Summer School Project: Agenda

I. Organization

- We will meet here in my office, Bloc. 507, each morning
- Start taking notes of what we're doing *now!*
  - Google Drive? Overleaf? Your call...
- Start slides *now!*
  - Overleaf is probably best
  - I can share a template, if you want
- Start a repo for our code...
  - version control?
  - VSLive would be ideal...

II. Mathematical Background

A. The Radiation Transport Equation

- Write out the equation we are studying, mention the meaning of all relevant variables

B. The Discrete Ordinates Approximation

- Show how we can perform collocation in angle to obtain a coupled system of (P)DEs

C. Scalar Transport: First Weak Formulation

- Derive first (Galerkin) formulation
- Talk about ways of enforcing boundary conditions...

III. Begin Coding: reading data and building a mesh

A. Create a new repository

We will write and reuse a lot of code— it is important to be organized. Separate sources from tests.

B. Class: `input_data`

Make a file called `input.py` containing a class called `input_data` which will read all the relevant material parameters, mesh info, *etc.* from a text file at runtime. It should contain:

- `n_zones` : number of materials or 'zones' for our problem
- `n_elements` : (list, 1 per zone) number of elements per zone
- `zone_length` : (list, 1 per zone) length of each zone
- `sigma_t` : (list, 1 per zone) total cross section σ^t in each zone
- `source` : (list, 1 per zone) source q in each zone
- `boundary_values` : (list, two entries) prescribed value on the left and right endpoint (resp).
- constructor : takes a file name, reads this info from said file

C. Class: `mesh`

In *another* file, create a class called `mesh` which takes the inputs and build the specified (1D) mesh:

- `n_points` : number of grid points
- `n_cells` : total number of cells
- `length` : total length of the domain
- `cells` : just a list [0 : `n_cells` -1] for indexing later on
- `gridpoints` : list of the physical location of the grid points (start from the origin)
- `h` : (list, indexed by cell) measure of respective cell
- `mat_id` : (list, indexed by cell) tells us which zone the cell belongs to
- constructor : takes in the `input_data`

D. Testing the Code

We need to be confident that the code you just wrote works. Here are some ideas for checking that:

- print out what it read from the files
- plot σ^t and q against your mesh
- check `total_length == sum(lengths) == sum(h)`

IV. Scalar Transport: Generate RHS

A. Recall Galerkin Formulation

Review this concept and talk about boundary enforcement (strong for now and maybe forever).

⊇ **Caution!** In our input file, we prescribe two boundary values but we would only use one!

B. Compute Source Vector By Hand

We will do this by-hand for the sake of concreteness.

C. Implementing this in Code

Create a new file called `assemble_system.py`. In it, create a function called `assemble_source(mu)`. This should return the relevant RHS vector, call it `right_hand_side`, using our mesh and the boundary conditions we obtain using the angle `mu`.

V. Scalar Transport: Generate System Matrix

A. Overview of CSR format

- Look up `SciPy CSR`
- Make it make sense
- What does this look like for our problem?

B. Generate Sparsity Pattern

In `assemble_system.py`, create a function to generate the sparsity pattern associated with our mesh. Call it `generate_sparsity_pattern`: it should return the rows, columns.

To test this code, print out what it looks like, just filling the non-zero entries with 1s.

C. Calculate the System Matrix

By hand, we will compute what the system matrix should look like.

D. Implement System Matrix Assembly

In `assemble_system.py`, make a function that returns a `scipy csr` matrix for our system using the sparsity pattern and filling in the `data` part according to our work by hand.

Call this function `assemble_system_matrix(mu)`. The bulk of it should look like a loop over the (interior) cells.

VI. Solve our First Transport Problem

A. Main Script for solving the transport equation

Create a new file (probably, a new folder too) : call it `transport.py`. In it, write a script to solve the transport problem using $\mu = 1$. It needs read an input file, set up the specified mesh, assemble the associated FE system, and solve the thing.

For the solver, use `spsolve` with the parameters `use_umfpack = true` and `permc_spec = false`.

B. Exploring Output

- make sure the code works when you set $\mu = -1$
- Give them the benchmark I like, start easy
- what happens as the mesh size changes, especially in relation to σ^t ?

C. Method of Manufactured Solutions

We can actually solve this equation by hand. We will implement an exact solution for the case of three zones and check that our output looks right 'in the eyeball norm.'

VII. Stabilization

A. Motivation

What do we see that goes wrong? Put this in your notes!

B. SUPG: Theory

How do we modify the BLF? What does this do to our matrix and RHS? We can frame it as an addition...

C. Implementing SUPG

Add a new field for the tuning parameter in the `input_data`.

In `assemble_system.py`:

- move our existing assembly functions to a version ending in the suffix `_galerkin` so we still have access
- make our new additions
- does it look right by eye?

D. (Optional) : Convergence Analysis

Record how the solution converges in L^1 as we refine the mesh uniformly. Calculate it the lazy way (ℓ_1 error of soln. vector— should be fine here) and get the rates the lazy way (*i.e.*, manually refine the mesh and record the error).

VIII. Radiation Transport: First Steps

A. RE: Discrete Ordinates Method, how to solve it?

- Notation on soln operator
- Source iteration
- (optional) why does it converge in the first place?

B. Creating an (angular) quadrature

(i). Modify input data

First, add a new field to `input_data` called `n_angles`. Go ahead and add `sigma_s` while you're there.

(ii). Create a class for the quadrature

In `source`, create a new file called `angular_quadrature.py` that defines a class of the same name. It should contain:

- `n_angles` : copied from the input
- `weights` : the weights
- `angles` : the points in the quadrature
- `total_weight` : it should be 2 for us
- constructor : deduce size from input data

Use a Gauss-Legendre quadrature on $[-1, 1]$ (*hint:* `scipy.special.roots_legendre`).

(iii). Create a function to average over the angles

Now, in the same file, we need to create a function to compute the operator $\{\psi^\ell\}_{\ell\in\mathcal{L}} \mapsto \frac{1}{|\mathbb{S}|}\sum_{\ell\in\mathcal{L}} w_\ell \psi^\ell$. Call it `average_over_quadrature(vectors, angular_quadrature)` or (probably better) just add it as a method in `angular_quadrature` so you only need to pass the vectors.

(iv). Test it

Make sure this quad rule looks right... try adding some vectors in a way where we know the average.

C. Implementing Source Iteration

(i). How to make our RHS

We will need to solve our transport problems with a different RHS, one given by $\sigma^s \phi$ for $\phi \in \mathbb{V}_h$. First, we need to add a field, `sigma_s`, in our input data. Next, we can compute how to make this vector by hand and then write a function for it in `assemble_system.py`. Call it `scatter_source(v, mu)` or something similar. You can mostly just copy your previous code. You can even test it using your scalar transport code if you're clever (pick σ^s and q in a smart way).

ƨ **Caution!** Don't forget the stabilization term!

(ii). Set up new script

Make a new repository, call it `solve_radiation`. In it, create a script called `source_iteration.py`. At this point, we should have all of the tools needed to perform the source iteration algorithm.

(iii). Test this code

Run some simple test cases like the Reed problem, a pure scatter, etc.

(iv). Put all this in slides

Even if we have time for the next section, now would be a good time for this.

IX. (Optional) Radiation Transport: DSA

If we have extra time, we can do DSA, we'll just see. We have already made a code for the Poisson problem and have an exact solution for two zones.