Data Layout

Consider a problem with P many scalar fields defined on a $D = 1 + D_{trans}$ dimensional domain.

There are two *fundamental units* that motivate our data layout:

1. A scalar field: a D dimensional array of points/modes describing the values of one field in space.
2. A "z pencil": a vector describing all fields across all z for one point/mode in the transverse domain.

Fast transforms require that memory is allocated by field, but pencils are the fundamental computational unit: each one corresponds to a unique $k_\perp$ with corresponding M and L matrices. This means that differentiation, z transforms, A and B matrix construction, and solves for each pencil are independent.

The idea is to allow the *transverse* transform routines (namely PFFTW) to divide the data however they like across the *transverse* domain, and then to build pencil objects, one for each $k_\perp$ assigned to a processor, using views into the distributed fields. As far as the rest of the code is concerned, each processor can have an arbitrary collection of complete pencils. We don't really care how the transforms distribute things in x space (i.e. transposed or not), since non-linear products are computed element-by-element and then transformed back into pencils.

The basic process is then:

- Locally compute necessary derivatives in k-space
- Globally perform transforms to real space
- Compute non-linear parts of F element-by-element
- Globalls transform back, yielding F pencils
- Locally construct Tau matrices and solve

Here, *local* means in-pencil. Since these operations are independent amongst pencils within a given processor, they may be further parallelized via e.g. multithreading.

The pencil-based approach also means that only the transforms are sensitive to the dimensionality of the problem: the rest of the code just sees a set of m pencils, where $m = 1$ for $D = 1$, $m = n_x$ for $D = 2$, and $m = n_x \times n_y$ for $D = 3$.

Conceptual overview

To motivate our class layout, we work backwards from the final solve until we've categorized and defined everything needed to iteratively update the solution.

Tau solve:

- Produces: $\mathcal{X}$ for each pencil → solution system u_{n+1}
- Requires: $\mathcal{M}, \mathcal{Y}$ for each pencil

Tau setup:

- Produces: $\mathcal{M}, \mathcal{Y}$ for each pencil
- Requires: A, B, f, c_L, c_R, b for each pencil; $\mathcal{S}, \mathcal{D}, \delta_L, \delta_R, e$ for the primary basis

Timestepper:

- Produces: A, B, f for each pencil
- Requires: M, L, F for each pencil; timestep δt; current system u_n; primary derivative (for $\partial_z u_n$)

Non-linear evaluator:

- Produces: F for each pencil
- Requires: current system u_n; primary derivative; transverse derivative; primary transform; transverse transform; F instructions

Primary basis:

- Defines: $\mathcal{S}, \mathcal{D}, \delta_L, \delta_R, e$; primary derivative; primary transform

Transverse basis:

- Defines: transverse derivative; transverse transform

Problem definition:

- Defines: M, L, c_L, c_R, b for each pencil; initial timestep δt_0; initial system u_0; F instructions

Class structure

The classes are layed out following this structure.

PrimaryBasis:

- Defines primary transforms, differentiation, and matrices needed for the Kronecker products
- Similar to a Dedalus Representation class, but without data storage.

TransverseBasis (**not implemented**):

- Defines transverse transforms and differentiation
- Controls data distribution across multiple processors

Domain:

- Encapsulates Primary and Transverse bases so that upstream objects don't have to know about dimensionality of transverse domain, etc.

Field:

- Actual data container for a field; can transform itself and take derivative by using domain and basis objects
- *Similar to Dedalus Representation class but without defining own transform

routines

System:

- Object containing the set of fields defined in the problem (in an ordered dictionary)
- Similar to Dedalus StateData class

Problem:

- Defines necessary fields, problem matrices, and directions for computing F (**not implemented**)
- Replaces Physics in Dedalus

Pencil:

- One instance exists for each $k_\perp$ on local processor
- Contains local M, L matrices with transverse derivative operators substituted in

TimeStepper:

- Builds A, B, f matrices for each pencil based on problem matrices
- Contains auxiliary systems needed for multistep methods, etc
- Similar to Dedalus' timestepper, but doesn't perform the actual solve for the next solution

Integrator:

- Coordinates evolution: makes pencil objects and directs timestepper to compute matrices
- Builds and solves Tau system to actually evolve the solution

Issues

The code can currently solve 1D linear, homogeneous, constant coefficient problems.

Not implemented:

- Transverse bases
- 2D or 3D fields and domains
- System for specifying and computing F (currently set to 0)
- Transform structure for z-dependent M, L

Other issues / things to think about:

- Python 3 compatability. Most (all?) dependencies now support it
- It might be worth changing to an out-of-place transform structure. We currently accumulate some round-off error just by changing fields from kspace to xspace because we want to view the xdata, not because we actually need to make changes in xspace. By keeping the fields in kspace (by copying kdata to a new field before transforming, etc.), we can reduce this unneccesary transform error.
- For equations that are second order in time (i.e. wave equation), we need "velocity" variables to reduce the equation to a first order system. For these equations, the basis arrays shouldn't drop the last row, because there isn't a spatial boundary condition to enforce.