Intro to Pedalus

What is it?

- Python library for solving PDEs
  - IVPs (simulations)
  - EVPs (stability, wave propagation, topology)
  - BVPs (usually for ICs)
    - + Linear
    - + Non-linear
- A "domain-specific language" for PDEs
  - ↳ A way of typing the eqns you want in & getting a solution out
- A philosophy about numerical work, tools, & the connection between physics, mathematics, and
- A welcoming community of students, postdocs, faculty serving as users, testers, developers

PHILOSOPHY

- RATIO OF SCALES

$$Re \equiv \frac{UL}{\nu} \sim \frac{\tau_{VISC}}{\tau_{ADV}}$$

$$N_6^3 \sim Re^{9/4} \Rightarrow \text{BIG SEP. OF SCALES}$$

DNS → DEFINE

NOT POSSIBLE $\Rightarrow$ FOR PRACTICAL

GEO/ASTRO : $Re \gtrsim 10^8$

$N \simeq (5 \times 10^6)^3$ CELLS...

~60 YRS VIA MOORE'S LAW

~3 PW (10x NIF)

- REDUCED MODELS : ASYMPTOTICS → NEW EQNS

$\Omega \to$ BIG → KEITH

$\beta \to$ SMALL → GEOFF V.

$h \to$ SMALL → SHALLOW H_2O

- GENERALIZATIONS : NON-NEWTONIAN FLUIDS

ACTIVE MATTER

MHD

↳ NEW EQNS.

$\Rightarrow$ MANY EQNS → MODELS

WE PREFER MODELS THAT CAN BE ENCODED IN PDEs

STORY IS MOSTLY ABOUT LINEAR ALGEBRA

WHY?

TOO MANY DOMAIN SPECIFIC CODES FOR CLOSELY RELATED PROBLEMS

$$\begin{cases} \partial_t u + u \cdot \nabla u = -\nabla p + \gamma \nabla u \\ \nabla \cdot u \end{cases}$$

$\partial_t \rho + \nabla \cdot (\rho u) = 0$ FULLY COMPRESSIBLE

$\nabla \cdot (\rho u) = 0$ ANELASTIC

$\partial_t T + u \cdot \nabla T = \chi \nabla T$ → CONVECTION

OUR AIM: SIMPLE GEOMETRIES
EXTREME ACCURACY
FLEXIBILITY

BASIC IDEA

FOR CONTINUUM PROBLEMS, E.G. HYDRODYNAMICS

$$\partial_t \vec{u} + \vec{u} \cdot \nabla \vec{u} = -\vec{\nabla} p + \nu \vec{\nabla}^2 \vec{u}$$

$$\vec{\nabla} \cdot \vec{u} = 0$$

NOTATION

$\vec{v}$, $|u\rangle$ → VECTOR

$\vec{\vec{A}}$, A_{ij} → MATRIX

SOLUTIONS $\vec{v}(t, \vec{x})$ LIVE IN ∞-DIM FUNCTION SPACE

COMPUTERS ARE DEFINITELY NOT INFINITE, SO ALL SIMULATIONS BOIL DOWN TO SOME SORT OF PROJECTION ONTO A FINITE-DIMENSIONAL VECTOR SPACE, USUALLY WITH AN INNER PRODUCT

$$t_n = n \Delta t$$

$$\vec{x}_i = i \Delta \vec{x}$$

$$\vec{v}(t, \vec{x}) \approx \underbrace{\vec{v}_N(t_n, \vec{x}_i)}$$

SOME SORT OF N-DIMENSIONAL APPROXIMATION.

"DISCRETE SOLUTION"

LINEAR ALGEBRA

INNER PRODUCT CONVECTIONS

$\langle a|b\rangle$ $\qquad (a,b) = (b,a) \Rightarrow a,b \in \mathbb{R}^n$

$= \overline{\langle b|a\rangle}$

$a,b \in \mathbb{C}^n$

$\downarrow$

$\langle \alpha a|b\rangle = \alpha^* \langle a|b\rangle$ $\qquad$ "PHYSICS" CONVENTION

$\langle a|\alpha b\rangle = \alpha \langle a|b\rangle$

$(\alpha a, b) = \alpha (a,b)$ $\qquad$ "MATHEMATICS" CONVENTION

$(a, \alpha b) = \alpha^* (a,b)$

DEFINE L_2-NORM

$$\|f\|_2 \equiv \langle f|f\rangle^{1/2} = \left(\int_{-\infty}^{\infty} w(x) f^*(x) f(x)\,dx\right)^{1/2}$$

THIS CORRESPONDS TO THE EXPECTATION VALUE IN QM $\langle f \rangle$, BUT WE WILL RESERVE THIS NOTATION FOR MEANS,

$$\langle f \rangle = \frac{1}{L_x}\int_0^{L_x} f(x)\,dx$$

L_2 NORM GIVES US A DEFINITION OF A WELL-POSED PROBLEM

$$\|u(x,t)\| \le K e^{\alpha t}\|u(x,0)\| \quad \forall\ u_0 \in L_2$$

FOR CONSTANTS K, α

α COULD BE $\alpha < 0 \rightarrow$ DECAYING SOLN

$\alpha = 0 \rightarrow$ CONSTANT

$\alpha > 0 \rightarrow$ GROWING

GENERALIZED

$$\langle f|g\rangle = \int_a^b w(x) f^*(x) g(x) dx$$

$w(x)$ → WEIGHT FUNCTION

LET'S CONSIDER HEAT EQN

$$\partial_t T - \alpha \partial_x^2 T = 0$$

IS IT WELL POSED? USE FOURIER BASIS

$$\partial_t \hat{T}_k + \alpha k^2 \hat{T}_k = 0 \quad \Rightarrow \quad \hat{T}_k = \hat{T}_k(0) e^{-\alpha k^2 t}$$

$$T(t,x) = \int e^{+ikx} \hat{T}(t,k)\, dk$$

$$\hat{T}(t,k) = \frac{1}{2\pi} \int_{-\infty}^{\infty} T(t,x)\, e^{-ikx}\, dx$$

$$\| T(t,x) \| = \| \hat{T}(t,k) \| \qquad \text{PARSEVAL}$$

$$\| \hat{T}(t,k) \| = \int_{-\infty}^{\infty} |\hat{T}(t,k)|^2\, dk$$

$$= \int_{-\infty}^{\infty} \left| e^{-\alpha k^2 t}\, \hat{T}(0,k) \right|^2 dk$$

$$\leq \underbrace{\sup_{-\infty < k < +\infty} \left| e^{-\alpha k^2 t} \right|^2}\; \| T(0,k) \|^2$$

Pseudospectral methods are also a great starting place for thinking about numerical analysis for physics students, as it allows us to re-use many ideas familiar to us from quantum mechanics. In quantum mechanics, the space of possible states is a Hilbert space and Hermitian operators represent observable quantities. The possible values of a given observable correspond to the eigenvalues of the corresponding operator; for this reason, quantum mechanics traditionally focused on Hermitian operators which have real eigenvalues. One of the first lessons one learns about Hermitian operators in quantum mechanics is that their eigenvectors form a complete basis for the Hilbert space. This idea of a complete basis forms a core part of the mathematics of quantum mechanics, and it is also crucial for our understanding of PDEs. The idea is this: there is a *non-unique* set of vectors that can span the space of solutions to a problem.

Of course, there is a key difference in quantum mechanics between *discrete* and *continuous* problems. These come from different *physical quantities*: the energies of a harmonic oscillator are discrete, while the position and momentum of a free particle are contiunous[9]. This is an inherent physical property of these systems. However, here we are always going to be interested in *continuous* functions, but we will construct *discrete* approximations to them. When we discretize a PDE (or its solution), we must first **choose a basis**. Next, we **truncate that basis** to a finite number of elements.

[9] you may remember that the distinction between discrete and continuous is *because* one is bound and the other not.

Let's consider a simple quantum state,

$$|\psi\rangle = \sum_i \alpha_i |E_i\rangle . \tag{1.17}$$

We say that we've expanded the state $|\psi\rangle$ in terms of a set of *basis vectors* $|E_n\rangle$. We can find the **coefficients** by projecting against the basis using the **inner product**,

$$\alpha_i = \langle E_i|\psi\rangle . \tag{1.18}$$

The power of this notation is that we have not specified what the inner product *actually is*, simply that it exists. As implied here by the notation, the set $|E_i\rangle$ is the **energy basis** which is itself the solution to the time independent Schrödinger equation,

$$H|E_n\rangle = E_n|E_n\rangle . \tag{1.19}$$

H is the Hamiltonian, a Hermitian operator that represents the total energy.[10] A Hermitian operator has two extremely important properties: its eigenvalues are *real* and as mentioned above, its eigenvectors form a complete, orthogonal set of vectors. **Orthogonality** can be written as

[10] the Hamiltonian is also the distinguished observable that encodes the time evolution of the states.

$$\langle a|b\rangle = \delta_{a,b}, \tag{1.20}$$

where $\delta_{a,b}$ is the Kronecker delta,

$$\delta_{i,j} = \begin{cases} 1, & \text{if } i = j. \\ 0, & \text{otherwise.} \end{cases} \tag{1.21}$$

When solving PDEs numerically, we are going to *project* an approximation to the solution function $f(x)$ onto a vector space of **orthogonal polynomials**.

1.5 The Pseudospectral Method

To approximate a function $f(x)$, we write it as an expansion

$$f(x) \simeq f_N(x) \equiv \sum_{i=0}^{N-1} \hat{f}_i \phi_i(x) \tag{1.22}$$

in terms of basis functions $\phi_i(x)$. $f_N(x)$ is the polynomial approximation to our function $f(x)$. We can write this in braket notation as

$$f(x) \simeq f_N(x) \equiv \sum_{i=0}^{N-1} \hat{f}_i \, |\phi_i(x)\rangle \, . \tag{1.23}$$

The most important thing about this formulation is that the set $\hat{f}_i$ are *constants* in x! That means that if we want to compute $g(x) = \partial_x f(x)$, we only need to take the derivatives of the basis functions $\phi_i(x)$.

The basis functions we use satisfy a series of properties that make them useful to do computations with:

- they are complete, orthogonal bases in the domain of interest.
- They are easy to differentiate
- There is a reasonably efficient transform between the coefficients and the grid points.

The simplest basis functions are the **Fourier polynomials**. The Fourier polynomials are defined by

$$\phi_n = e^{i2\pi nx/L}, \tag{1.24}$$

where L is the length of the domain we are considering. Typically, we clean up the notation by writing $k_n = 2\pi n/L$, but equation (1.24) shows the explicit inclusion of the integer n.

Ther derivatives of Fourier polynomials are quite simple,

$$\frac{\partial \phi_n(x)}{\partial x} = ik_n\phi_n(x). \tag{1.25}$$

There's two things here to note. First, the derivative is simply multiplication by ik_n. Second, as a (very important!) consequence, the derivative of the nth Fourier basis function involves only the nth basis function itself. So why do we not use Fourier bases all the time? In fact, we do use them in nearly every calculation. However, the main weakness of the Fourier basis is that it only works for **periodic** domains. There are very many fluid systems you might want to solve that have boundaries of some sort or another. Thus, we have to consider other families of basis functions.

1.6 Chebyshev Classical Orthogonal Polynomials

Dedalus makes use of many of the classical orthogonal polynomials, but we will only focus on one of them, the Chebyshev polynomials. In fact ''the'' Chebyshev polynomials is a bit of a misnomer. There are four sets of functions in common use that go by the name of Chebyshev. We will use only two of them,

1. Chebyshev T
2. Chebyshev U

These are our workhorse polynomials in Cartesian domains. When we consider curvilinear domains, we use other families, most importantly the Jacobi polynomials. For much more information about the classical orthogonal polynomails, see Chapter 18 of the NIST Digital Library of Mathematical Functions[11].

Let's consider the first of these[12].

[11] T. H. Koornwinder, R. F. Swarttouw, R. Koekoek, and R. Wong, *Chapter 18: Orthogonal Polynomials*, `https://dlmf.nist.gov/18`..

[12] sometimes also called ''Cheybyshev polynomials of the first kind''; U are the ''second kind''.

The first seven Chebyshev T polynomials can be written

$$T_0(x) = 1 \tag{1.26}$$
$$T_1(x) = x \tag{1.27}$$
$$T_2(x) = 2x^2 - 1 \tag{1.28}$$
$$T_3(x) = 4x^3 - 3x \tag{1.29}$$
$$T_4(x) = 8x^4 - 8x^2 + 1 \tag{1.30}$$
$$T_5(x) = 16x^5 - 20x^3 + 5x \tag{1.31}$$
$$T_6(x) = 32x^6 - 48x^4 + 18x^2 - 1. \tag{1.32}$$

There is so much to say about Chebyshev polynomials, but we will limit ourselves to the most important things for numerical simulation.

Again, let's consider a truncated series expansion $f_N(x)$ of a function $f(x)$ that we would like to approximate. This time, we'll expand it in T_n,

$$f_N(x) = \sum_{n=0}^{N-1} \hat{f}_n T_n(x). \tag{1.33}$$

Let's consider the derivatives of this function, $g(x) = \partial_x f(x)$. Again, we have only to compute the derivatives of $T_n(x)$:

$$g_N(x) = \sum_{n=0}^{N-1} \hat{f}_n \frac{\partial T_n(x)}{\partial x}. \tag{1.34}$$

Let's look at the fisrt few derivatives of T polynomials,

$$\frac{\partial T_0(x)}{\partial x} = 0 \tag{1.35}$$
$$\frac{\partial T_1(x)}{\partial x} = 1 \qquad = T_0(x) \tag{1.36}$$
$$\frac{\partial T_2(x)}{\partial x} = 4x \qquad = 4T_1(x) \tag{1.37}$$
$$\frac{\partial T_3(x)}{\partial x} = 12x^2 - 3 \qquad = 6T_2(x) + 3T_0(x) \tag{1.38}$$
$$\frac{\partial T_4(x)}{\partial x} = 32x^3 - 16x \qquad = 8T_3(x) + 8T_1(x) \tag{1.39}$$
$$\frac{\partial T_5(x)}{\partial x} = 80x^4 - 60x^2 + 5 \qquad = 10T_4(x) + 10T_2(x) + 5T_0(x) \tag{1.40}$$
$$\frac{\partial T_6(x)}{\partial x} = 192x^5 - 192x^3 + 36x = 12T_5(x) + 12T_3(x) + 12T_1(x). \tag{1.41}$$

The last column of equations (1.36- (1.41)) show the *projection* of the derivatives back onto the $T_n(x)$. There are a few interesting things to note here. The derivatives are not as simple as they are in the Fourier case. If we return to thinking about our polynomial approximation as a *vector* of coefficients, $\hat{f}_n \to |f\rangle$, we can think about how $g(x) = \partial_x f(x)$ can be represented in the T_n basis, $\hat{g}_n \to |g\rangle$. Let's work this out:

$$g(x) = \partial_x f(x) \tag{1.42}$$

$$|g\rangle = \sum_n \hat{f}_n \partial_x |T_n\rangle . \tag{1.43}$$

Now, we can ask what the coefficients of $g_N(x)$ are,

$$\hat{g}_n = \langle T_n | g \rangle = \sum_i \hat{f}_i \langle T_n | \partial_x T_i \rangle . \tag{1.44}$$

If we look back at equations (1.36)- (1.41), we note that the the derivatives of $T_n(x)$ are *not orthogonal* to $T_n(x)$ themselves, so we cannot simply diagonalize the braket in this equation. If we return to the idea that ∂_x is a linear operator, when we discretize it in a particular basis, we expect to get a *matrix*, just like in quantum mechanics. However, just like in quantum, we must be careful to distinguish the *operator* from the *matrix*: the latter depends on a the choice of basis! Equation (1.44) shows us that because the coefficients $\hat{f}_n$ are independent of x, the derivative of our function depends only on the matrix given by

$$D_{i,j} = \langle T_i | \partial_x T_j \rangle . \tag{1.45}$$

Exercise 1.2. Show that one can generalize equations (1.36)- (1.41) to arrive at

$$D_{i,j} = \frac{2j((j-i) mod 2)}{1 + \delta_{i,0}} [i < j], \tag{1.46}$$

where $[i < j]$ evaluates to 1 if the argument is true and zero otherwise.

Figure 1.1 shows the $D_{i,j}$ matrix and some simple python code to generate it. With this matrix in hand, we can numerically calculate derivatives for functions approximated using Chebyshev T polynomials simply by doing matrix multiplication on the vector of coefficients $|f\rangle$ describing the discretized function!

However, Dedalus does not do this. And you shouldn't either. Why? Look at figure 1.1 again. Note that it is an upper triangular matrix. We call such a matrix ''dense'', because 25% of its entries are non-zero. Remember, our goal here is not to calculate derivatives, in which case we'd only need to do matrix multiplication. Our goal here is to **solve differential equations**, meaning we know the derivative and we want the function. Written in math,

$$\mathcal{L}f(x) = g(x), \tag{1.47}$$

and here we're considering $\mathcal{L} = \partial_x$. Under discretization in $T_n(x)$, $\partial_x \to D_{i,j}$. We have g and want f, so our discretized equation is

$$D_{i,j}\,|f\rangle = |g\rangle \tag{1.48}$$

$$|f\rangle = D_{i,j}^{-1}\,|g\rangle\,. \tag{1.49}$$

Said another way, we need to *solve* for $|f\rangle$, and that means we really want to avoid dense matrices.

If you were waiting for the $U_n(x)$ polynomials to appear, wait no longer. One of the amazing facts about Chebyshev polynomials is that if we project the derivatives of the $T_n(x)$ onto $U_m(x)$, we get

$$\frac{\partial T_n(x)}{\partial x} = nU_{n-1}(x). \tag{1.50}$$

This is a *sparse* matrix with non-zero entries only on the first superdiagonal.[13] A key idea of Dedalus is that we use different basis functions at different stages of the calculations. There are **conversion matrices** that can convert from T to U and back. The details of these matrices are beyond the scope of these notes, but you can read more about them in our 2020 methods paper [14,15].

The final very nice property of the Cheybyshev polynomials, and the reason we use them so much is because they can also be written

$$T_n(\cos\theta) = \cos(n\theta), \tag{1.51}$$

which in turn means we can express them using a discrete cosine series. This allows us to use the **fast Fourier transform (FFT)** to compute the coefficients $hat f_n$ from a set of grid point values $f(x_i)$, where x_i is a set of grid points.

Now, we're in a position to understand how `d3`solves PDEs.

[13] This idea of projecting the derivatives of functions discretized on T onto U is also sometimes called the **ultraspherical** method, because $U_n = C_n^{(1)}$, where $C_n^{(\lambda)}$ are the ultraspherical polynomials. Confusingly, ultrasphericals are also called Gegenbauer polynomials.

[14] K. J. Burns, G. M. Vasil, J. S. Oishi, D. Lecoanet, and B. P. Brown, ''Dedalus: A Flexible Framework for Numerical Simulations with Spectral Methods,'' Physical Review Research **2**, 023068, DOI: `10.1103/PhysRevResearch.2.023068` (2020)..

[15] this covers the older version `2` of the code; there are some differences in `d3`, but for pedagogical purposes these are not important.

```
N = 7 # first seven T
def Dmatrix(n):
    D = np.ones((n, n))
    index = np.arange(n)
    j,i = np.meshgrid(index,index)
    D[:,:] = 2*j*((j-i)%2)
    D[i == 0] /= 2
    D[i >= j] = 0
    return D
Dtt = Dmatrix(N)
plt.imshow(Dtt, cmap='Greys')
plt.savefig('fig/D-T-to-T.pdf')
```

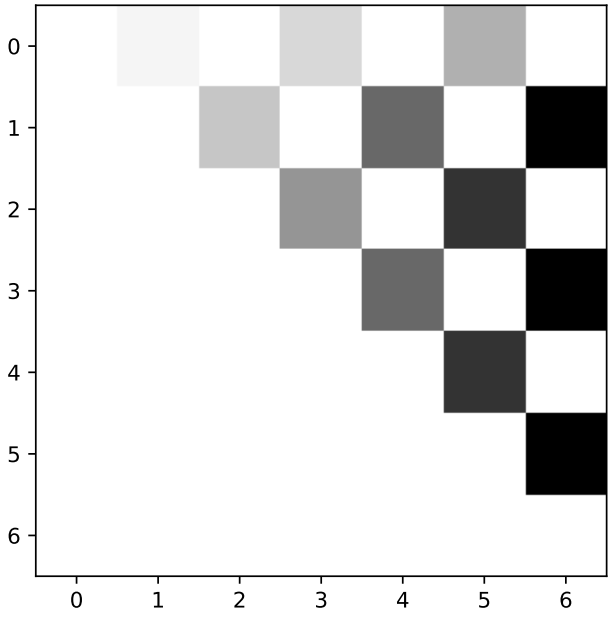

Figure 1.1. The derivative matrix $D_{i,j}$ for Chebyshev T polynomials projected back onto themselves. Note that it is an upper triangular matrix–a *dense* matrix.