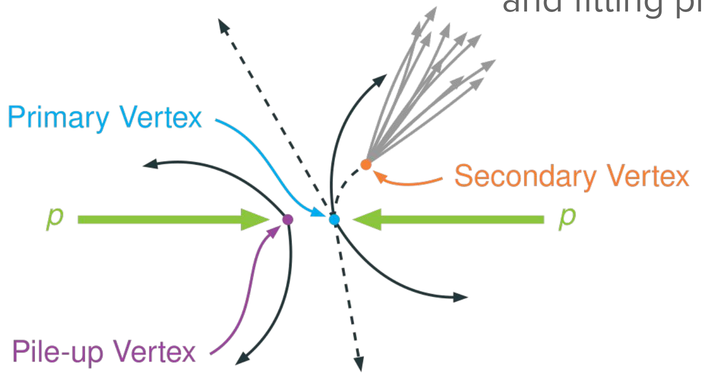
Vertexing

For ACTS Developers Workshop

Reminder: Objective

- Vertexing is the process of finding the tracks origins / interaction points and estimate their location
- It does not rely on any physics other than tracks ostensibly originating from a single point in space
- This is done with the input of reconstructed, fitted, and filtered tracks

- We distinguish between primary and secondary vertexing
  - ➢ Depending on the displacement of the vertex relative to the beamspot
  - ○ This correlates with the interaction being prompt or “delayed”, e.g. B meson decay
- Vertexing can be decomposed into a finding and fitting procedure



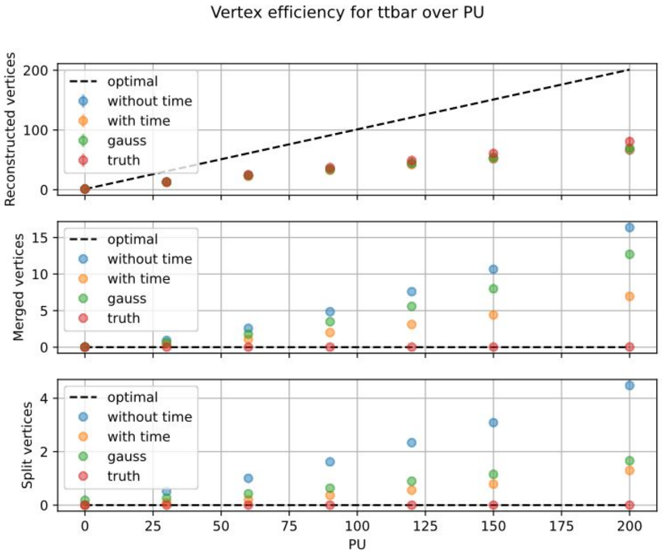
Reminder: Performance

Vertexing performance can be measured with various quantities, for example

- Finding efficiency
- Finding efficiency for hard-scatter vertex
- Technical efficiency
  - By trimming truth to reconstructible or reconstructed particles
- Resolution and pulls of the location estimates
- Merging, splitting, and fake rates
- Track contamination by mis-association

Similar to tracking performance, these quantities can be looked at over various other variables, for example

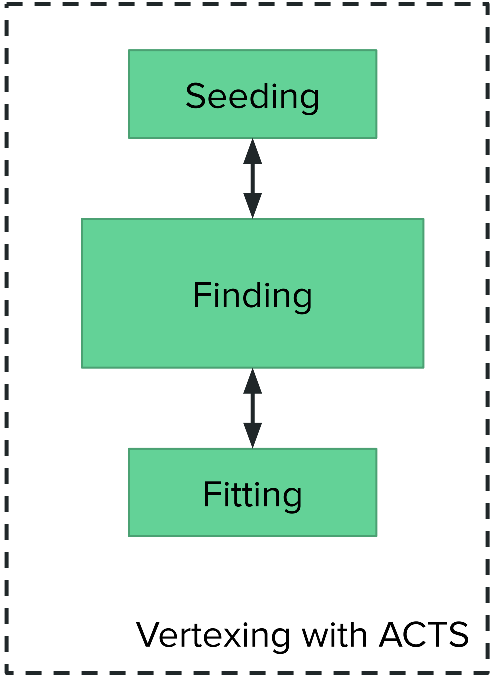
- Pile-up, pile-up density, pile-up contamination
- Number of reconstructed tracks from truth
- Truth sum pT squared



Vertexing with ACTS Core

Reminder: Vertexing with ACTS

- Vertexing is a long standing feature in ACTS
- ACTS primary vertexing is already used in Atlas Run 3
- We have two primary vertex finding algorithms:
  - Iterative Vertex Finder (IVF) [3]
  - Adaptive Multi Vertex Finder (AMVF) [3]
- There is no secondary vertex finding algorithm
- We have two general purpose fitting algorithms
  - Full-Billoir [1]
  - Kalman-based [2]
- Most of the vertexing components support time



Components

Interfaces
- IVertexFinder
- TrackLinearizer

Tools
- ImpactPointEstimator
- HelicalTrackLinearizer
- NumericalTrackLinearizer

EDM
- InputTrack
- Vertex
- TrackAtVertex
- LinearizedTrack

Primary vertex seeder
- TrackDensity
- GridDensity
- AdaptiveGridDensity

Primary vertex finder
- IVF
- AMVF

Vertex fitter
- Full-Billoir
- Kalman

Primary vertexing components

Concept	Core implementations			Examples implementations
Seeder	TrackDensity (w/o time)	GridDensity (w/o time)	AdaptiveGrid	Truth
Finder	IVF	AMVF		Truth
Fitter	Full-Billoir	Kalman		

Vertexing tools

HelicalTrackLinearizer

- Implementation of the `TrackLinearizer` interface for helical **and** straight tracks
- Direct calculation of jacobians via analytical formulas
- Trajectory is decomposed in circular and linear parts
  - Assumes XY is strictly the bending plane
- Was recently extended to 4D (credits PF, Felix)

`NumericalTrackLinearizer`

- Implementation of the `TrackLinearizer` interface w/o track shape constraints
- Developed by Felix to verify correct math and implementation of `HelicalTrackLinearizer`
- Does something similar to `RiddersPropagator`, but with differently defined global track parameters (x, y, z, time, phi, theta, q/p)
  - Use `Propagator` for nominal and small initial parameter delta propagations
  - Use difference quotient to get numerical derivatives

`ImpactPointEstimator`

- Versatile tool for vertexing algorithms

➢ Calculates smallest distance between track and vertex

➢ Estimation of impact parameters of track at vertex

➢ Track + vertex chi2 compatibility

- Quite math heavy but Felix wrote some nice derivations [5]

- **Note:** Some of this functionality assumes helical (or straight) tracks and uses iterative procedures (Newton) to minimize distances
    - Potentially some of this could be replaced by a `Propagator` with free track parameters and a `PointSurface`

Vertex fitting

FullBilloirVertexFitter

- Chi2 minimization without the need of inverting the full system matrix
- Fits vertex position and track momentum simultaneously
- Implemented as a `class`

```
/// @brief Fit method, fitting vertex for provided tracks with constraint
///
/// @param paramVector Vector of track objects to fit vertex to
/// @param vertexingOptions Vertexing options
/// @param fieldCache The magnetic field cache
///
/// @return Fitted vertex
Result<Vertex> fit(const std::vector<InputTrack>& paramVector,
                   const VertexingOptions& vertexingOptions,
                   MagneticFieldProvider::Cache& fieldCache) const;
```

KalmanVertexUpdater

- Fits vertex with track and vice versa iteratively
- Implemented as free functions

```
/// @brief Updates vertex with knowledge of new track
/// @note KalmanVertexUpdater updates the vertex when trk is added to the fit.
/// However, it does not add the track to the TrackAtVertex list. This to be
/// done manually after calling the method.
///
///
/// @param vtx Vertex to be updated
/// @param trk Track to be used for updating the vertex
/// @param nDimVertex number of dimensions of the vertex. Can be 3 (if we only
/// fit its spatial coordinates) or 4 (if we also fit time).
void updateVertexWithTrack(Vertex& vtx, TrackAtVertex& trk,
                           unsigned int nDimVertex);
```

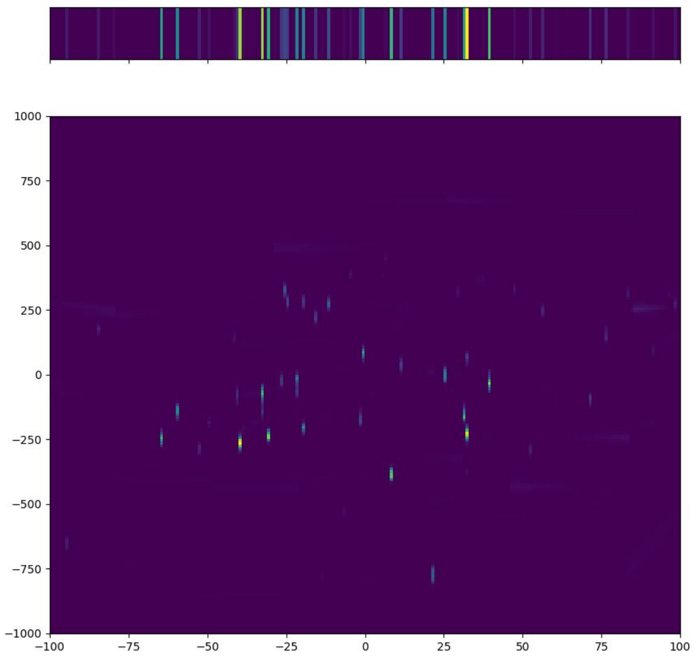
```
/// @brief Refits a single track with the knowledge of
/// the vertex it has originated from
///
/// @param track Track to update
/// @param vtx Vertex to use for updating the track
/// @param nDimVertex number of dimensions of the vertex. Can be 3 (if we only
/// fit its spatial coordinates) or 4 (if we also fit time).
void updateTrackWithVertex(TrackAtVertex& track, const Vertex& vtx,
                           unsigned int nDimVertex);
```

Primary vertex seeding

TrackDensityVertexFinder

- Analytical representation of track density using a sum of gaussians over all tracks [3]
- Maximization yields vertex seed and is done via Newton's method
- Displaced tracks are implicitly dampened with exponential of $d0$
- This algorithm does not support time as for now
  - Should be straightforward to add but no one did it so far
- Potentially slow for a lot of tracks as all the gaussians need to be evaluated for each iteration

w/o time



w/ time

w/o time

`GridDensityVertexFinder`

- Essentially the same as `TrackDensityVertexFinder`
- Uses a dense grid to evaluate the gaussians for single tracks and all tracks
- Grids for each track can be cached to allow removal from global grid
- Maximum search is essentially an `argmax` on an array
- This algorithm does not support time for now
    - Dense space+time grid would be very memory costly

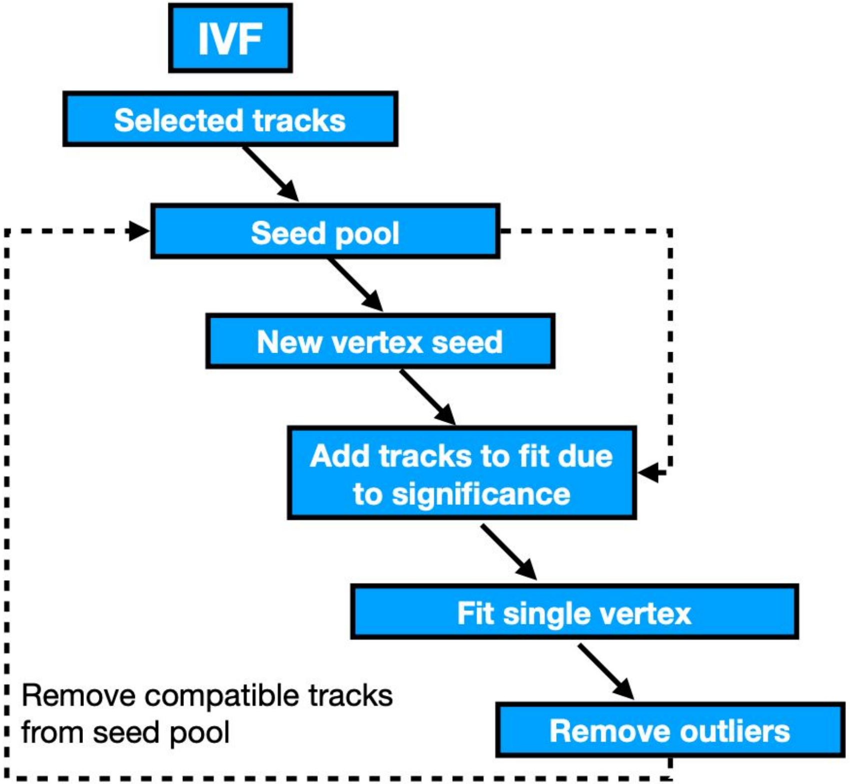
`AdaptiveGridDensityVertexFinder`

- Essentially the same as `GridDensityVertexFinder`
- Uses a sparse grid instead of a dense grid
- Supports time (needs to be toggled in the config)
- Track fills grid depending on its *`d0`* and *`time`* resolution
- Was significantly refactored and modified over the last year

Primary vertex finding

IterativeVertexFinder

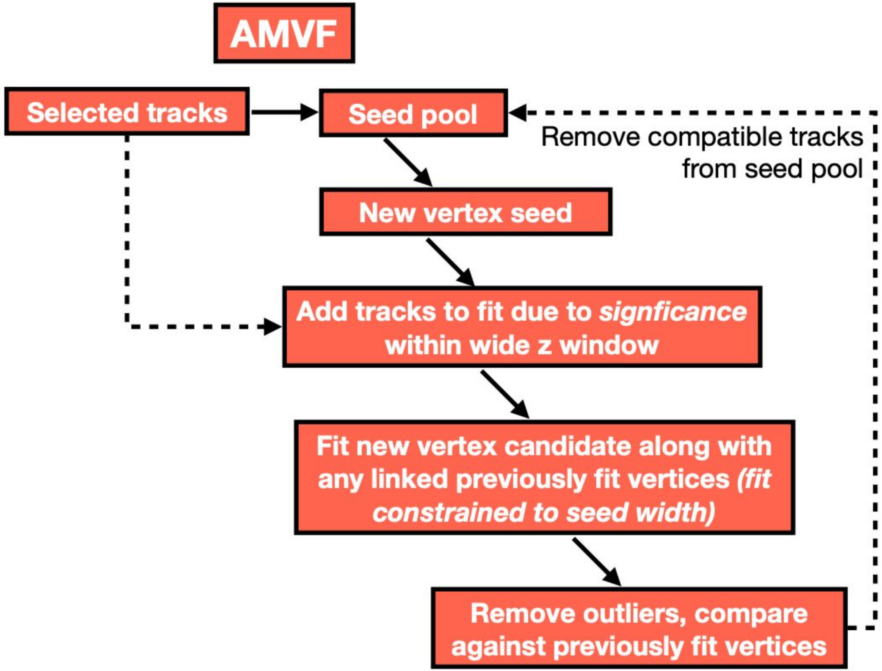
- Greedy vertex finder which selects tracks from vertex seeds
- Removes the tracks from the seeding pool after reconstructing a vertex
- Originally ported from Athena
  - Implementation potentially drifted away from original port
  - Never fully validated after Athena integration



[3]

AdaptiveMultiVertexFinder

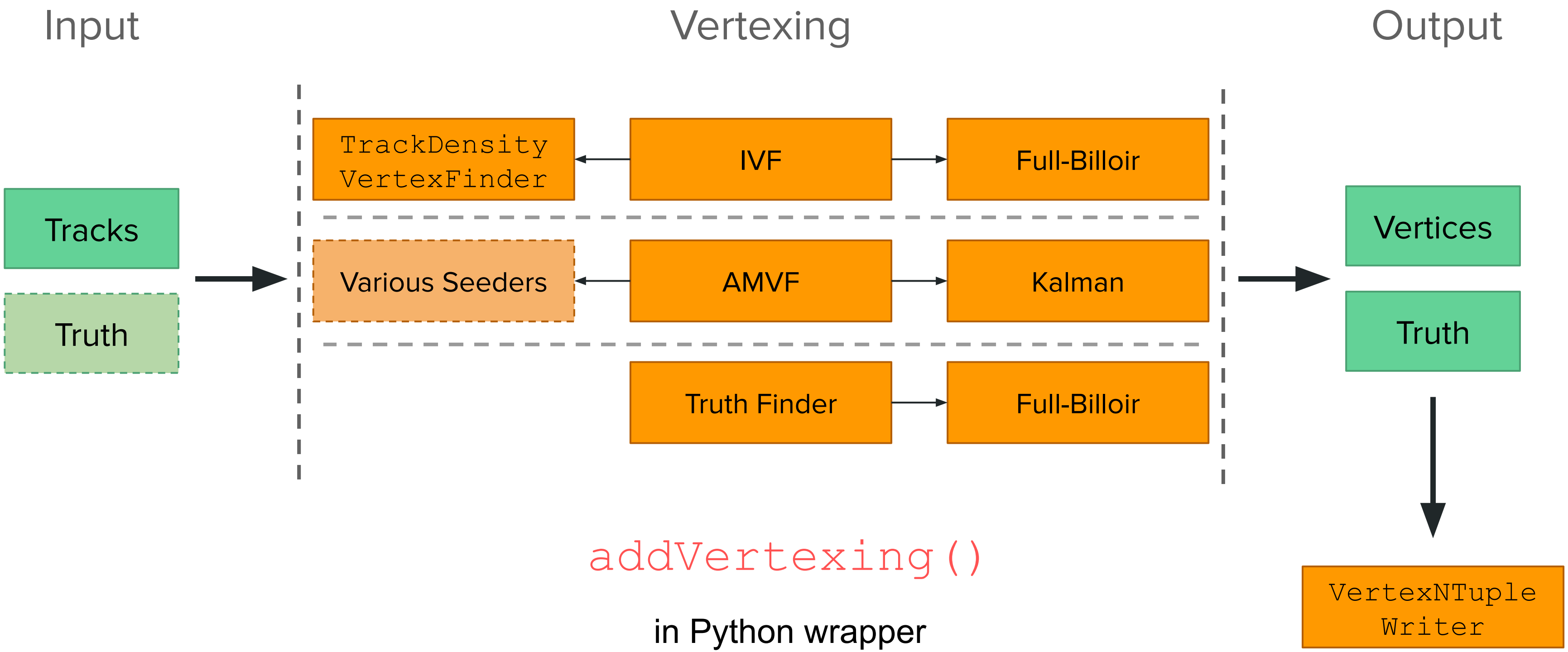
- Similar to IVF but a bit less greedy
- Vertices can share tracks, fit takes that into account
- Seeding done via seeding track pool but track selection done from all tracks
- Originally ported from Athena
  - Fully validated after Athena integration
  - Used in Run 3
  - Implementation potentially drifted away from original port



[3]

Vertexing in ACTS Examples

Vertexing in ACTS Examples



`addVertexing()`

in Python wrapper

`TruthVertexSeeder`

- Uses truth vertex position as vertex seed
- Allows to factorize performance by providing optimal seeding

➢ `TruthVertexFinder` is the perfect finder and leads to optimal fitting performance

➢ Especially with high pile-up, vertex finding is not a trivial task

➢ Vertex seeder can dominate the finding efficiency

- **Note:** Since IVF and AMVF are greedy, the order of provided seeds matters!

`VertexNTupleWriter`

- Former `VertexPerformanceWriter`
- Writes vertices, associated tracks, and matched truth to a ROOT TTree
- Was effectively rewritten over the last year
  - Existing code became unmaintainable as writing, and track and vertex truth matching was done in place with 5 levels of nested loops
- Does classification of clean, merged, split (inspired by ATLAS [3])

Summary

What changed?

Core

- Correct time handling for linearization and fitters (credits PF, Felix)
- De-templating (credits Paul)
    - Concrete extrapolator interface
    - Track EDM abstraction
    - Vertex finder interface
- Split compilation units for Kalman vertex fitter (credits Paul)

Examples

- Truth Vertex EDM / IO
- `VertexNTupleWriter`
    - Multiple iterations which resulted in a complete rewrite
    - Truth matching
    - Classification of clean, merged, split
- Truth seeding

What is missing?

Essential

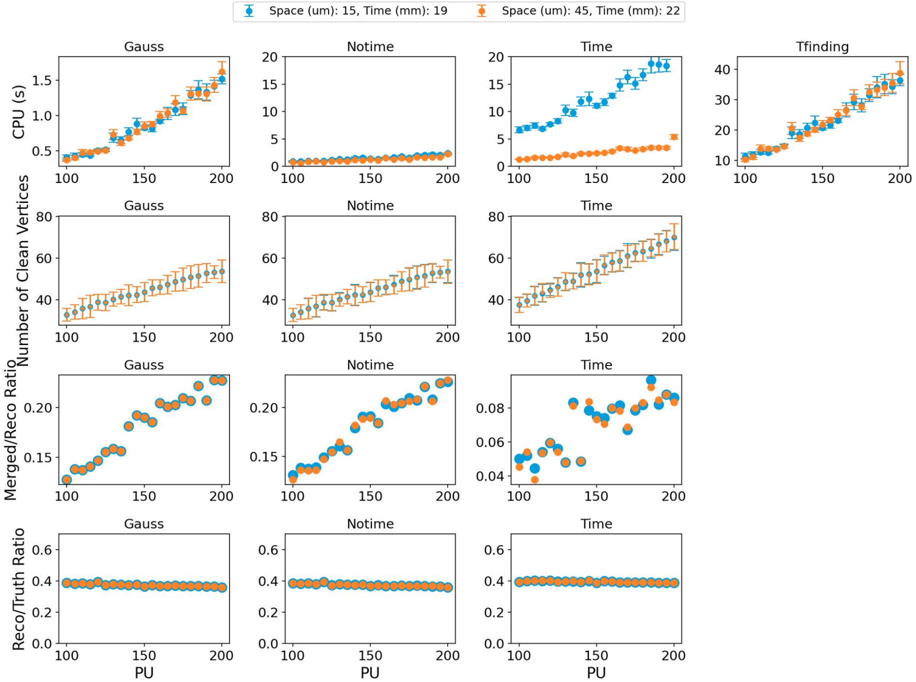
- Secondary vertex finding
- `TrackDensityVertexFinder` with time
- `PointSurface` instead of perigee

Nice to have

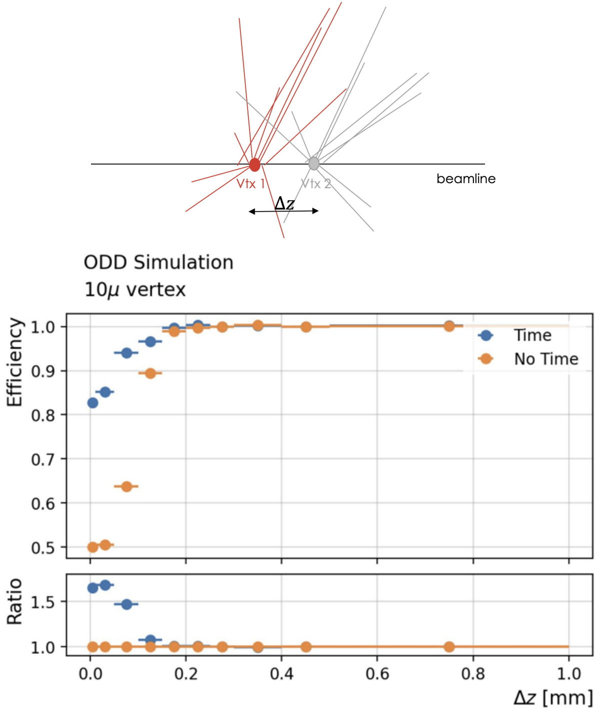
- Drop `LinIndices` and use `RiddersPropagator` for `NumericalTrackLinearizer`
- Vertex fitter interface
- Decouple vertex finders from fitters

Recent studies

- ODD ACTS computing performance of **Vertexing Finding** with time by Cléo Nicollin [4]



- ODD ACTS physics performance of **Vertex Finding / Fitting** with time



Conclusion

- ACTS Vertexing is a matured component with a lot of features
- Still no secondary vertex finding but tools in place to achieve that
- Time is handled in almost all components
- Code seems overall well tested with unit tests and physmon in place
- Felix left a lot of good documentation behind which goes beyond code and readthedocs [5] [6] [7]
- **Note:** Nobody in the ACTS community is actively working on vertexing / secondary vertexing

Resources

1. P. Billoir, S. Qian, Fast vertex fitting with a local parametrization of tracks
2. R. Frühwirth, Application of Kalman filtering to track and vertex fitting
3. ATL-PHYS-PUB-2019-015
4. https://indico.cern.ch/event/1435014/contributions/6038249/attachments/2902452/5090667/Poster2024-Nicollin.pdf
5. https://github.com/acts-project/acts/files/12794276/Track_Linearization_and_3D_PCA.pdf
6. https://acts.readthedocs.io/en/latest/white_papers/gaussian-track-densities.html
7. https://acts.readthedocs.io/en/latest/white_papers/billoir-covariances.html

Backup

Vertexing in ACTS Examples

- IVF is available via `IterativeVertexFinderAlgorithm`
  - Not really configurable
  - Seeder hardcoded to `TrackDensityVertexFinder`
- AMVF is available via `AdaptiveMultiVertexFinderAlgorithm`
  - Fairly configurable at this point
  - Seeder can be selected via enum
- `TruthVertexFinder` can be used with `VertexFitterAlgorithm`
  - Fitter hardcoded to `FullBilloirVertexFitter`
- `TruthVertexSeeder` can be used with AMVF

Vertexing EDM

`InputTrack`

- Non-owning, type-erased bound track parameter box type
- Allows for using concrete types in Core algorithms
- Implements `operator==`, `operator<`, `std::hash`
- `Extractor` delegate gets `BoundTrackParameters` from `InputTrack`

TrackAtVertex

- Captures information about a track, which was put on a vertex
- Does not point to the vertex
- Points to the original track
- Contains the eventual refitted track parameters at the vertex

```
/// Fitted perigee
BoundTrackParameters fittedParams;

/// Original input parameters
InputTrack originalParams;

/// Chi2 of track
double chi2Track = 0;

/// Number degrees of freedom
/// Note: Can be different from integer value
/// since annealing can result in effective
/// non-interger values
double ndf = 0;

/// Value of the compatibility of the track to the actual vertex, based
/// on the estimation of the 3d distance between the track and the vertex
double vertexCompatibility = 0;

/// Weight of track in fit
double trackWeight = 1;

/// The linearized state of the track at vertex
LinearizedTrack linearizedState;

/// Is already linearized
bool isLinearized = false;
```

`Vertex`

- Captures all the information about a reconstructed vertex
    - Position + covariance
    - Chi2 + degrees of freedom
    - Vector of `TrackAtVertex`
    - Seed position
- This is primarily meant to capture the final output of the vertexing
- **Note:** This class is also used for vertex seeds and final vertices

LinearizedTrack

- Fitters work with a linearized track model
- Simply a first order taylor expansion for the track parameter transport between reference surface and vertex position
- `LinearizedTrack` captures this expansion

```
/// @brief Constructor taking perigee parameters and covariance matrix
/// of track propagated to closest approach (PCA) of linearization point,
/// position and momentum Jacobian and const term.
///
/// @param paramsAtPCA Parameters at point of closest approach
/// @param parCovarianceAtPCA Parameter covariance matrix at point of closest
///                           approach
/// @param parWeightAtPCA The weight at the point of closest approach
/// @param linPoint Linearization point
/// @param posJacobian Position jacobian
/// @param momJacobian Momentum jacobian
/// @param position Position at point of closest approach
/// @param momentum Momentum at point of closest approach
/// @param constTerm Constant term in taylor expansion
LinearizedTrack(const BoundVector& paramsAtPCA,
                const BoundSquareMatrix& parCovarianceAtPCA,
                const BoundSquareMatrix& parWeightAtPCA,
                const Vector4& linPoint,
                const ActsMatrix<eBoundSize, 4>& posJacobian,
                const ActsMatrix<eBoundSize, 3>& momJacobian,
                const Vector4& position, const Vector3& momentum,
                const BoundVector& constTerm)
    : parametersAtPCA(paramsAtPCA),
      covarianceAtPCA(parCovarianceAtPCA),
      weightAtPCA(parWeightAtPCA),
      linearizationPoint(linPoint),
      positionJacobian(posJacobian),
      momentumJacobian(momJacobian),
      positionAtPCA(position),
      momentumAtPCA(momentum),
      constantTerm(constTerm) {}
```

LinIndices

- Parametrization for linearized tracks
- Note the difference to our usual `BoundIndices` and `FreeIndices`
  - `BoundIndices` use local position and phi, theta for direction
  - `FreeIndices` use global position and a unit vector for direction
- We somehow ended up with a third parametrization!

```
/// Enum to access the components of a track parameter vector.
///
/// Here, we parametrize the track via a 4D point on the track, the momentum
/// angles of the particle at that point, and q/p or 1/p.
///
/// @note It would make sense to rename these parameters if they are used outside of track linearization.
/// @note This must be a regular `enum` and not a scoped `enum class` to allow
/// implicit conversion to an integer. The enum value are thus visible directly
/// in `namespace Acts` and are prefixed to avoid naming collisions.
enum LinIndices : unsigned int {
  // Global spatial position of a point on the track, must be stored as one
  // continuous block.
  eLinPos0 = 0u,
  eLinPos1 = eLinPos0 + 1u,
  eLinPos2 = eLinPos0 + 2u,
  // Global time when particle is at the point
  eLinTime = 3u,
  // Angles of the particle momentum in the global frame at the point
  eLinPhi = 4u,
  eLinTheta = eLinPhi + 1u,
  // Global inverse-momentum-like parameter, i.e. q/p or 1/p, at the point
  // The naming is inconsistent for the case of neutral track parameters where
  // the value is interpreted as 1/p not as q/p. This is intentional to avoid
  // having multiple aliases for the same element and for lack of an acceptable
  // common name.
  eLinQOverP = 6u,
  // Total number of components
  eLinSize = 7u,
  // Number of space-time components (3+1)
  eLinPosSize = 4u,
  // Number of momentum components
  eLinMomSize = 3u,
};
```

Vertexing interfaces

TrackLinearizer

- Captures the creation of a `LinearizedTrack` from a track parameter at its reference surface (beamline) and a linearization point
- We have two implementations for this
  - `HelicalTrackLinearizer`
  - `NumericalTrackLinearizer`

```
using TrackLinearizer = Acts::Delegate<Result<LinearizedTrack>(
    const BoundTrackParameters& params, double linPointTime,
    const Surface& perigeeSurface, const Acts::GeometryContext& gctx,
    const Acts::MagneticFieldContext& mctx,
    MagneticFieldProvider::Cache& fieldCache)>;
```

IVertexFinder

- Interface for vertex finders **and** seeder
  - (Primary reason why finder and seeder use the same EDM)
- Type-erased `State` allows using virtual inheritance over templating while also keeping the usual interface convention (thread-safety argument)
- **Note:** seeders usually only return one `Vertex` while finders will return all

```cpp
/// Common interface for both vertex finders and vertex seed finders
class IVertexFinder {
 public:
  /// Type-erased wrapper for concrete state objects
  using State = Acts::AnyBase<128>;

  /// The main finder method that will return a set of found vertex candidates
  /// @param trackVector The input track collection
  /// @param vertexingOptions The vertexing options
  /// @param state The state object (needs to be created via @c makeState)
  /// @return The found vertex candidates
  virtual Result<std::vector<Vertex>> find(
      const std::vector<InputTrack>& trackVector,
      const VertexingOptions& vertexingOptions, State& state) const = 0;

  /// Function to create a state object for this concrete vertex finder
  /// @param mctx The magnetic field context
  /// @return The state object
  virtual State makeState(const MagneticFieldContext& mctx) const = 0;

  /// For vertex finders that have an internal state of active tracks, this
  /// method instructs them to mark used tracks for removal
  /// @param anyState The state object
  /// @param removedTracks The tracks to be removed
  virtual void setTracksToRemove(
      State& anyState, const std::vector<InputTrack>& removedTracks) const = 0;

  /// Virtual destructor
  virtual ~IVertexFinder() = default;
};
```