SWEET's data containers

Martin Schreiber

February 23, 2019

Version information

- 2016-10-05: First release
- 2019-02-23: Updated version

1 Introduction

SWEET provides various data containers and conversions between physical and spectral spaces. This document provides the information on the reasons how the simulation data is abstracted into classes.

Running simulations with global spectral methods turns out to be a task which also impacts software design. With the requirement of supporting simulations on the plane with the double-Fourier and on the sphere with Spherical Harmonics, also the way how data is stored and processed should be generated not only in a programmable way, but should also consider the underlying mathematical and performance requirements. Additional requirements arise from 3rd party libraries.

The focus will be put on simulations on the sphere.

2 Requirements

2.1 Mathematics

1. Non-linearities: Support for anti-aliasing
2. Multi-level methods: Support of different resolutions in spectral and physical space

2.2 Software framework

1. Programmability:
   (a) Support operator-styled arithmetic operations (e.g. a+b where a and b are data arrays)
   (b) Hide (parallel) iteration loops and anti-aliasing if requested
2. Clear interfaces: Support sphere with SPH and plane data with FFT in a similar way

2.3 HPC

1. Parallelization: Efficient implementation on nowadays HPC shared-memory systems (e.g. caring about NUMA awareness)
2. Memory: No additional memory consumption

3 Spectral methods: FFT and SPH

We rely on existing libraries to convert data from spectral to physical space. Libraries for the FFTW have restrictions which is the reason for this discussion document.

3.1 Requirements for spectral/physical conversions

Let (N, M) be the number of modes in spectral space and (X, Y) the resolution in physical space. For the FFT used for simulations on the plane, we use the FFTW. For the SPH for simulations on the sphere we use the SHTns library. We directly encounter a restriction by using the FFTW. This only supports transformations if with $X = N$ and $Y = M$.

On the other hand, SHTns and also related libraries for spherical harmonics support different size of spectral and physical spaces with $X! = N$ and $Y! = M$.

3.2 Comment on "plans"

Both libraries require the creation of "plans". These plans should be created only once and then reused. A reference counter is used to free plans if they are not required anymore.

3.3 (Anti-)aliasing

3.3.1 No anti-aliasing

Without anti-aliasing it is sufficient to have the same number of modes in spectral space as the resolution is in physical space: $X = N$ and $Y = M$.

3.3.2 SPH Anti-aliasing

Anti-aliasing treatment is required for evaluating non-linearities. Even for a linear operators, such non-linearities might be required to multiply e.g. with $(1-\mu^2)^{-1}$ which is done in physical space if using spherical harmonics. We only focus on implementations which support a fixed physical resolution for a given number of modes. (In the current implementation the physical space is always $X = N$ and $Y = M$ which might result in certain issues regarding anti-aliasing modes.)

Using SPH libraries, they typically support resolutions and a maximum number of modes which are not equivalent and we can easily support anti-aliasing. Here, the 2/3 rule can be directly realized by using different sizes for the spatial/spectral space. Just by choosing a lower number of modes allows using only one buffer for spectral space of size $N \times M$ and another buffer for the data in spatial space of size $X \times Y$.

3.3.3 FFTW anti-aliasing

However, FFTW requires to have identical spatial resolution and number of spectral modes, hence $X = N$ and $Y = M$. To cope with anti-aliasing, we require to run a FFT with a larger spectrum but with the higher modes set to zero before each of these FFTs. The size of the spectrum is given by (NL, ML) and is typically $(NL, ML) = (N3/2, M3/2)$. We also discuss how to handle the operators (add, sub) in spectral space. We continue to discuss different methods to cope with this issue:

- **Two buffers with padding:**
  MUL: There are two separate buffers of size (N, M) and (NL, ML). Computing a multiplication in physical space is then accomplished by

  1. Copy data from (N, M) to (NL, ML) and use *zero padding*.
  2. Use FFT with $(NL = X, ML = Y)$ to compute representation in physical space
  3. [multiplication in physical space]
  4. Use inverse FFT with $(NL = X, ML = Y)$ to compute representation in spectral space
  5. Copy corresponding modes from (NL, ML) to (N, M).

  ADD/SUB: All other operators such as adding two spectral representations can be then realized by a single for loop without knowing about boundaries of (N, M) and (NL, ML) but just of $I = N \times M$.
  DRAWBACKS: Handling of two buffers and either always allocating an additional buffer or requiring to allocate a buffer if required.
  ADVANTAGES: Optimal application of spectral operators due to the single for loop.

- **Merged buffers to single one with padding:**
  This idea is basically the same one as the one before. However, the difference is that both buffers are shared.
  DRAWBACKS: Copy data from (N, M) to (NL, ML) and use zero padding might not be accomplished in parallel
  ADVANTAGES: Avoid using 2nd buffer, optimal application of spectral operators.

- **Use only single spectral buffer with special loop iterations:**
  Here, we allocate a buffer of size (NL, ML) and handle all ADD/SUB of spectral data by only partial loop intervals (N, M). Computing a multiplication in physical space is then accomplished by

  1. Zero out modes $(> N, > M)$.
  2. Use FFT with $(NL = X, ML = Y)$ to compute representation in physical space
  3. [multiplication in physical space]
  4. Use inverse FFT with $(NL = X, ML = Y)$ to compute representation in spectral space

  ADD/SUB: A special treatment is required to handle the iterations in spectral space in order to avoid iterating over (NL, ML) since only an iteration over (N, M) is required.
  DRAWBACKS: Special iteration in spectral space required, maybe not optimal
  ADVANTAGES: Only a single buffer, reassembles the SPH implementation

4 Physical data

We require physical data to e.g. setup the initial conditions and, related to the spectral data, to evaluate multiplications in pseudo-spectral space.

5 Data Arrays

The data arrays are required as pure arrays, containing e.g. a list of departure points for semi-Lagrangian methods.

6 Realization

This is the summary of the different sphere-related classes in SWEET.

6.1 Data storage (SphereData_*)

- **SphereData_Physical**: Real-valued data in physical space
- **SphereData_PhysicalComplex**: Complex data in physical space
- **SphereData_Spectral**: Data in spectral space, corresponding to real-valued data in physical space
- **SphereData_SpectralComplex**: Spectral data based on complex-valued physical data
- **SphereData_Config**: Information how to convert data from/to physical/spectral space

6.2 Operations with data(SphereOperators_*)

- **SphereOperators_SphereData**: Real-valued operators
- **SphereOperators_SphereDataComplex**: Operators for complex-valued data
- **SphereOperators_Sampler_SphereDataPhysical**: Sampling operators for real-valued physical data

6.3 Time integration (SphereTimestepping_*)

- **SphereTimestepping_ExplicitLeapfrog**: Realization of Explicit Leapfrog time integration
- **SphereTimestepping_ExplicitRK**: Explicit time integration with Runge-Kutta
- **SphereTimestepping_SemiLagrangian**: SL-time stepping method

6.4 Helper routines (SphereHelpers_*)

- **SphereHelpers_Coordinates**: Helper routines for transformation coordinates between different systems
- **SphereHelpers_Diagnostics**: Extract diagnostic information from sphere data
- **SphereHelpers_SPHIdentities**: Useful identities for Spherical Harmonics

7 Converting between different data representations

7.1 Target: SphereDataPhysical

from\to	SphereDataPhysical
SphereDataPhysical	=
SphereDataPhysicalComplex	Convert_SphereDataPhysicalComplex_to_SphereDataPhysical
ScalarDataArray	Convert_ScalarDataArray_to_SphereDataPhysical
SphereDataSpectral	SphereDataSpectral.getSphereDataPhysical()

7.2 Target: SphereDataPhysicalComplex

from\to	SphereDataPhysicalComplex
SphereDataPhysical	[TODO]
SphereDataPhysicalComplex	=
ScalarDataArray	(not supported)
SphereDataSpectral	(not supported)
SphereDataSpectralComplex	SphereDataSpectralComplex.getSphereDataPhysicalComplex()

7.3 Target: ScalarDataArray

SphereDataPhysical is the only data field which can be used to setup ScalarDataArrays.

from\to	ScalarDataArray
SphereDataPhysical	Convert_SphereDataPhysical_to_ScalarDataArra
SphereDataPhysicalComplex	(not supported)
ScalarDataArray	=
SphereDataSpectral	(not supported)
SphereDataSpectralComplex	(not supported)

7.4 Target: SphereDataSpectral

from\to	SphereDataSpectral
SphereDataPhysical	SphereDataSpectral.loadSphereDataPhysical(...)
SphereDataPhysicalComplex	(not supported)
ScalarDataArray	(not supported)
SphereDataSpectral	=
SphereDataSpectralComplex	Convert_SphereDataSpectralComplex_to_SphereDataSpectral

7.5 Target: SphereDataSpectralComplex

from\to	SphereDataSpectralComplex
SphereDataPhysical	SphereDataSpectral.loadSphereDataPhysical(...)
SphereDataPhysicalComplex	(not supported)
ScalarDataArray	(not supported)
SphereDataSpectral	Convert_SphereDataSpectral_to_SphereDataSpectralComplex
SphereDataSpectralComplex	=