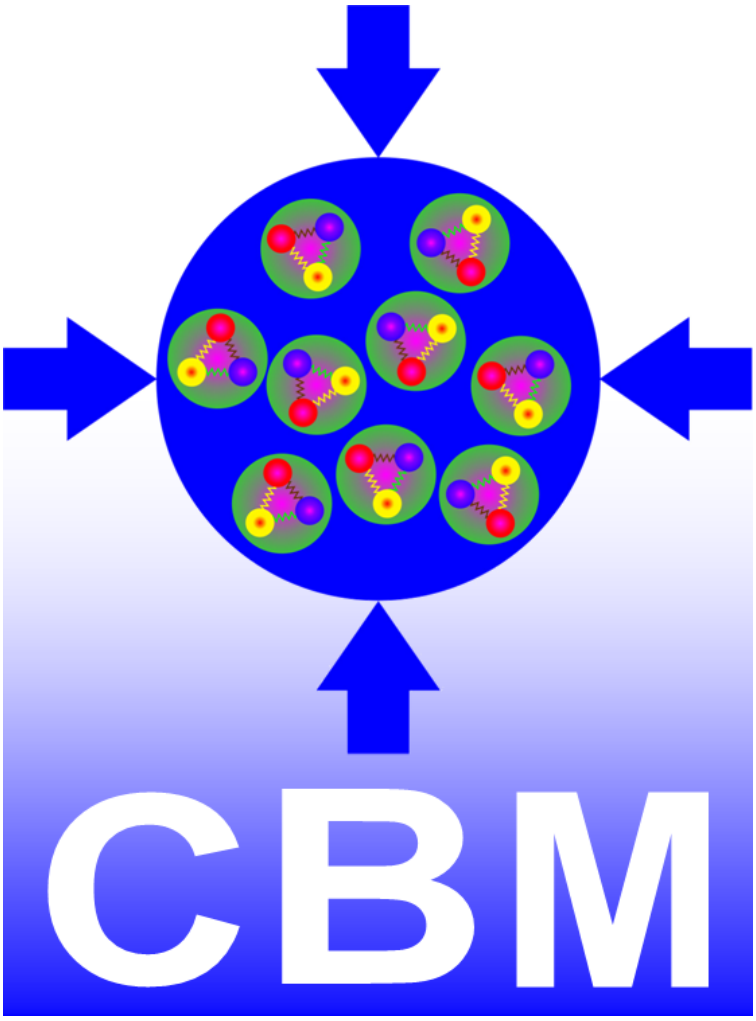
CBM szimuláció

Alex Olar

2017. október 1.



Bevezető

Egy hónapot töltöttem a nyár folyamán, július hónapba, Darmstadtban a GSI nevű kutatóközpontban. Kint tartózkodásom célja az volt, hogy többet megtudjam a CBM saját szimulációjáról, amely kutató csoport már az épülő FAIR[1] része. Ezalatt a hónap alatt megismerkedtem mélyebben a ROOT [2] nevű szoftverrel, a helyi cbmROOT-tal [3], valamint a C és C++ programozási nyelvekkel.

A kint létem alatt sokat tanultam a detektor technológiákról, valamint az azokban lejátszódó eseményekről és örömmel voltam részese ennek a nagyszabású projektnek és a mindennapi kutatói életnek.

Tartalomjegyzék

[1] Facility for Antiproton and Ion Research
[2] CERN szoftver részecske analízishez
[3] CBM (Compressed Barionic Matter) szoftver a GSI/FAIR által fejlesztve

I.. Alapok

I.1. QCD - BSc-s szemmel

A 20. század folyamán fizikusok szembeszültek azzal, hogy milyen abszolút fontos szerepet töltenek be a szimmetriák az univerzum és a körülöttünk lévő világ megismerésében. A szimmetriák vezették el őket a megmaradási tételekig, valamint az antirészecskék és kvarkok felfedezéséig többek között.

A kvarkok felfedezése végre rendet teremtett a részecske állatkertben (particle ZOO), ahogy az elemi részecskék folyamatosan növő számára Niels Nohr szellemesen referált. Kezdetben csak három kvark volt ismert, úgy mint: u (up), d (down), s (strange). A hadronok két csoportba oszthatók szét: *mezonok* és *barionok*, amelyek rendre egy kvark-antikvark párt vagy három kvarkot tartalmaznak. A mennyiséget ami a különböző kvarkokat bizonyos szempontból jellemzi *íz*nek hívjuk.

Az erős kölcsönhatás, ami a kvarkok között ható elemi kölcsönhatás, egyedi tulajdonsága a bezárás, ami megakadályozza a kvarkokat abban, hogy elszeparálva, izoláltan megtalálhatóak legyenek. Az erős kölcsönhatás töltését színnek hívjuk. A bezárás miatt, az elemi részecskék csak úgynevezett semleges színben létezhet, amit gyakran 'fehérnek' nevezünk. Az erős kölcsönhatást leíró alapvető elmélet a Kvantumszíndinamika (Quantum Chromo Dynamics) - QCD.

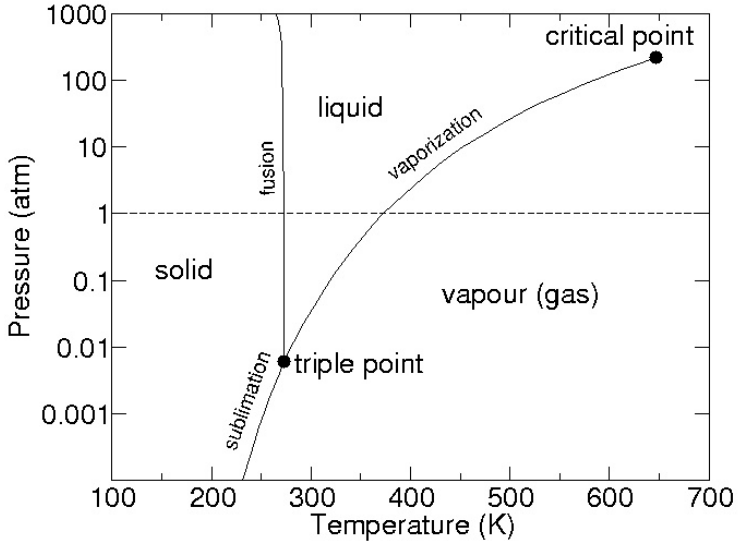
A QCD elemi részecskéi a kvarkok és antikvarkok, amelyek a gluonok által hatnak kölcsön, melyek szintén színtöltést hordoznak. A gluonoknak 8 fajtájuk van, hogy minden színtranszformáció leírható legyen segítségükkel. A gluonok önmagukkal is kölcsön tudnak hatni.

I.2. CBM fizika

A barionanyaggal foglalkozva az elsődleges cél, hogy megértsük és jobban megismerjük a fázisátmenetekhez tartozó diagramot és magukat az átmeneti folyamatokat. Először is rövid bevezetőként egy kis termodinamikai áttekintéssel kezdek a fázisokról és a fázisátalakulásokról, alapul véve a The CBM Book-ot:

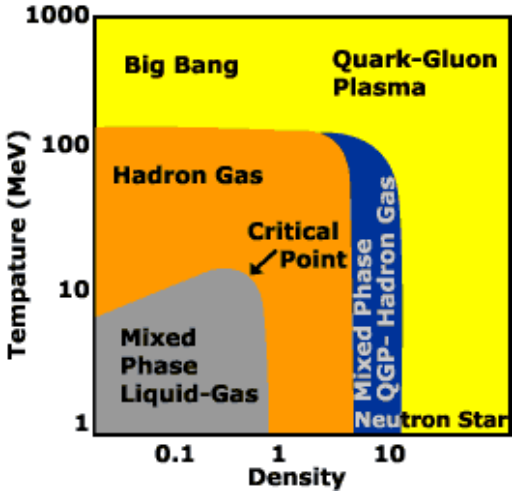
A víz fázisdiagramja megmutatja annak különböző fázisait egy nyomás-hőmérséklet rendszerben. Ismeretes, hogy adott körülmények között van egy hármaspont, amelyben a víz mindhárom halmazállapotában előfordul. A fázisok közötti vonalak mentén a víz szintén több (itt kettő) halmazállapotban előfordulhat és ezek kölcsönösen megtalálhatóak a megfelelő körülmények között. Elsőrendű fázisátmenetnek hívjuk, amikor ezen vonalakon 'áthaladva' halmazállapot-változás történik. Továbbá

megkülönböztetünk egy kritikus pontot is, amely után a fázisok nem különülnek el jelentős mértékben, ezután csak egy úgynevezett sima *crossover* figyelhető meg, nem elsőrendű fázisátmenet.

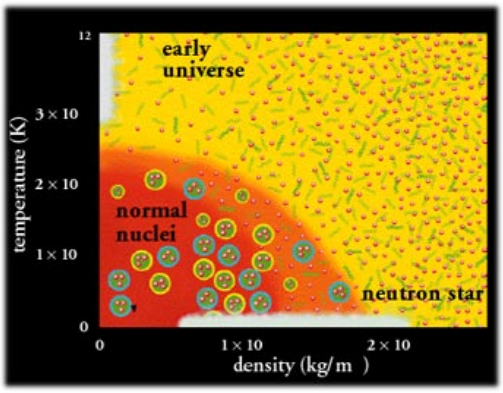


1. ábra. A víz fázisdiagramja.

Most, hogy gyorsan áttekintettem a víz fázisdiagramját, vagy legalábbis egy részét, ideje továbblépni és feltenni a kérdést, hogy mi a helyzet az erősen kölcsönható anyaggal. Az erősen kölcsönható anyag fázisdiagramja ugyanis elméleti stádiumban van, még nincs teljesen kísérletileg bizonyítva. Az ábrákon olyan különböző és elengedhetetlenül fontos fázisok vannak, amelyek a korai univerzumot jellemezték vagy éppen a neutron csillagok anyagát alkothatják. Itt mindkét ábrán egy hőmérséklet-sűrűség diagramot láthatunk.



(a) Hőmérséklet MeV-ban kifejezve, míg a sűrűség magsűrűségben van megadva, mindkét skála logaritmikus



(b) Ezen a diagramon a fázisok természetbeli előfordulását láthatjuk.

A fentebbi ábrákon is látható egy fázis, amit kvark-gluon plazmának nevezünk. Ez az állapot jelen volt a Nagy Bummnál, de később nem maradt fenn, a hőmérséklet hirtelen csökkenése miatt. Látható az is, hogy a neutron csillagok belseje is kvark-gluon plazmát tartalmazhat, de azokban nem a hőmérséklet kell igen magas legyen, hanem a csillag sűrűsége.

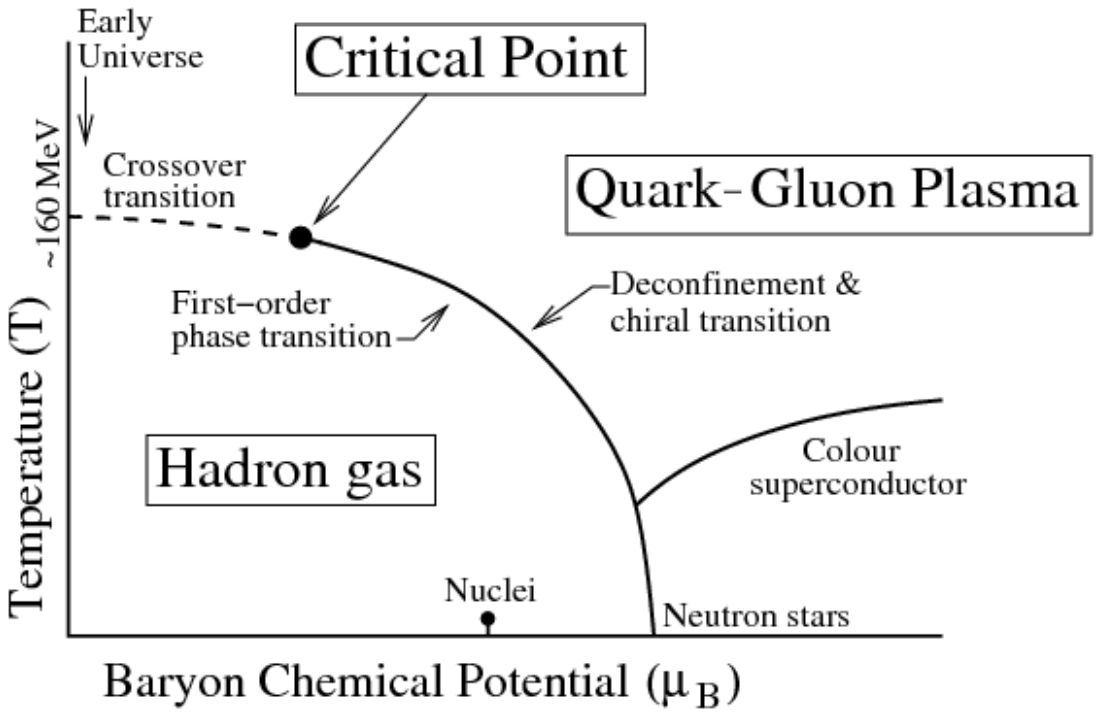
Nyilvánvalóan, a kvark-gluon plazma földi megfigyelésének egyetlen lehetőségét a nagy energiás részecske gyorsítók és azok ütköztetése biztosítja. A QCD jellemző tulajdonsága, hogy a kvarkok közti összetartás csökken, ahogy az ütközési energiát növeljük, ez a crossover jelensége. A szakirodalom aszimptotikus szabadságként hivatkozik rá. A részecske fizika egy másik fontos szimmetriája a kiralitással kapcsolatos. Ez lényegében arról beszél, hogy egy tömegtelen részecske spinje és sebességének iránya egymáshoz képest milyen irányba mutat. Hogyha azok egyirányúak, akkor a részecske jobb kezes, ha ellentétes irányúak akkor pedig bal kezes. Mivel az up és down kvarkok tömege közel azonos, ezért azt szokták mondani, hogy a QCD-nek körülbelüli királis szimmetriája van. Ellenben, ez spontán sérülhet alacsony hőmérsékleteken és sűrűségeken, ahol ez a kicsi tömegbeli különbség is jelentős lehet. Emiatt az egyik királis irány ekkor gyakoribb lesz, mint a másik, ezt nevezzük lényegében királis szimmetriasértésnek.

II.. A CBM detektor

II.1. Elmélet

A nehézionok ütközésének vizsgálata és az adatok feldolgozása egy borzasztóan komplex feladat a reakció tranziens természete miatt. Lényegében az a cél, hogy az ütközés során, 10^{-22} s-ig fennálló állapot segítségével következtessünk az erősen kölcsönható anyag fázisdiagramjára, a jelenség természetére. Az idő természetesen nagyon rövid, és az egész jelenség csak a melléktermékek révén vizsgálható.

Az elmúlt évtizedben a fő tudományos tevékenység a témában a brookheaven-i RHIC [4] központban és a CERN-ben található LHC [5] gyorsítónál zajlott. Ezek a központok nagyon fontos, és érdemleges adatot biztosítanak a fázisdiagram vizsgálatához. Ellenben ezek mind az alacsony sűrűségű régiót vizsgálják és csak a crossovert tudják feltérképezni a hadron gáz és a kvark-gluon plazma között. A FAIR projekt ezekkel szemben sokkal magasabb barion sűrűséget tervez elérni, hogy lehetőséget biztosítson az első rendű fázisátmenet vizsgálatára, valamint a kritikus pont környékének feltérképezésére.



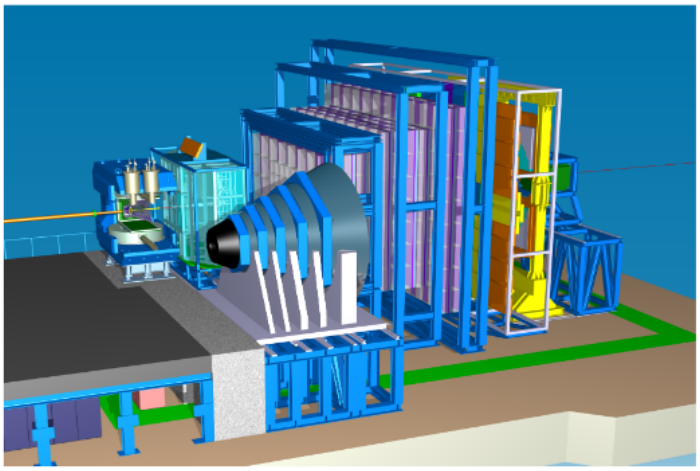
3. ábra. A fentebb említett elméleti fázisdiagram

[4]Relativistic Heavy Ion Collider - Brookhaven
[5]CERN - Large Hadron Collider

II.2. Detektor elrendezés és a szimuláció

A detektor elrendezése balról jobbra haladva a következő (ábra):

- CBM szupravezető mágnes szilícium spektrométerrel
- a micro vertex detektor (MVD) az előbbi belsejében
- a szilícium követő rendszer (STS) is
- Cserenkov-detektor (RICH - ring imaging Cherenkov detector - világos kék)
- ezt követi 4 réteg átmeneti sugárzás (TRD - transition radiation detector) detektor
- és egy time-of-flight (TOF) fal
- a fő detektorok után található még egy müon spektrométer és egy célfigyelő detektor (PSD)



4. ábra. A detektor elrendezés

A szilícium követő rendszer feladata az, hogy rekonstruálja majd a részecskék trajektóriáit. Csak töltött részecskék észlelésére képes, de képes mérni a töltés nagyságát és az impulzust is tud mérni. A TOF fal igen nagy felbontást tud elérni, nagyjából 60 ps-os felbontásra is lehetőség van.

A CBM projekt még egyenlőre csak terv szintjén létezik, a szimulációt folyamatosan fejlesztik. Jövőre, vagy legkésőbb 2019-re már tervben van egy miniCBM detektor építése az esetleg később felmerülő tervezési, kivitelezési problémák kivitelezésére. A FAIR létesítmény építése idén nyáron kezdődött és az első részecskenyaláb

2022-ben várható. A miniCBM projekt a meglévő GSI gyorsítónál fog tevékenykedni az addig fennmaradó időben, ahol megpróbálják a számítógépfarmot tökéletesíteni, hogy az adatokat minél gyorsabban feldolgozhassák.

A FAIR tudósai kifejlesztettek egy több tízezer soros szimulációt, ami a ROOT-on alapszik. Ezt ők cbmROOT-nak hívják, mivel teljes egészében a CBM-hez igazodik és ingyenesen elérhető bárki számára. Sok jól ismert nehézion szimulációs eljárást használnak, amik a CBM környezetre vannak szabva, úgy mint: UrQMD [6], valamint PHSD [7]. Ezek a szimulációs kódok széles körben használtak nem csak itt, hanem az egész tudomány területen.

III.. A Φ-mezonról röviden

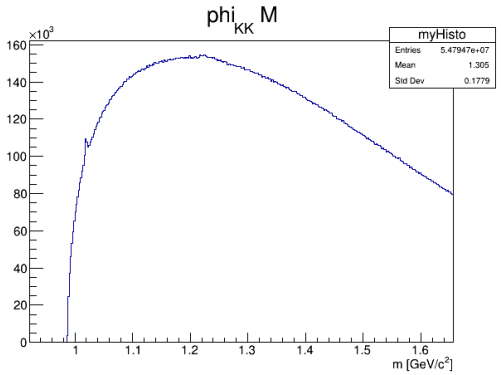
III.1. Φ-mezon rekonstrukció

A CBM detektor egy általános célú nehézion mérési eszköz lesz, hogy az erősen kölcsönható anyag fázisdiagramját vizsgálni lehessen. A rezonanciák nagyon fontosak, hogy a sűrű anyagot vizsgálni tudjuk az ütközés során. Az ilyen rezonanciák egyike ami fontos a CBM és a fázisdiagram vizsgálatának szempontjából pedig a Φ-mezon, aminek nagyon kicsi a hadronokra vett hatáskeresztmetszete így eléggé valószínűtlen, hogy kölcsönhat a nagy mennyiségű hadronnal, ami a reakció során keletkezik, vagyis jó indikátora a sűrű, kezdeti eseménynek. A Φ-mezon egy strange és egy anti-strange kvarkot tartalmaz és a kulcsa lehet az s kvark partonikus anyagban lévő keletkezésére. A Φ-mezon K^+, K^- párokra bomlik nagyjából 50%-os eséllyel és egyebekre (pl. dileptonokra is). Az közepes élettartama egészen kicsi, nagyjából $1.55 \cdot 10^{-22}$ s tehát még a TOF falat sem éri el, csak a bomlástermékei lesznek detektálva már korábban is. A tömege 1.019 MeV ami a kaonok invariáns tömegével kifejezve egy rezonancia csúcsként látható az ütközés/szimuláció után kinyert adatok között.

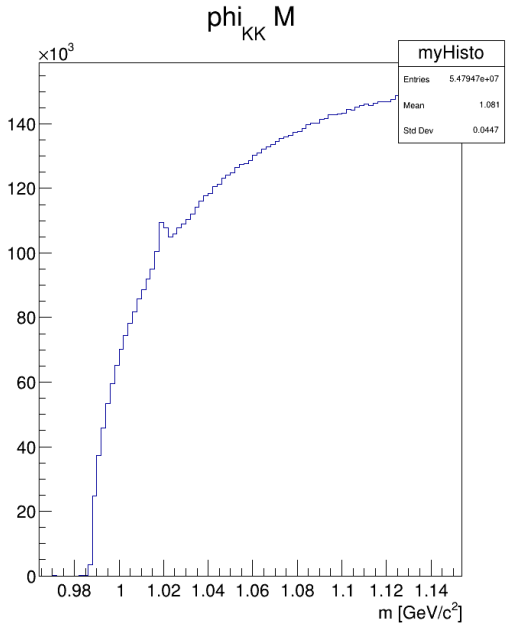
Én a PHSD adatait vizsgáltam, amin lefuttattam a CBM szimulációt. Egy Au+Au centrális ütközést vizsgáltam $\sqrt{s} = 10$ GeV energián. A CBM szimuláció kimenetét a cbmROOT-tal rekonstruáltam. Több mint 5 millió esemény szerepelt a kezdeti *.root* fájlban amit a szimulációhoz használtam.

A hisztogramokon az x-tengelyen a kaon párok invariáns tömege szerepel, az y-tengelyen pedig az adott energián a 'beütések' száma. Egy apró kiugrás látható a nagy kombinatorikus háttéren nagyjából 1.02 GeV környékén ami pontosan a Φ-mezonra utal. Azok voltak a keletkezett Φ-mezonok az ütközés során.

[6] Ultra Relativistic Quantum Molecular Dynamics
[7] Parton Hadron String Dynamics



(a) A kombinatorikus háttér és egy apró, de jól látható csúcs.

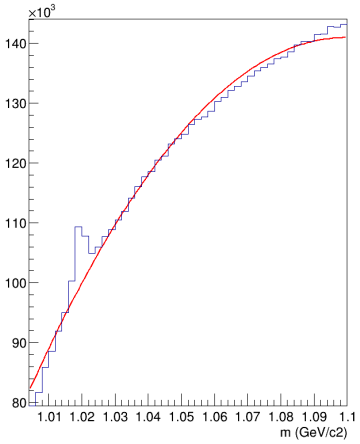


(b) A csúcs.

Egy másodfokú polinommal próbáltam becsülni a hátteret. Az illesztés paraméterei $(ax^2 + bx + c)$:

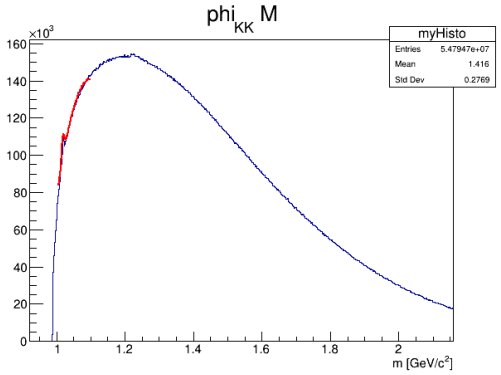
Parameter name	Value []	Error
a	-7.70559e+06	78750.5
b	1.42738e+07	150055
c	-6.49147e+06	71438.9

A háttér illesztése:

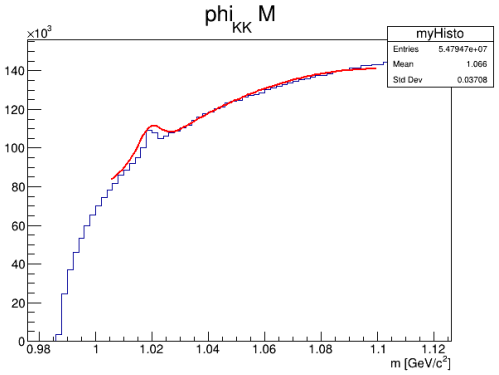


6. ábra. A háttérre vett illesztés a másodfokú polinommal.

A csúcshoz más módszert alkalmaztam. Egy alacsony multiplicitású jelet használtam a csúcs alakjának becsléséhez, amit egy Gauss-függvénnyel illesztettem, majd ezt skáláztam fel a csúcshoz, az állandó nagyságú háttér mellett, az eredmények a következők:



(a) A háttér és a csúcs illesztése.



(b) Ráközelítve.

Itt a ROOT makróm, amit az 'illesztéshez' használtam:

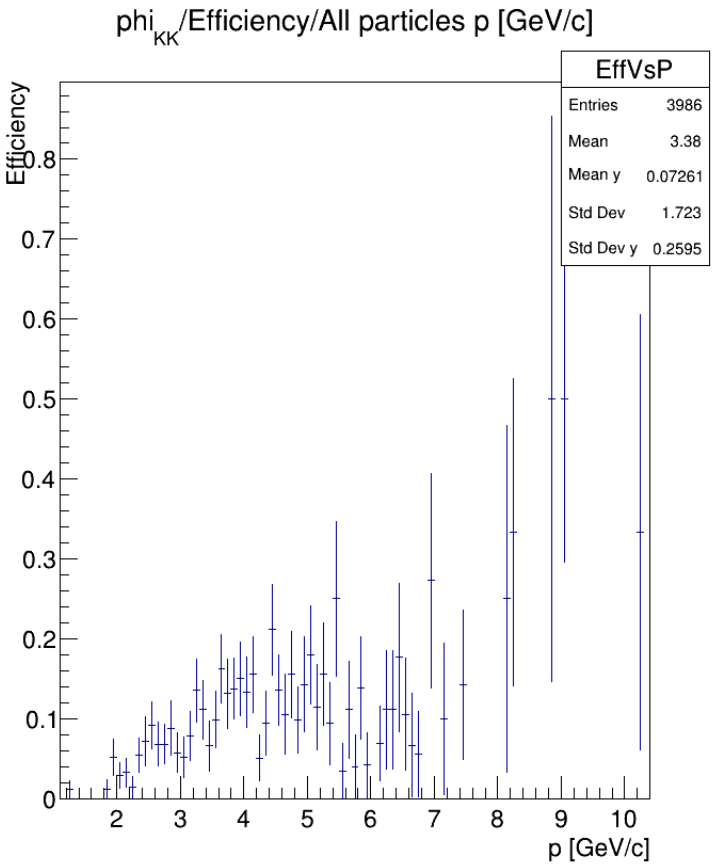
```
void fit(){

    TFile* srcFile = TFile::Open("KFParticleFinder_phsdwocsr_auau_10gev_centr_sis100_electron_5M_ToF.root");
    TDirectory* phi = (TDirectory*)srcFile->Get("KFTopoReconstructor/KFParticlesFinder/Particles/phi_{KK}/Parameters");
    TDirectory* phi_signal = (TDirectory*)srcFile->Get("KFTopoReconstructor/KFParticlesFinder/Particles/phi_{KK}/Parameters/Signal");

    TH1F* M = (TH1F*)phi->Get("M");
    TH1F* Msignal = (TH1F*)phi_signal->Get("M");
    Msignal->SetName("Msignal");

    TFile* myFile = new TFile("phi_fit.root","recreate");
    TH1F* myHisto = (TH1F*)M->Clone();
    myHisto->SetName("myHisto");
    TH1F* myBackground = (TH1F*)M->Clone();
    myBackground->SetName("myBackground");
    TH1F* mySignal = (TH1F*)Msignal->Clone();
    mySignal->SetName("mySignal");
    myFile->cd();

    srcFile->Close();

    TCanvas* canv = new TCanvas("canv","Total fit",640,480);

    TF1* background = new TF1("background","pol2",1.004,1.1);
```

```
25      TF1* signal = new TF1("signal","gaus",1.011,1.033);
26
27      TF1* total = new TF1("total","gaus(0) + pol2(3)",1.005,1.1);
28
29      background->SetParameters(0.2,1.01,108000.);
30      background->SetParNames("landau_1","landau_2","landau_3");
31      myBackground->Fit("background","R+");
32
33      Double_t params[6];
34      background->GetParameters(&params[3]);
35
36      signal->SetParameters(3200., 1.021, 3000.);
37      signal->SetParNames("scale","mean","sigma");
38      mySignal->Fit("signal","R+");
39      signal->GetParameters(&params[0]);
40
41      params[0]*=250.; // rescale
42      total->SetParameters(params);
43      total->SetParNames("scale","mean","sigma",
44              "la","b","c");
45      myHisto->Draw();
46      total->Draw("same");
47
48      signal->Write();
49      background->Write();
50      total->Write();
51      myHisto->Write();
52      canv->Write();
53
54      myFile->Close();
55
56  }
```

III.2. Φ-mezon a CBM-ben

Igen nehéz feladat lesz hatékonyan detektálni a Φ-mezonokat a CBM detektorrendszerrel. A részecskék nem csak rövid életűek, de egy hatalmas háttér is nehezíti az apró csúcs megtalálását. Ezért is kell hatalmas számú eseményt vizsgálni, hogy a csúcs a statisztikában már látható legyen. Ennek ellenére határozottan mondhatjuk, hogy a CBM detektor képes lesz a Φ-mezonok detektálására és ezáltal a strange kvark termelődésének megértésére az erősen kölcsönható anyagban.

A szimuláció hatékonysági mutatókat is biztosít. Mindezeket különböző részecske impulzusok esetén. A jelzett detektálás hatékonysági értékek nem túl magasak, de eléggé stabilak adott tartományokban az észleléshez:



8. ábra. Hatékonyság az impulzus függvényében

IV.. A szimuláció

IV.1. Telepítés

A CBM szimuláció telepítésének három fő komponense van, az egyik a FairROOT, majd a FairSoft és végül a cbmROOT. Bármilyen rendszerre telepíthetőek az alábbi linkről: `https://redmine.cbm.gsi.de/projects/cbmroot/wiki/InstallCbmRootAuto`

Erősen ajánlott a telepítést ezt követve megtenni, mivel rengeteg apró, de akadályozó probléma előjöhet a telepítés során. A teljes csomag tartalmazza a ROOT-ot is, így az egész nagyjából 25 GB helyet foglal.

IV.2. Bevezetés

Maga az ütközés a UrQMD és a PHSD programok segítségével játszódik le, a CBM szimuláció a detektor választ szimulálja, tehát az ezekből származó adatokat kapja meg kezdeti paraméternek. Ezek a modellek az ALICE, RHIC és LHC detektornak, valamint nem utolsó sorban a CBM detektornak lettek fejlesztve. Én főleg UrQMD adatokat használtam, de PHSD fájlokkal is találkoztam kint létem során.

Az első lépés az, hogy le kell futtatni egy Monte Carlo szimulációt, hogy képeset legyünk a 'valódi' adatokat összepárosítani a keltett eseményekkel. A program ezen része arra lett tervezve, hogy kiszűrje a találatokat a detektor anyagban és olyan pontokat találjon, amelyek később trajektóriákká összeállíthatók.

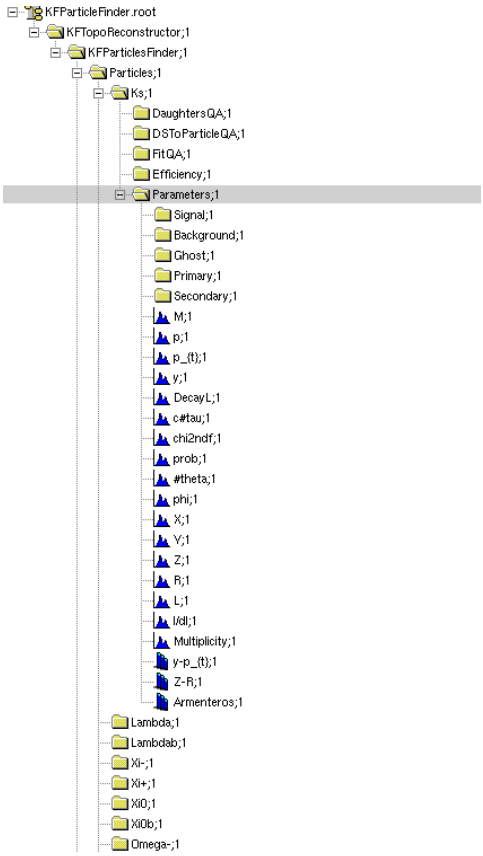
A program a Geant3 és Geant4 programokat használja, hogy a részecskék anyagon való áthaladását szimulálja. Ez is a Monte Carlo szimuláció része.

Az első makró kimenetén tehát egy szimulációs fájl van, ami az STS és az MVD detektorok által detektált találatokat tartalmazza valamint a TOF fal és egyéb detektorok adatait is. Ezeket felhasználva lép a program a második fázisba, a rekonstrukció részhez. A rekonstrukciós kód először is klasztereket próbál találni az MVD detektorban, hogy megtalálja, hogy hol volt az ütközés/ütközések kiinduló pontja. Ha ezt megtalálta továbbhalad és megpróbálja lekövetni a részecske pályákat. A töltött részecskék körpályára állnak az erős mágneses tér hatására így a pontokra köríveket próbálnak illeszteni és a legjobb illesztéssel bírókat fogadják csak el (van egy százalékos határ, ami alatt hibás detektálásnak ítélik). Én főleg az MVD és STD detektorokra koncentráltam, tehát a többit most nem említem itt.

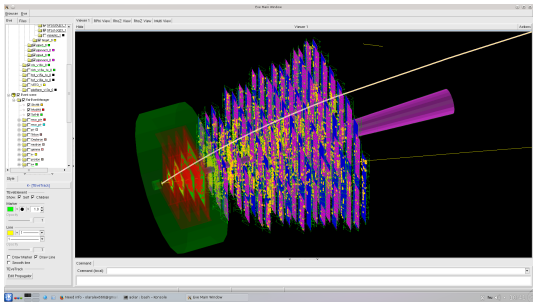
Nyilvánvalóan, a találatok és a pályákat többször próbálja meg a program helyesen megtalálni, azért, hogy elkerülje a hibákat. Kisebb az esélye így a hibás találatnak, vagy a hibásan illesztett trajektóriának. Ennék része a digitalizáció, ami lényegében azt jelenti, hogy a szimulációs program megpróbálja a detektor választ is számításba venni. Vegyük például az STS detektort. Ennek egy szálas, hálós elrendezése van, amikor egy részecske áthalad, akkor több szálban is detektáljuk, ezek metszéspontjában van a tényleges helye. De ha egyszerre két részecske ment át 'ugyan azon a ponton', akkor ezt nem láthatjuk, később a pályák illesztésénél probléma lehet. Ezért is van az, hogy ha az STS detektor több, mint 5%-a detektál, akkor a rendszer lényegében nem mér, nem szerez kiértékelhető adatokat.

A sikeres rekonstrukció után, ami a nyers adatokból létrehozta végső soron a trajektóriákat az egyetlen visszamaradó feladat a részecske felismeres és ezek pályákhoz való párosítása. Erre egy robusztus és hatékony program áll rendelkezésre, aminek a neve KFParticleFinder.

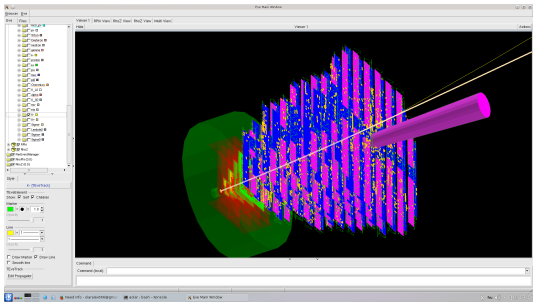
Ennek a programnak a kimenete egy .root fájl, ami rengeteg részecskét és hozzájuk tartozó adatot tartalmaz, detektálási hatékonyságról, háttérről, armenteros diagramokkal, bemenő és kimenő jelekkel, stb. . Az szerkezete nagyjából így néz ki:



9. ábra. A ROOT fájl struktúrájának egy része.



(a) A rekonstruált pályák az MVD és STS detektorokban.



(b) Másik szögből

IV.3. How-tos

Ahogy korábban említettem először a Monte Carlo szimulációt kell használni valamilyen bemeneti fájllal. Ez egy .root fájl vagy egy egyszerű ASCII fájl is lehet, a szimulációs kód képes mindkettő fogadására. Egy ilyen fájlban részecske ID-k és impulzusuk található. A kimenete a PHSD és a UrQMD szimulációknak általában egy .root fájl, de például a HIJING sima szöveges kimenetet produkál. A CBM szimulációnál különböző függvények teszik lehetővé mindkét adattípus feldolgozását.

Megtanultam használni a jelgenerátor programot, amivel bárki, bármit küldhet a detektor szimuláció bemenetére. Én főként arra használtam, hogy kontrollált körülmények között, csak Φ-mezonokat küldjek be, amivel vizsgálni lehet, hogy mi lesz a program kimenetén a KFParticleFinder által kiadott .root fájlban. Ahhoz, hogy a generátor által biztosított ASCII fájlt olvasni tudja a szimuláció a következő módosítások szükségesek:

```
1   1  0  0  0
333  0.349404  0.108345  2.17087
1   2  0  0  0
333  -0.601515  -1.42376  7.32593
1   3  0  0  0
333  0.604993  0.756893  8.0675
1   4  0  0  0
333  -0.605273  0.957298  2.78006
1   5  0  0  0
333  -0.561403  -0.245707  1.30767
1   6  0  0  0
333  -0.111909  0.297546  0.780414
1   7  0  0  0
333  0.479322  0.647613  1.23907
1   8  0  0  0
333  -0.495742  -0.65654  1.05797
1   9  0  0  0
333  -0.736586  -0.211334  2.19586
1   10  0  0  0
333  0.0558235  -0.109982  3.04292
```

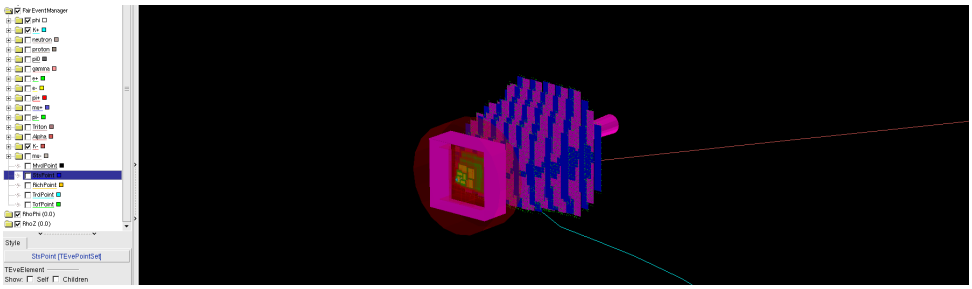
333 a részecske ID a Φ-mezonnál. Ezután a szimuláció tudni fogja, hogy hogyan dolgozza azt fel, és képes lesz azt elbomlasztani a megfelelő valószínűségekkel.

```
FairAsciiGenerator *SignalGen = new FairAsciiGenerator(inFile);
primGen->AddGenerator(SignalGen);
```

Fentebb a .root fájlokhoz használt *CbmUnigenGenerator* helyett ASCII fájlok esetén ezt kell használni. Még egy fontos lépes van itt. Ha szeretnénk vizualizálni a későbbiekben az eredményeinket, akkor engedélyeznünk kell a trajektóriák ilyen szintű mentését. Ez nyilvánvalóan nem hatékony hatalmas részecske számok esetén, de ha csak néhány részecskét küldünk be, akkor hasznos lehet látni, hogy hogyan is működik a program, esetleg hibákat is észrevehetünk.

```
// -Trajectories Visualization (TGeoManager Only )
run->SetStoreTraj(kTRUE);   //->
// -----------------------------------------------------
```

Tehát a rekonstrukció után, valamint a részecske felismerés végeztével, ha bekapcsoltuk a vizualizációt képesek vagyunk vizualizálni az eseményeket. Ehhez az *eventDisplay.C* makrót kell futtatnunk. Ez a makró az egész CBM geometriát tartalmazza, tehát az egész detektort átláthatjuk vele. Megjeleníthető benne az összes trajektória és a részecskék. Néhány kép arról, ahogy egy Φ-mezon két kaonra bomlott:



11. ábra. Vizualizáció az MVD és STS detektorokban

IV.4. Φ-mezonok generálása és a kimenő adatok elemzése

A jelgenerátorral 2500 eseményt generáltam ahol a Φ-mezonok pont a céltárgy közepében helyezkedtek el, tehát minta éppen ott keletkeztek volna az ütközés során. Az ilyen adatok elemzése azért fontos, mert ekkor kontrollált körülmények között, adott részecske számmal tudjuk vizsgálni a kimenő részeskék számít, eloszlását és ebből ismeretlen kezdeti részecskénél következtethetünk a Φ-mezonok számára így a strange keltés folyamatára.

A generátor makróban a bemenő nyaláb energiáját is változtathatjuk, valamint a környezet hőmérsékletét is (mindkettő GeV-ben) és persze azt is, hogy milyen részecskét akarunk generálni.

```
double fSlope = .154; // temperature
...
double eBeam = 10.; // beam energy
double pBeam = TMath::Sqrt(eBeam*eBeam - kProtonMass*kProtonMass);
...
 const int NParticlesPerEvent = 1;
 const double kSignalMass[NParticlesPerEvent] = {1.019455};     //  mass
    in GeV
 const int     kSignalID[NParticlesPerEvent] =  {333};
 ...
   for (int i=0; i<NEvent; i++){
   // Generate rapidity, pt and azimuth
```

```
12    outputfile<<NParticlesPerEvent<<"    "<<i + 1<<"   "<<0.<<"   "<<0.<<"
       "<<0.<<endl;
13    for(int j=0;j<NParticlesPerEvent;++j) {
14    double yD    = gRandom->Gaus(fYcm, fRapSigma);
15    double ptD   = fThermal[j].GetRandom();
16    double phiD = gRandom->Uniform(0., kTwoPi);
17
18    // Calculate momentum, energy, beta and gamma
19    double pxD     = ptD * TMath::Cos(phiD);
20    double pyD     = ptD * TMath::Sin(phiD);
21    double mtD     = TMath::Sqrt(kSignalMass[j]*kSignalMass[j] + ptD*ptD)
      ;
22    double pzD     = mtD * TMath::SinH(yD);
23
24    outputfile<<kSignalID[j]<<"   "<<pxD<<"   "<<pyD<<"   "<<pzD<<endl;
25
26    }
27    }
```

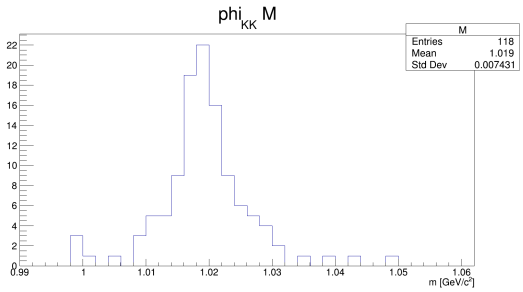
Jól látható, hogy ezt a makrót elég könnyű személyre szabni, tehát bárki könnyedén elkészítheti magának a számára megfelelő bemeneti fájlt. A bemeneti fájlt az események számával és a fájl nevével át kell adni a szimulációs programnak.

```
1 void run_mc_phi(TString inFile="Signal_phi_2500.txt", const char*
    setupName = "sis100_electron", Int_t nEvents = 2500)
2 {
3   TString outFile = "sim_phi_2500.root";
4   TString parFile = "param_phi_2500.root";
5 ...
6   // --- Define the target geometry
      ----------------------------------------
7   //
8   // The target is not part of the setup, since one and the same setup
      can
9   // and will be used with different targets.
10  // The target is constructed as a tube in z direction with the
      specified
11  // diameter (in x and y) and thickness (in z). It will be placed at
      the
12  // specified position as daughter volume of the volume present there.
       It is
13  // in the responsibility of the user that no overlaps or extrusions
      are
14  // created by the placement of the target.
15  //
16  TString  targetElement   = "Gold";
17  Double_t targetThickness = 0.025;  // full thickness in cm
18  Double_t targetDiameter  = 2.5;    // diameter in cm
19  Double_t targetPosX      = 0.;     // target x position in global c.s
      . [cm]
20  Double_t targetPosY      = 0.;     // target y position in global c.s
      . [cm]
```

```
21   Double_t targetPosZ      = 0.;     // target z position in global c.s
       . [cm]
22   Double_t targetRotY      = 0.;     // target rotation angle around
       the y axis [deg]
23 }
```

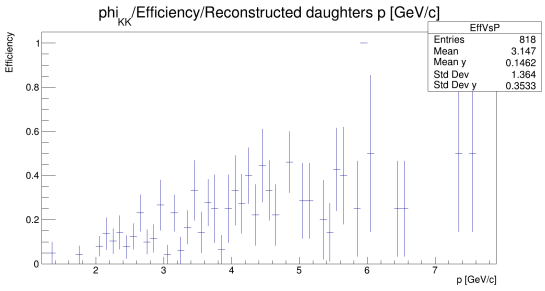
A kimenet egy .root fájl, ami mint már korábban említettem adatokat tartalmaz a beütésekkel a detektor anyagban. Fontos megemlíteni, hogy a céltárgyat is bárminek definiálhatjuk, a helyzetét is változtathatjuk, de a mi feladatunk, hogy helyesen tegyük, mert a szimuláció lefut úgy is, hogy a nyaláb el sem találja a céltárgyat.

Ezután a rekonstrukciós fájlt is kissé módosítani kell, majd ez a trajektóriákat találja meg. Majd a fizika makrót kell futtatni, hogy a KFParticleFinder megtalálja a pályákhoz tartozó részecskéket. A kimeneti .root fájlból néhány részlet:

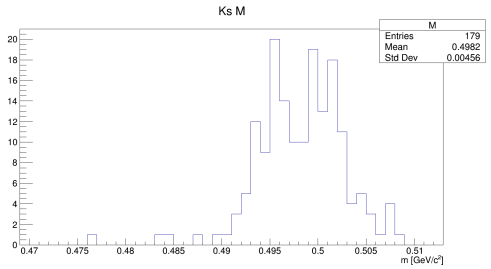


12. ábra. A kaonpárok invariáns tömegének diagramján 1.02 GeV-nél, a Φ-mezon

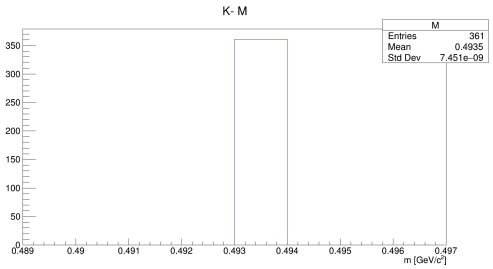
A jelben 2500 esemény volt, azaz 2500 db mezont generáltam. Nagyjából 50%-os eséllyel bomlottak el ezek kaon párokra, valamint a digitalizáció során, nagyjából 15%-os hatékonysággal tudott a program rekonstruálni, azaz nagyjából 180 db rekonstruált Φ-mezonra lehet számítani a *KFParticleFinder.root* fájlban. Mivel valószínűségekről van szó, így a fájlban lévő 120 db nem is rossz statisztikailag.



Könnyű megtalálni azokat a részecskéket is amikké a Φ-mezonok elbomlottak. Így találhatunk a bomlástermékek között pionokat, kaonokat, K_S^0 részecskéket.



(a) K_S^0 részecskék



(b) Negatív kaonok.

A pionokat itt nem tüntettem fel külön, mivel egy ilyen folyamat során azok nem adnak informatív képet, lévén, hogy nem csak a Φ-mezon tud úgy bomlani, hogy pion is van a bomlástermékek között, de a kaonok és egyéb részecskék is, így a pionok multiplicitása igen nagy.

IV.5. Összegzés

A CBM detektor képes lesz arra, hogy felismerje és megtalálja a Φ-mezon bomlásokat, ezáltal a strange termelődést és a partonikus anyagot vizsgálni tudja. A szimulációk azt sugallják, hogy a detektor minden valószínűség szerint képes lesz detektálni a szükséges részecskéket a megfelelő hatásfokokkal.

V.. Nehézion fizika itthon

A Wigner Fizikai Kutatóközpontban dolgozó témavezetőmtől, Wolf Györgytől azt a feladatot kaptam, hogy az általa írt nehézion reakciós programhoz írjak egy klaszterező programot. Ez a szimuláció a korábban említett, PHSD és UrQMD modellekhez hasonló, hazai fejlesztésű projekt. A hadron-mag és mag-mag reakciókat transzport-egyenletek segítségével vizsgálva, a BUU-modell[8] felhasználásával egy időfüggő, részecskék kölcsönhatását figyelembe vevő modell segítségével szimulálja ez a program.

Ennek kimenetén többek között szerepelhetnek bizonyos részecskék és azok momentum- és térbeli eloszlása. Detektortól függően máshogy lehet ezeket mérni. Ha olyan detektorunk van, ami csak töltött részecskéket mér, és a töltés nagyságát nem, akkor figyelembe kell vennünk, ha például térbeli (vagy impulzustérbeli) közelség miatt csak egy beütést kapunk. Így az én programom pontosan arra képes, hogy euklideszi-térben (vagy impulzustérben) klasztereket keres. Így a beütésszámra pontosabb jóslatot lehet majd adni tényleges detektor környezetben.

[8]Boltzmann-Uehling-Uhlenbeck modell

V.1. Az algoritmus

Nem tökéletesítettem még a programot, ha későbbiekben erre igény van természetesen fejlesztem. Egyenlőre hely- és impulzus-koordinátákat olvas be, majd ezután próbálja meg klaszterezni a részecskéket. A klaszterezéshez nem a legjobban ismert klaszterező algoritmust használtam hanem az úgynevezett minimális feszítő fa (vagy MST [9] a későbbiekben) algoritmust. Ezt egy gráfban a lehető legrövidebb utat találja meg. Két pont akkor van összekötve a gráfban, ha egy adott minimum távolságnál közelebb vannak. Természetesen ez a minimális távolság is a bemenetről állítható. Egy részecske egy klaszter része, ha legalább az egyik részecskéhez a klaszterben kellően közel van.

Ennek az algoritmusnak talán az a legnagyobb előnye, hogy nem kell előre feltételezni, hogy hány klaszter van és azt sem, hogy azok vajon hol helyezkedhetnek el. Elméletben az algoritmus hatékonysága $O(\log m + n)$ vagy $O(\log n \cdot n + m)$, ahol n a pontok száma a gráfban, míg m az élek száma. A hatékonyság a használt adatstruktúráktól függ. Ez természetesen Prim algoritmusára [10] igaz, vannak ennél hatékonyabb megoldások is, de számomra ez tűnt a legkényelmesebb, legmegvalósíthatóbb választásnak. Továbbá egy orosz kutatócsoport Dubnában hasonló nehézion fizikai szimulációjában is ezt az algoritmust javasolják [11].

Az egyik elméleti nehézség a megvalósítás során az volt, hogy az algoritmus képes legyen több klasztert formálni. Hiszen miután nem tud továbbhaladni egy klaszterben, azaz nem tud több pontot hozzáadni, ki kell venni az adathalmazból a klaszterezett pontokat és azt ki kell írni egy fájlba. Ezután lehet csak választani egy random pontot újra, és lefuttatni az eddigi algoritmust a már redukált gráfon.

V.2. A kód

A kódot mellékelem, ezután beszélek majd a bemenetéről és kimenetéről.

```
#include <iostream>
#include <vector>
#include <utility>
#include <set>
#include <array>
#include <string>
#include <fstream>
#include <numeric> // for std::accumulate
#include <sstream> // for std::istringstream
#include <algorithm> // for std::for_each, std::find, etc.
#include <cmath> // for std::sqrt
#include <list>
#include <chrono> // to measure time

```

[9] Minimal Spanning Tree

[10] https://en.wikipedia.org/wiki/Prim\%27s_algorithm

[11] PHQMD - J. Aichelin

```
15 double MAX_VAL = 6666.;
16
17 const int dim = 3;
18
19 int num_of_clusters = 0;
20
21 struct DataPoints{
22
23     std::vector<int> hadron;
24     std::vector<int> charge;
25     std::vector<double> mass;
26     std::vector< std::array< double, dim > > pos;
27     std::vector< std::array< double, dim > > mom;
28     std::vector<int> num_of_coll;
29
30 };
31
32 std::istream& operator>>(std::istream& is, DataPoints& points);
33
34 double distance( const std::array<double, dim>& a, const std::array<
    double, dim>& b);
35
36 std::ostream& operator<<(std::ostream& os, const std::array<double, 3>&
     arr);
37
38 std::ostream& operator<<(std::ostream& os, const std::array<double, 2>&
     arr);
39
40 bool prim(std::ostream& fout,
41           std::vector<bool>& visited,
42           std::vector< int >& ancest,
43           std::vector< std::pair< int, int > >& edges,
44           const DataPoints& data,
45           std::vector< std::vector< std::pair< int, double > > >& Graph
    ,
46           std::list<int>& open_nodes);
47
48 int main( int argc, char* argv[] )
49 {
50
51     std::ifstream fin(argv[1]);
52
53     MAX_VAL = atof(argv[2]);
54
55     DataPoints data;
56
57     fin >> data;
58
59     int num_of_vertices = data.pos.size(); // number of vertices
60
```

```
61      std::vector< std::vector< std::pair< int, double > > > Graph(
        num_of_vertices); // my graph
62
63      std::vector<bool> visited(num_of_vertices, false); // list of
        visited vertices
64
65      std::vector< int > ancest(num_of_vertices, -1);
66
67      std::vector< std::pair< int, int > > edges;
68
69      // calculate edges based on euclidian distance
70      // later only need to change pos to mom !!!
71
72      int P, Q;
73      double dist;
74
75      auto t0 = std::chrono::high_resolution_clock::now();
76
77      for( unsigned int i = 0; i < data.pos.size(); i++ ){
78
79          P = i;
80
81          for( unsigned int j = 0; j < i; j++ ){
82
83              Q = j;
84
85              dist = distance( data.pos[i],data.pos[j] );
86
87              if(dist < MAX_VAL){
88
89                  Graph[P].push_back(std::make_pair(Q, dist));
90
91                  Graph[Q].push_back(std::make_pair(P, dist));
92
93              }
94
95          }
96
97          /*
98
99          for( unsigned int j=i+1; j < data.pos.size(); j++ ){
100
101             Q = j;
102
103             dist = distance( data.pos[i],data.pos[j] );
104
105             Graph[P].push_back(std::make_pair(Q, dist));
106
107             Graph[Q].push_back(std::make_pair(P, dist));
108
109         }
```

```
110
111         */
112
113     }
114
115     auto t1 = std::chrono::high_resolution_clock::now();
116
117     std::cout << "Graph construction took: " << std::chrono::
    duration_cast< std::chrono::microseconds >(t1-t0).count() <<
118                 " microseconds\n";
119
120     std::list<int> open_nodes;
121
122     for(int i = 0; i < num_of_vertices; i++){
123
124         open_nodes.push_back(i);
125
126     }
127
128     std::ofstream fout("output.dat");
129
130     auto t2 = std::chrono::high_resolution_clock::now();
131
132     while(!prim(fout, visited, ancest, edges, data, Graph,open_nodes)
    && open_nodes.size()!=0);
133
134     auto t3 = std::chrono::high_resolution_clock::now();
135
136     std::cout << "Clustering took: " << std::chrono::duration_cast< std
    ::chrono::microseconds >(t3-t2).count() <<
137                 " microseconds\n";
138
139     return 0;
140
141 }
142
143 std::istream& operator>>(std::istream& is, DataPoints& points){
144
145     std::array< double, dim > temp_pos;
146     std::array< double, dim > temp_mom;
147
148     std::string line;
149
150     while(std::getline(is, line)){
151
152         double spare;
153
154         std::istringstream data(line);
155
156         data >> spare;
157         points.hadron.push_back(int(spare));
```

```
158
159         data >> spare;
160         points.charge.push_back(int(spare));
161
162         data >> spare;
163         points.mass.push_back(spare);
164
165         int i = 0;
166
167         while(i<dim){
168
169             data >> temp_mom[i];
170             i++;
171
172         }
173
174         i=0;
175
176         while(i<dim){
177
178             data >> temp_pos[i];
179             i++;
180
181         }
182
183         points.pos.push_back(temp_pos);
184         points.mom.push_back(temp_mom);
185
186         data >> spare;
187         points.num_of_coll.push_back(int(spare));
188
189     }
190
191     return is;
192
193 }
194
195 double distance( const std::array<double, dim>& a, const std::array<
    double, dim>& b){
196
197     std::array<double, dim> dist;
198
199     for(unsigned int i = 0; i < dim; i++){
200
201         dist[i] = a[i] - b[i];
202
203     }
204
205     return std::sqrt(std::accumulate(dist.begin(),dist.end(),0.0,[&](
    double x, double y){ return x+y*y; }));
206
```

```
207 }
208
209 std::ostream& operator<<(std::ostream& os, const std::array<double, 3>&
        arr){
210
211     os << " (" << arr[0] << ";" << arr[1] << ";" << arr[2] << ") ";
212
213     return os;
214
215 }
216
217 std::ostream& operator<<(std::ostream& os, const std::array<double, 2>&
        arr){
218
219     os << " (" << arr[0] << ";" << arr[1] << ") ";
220
221     return os;
222
223 }
224
225 bool prim(std::ostream& fout,
226           std::vector<bool>& visited,
227           std::vector< int >& ancest,
228           std::vector< std::pair< int, int > >& edges,
229           const DataPoints& data,
230           std::vector< std::vector< std::pair< int, double > > >& Graph
      ,
231           std::list<int>& open_nodes){
232
233     std::vector< double > dst(Graph.size(), MAX_VAL);
234
235     std::list<int> open_nodes_before = open_nodes;
236
237 /*
238
239     fout << "\nOpen nodes before: ";
240
241     std::for_each(open_nodes.begin(),open_nodes.end(),[&](int& x){ fout
        << x << " "; });
242
243     fout << "\n";
244
245 */
246
247     int start_vertex;
248
249     if(open_nodes.size()==0){
250
251         return true;
252
253     }
```

```
254
255     start_vertex = open_nodes.front();
256
257     open_nodes.remove(start_vertex);
258
259     std::set< std::pair< double, int > > mySet; // distance and vertex
260
261     for( int i = 0; i < visited.size(); i++ ){ // visited.size() is
    equal to number of vertices
262
263         mySet.insert(std::make_pair(dst[i],i));
264
265         // make pairs with vertex number and it's distance at the
    beginning
266         // dst[i]s are all set to MAX_VAL
267
268     }
269
270     // erase start value, then set its distance to zero
271
272     mySet.erase(mySet.find(std::make_pair(dst[start_vertex],
    start_vertex)));
273
274     dst[start_vertex] = 0.; // current distance from itself
275                             // all the others are set to MAX_VAL
276
277     mySet.insert(std::make_pair(dst[start_vertex],start_vertex));
278
279     // PRIM'S ALGORITHM
280
281     while(!mySet.empty()){
282
283         /*
284
285         std::for_each(mySet.begin(),mySet.end(),[](const std::pair<
    double, int >& i){ std::cout << i.first << " " << i.second << "\n";
     });
286         std::cout << "\n";
287
288         */
289
290         // a set is always ordered, the order is based on the first
    element of it, so the distance from itself
291
292         std::pair< double, int > top_vertex = *mySet.begin(); // gives
    back an iterator
293                                                 // to the top element
    of
294                                                 // the set so need to
    derefer it
295                                                 // and make it a pair
```

```
296
297         mySet.erase(mySet.begin());
298
299         int next_vertex = top_vertex.second; // current vertex closest
    neighbor
300
301         if(top_vertex.first == MAX_VAL){
302
303             break; // somethins is wrong !
304
305         }
306
307         visited[next_vertex] = true; // now this is visited
308
309         open_nodes.remove(next_vertex);
310
311         if(next_vertex != start_vertex
312         /* && ( std::find(open_nodes.begin(),open_nodes.end(),
    start_vertex ) != open_nodes.end()  ) */){
313                                         // at first 0 is the first
    vertex and zero is from 0 distance from itself
314                                         // so then it is skipped
    because the next_vertex is zero
315                                         // this starts with 0's
    neighbor when ancest is already set
316
317             edges.push_back(std::make_pair(ancest[next_vertex],
    next_vertex)); // make connection between next_vertex and
318
319             // its ancestor
320
321         }
322         for( unsigned int i = 0; i<Graph[next_vertex].size(); i++ ){
323
324             if( visited[Graph[next_vertex][i].first] == false ){ //
    Graph[next_vertex] is a vector of pairs to-vertex and its distance
325
326                 int next_next_vertex = Graph[next_vertex][i].first; //
    if next_vertexes neighbor is not visited then make it the
327                                                                     //
    next vertex to check
328                 double weight = Graph[next_vertex][i].second; // set
    the weight of the egde between them
329
330                 /*
331
332                 if(dst[next_next_vertex] > weight){ // if there's a
    connection between the vertices then the weight must be smaller
333                                                     // than the
    distance of the next vertex
```

```
334
335                                                     */
336
337                     mySet.erase(mySet.find( std::make_pair(dst[
    next_next_vertex],next_next_vertex) ));
338                     dst[next_next_vertex] = weight;
339
340                     mySet.insert( std::make_pair(dst[next_next_vertex],
    next_next_vertex) );
341
342                     ancest[next_next_vertex] = next_vertex;
343
344                     /*
345
346                 } */
347
348             }
349
350         }
351
352     }
353
354     std::list<int> clustered;
355
356     std::set_difference(open_nodes_before.begin(), open_nodes_before.
    end(),
357                         open_nodes.begin(), open_nodes.end(),
358                         std::back_inserter(clustered));
359
360     fout << "Cluster size: " << clustered.size() << "\n";
361
362     if(clustered.size() == 1){
363
364         fout << "Isolated point: " << data.pos[clustered.front()] << "\
    n";
365
366     }else{
367
368     /*
369
370     fout << "Clustered vertices: ";
371     std::for_each(clustered.begin(), clustered.end(), [&](int& x){ fout
    << x << " "; });
372     fout << "\n";
373
374
375     */
376
377     /*
378
379     fout << "MST is set:\n";
```

```
380
381     for( unsigned int i = 0; i < edges.size(); i++ ){
382
383             if( std::find(clustered.begin(), clustered.end(), edges[i].
    first) != clustered.end()
384             &&  std::find(clustered.begin(), clustered.end(), edges[i].
    second) != clustered.end() ){
385
386                 fout << data.pos[edges[i].first] << " --> " << data.pos
    [edges[i].second] << "\n";
387
388             }
389
390     }
391
392
393     */
394
395     }
396
397     edges.clear();
398
399     // delete the connecting branches to visited verticies
400     // if the size of the bracnhes of the graph add up to 0 then return
     true
401     // else return false
402
403     int counter = 0;
404
405     for( unsigned int i = 0; i < visited.size(); i++ ){
406
407         if(visited[i] == true){
408
409             Graph[i].clear();
410
411         }else{
412
413             counter++;
414
415         }
416
417     }
418
419     /*
420
421     fout << "Open nodes after: ";
422     std::for_each(open_nodes.begin(),open_nodes.end(),[&](int& x){ fout
    << x << " "; });
423     fout << "\n";
424
425     */
```

```
426
427     fout << "\n\n";
428
429     num_of_clusters++;
430     std::stringstream ss;
431     ss << num_of_clusters;
432
433     std::ofstream cluster("C"+ss.str()+".dat");
434
435     std::for_each(clustered.begin(), clustered.end(), [&](int& x){
436         cluster << data.hadron[x] << " " << data.charge[x] << " " <<
    data.mass[x] << " ";
437
438             for(unsigned int b = 0; b < dim; b++){
439
440                 cluster << data.mom[x][b] << " ";
441
442             }
443
444             for(unsigned int b = 0; b < dim; b++){
445
446                 cluster << data.pos[x][b] << " ";
447
448             }
449
450           cluster << data.num_of_coll[x] << "\n";
451      });
452
453     cluster.close();
454
455     if(counter == 0){
456         fout << "\n NUMBER OF CLUSTERS: " << num_of_clusters;
457         return true;
458     }else{
459         return false;
460     }
461
462
463
464 }
```

V.21. Bemeneti paraméterek

A program első paramétere a bemeneti fájl neve, ami egy szöveges formátum és egyenlőre részecske azonosítókat nem tartalmaz csak egyszerűen soronként 6 adatot, amik 3 térbeli, és 3 impulzus térbeli koordinátát jelentenek. A második paraméter a két részecske közötti távolság maximuma.

A példa kedvéért vegyünk egy olyan adatfájlt, ami 6000 sort tartalmaz, 2 db klaszterrel. Ezt én generáltam, vegyesen vannak benne pontok 3 dimenzióban le-

szórva két adott pont környékén.

```
...
24.6087 24.6546 25.1445 218.981 220.67 132.89
25.5476 24.6009 24.9727 141.339 77.3168 223.902
354.082 353.431 354.283 108.265 134.649 245.746
25.2794 25.6063 25.7068 61.4374 162.584 207.442
353.984 353.188 354.267 128.43 181.177 98.3733
25.1261 24.9143 25.2259 134.311 40.598 226.307
...
```

Ezen pontok generálásához az alábbi kódot használtam:

```
#include <random>
#include <fstream>
#include <algorithm>

int main()
{

    std::random_device dev;
    std::mt19937 mersenne(dev());
    std::normal_distribution<double> pos1_dist(25.,0.4);

    std::normal_distribution<double> pos2_dist(354.,0.54);

    std::normal_distribution<double> mom_dist(120.,73.);

    std::ofstream fout("data.dat",std::ios_base::out);

    // generate 6000 lines

    std::vector<double> output(3);

    for(int i=0; i<2000; i++){

        // 2 line generated with pos1_dist
        // 1 line genrated with pos2_dist

        // all momenta generated with mom_dist

        std::generate(output.begin(),output.end(),
                      [&]{ return pos1_dist(mersenne); });

        std::for_each(output.begin(),output.end(),
                      [&](double& x){ fout << x << " "; });

        std::generate(output.begin(),output.end(),
                      [&]{ return mom_dist(mersenne); });

        std::for_each(output.begin(),output.end(),
                      [&](double& x){ fout << x << " "; });

```

```
41          fout << "\n";
42
43          std::generate(output.begin(),output.end(),
44                        [&]{ return pos1_dist(mersenne); });
45
46          std::for_each(output.begin(),output.end(),
47                        [&](double& x){ fout << x << " "; });
48
49          std::generate(output.begin(),output.end(),
50                        [&]{ return mom_dist(mersenne); });
51
52          std::for_each(output.begin(),output.end(),
53                        [&](double& x){ fout << x << " "; });
54
55          fout << "\n";
56
57          std::generate(output.begin(),output.end(),
58                        [&]{ return pos2_dist(mersenne); });
59
60          std::for_each(output.begin(),output.end(),
61                        [&](double& x){ fout << x << " "; });
62
63          std::generate(output.begin(),output.end(),
64                        [&]{ return mom_dist(mersenne); });
65
66          std::for_each(output.begin(),output.end(),
67                        [&](double& x){ fout << x << " "; });
68
69          fout << "\n";
70
71      }
72
73      fout.close();
74
75      return 0;
76
77  }
```

V.22. A kimenet

A kimeneten egy output.txt nevezetű fájl van, ami megmutatja, hogy milyen pontok nem voltak klaszterizálva a futtatás előtt, majd azt, hogy melyikek voltak, és megmutatja, hogy melyik pontok között húzta meg a kapcsolatot az algoritmus, azaz pontonként megkapom az MST-t.

Minden pontnak az algoritmus futtatása során van egy sorszáma (lényegében ténylegesen a sor száma) és ez által hivatkozik rá a program, azaz a kimenetből részletek:

```
1 Open nodes before: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
      21 22 23 24 ...
```

```
Cluster size: 4000
Clustered vertices: 0 1 3 4 6 7 9 10 12 13 15 16 18 19 21 22 24 25 27
    28 30 31 33 34 36 ...
MST is set:
 (24.6087;24.6546;25.1445)  —>  (24.6238;24.6868;25.0871)
 (24.6238;24.6868;25.0871)  —>  (24.5937;24.7339;25.0635)
 (24.6732;24.7439;25.0026)  —>  (24.6342;24.7192;24.9762)
 (24.6342;24.7192;24.9762)  —>  (24.6278;24.7047;24.9755)
...
```

Majd természetesen ugyan ez ismétlődik, csak a másik 2000 tagú klaszterre. A program helyesen futott, hiszen én pontosan két távoli klasztert generáltam, az egyiket 4000 ponttal, a másikat 2000 ponttal.

Természetesen a sorszám által vissza lehet hivatkozni az adott pontra (részecskére) és ezáltal konkrétan megmondani, hogy melyiket alkottak egy klasztert.

V.23. Sebesség

A kód sebessége nagyban függ a definiált maximum távolságtól. Ha az előbbi algoritmust egy igen nagy (itt például 5000 egység) szám, akkor egy nagy klasztert találunk, viszonylag gyorsan, hiszen az MST keresés hiba nélkül lefut. Azonban minél kisebb távolságokra is éleket rakunk a kreált gráfba annál több számítást kell végeznünk és a sebesség nagyban leromlik.

Hivatkozások

[1] *The CBM Physics Book: Compressed Baryonic Matter in Laboratory Experiments* 2011 ed B Friman *et al* (Springer) Lect. Notes Phys.

[2] Tapia Takaki, J. D., ALICE Collaboration 2008, Journal of Physics G Nuclear Physics, 35, 044058

[3] V.Vovchenko I.Vassiliev I.Kisel M.Zyzak, Φ-meson production in Au+Au collisions and its feasibility in the CBM experiment, CBM Progress Report 2014

[4] Bravina, L., Csernai, L., Faessler, A., et al. 2003, Nuclear Physics A, 715, 665

[5] F. Wang, R. Bossingham, Q. Li, I. Sakrejda, and N. Xu, Φ-meson reconstruction in the STAR TPC, 1998

[6] Hans Rudolf Schmidt Hyperons at CBM-FAIR, Journal of Physics: Conference Series 736