File - /Users/matteo/repo/uni/ALICE-simulation/src/main.cpp

```cpp
#include <lyra/lyra.hpp>
#include <cmath>

#include "experiments/kaonSDecay.hpp"

int main(int argc, char** argv) {
  // Order of magnitude of number of generated events;
  int events = 4;

  auto cli = lyra::cli()
           | lyra::opt( events, "1-5" )
             ["-e"]("Order of magnitude of generated events.")
             .choices([](int value) -> bool { return value >= 1; });

  // Parse cli arguments
  auto result = cli.parse({ argc, argv });

  // And terminate if there is an error:
  if (!result) {
    std::cerr << result.errorMessage() << std::endl;
    exit(1);
  }

  // Create a new Kaon* decay experiment
  sim::Experiment* experiment = new sim::KaonSDecay();

  // Run the experiments. 100 particles for each event.
  experiment->run(std::pow(10, events), 100);

  // Write event data to root file
  experiment->save("kstar-decay-hist.root");
}

```

File - /Users/matteo/repo/uni/ALICE-simulation/src/Entities/entity.cpp

```cpp
#include "entity.hpp"

#include <iostream>

namespace sim {

// Constructor
////////////////////////////////////////////////////////////////////////////////////////////////////
Entity::Entity() = default;
Entity::Entity(bool isDecayProduct) :isDecayProduct_{isDecayProduct} {}
Entity::Entity(double px, double py, double pz)
  : px_{ px }
  , py_{ py }
  , pz_{ pz } {}

double Entity::width() const {
  // Most particles don't have a resonance width, and width() is not supposed to be called for them.
  // Defining this function here allows me to write less code, since I will need to override this function
  // only for resonance types.
  assert(false);

  return 0;
}

bool Entity::is(EntityType type) const {
  return this->type() == type;
}

void Entity::printAttributes() const {
  std::cout << "Type: " << type() << ", Mass: " << mass() << ", Charge: " << charge();
}

double Entity::energy() const {
  return std::sqrt(mass() * mass() + p2());
}

double Entity::traverseP() const {
  return std::sqrt((px() * px()) + (py() * py()));
}

void Entity::boost(double betaX, double betaY, double betaZ) {
  // fixme add some more comments here, unclear code.
  double energy = this->energy();

  // Boost this Lorentz vector
  double b2     = (betaX * betaX) + (betaY * betaY) + (betaZ * betaZ);
  double gamma  = 1.0 / sqrt(1.0 - b2);
  double bp     = betaX * px() + betaY * py() + betaZ * pz(); // <-- This made me waste a good ~20 hours.

  assert(gamma > 0);

  double gamma2 = (gamma - 1.0) / b2;

  px(px() + gamma2 * bp * betaX + gamma * betaX * energy);
  py(py() + gamma2 * bp * betaY + gamma * betaY * energy);
  pz(pz() + gamma2 * bp * betaZ + gamma * betaZ * energy);
}

double Entity::invariantMass(Entity& entity1, Entity& entity2) {
  const double energySumSquare   = std::pow(entity1.energy() + entity2.energy(), 2);
  const double momentumSumSquare = std::pow(entity1.px() + entity2.px(), 2)
                                 + std::pow(entity1.py() + entity2.py(), 2)
                                 + std::pow(entity1.pz() + entity2.pz(), 2);

  return std::sqrt(energySumSquare - momentumSumSquare);
}

}  // namespace sim

```

```cpp
#ifndef ENTITY_HPP
#define ENTITY_HPP

#include <cmath>
#include <cassert>

namespace sim {

enum EntityType
{
  pionP   = 0,
  pionM   = 1,
  kaonP   = 2,
  kaonM   = 3,
  protonP = 4,
  protonM = 5,
  kaonS   = 6
};

class Entity {
  double px_{ 0 };
  double py_{ 0 };
  double pz_{ 0 };

 protected:
  // Set to true only on particles generated by a decay.
  bool isDecayProduct_{ false };

 public:
  // Constructor (initialize individual particle information
  ) ////////////////////////////////////////////////////////
  Entity();
  explicit Entity(bool isDecayProduct);
  Entity(double px, double py, double pz);

  // Particle type information (static
  ) ////////////////////////////////////////////////////////////////////////////
  // Derived classes must make these values static constexpr.
  virtual EntityType type()   const = 0;
  virtual double     mass()   const = 0;
  virtual int        charge() const = 0;
  virtual double     width()  const;

  // Check if this entity is of "type"
  bool is(EntityType type) const;

  // Print entity attributes
  void printAttributes() const;

  // individual particle information
  ////////////////////////////////////////////////////////////////////////////////
  // Check if this entity is decay product.
  inline bool isDecayProduct() const {
    // This "isDecayProduct" attribute can probably be
    return isDecayProduct_;
  }

  // Get individual momentum components
  inline double px() const {
    return px_;
  };
  inline double py() const {
    return py_;
  }
  inline double pz() const {
    return pz_;
  }

  // Get momentum (squared/norm)
  inline double p2() const {
    return (px() * px()) + (py() * py()) + (pz() * pz());
  }
  inline double p() const {
    return std::sqrt(p2());
  }

```

```cpp
  // Get polar momentum coordinates
  // Polar angle
  inline double theta() const {
    return std::acos(pz() / p());
  }
  // Azimuth angle
  inline double phi() const {
    const double angle = std::atan(py() / px());
    const double x = px();
    const double y = py();

    // This certainly works. I Maybe it can be rewritten in a better way(?)
    return (x > 0 && y > 0) ? angle
         : (x < 0 && y > 0) ? M_PI + angle
         : (x < 0 && y < 0) ? M_PI + angle
         : (x > 0 && y < 0) ? M_PI * 2 + angle
         : 0;
  }

  // Set individual momentum components
  inline double px(double px) {
    return px_ = px;
  };
  inline double py(double py) {
    return py_ = py;
  }
  inline double pz(double pz) {
    return pz_ = pz;
  }

  // Set all momentum components
  inline void p(double px, double py, double pz) {
    this->px(px);
    this->py(py);
    this->pz(pz);
  }
  inline void pPolar(double p, double phi, double theta) {
    px(p * std::sin(theta) * std::cos(phi));
    py(p * std::sin(theta) * std::sin(phi));
    pz(p * std::cos(theta));
  }

  // Get total energy of the particle
  double energy() const;

  double traverseP() const;

  // Boost particle using lorentz vector transform.
  void boost(double betaX, double betaY, double betaZ);

  // Get invariant mass of two particles
  static double invariantMass(Entity& entity1, Entity& entity2);

  inline virtual int decayTo(Entity& entity1, Entity& entity2) {
    // Same considerations as width()
    assert(false);
    return -1;
  }

  // Destructor
  ////////////////////////////////////////////////////////////////////////////////////////////////////
  // We need a virtual destructor since we will be explicitly deleting derived pointers
  inline virtual ~Entity() = default;

  // And we need to obey to the 3/5/0 rule. These are deleted, since I am not using them.
  Entity(const Entity& copyFrom)            = delete;
  Entity(Entity&&)                          = delete;
  Entity& operator=(Entity&&)               = delete;
  Entity& operator=(const Entity& copyFrom) = delete;
};

}  // namespace sim

#endif  // define ENTITY_HPP
```

File - /Users/matteo/repo/uni/ALICE-simulation/src/Entities/entity-variants.hpp

```
 1 #ifndef ENTITY_VARIANTS_HPP
 2 #define ENTITY_VARIANTS_HPP
 3
 4 #include "resonances/KaonS.hpp"
 5
 6 #include "particles/PionP.hpp"
 7 #include "particles/PionM.hpp"
 8 #include "particles/KaonP.hpp"
 9 #include "particles/KaonM.hpp"
10 #include "particles/ProtonP.hpp"
11 #include "particles/ProtonM.hpp"
12
13 #endif // define ENTITY_VARIANTS_HPP
14
```

File - /Users/matteo/repo/uni/ALICE-simulation/src/Entities/particles/KaonM.hpp

```cpp
#ifndef ENTITY_KAONM_HPP
#define ENTITY_KAONM_HPP

#include "entity.hpp"

namespace sim {

class KaonM : public Entity {
  static constexpr EntityType type_{ kaonM };
  static constexpr double     mass_{ 0.49367 };
  static constexpr int        charge_{ -1 };

 public:
  inline explicit KaonM(bool isDecayProduct = false) : Entity(isDecayProduct) {}

  inline EntityType type() const override {
    return type_;
  }
  inline double mass() const override {
    return mass_;
  }
  inline int charge() const override {
    return charge_;
  }
};

}  // namespace sim

#endif  // define ENTITY_KAONM_HPP

```

File - /Users/matteo/repo/uni/ALICE-simulation/src/Entities/particles/KaonP.hpp

```cpp
#ifndef ENTITY_KAONP_HPP
#define ENTITY_KAONP_HPP

#include "entity.hpp"

namespace sim {

class KaonP : public Entity {
  static constexpr EntityType type_{ kaonP };
  static constexpr double     mass_{ 0.49367 };
  static constexpr int        charge_{ +1 };

 public:
  inline explicit KaonP(bool isDecayProduct = false) : Entity(isDecayProduct) {}

  inline EntityType type() const override {
    return type_;
  }
  inline double mass() const override {
    return mass_;
  }
  inline int charge() const override {
    return charge_;
  }
};

}  // namespace sim

#endif  // define ENTITY_KAONP_HPP

```

File - /Users/matteo/repo/uni/ALICE-simulation/src/Entities/particles/PionM.hpp

```cpp
#ifndef ENTITY_PIONM_HPP
#define ENTITY_PIONM_HPP

#include "entity.hpp"

namespace sim {

class PionM : public Entity {
  static constexpr EntityType type_{ pionM };
  static constexpr double     mass_{ 0.13957 };
  static constexpr int        charge_{ -1 };

 public:
  inline explicit PionM(bool isDecayProduct = false) : Entity(isDecayProduct) {}

  inline EntityType type() const override {
    return type_;
  }
  inline double mass() const override {
    return mass_;
  }
  inline int charge() const override {
    return charge_;
  }
};

}  // namespace sim

#endif  // define ENTITY_PIONM_HPP

```

File - /Users/matteo/repo/uni/ALICE-simulation/src/Entities/particles/PionP.hpp

```cpp
#ifndef ENTITY_PIONP_HPP
#define ENTITY_PIONP_HPP

#include "entity.hpp"

namespace sim {

class PionP : public Entity {
  static constexpr EntityType type_{ pionP };
  static constexpr double     mass_{ 0.13957 };
  static constexpr int        charge_{ +1 };

 public:
  inline explicit PionP(bool isDecayProduct = false) : Entity(isDecayProduct) {}

  inline EntityType type() const override {
    return type_;
  }
  inline double mass() const override {
    return mass_;
  }
  inline int charge() const override {
    return charge_;
  }
};

}  // namespace sim

#endif  // define ENTITY_PIONP_HPP

```

File - /Users/matteo/repo/uni/ALICE-simulation/src/Entities/particles/ProtonM.hpp

```cpp
#ifndef ENTITY_PROTONM_HPP
#define ENTITY_PROTONM_HPP

#include "entity.hpp"

namespace sim {

class ProtonM : public Entity {
  static constexpr EntityType type_{ protonM };
  static constexpr double     mass_{ 0.93827 };
  static constexpr int        charge_{ -1 };

 public:
  inline EntityType type() const override {
    return type_;
  }
  inline double mass() const override {
    return mass_;
  }
  inline int charge() const override {
    return charge_;
  }
};

}  // namespace sim

#endif  // define ENTITY_PROTONM_HPP

```

File - /Users/matteo/repo/uni/ALICE-simulation/src/Entities/particles/ProtonP.hpp

```cpp
#ifndef ENTITY_PROTONP_HPP
#define ENTITY_PROTONP_HPP

#include "entity.hpp"

namespace sim {

class ProtonP : public Entity {
  static constexpr EntityType type_{ protonP };
  static constexpr double     mass_{ 0.93827 };
  static constexpr int        charge_{ +1 };

 public:
  inline EntityType type() const override {
    return type_;
  }
  inline double mass() const override {
    return mass_;
  }
  inline int charge() const override {
    return charge_;
  }
};

}  // namespace sim

#endif  // define ENTITY_PROTONP_HPP

```

File - /Users/matteo/repo/uni/ALICE-simulation/src/Entities/resonances/KaonS.hpp

```cpp
#ifndef ENTITY_KAONS_HPP
#define ENTITY_KAONS_HPP

#include "entity.hpp"

namespace sim {

class KaonS : public Entity {
  static constexpr EntityType type_{ kaonS };
  static constexpr double     mass_{ 0.89166 };
  static constexpr int        charge_{ 0 };
  static constexpr double     width_{ 0.05 };

 public:
  inline EntityType type() const override {
    return type_;
  }
  inline double mass() const override {
    return mass_;
  }
  inline int charge() const override {
    return charge_;
  }
  inline double width() const override {
    return width_;
  }

  // Make this resonance decay into two entities. The entities need to be already generated with momentum = 0.
  // This function will set their momentum appropriately.
  inline int decayTo(Entity& entity1, Entity& entity2) override {
    if(mass() == 0.) {
      std::cout << "Decayment cannot be preformed if mass is zero" << std::endl;
      return 1;
    }

    double massMot = mass();
    double massDau1 = entity1.mass();
    double massDau2 = entity2.mass();

    assert(massDau1 ≠ 0);
    assert(massDau2 ≠ 0);

    // add width effect////////////////////////////////////

    // gaussian random numbers

    float x1, x2, w, y1, y2;

    double invnum = 1./RAND_MAX;
    do {
      x1 = 2.0 * rand()*invnum - 1.0;
      x2 = 2.0 * rand()*invnum - 1.0;
      w = x1 * x1 + x2 * x2;
    } while ( w ⩾ 1.0 );

    w = std::sqrt( (-2.0 * std::log( w ) ) / w );
    y1 = x1 * w;
    y2 = x2 * w;

    massMot += width() * y1;
    ////////////////////////////////////////////////////

    if(massMot < massDau1 + massDau2){
      printf("Decayment cannot be preformed because mass is too low in this channel\n");
      return 2;
    }

    double pout = sqrt((massMot*massMot - (massDau1+massDau2)*(massDau1+massDau2))*(massMot*massMot - (massDau1-massDau2)*(massDau1-massDau2)))/massMot*0.5;

    double norm = 2*M_PI/RAND_MAX;

    double phi = rand()*norm;
    double theta = rand()*norm*0.5 - M_PI/2.;
    entity1.p(pout*std::sin(theta)*std::cos(phi),pout*std::sin(theta)*std::sin(phi),pout*std::cos(theta));
```

```
75      entity2.p(-pout*std::sin(theta)*std::cos(phi),-pout*std::sin(theta)*std::sin(phi),-pout*std::cos(
    theta));
76
77      double energy = sqrt(px()*px() + py()*py() + pz()*pz() + massMot*massMot);
78
79      double bx = px() / energy;
80      double by = py() / energy;
81      double bz = pz() / energy;
82
83      entity1.boost(bx, by, bz);
84      entity2.boost(bx, by, bz);
85
86      return 0;
87    };
88 };
89
90 }  // namespace sim
91
92 #endif  // define ENTITY_KAONS_HPP
93
```

File - /Users/matteo/repo/uni/ALICE-simulation/src/Experiment/experiment.cpp

```cpp
#include "experiment.hpp"

```

File - /Users/matteo/repo/uni/ALICE-simulation/src/Experiment/experiment.hpp

```cpp
#ifndef EXPERIMENT_HPP
#define EXPERIMENT_HPP

#include <string>

namespace sim {

class Experiment {
 public:
  // Run experiment. The experiment will have eventCount events, and particlePerEvent particles for each event.
  virtual void run(int eventCount, int particlePerEvent) = 0;

  // Write experiment data to file.
  virtual void save(std::string fileName) = 0;
};

}  // namespace sim

#endif  // define EXPERIMENT_HPP

```

File - /Users/matteo/repo/uni/ALICE-simulation/src/Experiment/experiments/kaonSDecay.hpp

```cpp
#ifndef EXPERIMENT_KAONSDECAY_HPP
#define EXPERIMENT_KAONSDECAY_HPP

#include "experiment.hpp"
#include "entity-variants.hpp"

#include <string>
#include <vector>
#include <memory>
#include <TH1.h>
#include <TFile.h>
#include <TRandom3.h>

namespace sim {

class KaonSDecay : public Experiment {
  using EntityPtr         = std::unique_ptr<Entity>;  // fixme this makes for very funny syntax. Review it
  using EntityList        = std::vector<EntityPtr>;
  using EntityPtrIterator = EntityList::iterator;
  using TH1Ptr            = std::unique_ptr<TH1>;

  // Graphs that will be generated by this experiment
  enum Graph
  {
    ParticleDist,          // Distribution of generated particles
    AzimuthAngleDist,      // Distribution of phi angle of generated particles
    PolarAngleDist,        // Distribution of theta angle of generated particles
    MomentumDist,          // Distribution of momentum of generated particles
    TraverseMomentumDist,  // Distribution of traverse (xy plane) momentum of generated particles
    EnergyDist,            // Distribution of particles' total energy
    InvMass,               // Distribution of invariant mass of every pair of particles in an event
    InvMassOppCharge,      // Distribution of invariant mass of every pair of oppositely charged particles
    InvMassSameCharge,     // Distribution of invariant mass of every pair of same charged particles
    InvMassPKOppCharge,    // distribution of invariant mass between every oppositely charged pion-kaon couple
    InvMassPKSameCharge,   // distribution of invariant mass between every same charged pion-kaon couple
    InvMassPKCouple        // distribution of invariant mass between every decay-generated pion-kaon couple
  };

  // Root handlers
  ////////////////////////////////////////////////////////////////////////////////////////////////
  // Histogram handlers. Stored as pointers, since it's recommended. I think this is mandatory, since
  // we will use the Write() function.
  TH1Ptr   hists_[12];

  // Helper methods
  ////////////////////////////////////////////////////////////////////////////////////////////////
  // Fill histograms with entities
  inline void fillHistograms(EntityList& entities) {
    // Loop through each entity
    for (auto entity = entities.begin(); entity != entities.end(); ++entity) {
      // Handler needed a lot in this for.
      Entity* entityPtr = entity->get();

      // Compute invariant mass histograms
      for (auto entity2 = entity + 1; entity2 != entities.end(); ++entity2) {
        Entity* entity2Ptr = entity2->get();

        // We don't want to consider K* in invariant mass. Only decay products are considered.
        if (entityPtr->is(kaonS)) {
          break;
        }
        if (entity2Ptr->is(kaonS)) {
          continue;
        }

        // Compute now invariant mass, since we will need it a lot later.
        const double invMass = Entity::invariantMass(*entityPtr, *entity2Ptr);

        // Plot inv mass of all particles
        hists_[InvMass]->Fill(invMass);

        // Plot inv mass of opposite/same charged particles
        if (entityPtr->charge() * entity2Ptr->charge() < 0) {
          hists_[InvMassOppCharge]->Fill(invMass);
```

```
73          } else {  // Note that we don't have to check if charge is zero. since kaonS particles are already skipped.
74            hists_[InvMassSameCharge]->Fill(invMass);
75          }
76
77          // Plot inv mass of opposite-charged P-K couples
78          if ((entityPtr->is(pionP) && entity2Ptr->is(kaonM))
79              || (entityPtr->is(pionM) && entity2Ptr->is(kaonP))) {
80            hists_[InvMassPKOppCharge]->Fill(invMass);
81          }
82
83          // Plot inv mass of same-charged P-K couples
84          if ((entityPtr->is(pionP) && entity2Ptr->is(kaonP))
85              || (entityPtr->is(pionM) && entity2Ptr->is(kaonM))) {
86            hists_[InvMassPKSameCharge]->Fill(invMass);
87          }
88        }
89
90        if (!entityPtr->isDecayProduct()) {
91          // Every particle _except decay products_ needs to appear in these histograms.
92          hists_[ParticleDist]->Fill(entityPtr->type());
93          hists_[AzimuthAngleDist]->Fill(entityPtr->phi());
94          hists_[PolarAngleDist]->Fill(entityPtr->theta());
95          hists_[MomentumDist]->Fill(entityPtr->p());
96          hists_[TraverseMomentumDist]->Fill(entityPtr->traverseP());
97          hists_[EnergyDist]->Fill(entityPtr->energy()); // Note that k* is considered here, so 4 peaks are expected.
98        }
99
100       // Plot inv mass of decay-generated P-K couples
101       if (entityPtr->is(kaonS)) {
102         // the two particles right after a kaon* are guaranteed to be its children
103         hists_[InvMassPKCouple]->Fill(Entity::invariantMass(**(entity + 1), **(entity + 2)));
104       }
105     }
106   }
107
108   // Generate a random entity in the first available place of entities array, and return an iterator to it.
109   static inline EntityPtrIterator generateRandomEntity(EntityList& entities) {
110     // Generate polar components for current particle
111     const double p     = gRandom->Exp(1.);
112     const double phi   = gRandom->Uniform(0., 2. * M_PI);
113     const double theta = gRandom->Uniform(0., M_PI);
114
115     // This was made in order to avoid else blocks. I don't really like this syntax, I might change this
116     // with a goto or simply add back the else-if.
117     [&]() {
118       const double chance = gRandom->Rndm();
119
120       // Note that root Uniform generation appears to be slightly more likely to return lower numbers (based purely
121       // on my empirical observations). Anyways, even if this were true, it would probably be negligible for large
122       // numbers. (This is not a problem anymore, since i switched to using Rndm().
123       if (chance < .40) {
124         entities.push_back(std::make_unique<PionP>());
125         return;
126       }
127       if (chance < .80) {
128         entities.push_back(std::make_unique<PionM>());
129         return;
130       }
131       if (chance < .85) {
132         entities.push_back(std::make_unique<KaonP>());
133         return;
134       }
135       if (chance < .90) {
136         entities.push_back(std::make_unique<KaonM>());
137         return;
138       }
139       if (chance < .945) {
140         entities.push_back(std::make_unique<ProtonP>());
141         return;
142       }
143       if (chance < .99) {
```

```
144         entities.push_back(std::make_unique<ProtonM>());
145         return;
146       }
147       entities.push_back(std::make_unique<KaonS>());
148     }();
149
150     // Set momentum components
151     (entities.end() - 1)->get()->pPolar(p, phi, theta);
152
153     return entities.end() - 1;
154   }
155
156   // Generate decay particles and boost them. This function guarantees that the decay particles are generated right
157   // after entity.
158   // Note that we need to have entity passed as iterator (not Entity*), since we need to access entity + 1 and + 2.
159   static inline void handleDecay(EntityPtrIterator entity, EntityList& entities) {
160     // We need to guarantee that entities has the space for the child particles
161     entities.push_back(EntityPtr{});
162     entities.push_back(EntityPtr{});
163
164     assert(entity->get()->is(kaonS));
165
166     // Then we chose one of two possible outcomes
167     if (gRandom->Rndm(0) < .5) {
168       (*(entity + 1)) = std::make_unique<PionM>(true);
169       (*(entity + 2)) = std::make_unique<KaonP>(true);
170     } else {
171       (*(entity + 1)) = std::make_unique<PionP>(true);
172       (*(entity + 2)) = std::make_unique<KaonM>(true);
173     }
174
175     // The code here:           vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv is just a reference to the entity
176     // pointed.
177     entity->get()->decayTo(*((entity + 1)->get()), *((entity + 2)->get()));
178   }
179
180   inline void handleEvent(int particleCount) {
181     // Allocate space for new entities
182     EntityList entities;
183     entities.reserve(particleCount * 1.2);
184
185     while (particleCount --> 0) {
186       // Get iterator to randomly generated entity
187       auto entity = generateRandomEntity(entities);
188
189       // If entity is not K*, we don't have to worry about decay...
190       if (!entity->get()->is(kaonS)) {
191         continue;
192       }
193
194       // Otherwise, we must handle K* decay.
195       handleDecay(entity, entities);
196     }
197
198     // Fill histograms with data for this event
199     fillHistograms(entities);
200   }
201
202   // Public
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
203  public:
204   inline KaonSDecay() {
205     gRandom->SetSeed();
206
207     // Setup root histograms
208     hists_[ParticleDist]         = std::make_unique<TH1I>("ParticleDist", "Particle types", 7, 0, 7);
209     hists_[AzimuthAngleDist]     = std::make_unique<TH1F>("AzimuthAngleDist", "Azimuth angles", 360, 0, 2 * M_PI);
210     hists_[PolarAngleDist]       = std::make_unique<TH1F>("PolarAngleDist", "Polar angles", 360, 0, M_PI);
211     hists_[MomentumDist]         = std::make_unique<TH1F>("MomentumDist", "Momentum", 500, 0, 5);
212     hists_[TraverseMomentumDist] = std::make_unique<TH1F>("TraverseMomentumDist", "Traverse momentum",
```

```cpp
500, 0, 4); // rebin?
    hists_[EnergyDist]           = std::make_unique<TH1F>("EnergyDist", "Energy", 500, 0, 5); // rebin?
    hists_[InvMass]              = std::make_unique<TH1F>("InvMass", "Invariant mass", 600, 0, 6); // rebin?
    hists_[InvMassOppCharge]     = std::make_unique<TH1F>("InvMassOppCharge", "Invariant mass with opposite charge", 600, 0, 6);
    hists_[InvMassSameCharge]    = std::make_unique<TH1F>("InvMassSameCharge", "Invariant mass with same charge", 600, 0, 6);
    hists_[InvMassPKOppCharge]   = std::make_unique<TH1F>("InvMassPKOppCharge", "InvMassPKOppCharge", 600, 0, 6);
    hists_[InvMassPKSameCharge]  = std::make_unique<TH1F>("InvMassPKSameCharge", "InvMassPKSameCharge", 600, 0, 6);
    hists_[InvMassPKCouple]      = std::make_unique<TH1F>("InvMassPKCouple", "InvMassPKCouple", 500, 0.4, 1.4);

    hists_[ParticleDist]->GetXaxis()->SetBinLabel(1 + pionP, "pion+");
    hists_[ParticleDist]->GetXaxis()->SetBinLabel(1 + pionM, "pion-");
    hists_[ParticleDist]->GetXaxis()->SetBinLabel(1 + kaonP, "kaon+");
    hists_[ParticleDist]->GetXaxis()->SetBinLabel(1 + kaonM, "kaon-");
    hists_[ParticleDist]->GetXaxis()->SetBinLabel(1 + protonP, "proton+");
    hists_[ParticleDist]->GetXaxis()->SetBinLabel(1 + protonM, "proton-");
    hists_[ParticleDist]->GetXaxis()->SetBinLabel(1 + kaonS , "kaon*");

    // Needed for correct error handling when saving and subtracting histograms
    hists_[InvMassOppCharge]->Sumw2();
    hists_[InvMassSameCharge]->Sumw2();
    hists_[InvMassPKOppCharge]->Sumw2();
    hists_[InvMassPKSameCharge]->Sumw2();

    // Todo make histos pretty
  }

  inline void run(int eventCount, int particlesPerEvent) override {
    while (eventCount --> 0) {
      handleEvent(particlesPerEvent);
    }
  }

  inline void save(std::string fileName) override {
    // Open root file for writing. Clear and recreate file if already present.
    // ((use mutex to lock possible multi thread jank with root TFile?? very overkill for the scope of this project))
    TFile file(fileName.c_str(), "recreate");

    // Write every histogram to file.
    for (auto & histogram : hists_) {
      histogram->Write();
    }

    // Close file.
    file.Close();
  }
};

}  // namespace sim

#endif  // define EXPERIMENT_KAONSDECAY_HPP
```