Progress in the radio module

Nikolaos Karastathis

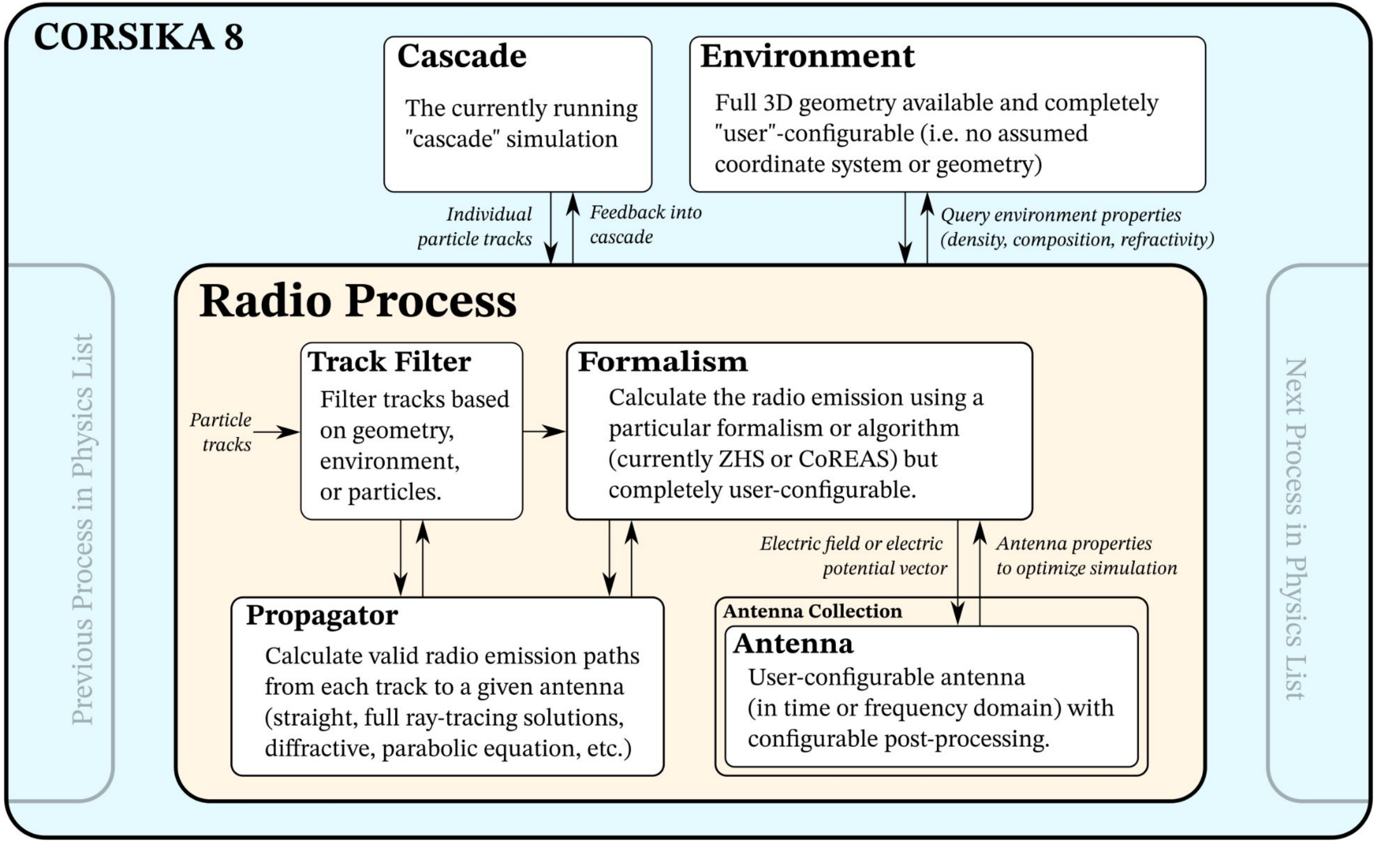
Outline

- Module design
- Develop your own propagator
- Propagators available right now
- Issues encountered recently

Module design

User-configurable parameters

- Filter
- Formalism
- Propagator
- Antenna



Propagators

- All propagators at the moment use the stray ray approximation
- Dummy Test Propagator
- Numerical Integrating Propagator
- Tabulated Flat Atmosphere Propagator

Propagator structure

1. Class declaration ⇨

```
template <typename TEnvironment>
class DummyTestPropagator final
    : public RadioPropagator<DummyTestPropagator<TEnvironment>, TEnvironment>
```

2. Constructor ⇨

```
DummyTestPropagator(TEnvironment const& env);
```

3. Propagate method ⇨

```
template <typename Particle>
SignalPathCollection propagate(Particle const& particle, Point const& source, Point const& destination);
```

4. Call function ⇨

```
template <typename TEnvironment>
  DummyTestPropagator<TEnvironment>
make_dummy_test_radio_propagator(TEnvironment const& env){
    return DummyTestPropagator<TEnvironment>(env);
```

Propagator structure

Returns a signal path collection. Each signal path consists of:

```
SignalPath(TimeType const propagation_time, double const average_refractive_index,
           double const refractive_index_source,
           double const refractive_index_destination,
           Vector<dimensionless_d> const& emit,
           Vector<dimensionless_d> const& receive, LengthType const R_distance,
           std::deque<Point> const& points);
```

Dummy Test Propagator

- Intended for fast simulations, tests and specific cases when a uniform refractive index is being used
- Used in unit tests, synchrotron radiation and clover leaf example
- Calculates the propagation time between a point in the shower and the antenna position using only 2 points and the straight ray approximation

```
// these are used for the direction of emission and reception of signal at the antenna
auto const emit_{(destination - source).normalized()};
auto const receive_{-emit_};

// the geometrical distance from the point of emission to an observer
auto const distance_{(destination - source).getNorm()};

// get the universe for this environment
auto const* const universe{Base::env_.getUniverse().get()};

// clear the refractive index vector and points deque for this signal propagation.
rindex.clear();
points.clear();

// get and store the refractive index of the first point 'source'.
// auto const* const nodeSource{universe->getContainingNode(source)};
auto const* const nodeSource{particle.getNode()};
auto const ri_source{nodeSource->getModelProperties().getRefractiveIndex(source)};
rindex.push_back(ri_source);
points.push_back(source);

// add the refractive index of last point 'destination' and store it.
auto const* const node{universe->getContainingNode(destination)};
auto const ri_destination{node->getModelProperties().getRefractiveIndex(destination)};
rindex.push_back(ri_destination);
points.push_back(destination);

// compute the average refractive index.
auto const averageRefractiveIndex_ = (ri_source + ri_destination) * 0.5;

// compute the total time delay.
TimeType const time = averageRefractiveIndex_ * (distance_ / constants::c);

return std::vector<SignalPath>(
    1, SignalPath(time, averageRefractiveIndex_, ri_source, ri_destination, emit_,
                  receive_, distance_, points));
```

Numerical Integrating Propagator

- Uses tweaked Simpson's rule to calculate the signal propagation time
- Is slow and is not recommended for simulations
- User can provide stepsize in the constructor

```
// Apply the standard Simpson's rule
auto const h = ((destination - source).getNorm()) / (N - 1);

for (std::size_t index = 1; index < (N - 1); index += 2) {
  sum += 4 * rindex.at(index);
  refra += rindex.at(index);
}
for (std::size_t index = 2; index < (N - 1); index += 2) {
  sum += 2 * rindex.at(index);
  refra += rindex.at(index);
}
index = N - 1;
sum = sum + rindex.at(index);
refra += rindex.at(index);

// compute the total time delay.
time = sum * (h / (3 * constants::c));
```

Tabulated Flat Atmosphere Propagator

- Works well with Gladstone Dale law refractive index profile
- Given 2 points and a step it creates a table for refractivity and integrated refractivity between upper limit and lower limit - 1km
- Propagate method checks where the "source" particle wrt the table indices and calculates propagation time accordingly
- Above maximum height (leaving the atmosphere even) or below ground (below lower limit) an interpolation is being performed, otherwise simply find the index

```
template <typename TEnvironment>
TabulatedFlatAtmospherePropagator<TEnvironment>
make_tabulated_flat_atmosphere_radio_propagator(TEnvironment const& env, Point const& upperLimit,
                                                Point const& lowerLimit, LengthType const step){
  return TabulatedFlatAtmospherePropagator<TEnvironment>(env, upperLimit, lowerLimit, step);
}
```

```
if ((sourceHeight_ + 0.5) >= lastElement_) { // source particle is above maximum height
```

```
else if ((sourceHeight_ + 0.5 < lastElement_) && sourceHeight_ > 0) { // source particle in the table
```

```
else if (sourceHeight_ == 0) { // source particle is exactly at the lower edge of the table
```

```
else if (sourceHeight_ < 0) { // source particle is in the ground.
```

Conceptual change

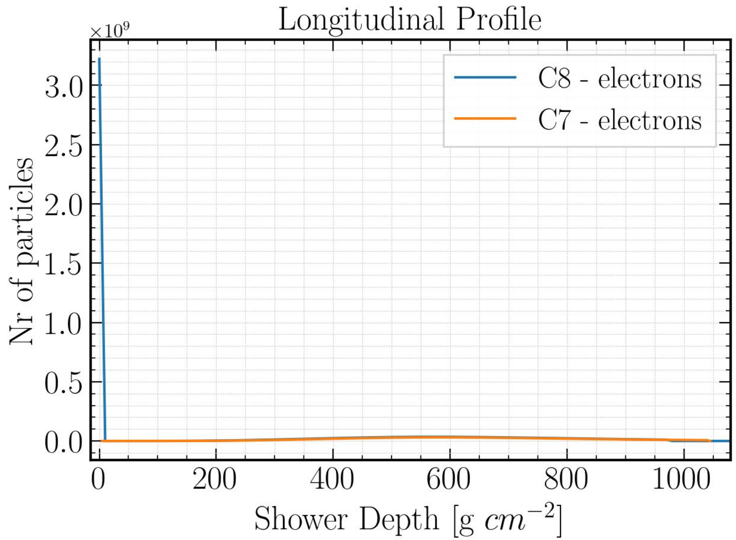
```
template <typename Particle>
SignalPathCollection propagate(Particle const& particle, Point const& source, Point const& destination);
```

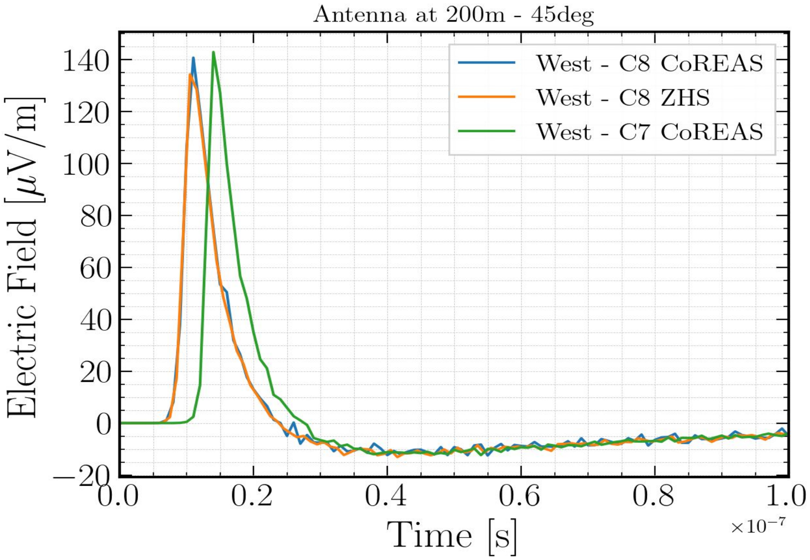
```
// get and store the refractive index of the first point 'source'
auto const* const nodeSource{universe->getContainingNode(source)};
auto const ri_source{nodeSource->getModelProperties().getRefractiveIndex(source)};
```

```
auto const* const nodeSource{particle.getNode()};
auto const ri_source{nodeSource->getModelProperties().getRefractiveIndex(source)};
```

Problems

- Time offset
- Propagator's issue?
- Tachyons?





Easy Interface

Harmonization of interfaces between ordinary radio and multithreaded radio branch

```
auto propagator = make_simple_radio_propagator(enviroment);
auto coreas     = make_radio_process_CoREAS(detector, propagator, nthreads);
auto    zhs     = make_radio_process_ZHS(detector, propagator, nthreads);
```

Thank you!