u = 0 \text{ on } \partial\Omega, \tag{7.68}$$

where $u(\boldsymbol{x})$ is the solution, $f(\boldsymbol{x})$ is a function and α a constant. The domain is a circular disc $\Omega = \{(x, y) : x^2 + y^2 < a^2\}$ with radius a. We use polar coordinates (θ, r), defined as

$$x = r\cos\theta, \tag{7.69}$$

$$y = r\sin\theta, \tag{7.70}$$

which leads to a Cartesian product mesh $(\theta, r) \in [0, 2\pi) \times [0, a]$ suitable for numerical implementations. Note that the two directions are ordered with θ first and then r, which is less common than (r, θ). This has to do with the fact that we will need to solve linear equation systems along the radial direction, but not the θ-direction, since Fourier matrices are diagonal. When the radial direction is placed last, the data in the radial direction will be contigeous in a row-major C memory, leading to faster memory access where it is needed the most. Note that it takes very few changes in `shenfun` to switch the directions to (r, θ) if this is still desired.

We will use Chebyshev or Legendre basis functions $\psi_j(r)$ for the radial direction and a periodic Fourier expansion in $\exp(\imath k\theta)$ for the azimuthal direction. The polar basis functions are as such

$$v_{kj}(\theta, r) = \exp(\imath k\theta)\psi_j(r), \tag{7.71}$$

and we look for solutions

$$u(\theta, r) = \sum_k \sum_j \hat{u}_{kj} v_{kj}(\theta, r). \tag{7.72}$$

A discrete Fourier approximation space with N basis functions is then

$$V_F^N = \text{span}\{\exp(\imath k\theta)\}, \text{ for } k \in K, \tag{7.73}$$

where $K = \{-N/2, -N/2 + 1, \ldots, N/2 - 1\}$. Since the solution $u(\theta, r)$ is real, there is Hermitian symmetry and $\hat{u}_{k,j} = \hat{u}^*_{k,-j}$ (with $*$ denoting a complex conjugate). For this reason we use only $k \in K = \{0, 1, \ldots, N/2\}$ in solving for $\hat{u}_{kj}$, and then use Hermitian symmetry to get the remaining unknowns.

The radial basis is more tricky, because there is a nontrivial 'boundary' condition (pole condition) that needs to be applied at the center of the disc $(r = 0)$

$$\frac{\partial u(\theta, 0)}{\partial \theta} = 0. \tag{7.74}$$

To apply this condition we split the solution into Fourier coefficients with wavenumber 0 and $K\backslash\{0\}$, remembering that the Fourier basis function with $k = 0$ is simply 1

$$u(\theta, r) = \sum_j \left(\hat{u}_{0j}\psi_j(r) + \sum_{k=1}^{N/2} \hat{u}_{kj} \exp(\imath k\theta)\psi_j(r) \right). \tag{7.75}$$

We then apply a different radial basis for the two ψ's in the above equation (renaming the first $\overline{\psi}$)

$$u(\theta, r) = \sum_j \left(\hat{u}_{0j}\overline{\psi}_j(r) + \sum_{k=1}^{N/2} \hat{u}_{kj} \exp(\imath k\theta)\psi_j(r) \right). \tag{7.76}$$

Note that the first term $\sum_j \hat{u}_{0j}\overline{\psi}_j(r)$ is independent of θ. Now, to enforce conditions

$$u(\theta, a) = 0, \tag{7.77}$$

$$\frac{\partial u(\theta, 0)}{\partial \theta} = 0, \tag{7.78}$$

it is sufficient for the two bases ($\overline{\psi}$ and ψ) to satisfy

$$\overline{\psi}_j(a) = 0, \tag{7.79}$$

$$\psi_j(a) = 0, \tag{7.80}$$

$$\psi_j(0) = 0. \tag{7.81}$$

Bases that satisfy these conditions can be found both with Legendre and Chebyshev polynomials. If $\phi_j(x)$ is used for either the Legendre polynomial $L_j(x)$ or the Chebyshev polynomial of the first kind $T_j(x)$, we can have

$$\overline{\psi}_j(r) = \phi_j(2r/a - 1) - \phi_{j+1}(2r/a - 1), \text{ for } j \in 0, 1, \ldots N - 1, \tag{7.82}$$

$$\psi_j(r) = \phi_j(2r/a - 1) - \phi_{j+2}(2r/a - 1), \text{ for } j \in 0, 1, \ldots N - 2. \tag{7.83}$$

Define the following approximation spaces for the radial direction

$$V_D^N = \text{span}\{\psi_j\}_{j=0}^{N-3} \tag{7.84}$$

$$V_U^N = \text{span}\{\overline{\psi}_j\}_{j=0}^{N-2} \tag{7.85}$$

and split the function space for the azimuthal direction into

$$V_F^0 = \text{span}\{1\}, \tag{7.86}$$

$$V_F^1 = \text{span}\{\exp(\imath k\theta)\}, \text{ for } k \in K\backslash\{0\}. \tag{7.87}$$

We then look for solutions

$$u(\theta, r) = u^0(r) + u^1(\theta, r), \tag{7.88}$$

where

$$u^0(r) = \sum_{j=0}^{N-2} \hat{u}_j^0 \overline{\psi}_j(r), \tag{7.89}$$

$$u^1(\theta, r) = \sum_{j=0}^{N-3} \sum_{k=1}^{N/2} \hat{u}_{kj}^1 \exp(\imath k\theta)\psi_j(r). \tag{7.90}$$

As such the Helmholtz problem is split in two smaller problems. The two problems read with the spectral Galerkin method:

Find $u^0 \in V_F^0 \otimes V_U^N$ such that

$$\int_\Omega (-\nabla^2 u^0 + \alpha u^0) v^0 w d\sigma = \int_\Omega f v^0 w d\sigma, \quad \forall\, v^0 \in V_F^0 \otimes V_U^N. \tag{7.91}$$

Find $u^1 \in V_F^1 \otimes V_D^N$ such that

$$\int_\Omega (-\nabla^2 u^1 + \alpha u^1) v^1 w d\sigma = \int_\Omega f v^1 w d\sigma, \quad \forall\, v^1 \in V_F^1 \otimes V_D^N. \tag{7.92}$$

Note that integration over the domain is done using polar coordinates with an integral measure of $d\sigma = r dr d\theta$. However, the integral in the radial direction needs to be mapped to $t = 2r/a - 1$, where $t \in [-1, 1]$, which suits the basis functions used, see (7.83). This leads to a measure of $0.5(t + 1)a dt d\theta$. Furthermore, the weight $w(t)$ will be unity for the Legendre basis and $(1 - t^2)^{-0.5}$ for the Chebyshev bases.

7.4.2 Implementation in shenfun

A complete implementation is found in the file unitdisc_helmholtz.py. Here we give a brief explanation for the implementation. Start by importing all functionality from shenfun and sympy, where Sympy is required for handeling the polar coordinates.

```
from shenfun import *
import sympy as sp

# Define polar coordinates using angle along first axis and radius second
theta, r = psi = sp.symbols('x,y', real=True, positive=True)
rv = (r*sp.cos(theta), r*sp.sin(theta)) # Map to Cartesian (x, y)
```

Note that Sympy symbols are both positive and real, θ is chosen to be along the first axis and r second. This has to agree with the next step, which is the creation of tensorproductspaces $V_F^0 \otimes V_U^N$ and $V_F^1 \otimes V_D^N$. We use `domain=(0, 1)` for the radial direction to get a unit disc, whereas the default domain for the Fourier bases is already the required $(0, 2\pi)$.

```
N = 32
F = Basis(N, 'F', dtype='d')
F0 = Basis(1, 'F', dtype='d')
L = Basis(N, 'L', bc='Dirichlet', domain=(0, 1))
L0 = Basis(N, 'L', bc='UpperDirichlet', domain=(0, 1))
T = TensorProductSpace(comm, (F, L), axes=(1, 0), coordinates=(psi, rv))
T0 = TensorProductSpace(MPI.COMM_SELF, (F0, L0), axes=(1, 0), coordinates=(psi, rv))
```

Note that since `F0` only has one component we could actually use `L0` without creating `T0`. But the code turns out to be simpler if we use `T0`, much because the additional θ-direction is required for the polar coordinates to apply. Using one single basis function for the θ direction is as such a generic way to handle polar 1D problems (i.e., problems that are only functions of the radial direction, but still using polar coordinates). Also note that `F` is created using the entire range of wavenumbers even though it should not include wavenumber 0. As such we need to make sure that the coefficient created for $k = 0$ (i.e., $\hat{u}^1_{0,j}$) will be exactly zero. Finally, note that `T0` is not distributed with MPI, which is accomplished using `MPI.COMM_SELF` instead of `comm` (which equals `MPI.COMM_WORLD`). The purely radial problem (**??**) is only solved on the one processor with rank = 0.

Polar coordinates are ensured by feeding `coordinates=(psi, rv)` to `TensorProductSpace`. Operators like `div()` `grad()` and `curl()` will now work on items of `Function`, `TestFunction` and `TrialFunction` using a polar coordinate system.

To define the equations (**??**) and (7.92) we first declare these test- and trialfunctions, and then use code that is remarkably similar to the mathematics.

```
v = TestFunction(T)
u = TrialFunction(T)
v0 = TestFunction(T0)
u0 = TrialFunction(T0)

mats = inner(v, -div(grad(u))+alpha*u)
if comm.Get_rank() == 0:
    mats0 = inner(v0, -div(grad(u0))+alpha*u0)
```

Here `mats` and `mats0` will contain several tensor product matrices in the form of `TPMatrix`. Since there is only one non-periodic direction the matrices can be easily solved using `SolverGeneric1NP`. But first we need to define the function $f(\theta, r)$. To this end we use sympy and the method of manufactured solution to define a possible solution `ue`, and then compute `f` exactly using exact differentiation

```
# Manufactured solution
alpha = 2
ue = (r*(1-r))**2*sp.cos(8*theta)-0.1*(r-1)
f = -ue.diff(r, 2) - (1/r)*ue.diff(r, 1) - (1/r**2)*ue.diff(theta, 2) + alpha*ue

# Compute the right hand side on the quadrature mesh
fj = Array(T, buffer=f)

# Take scalar product
f_hat = Function(T)
f_hat = inner(v, fj, output_array=f_hat)
if T.local_slice(True)[0].start == 0: # The processor that owns k=0
    f_hat[0] = 0

# For k=0 we solve only a 1D equation. Do the scalar product for Fourier
# coefficient 0 by hand (or sympy).
if comm.Get_rank() == 0:
    f0_hat = Function(T0)
    gt = sp.lambdify(r, sp.integrate(f, (theta, 0, 2*sp.pi))/2/sp.pi)(L0.mesh())
    f0_hat = L0.scalar_product(gt, f0_hat)
```

Note that for u^0 we perform the interal in the θ direction exactly using sympy. This is necessary since one Fourier coefficient is not sufficient to do this integral numerically. For the u^1 case we do the integral numerically as part of the `inner` product. With the correct right hand side assembled we can solve the linear system of equations

```
u_hat = Function(T)
Sol1 = SolverGeneric1NP(mats)
u_hat = Sol1(f_hat, u_hat)

# case k = 0
u0_hat = Function(T0)
if comm.Get_rank() == 0:
    Sol0 = SolverGeneric1NP(mats0)
    u0_hat = Sol0(f0_hat, u0_hat)
comm.Bcast(u0_hat, root=0)
```

Having found the solution in spectral space all that is left is to transform it back to real space.

```
# Transform back to real space. Broadcast 1D solution
sl = T.local_slice(False)
uj = u_hat.backward() + u0_hat.backward()[:, sl[1]]
```

7.4.3 Postprocessing

The solution can now be compared with the exact solution through

```
ue = Array(T, buffer=ue)
print('Error =', np.linalg.norm(uj-ue))
# ---> Error = 7.45930806417765e-15
```

We can also get the gradient of the solution. For this we need a space without boundary conditions, and a vector space

```
TT = T.get_orthogonal()
V = VectorTensorProductSpace(TT)
```

Notice that we do not have the solution in one single space in spectral space, since it is a combination of `u_hat` and `u0_hat`. For this reason we first transform the solution from real space `uj` to the new orthogonal space `TT`

```
ua = Array(TT, buffer=uj)
uh = ua.forward()
```

With the solution as a `Function` we can simply project the gradient to `V`

```
dv = project(grad(uh), V)
du = dv.backward()
```

Note that the gradient `du` now contains the contravariant components of the covariant basis vector `b`. The basis vector `b` is not normalized (it's length is not unity).

```
b = T.coors.get_covariant_basis()
```

The basis vectors are, in fact

$$\mathbf{b}_\theta = -r\sin(\theta)\,\mathbf{i} + r\cos(\theta)\,\mathbf{j}$$
$$\mathbf{b}_r = \cos(\theta)\,\mathbf{i} + \sin(\theta)\,\mathbf{j}$$

and we see that they are given in terms of the Cartesian unit vectors. The gradient we have computed is (and yes, it should be r^2 because we do not have unit vectors)

$$\nabla u = \underbrace{\frac{1}{r^2}\frac{\partial u}{\partial \theta}}_{du[0]}\mathbf{b}_\theta + \underbrace{\frac{\partial u}{\partial r}}_{du[1]}\mathbf{b}_r \tag{7.93}$$

Now it makes sense to plot the solution and its gradient in Cartesian instead of computational coordinates. To this end we need to project the gradient to a Cartesian basis

$$\frac{\partial u}{\partial x} = \nabla u \cdot \mathbf{i},$$
$$\frac{\partial u}{\partial y} = \nabla u \cdot \mathbf{j}.$$

We compute the Cartesian gradient by assembling (7.93) on the computational grid

```
ui, vi = TT.local_mesh(True)
bij = np.array(sp.lambdify(psi, b)(ui, vi))
gradu = du[0]*bij[0] + du[1]*bij[1]
```

Because of the way the vectors are stored, `gradu[0]` will now contain $\nabla u \cdot \mathbf{i}$ and `gradu[1]` will contain $\nabla u \cdot \mathbf{j}$. To validate we compute the exact gradient and compute the error norm

```
gradue = Array(V, buffer=list(b[0]*ue.diff(theta, 1)/r**2 + b[1]*ue.diff(r, 1)))
#or alternatively
#gradue = Array(V, buffer=grad(u).tosympy(basis=ue, psi=psi))
print('Error gradient', np.linalg.norm(gradu-gradue))
# ---> Error gradient 1.0856774538980375e-08
```

We now refine the solution to make it look better, and plot on the unit disc.

```
u_hat2 = u_hat.refine([N*3, N*3])
u0_hat2 = u0_hat.refine([1, N*3])
sl = u_hat2.function_space().local_slice(False)
ur = u_hat2.backward() + u0_hat2.backward()[:, sl[1]]
```

(continues on next page)

(continued from previous page)

```
# Wrap periodic plot around since it looks nicer
xx, yy = u_hat2.function_space().local_curvilinear_mesh()
xp = np.vstack([xx, xx[0]])
yp = np.vstack([yy, yy[0]])
up = np.vstack([ur, ur[0]])
# For vector no need to wrap around and no need to refine:
xi, yi = TT.local_curvilinear_mesh()

# plot
plt.figure()
plt.contourf(xp, yp, up)
plt.quiver(xi, yi, gradu[0], gradu[1], scale=40, pivot='mid', color='white')
plt.colorbar()
plt.title('Helmholtz - unitdisc')
plt.xticks([])
plt.yticks([])
plt.axis('off')
plt.show()
```

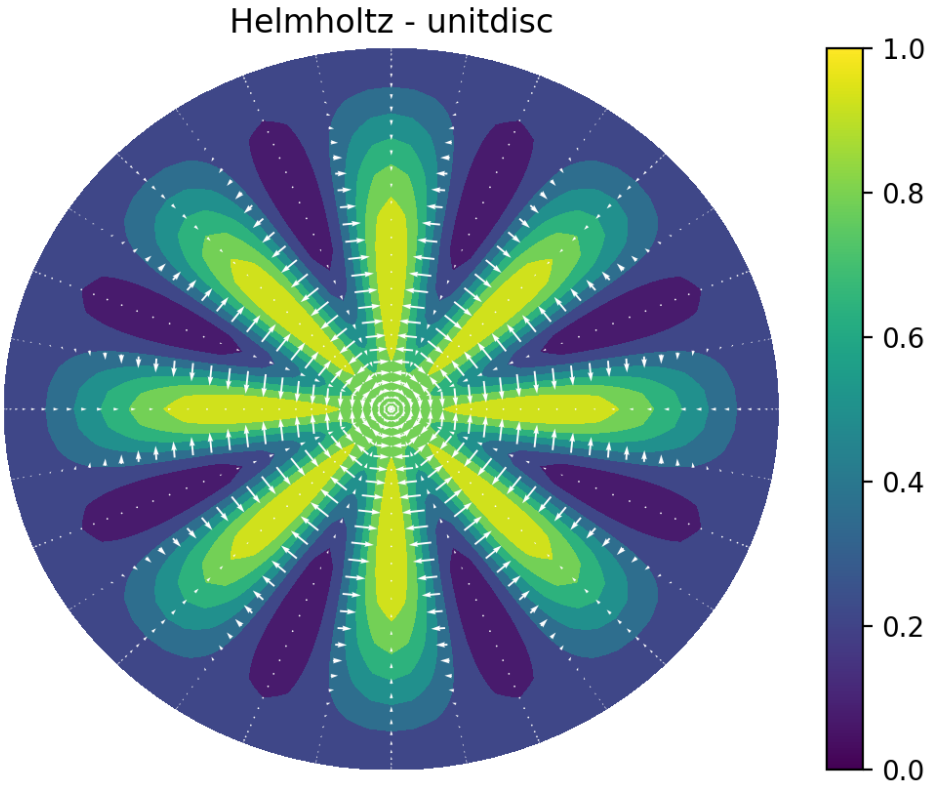


Fig. 4: Solution of Helmholtz equation, with gradient

7.5 Demo - Kuramato-Sivashinsky equation

Authors Mikael Mortensen (mikaem at math.uio.no)

Date Jun 7, 2020

Summary. This is a demonstration of how the Python module shenfun can be used to solve the time-dependent, nonlinear Kuramato-Sivashinsky equation, in a doubly periodic domain. The demo is implemented in a single Python file KuramatoSivashinsky.py, and it may be run in parallel using MPI.

7.5.1 The Kuramato-Sivashinsky equation

Movie showing the evolution of the solution u from Eq. (7.94).

Model equation

The Kuramato-Sivashinsky (KS) equation is known for its chaotic bahaviour, and it is often used in study of turbulence or turbulent combustion. We will here solve the KS equation in a doubly periodic domain $[-30\pi, 30\pi]^2$, starting from a single Gaussian pulse

$$\frac{\partial u(\boldsymbol{x},t)}{\partial t} + \nabla^2 u(\boldsymbol{x},t) + \nabla^4 u(\boldsymbol{x},t) + |\nabla u(\boldsymbol{x},t)|^2 = 0 \quad \text{for } \boldsymbol{x} \in \Omega = [-30\pi, 30\pi]^2 \tag{7.94}$$

$$u(\boldsymbol{x},0) = \exp(-0.01\boldsymbol{x}\cdot\boldsymbol{x})$$

Spectral Galerkin method

The PDE in (7.94) can be solved with many different numerical methods. We will here use the shenfun software and this software makes use of the spectral Galerkin method. Being a Galerkin method, we need to reshape the governing equations into proper variational forms, and this is done by multiplying (7.94) with the complex conjugate of a proper test function and then integrating over the domain. To this end we use testfunction $v \in V(\Omega)$, where $V(\Omega)$ is some suitable function space, and obtain

$$\frac{\partial}{\partial t}\int_\Omega u\,\overline{v}\,w\,dx = -\int_\Omega \left(\nabla^2 u + \nabla^4 u\ + |\nabla u|^2\right)\overline{v}\,w\,dx. \tag{7.95}$$

Note that the overline is used to indicate a complex conjugate, whereas w is a weight function. The function u is now to be considered a trial function, and the integrals over the domain are often referred to as inner products. With inner product notation

$$(u, v) = \int_\Omega u\,\overline{v}\,w\,dx.$$

the variational problem can be formulated as

$$\frac{\partial}{\partial t}(u, v) = -\left(\nabla^2 u + \nabla^4 u + |\nabla u|^2, v\right). \tag{7.96}$$

The space and time discretizations are still left open. There are numerous different approaches that one could take for discretizing in time. Here we will use a fourth order exponential Runge-Kutta method.

Discretization

We discretize the model equation in space using continuously differentiable Fourier basis functions

$$\phi_l(x) = e^{\imath \underline{l} x}, \quad -\infty < l < \infty, \tag{7.97}$$

where l is the wavenumber, and $\underline{l} = \frac{2\pi}{L} l$ is the scaled wavenumber, scaled with domain length L (here 60π). Since we want to solve these equations on a computer, we need to choose a finite number of test functions. A discrete function space V^N can be defined as

$$V^N(x) = \text{span}\{\phi_l(x)\}_{l \in \boldsymbol{l}}, \tag{7.98}$$

where N is chosen as an even positive integer and $\boldsymbol{l} = (-N/2, -N/2+1, \ldots, N/2-1)$. And now, since Ω is a two-dimensional domain, we can create a tensor product of two such one-dimensional spaces:

$$W^{\boldsymbol{N}}(x, y) = V^N(x) \otimes V^N(y), \tag{7.99}$$

where $\boldsymbol{N} = (N, N)$. Obviously, it is not necessary to use the same number (N) of basis functions for each direction, but it is done here for simplicity. A 2D tensor product basis function is now defined as

$$\Phi_{lm}(x, y) = e^{\imath \underline{l} x} e^{\imath \underline{m} y} = e^{\imath (\underline{l} x + \underline{m} y)}, \tag{7.100}$$

where the indices for y-direction are $\underline{m} = \frac{2\pi}{L} m$, and $\boldsymbol{m}$ is the same set as $\boldsymbol{l}$ due to using the same number of basis functions for each direction. One distinction, though, is that for the y-direction expansion coefficients are only stored for $m = (0, 1, \ldots, N/2)$ due to Hermitian symmetry (real input data).

We now look for solutions of the form

$$u(x, y) = \sum_{l=-N/2}^{N/2-1} \sum_{m=-N/2}^{N/2-1} \hat{u}_{lm} \Phi_{lm}(x, y). \tag{7.101}$$

The expansion coefficients $\hat{u}_{lm}$ can be related directly to the solution $u(x, y)$ using Fast Fourier Transforms (FFTs) if we are satisfied with obtaining the solution in quadrature points corresponding to

$$x_i = \frac{60\pi i}{N} - 30\pi \quad \forall\, i \in \boldsymbol{i}, \text{where } \boldsymbol{i} = (0, 1, \ldots, N-1), \tag{7.102}$$

$$y_j = \frac{60\pi j}{N} - 30\pi \quad \forall\, j \in \boldsymbol{j}, \text{where } \boldsymbol{j} = (0, 1, \ldots, N-1). \tag{7.103}$$

Note that these points are different from the standard (like $2\pi j/N$) since the domain is set to $[-30\pi, 30\pi]^2$ and not the more common $[0, 2\pi]^2$. We now have

$$u(x_i, y_j) = \mathcal{F}_y^{-1}\left(\mathcal{F}_x^{-1}\left(\hat{u}\right)\right) \forall\, (i, j) \in \boldsymbol{i} \times \boldsymbol{j}, \tag{7.104}$$

where $\mathcal{F}_x^{-1}$ is the inverse Fourier transform along direction x, for all $j \in \boldsymbol{j}$. Note that the two inverse FFTs are performed sequentially, one direction at the time, and that there is no scaling factor due the definition used for the inverse Fourier transform:

$$u(x_j) = \sum_{l=-N/2}^{N/2-1} \hat{u}_l e^{\imath \underline{l} x_j}, \quad \forall\, j \in \boldsymbol{j}. \tag{7.105}$$

Note that this differs from the definition used by, e.g., Numpy.

The inner products used in Eq. (7.96) may be computed using forward FFTs (using weight functions $w = 1/L$):

$$(u, \Phi_{lm}) = \hat{u}_{lm} = \frac{1}{N^2} \mathcal{F}_l\left(\mathcal{F}_m\left(u\right)\right) \quad \forall (l, m) \in \boldsymbol{l} \times \boldsymbol{m}, \tag{7.106}$$

From this we see that the variational forms may be written in terms of the Fourier transformed $\hat{u}$. Expanding the exact derivatives of the nabla operator, we have

$$(\nabla^2 u, v) = -(\underline{l}^2 + \underline{m}^2)\hat{u}_{lm}, \tag{7.107}$$

$$(\nabla^4 u, v) = (\underline{l}^2 + \underline{m}^2)^2\hat{u}_{lm}, \tag{7.108}$$

$$(|\nabla u|^2, v) = \widehat{|\nabla u|^2}_{lm} \tag{7.109}$$

and as such the equation to be solved for each wavenumber can be found directly as

$$\frac{\partial \hat{u}_{lm}}{\partial t} = \left(\underline{l}^2 + \underline{m}^2 - (\underline{l}^2 + \underline{m}^2)^2\right)\hat{u}_{lm} - \widehat{|\nabla u|^2}_{lm}, \tag{7.110}$$

7.5.2 Implementation

The model equation (7.94) is implemented in shenfun using Fourier basis functions for both x and y directions. We start the solver by implementing necessary functionality from required modules like Numpy, Sympy matplotlib and mpi4py, in addition to shenfun:

```
from sympy import symbols, exp, lambdify
import numpy as np
import matplotlib.pyplot as plt
from mpi4py import MPI
from shenfun import *
```

The size of the problem (in real space) is then specified, before creating the `TensorProductSpace`, which is using a tensor product of two Fourier bases as basis functions. We also create a `VectorTensorProductSpace`, since this is required for computing the gradient of the scalar field `u`. The gradient is required for the nonlinear term.

```
# Size of discretization
N = (128, 128)

comm = MPI.COMM_WORLD
K0 = Basis(N[0], 'F', domain=(-30*np.pi, 30*np.pi), dtype='D')
K1 = Basis(N[1], 'F', domain=(-30*np.pi, 30*np.pi), dtype='d')
T = TensorProductSpace(comm, (K0, K1), **{'planner_effort': 'FFTW_MEASURE'})
TV = VectorTensorProductSpace([T, T])
```

Test and trialfunctions are required for assembling the variational forms:

```
u = TrialFunction(T)
v = TestFunction(T)
```

and some arrays are required to hold the solution. We also create an array `gradu`, that will be used to compute the gradient in the nonlinear term. Finally, the wavenumbers are collected in list `K`. Here one feature is worth mentioning. The gradient in spectral space can be computed as `1j*K*U_hat`. However, since this is an odd derivative, and we are using an even number `N` for the size of the domain, the highest wavenumber must be set to zero. This is the purpose of the last keyword argument to `local_wavenumbers` below.

```
U = Array(T)
U_hat = Function(T)
gradu = Array(TV)
K = np.array(T.local_wavenumbers(True, True, eliminate_highest_freq=True))
```

Note that using this `K` in computing derivatives has the same effect as achieved by symmetrizing the Fourier series by replacing the first sum below with the second when computing odd derivatives.

$$u = \sum_{k=-N/2}^{N/2-1} \hat{u} e^{\imath k x} \tag{7.111}$$

$$u = \sum_{k=-N/2}^{N/2}{}' \hat{u} e^{\imath k x} \tag{7.112}$$

Here $\sum'$ means that the first and last items in the sum are divided by two. Note that the two sums are equal as they stand (due to aliasing), but only the latter (known as the Fourier interpolant) gives the correct (zero) derivative of the basis with the highest wavenumber.

Sympy is used to generate an initial condition, as stated in Eq (7.94)

```
# Use sympy to set up initial condition
x, y = symbols("x,y")
ue = exp(-0.01*(x**2+y**2))
ul = lambdify((x, y), ue, 'numpy')
```

Shenfun has a few integrators implemented in the `integrators` submodule. Two such integrators are the 4th order explicit Runge-Kutta method `RK4`, and the exponential 4th order Runge-Kutta method `ETDRK4`. Both these integrators need two methods provided by the problem being solved, representing the linear and nonlinear terms in the problem equation. We define two methods below, called `LinearRHS` and `NonlinearRHS`

```
def LinearRHS(self):
    # Assemble diagonal bilinear forms
    L = -(inner(div(grad(u))+div(grad(div(grad(u)))), v))
    return L

def NonlinearRHS(self, U, U_hat, dU):
    # Assemble nonlinear term
    global gradu
    gradu = TV.backward(1j*K*U_hat, gradu)
    dU = T.forward(0.5*(gradu[0]*gradu[0]+gradu[1]*gradu[1]), dU)
    return -dU
```

The code should, hopefully, be self-explanatory.

All that remains now is to initialize the solution arrays and to setup the integrator plus some plotting functionality for visualizing the results. Note that visualization is only nice when running the code in serial. For parallel, it is recommended to use `HDF5File`, to store intermediate results to the HDF5 format, for later viewing in, e.g., Paraview.

The solution is initialized as

```
#initialize
X = T.local_mesh(True)
U[:] = ul(*X)
U_hat = T.forward(U, U_hat)
```

And we also create an update function for plotting intermediate results with a cool colormap:

```
# Integrate using an exponential time integrator
plt.figure()
cm = plt.get_cmap('hot')
image = plt.contourf(X[0], X[1], U, 256, cmap=cm)
```

(continues on next page)

(continued from previous page)

```
plt.draw()
plt.pause(1e-6)
count = 0
def update(u, u_hat, t, tstep, **params):
    global count
    if tstep % params['plot_step'] == 0 and params['plot_step'] > 0:
        u = T.backward(u_hat, u)
        image.ax.clear()
        image.ax.contourf(X[0], X[1], U, 256, cmap=cm)
        plt.pause(1e-6)
        count += 1
        plt.savefig('Kuramato_Sivashinsky_N_{}_{}.png'.format(N[0], count))
```

Now all that remains is to create the integrator and call it

```
if __name__ == '__main__':
    par = {'plot_step': 100}
    dt = 0.01
    end_time = 100
    integrator = ETDRK4(T, L=LinearRHS, N=NonlinearRHS, update=update, **par)
    #integrator = RK4(T, L=LinearRHS, N=NonlinearRHS, update=update, **par)
    integrator.setup(dt)
    U_hat = integrator.solve(U, U_hat, dt, (0, end_time))
```

7.6 Demo - Stokes equations

Authors Mikael Mortensen (mikaem at math.uio.no)

Date Jun 7, 2020

Summary. The Stokes equations describe the flow of highly viscous fluids. This is a demonstration of how the Python module shenfun can be used to solve Stokes equations using a mixed (coupled) basis in a 3D tensor product domain. We assume homogeneous Dirichlet boundary conditions in one direction and periodicity in the remaining two. The solver described runs with MPI without any further considerations required from the user. The solver assembles a block matrix with sparsity pattern as shown below for the Legendre basis.

7.6.1 Model problem

Stokes equations

The Stokes equations are given in strong form as

$$\begin{aligned} \nabla^2 \mathbf{u} - \nabla p &= \mathbf{f} \quad \text{in } \Omega, \\ \nabla \cdot \mathbf{u} &= h \quad \text{in } \Omega \\ \int_\Omega p dx &= 0 \end{aligned}$$

where $\mathbf{u}$ and p are, respectively, the fluid velocity vector and pressure, and the domain $\Omega = [0, 2\pi]^2 \times [-1, 1]$. The flow is assumed periodic in x and y-directions, whereas there is a no-slip homogeneous Dirichlet boundary condition on $\mathbf{u}$ on the boundaries of the z-direction, i.e., $\mathbf{u}(x, y, \pm 1) = (0, 0, 0)$. (Note that we can configure shenfun with non-periodicity in any of the three directions. However, since we are to solve linear algebraic systems in the non-periodic direction, there is a speed benefit from having the nonperiodic direction last. This has to do with Numpy

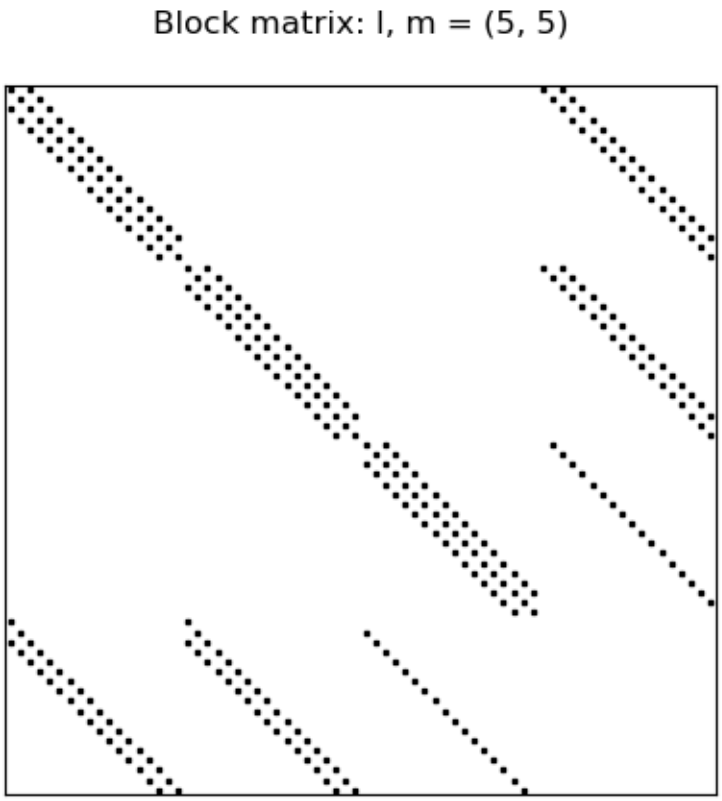


Fig. 5: *Coupled block matrix for Stokes equations*

using a C-style row-major storage of arrays by default.) The right hand side vector $\mathbf{f}(\mathbf{x})$ is an external applied body force. The right hand side h is usually zero in the regular Stokes equations. Here we include it because it will be nonzero in the verification, which is using the method of manufactured solutions. Note that the final $\int_\Omega p dx = 0$ is there because there is no Dirichlet boundary condition on the pressure and the system of equations would otherwise be ill conditioned.

To solve Stokes equations with the Galerkin method we need basis functions for both velocity and pressure. A Dirichlet basis will be used for velocity, whereas there is no boundary restriction on the pressure basis. For both three-dimensional bases we will use one basis function for the x-direction, $\mathcal{X}(x)$, one for the y-direction, $\mathcal{Y}(y)$, and one for the z-direction, $\mathcal{Z}(z)$. And then we create three-dimensional basis functions like

$$v(x, y, z) = \mathcal{X}(x)\mathcal{Y}(y)\mathcal{Z}(z). \tag{7.113}$$

The basis functions $\mathcal{X}(x)$ and $\mathcal{Y}(y)$ are chosen as Fourier exponentials, since these functions are periodic:

$$\mathcal{X}_l(x) = e^{\imath l x}, \forall\, l \in \mathbf{l}^{N_0}, \tag{7.114}$$

$$\mathcal{Y}_m(y) = e^{\imath m y}, \forall\, m \in \mathbf{m}^{N_1}, \tag{7.115}$$

where $\mathbf{l}^{N_0} = (-N_0/2, -N_0/2+1, \ldots, N_0/2-1)$ and $\mathbf{m}^{N_1} = (-N_1/2, -N_1/2+1, \ldots, N_1/2-1)$. The size of the discretized problem in real physical space is $\mathbf{N} = (N_0, N_1, N_2)$, i.e., there are $N_0 \cdot N_1 \cdot N_2$ quadrature points in total.

The basis functions for $\mathcal{Z}(z)$ remain to be decided. For the velocity we need homogeneous Dirichlet boundary conditions, and for this we use composite Legendre or Chebyshev polynomials

$$\mathcal{Z}^0_n(z) = \phi_n(z) - \phi_{n+2}(z), \forall\, n \in \mathbf{n}^{N_2-2}, \tag{7.116}$$

where ϕ_n is the n'th Legendre or Chebyshev polynomial of the first kind. $\mathbf{n}^{N_2-2} = (0, 1, \ldots, N_2 - 3)$, and the zero on $\mathcal{Z}^0$ is there to indicate the zero value on the boundary.

The pressure basis that comes with no restrictions for the boundary is a little trickier. The reason for this has to do with inf-sup stability. The obvious choice of basis is the regular Legendre or Chebyshev basis, which is denoted as

$$\mathcal{Z}_n(z) = \phi_n(z), \forall\, n \in \mathbf{n}^{N_2}. \tag{7.117}$$

The problem is that for the natural choice of $n \in (0, 1, \ldots, N_2 - 1)$ there is a nullspace and one degree of freedom remains unresolved. It turns out that the proper choice for the pressure basis is simply (7.117) for $n \in \mathbf{n}^{N_2-2}$. (Also remember that we have to fix $\int_\Omega p dx = 0$.)

With given basis functions we obtain the spaces

$$V^{N_0} = \text{span}\{\mathcal{X}_l\}_{l \in \mathbf{l}^{N_0}}, \tag{7.118}$$

$$V^{N_1} = \text{span}\{\mathcal{Y}_m\}_{m \in \mathbf{m}^{N_1}}, \tag{7.119}$$

$$V^{N_2} = \text{span}\{\mathcal{Z}_n\}_{n \in \mathbf{n}^{N_2-2}}, \tag{7.120}$$

$$V^{N_2}_0 = \text{span}\{\mathcal{Z}^0_n\}_{n \in \mathbf{n}^{N_2-2}}, \tag{7.121}$$

and from these we create two different tensor product spaces

$$W^{\mathbf{N}}_0(\mathbf{x}) = V^{N_0}(x) \otimes V^{N_1}(y) \otimes V^{N_2}_0(z), \tag{7.122}$$

$$W^{\mathbf{N}}(\mathbf{x}) = V^{N_0}(x) \otimes V^{N_1}(y) \otimes V^{N_2}(z). \tag{7.123}$$

The velocity vector is using a mixed basis, such that we will look for solutions $\mathbf{u} \in [W^{\mathbf{N}}_0]^3 (= W^{\mathbf{N}}_0 \times W^{\mathbf{N}}_0 \times W^{\mathbf{N}}_0)$, whereas we look for the pressure $p \in W^{\mathbf{N}}$. We now formulate a variational problem using the Galerkin method: Find $\mathbf{u} \in [W^{\mathbf{N}}_0]^3$ and $p \in W^{\mathbf{N}}$ such that

$$\int_\Omega (\nabla^2 \mathbf{u} - \nabla p) \cdot \overline{\mathbf{v}}\, dx_w = \int_\Omega \mathbf{f} \cdot \overline{\mathbf{v}}\, dx_w \quad \forall \mathbf{v} \in [W^{\mathbf{N}}_0]^3, \tag{7.124}$$

$$\int_\Omega \nabla \cdot \mathbf{u}\, \overline{q}\, dx_w = \int_\Omega h \overline{q}\, dx_w \quad \forall q \in W^{\mathbf{N}}. \tag{7.125}$$

Here $dx_w = w_x dx w_y dy w_z dz$ represents a weighted measure, with weights $w_x(x), w_y(y), w_z(z)$. Note that it is only Chebyshev polynomials that make use of a non-constant weight $w_x = 1/\sqrt{1-x^2}$. The Fourier weights are $w_y = w_z = 1/(2\pi)$ and the Legendre weight is $w_x = 1$. The overline in $\overline{\mathbf{v}}$ and $\overline{q}$ represents a complex conjugate, which is needed here because the Fourier exponentials are complex functions.

Mixed variational form

Since we are to solve for $\mathbf{u}$ and p at the same time, we formulate a mixed (coupled) problem: find $(\mathbf{u}, p) \in [W_0^{\mathbf{N}}]^3 \times W^{\mathbf{N}}$ such that

$$a((\mathbf{u}, p), (\mathbf{v}, q)) = L((\mathbf{v}, q)) \quad \forall (\mathbf{v}, q) \in [W_0^{\mathbf{N}}]^3 \times W^{\mathbf{N}}, \tag{7.126}$$

where bilinear (a) and linear (L) forms are given as

$$a((\mathbf{u}, p), (\mathbf{v}, q)) = \int_\Omega (\nabla^2 \mathbf{u} - \nabla p) \cdot \overline{\mathbf{v}}\, dx_w + \int_\Omega \nabla \cdot \mathbf{u}\, \overline{q}\, dx_w, \tag{7.127}$$

$$L((\mathbf{v}, q)) = \int_\Omega \mathbf{f} \cdot \overline{\mathbf{v}}\, dx_w + \int_\Omega h \overline{q}\, dx_w. \tag{7.128}$$

Note that the bilinear form will assemble to block matrices, whereas the right hand side linear form will assemble to block vectors.

7.6.2 Implementation

Preamble

We will solve the Stokes equations using the shenfun Python module. The first thing needed is then to import some of this module's functionality plus some other helper modules, like Numpy and Sympy:

```
import os
import sys
import numpy as np
from mpi4py import MPI
from sympy import symbols, sin, cos
from shenfun import *
```

We use `Sympy` for the manufactured solution and `Numpy` for testing. MPI for Python (`mpi4py`) is required for running the solver with MPI.

Manufactured solution

The exact solutions $\mathbf{u}_e(\mathbf{x})$ and $p(\mathbf{x})$ are chosen to satisfy boundary conditions, and the right hand sides $\mathbf{f}(\mathbf{x})$ and $h(\mathbf{x})$ are then computed exactly using `Sympy`. These exact right hand sides will then be used to compute a numerical solution that can be verified against the manufactured solution. The chosen solution with computed right hand sides are:

```
x, y, z = symbols('x,y,z')
uex = sin(2*y)*(1-z**2)
uey = sin(2*x)*(1-z**2)
uez = sin(2*z)*(1-z**2)
```

(continues on next page)

(continued from previous page)

```
pe = -0.1*sin(2*x)*cos(4*y)
fx = uex.diff(x, 2) + uex.diff(y, 2) + uex.diff(z, 2) - pe.diff(x, 1)
fy = uey.diff(x, 2) + uey.diff(y, 2) + uey.diff(z, 2) - pe.diff(y, 1)
fz = uez.diff(x, 2) + uez.diff(y, 2) + uez.diff(z, 2) - pe.diff(z, 1)
h = uex.diff(x, 1) + uey.diff(y, 1) + uez.diff(z, 1)
```

Bases and tensor product spaces

Bases are created using the `Basis()` function. A choice of polynomials between Legendre or Chebyshev can be made, and the size of the domain is given

```
N = (40, 40, 40)
family = 'Legendre'
K0 = Basis(N[0], 'Fourier', dtype='D', domain=(0, 2*np.pi))
K1 = Basis(N[1], 'Fourier', dtype='d', domain=(0, 2*np.pi))
SD = Basis(N[2], family, bc=(0, 0))
ST = Basis(N[2], family)
ST.slice = lambda: slice(0, ST.N-2)
```

Note that the last line of code is there to ensure that only the first $N_2 - 2$ coefficients are used, see discussion around Eq. (7.117). At the same time, we ensure that we are still using N_2 quadrature points, the same as for the Dirichlet basis.

Next the one-dimensional spaces are used to create two tensor product spaces Q = $W^{\mathbf{N}}$ and TD = $W_0^{\mathbf{N}}$, one vector V = $[W_0^{\mathbf{N}}]^3$ and one mixed space VQ = V $\times$ Q.

```
TD = TensorProductSpace(comm, (K0, K1, SD), axes=(2, 0, 1))
Q = TensorProductSpace(comm, (K0, K1, ST), axes=(2, 0, 1))
V = VectorTensorProductSpace(TD)
VQ = MixedTensorProductSpace([V, Q])
```

Note that we choose to transform axes in the order $1, 0, 2$. This is to ensure that the fully transformed arrays are aligned in the non-periodic direction 2. And we need the arrays aligned in this direction, because this is the only direction where there are tensor product matrices that are non-diagonal. All Fourier matrices are, naturally, diagonal.

Test- and trialfunctions are created much like in a regular, non-mixed, formulation. However, one has to create one test- and trialfunction for the mixed space, and then split them up afterwards

```
up = TrialFunction(VQ)
vq = TestFunction(VQ)
u, p = up
v, q = vq
```

With the basisfunctions in place we may assemble the different blocks of the final coefficient matrix. Since Legendre is using a constant weight function, the equations may also be integrated by parts to obtain a symmetric system:

```
if family.lower() == 'chebyshev':
    A = inner(v, div(grad(u)))
    G = inner(v, -grad(p))
else:
    A = inner(grad(v), -grad(u))
    G = inner(div(v), p)
D = inner(q, div(u))
```

Note: The inner products may also be assembled with one single line, as

```
AA = inner(v, div(grad(u))) + inner(v, -grad(u)) + inner(q, div(u))
```

However, this requires addition, not subtraction, of inner products and it is not possible to move the negation to -inner(v, grad(u))

The assembled subsystems `A, G` and `D` are lists containg the different blocks of the complete, coupled matrix. `A` actually contains 6 tensor product matrices of type `TPMatrix`. The first two matrices are for vector component zero of the test function `v[0]` and trial function `u[0]`, the matrices 2 and 3 are for components 1 and the last two are for components 2. The first two matrices are as such for

```
A[0:2] = inner(v[0], div(grad(u[0])))
```

Breaking it down the inner product is mathematically

$$\int_{\Omega} \mathbf{v}[0] \left(\frac{\partial^2 \mathbf{u}[0]}{\partial x^2} + \frac{\partial^2 \mathbf{u}[0]}{\partial y^2} + \frac{\partial^2 \mathbf{u}[0]}{\partial z^2} \right) w_x dx w_y dy w_z dz. \tag{7.129}$$

If we now use test function $\mathbf{v}[0]$

$$\mathbf{v}[0]_{lmn} = \mathcal{X}_l \mathcal{Y}_m \mathcal{Z}_n, \tag{7.130}$$

and trialfunction

$$\mathbf{u}[0]_{pqr} = \sum_p \sum_q \sum_r \hat{\mathbf{u}}[0]_{pqr} \mathcal{X}_p \mathcal{Y}_q \mathcal{Z}_r, \tag{7.131}$$

where $\hat{\mathbf{u}}$ are the unknown degrees of freedom, and then insert these functions into (7.129), then we obtain after performing some exact evaluations over the periodic directions

$$\Big(\underbrace{-\left(l^2 \delta_{lp} + m^2 \delta_{mq}\right) \int_{-1}^{1} \mathcal{Z}_r(z) \mathcal{Z}_n(z) w_z dz}_{A[0]} + \underbrace{\delta_{lp} \delta_{mq} \int_{-1}^{1} \frac{\partial^2 \mathcal{Z}_r(z)}{\partial z^2} \mathcal{Z}_n(z) w_z dz}_{A[1]} \Big) \hat{\mathbf{u}}[0]_{pqr}, \tag{7.132}$$

Similarly for components 1 and 2 of the test and trial vectors, leading to 6 tensor product matrices in total for `A`. Similarly, we get three components of `G` and three of `D`.

Eliminating the Fourier diagonal matrices, we are left with block matrices like

$$H(l,m) = \begin{bmatrix} A[0]+A[1] & 0 & 0 & G[0] \\ 0 & A[2]+A[3] & 0 & G[1] \\ 0 & 0 & A[4]+A[5] & G[2] \\ D[0] & D[1] & D[2] & 0 \end{bmatrix} \tag{7.133}$$

Note that there will be one large block matrix $H(l,m)$ for each Fourier wavenumber combination (l,m). To solve the problem in the end we will need to loop over these wavenumbers and solve the assembled linear systems one by one. An example of the block matrix, for $l = m = 5$ and $\mathbf{N} = (20, 20, 20)$ is given in Fig. *Coupled block matrix for Stokes equations*.

In the end we create a block matrix through

```
M = BlockMatrix(A+G+D)
```

The right hand side can easily be assembled since we have already defined the functions $\mathbf{f}$ and h, see Sec. *Manufactured solution*

```
# Get mesh (quadrature points)
X = TD.local_mesh(True)

# Get f and h on quad points
fh = Array(VQ, buffer=(fx, fy, fz, h))
f_, h_ = fh

# Compute inner products
fh_hat = Function(VQ)
f_hat, h_hat = fh_hat
f_hat = inner(v, f_, output_array=f_hat)
h_hat = inner(q, h_, output_array=h_hat)
```

In the end all that is left is to solve and compare with the exact solution.

```
# Solve problem
up_hat = M.solve(fh_hat, constraints=((3, 0, 0),))
up = up_hat.backward()
u_, p_ = up

# Exact solution
ux, uy, uz = Array(V, buffer=(uex, uey, uez))
pe = Array(Q, buffer=pe)

error = [comm.reduce(np.linalg.norm(ux-u_[0])),
         comm.reduce(np.linalg.norm(uy-u_[1])),
         comm.reduce(np.linalg.norm(uz-u_[2])),
         comm.reduce(np.linalg.norm(pe-p_))]
print(error)
```

Note that solve has a keyword argument `constraints=((3, 0, 0),)` that takes care of the restriction $\int_\Omega p dx = 0$ by indenting the row in M corresponding to the first degree of freedom for the pressure. The value $(3, 0, 0)$ indicates that pressure is in block 3 of the block vector solution (the velocity vector holds positions 0, 1 and 2), whereas the two zeros ensures that the first dof (dof 0) should obtain value 0.

Complete solver

A complete solver can be found in demo Stokes3D.py.

7.7 Demo - Lid driven cavity

Authors Mikael Mortensen (mikaem at math.uio.no)

Date Jun 7, 2020

Summary. The lid driven cavity is a classical benchmark for Navier Stokes solvers. This is a demonstration of how the Python module shenfun can be used to solve the lid driven cavity problem with full spectral accuracy using a mixed (coupled) basis in a 2D tensor product domain. The demo also shows how to use mixed tensor product spaces for vector valued equations. Note that the regular lid driven cavity, where the top wall has constant velocity and the remaining three walls are stationary, has a singularity at the two upper corners, where the velocity is discontinuous. Due to their global nature, spectral methods are usually not very good at handling problems with discontinuities, and for this reason we will also look at a regularized lid driven cavity, where the top lid moves according to $(1-x)^2(1+x)^2$, thus removing the corner discontinuities.

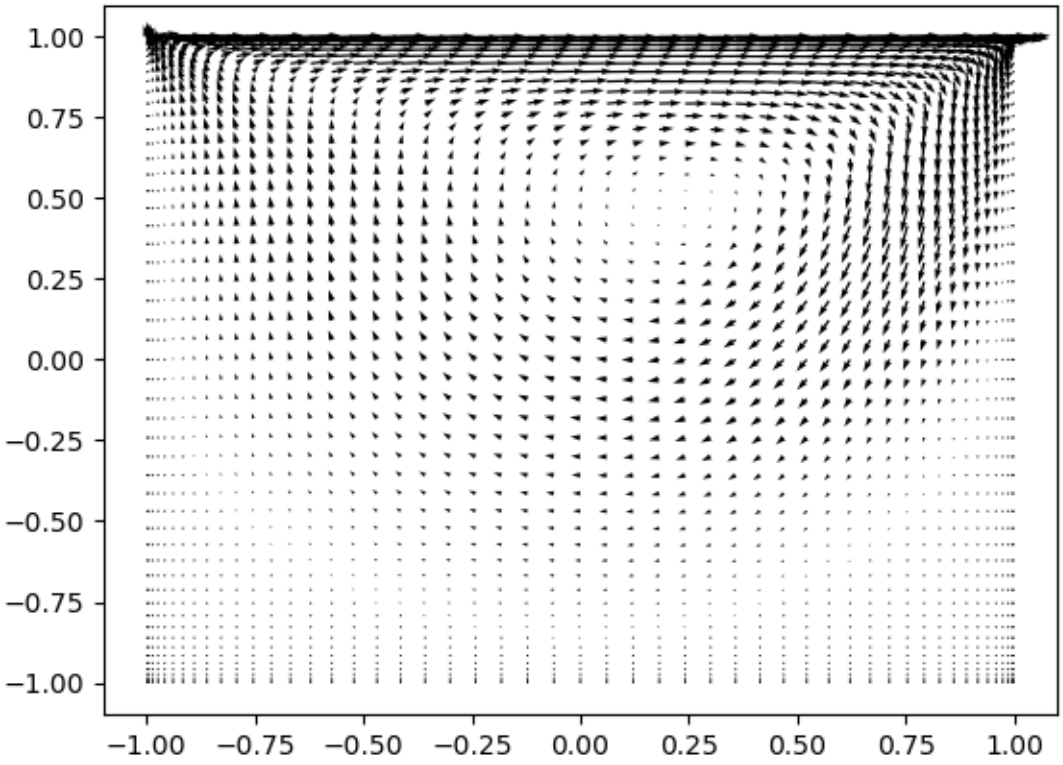


Fig. 6: Velocity vectors for $Re = 100$

7.7.1 Navier Stokes equations

The nonlinear steady Navier Stokes equations are given in strong form as

$$\begin{aligned}
\nu\nabla^2\boldsymbol{u} - \nabla p &= \nabla\cdot\boldsymbol{u}\boldsymbol{u} \quad \text{in } \Omega,\\
\nabla\cdot\boldsymbol{u} &= 0 \quad \text{in } \Omega\\
\int_\Omega p dx &= 0\\
\boldsymbol{u}(x, y=1) = (1,0) \text{ or } \boldsymbol{u}(x, y=1) &= ((1-x)^2(1+x)^2, 0)\\
\boldsymbol{u}(x, y=-1) &= (0,0)\\
\boldsymbol{u}(x=\pm 1, y) &= (0,0)
\end{aligned}$$

where $\boldsymbol{u}$, p and ν are, respectively, the fluid velocity vector, pressure and kinematic viscosity. The domain $\Omega = [-1, 1]^2$ and the nonlinear term $\boldsymbol{u}\boldsymbol{u}$ is the outer product of vector $\boldsymbol{u}$ with itself. Note that the final $\int_\Omega p dx = 0$ is there because there is no Dirichlet boundary condition on the pressure and the system of equations would otherwise be ill conditioned.

We want to solve these steady nonlinear Navier Stokes equations with the Galerkin method, using the shenfun Python package. The first thing we need to do then is to import all of shenfun's functionality

```
import matplotlib.pyplot as plt
from shenfun import *
from mpi4py import MPI
comm = MPI.COMM_WORLD
```

Note that MPI for Python (mpi4py) is a requirement for shenfun, but the current solver cannot be used with more than one processor.

7.7.2 Bases and tensor product spaces

With the Galerkin method we need basis functions for both velocity and pressure, as well as for the nonlinear right hand side. A Dirichlet basis will be used for velocity, whereas there is no boundary restriction on the pressure basis. For both two-dimensional bases we will use one basis function for the x-direction, $\mathcal{X}_k(x)$, and one for the y-direction, $\mathcal{Y}_l(y)$. And then we create two-dimensional basis functions like

$$v_{kl}(x, y) = \mathcal{X}_k(x)\mathcal{Y}_l(y), \tag{7.134}$$

and solutions (trial functions) as

$$u(x, y) = \sum_k\sum_l \hat{u}_{kl} v_{kl}(x, y). \tag{7.135}$$

For the homogeneous Dirichlet boundary condition the basis functions $\mathcal{X}_k(x)$ and $\mathcal{Y}_l(y)$ are chosen as composite Legendre polynomials (we could also use Chebyshev):

$$\mathcal{X}_k(x) = L_k(x) - L_{k+2}(x), \quad \forall\, k \in \boldsymbol{k}^{N_0-2}, \tag{7.136}$$

$$\mathcal{Y}_l(y) = L_l(y) - L_{l+2}(y), \quad \forall\, l \in \boldsymbol{l}^{N_1-2}, \tag{7.137}$$

where $\boldsymbol{k}^{N_0-2} = (0, 1, \ldots, N_0-3)$, $\boldsymbol{l}^{N_1-2} = (0, 1, \ldots, N_1-3)$ and $N = (N_0, N_1)$ is the number of quadrature points in each direction. Note that N_0 and N_1 do not need to be the same. The basis funciton (7.136) satisfies the homogeneous Dirichlet boundary conditions at $x = \pm 1$ and (7.137) the same at $y = \pm 1$. As such, the basis function $v_{kl}(x, y)$ satisfies the homogeneous Dirichlet boundary condition for the entire domain.

With shenfun we create these homogeneous spaces, $D_0^{N_0}(x) = \text{span}\{L_k - L_{k+2}\}_{k=0}^{N_0-2}$ and $D_0^{N_1}(y) = \text{span}\{L_l - L_{l+2}\}_{l=0}^{N_1-2}$ as

```
N = (51, 51)
family = 'Legendre' # or use 'Chebyshev'
quad = 'LG'         # for Chebyshev use 'GC' or 'GL'
D0X = Basis(N[0], family, quad=quad, bc=(0, 0))
D0Y = Basis(N[1], family, quad=quad, bc=(0, 0))
```

The spaces are here the same, but we will use `D0X` in the x-direction and `D0Y` in the y-direction. But before we use these bases in tensor product spaces, they remain identical as long as $N_0 = N_1$.

Special attention is required by the moving lid. To get a solution with nonzero boundary condition at $y = 1$ we need to add one more basis function that satisfies that solution. In general, a nonzero boundary condition can be added on both sides of the domain using the following basis

$$\mathcal{Y}_l(y) = L_l(y) - L_{l+2}(y), \quad \forall\, l \in \boldsymbol{l}^{N_1-2}. \tag{7.138}$$

$$\mathcal{Y}_{N_1-2}(y) = (L_0 + L_1)/2 \quad (= (1+y)/2)\,, \tag{7.139}$$

$$\mathcal{Y}_{N_1-1}(y) = (L_0 - L_1)/2 \quad (= (1-y)/2)\,. \tag{7.140}$$

And then the unknown component $N_1 - 2$ decides the value at $y = 1$, whereas the unknown at $N_1 - 1$ decides the value at $y = -1$. Here we only need to add the $N_1 - 2$ component, but for generality this is implemented in shenfun using both additional basis functions. We create the space $D_1^{N_1}(y) = \text{span}\{\mathcal{Y}_l(y)\}_{l=0}^{N_1-1}$ as

```
D1Y = Basis(N[1], family, quad=quad, bc=(1, 0))
```

where `bc=(1, 0)` fixes the values for $y = 1$ and $y = -1$, respectively. For a regularized lid driven cavity the velocity of the top lid is $(1-x)^2(1+x)^2$ and not unity. To implement this boundary condition instead, we can make use of sympy and quite straight forward do

```
import sympy
x = sympy.symbols('x')
#D1Y = Basis(N[1], family, quad=quad, bc=((1-x)**2*(1+x)**2, 0))
```

Uncomment the last line to run the regularized boundary conditions. Otherwise, there is no difference at all between the regular and the regularized lid driven cavity implementations.

The pressure basis that comes with no restrictions for the boundary is a little trickier. The reason for this has to do with inf-sup stability. The obvious choice of basis functions are the regular Legendre polynomials $L_k(x)$ in x and $L_l(y)$ in the y-directions. The problem is that for the natural choice of $(k, l) \in \boldsymbol{k}^{N_0} \times \boldsymbol{l}^{N_1}$ there are nullspaces and the problem is not well-defined. It turns out that the proper choice for the pressure basis is simply the regular Legendre basis functions, but for $(k, l) \in \boldsymbol{k}^{N_0-2} \times \boldsymbol{l}^{N_1-2}$. The bases $P^{N_0}(x) = \text{span}\{L_k(x)\}_{k=0}^{N_0-3}$ and $P^{N_1}(y) = \text{span}\{L_l(y)\}_{l=0}^{N_1-3}$ are created as

```
PX = Basis(N[0], family, quad=quad)
PY = Basis(N[1], family, quad=quad)
PX.slice = lambda: slice(0, N[0]-2)
PY.slice = lambda: slice(0, N[1]-2)
```

Note that we still use these spaces with the same $N_0 \cdot N_1$ quadrature points in real space, but the two highest frequencies have been set to zero.

We have now created all relevant function spaces for the problem at hand. It remains to combine these spaces into tensor product spaces, and to combine tensor product spaces into mixed (coupled) tensor product spaces. From the Dirichlet bases we create two different tensor product spaces, whereas one is enough for the pressure

$$V_1^{\boldsymbol{N}}(\boldsymbol{x}) = D_0^{N_0}(x) \otimes D_1^{N_1}(y), \tag{7.141}$$

$$V_0^{\boldsymbol{N}}(\boldsymbol{x}) = D_0^{N_0}(x) \otimes D_0^{N_1}(y), \tag{7.142}$$

$$P^{\boldsymbol{N}}(\boldsymbol{x}) = P^{N_0}(x) \otimes P^{N_1}(y). \tag{7.143}$$

With shenfun the tensor product spaces are created as

```
V1 = TensorProductSpace(comm, (D0X, D1Y))
V0 = TensorProductSpace(comm, (D0X, D0Y))
P = TensorProductSpace(comm, (PX, PY))
```

These tensor product spaces are all scalar valued. The velocity is a vector, and a vector requires a mixed vector basis like $W_1^{\boldsymbol{N}} = V_1^{\boldsymbol{N}} \times V_0^{\boldsymbol{N}}$. The vector basis is created in shenfun as

```
W1 = VectorTensorProductSpace([V1, V0])
W0 = VectorTensorProductSpace([V0, V0])
```

Note that the second vector basis, $W_0^{\boldsymbol{N}} = V_0^{\boldsymbol{N}} \times V_0^{\boldsymbol{N}}$, uses homogeneous boundary conditions throughout.

7.7.3 Mixed variational form

We now formulate a variational problem using the Galerkin method: Find $\boldsymbol{u} \in W_1^{\boldsymbol{N}}$ and $p \in P^{\boldsymbol{N}}$ such that

$$\int_\Omega (\nu \nabla^2 \boldsymbol{u} - \nabla p) \cdot \boldsymbol{v}\, dxdy = \int_\Omega (\nabla \cdot \boldsymbol{u}\boldsymbol{u}) \cdot \boldsymbol{v}\, dxdy \quad \forall \boldsymbol{v} \in W_0^{\boldsymbol{N}}, \tag{7.144}$$

$$\int_\Omega \nabla \cdot \boldsymbol{u}\, q\, dxdy = 0 \quad \forall q \in P^{\boldsymbol{N}}. \tag{7.145}$$

Note that we are using test functions $\boldsymbol{v}$ with homogeneous boundary conditions.

The first obvious issue with Eq (7.144) is the nonlinearity. In other words we will need to linearize and iterate to be able to solve these equations with the Galerkin method. To this end we will introduce the solution on iteration $k \in [0, 1, \ldots]$ as $\boldsymbol{u}^k$ and compute the nonlinearity using only known solutions $\int_\Omega (\nabla \cdot \boldsymbol{u}^k \boldsymbol{u}^k) \cdot \boldsymbol{v}\, dxdy$. Using further integration by parts we end up with the equations to solve for iteration number $k+1$ (using $\boldsymbol{u} = \boldsymbol{u}^{k+1}$ and $p = p^{k+1}$ for simplicity)

$$-\int_\Omega \nu \nabla \boldsymbol{u} : \nabla \boldsymbol{v}\, dxdy + \int_\Omega p \nabla \cdot \boldsymbol{v}\, dxdy = \int_\Omega (\nabla \cdot \boldsymbol{u}^k \boldsymbol{u}^k) \cdot \boldsymbol{v}\, dxdy \quad \forall \boldsymbol{v} \in W_0^{\boldsymbol{N}}, \tag{7.146}$$

$$\int_\Omega \nabla \cdot \boldsymbol{u}\, q\, dxdy = 0 \quad \forall q \in P^{\boldsymbol{N}}. \tag{7.147}$$

Note that the nonlinear term may also be integrated by parts and evaluated as $\int_\Omega -\boldsymbol{u}^k \boldsymbol{u}^k : \nabla \boldsymbol{v}\, dxdy$. All boundary integrals disappear since we are using test functions with homogeneous boundary conditions.

Since we are to solve for $\boldsymbol{u}$ and p at the same time, we formulate a mixed (coupled) problem: find $(\boldsymbol{u}, p) \in W_1^{\boldsymbol{N}} \times P^{\boldsymbol{N}}$ such that

$$a((\boldsymbol{u}, p), (\boldsymbol{v}, q)) = L((\boldsymbol{v}, q)) \quad \forall (\boldsymbol{v}, q) \in W_0^{\boldsymbol{N}} \times P^{\boldsymbol{N}}, \tag{7.148}$$

where bilinear (a) and linear (L) forms are given as

$$a((\boldsymbol{u}, p), (\boldsymbol{v}, q)) = -\int_\Omega \nu \nabla \boldsymbol{u} : \nabla \boldsymbol{v}\, dxdy + \int_\Omega p \nabla \cdot \boldsymbol{v}\, dxdy + \int_\Omega \nabla \cdot \boldsymbol{u}\, q\, dxdy, \tag{7.149}$$

$$L((\boldsymbol{v}, q); \boldsymbol{u}^k) = \int_\Omega (\nabla \cdot \boldsymbol{u}^k \boldsymbol{u}^k) \cdot \boldsymbol{v}\, dxdy. \tag{7.150}$$

Note that the bilinear form will assemble to a block matrix, whereas the right hand side linear form will assemble to a block vector. The bilinear form does not change with the solution and as such it does not need to be reassembled inside an iteration loop.

The algorithm used to solve the equations are:

- Set $k = 0$
- Guess $\boldsymbol{u}^0 = (0, 0)$
- while not converged:
  - assemble $L((\boldsymbol{v}, q); \boldsymbol{u}^k)$
  - solve $a((\boldsymbol{u}, p), (\boldsymbol{v}, q)) = L((\boldsymbol{v}, q); \boldsymbol{u}^k)$ for $\boldsymbol{u}^{k+1}, p^{k+1}$
  - compute error = $\int_\Omega (\boldsymbol{u}^{k+1} - \boldsymbol{u}^k)^2 \, dxdy$
  - if error $<$ some tolerance then converged = True
  - k += 1

7.7.4 Implementation of solver

We will now implement the coupled variational problem described in previous sections. First of all, since we want to solve for the velocity and pressure in a coupled solver, we have to create a mixed tensor product space $VQ = W_{\mathrm{I}}^{\boldsymbol{N}} \times P^{\boldsymbol{N}}$ that couples velocity and pressure

```
VQ = MixedTensorProductSpace([W1, P])     # Coupling velocity and pressure
```

We can now create test- and trialfunctions for the coupled space VQ, and then split them up into components afterwards:

```
up = TrialFunction(VQ)
vq = TestFunction(VQ)
u, p = up
v, q = vq
```

Note: The test function `v` is using homogeneous Dirichlet boundary conditions even though it is derived from `VQ`, which contains `W1`. It is currently not (and will probably never be) possible to use test functions with inhomogeneous boundary conditions.

With the basisfunctions in place we may assemble the different blocks of the final coefficient matrix. For this we also need to specify the kinematic viscosity, which is given here in terms of the Reynolds number:

```
Re = 100.
nu = 2./Re
A = inner(grad(v), -nu*grad(u))
G = inner(div(v), p)
D = inner(q, div(u))
```

Note: The inner products may also be assembled with one single line, as

```
AA = inner(grad(v), -nu*grad(u)) + inner(div(v), p) + inner(q, div(u))
```

But note that this requires addition, not subtraction, of inner products, and it is not possible to move the negation to `-inner(grad(v), nu*grad(u))`. This is because the `inner()` function returns a list of tensor product matrices of type `TPMatrix`, and you cannot negate a list.

The assembled subsystems `A, G` and `D` are lists containg the different blocks of the complete, coupled, coefficient matrix. `A` actually contains 4 tensor product matrices of type `TPMatrix`. The first two matrices are for vector

component zero of the test function `v[0]` and trial function `u[0]`, the matrices 2 and 3 are for components 1. The first two matrices are as such for

```
A[0:2] = inner(grad(v[0]), -nu*grad(u[0]))
```

Breaking it down the inner product is mathematically

$$\int_\Omega -\nu \left(\frac{\partial \boldsymbol{v}[0]}{\partial x}, \frac{\partial \boldsymbol{v}[0]}{\partial y} \right) \cdot \left(\frac{\partial \boldsymbol{u}[0]}{\partial x}, \frac{\partial \boldsymbol{u}[0]}{\partial y} \right) dxdy. \tag{7.151}$$

We can now insert for test function $\boldsymbol{v}[0]$

$$\boldsymbol{v}[0]_{kl} = \mathcal{X}_k \mathcal{Y}_l, \quad (k,l) \in \boldsymbol{k}^{N_0-2} \times \boldsymbol{l}^{N_1-2} \tag{7.152}$$

and trialfunction

$$\boldsymbol{u}[0]_{mn} = \sum_{m=0}^{N_0-3} \sum_{n=0}^{N_1-1} \hat{\boldsymbol{u}}[0]_{mn} \mathcal{X}_m \mathcal{Y}_n, \tag{7.153}$$

where $\hat{\boldsymbol{u}}$ are the unknown degrees of freedom for the velocity vector. Notice that the sum over the second index runs all the way to $N_1 - 1$, whereas the other indices runs to either $N_0 - 3$ or $N_1 - 3$. This is because of the additional basis functions required for the inhomogeneous boundary condition.

Inserting for these basis functions into (7.129), we obtain after a few trivial manipulations

$$-\sum_{m=0}^{N_0-3} \sum_{n=0}^{N_1-1} \nu \Big(\underbrace{\int_{-1}^{1} \frac{\partial \mathcal{X}_k(x)}{\partial x} \frac{\partial \mathcal{X}_m}{\partial x} dx \int_{-1}^{1} \mathcal{Y}_l \mathcal{Y}_n dy}_{A[0]} + \underbrace{\int_{-1}^{1} \mathcal{X}_k(x) X_m(x) dx \int_{-1}^{1} \frac{\partial \mathcal{Y}_l}{\partial y} \frac{\partial \mathcal{Y}_n}{\partial y} dy}_{A[1]} \Big) \hat{\boldsymbol{u}}[0]_{mn}. \tag{7.154}$$

We see that each tensor product matrix (both A[0] and A[1]) is composed as outer products of two smaller matrices, one for each dimension. The first tensor product matrix, A[0], is

$$\underbrace{\int_{-1}^{1} \frac{\partial \mathcal{X}_k(x)}{\partial x} \frac{\partial \mathcal{X}_m}{\partial x} dx}_{c_{km}} \underbrace{\int_{-1}^{1} \mathcal{Y}_l \mathcal{Y}_n dy}_{f_{ln}} \tag{7.155}$$

where $C \in \mathbb{R}^{N_0-2 \times N_1-2}$ and $F \in \mathbb{R}^{N_0-2 \times N_1}$. Note that due to the inhomogeneous boundary conditions this last matrix F is actually not square. However, remember that all contributions from the two highest degrees of freedom ($\hat{\boldsymbol{u}}[0]_{m,N_1-2}$ and $\hat{\boldsymbol{u}}[0]_{m,N_1-1}$) are already known and they can, as such, be moved directly over to the right hand side of the linear algebra system that is to be solved. More precisely, we can split the tensor product matrix into two contributions and obtain

$$\sum_{m=0}^{N_0-3} \sum_{n=0}^{N_1-1} c_{km} f_{ln} \hat{\boldsymbol{u}}[0]_{m,n} = \sum_{m=0}^{N_0-3} \sum_{n=0}^{N_1-3} c_{km} f_{ln} \hat{\boldsymbol{u}}[0]_{m,n} + \sum_{m=0}^{N_0-3} \sum_{n=N_1-2}^{N_1-1} c_{km} f_{ln} \hat{\boldsymbol{u}}[0]_{m,n}, \quad \forall (k,l) \in \boldsymbol{k}^{N_0-2} \times \boldsymbol{l}^{N_1-2},$$

where the first term on the right hand side is square and the second term is known and can be moved to the right hand side of the linear algebra equation system.

All the parts of the matrices that are to be moved to the right hand side can be extracted from A, G and D as follows

```
# Extract the boundary matrices
bc_mats = extract_bc_matrices([A, G, D])
```

These matrices are applied to the solution below (see `BlockMatrix BM`). Furthermore, this leaves us with square submatrices (A, G, D), which make up a symmetric block matrix

$$M = \begin{bmatrix} A[0] + A[1] & 0 & G[0] \\ 0 & A[2] + A[3] & G[1] \\ D[0] & D[1] & 0 \end{bmatrix} \tag{7.156}$$

This matrix, and the matrix responsible for the boundary degrees of freedom, can be assembled from the pieces we already have as

```
M = BlockMatrix(A+G+D)
BM = BlockMatrix(bc_mats)
```

We now have all the matrices we need in order to solve the Navier Stokes equations. However, we also need some work arrays for iterations and we need to assemble the constant boundary contribution to the right hand side

```
# Create Function to hold solution. Use set_boundary_dofs to fix the degrees
# of freedom in uh_hat that determines the boundary conditions.
uh_hat = Function(VQ).set_boundary_dofs()
ui_hat = uh_hat[0]

# New solution (iterative)
uh_new = Function(VQ).set_boundary_dofs()
ui_new = uh_new[0]

# Compute the constant contribution to rhs due to nonhomogeneous boundary conditions
bh_hat0 = Function(VQ)
bh_hat0 = BM.matvec(-uh_hat, bh_hat0) # Negative because moved to right hand side
bi_hat0 = bh_hat0[0]
```

Note that `bh_hat0` now contains the part of the right hand side that is due to the non-symmetric part of assembled matrices. The appended `set_boundary_dofs()` ensures the known boundary values of the solution are fixed for `ui_hat` and `ui_new`.

The nonlinear right hand side also requires some additional attention. Nonlinear terms are usually computed in physical space before transforming to spectral. For this we need to evaluate the velocity vector on the quadrature mesh. We also need a rank 2 Array to hold the outer product $\boldsymbol{uu}$. The required arrays and spaces are created as

```
bh_hat = Function(VQ)

# Create arrays to hold velocity vector solution
ui = Array(W1)

# Create work arrays for nonlinear part
QT = MixedTensorProductSpace([W1, W0])  # for uiuj
uiuj = Array(QT)
uiuj_hat = Function(QT)
```

The right hand side $L((\boldsymbol{v}, q); \boldsymbol{u}^k)$; is computed in its own function `compute_rhs` as

```
def compute_rhs(ui_hat, bh_hat):
    global ui, uiuj, uiuj_hat, V1, bh_hat0
    bh_hat.fill(0)
    ui = W1.backward(ui_hat, ui)
    uiuj = outer(ui, ui, uiuj)
    uiuj_hat = uiuj.forward(uiuj_hat)
    bi_hat = bh_hat[0]
    #bi_hat = inner(v, div(uiuj_hat), output_array=bi_hat)
    bi_hat = inner(grad(v), -uiuj_hat, output_array=bi_hat)
    bh_hat += bh_hat0
    return bh_hat
```

Here `outer()` is a shenfun function that computes the outer product of two vectors and returns the product in a rank two array (here `uiuj`). With `uiuj` forward transformed to `uiuj_hat` we can assemble the linear form either as `inner(v, div(uiuj_hat)` or `inner(grad(v), -uiuj_hat)`. Also notice that the constant contribution

from the inhomogeneous boundary condition, `bh_hat0`, is added to the right hand side vector.

Now all that remains is to guess an initial solution and solve iteratively until convergence. For initial solution we simply set the velocity and pressure to zero and solve the Stokes equations:

```
from scipy.sparse.linalg import splu
uh_hat, Ai = M.solve(bh_hat0, u=uh_hat, constraints=((2, 0, 0),), return_system=True)
↪# Constraint for component 2 of mixed space
Alu = splu(Ai)
uh_new[:] = uh_hat
```

Note that the `BlockMatrix` given by `M` has a solve method that sets up a sparse coefficient matrix `Ai` of size $\mathbb{R}^{3(N_0-2)(N_1-2)\times 3(N_0-2)(N_1-2)}$, and then solves using scipy.sparse.linalg.spsolve. The matrix `Ai` is then pre-factored for reuse with splu. Also note that the `constraints=((2, 0, 0),)` keyword argument ensures that the pressure integrates to zero, i.e., $\int_\Omega p dx dy = 0$. Here the number 2 tells us that block component 2 in the mixed space (the pressure) should be integrated, dof 0 should be fixed, and it should be fixed to 0.

With an initial solution from the Stokes equations we are ready to start iterating. However, for convergence it is necessary to add some underrelaxation α, and update the solution each time step as

$$\hat{\boldsymbol{u}}^{k+1} = \alpha\hat{\boldsymbol{u}}^* + (1-\alpha)\hat{\boldsymbol{u}}^k,$$
$$\hat{p}^{k+1} = \alpha\hat{p}^* + (1-\alpha)\hat{p}^k,$$

where $\hat{\boldsymbol{u}}^*$ and $\hat{p}^*$ are the newly computed velocity and pressure returned from `M.solve`. Without underrelaxation the solution will quickly blow up. The iteration loop goes as follows

```
converged = False
count = 0
alfa = 0.5
while not converged:
    count += 1
    bh_hat = compute_rhs(ui_hat, bh_hat)
    uh_new = M.solve(bh_hat, u=uh_new, constraints=((2, 0, 0),), Alu=Alu) #␣
↪Constraint for component 2 of mixed space
    error = np.linalg.norm(ui_hat-ui_new)
    uh_hat[:] = alfa*uh_new + (1-alfa)*uh_hat
    converged = abs(error) < 1e-10 or count >= 10000
    print('Iteration %d Error %2.4e' %(count, error))

up = uh_hat.backward()
u, p = up

X = V0.local_mesh(True)
plt.figure()
plt.quiver(X[0], X[1], u[0], u[1])
```

The last three lines plots the velocity vectors that are shown in Figure *Velocity vectors for Re=100*. The solution is apparently nice and smooth, but hidden underneath are Gibbs oscillations from the corner discontinuities. This is painfully obvious when switching from Legendre to Chebyshev polynomials. With Chebyshev the same plot looks like Figure *Velocity vectors for Re=100 using Chebyshev*. However, choosing instead the regularized lid, the solutions will be nice and smooth, both for Legendre and Chebyshev polynomials.

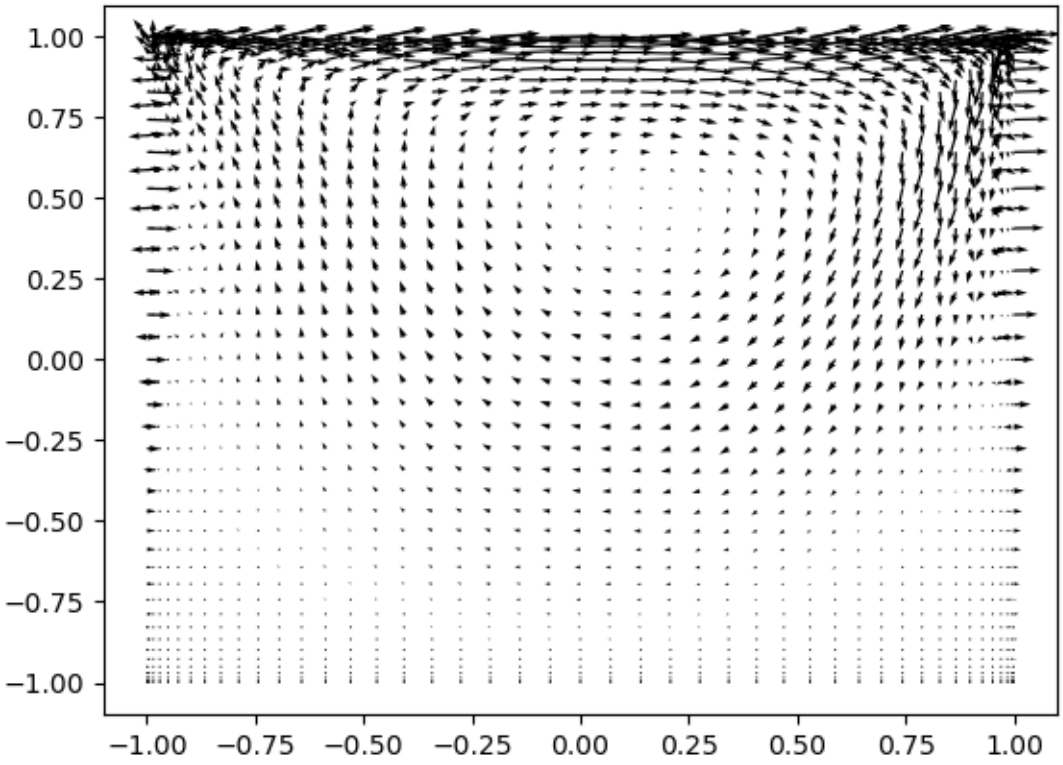


Fig. 7: *Velocity vectors for Re=100 using Chebyshev*

7.7.5 Complete solver

A complete solver can be found in demo NavierStokesDrivenCavity.py.

7.8 Demo - Rayleigh Benard

Authors Mikael Mortensen (mikaem at math.uio.no)

Date Jun 7, 2020

Summary. Rayleigh-Benard convection arise due to temperature gradients in a fluid. The governing equations are Navier-Stokes coupled (through buoyancy) with an additional temperature equation derived from the first law of thermodynamics, using a linear correlation between density and temperature.

This is a demonstration of how the Python module shenfun can be used to solve for these Rayleigh-Benard cells in a 2D channel with two walls of different temperature in one direction, and periodicity in the other direction. The solver described runs with MPI without any further considerations required from the user. Note that there is a more physically realistic 3D solver implemented within the spectralDNS project. To allow for some simple optimizations, the solver described in this demo has been implemented in a class in the RayleighBenardRk3.py module in the demo folder of shenfun. Below are two example solutions, where the first (movie) has been run at a very high Rayleigh number (*Ra*), and the lower image with a low *Ra* (laminar).

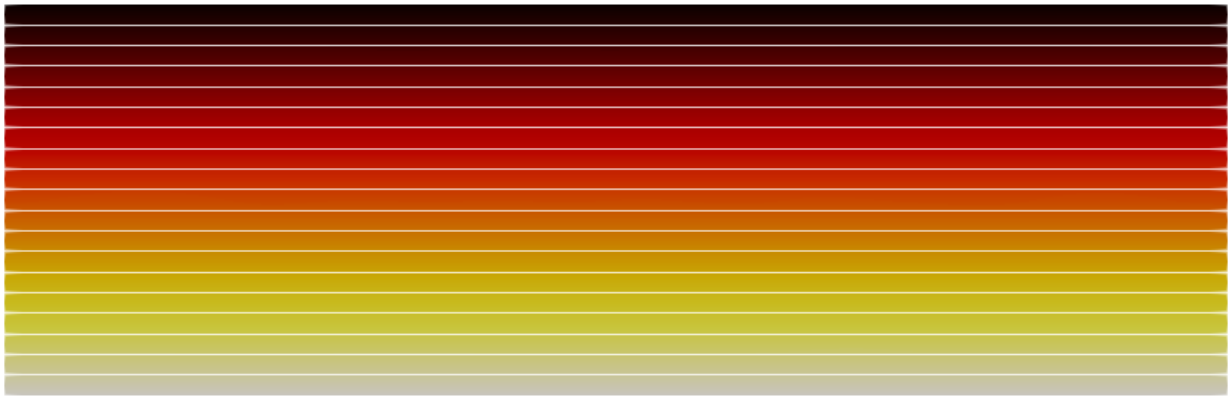


Fig. 8: Temperature fluctuations in the Rayleigh Benard flow. The top and bottom walls are kept at different temperatures and this sets up the Rayleigh-Benard convection. The simulation is run at *Ra* =100,000, *Pr* =0.7 with 100 and 256 quadrature points in *x* and *y*-directions, respectively

7.8.1 Model problem

The governing equations solved in domain $\Omega = [-1, 1] \times [0, 2\pi]$ are

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \sqrt{\frac{Pr}{Ra}}\nabla^2 \mathbf{u} + T\mathbf{i}, \tag{7.157}$$

$$\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T = \frac{1}{\sqrt{RaPr}}\nabla^2 T, \tag{7.158}$$

$$\nabla \cdot \mathbf{u} = 0, \tag{7.159}$$

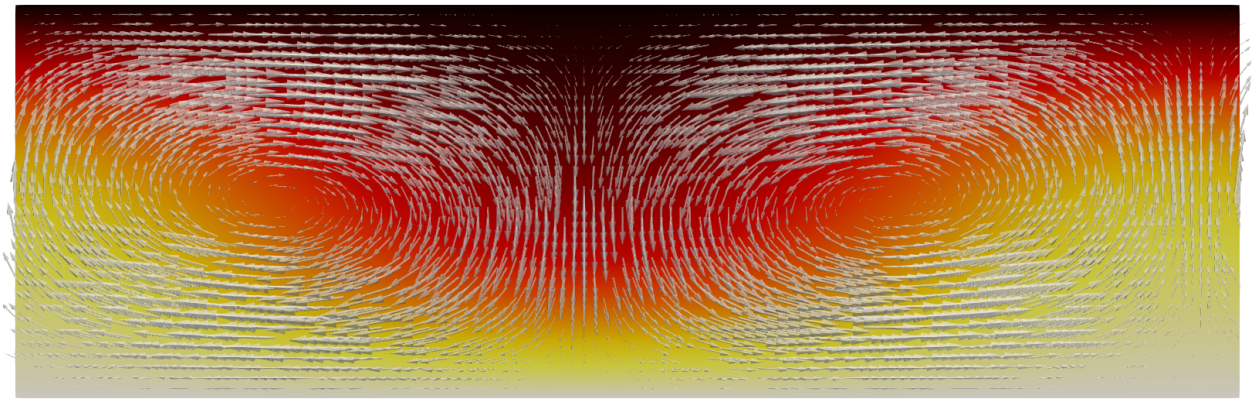


Fig. 9: Convection cells for a laminar flow. The simulation is run at *Ra* =100, *Pr* =0.7 with 40 and 128 quadrature points in *x* and *y*-directions, respectively

where $\mathbf{u}(x,y,t)(=u\mathbf{i}+v\mathbf{j})$ is the velocity vector, $p(x,y,t)$ is pressure, $T(x,y,t)$ is the temperature, and $\mathbf{i}$ and $\mathbf{j}$ are the unity vectors for the x and y-directions, respectively.

The equations are complemented with boundary conditions $\mathbf{u}(\pm 1,y,t)=(0,0), \mathbf{u}(x,2\pi,t)=\mathbf{u}(x,0,t), T(1,y,t)=1, T(-1,y,t)=0, T(x,2\pi,t)=T(x,0,t)$. Note that these equations have been non-dimensionalized according to [PSS18], using dimensionless Rayleigh number $Ra=g\alpha\Delta T h^3/(\nu\kappa)$ and Prandtl number $Pr=\nu/\kappa$. Here $g\mathbf{i}$ is the vector accelleration of gravity, ΔT is the temperature difference between the top and bottom walls, h is the hight of the channel in x-direction, ν is the dynamic viscosity coefficient, κ is the heat transfer coefficient and α is the thermal expansion coefficient. Note that the governing equations have been non-dimensionalized using the free-fall velocityscale $U=\sqrt{g\alpha\Delta T h}$. See [PSS18] for more details.

The governing equations contain a non-trivial coupling between velocity, pressure and temperature. This coupling can be simplified by eliminating the pressure from the equation for the wall-normal velocity component u. We accomplish this by taking the Laplace of the momentum equation in wall normal direction, using the pressure from the divergence of the momentum equation $\nabla^2 p=-\nabla\cdot\mathbf{H}+\partial T/\partial x$, where $\mathbf{H}=(H_x,H_y)=(\mathbf{u}\cdot\nabla)\mathbf{u}$

$$\frac{\partial\nabla^2 u}{\partial t}=\frac{\partial^2 H_y}{\partial x\partial y}-\frac{\partial^2 H_x}{\partial y\partial y}+\sqrt{\frac{Pr}{Ra}}\nabla^4 u+\frac{\partial^2 T}{\partial y^2}. \tag{7.160}$$

This equation is solved with $u(\pm 1)=\partial u/\partial x(\pm 1)=0$, where the latter follows from the divergence constraint. In summary, we now seem to have the following equations to solve:

$$\frac{\partial\nabla^2 u}{\partial t}=\frac{\partial^2 H_y}{\partial x\partial y}-\frac{\partial^2 H_x}{\partial y\partial y}+\sqrt{\frac{Pr}{Ra}}\nabla^4 u+\frac{\partial^2 T}{\partial y^2}, \tag{7.161}$$

$$\frac{\partial v}{\partial t}+H_y=-\frac{\partial p}{\partial y}+\sqrt{\frac{Pr}{Ra}}\nabla^2 v, \tag{7.162}$$

$$\frac{\partial T}{\partial t}+\mathbf{u}\cdot\nabla T=\frac{1}{\sqrt{RaPr}}\nabla^2 T, \tag{7.163}$$

$$\nabla\cdot\mathbf{u}=0. \tag{7.164}$$

However, we note that Eqs. (7.161) and (7.163) and (7.164) do not depend on pressure, and, apparently, on each time step we can solve (7.161) for u, then (7.164) for v and finally (7.163) for T. So what do we need (7.162) for? It appears to have become redundant from the elimination of the pressure from Eq. (7.161). It turns out that this is actually almost completely true, but (7.161), (7.163) and (7.164) can only provide closure for all but one of the Fourier

coefficients. To see this it is necessary to introduce some discretization and basis functions that will be used to solve the problem. To this end we use P_N, which is the set of all real polynomials of degree less than or equal to N and introduce the following finite-dimensional approximation spaces

$$V_N^B(x) = \{v \in P_N | v(\pm 1) = v(\pm 1) = 0\}, \tag{7.165}$$

$$V_N^D(x) = \{v \in P_N | v(\pm 1) = 0\}, \tag{7.166}$$

$$V_N^T(x) = \{v \in P_N | v(-1) = 0, v(1) = 1\}, \tag{7.167}$$

$$V_N^W(x) = \{v \in P_N\}, \tag{7.168}$$

$$V_M^F(y) = \{\exp(\imath l y) | l \in [-M/2, -M/2+1, \ldots M/2-1]\}. \tag{7.169}$$

Here $\dim(V_N^B) = N-4, \dim(V_N^D) = \dim(V_N^W) = N-2$, $\dim(V_N^T) = N$ and $\dim(V_M^F) = M$. We note that V_N^B, V_N^D, V_N^W and V_N^T can be used to approximate u, v, T and p, respectively, in the x-direction. Also note that for V_M^F it is assumed that M is an even number.

We can now choose basis functions for the spaces, using Shen's composite bases for either Legendre or Chebyshev polynomials. For the Fourier space the basis functions are already given. We leave the actual choice of basis as an implementation option for later. For now we use $\phi^B(x), \phi^D(x), \phi^W$ and $\phi^T(x)$ as common notation for basis functions in spaces V_N^B, V_N^D, V_N^W and V_N^T, respectively.

To get the required approximation spaces for the entire domain we use tensor products of the one-dimensional spaces in (7.165)-(7.169)

$$W_{BF} = V_N^B \otimes V_M^F, \tag{7.170}$$

$$W_{DF} = V_N^D \otimes V_M^F, \tag{7.171}$$

$$W_{TF} = V_N^T \otimes V_M^F, \tag{7.172}$$

$$W_{WF} = V_N^W \otimes V_M^F. \tag{7.173}$$

Space W_{BF} has 2D tensor product basis functions $\phi_k^B(x)\exp(\imath l y)$ and similar for the others. All in all we get the following approximations for the unknowns

$$u_N(x, y, t) = \sum_{k \in \boldsymbol{k}_B} \sum_{l \in \boldsymbol{l}} \hat{u}_{kl}(t) \phi_k^B(x) \exp(\imath l y), \tag{7.174}$$

$$v_N(x, y, t) = \sum_{k \in \boldsymbol{k}_D} \sum_{l \in \boldsymbol{l}} \hat{v}_{kl}(t) \phi_k^D(x) \exp(\imath l y), \tag{7.175}$$

$$p_N(x, y, t) = \sum_{k \in \boldsymbol{k}_W} \sum_{l \in \boldsymbol{l}} \hat{p}_{kl}(t) \phi_k^W(x) \exp(\imath l y), \tag{7.176}$$

$$T_N(x, y, t) = \sum_{k \in \boldsymbol{k}_T} \sum_{l \in \boldsymbol{l}} \hat{T}_{kl}(t) \phi_k^T(x) \exp(\imath l y), \tag{7.177}$$

where $\boldsymbol{k}_x = \{0, 1, \ldots \dim(V_N^x) - 1\}$, for $x \in (B, D, W, T)$ and $\boldsymbol{l} = \{-M/2, -M/2+1, \ldots, M/2-1\}$. Note that since the problem is defined in real space we will have Hermitian symmetry. This means that $\hat{u}_{k,l} = \overline{\hat{u}}_{k,-l}$, with an overbar being a complex conjugate, and similar for $\hat{v}_{kl}, \hat{p}_{kl}$ and $\hat{T}_{kl}$. For this reason we can get away with solving for only the positive l's, as long as we remember that the sum in the end goes over both positive and negative $l's$. This is actually automatically taken care of by the FFT provider and is not much of an additional complexity in the implementation. So from now on $\boldsymbol{l} = \{0, 1, \ldots, M/2\}$.

We can now take a look at why Eq. (7.162) is needed. If we first solve (7.161) for $\hat{u}_{kl}(t), (k, l) \in \boldsymbol{k}_B \times \boldsymbol{l}$, then we can use (7.164) to solve for $\hat{v}_{kl}(t)$. But here there is a problem. We can see this by creating the variational form required

to solve (7.164) by the spectral Galerkin method. To this end make $v = v_N$ in (7.164) a trial function, use $u = u_N$ a known function and take the weighted inner product over the domain using test function $q \in W_{DF}$

$$\left\langle \frac{\partial u_N}{\partial x} + \frac{\partial v_N}{\partial y}, q \right\rangle_w = 0. \tag{7.178}$$

Here we are using the inner product notation

$$\langle a, b \rangle_w = \int_{-1}^{1} \int_{0}^{2\pi} a\bar{b} dx_w dy_w \left(\approx \sum_i \sum_j a(x_i, y_j)\bar{b}(x_i, y_j) w(x_i) w(y_j) \right), \tag{7.179}$$

where the exact form of the weighted scalar product depends on the chosen basis; Legendre has $dx_w = dx$, Chebyshev $dx_w = dx/\sqrt{1-x^2}$ and Fourier $dy_w = dy/2\pi$. The bases also have associated quadrature weights $\{w(x_i)\}_{i=0}^{N-1}$ and $\{w(y_j)\}_{j=0}^{M-1}$ that are used to approximate the integrals.

Inserting now for the known u_N, the unknown v_N, and $q = \phi_m^D(x)\exp(\imath n y)$ the continuity equation becomes

$$\begin{aligned} \int_{-1}^{1} \int_{0}^{2\pi} \frac{\partial}{\partial x} \left(\sum_{k \in \boldsymbol{k}_B} \sum_{l \in \boldsymbol{l}} \hat{u}_{kl}(t) \phi_k^B(x) \exp(\imath l y) \right) \phi_m^D(x) \exp(-\imath n y) dx_w dy_w + \\ \int_{-1}^{1} \int_{0}^{2\pi} \frac{\partial}{\partial y} \left(\sum_{k \in \boldsymbol{k}_D} \sum_{l \in \boldsymbol{l}} \hat{v}_{kl}(t) \phi_k^D(x) \exp(\imath l y) \right) \phi_m^D(x) \exp(-\imath n y) dx_w dy_w = 0. \end{aligned} \tag{7.180}$$

The x and y domains are separable, so we can rewrite as

$$\begin{aligned} \sum_{k \in \boldsymbol{k}_B} \sum_{l \in \boldsymbol{l}} \int_{-1}^{1} \frac{\partial \phi_k^B(x)}{\partial x} \phi_m^D(x) dx_w \int_{0}^{2\pi} \exp(\imath l y) \exp(-\imath n y) dy_w \hat{u}_{kl} + \\ \sum_{k \in \boldsymbol{k}_D} \sum_{l \in \boldsymbol{l}} \int_{-1}^{1} \phi_k^D(x) \phi_m^D(x) dx_w \int_{0}^{2\pi} \frac{\partial \exp(\imath l y)}{\partial y} \exp(-\imath n y) dy_w \hat{v}_{kl} = 0. \end{aligned} \tag{7.181}$$

Now perform some exact manipulations in the Fourier direction and introduce the 1D inner product notation for the x-direction

$$(a, b)_w = \int_{-1}^{1} a(x) b(x) dx_w \left(\approx \sum_{j=0}^{N-1} a(x_j) b(x_j) w(x_j) \right). \tag{7.182}$$

By also simplifying the notation using summation of repeated indices, we get the following equation

$$\delta_{ln} \left(\frac{\partial \phi_k^B}{\partial x}, \phi_m^D \right)_w \hat{u}_{kl} + \imath l \delta_{ln} \left(\phi_k^D, \phi_m^D \right)_w \hat{v}_{kl} = 0. \tag{7.183}$$

Now l must equal n and we can simplify some more

$$\left(\frac{\partial \phi_k^B}{\partial x}, \phi_m^D \right)_w \hat{u}_{kl} + \imath l \left(\phi_k^D, \phi_m^D \right)_w \hat{v}_{kl} = 0. \tag{7.184}$$

We see that this equation can be solved for $\hat{v}_{kl}$ for $(k, l) \in \boldsymbol{k}_D \times [1, 2, \ldots, M/2]$, but try with $l = 0$ and you hit division by zero, which obviously is not allowed. And this is the reason why Eq. (7.162) is still needed, to solve for $\hat{v}_{k,0}$! Fortunately, since $\exp(\imath 0 y) = 1$, the pressure derivative $\frac{\partial p}{\partial y} = 0$, and as such the pressure is still not required. When used only for Fourier coefficient 0, Eq. (7.162) becomes

$$\frac{\partial v}{\partial t} + N_y = \sqrt{\frac{Pr}{Ra}} \nabla^2 v. \tag{7.185}$$

There is still one more revelation to be made from Eq. (7.184). When $l = 0$ we get

$$\left(\frac{\partial \phi_k^B}{\partial x}, \phi_m^D\right)_w \hat{u}_{k,0} = 0, \tag{7.186}$$

and the only way to satisfy this is if $\hat{u}_{k,0} = 0$ for $k \in \boldsymbol{k}_B$. Bottom line is that we only need to solve Eq. (7.161) for $l \in \boldsymbol{l}/\{0\}$, whereas we can use directly $\hat{u}_{k,0} = 0$ for $k \in \boldsymbol{k}_B$.

To sum up, with the solution known at $t = t - \Delta t$, we solve

Equation	For unknown	With indices
(7.161)	$\hat{u}_{kl}(t)$	$(k,l) \in \boldsymbol{k}_B \times \boldsymbol{l}/\{0\}$
(7.164)	$\hat{v}_{kl}(t)$	$(k,l) \in \boldsymbol{k}_D \times \boldsymbol{l}/\{0\}$
(7.185)	$\hat{v}_{kl}(t)$	$(k,l) \in \boldsymbol{k}_D \times \{0\}$
(7.163)	$\hat{T}_{kl}(t)$	$(k,l) \in \boldsymbol{k}_T \times \boldsymbol{l}$

7.8.2 Temporal discretization

The governing equations are integrated in time using a semi-implicit third order Runge Kutta method. This method applies to any generic equation

$$\frac{\partial \psi}{\partial t} = \mathcal{N} + \mathcal{L}\psi, \tag{7.187}$$

where $\mathcal{N}$ and $\mathcal{L}$ represents the nonlinear and linear contributions, respectively. With time discretized as $t_n = n\Delta t$, $n = 0, 1, 2, ...$, the Runge Kutta method also subdivides each timestep into stages $t_n^k = t_n + c_k \Delta t$, $k = (0, 1, .., N_s - 1)$, where N_s is the number of stages. The third order Runge Kutta method implemented here uses three stages. On one timestep the generic equation (7.187) is then integrated from stage k to $k + 1$ according to

$$\psi^{k+1} = \psi^k + a_k \mathcal{N}^k + b_k \mathcal{N}^{k-1} + \frac{a_k + b_k}{2}\mathcal{L}(\psi^{k+1} + \psi^k), \tag{7.188}$$

which should be rearranged with the unknowns on the left hand side and the knowns on the right hand side

$$\big(1 - \frac{a_k + b_k}{2}\mathcal{L}\big)\psi^{k+1} = \big(1 + \frac{a_k + b_k}{2}\mathcal{L}\big)\psi^k + a_k \mathcal{N}^k + b_k \mathcal{N}^{k-1}. \tag{7.189}$$

For the three-stage third order Runge Kutta method the constants are given as

$a_n/\Delta t$	$b_n/\Delta t$	$c_n/\Delta t$
8/15	0	0
5/12	17/60	8/15
3/4	5/12	2/3

For the spectral Galerkin method used by `shenfun` the governing equation is first put in a weak variational form. This will change the appearence of Eq. (7.189) slightly. If ϕ is a test function, ψ^{k+1} the trial function, and ψ^k a known function, then the variational form of (7.189) is obtained by multiplying (7.189) by ϕ and integrating (with weights) over the domain

$$\Big\langle (1 - \frac{a_k + b_k}{2}\mathcal{L})\psi^{k+1}, \phi \Big\rangle_w = \Big\langle (1 + \frac{a_k + b_k}{2}\mathcal{L})\psi^k, \phi \Big\rangle_w + \Big\langle a_k \mathcal{N}^k + b_k \mathcal{N}^{k-1}, \phi \Big\rangle_w. \tag{7.190}$$

Equation (7.190) is the variational form implemented by `shenfun` for the time dependent equations.

7.8.3 Shenfun implementation

To get started we need instances of the approximation spaces discussed in Eqs. (7.165) - (7.173). When the spaces are created we also need to specify the family and the dimension of each space. Here we simply choose Chebyshev and Fourier with 100 and 256 quadrature points in x and y-directions, respectively. We could replace 'Chebyshev' by 'Legendre', but the former is known to be faster due to the existence of fast transforms.

```
from shenfun import *

N, M = 100, 256
family = 'Chebyshev'
VB = Basis(N, family, bc='Biharmonic')
VD = Basis(N, family, bc=(0, 0))
VW = Basis(N, family)
VT = Basis(N, family, bc=(0, 1))
VF = Basis(M, 'F', dtype='d')
```

And then we create tensor product spaces by combining these bases (like in Eqs. (7.170)-(7.173)).

```
W_BF = TensorProductSpace(comm, (VB, VF))     # Wall-normal velocity
W_DF = TensorProductSpace(comm, (VD, VF))     # Streamwise velocity
W_WF = TensorProductSpace(comm, (VW, VF))     # No bc
W_TF = TensorProductSpace(comm, (WT, VF))     # Temperature
BD = MixedTensorProductSpace([W_BF, W_DF])    # Velocity vector
DD = MixedTensorProductSpace([W_DF, W_DF])    # Convection vector
```

Here the last two lines create mixed tensor product spaces by the Cartesian products `BD = W_BF` $\times$ `W_DF` and `DD = W_DF` $\times$ `W_DF`. These mixed space will be used to hold the velocity and convection vectors, but we will not solve the equations in a coupled manner and continue in the segregated approach outlined above.

We also need containers for the computed solutions. These are created as

```
u_  = Function(BD)      # Velocity vector, two components
u_1 = Function(BD)      # Velocity vector, previous step
T_  = Function(W_TF)    # Temperature
T_1 = Function(W_TF)    # Temperature, previous step
H_  = Function(DD)      # Convection vector
H_1 = Function(DD)      # Convection vector previous stage

# Need a container for the computed right hand side vector
rhs_u = Function(DD).v
rhs_T = Function(DD).v
```

In the final solver we will also use bases for dealiasing the nonlinear term, but we do not add that level of complexity here.

Wall-normal velocity equation

We implement Eq. (7.161) using the three-stage Runge Kutta equation (7.190). To this end we first need to declare some test- and trial functions, as well as some model constants

```
u = TrialFunction(W_BF)
v = TestFunction(W_BF)
a = (8./15., 5./12., 3./4.)
b = (0.0, -17./60., -5./12.)
c = (0., 8./15., 2./3., 1)
```

(continues on next page)

(continued from previous page)

```
# Specify viscosity and time step size using dimensionless Ra and Pr
Ra = 10000
Pr = 0.7
nu = np.sqrt(Pr/Ra)
kappa = 1./np.sqrt(Pr*Ra)
dt = 0.1

# Get one solver for each stage of the RK3
solver = []
for rk in range(3):
    mats = inner(div(grad(u)) - ((a[rk]+b[rk])*nu*dt/2.)*div(grad(div(grad(u)))), v)
    solver.append(chebyshev.la.Biharmonic(*mats))
```

Notice the one-to-one resemblance with the left hand side of (7.190), where ψ^{k+1} now has been replaced by $\nabla^2 u$ (or `div(grad(u))`) from Eq. (7.161). For each stage we assemble a list of tensor product matrices `mats`, and in `chebyshev.la` there is available a very fast direct solver for exactly this type of (biharmonic) matrices. The solver is created with `chebyshev.la.Biharmonic(*mats)`, and here the necessary LU-decomposition is carried out for later use and reuse on each time step.

The right hand side depends on the solution on the previous stage, and the convection on two previous stages. The linear part (first term on right hand side of (7.189)) can be assembled as

```
inner(div(grad(u_[0])) + ((a[rk]+b[rk])*nu*dt/2.)*div(grad(div(grad(u_[0])))), v)
```

The remaining parts $\frac{\partial^2 H_y}{\partial x \partial y} - \frac{\partial^2 H_x}{\partial y \partial y} + \frac{\partial^2 T}{\partial y^2}$ end up in the nonlinear $\mathcal{N}$. The nonlinear convection term $\boldsymbol{H}$ can be computed in many different ways. Here we will make use of the identity $(\boldsymbol{u} \cdot \nabla)\boldsymbol{u} = -\boldsymbol{u} \times (\nabla \times \boldsymbol{u}) + 0.5\nabla \boldsymbol{u} \cdot \boldsymbol{u}$, where $0.5\nabla \boldsymbol{u} \cdot \boldsymbol{u}$ can be added to the eliminated pressure and as such be neglected. Compute $\boldsymbol{H} = -\boldsymbol{u} \times (\nabla \times \boldsymbol{u})$ by first evaluating the velocity and the curl in real space. The curl is obtained by projection of $\nabla \times \boldsymbol{u}$ to the no-boundary-condition space `W_TF`, followed by a backward transform to real space. The velocity is simply transformed backwards.

Note: If dealiasing is required, it should be used here to create padded backwards transforms of the curl and the velocity, before computing the nonlinear term in real space. The nonlinear product should then be forward transformed with truncation. To get a space for dealiasing, simply use, e.g., `W_BF.get_dealiased()`.

```
# Get a mask for setting Nyquist frequency to zero
mask = W_DF.get_mask_nyquist()

def compute_convection(u, H):
    curl = project(Dx(u[1], 0, 1) - Dx(u[0], 1, 1), W_TF).backward()
    ub = u.backward()
    H[0] = W_DF.forward(-curl*ub[1])
    H[1] = W_DF.forward(curl*ub[0])
    H.mask_nyquist(mask)
    return H
```

Note that the convection has a homogeneous Dirichlet boundary condition in the non-periodic direction. With convection computed we can assemble $\mathcal{N}$ and all of the right hand side, using the function `compute_rhs_u`

```
def compute_rhs_u(u, T, H, rhs, rk):
    v = TestFunction(W_BF)
    H = compute_convection(u, H)
    rhs[1] = 0
```

(continues on next page)

(continued from previous page)

```
    rhs[1] += inner(v, div(grad(u[0])) + ((a[rk]+b[rk])*nu*dt/2.
↪)*div(grad(div(grad(u[0])))))
    w0 = inner(v, Dx(Dx(H[1], 0, 1), 1, 1) - Dx(H[0], 1, 2))
    w1 = inner(v, Dx(T, 1, 2))
    rhs[1] += a[rk]*dt*(w0+w1)
    rhs[1] += b[rk]*dt*rhs[0]
    rhs[0] = w0+w1
    rhs.mask_nyquist(mask)
    return rhs
```

Note that we will only use `rhs` as a container, so it does not actually matter which space it has here. We're using `.v` to only access the Numpy array view of the Function. Also note that `rhs[1]` contains the right hand side computed at stage `k`, whereas `rhs[0]` is used to remember the old value of the nonlinear part.

Streamwise velocity

The streamwise velocity is computed using Eq. (7.184) and (7.185). For efficiency we can here preassemble both matrices seen in (7.184) and reuse them every time the streamwise velocity is being computed. We will also need the wavenumber l, here retrived using `W_BF.local_wavenumbers(scaled=True)`. For (7.185) we preassemble the required Helmholtz solvers, one for each RK stage.

```
# Assemble matrices and solvers for all stages
B_DD = inner(TestFunction(W_DF), TrialFunction(W_DF))
C_DB = inner(TestFunction(W_DF), Dx(TrialFunction(W_BF), 0, 1))
v0 = TestFunction(VD)
u0 = TrialFunction(VD)
solver0 = []
for rk in range(3):
    mats0 = inner(v0, 2./(nu*(a[rk]+b[rk])*dt)*u0 - div(grad(u0)))
    solver0.append(chebyshev.la.Helmholtz(*mats0))

# Allocate work arrays and variables
u00 = Function(VD)
b0 = np.zeros((2,)+u00.shape)
w00 = np.zeros_like(u00)
dudx_hat = Function(W_DF)
K = W_BF.local_wavenumbers(scaled=True)[1]

def compute_v(u, rk):
    if comm.Get_rank() == 0:
        u00[:] = u_[1, :, 0].real
    dudx_hat = C_DB.matvec(u[0], dudx_hat)
    with np.errstate(divide='ignore'):
        dudx_hat = 1j * dudx_hat / K
    u[1] = B_DD.solve(dudx_hat, u=u[1])

    # Still have to compute for wavenumber = 0
    if comm.Get_rank() == 0:
        b0[1] = inner(v0, 2./(nu*(a[rk]+b[rj])*dt)*Expr(u00) + div(grad(u00)))
        w00 = inner(v0, H_[1, :, 0])
        b0[1] -= (2.*a/nu/(a[rk]+b[rk]))*w00
        b0[1] -= (2.*b/nu/(a[rk]+b[rk]))*b0[0]
        u00 = solver0[rk](u00, b0[1])
        u[1, :, 0] = u00
        b0[0] = w00
```

(continues on next page)

(continued from previous page)

```
    return u
```

Temperature

The temperature equation (7.158) is implemented using a Helmholtz solver. The main difficulty with the temperature is the non-homogeneous boundary condition that requires special attention. A non-zero Dirichlet boundary condition is implemented by adding two basis functions to the basis of the function space

$$\phi^D_{N-2} = 0.5(1+x), \tag{7.191}$$

$$\phi^D_{N-1} = 0.5(1-x), \tag{7.192}$$

with the approximation now becoming

$$T_N(x,y,t) = \sum_{k=0}^{N-1}\sum_{l\in\boldsymbol{l}} \hat{T}_{kl}\phi^D_k(x)\exp(\imath l y), \tag{7.193}$$

$$= \sum_{k=0}^{N-3}\sum_{l\in\boldsymbol{l}} \hat{T}_{kl}\phi^D_k(x)\exp(\imath l y) + \sum_{k=N-2}^{N-1}\sum_{l\in\boldsymbol{l}} \hat{T}_{kl}\phi^D_k(x)\exp(\imath l y). \tag{7.194}$$

The boundary condition requires

$$T_N(1,y,t) = \sum_{k=N-2}^{N-1}\sum_{l\in\boldsymbol{l}} \hat{T}_{kl}\phi^D_k(1)\exp(\imath l y), \tag{7.195}$$

$$= \sum_{l\in\boldsymbol{l}} \hat{T}_{N-2,l}\exp(\imath l y), \tag{7.196}$$

and

$$T_N(-1,y,t) = \sum_{k=N-2}^{N-1}\sum_{l\in\boldsymbol{l}} \hat{T}_{kl}\phi^D_k(-1)\exp(\imath l y), \tag{7.197}$$

$$= \sum_{l\in\boldsymbol{l}} \hat{T}_{N-1,l}\exp(\imath l y). \tag{7.198}$$

We find $\hat{T}_{N-2,l}$ and $\hat{T}_{N-1,l}$ using orthogonality. Multiply (7.196) and (7.198) by $\exp(-\imath m y)$ and integrate over the domain $[0, 2\pi]$. We get

$$\hat{T}_{N-2,l} = \int_0^{2\pi} T_N(1,y,t)\exp(-\imath l y)dy, \tag{7.199}$$

$$\hat{T}_{N-1,l} = \int_0^{2\pi} T_N(-1,y,t)\exp(-\imath l y)dy. \tag{7.200}$$

Using this approach it is easy to see that any inhomogeneous function $T_N(\pm 1, y, t)$ of y and t can be used for the boundary condition, and not just a constant. To implement a non-constant Dirichlet boundary condition, the `Basis` function can take any `sympy` function of `(y, t)`, for exampel by replacing the creation of `VT` by

```
import sympy as sp
y, t = sp.symbols('y,t')
f = 0.9+0.1*sp.sin(2*(y))*sp.exp(-t)
VT = Basis(N, family, bc=(0, f))
```

For merely a constant `f` or a `y`-dependency, no further action is required. However, a time-dependent approach requires the boundary values to be updated each time step. To this end there is the function `BoundaryValues.update_bcs_time`, used to update the boundary values to the new time. Here we will assume a time-independent boundary condition, but the final implementation will contain the time-dependent option.

Due to the non-zero boundary conditions there are also a few additional things to be aware of. Assembling the coefficient matrices will also assemble the matrices for the two boundary test functions. That is, for the 1D mass matrix with $u = \sum_{k=0}^{N-1} \hat{T}_k \phi_k^D$ and $v = \phi_m^D$, we will have

$$(u, v)_w = \left(\sum_{k=0}^{N-1} \hat{T}_k \phi_k^D(x), \phi_m^D \right)_w, \tag{7.201}$$

$$= \sum_{k=0}^{N-3} \left(\phi_k^D(x), \phi_m^D\right)_w \hat{T}_k + \sum_{k=N-2}^{N-1} \left(\phi_k^D(x), \phi_m^D\right)_w \hat{T}_k, \tag{7.202}$$

where the first term on the right hand side is the regular mass matrix for a homogeneous boundary condition, whereas the second term is due to the non-homogeneous. Since $\hat{T}_{N-2}$ and $\hat{T}_{N-1}$ are known, the second term contributes to the right hand side of a system of equations. All boundary matrices can be extracted from the lists of tensor product matrices returned by `inner`. For the temperature equation these boundary matrices are extracted using `extract_bc_matrices` below. The regular solver is placed in the `solverT` list, one for each stage of the RK3 solver.

```
solverT = []
lhs_mat = []
for rk in range(3):
    matsT = inner(q, 2./(kappa*(a[rk]+b[rk])*dt)*p - div(grad(p)))
    lhs_mat.append(extract_bc_matrices([matsT]))
    solverT.append(chebyshev.la.Helmholtz(*matsT))
```

The boundary contribution to the right hand side is computed for each stage as

```
rhs_T = lhs_mat[rk][0].matvec(T_, rhs_T)
```

The complete right hand side of the temperature equations can be computed as

```
def compute_rhs_T(u, T, rhs, rk):
    q = TestFunction(W_TF)
    rhs[1] = inner(q, 2./(kappa*(a[rk]+b[rk])*dt)*Expr(T)+div(grad(T)))
    rhs[1] -= lhs_mat[rk][0].matvec(T, w0)
    ub = u.backward()
    Tb = T.backward()
    uT_ = BD.forward(ub*Tb)
    w0[:] = 0
    w0 = inner(q, div(uT_), output_array=w0)
    rhs[1] -= (2.*a/kappa/(a[rk]+b[rk]))*w0
    rhs[1] -= (2.*b/kappa/(a[rk]+b[rk]))*rhs[0]
    rhs[0] = w0
    rhs.mask_nyquist(mask)
    return rhs
```

We now have all the pieces required to solve the Rayleigh Benard problem. It only remains to perform an initialization and then create a solver loop that integrates the solution forward in time.

```
# initialization
T_b = Array(W_TF)
X = W_TF.local_mesh(True)
```

(continues on next page)

(continued from previous page)

```
T_b[:] = 0.5*(1-X[0]) + 0.001*np.random.randn(*T_b.shape)*(1-X[0])*(1+X[0])
T_ = T_b.forward(T_)
T_.mask_nyquist(mask)

def solve(t=0, tstep=0, end_time=1000):
    while t < end_time-1e-8:
        for rk in range(3):
            rhs_u = compute_rhs_u(u_, T_, H_, rhs_u, rk)
            u_[0] = solver[rk](u_[0], rhs_u[1])
            if comm.Get_rank() == 0:
                u_[0, :, 0] = 0
            u_ = compute_v(u_, rk)
            u_.mask_nyquist(mask)
            rhs_T = compute_rhs_T(u_, T_, rhs_T, rk)
            T_ = solverT[rk](T_, rhs_T[1])
            T_.mask_nyquist(mask)

        t += dt
        tstep += 1
```

A complete solver implemented in a solver class can be found in RayleighBenardRk3.py, where some of the terms discussed in this demo have been optimized some more for speed. Note that in the final solver it is also possible to use a (y, t)-dependent boundary condition for the hot wall. And the solver can also be configured to store intermediate results to an `HDF5` format that later can be visualized in, e.g., Paraview. The movie in the beginning of this demo has been created in Paraview.

BIBLIOGRAPHY

[PSS18] Ambrish Pandey, Janet D. Scheel, and Jörg Schumacher. Turbulent superstructures in rayleigh-bénard convection. *Nature Communications*, 9(1):2118, 2018. doi:10.1038/s41467-018-04478-0.

[She94] Jie Shen. Efficient spectral-galerkin method i. direct solvers of second- and fourth-order equations using legendre polynomials. *SIAM Journal on Scientific Computing*, 15(6):1489–1505, 1994. doi:10.1137/0915089.

[She95] Jie Shen. Efficient spectral-galerkin method ii. direct solvers of second- and fourth-order equations using chebyshev polynomials. *SIAM Journal on Scientific Computing*, 16(1):74–87, 1995. doi:10.1137/0916006.

[She97] Jie Shen. Efficient spectral-galerkin methods iii: polar and cylindrical geometries. *SIAM Journal on Scientific Computing*, 18(6):1583–1604, 1997. doi:10.1137/S1064827595295301.

[Waz08] Abdul-Majid Wazwaz. New travelling wave solutions to the boussinesq and the klein-gordon equations. *Communications in Nonlinear Science and Numerical Simulation*, 13(5):889–901, 2008. doi:10.1016/j.cnsns.2006.08.005.