2.9 Efficient implementations in three dimensions

For simplicity, we only consider solving $-u_{xx} - u_{yy} - u_{zz} = f$ on a cube $\Omega = [0,1]^3$ with homogeneous Dirichlet boundary conditions.

Consider a uniform Cartesian mesh in 3D with grid points (x_i, y_j, z_k) $(i = 1, \cdots, N_x; j = 1, \cdots, N_y; k = 1, \cdots, N_z)$.

Following our choice of notation for 2D problems, for representing numerical solutions, let U be a 3D array of size $N_z \times N_y \times N_x$ with (k, j, i)-th entry $U(k, j, i)$ denoting the point value at (x_i, y_j, z_k).

For a 3D array U, we define a page as the matrix obtained by fixing the last index of U. Namely, $U(:, :, i)$ for any fixed i is a page of U. For a matrix $U(:, :, i)$ of size $N_z \times N_y$, recall that $vec(U(:, :, i))$ is a column vector of size $N_z N_y$. We define $\hat{U}$ as the following matrix of size $N_z N_y \times N_x$ obtained by reshaping U:

$$\hat{U} = \left[vec(U(:, :, 1))\ vec(U(:, :, 2)) \cdots vec(U(:, :, N_x)) \right].$$

Then we define $vec(U)$ as the vector of size $N_z N_y N_x \times 1$ by reshaping $\hat{U}$ in a column by column order.

With the notation above, it is straightforward to verify that

$$(A_1^\top \otimes A_2^\top \otimes A_3) vec(U) = vec((A_2^\top \otimes A_3)\hat{U} A_1). \tag{2.10}$$

Let Y be a 3D array of size $N_z \times N_y \times N_x$ defined by

$$vec(Y) = (A_1^\top \otimes A_2^\top \otimes A_3) vec(U). \tag{2.11}$$

With the simple property (2.10), among other choices, one simple and efficient implementation of multiplying the big matrix $A_1^\top \otimes A_2^\top \otimes A_3$ in (2.11) in MATLAB is to use two functions *tensorprod* and *pagemtimes*:

```
% Computing a 3D array Y of the same size as U defined above
Y = tensorprod(U,A1,3,1);
Y = pagemtimes(Y,A2);
Y = squeeze(tensorprod(A3,Y,2,1));
```

Let F be a 3D array with $F(k, j, i) = f(x_i, y_j, z_k)$, then the second order finite difference scheme can be written as

$$(K_x \otimes I_y \otimes I_z + I_x \otimes K_y \otimes I_z + I_x \otimes I_y \otimes K_z)\, vec(U) = vec(F).$$

With the eigenvalue decomposition $K = S\Lambda S^{-1}$, similar to the derivation 2D problems, the scheme is equivalent to

$$(S_x \otimes S_y \otimes S_z)\,(\Lambda_x \otimes I_y \otimes I_z + I_x \otimes \Lambda_y \otimes I_z + I_x \otimes I_y \otimes \Lambda_z)\,(S_x^{-1} \otimes S_y^{-1} \otimes S_z^{-1}) vec(U) = vec(F).$$

Define a 3D array Λ with its (k, j, i)-th entry being equal to $\Lambda_x(i, i) + \Lambda_y(j, j) + \Lambda_z(k, k)$, then it can be implemented efficiently as the follows in MATLAB:

```
% InvS denotes the inverse matrix of S
U = tensorprod(F,InvSx',3,1);
U = pagemtimes(U,InvSy');
U = squeeze(tensorprod(InvSz,U,2,1));
U = U./Lambda;
U = tensorprod(U,Sx',3,1);
U = pagemtimes(U,Sy');
U = squeeze(tensorprod(Sz,U,2,1));
```

Recall that $S^{-1} = S$ if we use orthonormal eigenvectors, as pointed out in Remark 2.9. In *Python*, with package *Jax*, it can be similarly implemented uisng the function *jax.numpy.einsum*:

```
    U = jnp.einsum('ijk,kl->ijl',F,invSx.transpose())
    U = jnp.einsum('ijk,jl->ilk',U,invSy.transpose())
    U = jnp.einsum('li,ijk->ljk',invSz,U)
    U = U/Lambda
    U = jnp.einsum('ijk,kl->ijl',U,Sx.transpose())
    U = jnp.einsum('ijk,jl->ilk',U,Sy.transpose())
    U = jnp.einsum('li,ijk->ljk',Sz,U)
```

Remark 2.10. *See [Liu* et al. *(2024b)] for how to use the MATLAB and Python scripts above on a GPU device. In particular, on one Nividia A100 80G GPU card, it costs around* 0.8 *seconds for inverting one 3D discrete Laplacian of size* 1000^3.

Problem 2.20. *Derive the exact number of operations (only counting the number of multiplication/division) needed for the method in this section inverting* $K \otimes I \otimes I + I \otimes K \otimes I + I \otimes I \otimes K$ *where* K *and* I *are matrices of size* $N \times N$.

Problem 2.21. *Write a MATLAB code for solving* $-u_{xx} - u_{yy} - u_{zz} = 14\pi^2 \sin(\pi x)\sin(2\pi y)\sin(3\pi z)$ *with homogeneous Dirichlet b.c. on the domain* $[0,1]^3$. *Test your code for the exact solution* $u(x,y,z) = \sin(\pi x)\sin(2\pi y)\sin(3\pi z)$. *Test the convergence rate on several different grids. On a modern desktop computer, the implementation in this section should be quite efficient for a grid of size* 200^3 *or even* 300^3. *If your computer has large enough memory, try to push the grid size to as large as it allows.*

Problem 2.22. *If you have access to computers on which you can run MATLAB with GPU, follow the script Poisson3D.m in [Liu* et al. *(2024b),*