# Discussion on the pulse-building algorithm

**Minho Kim**
Argonne National Laboratory

**BIC Simulation Meeting**
July 28, 2025

Argonne NATIONAL LABORATORY

# Time sorting

```
for (const auto& sh : *simhits) {
  // Fill the contributions to be sorted by time
  std::vector<const edm4hep::CaloHitContribution*> contribs;
  for (const auto& contrib : sh.getContributions()) {
    contribs.push_back(&contrib);
  }

  // Sort the contributions by time.
  std::sort(contribs.begin(), contribs.end(),
            [](const auto* a, const auto* b) { return a->getTime() < b->getTime(); });

  // Group the contributions that are close in time.
  std::vector<std::vector<const edm4hep::CaloHitContribution*>> contrib_clusters;
  for (const auto& contrib : contribs) {
    if (contrib_clusters.empty())
      contrib_clusters.push_back({contrib});
    else {
      const auto* last_contrib = contrib_clusters.back().back();
      if (contrib->getTime() - last_contrib->getTime() < m_cfg.minimum_separation) {
        contrib_clusters.back().push_back(contrib);
      } else {
        contrib_clusters.push_back({contrib});
      }
    }
  }
```

Pointers were used rather than objectives to make this algorithm more efficient.

Group the contributions that are close in time.

# Pulse building

```
for (const auto& contribs : contrib_clusters) {
  double time = contribs.front()->getTime();
  double pulse_height =
      std::accumulate(contribs.begin(), contribs.end(), 0.0,
                      [](double sum, const edm4hep::CaloHitContribution* contrib) {
                        return sum + contrib->getEnergy();
                      });

  // Convert energy deposit to npe and apply poisson smearing ** if necessary **
  if (m_edep_to_npe) {
    double npe = pulse_height * m_edep_to_npe.value();
    std::poisson_distribution<> poisson(npe);
    pulse_height = poisson(m_gen);
  }

  // If the pulse height is lower than m_ignore_thres, it is not necessary to scan it.
  if ((*m_pulse)(m_pulse->getMaximumTime(), pulse_height) < m_ignore_thres)
    continue;
```

A pulse is built and stored from each cluster.

Poisson smearing

# Pulse building

```
double signal_time = m_cfg.timestep * std::floor(time / m_cfg.timestep);

bool passed_threshold   = false;
std::uint32_t skip_bins = 0;
float integral          = 0;
std::vector<float> amplitudes;

// Build pulse and scanning the amplitudes.
for (std::uint32_t i = 0; i < m_cfg.max_time_bins; i++) {
  double t       = signal_time + i * m_cfg.timestep - time;
  auto amplitude = (*m_pulse)(t, pulse_height);
  if (std::abs(amplitude) < m_cfg.ignore_thres) {
    if (!passed_threshold) {
      skip_bins = i;
      continue;
    }
    if (t > m_min_sampling_time) {
      break;
    }
  }
  passed_threshold = true;
  amplitudes.push_back(amplitude);
  integral += amplitude;
}
```

Now, since pulses are no longer created from all contributions, the pulse-building part has become more similar to that of SiliconPulseGeneration.

However, it was a bit tricky and didn't seem very efficient because the calorimeter pulses still need to be generated from the contributions.

Since this algorithm may also evolve in a more calorimeter-specific direction, I propose keeping CalorimeterPulseGeneration as it is for now, without implementing the shared template.

# Others

```
41 + public:
42 +   virtual ~SignalPulse()                                  = default; // Virtual destructor
43 +   virtual double operator()(double time, double charge) = 0;
```

**wdconinc** on Jun 28 — Member

This getter must be `const`. It should have always been `const`, and the `EvaluatorPulse` will need to deal with the map of parameters in a different way.

SiliconPulseGeneration.cc

```
double EvaluatorPulse::operator()(double time, double charge) {
  param_map["time"]   = time;
  param_map["charge"] = charge;
  return m_evaluator(param_map);
}
```

```
double EvaluatorPulse::operator()(double time, double charge) const {
  auto local_map = param_map;
  local_map["time"]   = time;
  local_map["charge"] = charge;
  return m_evaluator(local_map);
}
```

# Others

```
      84  +   if (type == "LandauPulse") {
      85  +     return std::make_unique<LandauPulse>(params);
102   86      }
```

**wdconinc** on Jun 28 — Member

Factory must support all types.

## SiliconPulseGeneration.h

```
class PulseShapeFactory {

public:
  static std::unique_ptr<SignalPulse> createPulseShape(const std::string& type,
                                                       const std::vector<double>& params);
};
```

## SiliconPulseGeneration.cc

```
PulseShapeFactory::createPulseShape(const std::string& type, const std::vector<double>& params) {
  if (type == "LandauPulse") {
    return std::make_unique<LandauPulse>(params);
  }
  //
  // Add more pulse shape variants here as needed

  // If type not found, try and make a function using the ElavulatorSvc
  try {
    return std::make_unique<EvaluatorPulse>(type, params);
  } catch (...) {
    throw std::invalid_argument("Unable to make pulse shape type: " + type);
  }
}
```

```
class EvaluatorPulse : public SignalPulse {

public:
  EvaluatorPulse(const std::string& expression, const std::vector<double>& params);
```

# Others

```
      84  +   if (type == "LandauPulse") {
      85  +     return std::make_unique<LandauPulse>(params);
102   86      }
```

**wdconinc** on Jun 28 — Member

Factory must support all types.

## SiliconPulseGeneration.h

```
class PulseShapeFactory {

public:
  static std::unique_ptr<SignalPulse> createPulseShape(const std::string& type,
                                                       const std::vector<double>& params,
                                                       const std::optional<std::string>& expression = std::nullopt);
};
```

## SiliconPulseGeneration.cc

```
PulseShapeFactory::createPulseShape(const std::string& type, const std::vector<double>& params,
                const std::optional<std::string>& expression) {
  if (type == "LandauPulse") {
    return std::make_unique<LandauPulse>(params);
  }
  else if (type == "EvaluatorPulse") {
    return std::make_unique<EvaluatorPulse>(*expression, params);
  }
  throw std::invalid_argument("Unable to make pulse shape type: " + type);
}
```