The radio module in CORSIKA 8

Nikolaos Karastathis, Remy Prechelt, Juan Ammerman-Yebra and Tim Huege

for the CORSIKA 8 collaboration

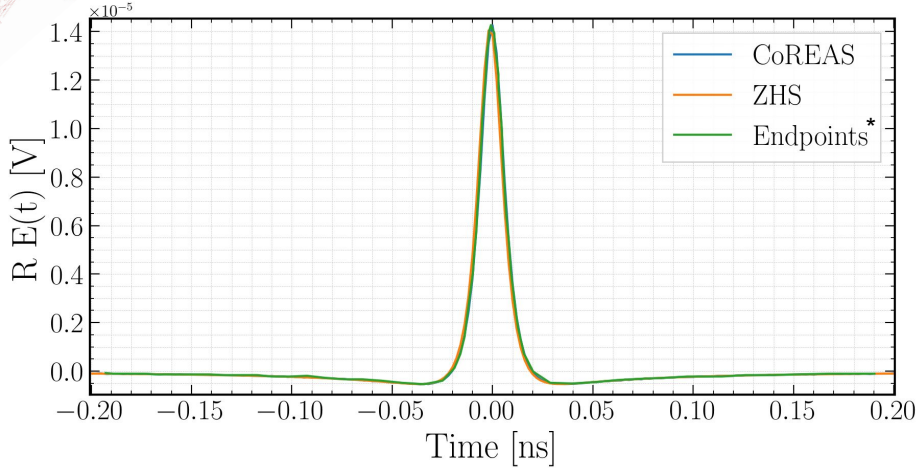
Outline

- Status and results of the radio module
- Current problems
- Design and implementation
- Examples and future development

Status and results of the radio module

(the radio branch is under code review and is to be merged to master very soon)

Electron in a uniform magnetic field

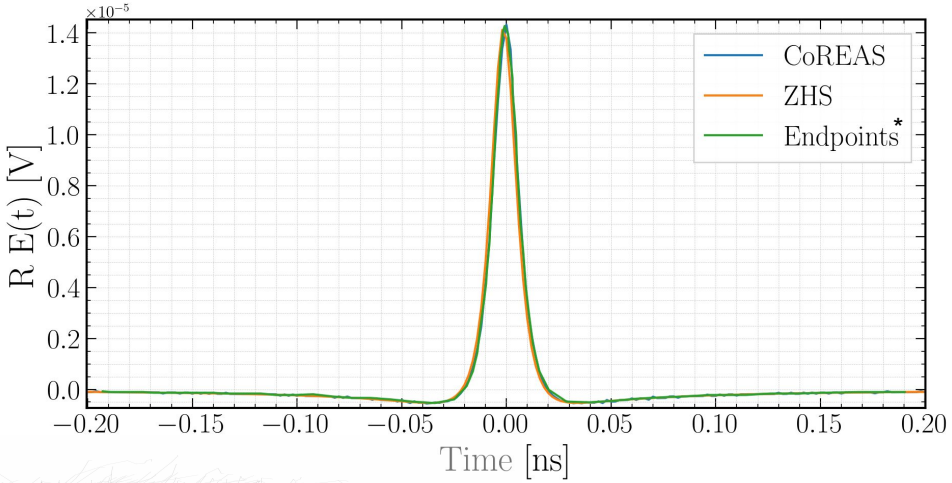


→ CORSIKA 8 tracking algorithm

Used C8's LeapFrog magnetic field tracking algorithm. Created a suitable environment with the corresponding values for magnetic field and gyrofrequency of the relativistic electron.

→ Manual tracking algorithm

100.000 points on a circle (L = 100m) connected by straight track segments. The relativistic electron of fixed energy, is allowed to travel on these tracks.



* C.W. James, H. Falcke, T. Huege and M. Ludwig, doi: 10.1103/PhysRevE.84.056602

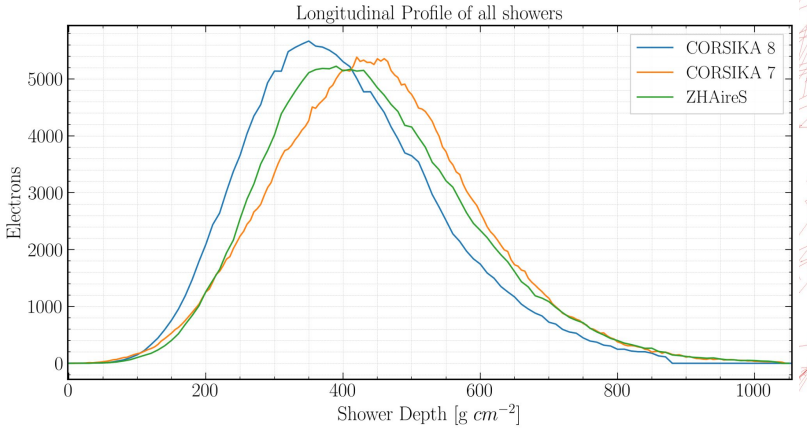
Simulation of an extensive air shower

- 10 TeV electron-induced vertical shower
- US Standard atmosphere – uniform refractive index (n=1.000327)
- Horizontal geomagnetic field B = 50 μT aligned in the x direction
- $X_{\max} \approx 390\, g\, cm^{-2}$
- Star-shaped grid of antennas – 20 concentric rings (25m - 500m)

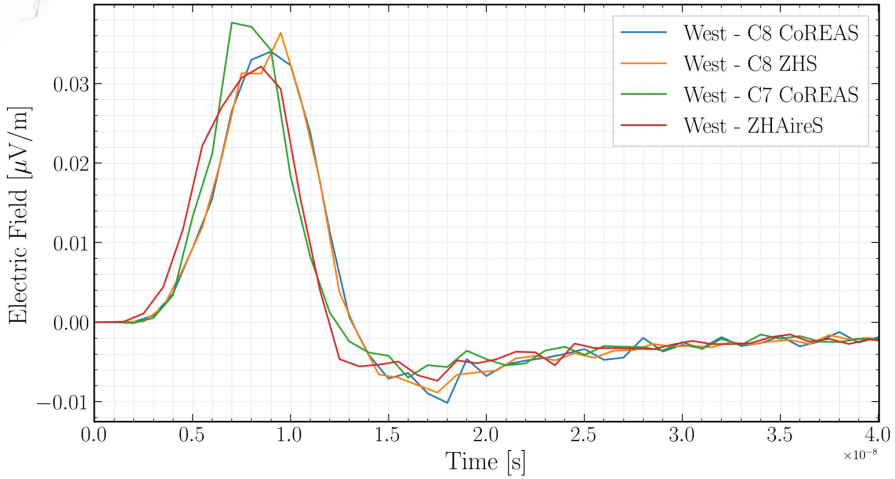
Comparison

- CORSIKA 8 – CoREAS
- CORSIKA 8 – ZHS
- CORSIKA 7 – CoREAS
- ZHAireS

PROPOSAL v. 7.3.1 for C8 simulations



Pulses comparison

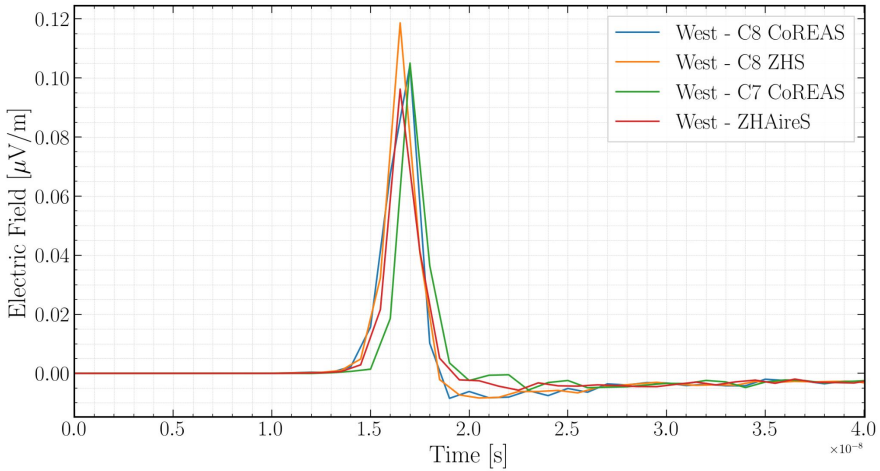


Antenna at 50m from the shower core

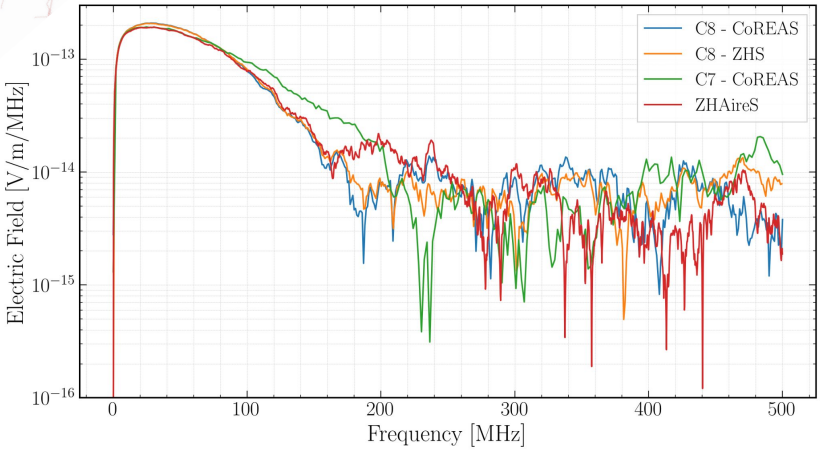
- Very good agreement between C8 (both formalisms) and C7 in pulse amplitude
- ZHAireS seems slightly off

Antenna at 200m from the shower core

- Very good agreement between C7, ZHAireS and C8 CoREAS in pulse amplitude
- C8 ZHS is a bit higher



Frequency spectra comparison

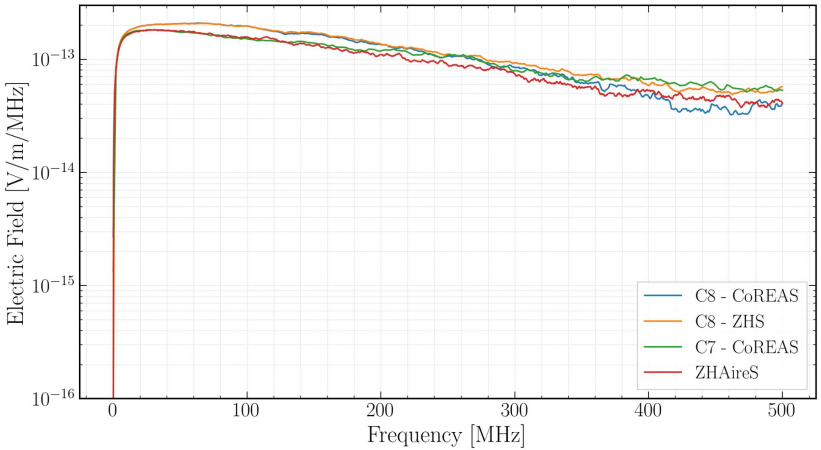


Antenna at 50m from the shower core

- Increase in power below 80 MHz for C8 spectra
- Increase in power between 300 and 400 MHz for C8 spectra
- C8 CoREAS and C8 ZHS diverge after 420 MHz
- Overall decent agreement

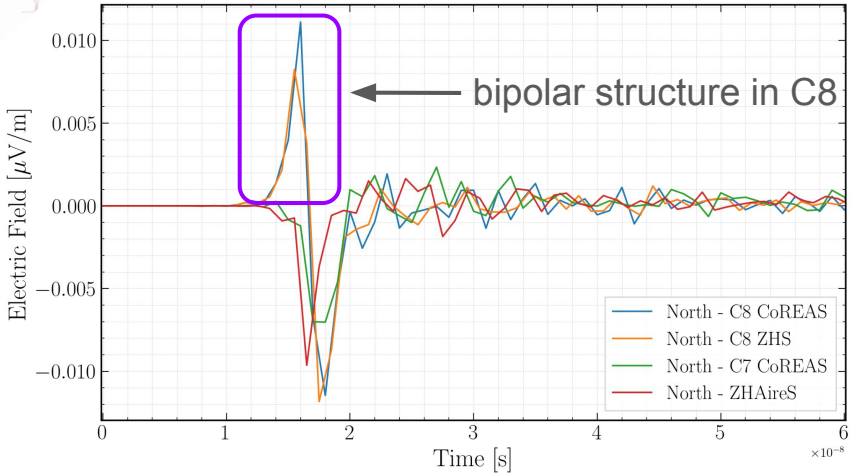
Antenna at 200m from the shower core

- Slight increase in power below 200 MHz for C8
- C8 CoREAS and C8 ZHS diverge slightly after 200 MHz
- Overall decent agreement



Current problems

Polarization properties

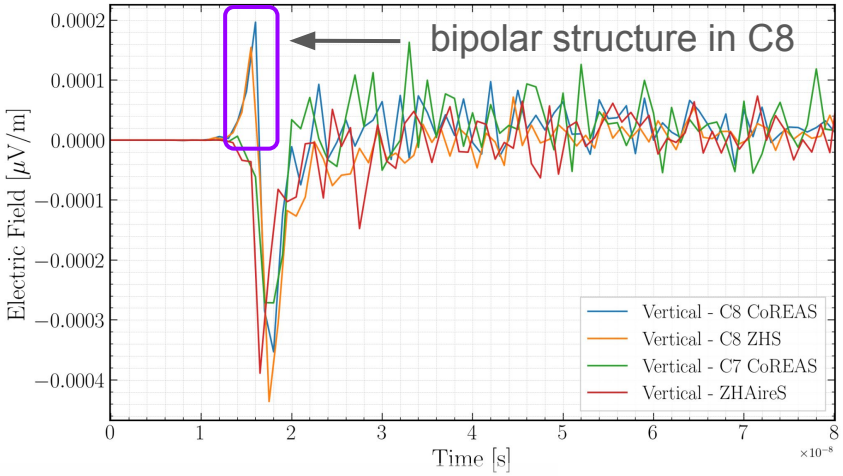


Antenna at 200m from the shower core

- Polarization behaviour matches
- C8 pulses have a bipolar structure instead of a unipolar pulse
- This affects the symmetry

Antenna at 200m from the shower core

- Polarization behaviour matches
- C8 pulses have a bipolar structure instead of a unipolar pulse
- This affects the symmetry



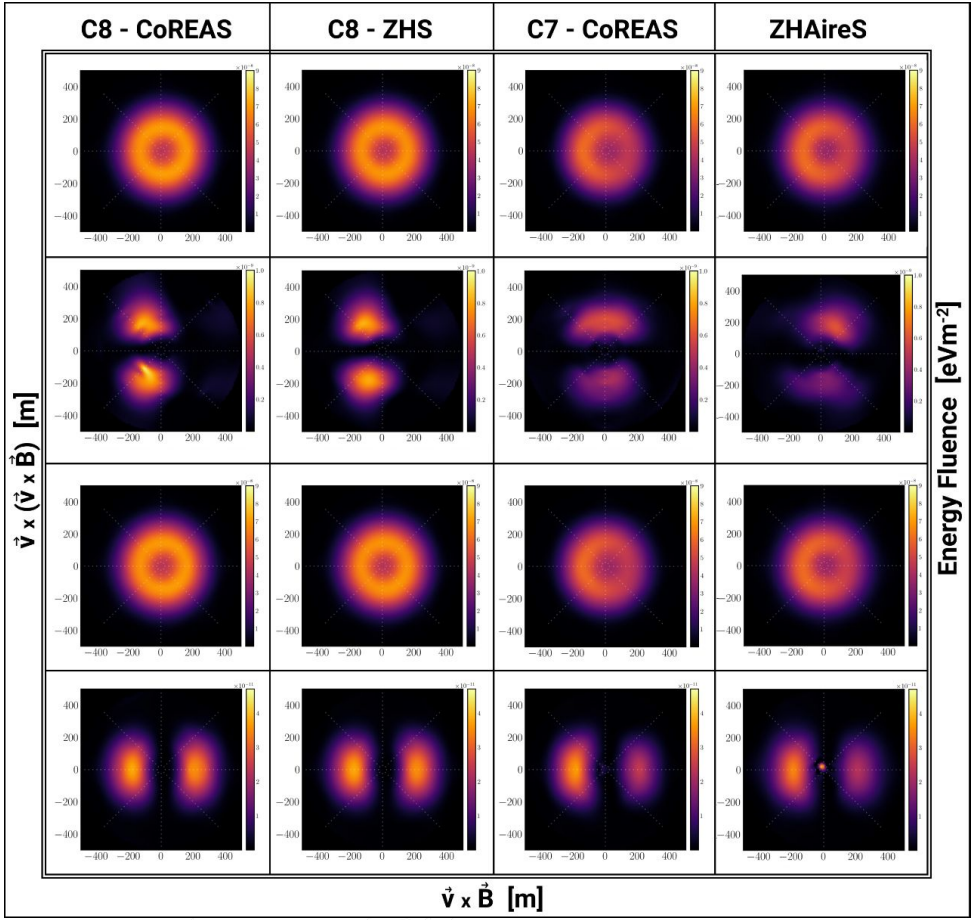
Energy fluence 2D maps (30 - 80 MHz)

All polarizations

$\vec{v} \times \left(\vec{v} \times \vec{B}\right)$

$\vec{v} \times \vec{B}$

$\vec{v}$



Cascade bug

- This analysis was made with applying the Cascade.inl “quick fix”, in order to avoid completely neglecting multiple scattering and messing ”less” with particle direction.

More details on:
MR426, issue 482, issue 483

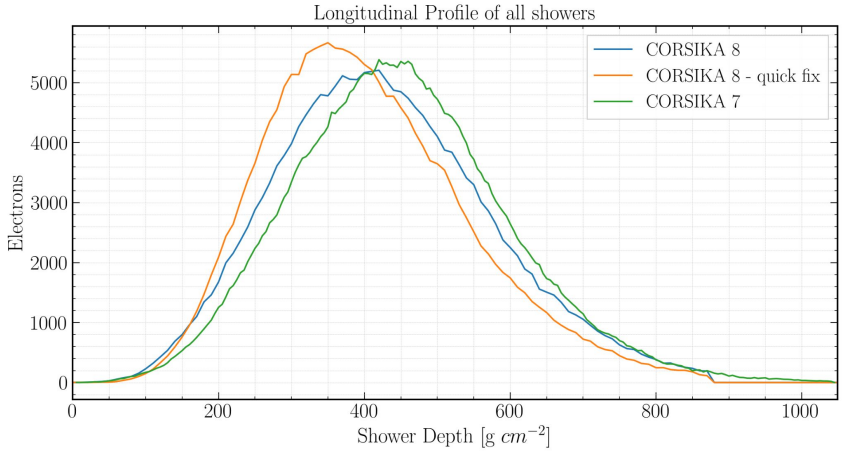
```
// apply all continuous processes on particle + track
if (sequence_.doContinuous(particle, step, limitingId) ==
    ProcessReturn::ParticleAbsorbed) {
  CORSIKA_LOG_DEBUG("Cascade: delete absorbed particle PID={} E={} GeV",
                    particle.getPID(), particle.getEnergy() / 1_GeV);
  if (particle.isErased()) {
    CORSIKA_LOG_WARN(
        "Particle marked as Absorbed in doContinuous, but prematurely erased. This "
        "may be bug. Check.");
  } else {
    particle.erase();
  }
  return; // particle is gone -> return
}
particle.setPosition(step.getPosition(1));
particle.setDirection(step.getDirection(1));
particle.setTime(particle.getTime() + step.getDuration());

if (isContinuous) {
  return; // there is nothing further, step is finished
}
```

“quick fix”

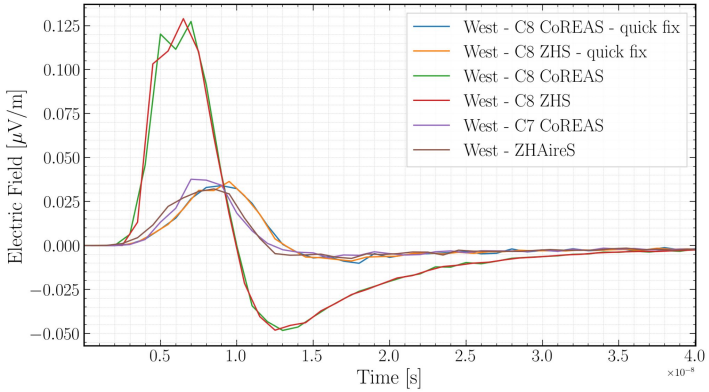
```
particle.setPosition(step.getPosition(1));
particle.setDirection(step.getDirection(1));

// apply all continuous processes on particle + track
if (sequence_.doContinuous(particle, step, limitingId) ==
    ProcessReturn::ParticleAbsorbed) {
  CORSIKA_LOG_DEBUG("Cascade: delete absorbed particle PID={} E={} GeV",
                    particle.getPID(), particle.getEnergy() / 1_GeV);
  if (particle.isErased()) {
    CORSIKA_LOG_WARN(
        "Particle marked as Absorbed in doContinuous, but prematurely erased. This "
        "may be bug. Check.");
  } else {
    particle.erase();
  }
  return; // particle is gone -> return
}
particle.setTime(particle.getTime() + step.getDuration());

if (isContinuous) {
  return; // there is nothing further, step is finished
}
```

Profiles & Pulses

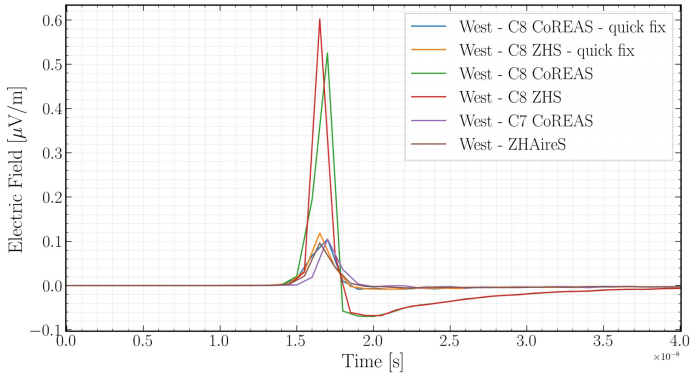
- more detailed profile studies were done in Jean-Marco's talk

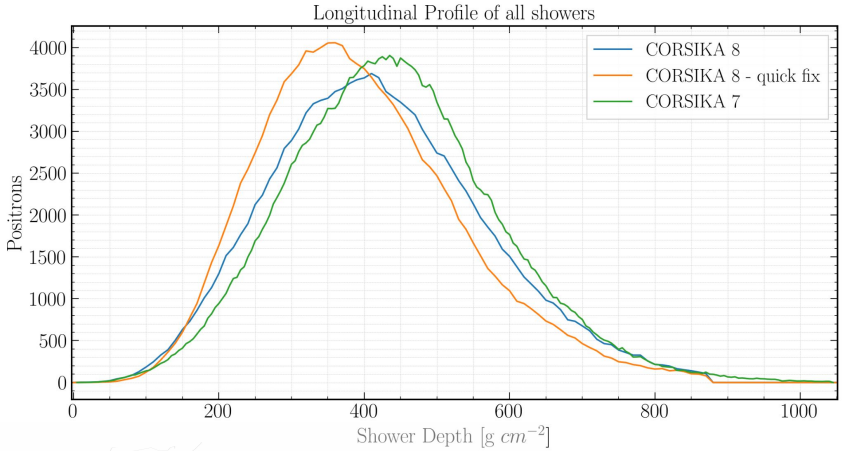


Antenna at 50m



Antenna at 200m





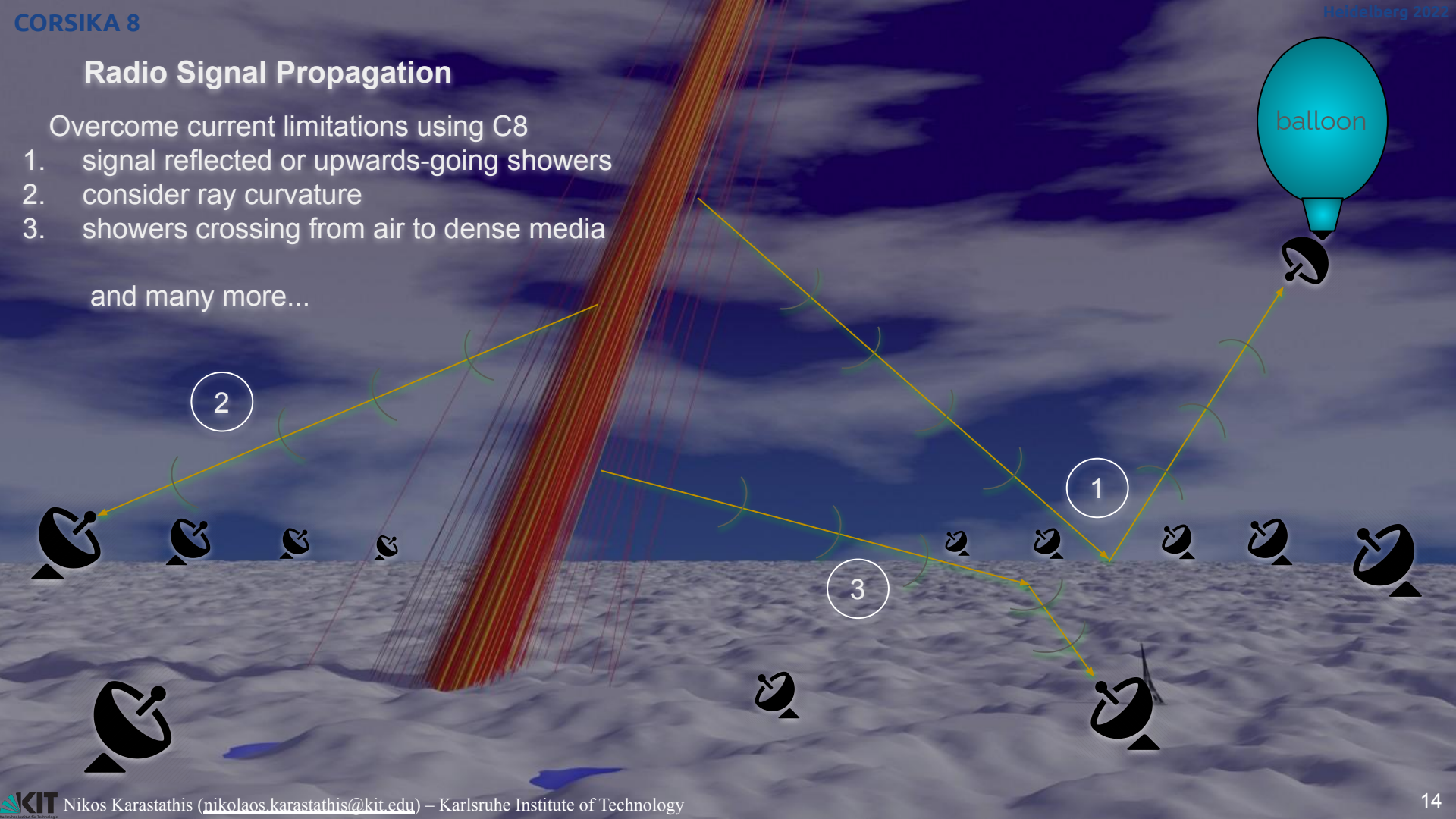
Design and implementation

Radio Signal Propagation

Overcome current limitations using C8

1. signal reflected or upwards-going showers
2. consider ray curvature
3. showers crossing from air to dense media

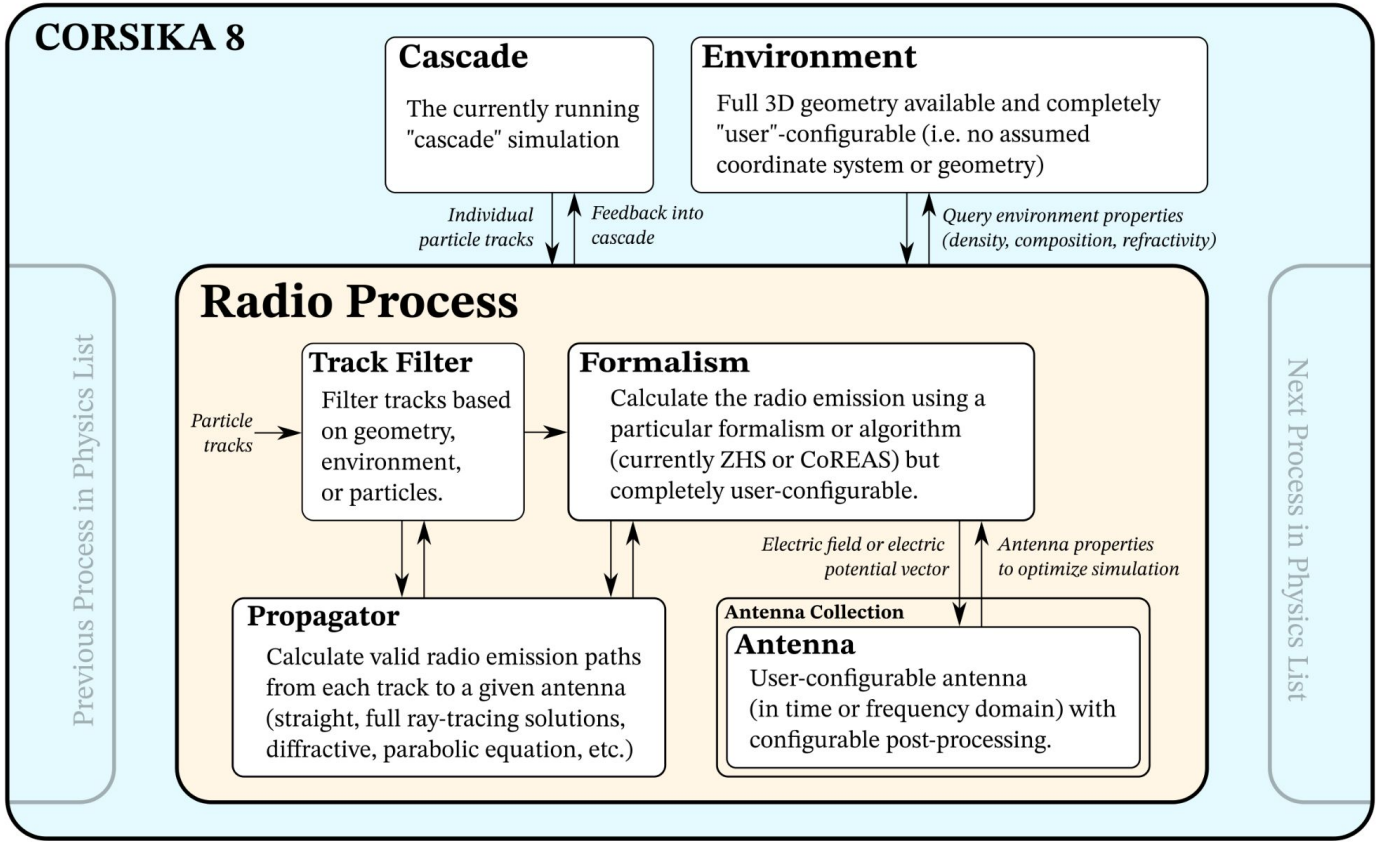
and many more...



Radio module architecture

User-configurable parameters

- Filter
- Formalism
- Propagator
- Antenna



Examples and future development

Usage example

1. Create an antenna collection object 

```
AntennaCollection<TimeDomainAntenna> detectorCoREAS;
```

2. Create any pattern of antennas you wish and put it in the antenna collection
   – In this case a star shaped pattern

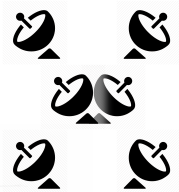
```
// // setup CoREAS antennas
for (auto radius_1 = 25_m; radius_1 <= 500_m; radius_1 += 25_m) {
  for (auto phi_1 = 0; phi_1 <= 315; phi_1 += 45) {
    auto phiRad_1 = phi_1 / 180. * M_PI;
    auto rr_1 = static_cast<int>(radius_1 / 1_m);
    auto const point_1{Point(rootCS, showerCoreX_ + radius_1 * cos(phiRad_1),
                             showerCoreY_ + radius_1 * sin(phiRad_1),
                             constants::EarthRadius::Mean)};
    std::cout << "Antenna point: " << point_1 << std::endl;
    auto triggertime_1{(triggerpoint_ - point_1).getNorm() / constants::c};
    std::string name_1 = "CoREAS_R=" + std::to_string(rr_1) +
                         "_m--Phi=" + std::to_string(phi_1) + "degrees";
    TimeDomainAntenna antenna_1(name_1, point_1, rootCS, triggertime_1, duration_, sampleRate_,
                                triggertime_1);
    detectorCoREAS.addAntenna(antenna_1);
  }
}
```

3. Create a radio process using the antenna collection, the algorithm and the propagator you want

```
// initiate CoREAS
RadioProcess<decltype(detectorCoREAS),
             CoREAS<decltype(detectorCoREAS), decltype(SimplePropagator(env))>,
             decltype(SimplePropagator(env))>
    coreas(detectorCoREAS, env);

// register CoREAS with the output manager
output.add("CoREAS", coreas);
```

4. Put the radio process in the process sequence

```
auto sequence = make_sequence(emCascade, emContinuous, longprof, cut, coreas,
                              observationLevel, tracks);
```



A simple propagator

```
namespace corsika {

  template <typename TEnvironment>
  inline SimplePropagator<TEnvironment>::SimplePropagator(TEnvironment const& env)
      : RadioPropagator<SimplePropagator, TEnvironment>(env){};

  template <typename TEnvironment>
  inline typename SimplePropagator<TEnvironment>::SignalPathCollection
  SimplePropagator<TEnvironment>::propagate(Point const& source, Point const& destination,
                                            LengthType const stepsize) const {

    /**
     * This is the simplest case of straight propagator
     * where no integration takes place.
     * This can be used for fast tests and checks of the radio module.
     *
     */

    // 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()};
```

```
    // the points that consist the signal path (source & destination).
    std::deque<Point> points;

    // store value of the refractive index at points.
    std::vector<double> rindex;
    rindex.reserve(2);

    // 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)};
    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) / 2;

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

    return {SignalPath(time, averageRefractiveIndex_, ri_source, ri_destination, emit_,
                       receive_, distance_, points)};

  } // END: propagate()

} // namespace corsika
```

Future development

- Merge radio branch once the code review is done
- Implement new propagators for more sophisticated signal propagation scenarios (see Juan's talk)
- Implement a faster propagator - probably tabulated for a fixed standard environment
- Implement more advanced filters

Thank you!

Nikos Karastathis (nikolaos.karastathis@kit.edu) – Karlsruhe Institute of Technology