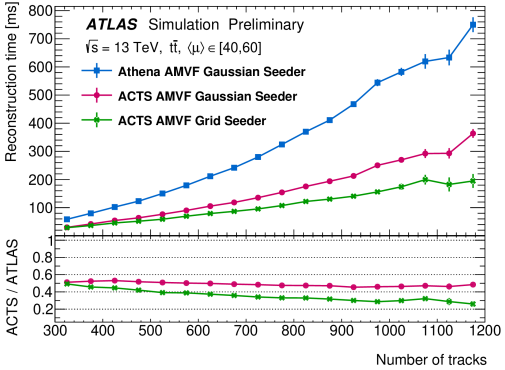
ACTS for clusterization

Louis-Guillaume Gagnon

ePIC Track reconstruction meeting
2023/03/16

ACTS: A Common Tracking Software project

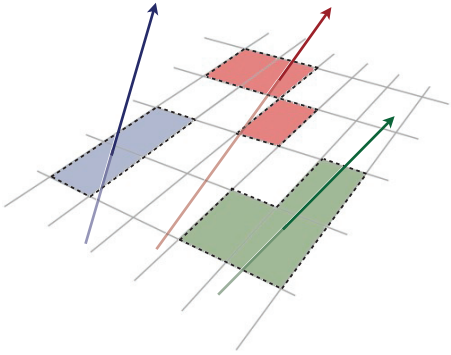
- Experiment-independent toolkit for tracking
- Free software (Mozilla Public License v2.0)
- Considered for use by Belle II, CEPC, sPHENIX, PANDA, FASER, ATLAS, EIC, ...
- Three overarching goals:
  1. Preserve current tracking approaches while enabling development for HL-LHC
  2. Serve as an algorithmic test bed including ML (ONNX) and accelerators (traccc)
  3. Enable rapid and realistic development of new tracking detectors
- Overview paper: [2106.13593]
- Project webpage: acts.readthedocs.io
- Code repository: github.com/acts-project/acts



- ACTS already used in ATLAS Run 3 vertexing
- Gains in reconstruction speed when compared to Run 2 algorithm!

Clusterization

- Clusterization $\equiv$ forming composite objects from discrete readout elements
- "Readout elements": e.g. Silicon pixel (2D) or strip (1D) sensors
- Based on geometric connectivity: elements laid out on N-dimensional grid
- Connection logic can be extended e.g. with timing information
- Depending on context, clusterization can also include task of estimating crossing position



Clusterization in ACTS

- Fast clusterization recently implemented in ACTS: Clusterization.hpp
- Based on Hoschen-Kopelman connected component labeling algorithm
- 2-pass algorithm:
  1. Assign labels to each sensor element and record equivalences ($\equiv$ connections)
  2. Re-assign a "canonical" label to each element using recorded equivalences
- Equivalences between labels efficiently tracked by disjoint-set datastructure
- Algorithm requires sensors laid-out on 1-D or 2-D grid
  - 3-D would be possible but not implemented
  - Non-geometrical dimensions supported through extensible `Connection` C++ type
- Only cluster formation is implemented; position estimation is user's responsibility
  - In general, initial position estimation requires knowledge of detector-specific readout geometry
  - Not available in ACTS detector geometry implementation
  - Precise position calibration requires knowledge of trajectory: Interface being designed

The Hoschen-Kopelman algorithm

- Input: A one-dimensional collection of “cells” (readout elements)
- Output: None, the algorithm simply assigns labels to the cells

1. Sort cells (Column-wise order used for 2-D cells)
2. Loop over cells in forward direction. For each cell:
   2.1 Assign a new label to the cell
   2.2 Loop backwards starting at nearest cell. For each cell pair:
      2.2.1 If they are more than one column away: break out of inner loop
      2.2.2 Else, if they are more than one row away: continue inner loop
      2.2.3 Else, they are connected: Mark their labels as equivalent and continue
3. Loop over cells in forward direction. For each cell:
   3.1 Replace its label by “canonical” label based on recorded equivalences

- After second pass, all cells in a cluster have the same label

Clusterization algorithm details

- First pass

```
// Sort cells by position to enable in-order scan
std::sort(cells.begin(), cells.end(), internal::Compare<Cell, GridDim>());

// First pass: Allocate labels and record equivalences
for (auto it = cells.begin(); it != cells.end(); ++it) {
  const internal::Connections<GridDim> seen =
      internal::getConnections<Cell, Connect, GridDim>(it, cells, connect);
  if (seen.nconn == 0) {
    // Allocate new label
    getCellLabel(*it) = ds.makeSet();
  } else {
    // Sanity check: first element should always have
    // label if nconn > 0
    if (seen.buf[0] == NO_LABEL) {
      throw std::logic_error("seen.nconn > 0 but seen.buf[0] == NO_LABEL");
    }

    // Record equivalences
    for (size_t i = 1; i < seen.nconn; i++) {
      // Sanity check: since connection lookup is always backward
      // while iteration is forward, all connected cells found here
      // should have a label
      if (seen.buf[i] == NO_LABEL) {
        throw std::logic_error("i < seen.nconn but see.buf[i] == NO_LABEL");
      }
      // Only record equivalence if needed
      if (seen.buf[0] != seen.buf[i]) {
        ds.unionSet(seen.buf[0], seen.buf[i]);
      }
    }
    // Set label for current cell
    getCellLabel(*it) = seen.buf[0];
  }
}
```

- Default connection logic

```
// Cell collection logic
template <typename Cell, typename Connect, size_t GridDim>
Connections<GridDim> getConnections(typename std::vector<Cell>::iterator it,
                                    std::vector<Cell>& set, Connect connect) {
  Connections<GridDim> seen;
  typename std::vector<Cell>::iterator it_2{it};

  while (it_2 != set.begin()) {
    it_2 = std::prev(it_2);

    ConnectResult cr = connect(*it, *it_2);
    if (cr == eNoConnStop) {
      break;
    }
    if (cr == eNoConn) {
      continue;
    }
    if (cr == eConn) {
      seen.buf[seen.nconn++] = getCellLabel(*it_2);
      if (seen.nconn == seen.buf.size()) {
        break;
      }
    }
  }
  return seen;
}
```

- Second pass

```
// Second pass: Merge labels based on recorded equivalences
for (auto& cell : cells) {
  Label& lbl = getCellLabel(cell);
  lbl = ds.findSet(lbl);
}
```

How to use? Cell labeling

- Algorithm does not require a specific "Cell" type
- Instead, algorithm is templated on cell type
- User needs to define three functions:
  - `int getCellColumn(const Cell&)`, → The primary sorting key
  - `int getCellRow(const Cell&)`, → The secondary sorting key (only if 2-D grid)
  - `int& getCellLabel(Cell&)` → Where the label gets stored
- This enables use of `labelClusters` function:

```
template <typename CellCollection, size_t GridDim = 2,
          typename Connect =
              DefaultConnect<typename CellCollection::value_type, GridDim>>
void labelClusters(CellCollection& cells, Connect connect = Connect());
```

- For 2-D grid with default connection (geometric-only), use is as simple as:

```
std::vector<Cell> cells = ...;
labelClusters(cells)
```

How to use? Cell connection

- Default connection is geometric only, based on adjacency
- Cell connection is extensible
- Example if want to use a pixel detector with timing information:

```
class MyCell;
struct MyConnect : Acts::Ccl::DefaultConnect<MyCell, 2> {
    ConnectResult operator()(const Cell& ref, const Cell& iter) {
        switch(Acts::Ccl::DefaultConnect<MyCell, 2>(ref, iter)) {
        case eNoConn:     return eNoConn;
        case eNoConnStop: return eNoConnStop;
        case eConn:       return timing_match(ref, iter)? eConn : eNoConn;
        }
    }
}
```

- Then:

```
std::vector<MyCell> cells = ...;
labelClusters<std::vector<MyCell>, 2, MyConnect>(cells, MyConnect());
```

How to use? Cluster creation

- After labelling, creating clusters based on labels is easy
- For convenience, ACTS provides `mergeClusters` function
- Again, not imposing a specific cluster or cell type
- User needs to provide one function:
  - `void clusterAddCell(Cluster&, const Cell&)`
  - the `Cluster` type needs a default constructor
- Usage is straigtforward:

```
std::vector<MyCell> cells = ...;
labelClusters(cells);
std::vector<MyCluster> clusters
  = mergeClusters<std::vector<MyCell>, std::vector<MyCluster>>(cells)
```

How to use? End to end

- In most use-cases, always chaining call to `labelClusters` and `mergeClusters`
- For convenience, ACTS provide a wrapper that does exactly this: `createClusters`
- Usage is straigtforward. E.g. In default scenario (2-D grid, default connectivity)

```
std::vector<MyCell> cells = ...;
std::vector<MyCluster> clusters= createClusters(cells)
```

Real world examples

- I gave some simple examples in previous slides
- Call signatures can get messier when deviate from default use case
- Can find real examples of the 1-D and 2-D cases in ATLAS Run 4 ACTS-based tracking:
  - 1-D: ActsStripClusteringTool
  - 2-D: ActsPixelClusteringTool
- Examples in ACTS itself:
  - In Digitization example: ModuleClusters.cpp
  - In test suite: Clusterization unit tests
- Otherwise, documentation is sorely lacking at this point
- Will be fixed Soon™
- I'm always happy to answer any questions

Performance Notes

- Unfortunately no plots to show: my benchmarking uses ATLAS MC, so has to stay internal...
- Implementation performs best when:
  - Detector modules are $\approx$ sparsely activated; or
  - if occupancy high, dominated by many small clusters
- In practice, these assumptions hold for overwhelming majority of ATLAS data
- Other experiments *may* have other requirements: let us know!
  - Optimal solution may be different in this case, and we want to support everyone

Conclusion

- ACTS has fast clusterization implementation based on Hoschen-Kopelman alogrithm
- Only cluster formation – position estimation for now left to users
  - In future, will have support for at least position calibration
- Gave a few examples of usage and references to real-world code
- Implementation performs best for sparse module and/or small-cluster regime
  - Let us know if that's not your use case!
- More documention will come at some point!