Modernisation of RooFit

S. Hageboeck (CERN, EP-SFT) for the ROOT team

ROOT
Data Analysis Framework

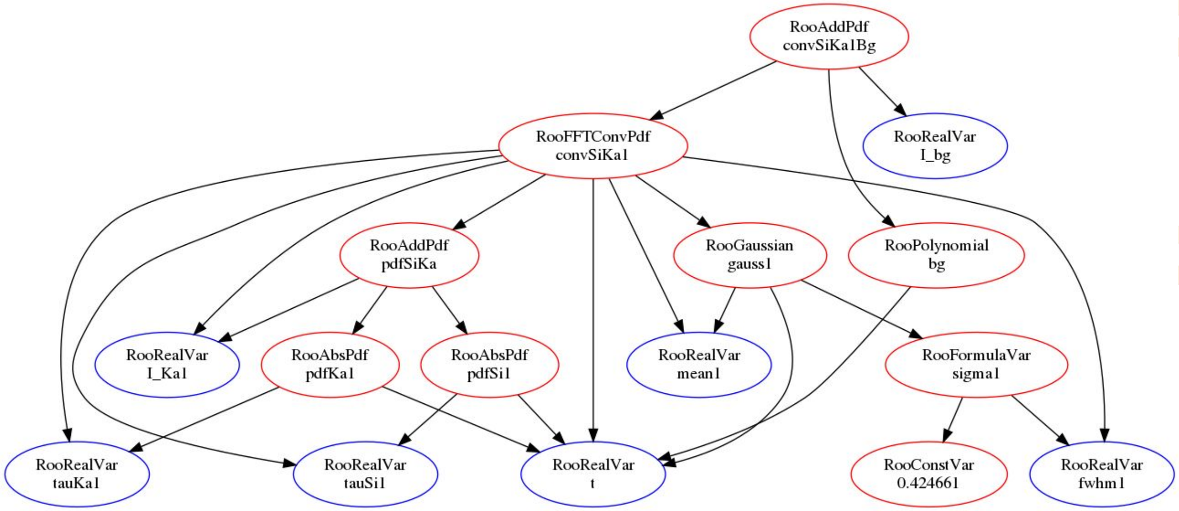
https://root.cern

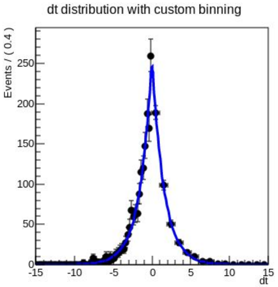
Introduction

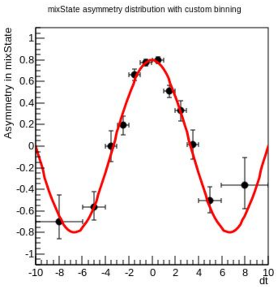
- RooFit used in all LHC (+ other) experiments
  - Express statistical models (binned / unbinned likelihoods)
  - **Parameter estimation** (i.e. errors!)
  - **Statistical tests** (e.g. Higgs Discovery)
- Development started before ~2005 until ~2011, not touched much in recent years

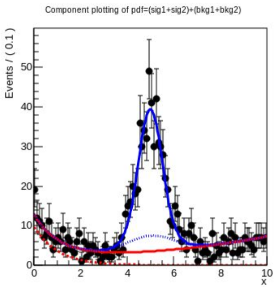
- **Challenges**: Data statistics in LHC’s Run 3
  - More events to be processed (*e.g.* LHCb: ~10x more)
  - Higher statistics → allow for more complex models
  - Goal: speed up >= 10x

RooFit’s Strengths




dt distribution with custom binning


mixState asymmetry distribution with custom binning


Component plotting of pdf=(sig1+sig2)+(bkg1+bkg2)

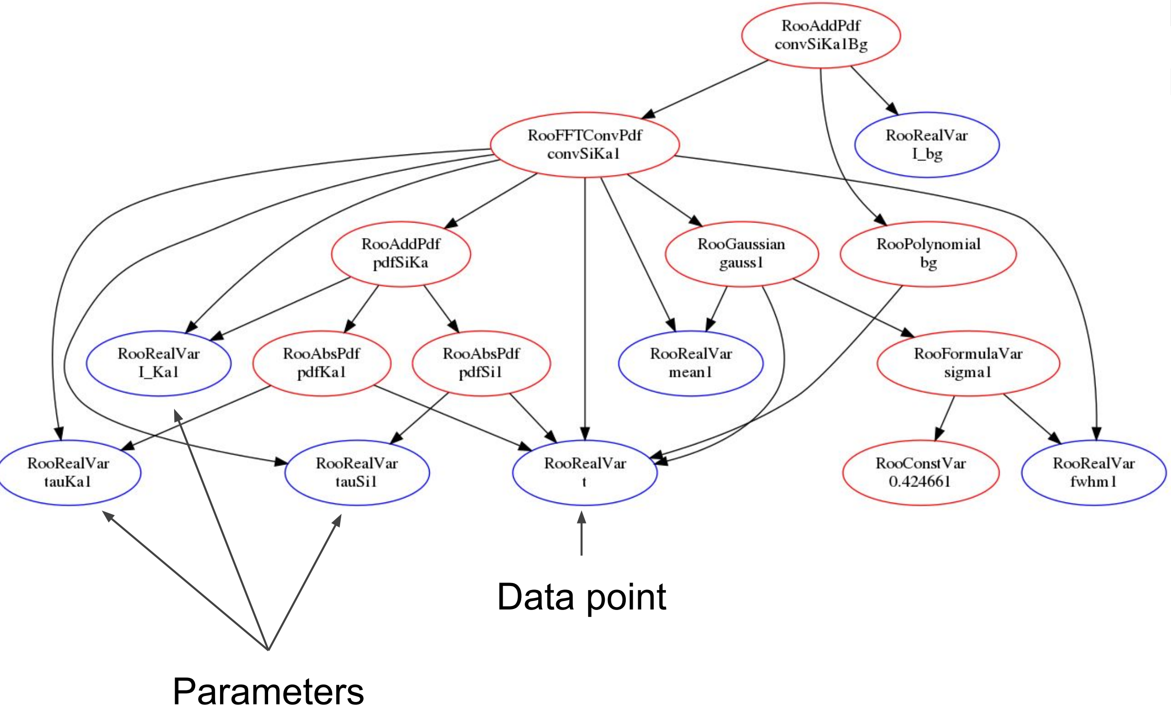
- Compose PDFs as trees of functions & variables

  RooFit classes can be stitched together to evaluate complex functions

- Each PDF can be:
  - evaluated
  - normalised
  - fitted to data
  - plotted
  - Parameter estimation
  - Toy experiments
  - …

RooFit’s Weakness

A random PDF
from a question in the forum



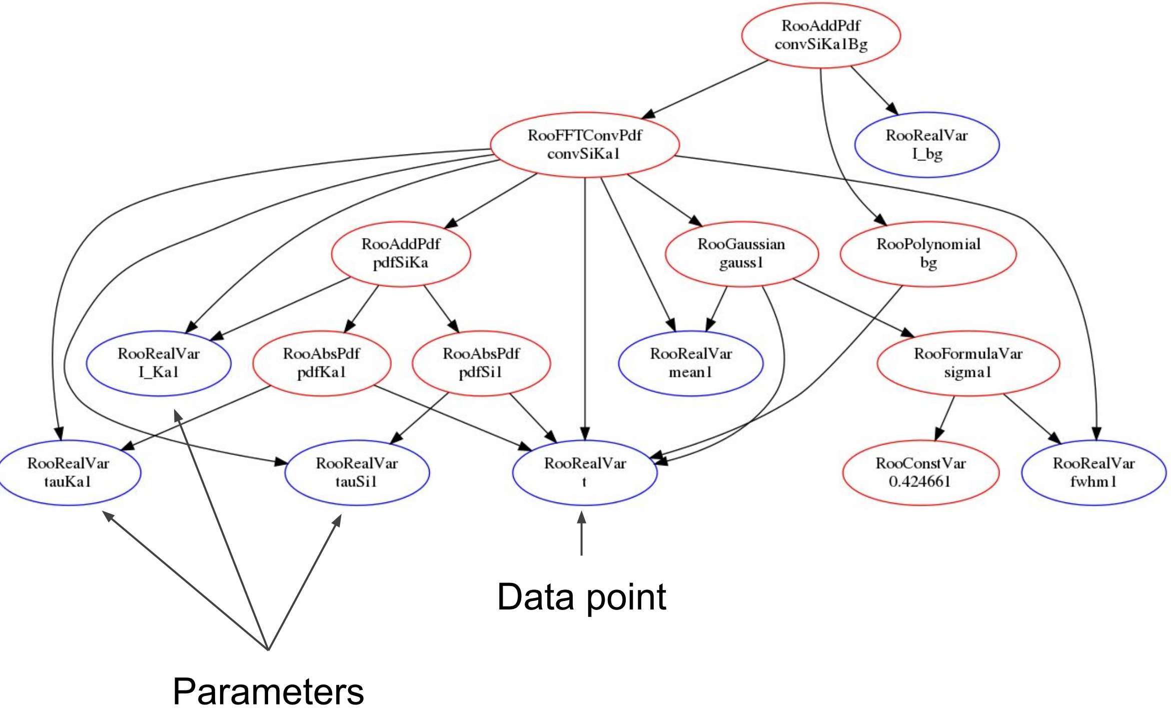
Likelihood:
Probability of observing the data given a probability model

Maximum-likelihood fit:

- ▶ Adjust parameters until likelihood maximal
- ▶ **One virtual call per**:
  - Data point
  - PDF node
  - Set of parameters tested
- ▶ Large fit: 1M data points * 1000 elements * 1000 fit steps
= 1 trillion calls
- ▶ + 1 billion normalisation integrals when parameters change

RooFit’s Weakness

A random PDF
from a question in the forum



Data point

Parameters

Flow of data:

- **A single** data point is loaded into the variables
- The whole expression tree (except for constant branches) is evaluated
- By the time execution returns to the data point, the cache line almost certainly disappeared
    - Some simple profiling for a large fit model:
      50% of data points from DRAM
- 0 chance to vectorise computations

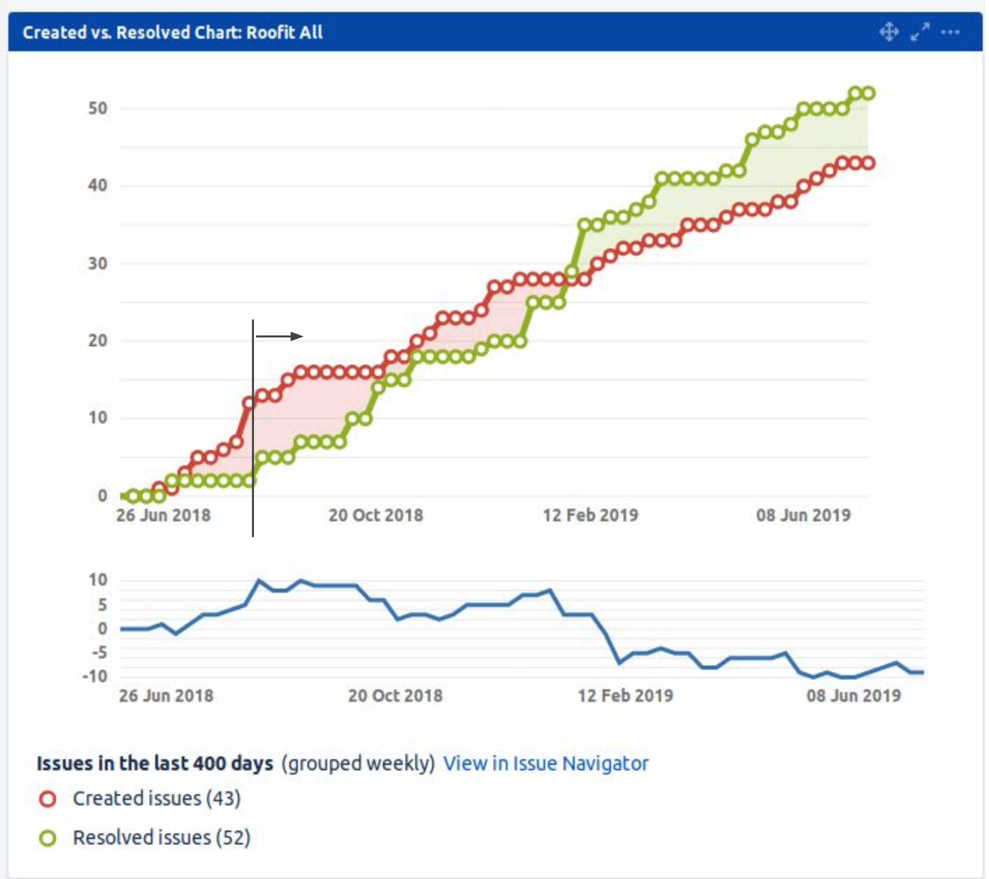
My Initial Plan for RooFit

1. Fix the most pressing issues
2. LinkedList → std::vector<RooAbsArg*>
3. Batched evaluation
   - Walk expression tree only once for all data points
   - Reduce number of virtual calls by factor of batch size
   - No change of state, no copying subtree (→ threads)
   - Data come as std::vector<double> and are accessed consecutively (cache-friendly)
4. Vectorise loops inside batches
5. Batched generation of toy data
   - Bottleneck for some analyses
6. Threads

https://sft.its.cern.ch/jira/browse/ROOT-9815

Pressing Issues

- Static destruction order fiasco crashed ROOT when trying to quit after using RooFit
- Memory leaks were preventing toy studies
- Unable to read ROOT 5 workspaces because of cint ←→ cling differences (*e.g.* Higgs discovery)
- + Most common problems in the forum

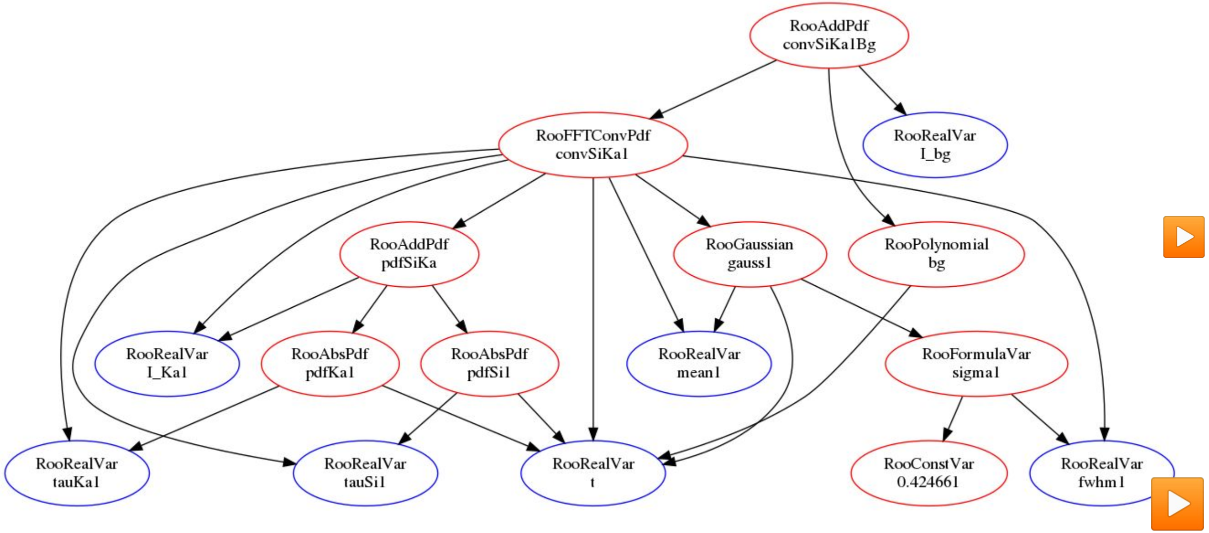


Issues in the last 400 days (grouped weekly) View in Issue Navigator

○ Created issues (43)

○ Resolved issues (52)

RooAbsCollection



Collections:

▶ Expression tree (+ almost everything else in RooFit) stored as RooLinkedList<RooAbsArg*>

- Often small search & iterates
- Optional hash table to compensate slow iterations

▶ Toy MonteCarlo generation:

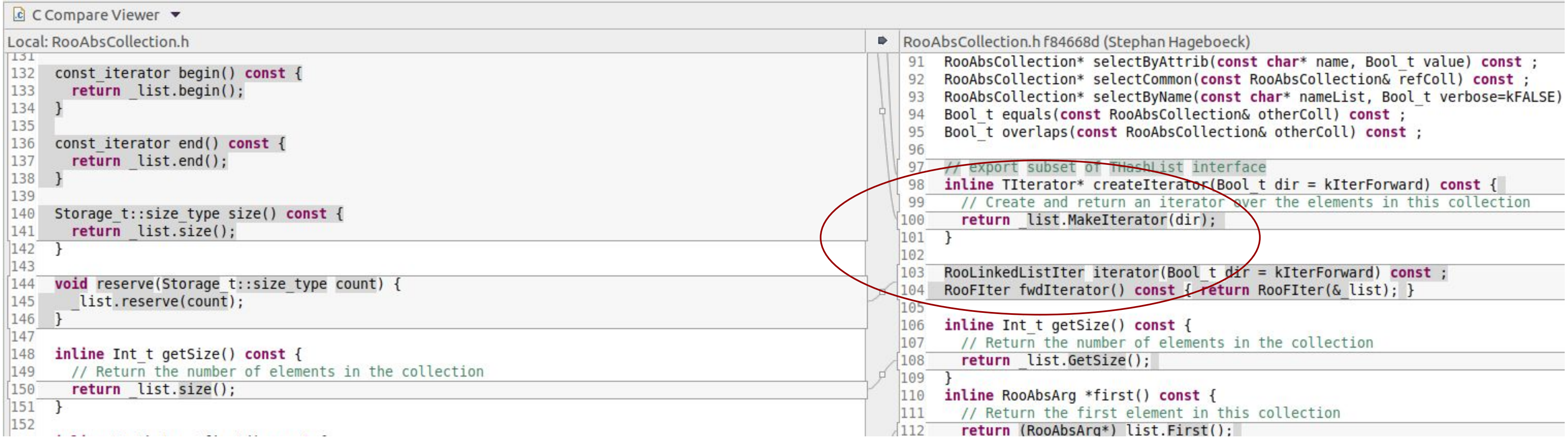
- ~50% of L3 misses due to linked list + hash table operations

▶ **The plan**:

- Replace LinkedList by std::vector
- Provide STL-like interface

The Challenge

- Axel: “How much user code are you going to break?”
→ The answer would have been "Almost everything" …
- The old collections directly expose the underlying storage implementation through the iterators



Local: RooAbsCollection.h

```
  const_iterator begin() const {
    return _list.begin();
  }

  const_iterator end() const {
    return _list.end();
  }

  Storage_t::size_type size() const {
    return _list.size();
  }

  void reserve(Storage_t::size_type count) {
    _list.reserve(count);
  }

  inline Int_t getSize() const {
    // Return the number of elements in the collection
    return _list.size();
  }

```

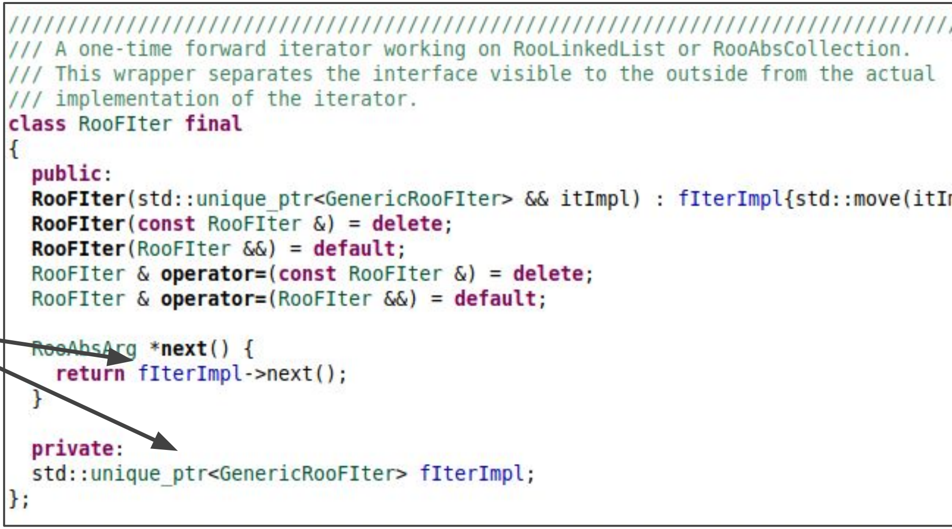
RooAbsCollection.h f84668d (Stephan Hageboeck)

```
  RooAbsCollection* selectByAttrib(const char* name, Bool_t value) const ;
  RooAbsCollection* selectCommon(const RooAbsCollection& refColl) const ;
  RooAbsCollection* selectByName(const char* nameList, Bool_t verbose=kFALSE)
  Bool_t equals(const RooAbsCollection& otherColl) const ;
  Bool_t overlaps(const RooAbsCollection& otherColl) const ;

  // export subset of THashList interface
  inline TIterator* createIterator(Bool_t dir = kIterForward) const {
    // Create and return an iterator over the elements in this collection
    return _list.MakeIterator(dir);
  }

  RooLinkedListIter iterator(Bool_t dir = kIterForward) const ;
  RooFIter fwdIterator() const { return RooFIter(&_list); }

  inline Int_t getSize() const {
    // Return the number of elements in the collection
    return _list.GetSize();
  }
  inline RooAbsArg *first() const {
    // Return the first element in this collection
    return (RooAbsArg*)_list.First();
```

Solution

- Three kinds of old iterators need to be supported (all in use)
- RooLinkedList needs continued support (user code)

- Implemented wrapper that delegates to RooLinkedList or STL as needed
- Downside: slower
  - Extra layer with virtual dispatch
  - Need to create&destroy iterators and hand into userland

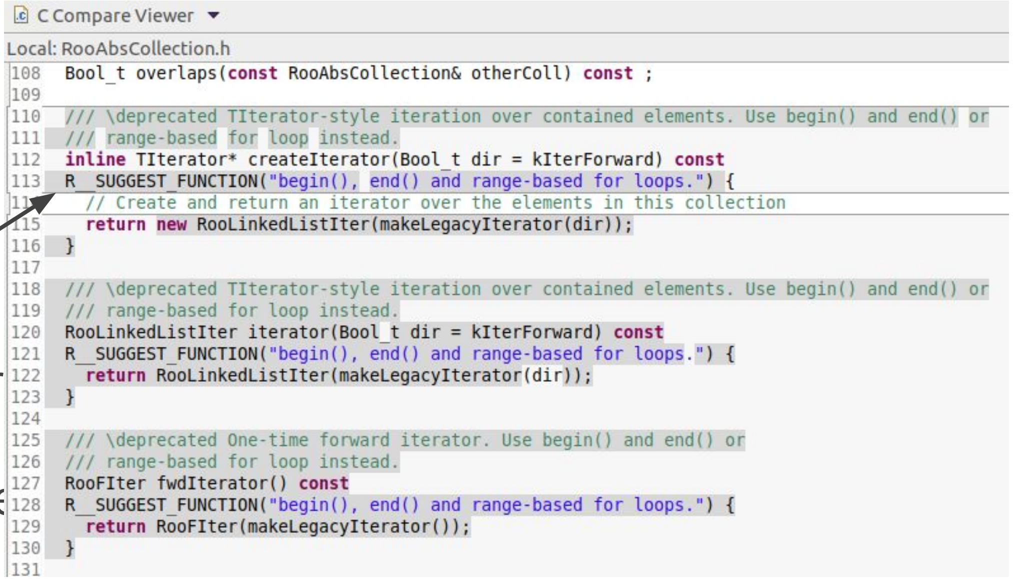


```
///////////////////////////////////////////////////////////////////////////////
/// A one-time forward iterator working on RooLinkedList or RooAbsCollection.
/// This wrapper separates the interface visible to the outside from the actual
/// implementation of the iterator.
class RooFIter final
{
  public:
  RooFIter(std::unique_ptr<GenericRooFIter> && itImpl) : fIterImpl{std::move(itIm
  RooFIter(const RooFIter &) = delete;
  RooFIter(RooFIter &&) = default;
  RooFIter & operator=(const RooFIter &) = delete;
  RooFIter & operator=(RooFIter &&) = default;

  RooAbsArg *next() {
    return fIterImpl->next();
  }

  private:
  std::unique_ptr<GenericRooFIter> fIterImpl;
};
```

The Legacy Iterators now

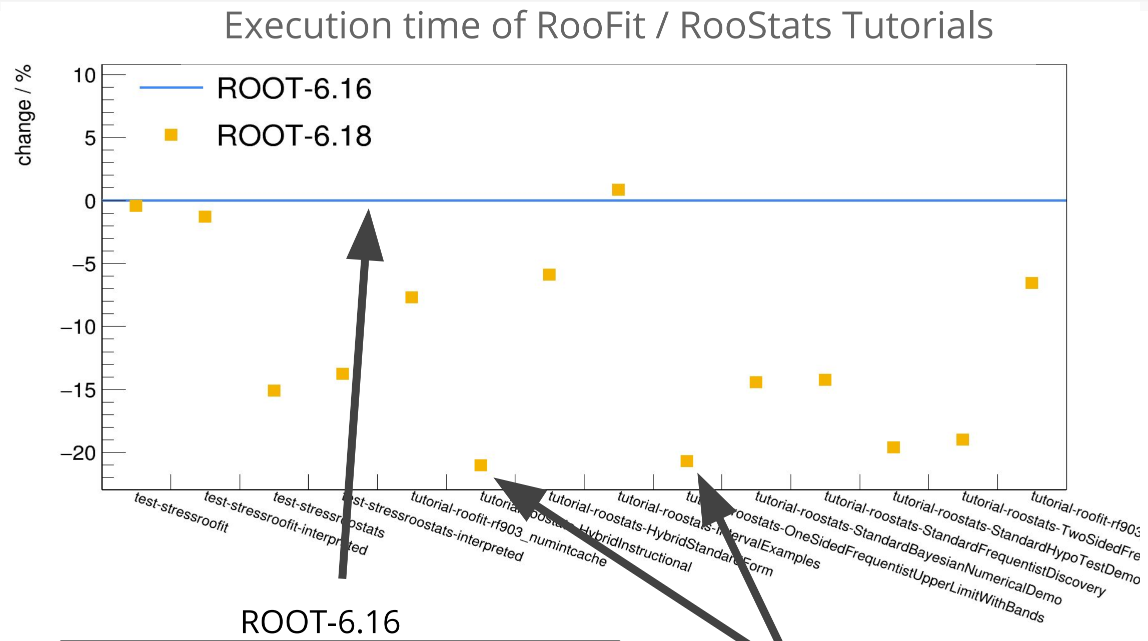
- All legacy iterators work
- 10 - 20% slower than before

- Flagged with R__SUGGEST_ALTERNATIVE:
  - Requested during ROOT user workshop
  - Flags functions/classes whose use is discouraged, but won’t be fully deprecated
  - https://github.com/root-project/root/pull/3100

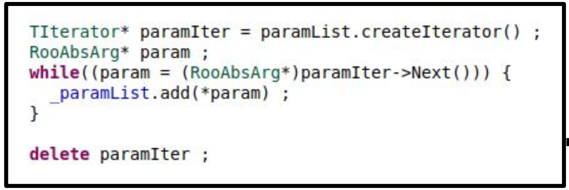


```cpp
Bool_t overlaps(const RooAbsCollection& otherColl) const ;

/// \deprecated TIterator-style iteration over contained elements. Use begin() and end() or
/// range-based for loop instead.
inline TIterator* createIterator(Bool_t dir = kIterForward) const
R__SUGGEST_FUNCTION("begin(), end() and range-based for loops.") {
  // Create and return an iterator over the elements in this collection
  return new RooLinkedListIter(makeLegacyIterator(dir));
}

/// \deprecated TIterator-style iteration over contained elements. Use begin() and end() or
/// range-based for loop instead.
RooLinkedListIter iterator(Bool_t dir = kIterForward) const
R__SUGGEST_FUNCTION("begin(), end() and range-based for loops.") {
  return RooLinkedListIter(makeLegacyIterator(dir));
}

/// \deprecated One-time forward iterator. Use begin() and end() or
/// range-based for loop instead.
RooFIter fwdIterator() const
R__SUGGEST_FUNCTION("begin(), end() and range-based for loops.") {
  return RooFIter(makeLegacyIterator());
}
```

Iterating Through Collections in RooFit

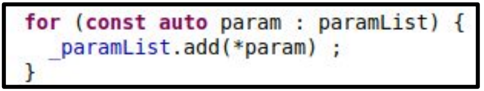


ROOT-6.16



```
TIterator* paramIter = paramList.createIterator() ;
RooAbsArg* param ;
while((param = (RooAbsArg*)paramIter->Next())) {
  _paramList.add(*param) ;
}

delete paramIter ;
```

ROOT-6.18
20% faster



```
for (const auto param : paramList) {
  _paramList.add(*param) ;
}
```

- New iterators look & feel like STL
- They are ~ 25% faster
- Same results
- **No code changes for users**
- Updating makes loops faster

My Plan for RooFit

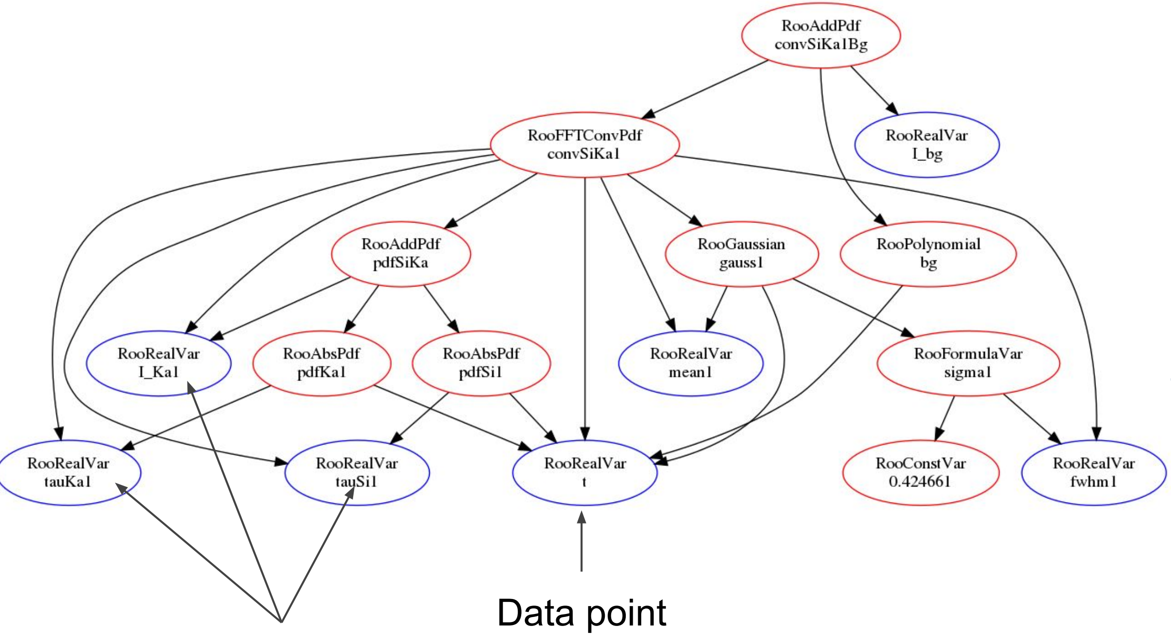
1. Fix the most pressing issues ROOT 6.16
2. LinkedList → std::vector<RooAbsArg*> ROOT 6.18
   - Much more memory friendly, 20% faster iterate/allocate/destroy + *much* faster index access
3. Batched evaluation
   - Walk expression tree only once for all data points
   - Reduce number of virtual calls by factor of batch size
   - No change of state, no copying subtree (→ threads)
   - Data come as std::vector<double> and are accessed consecutively (cache-friendly)
4. Vectorise loops inside batches
5. Batched generation of toy data
   - Bottleneck for some analyses
6. Threads

https://sft.its.cern.ch/jira/browse/ROOT-9815

Batched function evaluations

A random PDF

from a question in the forum



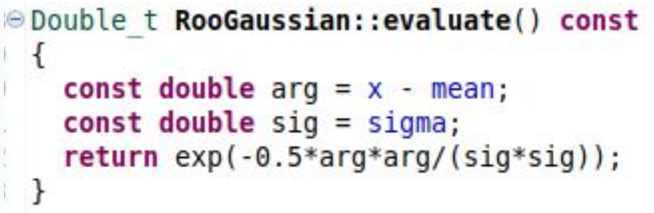
Parameters

Data point
→ Data array

- **Now: A single** data point is loaded into the variables
- The whole (minus cached branches) expression tree is walked over
- Execution returns to the data point, cache line disappeared
  - Simple profiling:
    50% L3 misses
- 0 chance to vectorise computations
- **My plan:**
  - **Evaluate a batch of data points in a single call**
  - **Exploit vectorised fp instructions**

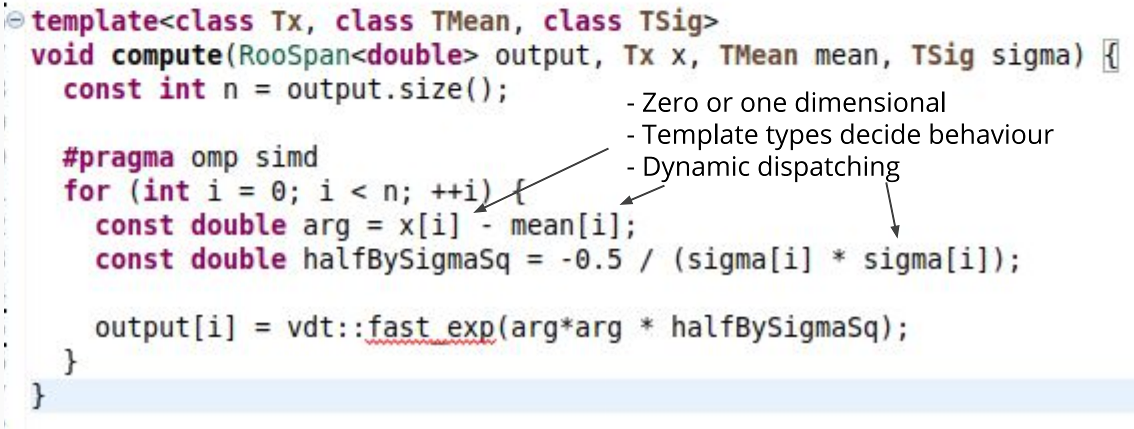
Batched and Auto-Vectorised Gaussian

Old:



```
Double_t RooGaussian::evaluate() const
{
  const double arg = x - mean;
  const double sig = sigma;
  return exp(-0.5*arg*arg/(sig*sig));
}
```

New:



```
template<class Tx, class TMean, class TSig>
void compute(RooSpan<double> output, Tx x, TMean mean, TSig sigma) {
  const int n = output.size();

  #pragma omp simd
  for (int i = 0; i < n; ++i) {
    const double arg = x[i] - mean[i];
    const double halfBySigmaSq = -0.5 / (sigma[i] * sigma[i]);

    output[i] = vdt::fast_exp(arg*arg * halfBySigmaSq);
  }
}
```

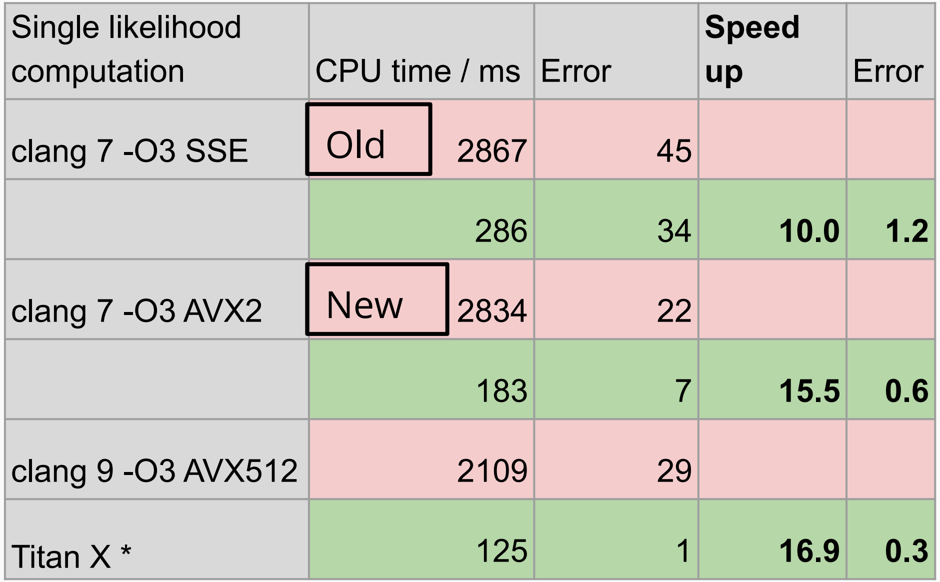
- Zero or one dimensional
- Template types decide behaviour
- Dynamic dispatching

Challenge:

- Whether a node is a parameter or a batch is decided at run time (might even change at RT)
- Solved with classes that either collapse to a constant or an array (completely inlinable)
- VDT math functions for auto vectorisation

Batch & Vectorisation Benchmark

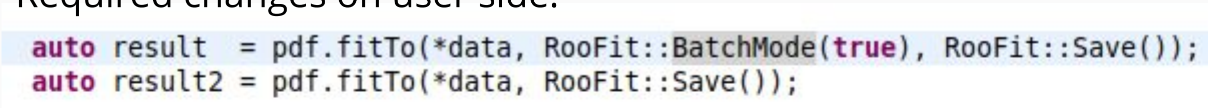
L(x | P) = Gauss(x | P1) + Gauss(x | P2) + Exp(x | P3)



Single likelihood computation	CPU time / ms	Error	**Speed up**	Error
clang 7 -O3 SSE	Old 2867	45		
	286	34	**10.0**	**1.2**
clang 7 -O3 AVX2	New 2834	22		
	183	7	**15.5**	**0.6**
clang 9 -O3 AVX512	2109	29		
Titan X *	125	1	**16.9**	**0.3**

- Optimised Gauss, Exp, Sum, Poisson
- Batches & better cache locality result in 10x faster likelihood computation
- With AVX2, 16x faster LH possible
- (*) AVX512 should allow for more speed up, but CPU likely throttling

Required changes on user side:



```
auto result  = pdf.fitTo(*data, RooFit::BatchMode(true), RooFit::Save());
auto result2 = pdf.fitTo(*data, RooFit::Save());
```

Batch & Vectorisation Full Fit

L(x | P) = Gauss(x | P1) + Gauss(x | P2) +
Exp(x | P3)

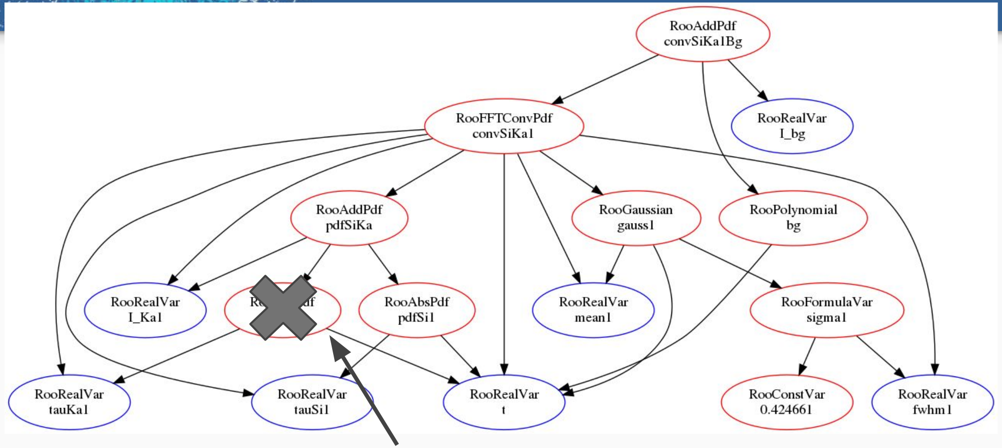
Full fit + error estimation	CPU time / s	**Speed up**
clang 7 -O3 SSE	9.61	
	2.45	**3.9**
clang 7 -O3 AVX2	9.97	
	1.32	**7.5**
clang 9 -O3 AVX512	6.53	
Titan X *	0.68	**9.7**

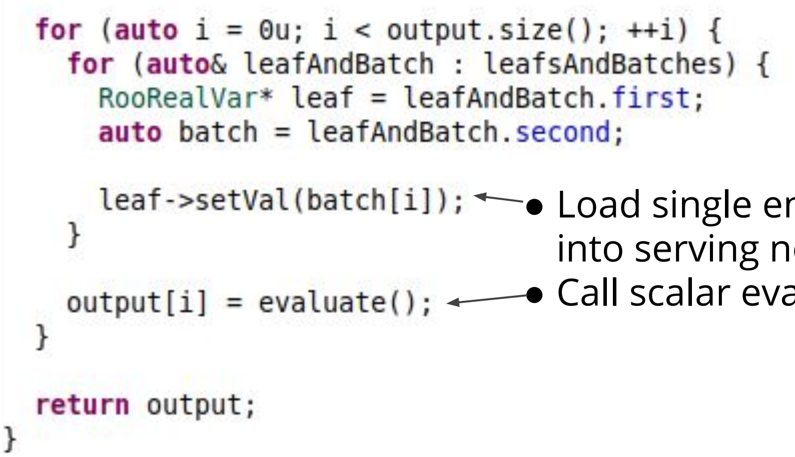
▶ Full fit can be 7 to 10 times faster with batches and vectorisation

▶ Results identical to 10E-14

- Unit tests running batch against scalar code
- Minimal differences expected (e.g. vdt::exp vs std::exp)

Compatibility Mode





```
  for (auto i = 0u; i < output.size(); ++i) {
    for (auto& leafAndBatch : leafsAndBatches) {
      RooRealVar* leaf = leafAndBatch.first;
      auto batch = leafAndBatch.second;

      leaf->setVal(batch[i]);
    }

    output[i] = evaluate();
  }

  return output;
}
```

- Load single entries into serving nodes
- Call scalar evaluate()

- Only a few PDFs batched & vectorised
- My summer student Manos will update more

- Remaining PDFs can run in "compatibility mode"
  - Scalar loop in inherited from base class
  - Fill batch & return
  - ~25% faster

My Plan for RooFit

1. Fix the most pressing issues ROOT 6.16
2. LinkedList → std::vector<RooAbsArg*> ROOT 6.18
   - Much more memory friendly, faster to iterate/allocate/destroy/index access
3. Batched evaluation

   Working demo being finalised

   - Walk expression tree only once for all data points
   - Reduce number of virtual calls by factor of batch size
   - No change of state, no copying subtree (→ threads)
   - Data come as std::vector<double> and are accessed consecutively (cache-friendly)
4. Vectorise loops inside batches Up to 10x speed up
5. Batched & threaded generation of toy data
   - Bottleneck for some analyses
6. Threads

https://sft.its.cern.ch/jira/browse/ROOT-9815

Summary

- Users are starting to realise that RooFit is evolving again
- See no obstacles to have the batch & vectorise demo in ROOT 6.20 (autumn / winter) **with >= 10x speed up**
- MP / MT
  - RooFit has simple MP capabilities, batch mode + MP needs testing
  - Will test threads soon
  - Batch & vectorise interface designed with threads in mind

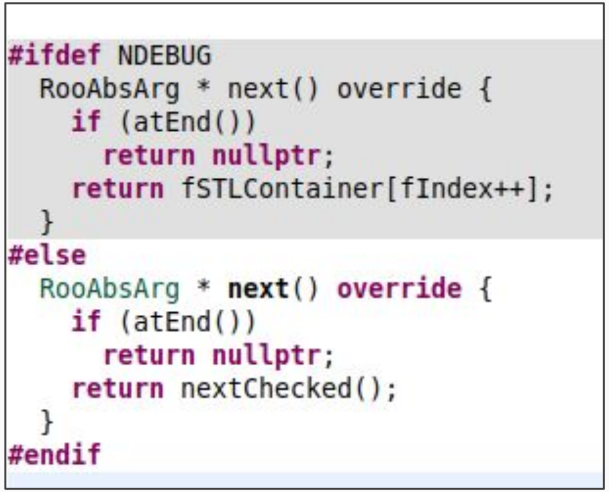
    Caveat: PDF normalisation has lots of thread-hostile code. Expect to need lots of locks in the beginning.
- More ideas in pipeline:
  - RNTuple as storage backend ROOT-10206 to allow for bulk reading
  - Likelihood gradient parallelisation (collaboration with NIKHEF)

Backup

The Challenge II

- RooLinkedList:
  - Remove/add/replace before and after current iterator
  - No reallocations → iterator valid

- Solution: Legacy-to-STL adapters count
  - Can remove/add after iterator
  - Can replace everywhere
  - Safe also if reallocating
  - **But: Will break** when removing/adding **before** iterator



```
#ifdef NDEBUG
  RooAbsArg * next() override {
    if (atEnd())
      return nullptr;
    return fSTLContainer[fIndex++];
  }
#else
  RooAbsArg * next() override {
    if (atEnd())
      return nullptr;
    return nextChecked();
  }
#endif
```