Redo $D^0 \rightarrow K^- \pi^+$ using DChain

Miscellaneous Topics:
extracting information, Processor parameters, HINtupleArray
Recap

- Yesterday, you learned to reconstruct $D^0 \rightarrow K^-\pi^+$ with a double loop over tracks:

  - Selecting good tracks—discard junk not caused by charged particles from the interaction point (IP).
  - Checking each track’s particle ID ($dE/dx$)—is it a $\pi$ or a $K$ or both?
  - Checking for oppositely-charged tracks.

- Double loop is good for understanding details of combinatorics.
- But it is cumbersome. There is a simpler way: DChain.

Calculate $M_{BC}$ and $\Delta E$ for each of the 9 combinations.
What is DChain?

- DChain is a “meta-language” for combinatorics which does the looping for you. Output is equivalent to the explicit double loop:

\[
d2kpiList = \text{pionList.plus()} * \text{kaonList.minus()} ;
\]

- No explicit loops: lists are multiplied (*) or added (+) together.
- It only keeps combinations with the right charge.
- Automatic charge conjugation: you get both \(K^-\pi^+\) and \(K^+\pi^-\)
- It doesn’t use the same physical track more than once per candidate.
- Filtering (cuts) can be applied at any stage (via “selectors”).
- Output list can be used as input to another operation.
DChain and CleoDChain Classes

- **CDCandidate**: can represent tracks, showers, $\pi^0$, $\eta$, $K^0_S$, $\Lambda$
  - mass()
  - energy()
  - momentum() (HepVector3D)
  - charge()
  - kinematicData()

- To find out what kind of candidate you’ve got:
  - builtFromTrack()
    - track() returns NavTrack
  - builtFromCDPhoton()
    - photon() returns NavShower
  - Analogous functions exist for $\pi^0$, $\eta$, $K^0_S$, $\Lambda$.

- **CDDecay**
  - inherits from CDCandidate
  - has children (e.g. $D^0 \rightarrow K\pi^+$)
  - can loop over children
  - can isolate one particular child

- Lists of candidates:
  - CDChargedPionList
  - CDPi0List
  - CDEtaList
  - CDKsList
  - CDLambdaList
  - CDDecayList

Filled from FATable<NavXXX>

Output of list multiplication or addition.

/nfs/cleo3/other_sources/CLHEP/Geometry/Vector3D.h
/nfs/cleo3/other_sources/CLHEP/Vector/ThreeVector.h

By Werner Sun, presented by Eric White  13 June 2007 / CLEO 101, Day 4
Code for $D^0 \rightarrow K^-\pi^+$

- Repeat yesterday’s example using DChain instead of explicit loops.

```cpp
// Declare lists of pi's, K's
CDChargedPionList pions;
CDChargedKaonList kaons;

// Create a table of tracks and fill it
FATable< NavTrack > trackTable;
extract(iFrame.record(Stream::kEvent), trackTable);
pions = trackTable;
kaons = trackTable;

CDDecayList dtoKPiList;
dtoKPiList = kaons.minus() * pions.plus();
```

- Multiplication (*) operator → form all valid combinations with one element from each list.
- Automatic charge conjugation: you get $K^-\pi^+$ and $K^+\pi^-$ combinations.
Adding More Modes

\[ D^+ \rightarrow K^+\pi^+\pi^+ \]

\[ D^0 \rightarrow K^-\pi^+\pi^0 \]

\[ D^*+ \rightarrow D^0\pi^+ \]

\[ B^0 \rightarrow D^-\pi^+ \]

CDDecayList dtoKPiPiList;
dtoKPiPiList = kaons.minus() * pions.plus() * pions.plus();

FATable<NavPi0ToGG> pi0Table;
extract(iFrame.record(Stream::kEvent), pi0Table);
CDPi0List pi0s = pi0Table;
CDDecayList dtoKPiPi0List;
dtoKPiPi0List = kaons.minus() * pions.plus() * pi0s;

D0s = dtoKPiList + dtoKPiPi0List;

cLeodChain knows \( \pi^0 \) is self-conjugate

\[ D^0 \rightarrow K^-\pi^+\pi^0 \]

“+” operator merges two lists

D*+ \rightarrow D^0\pi^+
Examining Individual Candidates

```cpp
#include "HistogramInterface/HINtupleArray.h"
...
// example of looping over DChain candidates
CDDecayList::iterator dListEnd = dtoKPiList.particle_end();
CDDecayList::iterator dItr = dtoKPiList.particle_begin();
for( ; dItr != dListEnd; ++dItr )
{
    const CDDecay& dCand = (*dItr).particle();
    double dMass = dCand.mass();
    double dEnergy = dCand.energy();
    HepVector3D threeMomentum = dCand.momentum();
    double tuple[kDNtupleVars];

    tuple[kRun] = eventHeader->run();
    tuple[kEvent] = eventHeader->number();
    tuple[kbeamEnergy] = beamEnergy;
    ...
    tuple[kESMass] = sqrt(beamEnergy*beamEnergy -
                          threeMomentum.mag2());
    tuple[kDeltaE] = dEnergy - beamEnergy;
    m_dNtuple.fill(tuple);
}
```

Cannot use `dItr->particle()` with DChain iterators
Applying Selection Criteria

- **Three ways:**
  - Simple selectors: math-like expressions in analysis code:
    - Simple to use, straightforward syntax.
    - Variables limited to: mass, energy, charge, p_mag.
  - Selection functions:
    - Static functions that live in your Processor’s .cc file.
    - Can cut on any member function of CDCandidate.
    - Function cannot be reused in another Processor.
  - Selection class:
    - Can pass arguments to selection code.
      - Good for quantities that are not constant, like beam energy.
    - Modular coding: can use selection class in any Processor.
    - Not discussed today. See CLEO Software Workshop, Session 6

- **Syntax:** initialize a list of candidates with a selector
  - CDChargedPionList pions(myPionSelector);
  - When the list is filled, only good candidates get in.
Selection Function Example

static DChainBoolean myPionSelector( CDChargedPion& iPion )
{
    FAItem< TDKinematicFit > fit = iPion.track().pionFit();
    if( (*fit).pmag() < 50*k_MeV ) return false;
    if( (*fit).pmag() > 2.0*k_GeV ) return false;
    if( fabs((*fit).momentum().cosTheta()) > 0.93 ) return false;

    FAItem< TRHelixFit > helix = iPion.track().pionHelix();
    if( fabs( (*helix).d0() ) > 5*k_mm ) return false;
    if( fabs( (*helix).z0() ) > 5*k_cm ) return false;

    return true; //if we got this far, it is a good track
}

- iPion.track() is a NavTrack.
- Instead of (*fit).pmag() and (*helix).d0(), can also write fit->pmag() and helix->d0().

By Werner Sun, presented by Eric White
13 June 2007 / CLEO 101, Day 4
goodTrack()

```cpp
static DABoolean
goodTrack( const NavTrack& track, const DBCandidate::Hypo hypo )
{
    FAItem< TRSeedTrackQuality > seedQual = track.seedQuality();
    if( (*seedQual).originUsed() ) return false;
    if( (*seedQual).numberHitsExpected() == 0 ) return false;

    FAItem< TRTrackFitQuality > fitQual = track.quality(hypo);
    if( (*fitQual).fitAbort() ) return false;
    if( (*fitQual).chiSquare() > 100000. ) return false;
    if( (*fitQual).chiSquare() <= 0. ) return false;
    if( (*fitQual).ratioNumberHitsToExpected() < 0.3 ) return false;

    FAItem< TDKinematicFit > fit = track.kinematicFit(hypo);
    if( (*fit).pmag() < 50*k_MeV ) return false;
    if( (*fit).pmag() > 2.0*k_GeV ) return false;
    if( fabs((*fit).momentum().cosTheta()) > 0.93 ) return false;

    FAItem< TRHelixFit > helix = track.helixFit(hypo);
    if( fabs( (*helix).d0() ) > 5*k_mm ) return false;
    if( fabs( (*helix).z0() ) > 5*k_cm ) return false;

    return true; // If we survived to here, it is a good track!!
}
```
More Complicated Selectors

- Add track quality and particle ID cuts to $D^0 \rightarrow K\pi^+$ example.
- Two selection functions, one for $\pi$, one for $K$.

```c++
static DChainBoolean pionSelector( CDChargedPion& iPion )
{
  // Is this a good track??
  if (! goodTrack( iPion.track(), iPion.hypo() ) )
    return false;

  // Pion particle ID by dE/dx
  const FAItem<DedxInfo>& dedxinfo = iPion.track().dedxInfo();
  return (dedxinfo.valid() && (*dedxinfo).valid() &&
          fabs((*dedxinfo).piSigma()) < 3.0)
}
```

- And a similar function for kaons.
- Separate out common bits of code into other functions:
  - `goodTrack` for track quality cuts.
  - Modular code encourages reuse and fewer mistakes.
Code for $D^0 \rightarrow K^-\pi^+$ Revisited

\begin{verbatim}
CDChargedPionList pions;
CDChargedKaonList kaons;

FATable< NavTrack > trackTable;
extract(iFrame.record(Stream::kEvent), trackTable);
pions = trackTable;
kaons = trackTable;

CDDecayList dtoKPiList;
dtoKPiList = kaons.minus() * pions.plus();
\end{verbatim}

now becomes

\begin{verbatim}
CDChargedPionList pions( pionSelector );
CDChargedKaonList kaons( kaonSelector );

FATable< NavTrack > trackTable;
extract(iFrame.record(Stream::kEvent), trackTable);
pions = trackTable;
kaons = trackTable;

CDDecayList dtoKPiList;
dtoKPiList = kaons.minus() * pions.plus();
\end{verbatim}
#include "DChain/Function/DCSimpleSelector.h"
#include "DChain/Function/simple_selector_math.h"
#include "CleoDChain/candidate_functions.h"

DChain::Var<DChain::energy> vEnergy;
DChain::Var<DChain::p_mag> vMom;

DCSimpleSelector<CDDecay> dSelector =
    DChain::abs(vEnergy-beamEnergy) <= 100*k_MeV &&
    DChain::sqrt(beamEnergy*beamEnergy - vMom*vMom) >=
    1.83*k_GeV;

static DChainBoolean dSelector(CDDecay& iD0Cand) {
    return (fabs(iD0Cand.mass() - 1.8645*k_GeV) <= 0.1
            && fabs(iD0Cand.energy() - beamEnergy) <= 100*k_MeV);
}

CDDecayList dtoKPiList(dSelector);
dtoKPiList = kaons.minus() * pions.plus();
Extracting Other Useful Information

```cpp
#include "CleoDB/DBEventHeader.h"
#include "BeamEnergy/BeamEnergy.h"
...

ActionBase::ActionResult
MyDChainProc::event( Frame& iFrame )          // anal3 equiv.
{

  // Get the event header (for Run and event number)
  FAItem< DBEventHeader > eventHeader;
  extract( iFrame.record( Stream::kEvent ), eventHeader );
  int run =(*eventHeader).run();
  int event = (*eventHeader.)number();

  // Get the beam energy
  FAItem< BeamEnergy > beamEnergyItem;
  extract(iFrame.record( Stream::kStartRun), beamEnergyItem);
  double beamEnergy = (*beamEnergyItem).value();
```
Parameters for Job Control

- A Processor can contain “parameters” that are set by the user at run-time.
- Modify Processor’s behavior without recompiling your code.
  - Vary cuts.
  - Control printout, filling of histograms or ntuples.
- See today’s example processor, MyDChainProc:
  - .h file:
    - `#include "CommandPattern/Parameter.h"
    - add data member: `Parameter<DABoolean> m_verbose;`
  - .cc file:
    - in constructor:
      ```cpp
      MyDChainProc::MyDChainProc( void )
      : Processor( "MyDChainProc" ),
        m_verbose( "Verbose", this, false)
      ```
    - to use:
      ```cpp
      if(m_verbose.value()) cout << "Being verbose" << endl;
      ```
- `In suez: param MyDChainProc Verbose true`
Exercises

- **Get started by**
  mkproc -dchain -dedx MyDChainProc

- **Edit the default code to reconstruct** \(D^0 \rightarrow K^- \pi^+\)

- **My example is on the Day 4 web page, called** MyDChainProc.
  (also a ‘.tcl’ file... which you should know by now!)

- **Try some other decay modes**
  - \(D^+ \rightarrow K^- \pi^+ \pi^+\) (do you see \(D^+ \rightarrow K^+ \pi^- \pi^+\)?)
  - \(D^+ \rightarrow K_S^+ \pi^+\) (write a \(K_S^+\) selector)
  - \(D^0 \rightarrow K^- \pi^+ \pi^0\) (write a \(\pi^0\) selector)
    - Try \(D^0 \rightarrow K^- \pi^+, K^- \rightarrow K^- \pi^0\) and \(D^0 \rightarrow K^- \rho^+, \rho^+ \rightarrow \pi^+ \pi^0\)
  - \(D^0 \rightarrow K_S^+ \pi^-\)
  - \(D^0 \rightarrow K_S^+ \pi^- \pi^0\) (\(K^- \rho^+\) and \(K_S\) \(\omega\), \(\omega \rightarrow \pi^+ \pi^- \pi^0\))
Discussion Points

- I’m in the Illinois section of the CLEO trailer. Stop by any time!
- “static” has 3 different meanings, depending on context:
  - A static standalone function (like a selection function) is available only in file scope.
  - A static variable inside a function is not reinitialized every time the function is called—it keeps its value from the last call.
  - A static member function of a class can be invoked without having an instance of the class.
- CLEO coding convention: class names are capitalized, variable names are not.
- DBCandidate::Hypo is an enum for the inward and outward fits of the five charged track mass hypotheses (e, μ, π, K, p).
  - A variable of type DBCandidate::Hypo just acts like an int.
  - Search for “DBCandidate.h” in the software cross-reference to see the list of values in this enum.