Introduction

Aim

Our aim is to simulate dynamics of water in graphs, i.e., in an unstructured grid. Each node holds a given amount of water, and it is transported to other connected nodes only through links. In the simple cases where the graph is either an infinite line or an infinite square lattice, we want our framework to recover some sort of finite differences scheme. Also, mass of water must be conserved. Specifically, we want to find analogs of the first and second order (laplacian) derivative operators, and preferrably, we want all computations to be done vectorially, i.e., with matrix multiplication (as oposed to iterating through nodes in the network).

Literature is scarce

Such an elementary extension to the usual grid discretization of continuous processes must be a well-known established field, right? Well, think again. After a considerable amount of browsing the literature, I found out that there are surprisingly few works on this topic. Perhaps it's because I didn't use the right keywords. Even while writing this, I'm afraid that I might be wrong and that a beautiful book must exist somewhere about these kinds of simulations.

Here's a list of references I have gathered over the last week:

- RAK . Afaik, this is the most thorough description of the advection Laplacian.
- [CHAPMAN]

Chapman and Mesbahi - Advection on Graphs.pdf

  This seems to be the "first paper" on the subject, according to [RAK]
- Shock waves on complex networks, 2014, Mones et al. → In the methods, brief description of how to set up a simulation for 1D spatial derivative in networks
- No package in Python does these kind of simulations, but I found one in written in julia: the NetworkDynamics.jl package. I have exchanged a few emails with the creators, but still I haven't found a straight response to my queries. I'll keep trying!

Two ways of simulating PDEs

There is the vectorial way to simulate the PDE, where the spatial derivatives are discretized using matrix operators, and there is the iterative way, where you iterate over every node.

Vectorial way

The 2nd spatial derivative is easy to do, the 1st derivative is harder.

The 2nd derivative

The graph Laplacian, L, plays the role of the continuous Laplacian. It may be defined as $L = diag(d) - A$, where $diag(d)$ is a diagonal matrix with the degrees of the nodes on the diagonal, and A is the adjacency matrix of the undirected graph.

Let's define the 'straight line graph', that I will use now and later as a special case to check the relationship with finite differences. The straight line graph is a graph in a line with 4 nodes and 3 edges connecting them, ordered like so:

1 → 2 → 3 → 4

So it is a 1D finite difference mesh in disguise.

L looks like this for the straight line graph:

$$L = \begin{pmatrix} 1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 1 \end{pmatrix}$$

This is equivalent (up to a sign!) to the matrix arising from a central finite difference scheme for the second derivative,

$$\begin{pmatrix} BC & BC & BC & BC \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ BC & BC & BC & BC \end{pmatrix}$$

with Neumann boundary conditions in the boundary nodes.

The 1st derivative

Based on the literature above (RAK and CHAPMAN), it seems that the 'modified Laplacian' L_{adv} is the best candidate to represent the first spatial derivative on a graph:

$$L_{adv} = diag(d)_{out} - A_{in}$$

Here, $diag(d)_{out}$ is the matrix of outgoing edges in the diagonal, and A_{in} is the usual adjacency matrix of an directed graph (the subscript in is there to remind us that the adj matrix of directed graphs are composed only of incoming edges). So, in order to compute it, we need a directed graph. In practice, this means fixing the direction of water movement

beforehand, but it's not a big deal because water flows following the gradient, and we know that information.

It can be checked (RAK) that this approach conserves mass.

Note: this is not the most general L_{adv} because we have fixed the weights of edges to be 1. This does not mean that we fix all the gradients between nodes, since the gradients are explicitly present in the definition of L_{adv}, but rather that all edges are equal. Or, more precisely, that the velocity field in which the water flows is constant.

When the graph is the simple straight line this results in

$$L_{adv} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

This is very similar to the 1st order backward finite differences matrix with Dirichlet boundary conditions in the first node,

$$\begin{pmatrix} BC & BC & BC & BC \\ -1 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 1 \end{pmatrix}$$

which arises from $y'(x) = [y(x) - y(x - \Delta x)]/\Delta x$. But with one important difference: the last row of L_{adv} imposes a sort of strange boundary condition for the last node. It only depends on the previous node, but not on itself. This is not strange, of course, if we think of how L_{adv} was generated: the last node has no outgoing edges and so it doesn't have anything in the diagonal.

In practice, this is not a big problem, since we probably want to treat those boundary nodes differently (e.g., by applying some sort of boundary conditions) due to physical reasons.

Let's check the sanity of this setup with another approach to get the same discretized equations.

Final expression

This has not been tested yet, but I think that the update for an explicit forward Euler time discretization of the equation

$$\frac{du}{dt} = a\frac{d^2u}{dx^2} + b\frac{du}{dx} + f(u)$$

would looke like

$$u^{t+1} \leftarrow u^t + \Delta t(-aL/\Delta x^2 + bL_{adv}/\Delta x)u^t + f(u^t),$$

where the minus sign in front of L corrects the sign. Note that Appropriate boundary conditions have to be imposed!

Neighbour iteration way

NOTE: MISSING FINAL EXPRESSION FOR THE ADVECTION-DIFFUSION EQ.

Conceptually, this is a simpler approach. Just iterate over all the nodes in the network, for each node find all its neighbours and perform a balanced trade of mass. That is, if u_i^t is the amount of water at timestep t at node i, then the change in water per timestep for that node is

$$\Delta u_i = \sum_{j \in neighbors} (u_j^t - u_i^t)/\Delta x.$$

Note that this takes care of direction of water flow because Δu is positive if the neighbour has more water than the targe node, and negative otherwise. We are summing over all neighbors, out and ingoing.

We can discretize time in a explicit forward Euler fashion to see how the value of u_i^t may get updated over time:

$$u_i^{t+1} = u_i^t + \Delta t \Delta u_i$$

But what are we computing here? This conserves mass, sure, but does this represent a 1st order derivative, or a 2nd one? At first glance it seems that, since in the Δu_i equation we are computing a discrete version of the gradient, it must be related to the 1st derivative of u_i, i.e., $\frac{du_i}{dx}$. It turns out this is not true! We are, quite unexpectedly, describing $\frac{d^2u_i}{dx^2}$. Here's why.

First of all, note that we can rewrite Δu_i in matrix form using the (undirected) adjacency matrix, A:

$$\Delta u_i \Delta x = A u_i^t - \sum_{j \in neighbors} u_i^t = A u_i^t - n_{neighbors} u_i^t = (A - diag(d)) u_i^t = -L u_i^t.$$

We have just found out that it is the Laplacian, L, who was hiding there. Or rather, the Laplacian with a minus sign. (This sign is anoying but seems to be there for historic reasons). In fact, every time we write a quantity that contains a discrete diference between a node and its neighbors, L will pop out!

Wait a minute... But how do we express the 1st derivative of u_i in this setup, then? I struggled for a while to understand this, but got an answer in the end: we still want to keep the discrete gradient of Δu_i, but instead of suming over all neighbors, we want to sum over incoming ones (I am sure outgoing works too, but I haven't checked the details). This links nicely to what we saw in the matrix section with the modified advection Laplacian L_{adv}, but it also makes intuitive sense if we look at it from the finite differences lens: In finite differences, in order to get a first derivative, we usually take forward or backwards differences,

$$y'(x) = (y(x) - y(x - \Delta x))/\Delta x$$

or

$$y'(x) = (y(x + \Delta x) - y(x))/\Delta x$$

that is, only one of the two adjaacent neighbors of the target node are taken into account. Conversely, when computing the second derivative we use terms such as the centered difference,

$$y''(x) = 1/\Delta x^2[y(x + \Delta x) - 2y(x) + y(x - \Delta x)]$$

i.e., we are considering the two adjacent neighbors. This last section is intented only as an intuitive relationship. There are many finite difference stencils that would blow these arguments up.

Final expression

The pseudocode implementation of a solution to the equation

$$\frac{du}{dt} = a\frac{d^2u}{dx^2} + b\frac{du}{dx} + f(u)$$

would look something like

```
nodes = all nodes of the graph
for i in nodes:
    in_neighbors = neighbors of i
    for j in neighbors:
        TO BE CONTINUED...
```

TODO: add comment Synchronous vs asynchronous updating

Date: 15.12.2020