GLNumericalModelingKit Transforms

Jeffrey J. Early

October 14, 2013

1 Introduction

GLNumericalModelingKit is an Objective-C framework designed to enable high performance numerical modeling of differential equations.

The primary design goal of this framework is to eliminate the tedious an error prone programming required for numerical modeling by introducing a layer of abstraction on top of the usual code. From the outset, I wanted to be able to simply write `[psi x]` in objective-c and have that automatically take the x derivative of my variable psi, instead of fussing with memory buffers, differentiation matrices and Fourier transforms. GLNumericalModelingKit makes that possible, but also tries to be flexible enough to accommodate the unique demands that inevitably come with each project.

GLNumericalModelingKit consists of three major classes *dimensions*, *variables*, and *variable operations* that map to familiar mathematical concepts. There are three types of variables, *scalars* (which have no dimensions), *functions* (which are defined on a set of dimensions) and *linear transforms* (which transform functions between dimensions). The variable operations are used to perform all sorts of operations on the variables, including things such as addition, multiplication, interpolation, exponentiation, writing to files, Runge-Kutta time stepping, and so on. The linear transformations are used for Fourier, sine, cosine and Chebyshev transformations, differentiation, and other custom defined transformations. Taken together, these allow one to solve a fairly diverse set of mathematical problems.

2 Dimensions

Dimensions are what you would expect, defining a coordinate with respect to some basis. Given that this is a numerical modeling framework, dimensions are defined on *discrete* grids, rather than a continuous interval of some sort.

2.1 Basis

All dimensions are defined with respect some basis. This determines how a function is interpreted and how a function is differentiated. For example, the default basis is kGLDeltaBasis, conceptually a series of delta functions at each point.

At the heart of GLNumericalModelingKit is the ability to transform a variable between basis functions. If a one-dimensional function is defined on a dimension with basis kGLDeltaBasis, then you may transform this to kGLExponentialBasis or kGLCosineHalfShiftBasis, for example. The new coordinate defined with respect to the cosine basis, for example, should be thought of as coefficients of the cosine series.

Basis Name	Discrete value	Basis
`kGLDeltaBasis`	x_n	$\delta(x - x_n)$
`kGLExponentialBasis`	f_n	$e^{-2\pi i f_n x}$
`kGLSineHalfShiftBasis`	f_n	$\sin(2\pi f_n x)$
`kGLCosineHalfShiftBasis`	f_n	$\cos(2\pi f_n x)$
`kGLSineBasis`	f_n	$\sin(2\pi f_n x)$
`kGLCosineBasis`	f_n	$\cos(2\pi f_n x)$
`kGLChebyshevBasis`	n	$T_n(x)$

The different sine and cosine basis are due to different assumptions about the symmetry of the underlying function.

2.2 Grids

A particular grid can be defined given the length L and total number of points N. The following grids are supported,

Grid Name	$x_n - x_{\min}$	Use
`kGLEndpointGrid`	$n\left(\frac{L}{N-1}\right)$	sine & cosine transform, finite difference
`kGLInteriorGrid`	$\left(n+\frac{1}{2}\right)\left(\frac{L}{N}\right)$	sine & cosine half-shift transform
`kGLPeriodicGrid`	$n\left(\frac{L}{N}\right)$	exponential transform
`kGLChebyshevEndpointGrid`	$\frac{L}{2}\cos\left(\frac{\pi n}{N-1}\right)+\frac{L}{2}$	Chebyshev transform
`kGLChebyshevInteriorGrid`	$\frac{L}{2}\cos\left(\frac{\pi(n+\frac{1}{2})}{N}\right)+\frac{L}{2}$	Chebyshev transform

There are a standard collection of grids that are used, some of which are well suited to transformations into a particular basis.

Note that the Chebyshev grids are monotonically *decreasing*, a necessary choice given the fast cosine transformation implementations provided by FFTW.

Let's say we as for a grid with minimum of 0, a maximum 12 and a total of four points. This will result in the following values,

Grid Name	Values
`kGLEndpointGrid`	0,4,8,12
`kGLInteriorGrid`	1.5, 4.5, 7.5, 10.5
`kGLPeriodicGrid`	0, 3, 6, 9
`kGLChebyshevEndpointGrid`	12, 9, 3, 0
`kGLChebyshevInteriorGrid`	11.54, 8.29, 3.70, 0.45

2.3 Periodicity

Understanding periodicity in dimensions and how they related to the various transforms is fairly complicated.

It is important to understand how periodicity and domain length (L) affect the sample interval (Δ) in a dimension. Mathematically,

$$\Delta = \begin{cases} \frac{L}{N} & \text{periodic,} \\ \frac{L}{N-1} & \text{otherwise.} \end{cases} \tag{1}$$

where N is the number of points. This reason for this distinction is because in a periodic domain the start point at 0 is defined as being equal to the end point at 1 and therefore we don't need to locate at both the start point and the end point. This is in contrast to an aperiodic dimension where the value of a function may differ at the two end points. In a periodic dimension you essentially get another sample point for free.

This distinction because very significant when you make a transformation to another basis, like an exponential, sine or cosine basis which assumes a particular form of periodicity.

An exponential basis always assumes that the domain is chopped into N sample intervals and is thus directly compatible with the notion of a dimension being periodic. With the different types of sine and cosine basis, however, this isn't so simple. In particular, the DCT-I cosine transform assumes that the domain is chopped into $N-1$ sample intervals, while the DCT-II cosine transform assumes the domain is chopped into N sample intervals. For this reason it's important to define a dimension correctly based on the transform you're intending to use.

3 Variables

GLNumericalModelingKit defines three types of variables: scalars, functions and linear transformations.

3.1 Symmetry

When a variable is created a symmetry can be specified which will reduce the number of points that need to be stored. For example, if a function is defined as having hermitian symmetry with respect to some dimension, then negative points do not need to be created as it is assumed that the function can be recovered at negative points by applying the symmetry. The same is true of both the even and odd symmetries.

Symmetry Name	Definition
`kGLNoSymmetry`	$f(-x) = g(x)$
`kGLZeroSymmetry`	$f(-x) = f(x) = 0$
`kGLEvenSymmetry`	$f(-x) = f(x)$
`kGLOddSymmetry`	$f(-x) = -f(x)$

GLNumericalModelingKit is slightly more specific and actually defines a symmetry for the real part and the imaginary with respect to each dimension. Thus, a dimension with hermitian symmetry has even symmetry on the real part, and odd symmetry on the imaginary part.

4 Basis Transformations

Forward transformations take a function H_n defined in physical space on a grid x_n and transform it to a function h_k in spectral space on a grid f_k.

Basis	FFTW	Forward Transformation
`kGLExponentialBasis`	`FFTW_FORWARD`	$h_k = \sum_{n=0}^{N-1} H_n e^{-2\pi ikn/N}$
`kGLSineHalfShiftBasis`	`RODFT10`	$h_k = 2\sum_{n=0}^{N-1} H_n \sin\left[\pi(n+1/2)(k+1)/N\right]$
`kGLCosineHalfShiftBasis`	`REDFT10`	$h_k = 2\sum_{n=0}^{N-1} H_n \cos\left[\pi(n+1/2)k/N\right]$
`kGLSineBasis`	`RODFT00`	$h_k = 2\sum_{n=0}^{N-1} H_n \sin\left[\pi(n+1)(k+1)/(N+1)\right]$
`kGLCosineBasis`	`REDFT00`	$h_k = H_0 + (-1)^k H_{n-1} + 2\sum_{n=1}^{N-2} H_n \cos\left[\pi nk/(N-1)\right]$

Given function h_k in the frequency domain, the following transforms return function H_n in the spatial domain.

Basis	FFTW	Inverse Transformation
`kGLExponentialBasis`	`FFTW_INVERSE`	$H_n = \sum_{k=0}^{N-1} h_k e^{2\pi ikn/N}$
`kGLSineHalfShiftBasis`	`RODFT01`	$H_n = (-1)^n h_{N-1} + 2\sum_{k=0}^{N-2} h_k \sin\left[\pi(k+1)(n+1/2)/N\right]$
`kGLCosineHalfShiftBasis`	`REDFT01`	$H_n = h_0 + 2\sum_{k=1}^{N-1} h_k \cos\left[\pi(n+1/2)k/N\right]$
`kGLSineBasis`	`RODFT00`	$H_n = 2\sum_{k=0}^{N-1} h_k \sin\left[\pi(n+1)(k+1)/(N+1)\right]$
`kGLCosineBasis`	`REDFT00`	$H_n = h_0 + (-1)^n h_{k-1} + 2\sum_{k=1}^{N-2} h_k \cos\left[\pi nk/(N-1)\right]$

Each of these can be written in terms of a frequency and position, although we will ignore the two transformations that are not in the half-shift basis.

Basis	x_n	f_k	Forward Transformation
`kGLExponentialBasis`	$n\Delta$	$\frac{k}{N\Delta}$	$h_k = \sum_{n=0}^{N-1} H_n e^{-2\pi i f_k x_n}$
`kGLSineHalfShiftBasis`	$n\Delta + \frac{\Delta}{2}$	$\frac{k+1}{2N\Delta}$	$h_k = 2\sum_{n=0}^{N-1} H_n \sin\left[2\pi f_k x_n\right]$
`kGLCosineHalfShiftBasis`	$n\Delta + \frac{\Delta}{2}$	$\frac{k}{2N\Delta}$	$h_k = 2\sum_{n=0}^{N-1} H_n \cos\left[2\pi f_k x_n\right]$
`kGLSineBasis`	$n\Delta + \Delta$	$\frac{k+1}{2\Delta(N+1)}$	$h_k = 2\sum_{n=0}^{N-1} H_n \sin\left[2\pi f_k x_n\right]$
`kGLCosineBasis`	$n\Delta$	$\frac{k}{2\Delta(N-1)}$	$h_k = H_0 + (-1)^k H_{n-1} + 2\sum_{n=1}^{N-2} H_n \cos\left[2\pi f_k x_n\right]$

Basis	x_n	f_k	Inverse Transformation
`kGLExponentialBasis`	$n\Delta$	$\frac{k}{N\Delta}$	$H_n = \sum_{k=0}^{N-1} h_k e^{2\pi i f_k x_n}$
`kGLSineHalfShiftBasis`	$n\Delta + \frac{\Delta}{2}$	$\frac{k+1}{2N\Delta}$	$H_n = (-1)^n h_{N-1} + 2\sum_{k=0}^{N-2} h_k \sin\left[2\pi f_k x_n\right]$
`kGLCosineHalfShiftBasis`	$n\Delta + \frac{\Delta}{2}$	$\frac{k}{2N\Delta}$	$H_n = h_0 + 2\sum_{k=1}^{N-1} h_k \cos\left[2\pi f_k x_n\right]$
`kGLSineBasis`	$n\Delta + \Delta$	$\frac{k+1}{2\Delta(N+1)}$	$H_n = 2\sum_{k=0}^{N-1} h_k \sin\left[2\pi f_k x_n\right]$
`kGLCosineBasis`	$n\Delta$	$\frac{k}{2\Delta(N-1)}$	$H_n = h_0 + (-1)^n h_{n-1} + 2\sum_{k=1}^{N-2} h_k \cos\left[2\pi f_k x_n\right]$

One thing to notice about the sine and cosine transforms in the *half-shift basis* is that they have the same collocation points and that they're defined at the same frequencies. The only difference is that the sine basis doesn't have a zero frequency, and the cosine basis doesn't have the Nyquist frequency. One can translate between the two frequencies using that that $f_k^{\text{sine}} = f_{k+1}^{\text{cosine}}$, or simply $f_k^{\text{s}} = f_{k+1}^{\text{c}}$ for short.

In contrast, the sine and cosine basis transforms are not defined at the same frequencies

Basis	Forward Transformation
`kGLSineBasis`	$h_k = 2 \sum_{n=0}^{N-1} H_n \sin\left[\pi(n+1)(k+1)/(N+1)\right]$
`kGLCosineBasis`	$h_k = H_0 + (-1)^k H_{n-1} + 2 \sum_{n=1}^{N-2} H_n \cos\left[\pi n k/(N-1)\right]$

4.1 Chebyshev transformation

The Chebyshev polynomials are defined as,

$$T_k(x) = \cos\left(k \cos^{-1} x\right). \tag{2}$$

The normalization coefficients c_k are are found from,

$$\int_{-1}^{1} T_k^2(x) \frac{dx}{\sqrt{1-x^2}} = c_k \frac{\pi}{2} \tag{3}$$

and are,

$$c_k = \begin{cases} 2 & k = 0, \\ 1 & \text{otherwise.} \end{cases} \tag{4}$$

With this, we can write a function $u(x)$ in a Chebyshev basis as

$$u(x) = \sum_{k=0} \hat{u}_k T_k(x) \tag{5}$$

using that,

$$\hat{u}_k = \frac{2}{\pi c_k} \int_{-1}^{1} u(x) T_k(x) \frac{dx}{\sqrt{1-x^2}}. \tag{6}$$

According to Canuto equation 2.4.15, for the Gauss-Lobatto grid the discrete Chebyshev transformation is,

$$C_{kn} = \frac{2}{(N-1)c_n c_k} \cos \frac{\pi n k}{N-1} \tag{7}$$

where now the normalization coefficients are,

$$c_n = \begin{cases} 2 & n = 0, N-1 \\ 1 & \text{otherwise.} \end{cases} \tag{8}$$

Written out a bit more, this means that we can find the spectral component $\hat{u}_k$ with

$$\hat{u}_k = \frac{1}{(N-1)c_k} \left[u_0 + (-1)^k u_{N-1} + 2 \sum_{n=1}^{N-2} u_n \cos \frac{\pi n k}{N-1} \right], \tag{9}$$

which is nearly the fast cosine transform implemented in FFTW above. We are actually going to compute,

$$\hat{h}_k = \frac{1}{2(N-1)} \left[u_0 + (-1)^k u_{N-1} + 2 \sum_{n=1}^{N-2} u_n \cos \frac{\pi n k}{N-1} \right], \tag{10}$$

which means that $\hat{h}_k = \frac{c_k \hat{u}_k}{2}$. We want the inverse transform to be

$$u_n = \sum_{k=0}^{N-1} \hat{u}_k \cos \frac{\pi n k}{N-1}. \tag{11}$$

Written out,

$$u_n = \hat{u}_0 + (-1)^n \hat{u}_{N-1} + \sum_{k=1}^{N-2} \hat{u}_k \cos \frac{\pi n k}{N-1}. \tag{12}$$

or, if we use $\hat{h}_k$ instead,

$$u_n = \hat{h}_0 + (-1)^n \hat{h}_{N-1} + 2 \sum_{k=1}^{N-2} \hat{h}_k \cos \frac{\pi n k}{N-1}. \tag{13}$$

5 Fast Transforms

In addition to the discrete linear transformation (which typically requires n^2 operations), some transformations may have a *fast transform* algorithm defined. The fast algorithm is provides the same result as doing the slower transformation, but requires fewer operations. In particular, the Fourier, sine, cosine, and Chebyshev transformation all have corresponding fast transforms which require only $n \log n$ operations. Finite differencing operation can also be defined as a banded diagonal matrix, or as some memory local stencil which may be significantly faster.

GLNumericalModelingKit does not implement the fast algorithms directly, but instead uses external libraries such as FFTW.

5.1 FFTW transformations

5.1.1 Forward Transforms

- **Real-to-real** transforms can *only* act on variables that are purely real and *only* on dimensions in the kGLDeltaBasis. These transformations are the sine and cosine basis transformations.

- **Real-to-complex** transforms can *only* act on variables that are purely real and *only* on dimensions in the kGLDeltaBasis. The transformed dimensions will be in the kGLExponentialBasis. The last transformed dimension will have strictly positive frequencies, exploiting the underlying Hermitian symmetry where the real part is even symmetric and the imaginary part odd symmetric.
- **Complex-to-complex** transforms can *only* act on complex variables, although they may be purely real or otherwise. A complex-to-complex transformation will only transform dimensions in the kGLDeltaBasis to the kGLExponentialBasis. Furthermore, this transform will not act on a complex variable with existing Hermitian symmetry on other dimensions in an exponential basis to prevent confusion.

5.1.2 Inverse Transforms

- **Real-to-real** transforms can *only* act on variables that are purely real and *only* on dimensions in the sine and cosine basis, returning them to the kGLDeltaBasis.
- **Complex-to-real** transforms can *only* act on variables with a Hermitian symmetry defined on dimensions in the kGLExponentialBasis. The transformed dimensions will be in the kGLDeltaBasis. The last transformed dimension must have strictly positive frequencies.
- **Complex-to-complex** transforms can *only* act on complex variables, although they may be purely real or otherwise. A complex-to-complex transformation will only transform dimensions in the kGLExponentialBasis to the kGLDeltaBasis. Furthermore, this transform will not act on a complex variable with existing Hermitian symmetry on other dimensions in an exponential basis? Why not?

5.2 Wisdom

FFTW has acquires 'wisdom' about the optimal execution algorithm that should be saved in certain cases. We can and should do this on a per-machine basis in the Application Support folder.

5.3 Data Format

A separate concept to this is the *data format.* In GLNumericalModelingKit we current have three different data formats: purely real, split complex, and half complex. It's temping to support interleaved because this corresponds directly to the C99 imaginary number definition which may make for easy programming. The half complex format is defined to exist on a particular dimension. FFTW automatically puts it on the last dimension in the real-to-complex operations, but it can be on any (or multiple!) dimensions in the real-to-real transforms. We will *not* support half-complex format on multiple dimensions.

So now we need to consider the different transformations that are possible, and how we will actually perform these in practice.

- **real-to-real** In this case we'll be taking a real function and transforming it to a mixed sine/cosine basis. This is easy and we just use `fftw_plan_guru_r2r`.
- **real-to-exponential** In this case we'll be taking a real function and transforming it to an exponential basis. I believe we can just use `fftw_plan_guru_split_dft_r2c`. We need to check if the complex output really is reduced in size. It appears so.
- **real-to-mixed** There are actually two sub cases here. In one case, exactly *one* dimension will transform to an exponential basis. In this case we can can use `fftw_plan_guru_r2r` and end up with half-complex data formatting on that dimension. In the second case, two or more dimensions need to be transformed to an exponential basis. Here, we should be able to transform first with `fftw_plan_guru_r2r` set to stride out the untransformed dimensions. This is then followed by `fftw_plan_guru_split_dft_r2` set to stride out the real dimensions.

If I'm thinking of this correctly, we can always do a real-to-real transform first on the appropriate dimensions, and then

6 Differential Transformations

A differential operators acts on a variable and returns a new differentiated variable. The method of differentiation that is used can be set by choosing a basis for differentiation. For example, choosing kGLDeltaBasis will use finite differencing, whereas kGLExponentialBasis will do a Fourier transform and differentiate in wavenumber space.

A default differentiation basis can be selected for a given equation, but can be overridden on a variable by variable case. Internally, GLNumericalModelingKit uses the following logic to determine which differential operator to use.

1. User requests differentiation with a string such as 'xx' or 'laplacian'.
2. GLDifferentialOperatorPool looks for a cached version of the operator in the pool associated with the preferred differentiation basis.
3. If a cached version can't be found, GLDifferentialOperatorPool checks to see if it knows how to build the requested operator.

4. Otherwise it throws an exception.

An individual differential operator knows how to differentiate a specific set of dimensions—typically spatial dimensions such as x, y and z—which we denote as the *differentiable dimensions*. However, the operator may need to transform the variable to a different basis in order to actually carry out the differentiation, these are the *transformed dimensions*. Thus, so long as the variable can be transformed to the *transformed dimensions*, then the operator is capable of differentiating that variable.

More specifically then, when the GLDifferentialOperatorPool looks for a cached version of an operator for a particular variable it must

1. return a list of operators with that name associated with the preferred differentiation basis,
2. determine whether or not the variables dimensions can be transformed into the operators *transformed dimensions*.

This particular search algorithm allows the user to override internal instructions for building an operator, by simply caching the operator with that name.

A GLDifferentialOperatorPool is specific to a particular set of *differentiable dimensions* and *transformed dimensions*

Important note: The advantage to using the half-shift basis, aside from speed, is that the colocation points in the spatial domain are the same. This is *not* the case for the basic sine and cosine transform. The basic cosine transform requires the function be specified at the end points of the domain, whereas the sine transform does not. This effectively gives the sine transform two more points to work with and also therefore makes it difficult to define an obvious transformation between the two different basis.

6.1 Notation

For the following sections dealing with the transformation of functions, we treat the discrete functions as vectors in a function space. The components of the functions are represented as superscripted contravariant tensors, f^i, where as the vector basis functions are represented as $\bar{e}_j$. Summation is assumed when an index appears in an expression twice, once as a superscript and once as a subscript.

In our notation all indices are zero based in order to be consistent with C programming conventions and therefore run from 0 through $n-1$. Superscripted indices indicate the row number, and subscripted indices indicate the column number.

Some examples include a column vector,

$$v^i = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ \vdots \\ v_{n-1} \end{bmatrix} \tag{14}$$

the Kronecker delta,

$$\delta^i_j = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{15}$$

the sub-diagonal Kronecker delta,

$$\delta^i_{j+1} = \delta^{i-1}_j = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{16}$$

and the super-diagonal Kronecker delta,

$$\delta^i_{j-1} = \delta^{i+1}_j = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}. \tag{17}$$

A general sub-diagonal matrix can be represented as,

$$a^i \delta^i_{j+1} = a^{i-1} \delta^{i-1}_j = \begin{bmatrix} 0 & 0 & 0 & 0 \\ a_1 & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 \\ 0 & 0 & a_{n-1} & 0 \end{bmatrix} \tag{18}$$

where the index i is not summed because it does not appear as both a superscript and a subscript. A general super-diagonal matrix can similarly be represented as,

$$c^i \delta^i_{j-1} = c^{i+1} \delta^{i+1}_j = \begin{bmatrix} 0 & c_0 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & c_{n-2} \\ 0 & 0 & 0 & 0 \end{bmatrix}. \tag{19}$$

6.2 Implementation

Designing the GLDifferentialOperator class is actually quite tricky. On the one hand, a differential operator is obviously a unary operator—it takes a single variable and returns a single differentiated variable. On the other hand, differential operators are variables—wavenumber vectors for spectral differentiation or sparse differentiation matrices for finite differencing.

To get the best of both worlds, the GLDifferentialOperator class is a subclass of GLVariableOperation that has a GLVariable property to store the differentiation matrix.

The role of the GLDifferentialOperatorPool is slightly expanded. The pool is now able to store and return differential operators, as well as differential operation matrices. When a differential matrix is set, the *toBasis* must also be set (the *fromBasis* is already defined at the operator pool level). This requires the mathematician be much more aware of what they are doing, which I think is good. When a differential operator is requested (of the same name as the matrix), then a new operation is created with the right matrix and toBasis.

The best shortcut for differentiation would look like this,

-[GLVariable differentiateWithOperator: (NSString *) opName byTransformingToBasis: (NSArray *) orderedBasis];

When do I initialize the operator? It's going to be a slightly different prototype. We init

6.3 Differentiation Matrices

We can determine the differentiation operator using the definition of the inverse transforms.

6.3.1 kGLExponentialBasis

$$\partial_x H_n = \partial_x \left(\sum_{k=0}^{N-1} h_k e^{2\pi i f_k x_n} \right) \tag{20}$$

$$= \sum_{k=0}^{N-1} h_k (2\pi i f_k) e^{2\pi i f_k x_n} \tag{21}$$

This means that the differentiation operator D_x^{e} is defined as,

$$D_x^{\mathrm{e}} \equiv \left(2\pi i f_k^{\mathrm{e}}\right) \delta_m^k. \tag{22}$$

and take a function in the exponential basis and transforms it to a differentiated function in the exponential basis.

This linear transformation can be represented by a diagonal matrix.

6.3.2 kGLSineHalfShiftBasis

We denote the k frequency component in the sine basis by h_k^{s}.

$$\partial_x H_n = \partial_x \left((-1)^n h_{N-1} + 2 \sum_{k=0}^{N-2} h_k^{\mathrm{s}} \sin\left[2\pi f_k^{\mathrm{s}} x_n\right] \right) \tag{23}$$

$$= 2 \sum_{k=0}^{N-2} h_k^{\mathrm{s}} \left(2\pi f_k^{\mathrm{s}}\right) \cos\left[2\pi f_k^{\mathrm{s}} x_n\right] \tag{24}$$

Now let $k = m - 1$, so that

$$\partial_x H_n = 2 \sum_{m=1}^{N-1} h_k^{\mathrm{s}} \left(2\pi f_k^{\mathrm{s}}\right) \cos\left[2\pi f_{m-1}^{\mathrm{s}} x_n\right] \tag{25}$$

$$= 2 \sum_{m=1}^{N-1} h_k^{\mathrm{s}} \left(2\pi f_k^{\mathrm{s}}\right) \cos\left[2\pi f_m^{\mathrm{c}} x_n\right] \tag{26}$$

where we used that $f_k^{\mathrm{s}} = f_{k+1}^{\mathrm{c}}$ in the last step. Let's define g_m^{c} to be the coefficient of the sum. In that case,

$$g_m^{\mathrm{c}} = \delta_{m-1}^k h_k^{\mathrm{s}} \left(2\pi f_k^{\mathrm{s}}\right). \tag{27}$$

This means that the differentiation operator D_x^{s} is defined as,

$$D_x^{\mathrm{s}} \equiv \left(2\pi f_k^{\mathrm{s}}\right) \delta_{m-1}^k. \tag{28}$$

This operator takes a function defined in a sine basis, and transforms it to a differentiated function in the cosine basis.

This linear transformation can be represented by a subdiagonal matrix.

6.3.3 kGLCosineHalfShiftBasis

We denote the k frequency component in the cosine basis by $h_k^{\rm c}$.

$$\partial_x H_n = \partial_x \left(h_0 + 2 \sum_{k=1}^{N-1} h_k^{\rm c} \cos\left[2\pi f_k^{\rm c} x_n\right] \right) \tag{29}$$

$$= 2 \sum_{k=1}^{N-1} h_k^{\rm c} \left(-2\pi f_k^{\rm c}\right) \sin\left[2\pi f_k^{\rm c} x_n\right] \tag{30}$$

Now let $k = m + 1$, so that

$$\partial_x H_n = 2 \sum_{m=0}^{N-2} h_k^{\rm c} \left(-2\pi f_k^{\rm c}\right) \sin\left[2\pi f_{m+1}^{\rm c} x_n\right] \tag{31}$$

$$= 2 \sum_{m=0}^{N-2} h_k^{\rm c} \left(-2\pi f_k^{\rm c}\right) \sin\left[2\pi f_m^{\rm s} x_n\right] \tag{32}$$

where we used that $f_k^{\rm s} = f_{k+1}^{\rm c}$ in the last step. Let's define $g_m^{\rm s}$ to be the coefficient of the sum. In that case,

$$g_m^{\rm s} = \delta_{m+1}^k h_k^{\rm c} \left(-2\pi f_k^{\rm c}\right). \tag{33}$$

This means that the differentiation operator $D_x^{\rm c}$ is defined as,

$$D_x^{\rm c} \equiv \left(-2\pi f_k^{\rm c}\right) \delta_{m+1}^k. \tag{34}$$

This operator takes a function defined in a cosine basis, and transforms it to a differentiated function in the sine basis.

This linear transformation can be represented by a superdiagonal matrix.

6.3.4 kGLChebyshevBasis

We have the series,

$$u_n = \sum_{k=0}^{N-1} \hat{u}_k T_k(x_n) \tag{35}$$

and we want to find its derivative. We can show that the following relationship is true,

$$2T_k = \frac{c_k}{k+1} T'_{k+1} - \frac{1}{k-1} T'_{k-1} \tag{36}$$

where the T'_k polynomials are zero for $k < 1$ and $k > N-1$. If we denote the coefficients for the derivative of the function as $\hat{u}_k^{(1)}$, then by definition

$$\begin{aligned}\sum_{k=0}^{N-1} \hat{u}_k T'_k(x_n) =& \sum_{k=0}^{N-1} \hat{u}_k^{(1)} T_k(x_n) && (37)\\ =&\frac{1}{2}\sum_{k=0}^{N-1} \hat{u}_k^{(1)} \left(\frac{c_k}{k+1} T'_{k+1}(x_n) - \frac{1}{k-1} T'_{k-1}(x_n) \right) && (38)\\ =&\frac{1}{2}\sum_{k=0}^{N} \hat{u}_{k-1}^{(1)} \frac{c_{k-1}}{k} T'_k(x_n) - \sum_{k=-1}^{N-2} \hat{u}_{k+1}^{(1)} \frac{1}{k} T'_k(x_n) && (39)\\ =&\frac{1}{2}\sum_{k=0}^{N-1} \frac{1}{k} \left(c_{k-1}\hat{u}_{k-1}^{(1)} - \hat{u}_{k+1}^{(1)} \right) T'_k(x_n) && (40)\end{aligned}$$

and we conclude that,

$$2k\hat{u}_k = c_{k-1}\hat{u}_{k-1}^{(1)} - \hat{u}_{k+1}^{(1)}. \tag{41}$$

The appropriate operator for computing the derivative is thus most efficiently defined as the recursion,

$$c_k\hat{u}_k^{(1)} = 2(k+1)\hat{u}_{k+1} + \hat{u}_{k+2}^{(1)} \tag{42}$$

To properly scale this, a factor of $\frac{L}{2}$ needs to be out front each derivative term. So that,

$$c_k\hat{u}_k^{(1)} = \frac{4}{L}(k+1)\hat{u}_{k+1} + \hat{u}_{k+2}^{(1)}. \tag{43}$$

But we can do even better, and write this in terms of what we're actually computing, $\hat{h}_k = \frac{c_k \hat{u}_k}{2}$, so that,

$$\hat{h}_k^{(1)} = \frac{4}{L}(k+1)\hat{h}_{k+1} + \hat{h}_{k+2}^{(1)}. \tag{44}$$

Note that we can drop the c_k factor on the terms on the right-hand-side because they're all greater than 0 where $c_k = 1$. Let's write out a few of these to get a sense of what they look like.

$$\begin{aligned}\hat{h}_{N-1}^{(1)} =&0 && (45)\\ \hat{h}_{N-2}^{(1)} =&\frac{4}{L}\Big((N-1)\hat{h}_{N-1} \Big) && (46)\\ \hat{h}_{N-3}^{(1)} =&\frac{4}{L}\Big((N-2)\hat{h}_{N-2} \Big) && (47)\\ \hat{h}_{N-4}^{(1)} =&\frac{4}{L}\Big((N-3)\hat{h}_{N-3} + (N-1)\hat{h}_{N-1} \Big) && (48)\\ \hat{h}_{N-5}^{(1)} =&\frac{4}{L}\Big((N-4)\hat{h}_{N-4} + (N-2)\hat{h}_{N-2} \Big) && (49)\end{aligned}$$

We can write this as a transformation matrix,

$$\hat{h}_k^{(1)} = \frac{4}{L} \sum_{p=k+1, p+k odd} p\hat{h}_p \tag{50}$$

The obvious thing to do is to set the frequencies to $f_k = \frac{2k}{L}$. In this case,

$$\hat{h}_k^{(1)} = \sum_{p=k+1, p+k odd} 2 f_p \hat{h}_p \tag{51}$$

6.3.5 Other

Taking derivatives of the function in the different basis,

$$h'_k = \sum_{n=0}^{N-1} H_n \left(\frac{2\pi i n}{N} \right) e^{2\pi i k n/N} \tag{52}$$

a kGLSineBasis,

$$\begin{aligned} h'_k &= 2 \sum_{n=0}^{N-1} H_n \left(\frac{\pi(n+1)}{N+1} \right) \cos\left[\pi(n+1)(k+1)/(N+1)\right] && (53) \\ &= 2 \sum_{n=1}^{N-1} H_{n-1} \left(\frac{\pi n}{N} \right) \cos\left[\pi n(k+1/2)/N\right] && (54) \end{aligned}$$

a kGLSineHalfShiftBasis,

$$\begin{aligned} h_k &= (-1)^k H_{N-1} + 2 \sum_{n=0}^{N-2} H_n \sin\left[\pi(n+1)(k+1/2)/N\right] && (55) \\ h'_k &= 2 \sum_{n=0}^{N-2} H_n \left(\frac{\pi(n+1)}{N} \right) \cos\left[\pi(n+1)(k+1/2)/N\right] && (56) \\ &= 2 \sum_{n=1}^{N-1} H_{n-1} \left(\frac{\pi n}{N} \right) \cos\left[\pi n(k+1/2)/N\right] && (57) \end{aligned}$$

and a cosine basis,

$$h_k = H_0 + 2\sum_{n=1}^{N-1} H_n \cos\left[\pi n(k+1/2)/N\right] \tag{58}$$

$$h'_k = -2\sum_{n=1}^{N-1} H_n \left(\frac{\pi n}{N}\right) \sin\left[\pi n(k+1/2)/N\right] \tag{59}$$

$$= -2\sum_{n=0}^{N-2} H_{n+1} \left(\frac{\pi(n+1)}{N}\right) \sin\left[\pi(n+1)(k+1/2)/N\right]. \tag{60}$$

6.4 Example

Consider two different variations of the cosine transform. In the first case, DCT-I, the cosine function is even around $n = 0$ and $n = N - 1$ where $n = 0..N - 1$. This means that if we define $x_n = n * \Delta$ where $\Delta = 1/(N - 1)$, then $3\cos(10 \cdot 2\pi)$ should give an amplitude of 3.

Alternatively, DCT-II, the cosine function is even around $n = -1/2$ and $n = N - 1/2$. This means that if we define $x_n = (n + 1/2) * \Delta$ where $\Delta = 1/N$

7 Linear Solvers

We need to be able to solve the linear matrix equation,

$$M\vec{x} = \vec{b} \tag{61}$$

for $\vec{x}$ given M and $\vec{b}$.

Depending on whether the matrix is the identity, diagonal, tridiagonal, or dense, we need to choose a different solution algorithm. The matrix may be any one of these types in *each* dimension, making the problem far more tricky. The current need is such that we only expect one dimension to be tridiagonal or dense, effectively meaning that we only need to solve the one-dimensional non-trivial problem. Specifically, we can solve the following types of equations,

1. All matrix dimensions are the identity.
2. All matrix dimension are the identity or diagonal.
3. One matrix dimension is tridiagonal, all others are the identity or diagonal.
4. One matrix dimension is dense, all others are the identity or diagonal.

In the first case we simply return $\vec{b}$. In the second case, we simply have to divide $\vec{b}$ elements by the elements of M (some repeated). In the third and fourth case, we need to loop over the elements in the diagonal dimensions, solving the tridiagonal or dense matrix problem in the other dimension.

7.1 Algorithm

Assuming we have only one non-diagonal dimension, the linear solver problem can be thought of as a number of independent one-dimensional linear problems. We need to find the stride between equations and total number of equations, and then just repeat the solver. In addition, a trivial dimension (one with the identity) will just repeat an existing solution across that dimension.

8 Linear Transformations

Now it's time to implement finite differencing and more general linear transformations. These require generalizations of the concepts above.

A linear transformation is a tensor that transforms a vector from one basis to another. The Fourier transforms are simply linear transformations and we could, in fact, write down an $n \times n$ transformation matrix instead of using the FFT. A sufficient generalization then for now is to consider linear transformations that transformation from one set of dimension to another set the same length. In general this doesn't have to be true, but it's a good start.

Note there is still a distinction to be made between transformations to and from a kGLDeltaBasis and those transformations that go to and from a trigonometric basis. The reason is that the definition of differentiation is quite different, as the basis vectors need to be differentiated in the trigonometric basis.

Okay, so backing up a bit, FFTs and matrix operations are both types of linear transformations. In the former case we don't need to store an values in a variable, but only because our library that we're calling is doing this for us. In the latter case, we may need to store a complete matrix, or perhaps just a sparse subset of this.

Now, the differential operators are also interesting. They too are linear transformations that potentially involve changing the basis vectors, or not. So that part is not different. The only part that is different, is that follow Liebnitz rule. I think that's it. In the case of differentiating in a trigonometric basis, you simply multiply by a diagonal matrix. Differentiating in a delta basis may be multiplying by a tridiagonal matrix.

9 Storing linear transformations

Up until now, we have defined our variables as vectors with respect to some basis. This is, definitely, a concept separate from defining a linear transformation. On the other hand, a linear transformation is simply a bunch of these variables defined as a coefficient for each basis vector. In all cases, a linear transformation has a from-basis, and a to-basis. So I think that it's fair to define a linear transformation as a subclass of GLVariable that has slightly more structure.

Technically then, I suppose we might define a GLScalar, GLVector, GLMatrix—all of which are subclasses of GLTensor.

The puzzling thing about all of this is that my GLVariables, I have thought of as scalar functions defined on their dimensions. That *is* correct. There's nothing wrong with that thinking. And yeah, when I take derivatives, I'm taking derivatives of scalars. So my basis is a function basis. And that's why I shouldn't be thinking in terms of eigenvectors so much as eigenfunctions.

So let's go back. One linear transformation I'm interested in might transform a function from an existing basis to an eigenbasis. Another linear transformation might transform a function to its derivative. The concept of Linear transformation is fine here, and I see no reason to further mess with it.

Okay, now the issue becomes how to create an appropriate class for this. A linear transformation goes from one basis to another (maybe the same) basis. That must be part of the definition. It also has some storage needs for how to do this, maybe nothing, but maybe a lot. The question now is how to specify the storage needs—and just reuse most of what we've done already.

One key thing is that we've been defining our differential operators as straight variables. Of course, they should be diagonal linear transformations. This wouldn't change too much about how we do things. And it would formalize the equivalence between finite differencing and solving a matrix spectrally.

One could specify the columns, rows, or diagonals of this transform. So, specifying a diagonal, would be taking vector v_i and turning it into the matrix transformation $v_l \delta^l_i$. The vector could easily be placed off-diagonal, $v_l \delta^l_{i-1}$.

Assuming that we're transforming from a dimension of length n to one of length n, the storage requirements are such:

Type	Form	Storage
kGLIdentityMatrixFormat	δ^i_j	0
kGLDenseMatrixFormat	m^i_j	n^2
kGLDiagonalMatrixFormat	$b^i\delta^i_j$	n
kGLSubdiagonalMatrixFormat	$a^i\delta^i_{j+1}$	n
kGLSuperdiagonalMatrixFormat	$c^i\delta^i_{j-1}$	n
kGLTridiagonalMatrixFormat	$a^i\delta^i_{j+1} + b^i\delta^i_j + c^i\delta^i_{j-1}$	$3n$

In the above table, the components a_0 and c_{n-1} are *always* zero.

Note that v^j is a column vector where the index j iterates through the rows. A transformation matrix is represented by R^i_j where i indexes the matrix row and j indexes the matrix column.

For reference, the diagonal format takes the following form,

$$\begin{bmatrix} b_0 & c_0 & & & 0 \\ a_1 & b_1 & c_1 & & \\ & a_2 & b_2 & \ddots & \\ & & \ddots & \ddots & c_{n-2} \\ 0 & & & a_{n-1} & b_{n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix} \tag{62}$$

We also need to side how rows and columns get stored. In a typical matrix multiplication, one would multiply the rows of a matrix by the column vector.

10 Class Hierarchy

It's proper to think of GLVariable objects as discrete functions. They are functions defined on some manifold with respect to some dimensions. We also have some GLVariables that are dimensionless that we've had to accommodate. Now we have Linear Transformations which are defined with respect to two sets of dimensions—those being transformed from, and those being transformed to. These linear transformations also happen to be operators that act on functions. This duality explains why it was a challenge to determine the best way to define differential operators (as a subclass of GLVariable, or a subclass of GLVariableOperation?).