2D eigensolver

Daniel Huantes

June 2022

1 What: What are we doing

We are creating a 2D eigensolver in python. Specifically, we are applying a similar method to our 1D eigensolver in an attempt to model a particle confined to some 2D box.

$$\hat{H} |\psi_n\rangle = E_n |\psi_n\rangle \tag{1}$$

Again, we observe the schrodinger equation in Hamiltonian form, but this time we expand it slightly to explain where the '2D' comes in

$$\begin{aligned}
\hat{H} |\psi_n\rangle &= (\hat{V}(x,y) + \hat{T}) |\psi_n\rangle &(2)\\
&= (\hat{V}(x,y) + \frac{\vec{p}^2}{2m}) |\psi_n\rangle &(3)\\
&= (\hat{V}(x,y) + \frac{\mathbf{p}_x^2}{2m} + \frac{\mathbf{p}_y^2}{2m}) |\psi_n\rangle &(4)
\end{aligned}$$

2 Why: Why are we doing it

In theory we could expand to a full 3D, but the ultimate problem we are concerned with is the propagation of a beam through variable refractive index. Treating the changes in refractive index, the Helmholtz equation has the same form as the schrödinger equation, so the 2 spacial dimensions of each 'slice' are the 2 spacial dimensions the 'particle' finds itself in. The beam propagation direction is represented by time.

3 How: What is the design method for this

the state vector $|\psi\rangle$ can be cast to multiple different forms, most commonly that of the probability amplitude function $\psi(x) = \langle x|\psi\rangle$. In the 1D case, we discretize the domain of this function, restricting it to some finite range, and representing it as a 1D numpy array. The utility of using a 1D numpy array is that the Hamiltonian operator can be represented as a 2D matrix, and we can take advantage of existing libraries and methods to calculate it's eigenvectors.

We want to find a representation for a 2D version of our state vector, some code representation that can allow us to keep the hamiltonian in a form we can reasonably calculate the eigenvectors of. My initial idea was to use a 2D grid,

$$\psi(x,y) = \begin{bmatrix} \psi(x_0,y_0) & \psi(x_0,y_0+\Delta y) & \psi(x_0,y_0+2\Delta y) & ... \\ \psi(x_0+\Delta x,y_0) & \psi(x_0+\Delta x,y_0+\Delta y) & \psi(x_0+\Delta x,y_0+2\Delta y) & ... \\ \psi(x_0+2\Delta x,y_0) & \psi(x_0+\Delta x,y_0+\Delta y) & \psi(x_0+2\Delta x,y_0+2\Delta y) & ... \\ \vdots & \vdots & \ddots & \ddots \end{bmatrix} \tag{5}$$

This allows for the $\mathbf{p}_x^2$ to remain the same.

$$\mathbf{p}_x^2 = -\hbar^2\nabla_x^2 = -\hbar^2\frac{\partial^2}{\partial x^2} \tag{6}$$

In the 1D case we wrote the x-momentum operator as a matrix taking advantage of the finite difference method to write the 2nd derivative

$$\mathbf{p}_x^2 = -\frac{\hbar^2}{\Delta x^2}\begin{bmatrix} -2 & 1 & 0 & 0 & \dots \\ 1 & -2 & 1 & 0 & \dots \\ 0 & 1 & -2 & 1 & \dots \\ \vdots & \vdots & \vdots & \ddots & \ddots \end{bmatrix} \tag{7}$$

using a more concise α and β as shorthand

$$\mathbf{p}_x^2\left|\psi\right\rangle = \begin{bmatrix} \alpha & \beta & 0 & 0 & \dots \\ \beta & \alpha & \beta & 0 & \dots \\ 0 & \beta & \alpha & \beta & \dots \\ \vdots & \vdots & \vdots & \ddots & \ddots \end{bmatrix}\begin{bmatrix} \psi(x_0) \\ \psi(x_0+\Delta x) \\ \psi(x_0+2\Delta x) \\ \vdots \\ \psi(x_f) \end{bmatrix} \tag{8}$$

we can observe then that this matrix ALSO works for $\psi(x,y)$

$$\mathbf{p}_x^2\psi(x,y) =$$
$$\begin{bmatrix} \alpha & \beta & 0 & 0 & \dots \\ \beta & \alpha & \beta & 0 & \dots \\ 0 & \beta & \alpha & \beta & \dots \\ \vdots & \vdots & \vdots & \ddots & \ddots \end{bmatrix}\begin{bmatrix} \psi(x_0,y_0) & \psi(x_0,y_0+\Delta y) & \psi(x_0,y_0+2\Delta y) & ... \\ \psi(x_0+\Delta x,y_0) & \psi(x_0+\Delta x,y_0+\Delta y) & \psi(x_0+\Delta x,y_0+2\Delta y) & ... \\ \psi(x_0+2\Delta x,y_0) & \psi(x_0+\Delta x,y_0+\Delta y) & \psi(x_0+2\Delta x,y_0+2\Delta y) & ... \\ \vdots & \vdots & \ddots & \ddots \end{bmatrix}$$

At least in the case of $\mathbf{p}_x^2$, this representation works. The trouble beings when inspecting the case of $\mathbf{p}_y^2$. By observation, all I've found in the way of $\mathbf{p}_y^2$ is the following result

$$\mathbf{p}_y^2\left|\psi\right\rangle = (\mathbf{p}_x^2\boldsymbol{\psi}(x,y)^T)^T \tag{9}$$
$$= \boldsymbol{\psi}(x,y)\mathbf{p}_x^{2\,T} \tag{10}$$

But at least with the methods we are using, I don't believe we can apply post-multiplication of matrices as an operator, as our Hamiltonian will be interpreted as pre-multiplying. so, instead I propose another method.

Similar to how matrices are stored in memory, we can instead encod all of our 2D data into a single 1D eigenvector (numpy array), keeping track of the 'stride' to parse information. The trouble with representing ψ as a 2D array can be shown as follows, what we want is an operator acting on our state vector that produces a state vector with a (numerical approximation of) the derivative evaluated at that point in a given direction.

$$\begin{bmatrix} M_{11} & M_{12} & \dots \\ M_{21} & M_{12} & \dots \\ \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} \psi_{11} & \psi_{12} & \dots \\ \psi_{21} & \psi_{12} & \dots \\ \vdots & \vdots & \ddots \end{bmatrix} = \begin{bmatrix} \partial^2\psi|_{x_o,y_o} & \partial^2\psi|_{x_o,y_1} & \dots \\ \partial^2\psi|_{x_1,y_o} & \partial^2\psi|_{x_1,y_1} & \dots \\ \vdots & \vdots & \ddots \end{bmatrix} \tag{11}$$

In the case of the y-direction though, because of matrix-matrix multiplication rules, the 11 entry of the resultant matrix could only be calculated using information about ψ values with the same x coordinate.

Instead, by representing the state vector as a single 1D vector, and having the operator be a matrix-vector multiplication, we can have global information available at every single entry's calculation, allowing us to preform any kind of derivative, directional or along an axis, via finite-diference methods.

We transition to something like this

$$\begin{bmatrix} M_{11} & M_{12} & \dots \\ M_{21} & M_{12} & \dots \\ \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} \psi_{x_o,y_o} \\ \psi_{x_1,y_o} \\ \vdots \\ \psi_{x_o,y_1} \\ \psi_{x_1,y_1} \\ \vdots \\ \vdots \\ \psi_{x_o,y_f} \\ \psi_{x_1,y_f} \\ \vdots \end{bmatrix} = \begin{bmatrix} \partial^2\psi_{x_o,y_o} \\ \partial^2\psi_{x_1,y_o} \\ \vdots \\ \partial^2\psi_{x_o,y_1} \\ \partial^2\psi_{x_1,y_1} \\ \vdots \\ \vdots \\ \partial^2\psi_{x_o,y_f} \\ \partial^2\psi_{x_1,y_f} \\ \vdots \end{bmatrix} \tag{12}$$

We can now actually write our operators for p_x and p_y.

We will write them via submatrices, first defining a couple of useful submatrices. The nice thing about these definitions is they actually mirror the usual finite difference method. We will again be using α and β to represent our finite difference coefficients.

If we are dealing with a uniform square grid on the xy plane divided into $N \times N$ squares, as represented in our single vector, two indices i and j refer to ψ at the same x location if $\lfloor i/N \rfloor = \lfloor j/N \rfloor$ and the same y location if $i \equiv j (\mathrm{m}odN)$. That is all to say our 'stride' is N

Because of this, each 'row' becomes a group of N contiguous values in our vector, and each 'column' becomes the set of the i^{th} value in all of these groupings.

If we (again) define the familiar $\mathbf{p}_x^2$matrix, but this time giving notation to indicate it is to be a submatrix

$$\left[\mathbf{p}_x^2\right] \equiv \mathbf{p}_x^2 = \begin{bmatrix} \alpha & \beta & 0 & 0 & \dots \\ \beta & \alpha & \beta & 0 & \dots \\ 0 & \beta & \alpha & \beta & \dots \\ \vdots & \vdots & \vdots & \ddots & \ddots \end{bmatrix} \tag{13}$$

and we introduce the *new* matrices $\hat{\boldsymbol{\alpha}}$and $\hat{\boldsymbol{\beta}}$

$$\left[\hat{\boldsymbol{\alpha}}\right] \equiv \hat{\boldsymbol{\alpha}} = \alpha\hat{\mathbf{1}} \tag{14}$$

$$\left[\hat{\boldsymbol{\beta}}\right] \equiv \hat{\boldsymbol{\beta}} = \beta\hat{\mathbf{1}} \tag{15}$$

where $\hat{\mathbf{1}}$is the familiar identity matrix.
An important note is that all of these submatrices are $N \times N$. The final definition is simple, a $N \times N$ matrix with 0 for every entry: $\left[0\right]$. Now we are finally prepared to construct our momentum operators

$$\mathbf{p}_x^2 = \begin{bmatrix} \left[\mathbf{p}_x^2\right] & \left[0\right] & \left[0\right] & \dots \\ \left[0\right] & \left[\mathbf{p}_x^2\right] & \left[0\right] & \dots \\ \left[0\right] & \left[0\right] & \left[\mathbf{p}_x^2\right] & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{16}$$

$$\mathbf{p}_y^2 = \begin{bmatrix} \left[\hat{\boldsymbol{\alpha}}\right] & \left[\hat{\boldsymbol{\beta}}\right] & \left[0\right] & \left[0\right] & \dots \\ \left[\hat{\boldsymbol{\beta}}\right] & \left[\hat{\boldsymbol{\alpha}}\right] & \left[\hat{\boldsymbol{\beta}}\right] & \left[0\right] & \dots \\ \left[0\right] & \left[\hat{\boldsymbol{\beta}}\right] & \left[\hat{\boldsymbol{\alpha}}\right] & \left[\hat{\boldsymbol{\beta}}\right] & \dots \\ \vdots & \vdots & \vdots & \ddots & \ddots \end{bmatrix} \tag{17}$$

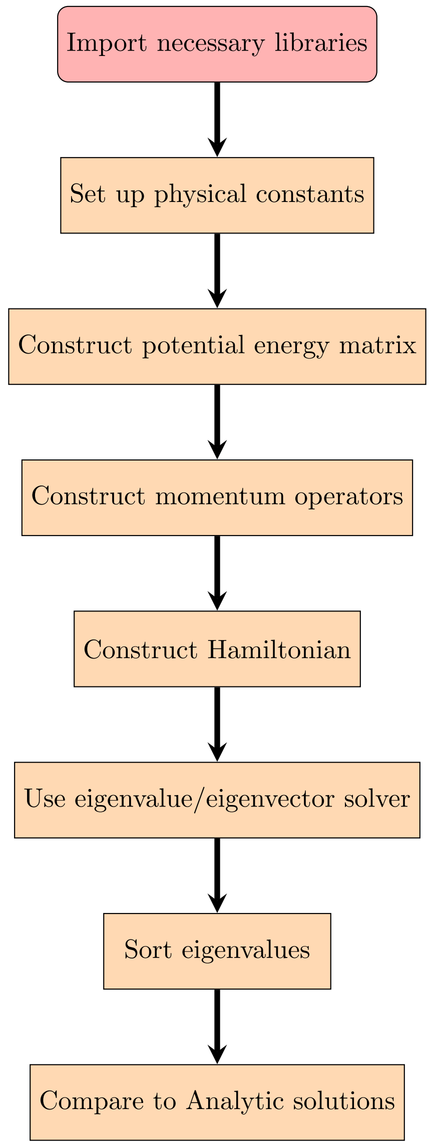
In their current form they seem familiar, interestingly there is a kind of symmetry, the $\mathbf{p}_x^2$operator and $\mathbf{p}_y^2$operator can be contstructed by replacing each nonzero entry with the $\mathbf{p}_x^2$operator and each zero entry with the zero matrix, and the other by replacing each entry of the $\mathbf{p}_x^2$matrix with that entry times the identity matrix

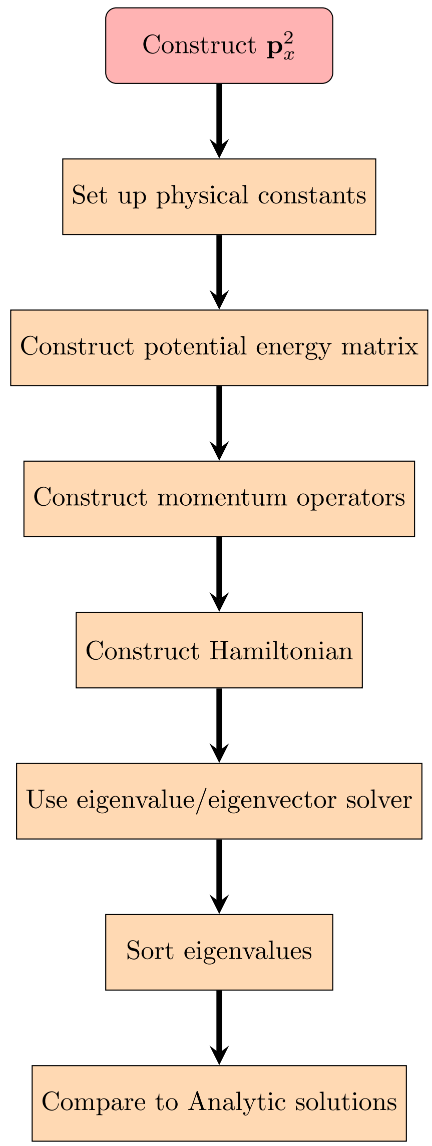
4 How: How is this implemented in code

These matrices are both banded, sparse, $N^2 \times N^2$ matrices. This size increase does introduce a more significant computational cost to calculating the eigenvectors of the Hamiltonian, however there does exist methods for dealing with increasing the efficiency of these operations for sparse matrices.

A code implementation of a non-optimized proof of concept python script is included *here*. The pseudo code for that project is also included as scaffolding $\langle a|b\rangle$

4.1 pseudo


Import necessary libraries
Set up physical constants
Construct potential energy matrix
Construct momentum operators
Construct Hamiltonian
Use eigenvalue/eigenvector solver
Sort eigenvalues
Compare to Analytic solutions



5 How: How can this be used in the future