Project 4: 3-D Pattern Recognition

Background

High energy particle and nuclear collisions, such as those at the CERN Large Hadron Collider, produce extremely large volumes of data. For example, the ALICE experiment records $\simeq$ 1.25 GB of data per second during nucleus-nucleus collisions. The majority of this data volume comes from the Time Projection Chamber (TPC) - a detection device that acts like the world's largest 3D digital camera. When a collision occurs, thousands of particles are produced that fly through the detector in under 50 nanoseconds. The charged particles leave behind a signature of their passage by ionizing the gas inside the chamber as they go. The trails of ionization left behind are drifted toward charge collection regions by high voltage electric fields. The charge is amplified then sampled in discrete packets which are digitized and stored with position and timing information. Typically there are five numbers associated with a voxel - volume, row, column, time bucket, and energy deposited in digitized units (ADC) - and in the case of ALICE, millions of voxels in the detector volumes. All of this happens on very short time-scales, typically microseconds or less. The time it takes to fully read out the detector is dominated by the drift speed of the ionization clouds through the large ($\sim$1-5 meters) gas volume. Figure 1 shows a typical nuclear collision in the ALICE TPC, with the thousands of ionization trails shown.

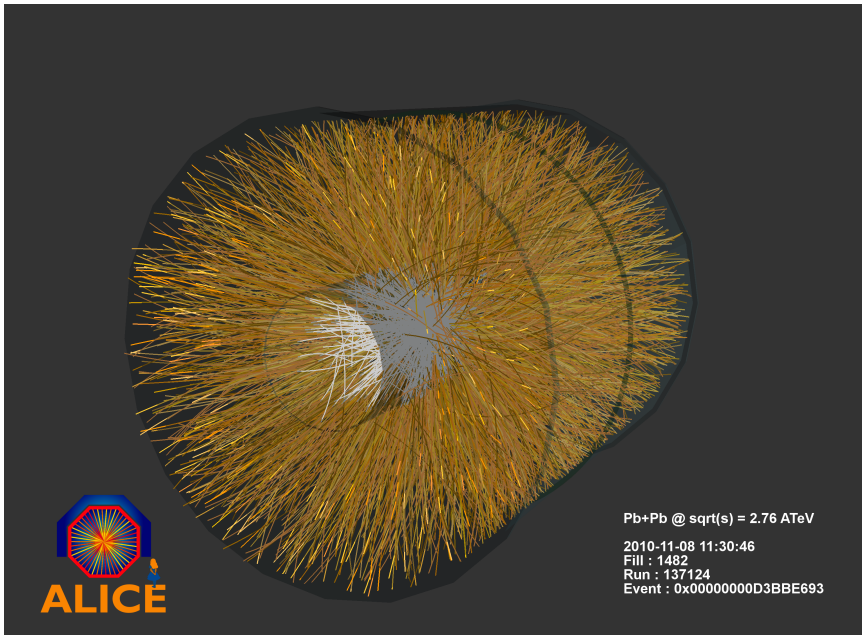

Figure 1: Typical nuclear collision event in the ALICE time projection chamber. Thousands of charged particles leave behind curved tracks of ionization.

Once the data has been collected, the first step in reconstructing the collision debris is to perform pattern recognition. Humans are very good at picking out patterns, but it would be impossible to use human power to reconstruct the billions of events collected in a typical dataset. That's where computers and pattern recognition algorithms come in. There are many different types of algorithms in use. Their complexity depends on the type of patterns they are trying to detect and the level of noise they have to filter out. In experiments such as ALICE, there are thousands of particles in a given event and their trajectories cross and overlap. In addition, ALICE uses a magnetic field to cause the charged particles to curve (due to the Lorentz force, $\vec{F} = q\vec{v} \times \vec{B}$) as they travel, leaving helical trajectories behind. Combine these two effects - high data volume and complicated trajectories - and the pattern recognition gets very challenging.

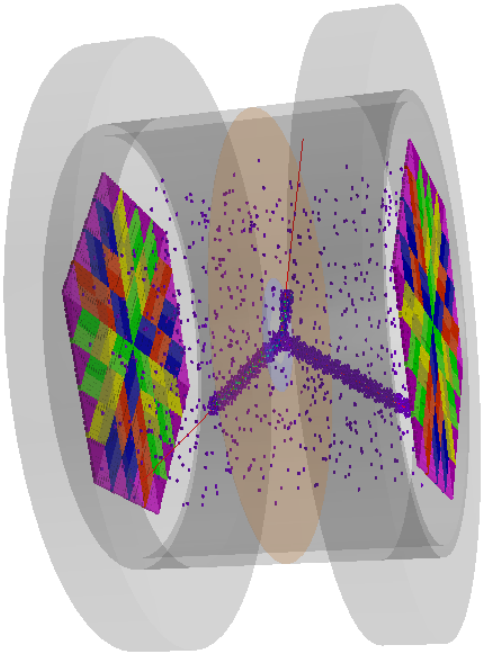

Figure 2: Typical fission fragment event in the NIFFTE TPC. Only a few particles with straight line tracks that stop within the volume are detected per event.

High energy collisions are not the only application for time projection chambers. For example, the Neutron Induced Fission Fragment Tracking Experiment (NIFFTE) has built a small (coffee-can sized) time projection chamber to measure the charged nuclear fragments that come from neutron-induced nuclear fission events at relatively low energies. In this experiment, there is no magnetic field, so the trajectories of the particles are straight lines. Each event produces only ∼1-5 particles and they do not travel very far in the drift gas before they are slowed down to a complete stop inside the detector. The simplicity of

these events makes them much easier to reconstruct, but writing fast, efficient algorithms that don't miss or misidentify any of the tracks is still challenging enough. Figure 2 shows a typical collision event inside the NIFFTE drift volume, which has approximately 3000 individual x, y pixels arranged in a hexagonal pattern and 100-150 time buckets in the z direction on each side of the target plane through the center, for a total voxel count of ∼600,000.

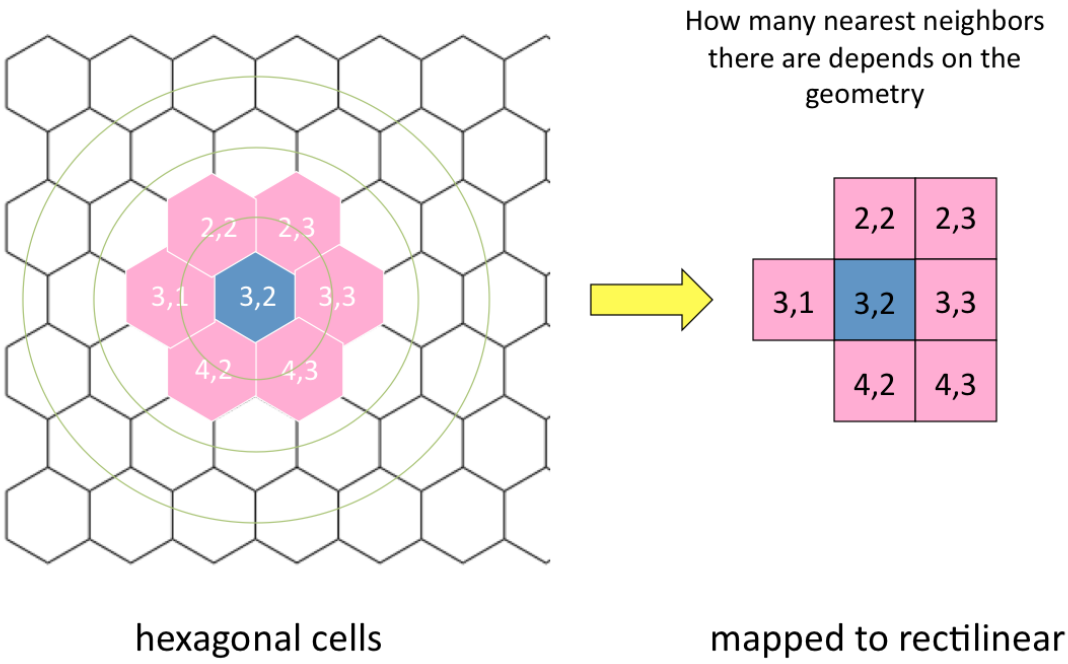

Figure 3: Nearest neighbors in a hexagonal geometry map to rectilinear space with two extra neighbors.

In this case, it is the hexagonal geometry that introduces an additional complexity. When you want to find the trajectories among sparse data, you need to be able to identify the voxels with charge in them and associate them to their nearest neighbors that also have charge. In cartesian (rectilinear) space, a given 2-D square has four nearest neighbors that share an edge, but in hexagonal space there are two additional edges. If you try to map the numbering of hexagonal voxels to rectilinear space, it amounts to adding the corner elements on one side (depending on the orientation of the hexagons). Figure 3 illustrates the point. Any algorithm that searches on nearest neighbors needs to account for the additional voxels.

In order to find neighbors with signals that should be combined into a single trajectory,

a common method is to order the voxels by ADC value, start with the highest value voxel and calculate the gradients (directional derivative) in all directions with its nearest neighbors to determine which ones should be combined. Numerically, the gradient is just the difference in the ADC values divided by the difference in position of the voxels. If the gradient between two neighbors is low, they should be considered part of the same trajectory and combined. If the gradient is high, then the neighbor should be discarded, as this represents an edge to the distribution. In hexagonal geometry, calculating the gradients in 3D amounts to computing the differences between all the hexagonal neighbors of one plane as well as those in the plane ahead and behind, as illustrated in Figure 4.

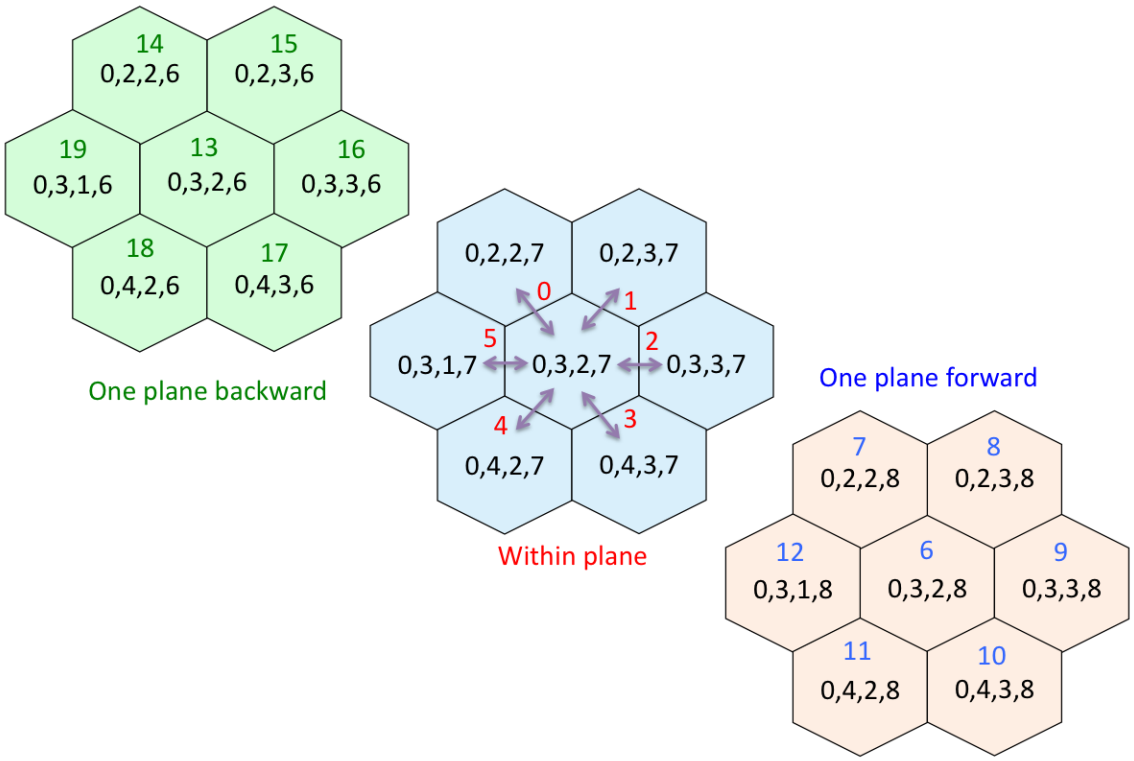

Figure 4: Hexagonal 3D nearest neighbors.

The colored numbers represent the gradients that must be calculated for a given voxel, the central one in the middle plane. This is just an example for a particular set of volume, row, column, and bucket numbers (V, R, C, B) representing the spatial position of these particular voxels. Once a set of voxels has been determined to be part of a trajectory, they are removed from the list of voxels. The remaining ones are sorted again and the process is repeated until all voxels have been used. In some cases noisy or dead electronics

can introduce spurious signals or drop some signals and intelligent algorithms attempt to account for this. Noisy channels will generally be isolated, so a threshold number of connected voxels can be used to discard these. Care must be taken, however, because in some cases, dead electronics may make it appear that there are isolated channels, or split a track in two pieces. The pattern recognition algorithm must be able to consider such effects and "jump over" dead voxels if necessary. There are other methods to handle special cases - and there are *always* special cases - which must also be included in a complete reconstruction algorithm.

Project description

For this project, you will construct a 3D hexagonal geometry edge finding algorithm that can combine individual voxels from NIFFTE events fission into connected trajectories. Once the trajectories have been determined, you will produce a 3D visualization of the events using Matplotlib's mplot3d, with the different trajectories color-coded. You will also keep track of how many tracks are found per event, how many voxels go unused or unconnected to a trajectory and generate histograms of these variables to quantify the performance of your algorithm. Here are a set of steps to help you get started:

Algorithm development:

- Write code to read in and store the data from the example file `niffte_data.txt` on PolyLearn. Its structure is

  ```
  #### Event <Nev> ### Ndigits <Ndig> ###
  <Volume> <Row> <Column> <Bucket> <ADC>
  <Volume> <Row> <Column> <Bucket> <ADC>
  ...
  ```

  The file contains 100 alpha particle events. Here, a "digit" is the same as a voxel. Volume, Row, Column and Bucket refer to the NIFFTE indexing convention as seen in Figure 4. ADC is a digitized unit of charge or energy deposition in the indexed location. Each event should be processed separately.

- Sort the voxels according to ADC value and compute the set of gradients for that voxel.

- Order the gradient voxels according to the gradient value (smallest to largest, since small gradient means more likely connected). If `(ADC-gradient) < cutoff` and the new voxel is estimated to be in the same direction as the others (forming a track), add the voxel to the track. Let `cutoff` be a parameter that you can vary. Start it off as some percentage of a typical ADC value.

- You can try to compute the direction of the track after adding a voxel by looking at the spatial averages of the voxels. Early in the development of the track this could be unreliable, but as a trajectory builds up this will become a better direction estimator.
- Once a voxel has been added to a track, let that one be the current voxel and repeat the gradient calculations to see if it should be added or not.
- After all voxels have been tried, determine the number of unique trajectories (groups of connected voxels) and number of unused or unconnected voxels for this event and store for later analysis.

Visualization:

- For each event, a 3D visualization of the data should be created. Use markers to represent the locations of the voxels in the detector volume and color code them by the trajectory they belong to.
- Allow orphaned voxels to be turned on/off in the visualization.
- Plot the number of occurrences of a particular number of reconstructed tracks per event.
- Plot the number of occurrences of a particular number of orphaned voxels per event.
- Use the visualization and the plots to "tune" your algorithm to minimize the number of orphans and reconstruct the correct number of tracks in each event. The event visualization will allow your human pattern recognition to be the judge of whether the algorithm does a good job or not.

How does your result depend on the cutoff value for the gradient? Compute the reconstruction efficiency for your algorithm for 100 events (i.e. how many tracks you found divided by the real number of tracks in the event). What additional tricks or tests could you add to your program to improve the efficiency?

What to turn in

Your project directory on github should contain the python script files that run your program and a documentation notebook that

1. Describes the program - what it does and how.
2. Shows how to run the program.
3. Plots the results from running the program.

4. Answers the questions included in the project description.

5. Includes references to any source material used to construct the program.

You should also include a markdown `README.md` file in the directory that has the title of your project, your name, the date, your academic honesty pledge and instructions to the user to load the notebook file to see more about the program.

Extensions

If you are really ambitious, you could also code the 3-dimensional straight line fits to determine the track direction vectors and lengths. It turns out that for straight lines in 3D, there is a set of exact equations that can be used to perform a 3D least-squares minimization of the set of data points that are supposed to form the line. The 3D geometry is a little complicated, though. If you do this, then you can evaluate the residuals of each space point to the line parameters (how far each point lies from the line). Optimizing the algorithm amounts to finding ways to minimize the width of the residual distributions.

In the visualization you could try to incorporate the NIFFTE geometry conversion from (V, R, C, B) to (x, y, z) and/or include the hexagonal geometry of the (R, C) plane.

Finally, try adding on "afterburners" to take care of any special cases, such as combining split tracks (two tracks found where one should have been), dealing with dead voxels that leave gaps, and handling tracks that cross or originate from the same vertex.