Explorations of Functional Programming with HEP Use Cases

LATBauerdick/Fermilab

Diana Meeting 2016-12-12

Functional Programming to fight Complexity

- 2006 paper "Out of the Tarpit" by Ben Moseley and Peter Marks
  - “Complexity is the single major difficulty in the successful development of large-scale software systems.”
  - “…distinguish *accidental* from *essential* difficulty”
  - “… most complexity … in contemporary systems is [NOT] essential”
- common causes of “accidental complexity”
  - state, in particular *hidden internal state*
  - control, order in which things happen
  - others, like code volume etc

Inherent Complexity: Example: Vertex Fitting

$$\boldsymbol{x}_k(\boldsymbol{x}_{k-1}) = \boldsymbol{x}_{k-1} =: \boldsymbol{x}$$
$$\boldsymbol{p}_k = \boldsymbol{h}_k(\boldsymbol{x}, \boldsymbol{q}_k) + \boldsymbol{\epsilon}_k, \qquad \mathrm{cov}(\boldsymbol{\epsilon}_k) = \boldsymbol{V}_k = {\boldsymbol{G}_k}^{-1}$$

A *linearized track model* is obtained by approximating $\boldsymbol{h}_k$ by a first order Taylor ansatz around some "expansion point" $(\boldsymbol{x}_e, \boldsymbol{q}_{k,e})$:

$$\boldsymbol{h}_k(\boldsymbol{x}, \boldsymbol{q}_k) \approx \boldsymbol{h}_k(\boldsymbol{x}_e, \boldsymbol{q}_{k,e}) + \boldsymbol{A}_k(\boldsymbol{x} - \boldsymbol{x}_e) + \boldsymbol{B}_k(\boldsymbol{q}_k - \boldsymbol{q}_{k,e}) = \\ = \boldsymbol{A}_k\boldsymbol{x} + \boldsymbol{B}_k\boldsymbol{q}_k + \boldsymbol{c}_{k,e}$$

with $\boldsymbol{A}_k = [\partial \boldsymbol{p}_k/\partial \boldsymbol{x}]_e$ and $\boldsymbol{B}_k = [\partial \boldsymbol{p}_k/\partial \boldsymbol{q}_k]_e$ being the Jacobian matrices of derivatives at $(\boldsymbol{x}_e, \boldsymbol{q}_{k,e})$. The constants $\boldsymbol{c}_{k,e}$ can be transformed away by re-defining $\boldsymbol{p}_k \to \boldsymbol{p}_k - \boldsymbol{c}_{k,e}$ and are in the following omitted ("homogeneization"):

$$\boldsymbol{p}_k = \boldsymbol{A}_k\boldsymbol{x} + \boldsymbol{B}_k\boldsymbol{q}_k + \boldsymbol{\epsilon}_k$$

Filter formulae for estimate, residual, their covariances, and chi-squares:

$$\tilde{\boldsymbol{x}}_k = \boldsymbol{C}_k[{\boldsymbol{C}_{k-1}}^{-1}\tilde{\boldsymbol{x}}_{k-1} + {\boldsymbol{A}_k}^T{\boldsymbol{G}_k}^B\boldsymbol{p}_k], \\ \text{with } {\boldsymbol{G}_k}^B = \boldsymbol{G}_k - \boldsymbol{G}_k\boldsymbol{B}_k\boldsymbol{W}_k{\boldsymbol{B}_k}^T\boldsymbol{G}_k$$
$$\tilde{\boldsymbol{q}}_k = \boldsymbol{W}_k{\boldsymbol{B}_k}^T\boldsymbol{G}_k(\boldsymbol{p}_k - \boldsymbol{A}_k\tilde{\boldsymbol{x}}_k), \qquad \text{with } \boldsymbol{W}_k = ({\boldsymbol{B}_k}^T\boldsymbol{G}_k\boldsymbol{B}_k)^{-1}$$
$$\mathrm{cov}(\tilde{\boldsymbol{x}}_k) \equiv \boldsymbol{C}_k = ({\boldsymbol{C}_{k-1}}^{-1} + {\boldsymbol{A}_k}^T{\boldsymbol{G}_k}^B\boldsymbol{A}_k)^{-1}$$
$$\mathrm{cov}(\tilde{\boldsymbol{q}}_k) \equiv \boldsymbol{D}_k = \boldsymbol{W}_k + {\boldsymbol{E}_k}^T{\boldsymbol{C}_k}^{-1}\boldsymbol{E}_k$$
$$\mathrm{cov}(\tilde{\boldsymbol{x}}_k, \tilde{\boldsymbol{q}}_k) \equiv \boldsymbol{E}_k = -\boldsymbol{C}_k{\boldsymbol{A}_k}^T\boldsymbol{G}_k\boldsymbol{B}_k\boldsymbol{W}_k$$
$$\boldsymbol{r}_k = \boldsymbol{p}_k - \boldsymbol{A}_k\tilde{\boldsymbol{x}}_k - \boldsymbol{B}_k\tilde{\boldsymbol{q}}_k$$
$$\mathrm{cov}(\boldsymbol{r}_k) \equiv \boldsymbol{R}_k = \boldsymbol{V}_k({\boldsymbol{G}_k}^B - {\boldsymbol{G}_k}^B\boldsymbol{A}_k\boldsymbol{C}_k{\boldsymbol{A}_k}^T{\boldsymbol{G}_k}^B)\boldsymbol{V}_k$$
$$\chi^2_{k,F} = (\tilde{\boldsymbol{x}}_k - \tilde{\boldsymbol{x}}_{k-1})^T{\boldsymbol{C}_{k-1}}^{-1}(\tilde{\boldsymbol{x}}_k - \tilde{\boldsymbol{x}}_{k-1}) + {\boldsymbol{r}_k}^T\boldsymbol{G}_k\boldsymbol{r}_k \quad \text{(filtered)}$$
$$\chi^2_k = \chi^2_{k-1} + \chi^2_{k,F} \quad \text{(total when } k = n\text{)}$$

If there exists prior information of the vertex position $\tilde{\boldsymbol{x}}_0$ and its covariance matrix $\boldsymbol{C}_0$ (e.g. from the beam interaction profile), use it as additional measurement. Otherwise, set $\tilde{\boldsymbol{x}}_0 = \boldsymbol{x}_e$ and $\boldsymbol{C}_0^{-1} = \mathrm{diag}(\zeta)$, with $0 < \zeta \ll 1$.

Smoother formulae for estimate, residual, their covariances, and chi-square:

$$\tilde{\boldsymbol{q}}^n_k = \boldsymbol{W}_k{\boldsymbol{B}_k}^T\boldsymbol{G}_k(\boldsymbol{p}_k - \boldsymbol{A}_k\tilde{\boldsymbol{x}}_n)$$
$$\mathrm{cov}(\tilde{\boldsymbol{q}}^n_k) \equiv \boldsymbol{D}^n_k = \boldsymbol{W}_k + {\boldsymbol{E}^n_k}^T{\boldsymbol{C}_n}^{-1}\boldsymbol{E}^n_k$$
$$\mathrm{cov}(\tilde{\boldsymbol{x}}_n, \tilde{\boldsymbol{q}}^n_k) \equiv \boldsymbol{E}^n_k = -\boldsymbol{C}_n{\boldsymbol{A}_k}^T\boldsymbol{G}_k\boldsymbol{B}_k\boldsymbol{W}_k$$
$$\mathrm{cov}(\tilde{\boldsymbol{q}}^n_k, \tilde{\boldsymbol{q}}^n_j) = {\boldsymbol{E}^n_k}^T{\boldsymbol{C}_n}^{-1}\boldsymbol{E}^n_j \qquad (\text{for } k \neq j)$$
$$\boldsymbol{r}^n_k = \boldsymbol{p}_k - \boldsymbol{A}_k\tilde{\boldsymbol{x}}_n - \boldsymbol{B}_k\tilde{\boldsymbol{q}}^n_k$$

Frühwirth, Algebra ca 1987

Accidental Complexity — Noise

```
C -- calculate start values v0, C0 from x0, Cx0
C -- v, Cv are in carth. coords too, ergo just copy
      call fvCopy(v0,x0,dv,1)
      call fvCopy(C0,Cx0,dv,dv)

      if (print) then
        write(plun,'(1x,a)')
     1      'start values for Filter'
        write(plun,'(1x,3(g10.3,a,g10.3))') (v0(i0), ' +/-',
     1       sqrt(C0(i0,i0)) ,i0=1,dv)
      end if

C -- calculate inverse of covariance matrices for v0
        status = fvCalcG(Gv0,C0,dv)
        if (iand(status,1).NE.1) then
          fvFit = status
          return
        end if

      do i = 1, nt
        it = tList(i)
        if (it .NE. 0) then

          status = fvFilter(v,C,Gv,ql(1,it),Cql(1,1,it),E,chi2,
     1                      v0,Gv0,hl(1,it),Ghl(1,1,it))
          if (iand(status,1).NE.1) then
            write(text,'(a,i4)')
     1        'Problems in fvFilter for Track ', it
            call fvError(text)
            fvFit = status
          end if

C -- use v and Gv as input vertex v0, Gv0 for next track
          call fvCopy(v0,v,dv,1)
          call fvCopy(Gv0,Gv,dv,dv)
          call fvCopy(C0,C,dv,dv)
        end if
      end do
C
C -- the smoother
C -- v, C, Gv contain vertex, cov. matrix and cov^(-1)
      chi2t = 0d0
      do i = 1, nt
        it = tList(i)
        if (it .NE. 0) then
C -- some printout
          if (print) then
            write(plun,'(1x,a,i5)')
     1        'Smoother for Track ', it
          end if

          status = fvSmooth(ql(1,it),Cql(1,1,it),E,chi2,
     1                      v,C,ql(1,it),hl(1,it),Ghl(1,1,it))
          if (iand(status,1).NE.1) then
            write(text,'(a,i4)')
     1        'Problems in smoother for track ', it
            call fvError(text)
            fvFit = status
          end if

C -- calculate total chi2
          chi2t = chi2t + chi2

          if (print) then
C -- print out some results
            write(plun,'(1x,a,g10.3,a,g10.3)')
     1        'chi2 =', chi2, 'prob (2 d.o.f.) =', fvProb(chi2, 2)
          status = fvLUinv(Cpp, Ghl(1,1,it),dh)
          if (iand(status,1).NE.1) fvFit = status
          write(plun,'(1x,5(g10.3,a,g10.3))') (hl(i0,it), ' +/-',
     1       sqrt(Cpp(i0,i0)) ,i0=1,dh)
C -- calculate new track parameters and error matrix for printout
          status = fvhCh(pp,Cpp, v,ql(1,it),C,C ql(1,1,it),E)
          if (iand(status,1).NE.1) fvFit = status
          write(plun,'(1x,5(g10.3,a,g10.3))') (pp(i0), ' +/-',
     1      sqrt(Cpp(i0,i0)) ,i0=1,dh)
        end if
      end if
      end do

C -- save smoothed vertex position in x = {x, y, z}
      call fvCopy(x,v,dv,1)
      call fvCopy(Cx,C,dv,dv)

C -- smoothed vertex position still in v, C, Gv

      if (print) then
        write(plun,'(1x,a)')
     1      'Smoothed result'
        write(plun,'(1x,a,g10.3,a,i3,a,i3,a,g10.3,a,g10.3,a)')
     1      'chi2 =', chi2t,
     1      'prob for ',2*nt,' (',2*nt-dv,') d.o.f. =',
     1      fvProb(chi2t,2*nt),
     1      ' (',fvProb(chi2t,2*nt-dv),')'
        write(plun,'(1x,3(g10.3,a,g10.3))') (v(i0), ' +/-',
     1     sqrt(C(i0,i0)) ,i0=1,dv)
      end if

C -- check for outliers
C -- calculate chi2 for each track to belong to this vertex
C -- set alpha = cut on probaility to zero, do not remove any track
```

Aleph, Fortran ca 1990

Inherent Complexity: Example: Vertex Fitting

$$\boldsymbol{x}_k(\boldsymbol{x}_{k-1}) = \boldsymbol{x}_{k-1} =: \boldsymbol{x}$$
$$\boldsymbol{p}_k = \boldsymbol{h}_k(\boldsymbol{x}, \boldsymbol{q}_k) + \boldsymbol{\epsilon}_k, \qquad \mathrm{cov}(\boldsymbol{\epsilon}_k) = \boldsymbol{V}_k = {\boldsymbol{G}_k}^{-1}$$

A *linearized track model* is obtained by approximating $\boldsymbol{h}_k$ by a first order Taylor ansatz around some "expansion point" $(\boldsymbol{x}_e, \boldsymbol{q}_{k,e})$:

$$\boldsymbol{h}_k(\boldsymbol{x}, \boldsymbol{q}_k) \approx \boldsymbol{h}_k(\boldsymbol{x}_e, \boldsymbol{q}_{k,e}) + \boldsymbol{A}_k(\boldsymbol{x} - \boldsymbol{x}_e) + \boldsymbol{B}_k(\boldsymbol{q}_k - \boldsymbol{q}_{k,e}) = \\ = \boldsymbol{A}_k\boldsymbol{x} + \boldsymbol{B}_k\boldsymbol{q}_k + \boldsymbol{c}_{k,e}$$

with $\boldsymbol{A}_k = [\partial \boldsymbol{p}_k/\partial \boldsymbol{x}]_e$ and $\boldsymbol{B}_k = [\partial \boldsymbol{p}_k/\partial \boldsymbol{q}_k]_e$ being the Jacobian matrices of derivatives at $(\boldsymbol{x}_e, \boldsymbol{q}_{k,e})$. The constants $\boldsymbol{c}_{k,e}$ can be transformed away by re-defining $\boldsymbol{p}_k \to \boldsymbol{p}_k - \boldsymbol{c}_{k,e}$ and are in the following omitted ("homogeneization"):

$$\boldsymbol{p}_k = \boldsymbol{A}_k\boldsymbol{x} + \boldsymbol{B}_k\boldsymbol{q}_k + \boldsymbol{\epsilon}_k$$

Filter formulae for estimate, residual, their covariances, and chi-squares:

$$\tilde{\boldsymbol{x}}_k = \boldsymbol{C}_k[{\boldsymbol{C}_{k-1}}^{-1}\tilde{\boldsymbol{x}}_{k-1} + {\boldsymbol{A}_k}^T{\boldsymbol{G}_k}^B\boldsymbol{p}_k],$$
$$\text{with } {\boldsymbol{G}_k}^B = \boldsymbol{G}_k - \boldsymbol{G}_k\boldsymbol{B}_k\boldsymbol{W}_k{\boldsymbol{B}_k}^T\boldsymbol{G}_k$$
$$\tilde{\boldsymbol{q}}_k = \boldsymbol{W}_k{\boldsymbol{B}_k}^T\boldsymbol{G}_k(\boldsymbol{p}_k - \boldsymbol{A}_k\tilde{\boldsymbol{x}}_k), \qquad \text{with } \boldsymbol{W}_k = ({\boldsymbol{B}_k}^T\boldsymbol{G}_k\boldsymbol{B}_k)^{-1}$$
$$\mathrm{cov}(\tilde{\boldsymbol{x}}_k) \equiv \boldsymbol{C}_k = ({\boldsymbol{C}_{k-1}}^{-1} + {\boldsymbol{A}_k}^T{\boldsymbol{G}_k}^B\boldsymbol{A}_k)^{-1}$$
$$\mathrm{cov}(\tilde{\boldsymbol{q}}_k) \equiv \boldsymbol{D}_k = \boldsymbol{W}_k + {\boldsymbol{E}_k}^T{\boldsymbol{C}_k}^{-1}\boldsymbol{E}_k$$
$$\mathrm{cov}(\tilde{\boldsymbol{x}}_k, \tilde{\boldsymbol{q}}_k) \equiv \boldsymbol{E}_k = -\boldsymbol{C}_k{\boldsymbol{A}_k}^T\boldsymbol{G}_k\boldsymbol{B}_k\boldsymbol{W}_k$$
$$\boldsymbol{r}_k = \boldsymbol{p}_k - \boldsymbol{A}_k\tilde{\boldsymbol{x}}_k - \boldsymbol{B}_k\tilde{\boldsymbol{q}}_k$$
$$\mathrm{cov}(\boldsymbol{r}_k) \equiv \boldsymbol{R}_k = \boldsymbol{V}_k({\boldsymbol{G}_k}^B - {\boldsymbol{G}_k}^B\boldsymbol{A}_k\boldsymbol{C}_k{\boldsymbol{A}_k}^T{\boldsymbol{G}_k}^B)\boldsymbol{V}_k$$
$$\chi^2_{k,F} = (\tilde{\boldsymbol{x}}_k - \tilde{\boldsymbol{x}}_{k-1})^T{\boldsymbol{C}_{k-1}}^{-1}(\tilde{\boldsymbol{x}}_k - \tilde{\boldsymbol{x}}_{k-1}) + {\boldsymbol{r}_k}^T\boldsymbol{G}_k\boldsymbol{r}_k \quad \text{(filtered)}$$
$$\chi^2_k = \chi^2_{k-1} + \chi^2_{k,F} \quad \text{(total when } k = n\text{)}$$

If there exists prior information of the vertex position $\tilde{\boldsymbol{x}}_0$ and its covariance matrix $\boldsymbol{C}_0$ (e.g. from the beam interaction profile), use it as additional measurement. Otherwise, set $\tilde{\boldsymbol{x}}_0 = \boldsymbol{x}_e$ and $\boldsymbol{C}_0^{-1} = \mathrm{diag}(\zeta)$, with $0 < \zeta \ll 1$.

Smoother formulae for estimate, residual, their covariances, and chi-square:

$$\tilde{\boldsymbol{q}}^n_k = \boldsymbol{W}_k{\boldsymbol{B}_k}^T\boldsymbol{G}_k(\boldsymbol{p}_k - \boldsymbol{A}_k\tilde{\boldsymbol{x}}_n)$$
$$\mathrm{cov}(\tilde{\boldsymbol{q}}^n_k) \equiv \boldsymbol{D}^n_k = \boldsymbol{W}_k + {\boldsymbol{E}^n_k}^T{\boldsymbol{C}_n}^{-1}\boldsymbol{E}^n_k$$
$$\mathrm{cov}(\tilde{\boldsymbol{x}}_n, \tilde{\boldsymbol{q}}^n_k) \equiv \boldsymbol{E}^n_k = -\boldsymbol{C}_n{\boldsymbol{A}_k}^T\boldsymbol{G}_k\boldsymbol{B}_k\boldsymbol{W}_k$$
$$\mathrm{cov}(\tilde{\boldsymbol{q}}^n_k, \tilde{\boldsymbol{q}}^n_j) = {\boldsymbol{E}^n_k}^T{\boldsymbol{C}_n}^{-1}\boldsymbol{E}^n_j \qquad (\text{for } k \neq j)$$
$$\boldsymbol{r}^n_k = \boldsymbol{p}_k - \boldsymbol{A}_k\tilde{\boldsymbol{x}}_n - \boldsymbol{B}_k\tilde{\boldsymbol{q}}^n_k$$

Frühwirth, Algebra ca 1987

Accidental Complexity — Noise

```
254 // The method where the vertex fit is actually done!
255 //
256 template <unsigned int N>
257 CachingVertex<N>
258 SequentialVertexFitter<N>::fit(const std::vector<RefCountedVertexTrack> & tracks,
259                                const VertexState priorVertex, bool withPrior ) const
260 {
261   std::vector<RefCountedVertexTrack> initialTracks;
262   GlobalPoint priorVertexPosition = priorVertex.position();
263   GlobalError priorVertexError = priorVertex.error();
264
265   CachingVertex<N> returnVertex(priorVertexPosition,priorVertexError,initialTracks,0
266   if (withPrior) {
267     returnVertex = CachingVertex<N>(priorVertexPosition,priorVertexError,
268                 priorVertexPosition,priorVertexError,initialTracks,0);
269   }
270   CachingVertex<N> initialVertex = returnVertex;
271   std::vector<RefCountedVertexTrack> globalVTracks = tracks;
272
273   // main loop through all the VTracks
274   bool validVertex = true;
275   int step = 0;
276   GlobalPoint newPosition = priorVertexPosition;
277   GlobalPoint previousPosition;
278   do {
279     CachingVertex<N> fVertex = initialVertex;
280     // make new linearized and vertex tracks for the next iteration
281     if(step != 0) globalVTracks = reLinearizeTracks(tracks,
282                                        returnVertex.vertexState());
283
284     // update sequentially the vertex estimate
285     for (typename std::vector<RefCountedVertexTrack>::const_iterator i
286            = globalVTracks.begin(); i != globalVTracks.end(); i++) {
287       fVertex = theUpdator->add(fVertex,*i);
288       if (!fVertex.isValid()) break;
289     }
290
291     validVertex = fVertex.isValid();
292     // check tracker bounds and NaN in position
293     if (validVertex && hasNan(fVertex.position())) {
294       LogDebug("RecoVertex/SequentialVertexFitter")
295          << "Fitted position is NaN.\n";
296       validVertex = false;
297     }
298
299     if (validVertex && !insideTrackerBounds(fVertex.position())) {
300       LogDebug("RecoVertex/SequentialVertexFitter")
301          << "Fitted position is out of tracker bounds.\n";
302       validVertex = false;
303     }
304
305     if (!validVertex) {
306       // reset initial vertex position to (0,0,0) and force new iteration
307       // if number of steps not exceeded
308       ROOT::Math::SMatrixIdentity id;
309       AlgebraicSymMatrix33 we(id);
310       GlobalError error(we*10000);
311       fVertex = CachingVertex<N>(GlobalPoint(0,0,0), error,
312                                  initialTracks, 0);
313     }
314
315     previousPosition = newPosition;
316     newPosition = fVertex.position();
317
318     returnVertex = fVertex;
319     globalVTracks.clear();
320     step++;
321   } while ( (step != theMaxStep) &&
322             (((previousPosition - newPosition).transverse() > theMaxShift) ||
323                 (!validVertex) ) );
324
325   if (!validVertex) {
326     LogDebug("RecoVertex/SequentialVertexFitter")
327        << "Fitted position is invalid (out of tracker bounds or has NaN). Returned ve
328     return CachingVertex<N>(); // return invalid vertex
329   }
330
331   if (step >= theMaxStep) {
332     LogDebug("RecoVertex/SequentialVertexFitter")
333        << "The maximum number of steps has been exceeded. Returned vertex is invali
334     return CachingVertex<N>(); // return invalid vertex
335   }
336
337   // smoothing
338   returnVertex = theSmoother->smooth(returnVertex);
339
340   return returnVertex;
341 }
342
343 template class SequentialVertexFitter<5>;
344 template class SequentialVertexFitter<6>;
```

CMS, C++ ca 2006

The Pragmatic Programmers

Seven Languages in Seven Weeks

A Pragmatic Guide to Learning Programming Languages

Bruce A. Tate

Edited by Jacquelyn Carter

The Pragmatic Programmers

Seven More Languages in Seven Weeks

Languages That Are Shaping the Future

Bruce A. Tate, Fred Daoud, Ian Dees, and Jack Moffitt

Foreword by José Valim

Edited by Jacquelyn Carter

- Ruby, Io, Prolog, *Scala*, *Erlang*, **Clojure**, **Haskell**
- Lua, Factor, *Elm*, **Elixir**, *Julia*, miniKanren, **Idris**

Functional Programming Promises

- "**FP**[…] treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data" (Wikipedia)
  - Without mutable state and with pure functions one gets **referential transparency**: calling any function *f* with the same value for an argument *x* will *always* produce the same result *f(x)*
  - Thus any expression can replace function calls with the result of the function, helping **refactoring**, allows **parallelization** and **concurrency**
- With non-strict or **lazy evaluation**, function arguments are evaluated only when their values are required to evaluate the function call itself —> "computation on demand"
- Any "time domain" aspect is no longer part of the code, which becomes merely the specification of **data transformations**
- **Optimization, parallelization, moving to GPUs or FPGAs** with help of highly optimizing compilers, maybe w/ some code refactoring or compiler hints

Functional Programming Promises

- **composition** of functions can be done “cleanly”, and there are even more general ways to compose data transformations that have a clean **foundation in math**
  - $(g \circ f)(x) = g(f(x))$, map and reduce-type operations on lists/vectors, Monadic compositions etc
  - **side effects and state** (I/O, random numbers, data bases, …) are being **handled explicitly** (e.g., monads in Haskell, or by passing around state data structures)
- **type systems** and **algebraic datatypes** make manipulation of complex data structures convenient and reduce them to their “math equivalence” — don’t need to be “re-invented”, are highly optimized
  - strong compile-time type checking (together with type inference of compilers) makes programs more reliable while freeing programmer from the need to manually declare types
- “**nice**” / **consistent** to program: syntactic sugar like pattern matching, list comprehension, etc
- naively, **performance** can be an issue (log in the # memory cells) but can also be very good:
  - FP allows compilers to make assumptions that are unsafe in an imperative language, thus increasing opportunities for e.g. inline expansion. Lazy evaluation / on-demand computation helps, but also requires careful orchestration for modern processors with deep pipelines and multi-level caches — which however can be done by experts, and would be mostly transparent to the physicist providing the algorithm

- **Reality Check: Use Case of Vertex Fitting, implemented in Clojure and Haskell**

Use Case:
Vertex Fitting and Vertex Finding using Kalman Filter

- "Mature" algorithm (since ~1987!), e.g. used in Aleph 1990

- CMS implementation ~2006 (T.Speer et at):
  primary vertex, "vertex tools" etc

- Approach: use Kaman filter to combine helices into a common vertex by "adding" more and more helices finding the "best match"

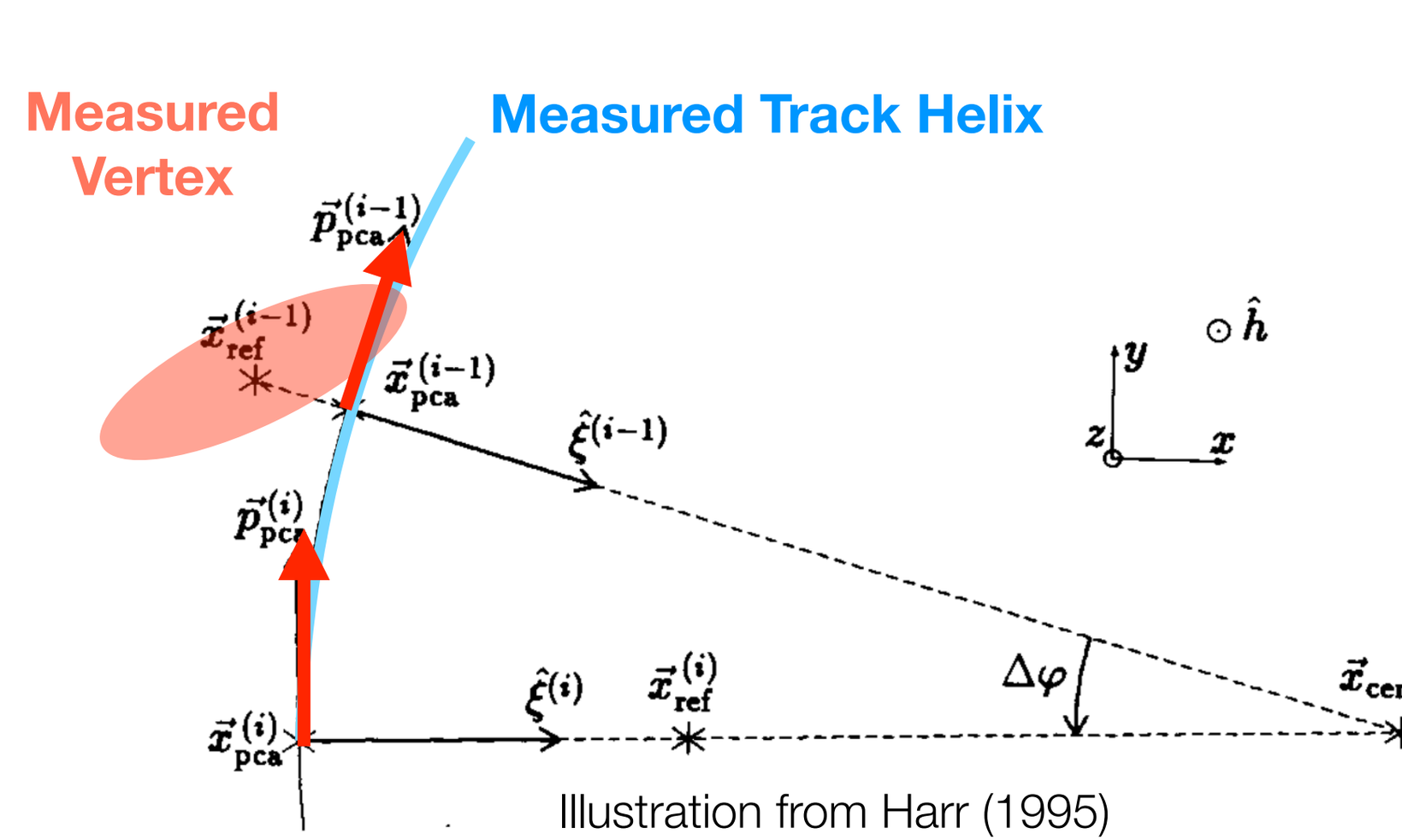


Illustration from Harr (1995)

Math description of algorithm translates ~directly to functional programming code:

- State Vector: $\boldsymbol{x}, \boldsymbol{q}_k$ vertex position and momenta of tracks at vertex
- filter step: add a new track $\boldsymbol{p}_k$ to a vertex already fitted with k–1 tracks, updating its position estimate $\boldsymbol{x}_{k-1} \longrightarrow \boldsymbol{x}_k$ and estimating the track's $\boldsymbol{q}_k$ at the vertex
- Smoothing is an update of the filtered estimates $\boldsymbol{q}_k$ for k < n, just using the final estimate of the vertex position $\boldsymbol{x}_n$

$$\boldsymbol{x}_k(\boldsymbol{x}_{k-1}) = \boldsymbol{x}_{k-1} =: \boldsymbol{x}$$

$$\boldsymbol{p}_k = \boldsymbol{h}_k(\boldsymbol{x}, \boldsymbol{q}_k) + \boldsymbol{\epsilon}_k, \quad \mathrm{cov}(\boldsymbol{\epsilon}_k) = \boldsymbol{V}_k = \boldsymbol{G}_k{}^{-1}$$

- Taylor expansion

$$\boldsymbol{h}_k(\boldsymbol{x}, \boldsymbol{q}_k) \approx \boldsymbol{h}_k(\boldsymbol{x}_e, \boldsymbol{q}_{k,e}) + \boldsymbol{A}_k(\boldsymbol{x} - \boldsymbol{x}_e) + \boldsymbol{B}_k(\boldsymbol{q}_k - \boldsymbol{q}_{k,e})$$
$$= \boldsymbol{A}_k\boldsymbol{x} + \boldsymbol{B}_k\boldsymbol{q}_k + \boldsymbol{c}_{k,e}$$

- with Jacobian matrices

$$\boldsymbol{A}_k = [\partial \boldsymbol{p}_k / \partial \boldsymbol{x}]_e \qquad \boldsymbol{B}_k = [\partial \boldsymbol{p}_k / \partial \boldsymbol{q}_k]_e$$

```
106  {-
107    data Prong = Prong N XMeas [QMeas] [Chi2] ...
108    data VHMeas = VHMeas XMeas [HMeas] ...
109    instance Monoid VHMeas where ...
110  -}
111  fit :: VHMeas -> Prong
112  fit = ksmooth . kFilter
113
114  kFilter :: VHMeas -> VHMeas
115  ksmooth :: VHMeas -> Prong
116  kFilter (VHMeas x ps) = VHMeas (foldl kAdd x ps) ps
117
118  kAdd :: XMeas -> HMeas -> XMeas
119  kAdd (XMeas v vv) (HMeas h hh w0) = kAdd' x_km1 p_k x_e q_e 1e6 0 whe
120    x_km1 = XMeas v (inv vv)
121    p_k   = HMeas h (inv hh) w0
122    x_e   = v
123    q_e   = Coeff.hv2q h v
124
125  kAdd' :: XMeas -> HMeas -> X3 -> Q3 -> Double -> Int -> XMeas
126  kAdd' (XMeas v0 uu0) (HMeas h gg w0) ve qe χ2_0 iter = x_k where
127    Jaco aa bb h0 = Coeff.expand ve qe
128    aaT   = tr aa; bbT = tr bb
129    ww    = inv (sw bb gg)
130    gb    = gg - sw gg (sw bbT ww)
131    uu    = uu0 + sw aa gb; cc = inv uu
132    m     = h - h0
133    v     = cc * (uu0 * v0 + aaT * gb * m)
134    dm    = m - aa * v
135    q     = ww * bbT * gg * dm
136    χ2    = scalar $ sw (dm - bb * q) gg + sw (v - v0) uu0
137    x_k   = if goodEnough χ2_0 χ2 iter
138              then XMeas v cc
139              else kAdd' (XMeas v0 uu0) (HMeas h gg w0) v q χ2 (iter+1)
```

Math description of algorithm translates ~directly into functional program

- State Vector: $\boldsymbol{x}, \boldsymbol{q}_k$ vertex position and momenta of tracks
- filter step: add a new tr already fitted with k–1 t position estimate $\boldsymbol{x}_{k-1} \longrightarrow \boldsymbol{x}_k$ and estimating the track's
- Smoothing is an update estimates $\boldsymbol{q}_k$ for k < n, estimate of the vertex p

$$\boldsymbol{x}_k(\boldsymbol{x}_{k-1}) = \boldsymbol{x}_{k-1} =: \boldsymbol{x}$$

$$\boldsymbol{p}_k = \boldsymbol{h}_k(\boldsymbol{x}, \boldsymbol{q}_k) + \boldsymbol{\epsilon}_k, \quad \mathrm{cov}(\boldsymbol{\epsilon}_k) = \boldsymbol{V}_k = \boldsymbol{G}_k{}^{-1}$$

- Taylor expansion

$$\boldsymbol{h}_k(\boldsymbol{x}, \boldsymbol{q}_k) \approx \boldsymbol{h}_k(\boldsymbol{x}_e, \boldsymbol{q}_{k,e}) + \boldsymbol{A}_k(\boldsymbol{x} - \boldsymbol{x}_e) + \boldsymbol{B}_k(\boldsymbol{q}_k - \boldsymbol{q}_{k,e})$$
$$= \boldsymbol{A}_k\boldsymbol{x} + \boldsymbol{B}_k\boldsymbol{q}_k + \boldsymbol{c}_{k,e}$$

- with Jacobian matrices

$$\boldsymbol{A}_k = [\partial \boldsymbol{p}_k / \partial \boldsymbol{x}]_e \qquad \boldsymbol{B}_k$$

type constructor functions defined elsewhere

KalFilter is "folding" over list of p_k, updating x on the way

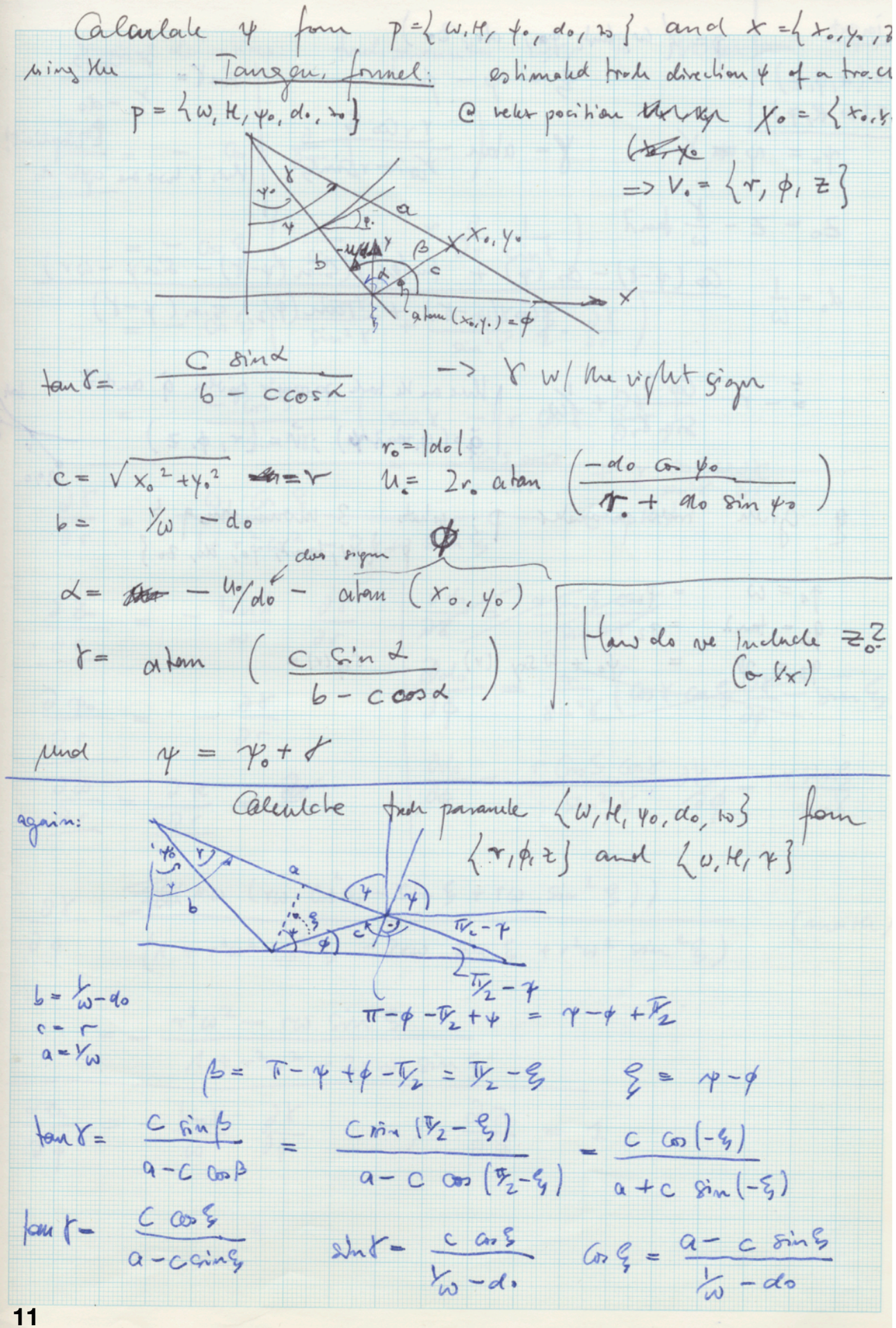
this gets called for each p_k, interfacing data to worker function `kAdd'`

this is really just the math from slide 3!
matrix helper funcs
`tr` is transpose $\mathbf{A}^T$
`sw` is $\mathbf{A}^T \cdot \mathbf{B} \cdot \mathbf{A}$
`inv` is inverse $\mathbf{A}^{-1}$

```haskell
{-
  data Prong = Prong N XMeas [QMeas] [Chi2] ...
  data VHMeas = VHMeas XMeas [HMeas] ...
  instance Monoid VHMeas where ...
-}
fit :: VHMeas -> Prong
fit = ksmooth . kFilter

kFilter :: VHMeas -> VHMeas
ksmooth :: VHMeas -> Prong
kFilter (VHMeas x ps) = VHMeas (foldl kAdd x ps) ps

kAdd :: XMeas -> HMeas -> XMeas
kAdd (XMeas v vv) (HMeas h hh w0) = kAdd' x_km1 p_k x_e q_e 1e6 0 whe
  x_km1 = XMeas v (inv vv)
  p_k   = HMeas h (inv hh) w0
  x_e   = v
  q_e   = Coeff.hv2q h v

kAdd' :: XMeas -> HMeas -> X3 -> Q3 -> Double -> Int -> XMeas
kAdd' (XMeas v0 uu0) (HMeas h gg w0) ve qe χ2_0 iter = x_k where
  Jaco aa bb h0 = Coeff.expand ve qe
  aaT   = tr aa; bbT = tr bb
  ww    = inv (sw bb gg)
  gb    = gg - sw gg (sw bbT ww)
  uu    = uu0 + sw aa gb; cc = inv uu
  m     = h - h0
  v     = cc * (uu0 * v0 + aaT * gb * m)
  dm    = m - aa * v
  q     = ww * bbT * gg * dm
  χ2    = scalar $ sw (dm - bb * q) gg + sw (v - v0) uu0
  x_k   = if goodEnough χ2_0 χ2 iter
            then XMeas v cc
            else kAdd' (XMeas v0 uu0) (HMeas h gg w0) v q χ2 (iter+1)
```

In Haskell, calling function *f*(*x*) is written **`f x`** **`[a]`** is a list of values e.g. **`[0,1,2,3…]`** equation"

call to calculate Jacobian matrices, implementing the linearized "measurement equation" to get us from "helices" to "vertex, momenta"

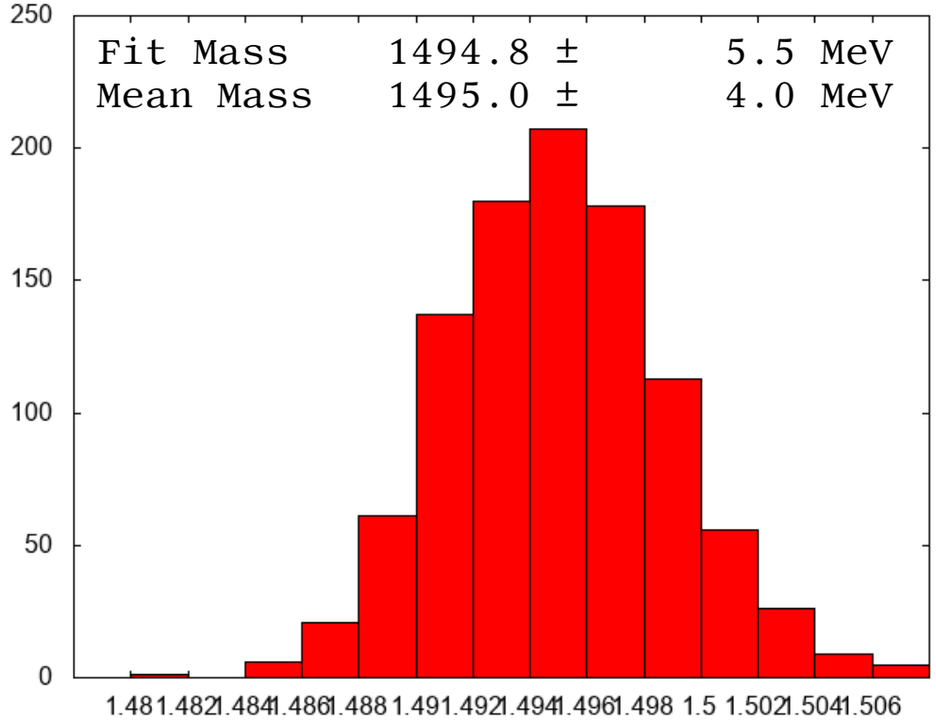


+ Math to move momentum vectors along helices etc…

```haskell
expand :: M -> M -> Jaco
expand v q = Jaco aa bb h0 where
  [xx, yy, z] = toList 3 v
  r = sqrt $ xx*xx + yy*yy
  phi  = atan2 yy xx
  [w, tl, psi] = toList 3 q
  -- some more derived quantities
  xi = mod' (psi - phi + 2.0*pi) (2.0*pi)
  cxi = cos xi
  sxi = sin xi
  oow = 1.0 / w
  rw  = r * w

  gamma = atan $ r*cxi/(oow - r*sxi)
  sg    = sin gamma
  cg    = cos gamma

  -- calculate transformed quantities
  psi0  = psi - gamma
  d0    = oow - (oow - r*sxi)/cg
  z0    = z - tl*gamma/w

  -- calc Jacobian
  [drdx, drdy, rdxidx, rdxidy] =
    if r /= 0 then [xx/r, yy/r, yy/r, -xx/r]
              else [0, 0, 0, 0]
  dgdvar0 =    1.0/(1.0 + rw*rw - 2.0*rw*sxi)
  dgdx    =    dgdvar0*(w*cxi*drdx + w*(rw - sxi)*rdxidx)
  dgdy    =    dgdvar0*(w*cxi*drdy + w*(rw - sxi)*rdxidy)
  dgdw    =    dgdvar0*r*cxi
  dgdpsi  =    dgdvar0*rw*(rw - sxi)

  --  fill matrix:
  -- d w / d r, d phi, d z
```

Testing, I/O, random numbers, Monads, …

- Initial test: use a sample of 6-prong tau decay (Aleph) $Z \rightarrow \tau\ (\tau \rightarrow 5\ \pi)$
  - vertex fit to constrain ν_τ mass in $\tau \rightarrow 5\ \pi\ \nu_\tau$
  - calc invariant mass of vertexed momenta
  - test fit robustness and error propagation by comparing propagated error with MC of randomized helices



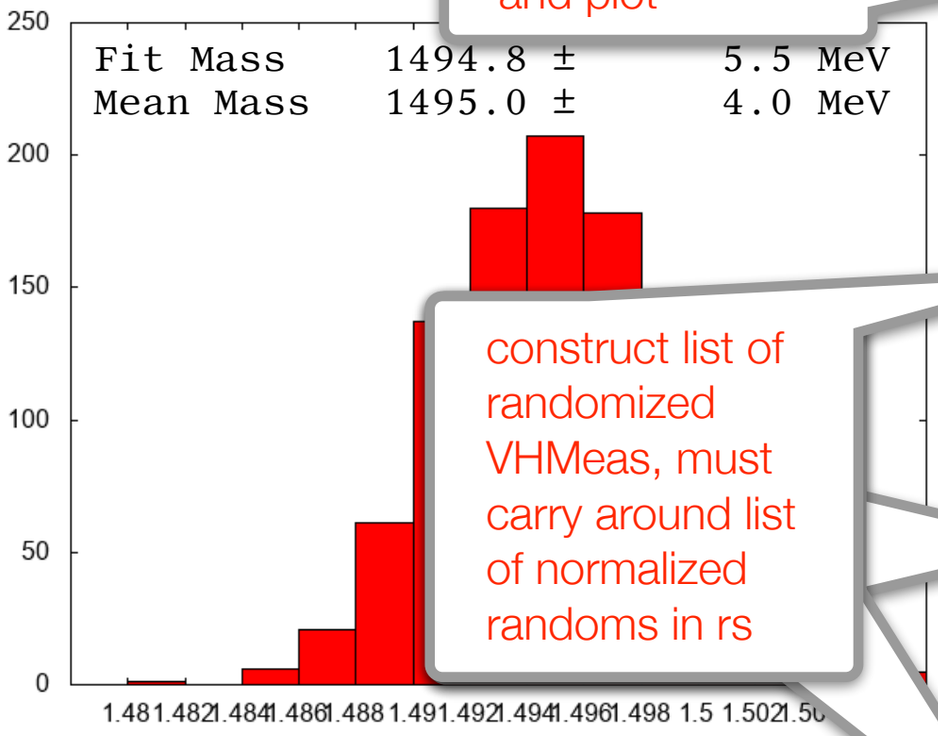
- Now testing it w/ CMS data for primary vertex fit, playing with adaptive vertex finding etc —> it's already a rather flexible vertex tool set!

```
53  {-
54  VHMeas v hl <- hSlurp thisFile
55  doRandom 1000 (VHMeas v (hFilter hl [0,2,3,4,5])) -}
56  doRandom :: Int -> VHMeas -> IO ()
57  doRandom cnt vm = do
58    let Prong _ _ ql _ = fit vm
59    putStrLn $ "Fit Mass  " ++ (show . invMass . map q2p) ql
60
61    g <- newStdGen
62    let hf :: V.Vector Double
63        hf = V.fromListN cnt $
64                unfoldr (randomize vm fitMass) . normals $ g
65        (mean, var) = meanVariance hf
66    putStrLn $ "Mean Mass " ++ show (MMeas mean (sqrt var))
67    let hist = histogram binSturges (V.toList hf)
68    _ <- plot "invMass.png" hist
69    return ()
70
71  randomize :: VHMeas -> (VHMeas -> Double) -> [Double] -> Maybe (Double, [Double
72  randomize vh f rs = Just (f vh', rs') where
73    (vh', rs') = randVH vh rs
74
75  -- randomize the helices in the supplied VHMeas
76  -- and return randomized VHMeas and remaining randoms list
77  randVH :: VHMeas -> [Double] -> (VHMeas, [Double])
78  randVH (VHMeas v hl) rs = (VHMeas v hl', rs') where
79    (rs', hl') = mapAccumL randH rs hl
80
81  -- randomize a single helix parameters measurment, based on the cov matrix
82  -- return randomized helix and "remaining" random numbers
83  randH :: [Double] -> HMeas -> ([Double], HMeas)
84  randH (r0:r1:r2:r3:r4:rs) (HMeas h hh w0) = (rs, HMeas h' hh w0) where
85    h' = v5 $ zipWith (+) (l5 h) (l5 (chol hh * v5 [r0,r1,r2,r3,r4]))
```


https://github.com/LATBauerdick/fv.hs

Testing, I/ numbers,

- Initial test: use a sam ecay (Aleph) $Z \to \tau$ ($\tau \to$
  - vertex fit to cons
  - calc invariant ma
  - test fit robustnes n by comparing propa f randomized heli



- Now testing it w/ CMS data for primary vertex fit, playing with adaptive vertex finding etc —> it's already a rather flexible vertex tool set!

this gets called from test harness

do the fit, calc mass, print

Construct vector of mass values calculated from randomized VHMeas

print, histogram and plot

construct list of randomized VHMeas, must carry around list of normalized randoms in rs

Creates an infinite (lazy) list of normal distributed randoms

```haskell
{-
VHMeas v hl <- hSlurp thisFile
doRandom 1000 (VHMeas v (hFilter hl [0,2,3,4,5])) -}
doRandom :: Int -> VHMeas -> IO ()
doRandom cnt vm = do
  let Prong _ _ ql _ = fit vm
  putStrLn $ "Fit Mass  " ++ (show . invMass . map q2p) ql

  g <- newStdGen
  let hf :: V.Vector Double
      hf = V.fromListN cnt $
               unfoldr (randomize vm fitMass) . normals $ g
      (mean, var) = meanVariance hf
  putStrLn $ "Mean Mass " ++ show (MMeas mean (sqrt var))
  let hist = histogram binSturges (V.toList hf)
  _ <- plot "invMass.png" hist
  return ()

randomize :: VHMeas -> (VHMeas -> Double) -> [Double] -> Maybe (Double, [Double
randomize vh f rs = Just (f vh', rs') where
  (vh', rs') = randVH vh rs

-- randomize the helices in the supplied VHMeas
-- and return randomized VHMeas and remaining randoms list
randVH :: VHMeas -> [Double] -> (VHMeas, [Double])
randVH (VHMeas v hl) rs = (VHMeas v hl', rs') where
  (rs', hl') = mapAccumL randH rs hl

-- randomize a single helix parameters measurment, based on the cov matrix
-- return randomized helix and "remaining" random numbers
randH :: [Double] -> HMeas -> ([Double], HMeas)
randH (r0:r1:r2:r3:r4:rs) (HMeas h hh w0) = (rs, HMeas h' hh w0) where
  h' = v5 $ zipWith (+) (l5 h) (l5 (chol hh * v5 [r0,r1,r2,r3,r4]))
```


https://github.com/LATBauerdick/fv.hs

Some Observations and Conclusions

- **This is really fun!** For me, learning a new language was mind opening
- **This is powerful stuff:**
  - functional declarative description of application domain problem,
    which in HEP almost always are “advanced” math problems anyway
  - tools to deal with complexity are powerful math constructs (e.g. category theory) which might be a great match to physics algorithms, and lend themselves to efficient hardware implementations
  - compiler, runtime, language features to optimize, parallelize, vectorize, put on GPUs, FPGAs etc
  - I do want want strong typing (Idris and Haskell vs. Clojure!) —> compiler supports type inference
- **There’s a learning curve; does FP help writing comprehensible, maintainable s/w? Maybe not!**
  - certainly can’t expect a physicist to learn e.g. Haskell just to make a few plots or to try out ideas
  - C++ w/ templates etc is really hard, too, and FP enables an appealing cleanness and brevity
  - also “division of labor” and separation of concerns: math algorithm vs run-time optimization etc
- **DSL for HEP, based on FP, could be very powerful, might be best bet for physicists use**
  - after all, ROOT C++ macro language is a “DSL”, too — just not a very clean one…
- **I see many reasons why a closer look at FP in HEP would be very worthwhile**
  - Lots to learn and do, next steps: interfacing to CMS data sets and looking at performance
  - Am looking for “fellow travelers” on this exciting journey!

Podcast series https://www.functionalgeekery.com/category/podcasts/

Category Theory: B.Milewsky https://www.youtube.com/watch?v=I8LbkfSSR58