GROUP 11 PROJECT: A QUANTUM GRAPH SOLVER WITH 2 METHODS

TOMMENIX YU, REX XU, SANSEN WEI

ABSTRACT. In this project, we implement a package which contains two methods, the direct finite difference method and the optimization method to solve a quantum graph of a shifted Laplace equation given boundary conditions. We found that the former method is better with "sparse" graphs (more vertices than edges), and the latter method is better for "dense" graphs (fewer vertices than edges).

CONTENTS

1. OVERVIEW AND BACKGROUND

Numerous fields of research, engineering, and social science have used the generic mathematical concept of a graph (network) as a set of items connected by some relation. Graphs can be used to represent a traffic engineer's street network, a neuroscientist's network of neurons, and the organization of databases in computer science.

In terms of the diffusion equation or the Schrodinger equation, the Laplacian on a metric graph has recently attracted a lot of interest in physics and mathematics. Though they are now referred to as quantum graphs, many components are explored under the titles quantum networks and quantum wires. Physics and mathematics have a long history.

The earliest use of this concept in physics was presumably made by Pauling in the context of free electron models for organic compounds approximately 70 years ago. Since then, this method has undergone additional development. Acoustic and electromagnetic waveguide networks, the Anderson transition in a disordered wire, quantum Hall systems, fracton excitations in fractal structures, mesoscopic quantum systems, and superconductivity in granular and synthetic materials have all been successfully studied using quantum graphs. The experimental simulation of quantum graphs has also been done.

For our purposes, it's sometimes need for use in quantum physics to solve systems of PDEs with connected boundaries, for instance we might be interested in the forces between particles, where each particle is affected by several others. We can simplify that to a PDE problem with its boundary conditions represented by a vertex-edge graph.

This is done by the following process: we start with a graph with nodes and edges. Then, we assign to each of its edges both a differential equation and an interval that the equation is defined on. Thus, each vertex must have values that satisfy the PDEs assigned to every neighboring edge. Hence we get equations that the PDE system must satisfy. Further, we can also assume that the nodes are smooth in terms of that the kth order (start with 1) derivative also satisfies the equation at the nodes.

To make matters concrete, we want to solve the system

$$\frac{\partial^2 u_1}{\partial x^2} = f_1(x) \;\; ; \;\; \dots \;\; ; \;\; \frac{\partial^2 u_n}{\partial x^2} = f_n(x)$$

and boundary conditions $\forall k = 1, 2, \dots, |V|$:

$$\begin{cases} f_{i1}(x_{i1}(v_k)) = f_{i2}(x_{i2}(v_k)) = \cdots = f_{in}(x_{in}(v_k)) & \text{for } e_{ij} \text{ connected to } v_k \\ \frac{\partial f_{i1}}{\partial x} x_{i1}(v_k) + \frac{\partial f_{i2}}{\partial x} x_{i2}(v_k) + \cdots + \frac{\partial f_{in}}{\partial x} x_{in}(v_k) = 0 & \text{for } e_{ij} \text{ connected to } v_k \end{cases}$$

where $n = |E| := \#$ number of edges, and $x_e(v_k)$ is the value of the node v_k's corresponding value in the interval.

2. Statement of the problem

In our project, we would like to address and solve the following problem. Suppose there is a graph structure that consists of vertexes and edges, and an interval on each edge. We further add a function f on each edge so that it satisfies the equation

$$\Delta u = f$$

where $\Delta u = \dfrac{d^2u}{dx^2}$, and the boundary condition is that for all vertex v, suppose there's k edge connected to it, then

$$\sum_{i=1}^{k} \text{sign}(i) \frac{df_i}{dx}(E_i(v)) = 0$$

where $E_i(v)$ picks out the value of the vertex at every edge it is on. Also, we want a solution that is connected, which means that the value of any vertex is the same on any edge it is connected to, i.e.

$$u_1(E_1(v)) = \cdots = u_k(E_k(v)).$$

But we realized that the solution is not unique due to addition of a linear term. Therefore we modify the equation to the form

$$\Delta u - u = f$$

in order to get a unique solution.

To solve this problem, We will gradually build up to our solver to the system along with detailed explanation, and a user's manual function to input such a system easily into the program. Specifically, we are using two methods: Direct finite difference method and Optimization method to solve this problem.

3. Description of implementation

In our project, four classes called Graph, Metric-Graph, Graph-PDE and Graph-PDE-BVP are defined such that the following class inherit from the previous class. Also, we define some function such that we can call them inside our class. Here are some brief descriptions of our four classes. Our two methods are included in these objects.

3.1. **Graph Object.**

This is the mother object of this project. We are able to obtain a graph out of a vertex-(connected vertex) dictionary by creating such an object. So the input for creating this object is a graph dictionary whose keys are the nodes and values are nodes connected to them via an edge, and the output is a graph object, which also contain several methods that are useful to modify the graph object such as adding edges and vertices, finding edges and vertices, a print method to print the object and a visualize method to show the actual shape of the graph.

3.2. **Metric-Graph Object.**

The class Metric-Graph is a subclass of the Graph class. Metric graph is a graph with an interval on each edge, thus we need to input another edge-interval dictionary. As a result, the input is a graph dictionary plus an edge-interval dictionary, and the output is a metric graph object.

Note that this is another base object. So we are not solving equations directly here.

3.3. **Graph-PDE Object.**

This is the quantum graph object. This class is a subclass of the Metric-Graph class. The input also includes an edge-pde dictionary, in addition to the graph dictionary and the edge-interval dictionary.(Even though we always input ODE here, there really is potential to solve PDEs and that's our initial goal, so we put it here.) The output is a graph-PDE object.

We implement two methods of solving the system here, which would be introduced in more detail in the following section. The first is a self-contained solver of the system using discretization. The other changes the problem into a bvp problem plus an optimization problem.

3.4. **Graph-PDE-BVP Object.**

This is the BVP quantum graph object. This class is a subclass to the Graph-PDE class. In addition to the graph dictionary and the edge-interval dictionary, edge-pde-dictionary input, there's also a vertex-value dictionary input. (Even though we always input ODE here, there really is potential to solve PDEs and that's our initial goal, so we put it here.) And the output is a graph-PDE-BVP object. We use this part just to solve the BVP problem, not directly solving the quantum graph problem. However this is everything we need to solve the quantum graph using optimization method.

4. Two Methods

4.1. **Direct Finite Difference Method.**

Our goal here is to find a way to solve for u given f and following 3 types of conditions:

Type 1. second order derivative:

$$\Delta u = f$$

Type 2. value of u agrees on nodes. For a vertex v, say u_i, $i = 1, \cdots, k$ are functions on edges connected to v. Then:

$$u_1(v) = u_2(v) = \cdots = u_k(v)$$

Type 3. first order derivative: Let sign(i) depends on direction of the edge.

$$\sum_{i=1}^{k} \text{sign}(i) \frac{du_i}{dx}(v) = 0$$

Now, say the graph has m edges, p vertices, and each intervals (on edges) are divided into n points. f_i, $i = 1, \cdots, m$ denotes the function f_i over i-th edge. Let $x_{k,0}, \cdots, x_{k,n-1}$ be the n grid points for the k-th edge. Let $T_{k,0}, \cdots, T_{k,n-1}$ be the solution (u's values) on the grids.

For type 1 conditions, we use the **centered** second-order accurate second-order derivative matrix for the discretization. Hence the values of f at starting point and ending point are not included. For edge k, we have:

$$A_k = \frac{1}{\Delta x^2}\begin{pmatrix} 1 & -2 & 1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \ddots & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \ddots & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 0 & \ddots & 0 & 0 & 1 & -2 & 1 \end{pmatrix}$$

And

$$A_k \begin{pmatrix} T_{k,0} \\ T_{k,1} \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ T_{k,n-1} \end{pmatrix} = \begin{pmatrix} f_k(x_{k,1}) \\ f_k(x_{k,2}) \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ f_k(x_{k,n-2}) \end{pmatrix}$$

Let $b_{k,l}$ denotes $f_k(x_{k,l})$ For each $k = 1, \cdots, m$, this gives $(n-2)$ linear equations. We can stack all the equations together blockwise on the diagonal. This gives a big matrix A of size $(n-2)m \times nm$. For example, if $m = 2$. We have the 2 matrices A_1 and A_2 each representing the ODE on each edge:

$$A = \begin{pmatrix} A_1 & 0 \\ 0 & A_2 \end{pmatrix}$$

And thus we have the new equation:

$$\begin{pmatrix} A_1 & 0 \\ 0 & A_2 \end{pmatrix} \begin{pmatrix} T_{1,0} \\ \vdots \\ T_{1,n-1} \\ T_{2,0} \\ \vdots \\ T_{2,n-1} \end{pmatrix} = \begin{pmatrix} b_{1,1} \\ \vdots \\ b_{1,n-2} \\ b_{2,1} \\ \vdots \\ b_{2,n-2} \end{pmatrix}$$

Boundary contidions:

For type 2 conditions, for a vertex v with degree d_v and values of $u_{v,1}, \cdots, u_{v,d_v}$ over edges: $T_{v,1}, \cdots, T_{v,d_v}$. We want the corresponding entries (represent $u_d(v)$) in $T_{v,d}$ matches, $d = 1, \cdots, d_v$. This is done by adding a row containing an 1 and -1 at corresponding column indices. For example, if we want $T_{1,n-1} = T_{2,0}$, we have this row equation, where we have 1

at the n-th column and -1 at the $n+1$-the column:

$$\begin{pmatrix} 0 & \dots & 0 & 1 & -1 & 0 & \dots & 0 \end{pmatrix} T = (0)$$

This results in $(2 * m - p)$ row equations.

For type 3 conditions, suppose at vertex "b", we have an interval T_1 ending at "b" and an interval T_2 starting at "b". For u' starting at "b", we use the forward difference with coefficients -1.5, 2, 0.5. For u' ending at "b", we use the backward difference with coefficients 1.5, -2, 0.5. We put them on the same row. For example (we may multiply these coefficients by -1 depending on direction of the derivative:

$$\frac{1}{\Delta x}\begin{pmatrix} 0 & \dots & 0 & 1.5 & -2 & 0.5 & -1.5 & 2 & -0.5 & 0 & \dots & 0 \end{pmatrix} T = (0)$$

This gives p equations in total for p vertices.

Now, we simply vertically stack all the linear equations (rows, matrices, and values) form a full linear system $M * T = b$, where M is of size $nm \times nm$. Vector T has nm entries, and b has $(n-2) * m$ entries with values from f and $2 * m$ zeros.

With this method we can solve the quantum graph using your favorite way to solve $M * T = b$.

Furthermore we discuss how to change only the equation from $\Delta u = f$ into $\Delta u - u = f$. This is actually very simple since we only have to change A into

$$B = A - Id$$

since $AT \approx T''$ and $IdT = T$, where the right hand stays the same.

All other trivialities in the modification of b and addition of rows of A stays the same through this method, and we will get the solution for the shifted Laplace equation.

This also gives us insight on how to generate the system to any kind of PDEs: simply modifying the solver for 1 edge graph is enough as we will repeatedly use that or stack that up to a larger matrix, which means that our package is actually easy to extend to a fast and general quantum graph solver, given that we have a fast and general solution to a 1D PDE equation.

For the second method, we'll not discuss how the shift is done since it is the same.

4.2. **Optimization Method.**

This is another way to solve the question via optimization. To do this, we still need to solve a boundary value problem with the discretization method. Then, we write up a net flow function that computes the flux (sum of derivative at each vertex) and do minimization on this value.

Step 1: solving the BVP problem.

So to do this, the first step is to create a Graph-PDE-BVP object.

As discussed in the previous section, this is an object where with the basic setting of a quantum graph, we require an additional vertex-value dictionary. The reason for this is that we want to solve a BVP problem on the graph. (For more detailed user's manual, look below.)

Note that since we know the value at each vertex, solving an extremely complicated graph is just as easy as solving many individual graphs together. Also note that we are not posing any condition on the derivative at the end points. (So Nuemann condition turns into Dirichlet condition).

How to do this? We use the same finite difference matrix A and get the $AT = b$ equation.

Note that the derivative at x_1 (second grid point) is

$$\frac{T_0 - 2T_1 + T_2}{\Delta x^2} = b_1$$

Thus, to encode the boundary condition $T_0 = aval$ we only need to let

$$b_1' = b_1 - \frac{aval}{\Delta x^2},$$

and because of this extra constrain we can discard the first row and manually add the value of

$$T_0 = aval.$$

Similarly we have

$$b_{n-1} = b_{n-1} - \frac{bval}{\Delta x^2} \quad \text{and} \quad T_n = bval.$$

Thus, we can truncate the original matrix to

$$A' = \frac{1}{\Delta x^2}\begin{pmatrix} -2 & 1 & 0 & 0 & \dots & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & \dots & 0 & 0 & 0 \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ 0 & 0 & 0 & \ddots & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & \ddots & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & \ddots & 0 & 0 & 1 & -2 \end{pmatrix}$$

that is of size $(nx-2)\times(nx-2)$, where nx is the length of the grid of the interval, containing both endpoints. So the equation becomes

$$\frac{1}{\Delta x^2}\begin{pmatrix} -2 & 1 & 0 & 0 & \dots & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & \dots & 0 & 0 & 0 \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ 0 & 0 & 0 & \ddots & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & \ddots & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & \ddots & 0 & 0 & 1 & -2 \end{pmatrix}\begin{pmatrix} T_1 \\ T_2 \\ \vdots \\ \vdots \\ \vdots \\ T_{n-2} \\ T_{n-1} \end{pmatrix} = \begin{pmatrix} b_1 - \frac{aval}{\Delta x^2} \\ b_2 \\ \vdots \\ \vdots \\ \vdots \\ b_{n-2} \\ b_{n-1} - \frac{bval}{\Delta x^2} \end{pmatrix}$$

Thus we have solved the BVP from some required vertex-values.

Step 2: Solving the minimize problem.

Now that we can solve the equation for any given vertex-value problem on a graph, we can count the netflow of the solution related to it.

To be precise, we use again finite difference to approximate the flow at each vertex. The good thing about finite difference is that we won't have to worry about the direction of the flow anymore since they are already encoded in the expressions

$$F_{in} = -\frac{3}{2}T_0 + 2T_1 - \frac{1}{2}T_2 \quad \text{and} \quad F_{out} = -\frac{3}{2}T_n + 2T_{n-1} - \frac{1}{2}T_{n-2}.$$

So what we need to do in this finding netflow function is to find each vertex, sum all fluxes around it using above expressions (make sure in/out matches), taking the absolute value of this flux at one point, and summing up the value for each point.

This done, we can just do minimization on this value for some given starting initial value of each interval ($\in \mathbb{R}^{|V|}$).

5. Results

We generate some tests that we already know the answers to test the correctness, accuracy and time complexity for both methods.

To test the direct finite difference method, we come up with a simple example, that is to solve $u'' - u = f$, where $f = -(x^4 - 4x^3 - 8x^2 + 24x - 8)$. The solution we expect is $x^2(x-2)^2$, so that it is flat at 0 and 2, and we split the section $[0, 2]$ into $[0, 1]$ and $[1, 2]$, and the result goes as our expectation. It does not take a long time for this code to run.

On the other hand, we use the expected solution $u = \cos(x)$ for the $u'' - u = f$ equation for testing the optimization method. By computation we have

$$f = u'' - u = -2\cos(x)$$

and we can compute the solution for $u'' = f$ is $u = 2\cos(x)$.We do that on the intervals $[0, \pi/2]$, $[\pi/2, \pi]$, connected at the middle and alone at the end. The solution is indeed what we expect, which is exactly the cos curve. However, the running time is very slow.

The analysis of both methods tells us that if the graph is simple, like the ones we used as toy examples and tests, then the direct finite sum method should be much faster since it involves only one not-so-large matrix multiplication. Whereas the optimization usually compute around 300 steps of guesses and for each step it needs to compute $|E|$ small matrix computations.

However, if the graph is large and complicated, which means $nx \times |E|$ is a huge matrix that we cannot afford to do even with machines, then using the optimization method would be much faster since it is $O(|E|nx^2)$ where as the direct method is $O(|E|^2nx^2)$.

6. Conclusions and Future Work

We draw a conclusion that in general, for "sparser" graphs, which means the number of vertices is a lot larger than the number of edges, it's in principle faster to use the finite difference method.

On the contrary, if the graph is "dense", which means the number of vertices is a lot smaller than the number of edges, then it's expected faster to use the optimization method, even it's a lot slower for simple examples.

Some directions of future work could be:

(1) The first thing to do is to use a better method to do the optimization, so that the optimization method doesn't take such long time.
(2) There's the issue with condition number of the finite difference method, which is quite large. Thus it might produce results with slightly off value at the endings.

    To improve this, we've already tried methods to make it better conditioned, but more or less the result doesn't improve much.

    So we'd like to find more ways to solve this problem of ill-conditioning.
(3) As can be seen from the method we use, it really is a matter of which finite difference matrix you use to solve the problem. Hence, we can solve not only for $u'' = f$ and $u'' - u = f$, but for any other form of ODE and PDE (with the right matrix) easily.
(4) Some of the initial thoughts about this project is based on a PDE structure of the problem, which we cannot do here. But let's say that we have a solver for a specific kind of PDE, maybe just Heat Equation. (note that if the heat equation is homogeneous, we're already done.) Then we can add the following features which may require efforts:
    - The long term behavior of the PDE. That is, in finding the solution, we are interested in the equilibrium state solution, or whether that exist.
      A first step in finding that is to find a steady state equilibrium, and for periodic or even chaotic equations we might figure things out or not...
    - Now, we can implement detectors of the type of equilibrium.
    - So if we can finish the above, it's tempting to add a perturbation to the problem, i.e. a multiplied normal distribution on some edge.
      This is no easy work and requires recordings of the state of the system in each time step.
(5) We can work more on the graph structure, which, even though we've implemented, isn't explored much.

A possible thing to do is to find for what kind of graph it's possible to have solution for some input functions. A thing that I personally is curious about is whether the "convertability to a planary map" of the graph relates to the stability of the solution. If there are interesting patterns, then there might even be interesting topological/geometry results to relate to or discover.

APPENDIX A. REFERENCE

For basic graph objects:

https://python-course.eu/applications-python/graphs-python.php

For using networkx to visualize a graph:

https://www.geeksforgeeks.org/visualize-graphs-in-python/

Discritization to solve a Laplace equation:

https://aquaulb.github.io/book_solving_pde_mooc/solving_pde_mooc/notebooks/03_FiniteDifferences/03_03_BoundaryValueProblems.html