RooFit Programmers Tutorial

Wouter Verkerke (UC Santa Barbara)
David Kirkby (UC Irvine)
RooFit users tutorial

RooFit design philosophy

Mathematical concepts as C++ objects
General rules for RooFit classes
RooFit core design philosophy

- Mathematical objects are represented as C++ objects

<table>
<thead>
<tr>
<th>Mathematical concept</th>
<th>RooFit class</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable $x$</td>
<td>RooRealVar</td>
</tr>
<tr>
<td>function $f(x)$</td>
<td>RooAbsReal</td>
</tr>
<tr>
<td>PDF $f(x)$</td>
<td>RooAbsPdf</td>
</tr>
<tr>
<td>space point $\vec{x}$</td>
<td>RooArgSet</td>
</tr>
<tr>
<td>integral $\int_{x_{\text{min}}}^{x_{\text{max}}} f(x) dx$</td>
<td>RooRealIntegral</td>
</tr>
<tr>
<td>list of space points</td>
<td>RooAbsData</td>
</tr>
</tbody>
</table>

Wouter Verkerke, UCSB
RooFit core design philosophy

- Represent relations between variables and functions as client/server links between objects

Math

\[ f(x,y,z) \]

RooFit diagram

```
RooRealVar x("x","x",5) ;
RooRealVar y("y","y",5) ;
RooRealVar z("z","z",5) ;
RooBogusFunction f("f","f",x,y,z) ;
```
RooFit core design philosophy

- Composite functions → Composite objects

\[
f(w, z) \quad g(x, y) \quad \rightarrow \quad f(g(x, y), z) = f(x, y, z)
\]

Math

\[
f(w, z) = g(x, y)
\]

RooFit diagram

```
RooRealVar x("x","x",2) ;
RooRealVar y("y","y",3) ;
RooGooFunc g("g","g",x,y) ;
RooRealVar w("w","w",0) ;
RooRealVar z("z","z",5) ;
RooFooFunc f("f","f",w,z) ;
```

```
RooRealVar x("x","x",2) ;
RooRealVar y("y","y",3) ;
RooGooFunc g("g","g",x,y) ;
RooRealVar w("w","w",0) ;
RooRealVar z("z","z",5) ;
RooFooFunc f("f","f",g,z) ;
```

Wouter Verkerke, UCSB
RooFit core design philosophy

- Represent integral as an object, instead of representing integration as an action

Math

\[ g(x,m,s) \quad \xrightarrow{\int} \quad \int_{x_{\text{min}}}^{x_{\text{max}}} g(x,m,s) dx = G(m,s,x_{\text{min}},x_{\text{max}}) \]

RooFit diagram

RooFit code

\[
\begin{align*}
\text{RooRealVar} & \quad \text{x}("x","x",2,-10,10) \\
\text{RooRealVar} & \quad \text{s}("s","s",3) \\
\text{RooRealVar} & \quad \text{m}("m","m",0) \\
\text{RooGaussian} & \quad g("g","g",x,m,s) \\
\text{RooAbsReal} & \quad \ast G = \\
& \quad g.\text{createIntegral}(\text{x}) \\
\end{align*}
\]
RooFit designed goals for easy-of-use in macros

• Mathematical concepts mimicked as much as possible in class design
  – Intuitive to use

• Every object that can be constructed through composition should be fully functional
  – No implementation level restrictions
  – No zombie objects

• All methods must work on all objects
  – Integration, toyMC generation, etc
  – No half-working classes

Only current exception: convolutions
RooFit designed for easy-of-use in macros

- At the same time, RooFit class structure designed to facilitate lightweight implementation-level classes
  - All value representing classes inherit from a common base class: RooAbsArg

- RooAbsArg and other intermediate abstract base classes handle bulk of the logistics
  - In most cases only one method is required: evaluate()
  - Implementation of common techniques such as integral calculation or ToyMC generator not mandatory
  - Base classes provide default numerical/generic methods

- RooAbsArg implementation must follow a minimal set of coding rules
Coding rules for RooAbsArg derived classes

1. Write well-behaved classes.
   - RooAbsArg objects classes are not glorified structs, well-defined copy semantics are essential: write a functional copy constructor

2. Every concrete class must have a clone() method

3. Do not store pointers to other RooAbsArg objects
   - Many high-level RooFit operations, such as plotting, fitting and generating, clone composite PDFs and need to readjust links
   - Use RooXXXProxy classes to store references

Original 1-Components cloned 2-Links adjusted

Composite Cloning Process (2 steps)
Class hierarchy

Introduction of various abstract base classes
Coding examples
Hierarchy of classes representing a value or function

- **RooAbsArg**
  - Abstract value holder

- **RooAbsReal**
  - Abstract real-valued objects

- **RooAbsRealLValue**
  - Abstract real-valued object that can be assigned to

- **RooRealVar**
- **RooLinearVar**

- **RooConvolutetedPdf**
  - Abstract convoluted physics model

- **RooDecay**
- **RooBMixDecay**

- **RooAbsPdf**
  - Abstract probability density function

- **RooGaussian**
- **RooArgusBG**
- **RooAddPdf**
- **RooProdPdf**
- **RooExtendPdf**

- **RooAbsGoodnessOfFit**
  - Abstract goodness of fit from a dataset and a PDF

- **RooNLLVar**
- **RooChiSquareVar**

- **RooResolutionModel**
  - Abstract resolution model

- **RooDecay**
- **RooBMixDecay**

Wouter Verkerke, UCSB
Class **RooAbsArg**

**RooAbsArg**

Abstract value holder

Implementations can represent any type of data.

Top-level class for objects representing a value

The main role of RooAbsArg is to manage client-server links between RooAbsArg instances that are functionally related to each other.

**RooAbsRealLValue**

Abstract real-valued object that can be assigned to

**RooRealVar**

**RooLinearVar**

The main role of RooAbsArg is to manage client-server links between RooAbsArg instances that are functionally related to each other.

**RooConvolutedPdf**

Abstract convoluted physics model

**RooDecay**

**RooBMixDecay**

Wouter Verkerke, UCSB
Class **RooAbsReal**

Abstract base class for objects representing a real value

**RooAbsReal**
Abstract real-valued objects

Class **RooAbsReal** implements 
<lazy evaluation>: 
`getVal()` only calls `evaluate()` 
if any of the server objects changed value

Implementations may advertise analytical integrals

Wouter Verkerke, UCSB
Class **RooAbsRealLValue**

Abstract base class for objects representing a real value that can be assigned to (C++ 'lvalue')

- **RooAbsRealLValue**
  - Abstract real-valued object that can be assigned to
  - **RooRealVar**
  - **RooLinearVar**

An **lvalue** is an object that can appear on the left hand side of the = operator

Few implementations as few functions are generally invertible:
- **RooRealVar**: parameter object
- **RooLinearVar**: linear transformation

Wouter Verkerke, UCSB
Class **RooAbsPdf**

Abstract base class for probability density functions

Defining property

\[ \int f(\vec{x}, \vec{p})\,d\vec{x} \equiv 1 \]

Where \( \vec{x} \) are the observables and \( \vec{p} \) are the parameters
Class **RooConvolutedPdf**

Abstract convoluted physics model

Implements \( f_i(dt,...) \otimes R(dt,...) \)

**RooResolutionModel**

\[
P(dt,...) = \sum_k c_k(...)(f_k(dt,...) \otimes R(dt,...))
\]

**RooConvolutedPdf** (physics model)

Implements \( c_k \), declares list of \( f_k \) needed

*No convolutions calculated in this class!*

Abstract base class for PDFs that can be convoluted with a resolution model
Class **RooResolutionModel**

Implementations of **RooResolutionModel** are regular PDFs with the added capability to calculate their function convolved with a series of ‘basis’ functions.

Resolution model advertises which basis functions it can handle.

To be used with a given **RooConvolutedPdf** implementation, a resolution model must support *all* basis functions used by the RooConvolutedPdf.

---

Wouter Verkerke, UCSB
Class *RooAbsGoodnessOfFit*

Provides the framework for efficient calculation of goodness-of-fit quantities.

A goodness-of-fit quantity is a function that is calculated from

- A dataset
- the PDF value for each point in that dataset

Built-in support for

- **Automatic constant-term optimization**
  activated when used by RooMinimizer(MINUIT)
- **Parallel execution on multi-CPU hosts**
- Efficient **calculation of RooSimultaneous** PDFs
Class tree for discrete-valued objects

- **RooAbsArg**
  - Generic value holder

- **RooAbsCategory**
  - Generic discrete-valued objects

  - **RooAbsCategoryLValue**
    - Generic discrete-valued object that can be assigned to

  - **RooCategory**

  - **RooSuperCategory**

- **RooMappedCategory**

- **RooThresholdCategory**

- **RooMultiCategory**
RooFit users tutorial

Code examples

Implementing a RooAbsReal
Providing analytical integrals
Implementing a RooAbsPdf
Providing an internal generator
Implementing a RooConvolutedPdf/RooResolutionModel
Implementing a RooAbsGoodnessOfFit

Wouter Verkerke, UCSB
Writing a real-valued function – class `RooAbsReal`

- Class declaration

```cpp
class RooUserFunc : public RooAbsReal {
public:
    RooUserFunc(const char *name, const char *title,
                 RooAbsReal& _x, RooAbsReal& _mean,
                 RooAbsReal& _sigma);
    virtual TObject* clone(const char* newname) const {
        return new RooUserFunc(*this,newname);
    }
    inline virtual ~RooUserFunc() { }
protected:
    RooRealProxy x ;
    RooRealProxy mean ;
    RooRealProxy sigma ;

    Double_t evaluate() const ;
private:
    ClassDef(RooUserFunc,0) // Gaussian PDF
};
```

Real-valued functions inherit from `RooAbsReal`
Writing a function – class **RooAbsReal**

- **Mandatory methods**

  - **Constructor**
    ```cpp
class RooUserFunc : public RooAbsPdf {
  public:
    RooUserFunc(const char *name, const char *title,
                 RooAbsReal& _x, RooAbsReal& _mean,
                 RooAbsReal& _sigma);
  
  public:
    RooUserFunc(const RooUserFunc& other,
                 const char* name=0) ;
    virtual TObject* clone(const char* newname) const {
      return new RooUserFunc(*this,newname);  
    }
  } inline virtual ~RooUserFunc() { }
protected:
  RooRealProxy x ;
  RooRealProxy mean ;
  RooRealProxy sigma ;

  Double_t evaluate() const ;

private:
  ClassDef(RooUserFunc,0) // Gaussian PDF
};
```

  - **Copy constructor**

  - **Clone**

  - **Destructor**

  - **evaluate**
    *Calculates your PDF return value*
Writing a function – class **RooAbsReal**

- Constructor arguments

```cpp
class RooUserFunc : public RooAbsPdf {
public:
   RooUserFunc(const char *name, const char *title,
               RooAbsReal& _x, RooAbsReal& _mean,
               RooAbsReal& _sigma);
   RooUserFunc(const RooUserFunc& other,
               const char* name=0) ;
   virtual TObject* clone(const char* newname) const {
      return new RooUserFunc(*this,newname);
   }
   inline virtual ~RooUserFunc() { }
protected:
   RooRealProxy x ;
   RooRealProxy mean ;
   RooRealProxy sigma ;
   Double_t evaluate() const ;
private:
   ClassDef(RooUserFunc,0) // Gaussian PDF
};
```

Try to be as generic as possible, i.e.

Use **RooAbsReal**& to receive real-valued arguments
Use **RooAbsCategory**& to receive discrete-valued arguments

Allows user to plug in either
a variable (**RooRealVar**) or a function (**RooAbsReal**)
Writing a function – class RooAbsReal

• Storing RooAbsArg references

Always use proxies to store RooAbsArg references:

- **RooRealProxy** for RooAbsReal
- **RooCategoryProxy** for RooAbsCategory
- **RooSetProxy** for a set of RooAbsArgs
- **RooListProxy** for a list of RooAbsArgs

Storing references in proxies allows RooFit to adjust pointers

This is essential for cloning of composite objects
Writing a function – class **RooAbsReal**

- ROOT-CINT dictionary methods

```cpp
class RooUserFunc : public RooAbsPdf {
public:
    RooUserFunc(const char *name, const char *title,
                 RooAbsReal& _x, RooAbsReal& _mean,
                 RooAbsReal& _sigma);
    RooUserFunc(const RooUserFunc& other,
                 const char* name=0);
    virtual TObject* clone(const char* newname) const {
        return new RooUserFunc(*this,newname);
    }
    inline virtual ~RooUserFunc() { }
protected:
    RooRealProxy x;
    RooRealProxy mean;
    RooRealProxy sigma;
    Double_t evaluate() const;
private:
    ClassDef(RooUserFunc,1) // Gaussian PDF
};
```

Don’t forget ROOT **ClassDef** macro
No semi-colon at end of line!

Description here will be used in auto-generated **THtml** documentation
Writing a function – class \texttt{RooAbsReal}

- Constructor implementation

```cpp
RooUserFunc::RooUserFunc(const char *name, const char *title,
RooAbsReal& _x, RooAbsReal& _mean,
RooAbsReal& _sigma) :
RooAbsPdf(name,title),
x("x","Dependent",this,_x),
mean("mean","Mean",this,_mean),
sigma("sigma","Width",this,_sigma)
{
}
```

- Constructor implementation

```cpp
RooUserFunc::RooUserFunc(const RooUserFunc& other,
const char* name) :
RooAbsPdf(other,name),
x("x",this,other.x),
mean("mean",this,other.mean),
sigma("sigma",this,other.sigma)
{
}
```

- \texttt{evaluate} function implementation

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

**Initialize the proxies from the \texttt{RooAbsArg} method arguments**

**Pointer to owning object is needed to register proxy**

**Name and title are for description only**
Writing a function – class \texttt{RooAbsReal}

- Implement a copy constructor!

\begin{verbatim}
RooUserFunc::RooUserFunc(const char *name, const char *title,
    RooAbsReal& _x, RooAbsReal& _mean,
    RooAbsReal& _sigma) :
    RooAbsPdf(name,title),
    x("x","Dependent",this,_x),
    mean("mean","Mean",this,_mean),
    sigma("sigma","Width",this,_sigma)
{
}

RooUserFunc::RooUserFunc(const RooUserFunc& other,
    const char* name) :
    RooAbsPdf(other.name),
    x(this,other.x),
    mean(this,other.mean),
    sigma(this,other.sigma)
{
}

Double_t RooUserFunc::evaluate() const
{
    Double_t arg= x - mean;
    return exp(-0.5*arg*arg/(sigma*sigma)) ;
}
\end{verbatim}

In the class copy constructor, call all proxy copy constructors

Pointer to owning object is (again) needed to register proxy
Writing a function – class **RooAbsReal**

- Write evaluate function

```cpp
RooUserFunc::RooUserFunc(const char *name, const char *title,
    RooAbsReal& _x, RooAbsReal& _mean,
    RooAbsReal& _sigma) :
    RooAbsPdf(name,title),
    x("x","Dependent",this,_x),
    mean("mean","Mean",this,_mean),
    sigma("sigma","Width",this,_sigma)
{
}

RooUserFunc::RooUserFunc(const RooUserFunc& other,
    const char* name) :
    RooAbsPdf(other,name),
    x("x",this,other.x),
    mean("mean",this,other.mean),
    sigma("sigma","Width",this,other.sigma)
{
}
```

In `evaluate()`, calculate and return the function value

```cpp
Double_t RooUserFunc::evaluate() const
{
    Double_t arg= x - mean;
    return exp(-0.5*arg*arg/(sigma*sigma)) ;
}
```
Working with proxies

- **RooRealProxy/RooCategoryProxy** objects automatically cast to the value type they represent
  - Use as if they were fundamental data types
    
    ```cpp
    RooRealProxy x;
    Double_t func = x*x;
    ```
    
    Use as `Double_t`
    
    ```cpp
    RooCategoryProxy c;
    if (c=="bogus") {...}
    ```
    
    Use as `const char*`

- To access the proxied **RooAbsReal/RooAbsCategory** object use the `arg()` method
  
  ```cpp
  RooRealProxy x;
  RooCategoryProxy c;
  RooAbsReal& xarg = x.arg();
  RooAbsCategory& carg = c.arg();
  ```

- Set and list proxy operation completely transparent
  - Use as if they were **RooArgSet/RooArgList** objects
    
    NB: the value or `arg()` may change during the lifetime of the object (e.g. if a composite cloning operation was performed)
Lazy function evaluation & caching

- Method `getVal()` does not always call `evaluate()`
  - Each RooAbsReal object caches its last calculated function value
  - If none of the dependent values changed, no need to recalculate

- Proxies are used to track changes in objects
  - Whenever a RooAbsArg changes value, it notifies all its client objects that recalculation is needed
  - Messages passed via client/server links that are installed by proxies
  - Only if recalculate flag is set `getVal()` will call `evaluate()`

- Redundant calculations are automatically avoided
  - Efficient optimization technique for expensive objects like integrals
  - No need to hand-code similar optimization in function code: `evaluate()` is only called when necessary
Writing a function – analytical integrals

- **Analytical integrals are optional!**
- Implementation of analytical integrals is separated in two steps
  - Advertisement of available (partial) integrals:
  - Implementation of partial integrals
- Advertising integrals:
  - `getAnalyticalIntegral()`

```c++
int RooUserFunc::getAnalyticalIntegral(
    RooArgSet& allVars, RooArgSet& analVars) const {
  if (matchArgs(allVars, analVars, x)) return 1;
  return 0;
}
```

Integration is requested over all variables in set `allVars`

Task of `getAnalyticalIntegral()`:
1) find the *largest subset* that function can integrate analytically
2) Copy largest subset into `analVars`
3) Return unique identification code for this integral
Writing a function – advertising integrals

Task of `getAnalyticalIntegral()`:
1) find the *largest subset* that function can integrate analytically
2) Copy largest subset into analVars
3) Return unique identification code for this integral

```cpp
Int_t RooUserFunc::getAnalyticalIntegral(
    RooArgSet& allVars, RooArgSet& analVars) const
{
    if (matchArgs(allVars,analVars,x)) return 1 ;
    return 0 ;
}
```

Utility method `matchArgs()` does all the work for you:

If `allVars` contains the variable held in proxy `x` variable is copied to `analVars` and `matchArgs()` returns `kTRUE` If not, it returns `kFALSE`
Writing a function – advertising multiple integrals

```cpp
Int_t RooUserFunc::getAnalyticalIntegral(
    RooArgSet& allVars, RooArgSet& analVars) const
{
    if (matchArgs(allVars, analVars, x, m)) return 3;
    if (matchArgs(allVars, analVars, m)) return 2;
    if (matchArgs(allVars, analVars, x)) return 1;
    return 0;
}
```

If multiple integrals are advertised, test for the largest one first.

You may advertise analytical integrals for both *real-valued* and *discrete-valued* integrands.
Writing a function – implementing integrals

- Implementing integrals: `analyticalIntegral()`
  - One entry point for all advertised integrals

```c++
Double_t RooGaussian::analyticalIntegral(Int_t code) const {
  static const Double_t root2 = sqrt(2);
  static const Double_t rootPiBy2 = sqrt(atan2(0.0,-1.0)/2.0);
  Double_t xscale = root2*sigma;
  return rootPiBy2*sigma * (erf((x.max()-mean)/xscale) - erf((x.min()-mean)/xscale));
}
```

Integral identification code assigned by `getAnalyticalIntegral()`

Integration limits for real-valued integrands can be accessed via the `min()` and `max()` method of each proxy

Discrete-valued integrands are always summed over all states
Calculating integrals – behind the scenes

• Integrals are calculated by **RooRealIntegral**
  - To create an **RooRealIntegral** for a **RooAbsReal**

```cpp
RooAbsReal* f; // f(x)
RooAbsReal* int_f = f.createIntegral(x) ;

RooAbsReal* g ; // g(x,y)
RooAbsReal* inty_g = g.createIntegral(y) ;
RooAbsReal* intxy_g = g.createIntegral(RooArgSet(x,y)) ;
```

• Tasks of **RooRealIntegral**
  - Structural analysis of composite
  - Negotiate analytical integration with components PDF
  - Provide numerical integration where needed

• **RooRealIntegral** works universally on simple and composite objects

A **RooRealIntegral** is also a **RooAbsReal**

**RooRealIntegral** is RooFits most complex class!
Why advertised analytical integrals are sometimes not used

- Integration variable is not a fundamental
  - Suppose $f(x,y)$ advertises analytical integration over $x$

$$f(x,a), g(b,c) \rightarrow f(g(b,c),a)$$

(Exception: $g(x,y)$ is an invertable function (RooAbsRealLValue) with a constant Jacobian term)

Wouter Verkerke, UCSB
Why advertised analytical integrals are sometimes not used

• Function depends more than once on integration variable
  – Suppose $f(x,y)$ advertises analytical integration over $x$

$$f(x, y) \rightarrow f(a, a)$$

Composition introduces unforeseen correlation between variables
2D function is now 1D function

Must calculate integral numerically in terms of $a$
Why advertised analytical integrals are sometimes not used

- Function depends more on integration variable via more than one route
  - Suppose \( f(x,y) \) advertises analytical integration over \( x \)

\[
f(x, y), g(a, x) \rightarrow f(x, g(a, x))
\]

Value held in proxy \( x \) is not a variable but a function.

Must calculate integral numerically in terms of \( a,b \)
Class documentation

• General description of the class functionality should be provided at the beginning of your .cc file

```
// -- CLASS DESCRIPTION [PDF] --
// Your description goes here
```

Magic line for THtml

PDF Keyword causes class to be classified as PDF class

• First comment block in each function will be picked up by THtml as the description of that member function
  – Put some general, sensible description here
Writing a PDF – class **RooAbsPdf**

- Class declaration

```cpp
class RooUserPdf : public RooAbsPdf {
public:
    RooUserFunc(const char *name, const char *title,
    RooAbsReal& _x, RooAbsReal& _mean,
    RooAbsReal& _sigma);
    RooUserPdf(const RooGaussian& other,
    const char* name=0);
    virtual TObject* clone(const char* newname) const {
        return new RooUserPdf(*this,newname);
    }
    inline virtual ~RooUserPdf() { }
protected:
    RooRealProxy x ;
    RooRealProxy mean ;
    RooRealProxy sigma ;
    Double_t evaluate() const ;
private:
    ClassDef(RooUserPdf,0) // Gaussian PDF
};
```

PDFs inherit from **RooAbsPdf**

This is the *only* difference with a **RooAbsReal**

**RooAbsPdf::getVal()** will automatically normalize your return value by dividing it by the integral of the PDF. No further action is needed!

**RooRealIntegral** used for integral calculation

**RooAbsPdf** owns RRI configured for last normalization configuration. If normalization set Changes, new RRI as created on the fly..
Writing a PDF – Normalization

- Do not under any circumstances attempt to normalize your PDF in `evaluate()` via explicit or implicit integration

- You do not know over what variables to normalize
  - In RooFit, parameter/observable distinction is dynamic, a PDF does not have a unique normalization/return value

- You don’t even now know how to integrate yourself!
  - Your PDF may be part of a larger composite structure. Variables may be functions, your internal representation may have a difference number of dimensions that the actual composite object.
  - `RooRealIntegral` takes proper care of all this

- But you can help!
  - Advertise all partial integrals that you can calculate
  - They will be used in the normalization when appropriate
    - Function calling overhead is minimal
PDF Event generation – Accept/reject method

- By default, toy MC generation from a PDF is performed with accept/reject sampling
  - Determine maximum PDF value by repeated random sample
  - Throw a uniform random value \(x\) for the observable to be generated
  - Throw another uniform random number between 0 and \(f_{\text{max}}\)
  If \(\text{ran} \times f_{\text{max}} < f(x)\) accept \(x\) as generated event
PDF Event generation – Accept/reject method

- Accept/reject method can be very inefficient
  - Generating efficiency is
    \[ \frac{\int_{x_{\min}}^{x_{\max}} f(x) dx}{(x_{\max} - x_{\min}) \cdot f_{\max}} \]
  - Efficiency is very low for narrowly peaked functions
  - Initial sampling for f_{\max} requires very large trials sets in multiple dimension (\sim 10000000 in 3D)
PDF Event generation – Optimizations

- **RooFit ‘operator’ PDFs** provide various optimizations

- **RooProdPdf** – Components PDFs generated separately
  - Breaks down N dimensional problem to n m-dimensional problems
  - Large initial $f_{\text{max}}$ sampling penalty not incurred

- **RooAddPdf** – Only one component generated at a time
  - RooAddPdf randomly picks a component PDF to generate for each event. Component probabilities weighted according to fractions
  - Helps to avoid accept/reject sampling on narrowly peaked distributions, if narrow and wide component are separately generated

- **RooSimultaneous** - Only one component generated at a time
  - Technique similar to **RooAddPdf**
PDF Event generation – Internal generators

• For certain PDFs alternate event generation techniques exist that are more efficient than accept/reject sampling
  – Example: Gaussian, exponential,...

• If your PDF has such a technique, you can advertise it
  – Interface similar to analytical integral methods
    RooAbsPdf::getGenerator()
    RooAbsPdf::initGenerator()
    RooAbsPdf::generateEvent()

• You don’t have to be able to generate all observables
  – Generator context can combine accept/reject and internal methods within a single PDF

• This is an optional optimization
  – PDF can always generate events via accept/reject method
Writing a PDF – advertising an internal generator

Task of \textit{getGenerator}():
1) find the \textit{largest subset} of observables PDF can generate internally
2) Copy largest subset into dirVars
3) Return unique identification code for this integral

\begin{verbatim}
Int_t RooUserFunc::getGenerator(
    RooArgSet& allVars, RooArgSet& dirVars, Bool_t staticOK) const
{
    if (matchArgs(allVars,dirVars,x)) return 1 ;
    return 0 ;
}
\end{verbatim}

Utility method \texttt{matchArgs()} does all the work for you:

If \texttt{allVars} contains the variable held in proxy \texttt{x}
variable is copied to \texttt{dirVars} and \texttt{matchArgs()} returns \texttt{kTRUE}
If not, it returns \texttt{kFALSE}
For certain internal generator implementations it can be efficient to do a one-time initialization for each set of generated events. Example: precalculate fractions for discrete variables.

**Caveat:** one-time initialization only safe if no observables are generated from a prototype dataset. Only advertise such techniques if staticOK flag is true.

Writing a PDF – advertising an internal generator

```cpp
Int_t RooBMixDecay::getGenerator(const RooArgSet& directVars,
                                   RooArgSet &generateVars, Bool_t staticInitOK) const
{
    if (staticInitOK) {
        if (matchArgs(directVars,generateVars,t,mix,tag)) return 4 ;
        if (matchArgs(directVars,generateVars,t,mix)) return 3 ;
        if (matchArgs(directVars,generateVars,t,tag)) return 2 ;
    }
    if (matchArgs(directVars,generateVars,_t)) return 1 ;
    return 0 ;
}
```

If you advertise multiple configurations, try the most extensive one first.
Writing a PDF – implementing an internal generator

- Implementing a generator: \texttt{generateEvent()} 
  - One entry point for all advertised event generators

```c
void RooGaussian::generateEvent(Int_t code) {
    Double_t xgen;
    while(1) {
        xgen = RooRandom::randomGenerator()->Gaus(mean,sigma);
        if (xgen<x.max() && xgen>x.min()) {
            x = xgen;
            break;
        }
    }
    return;
}
```

Generator identification code assigned by \texttt{getGenerator()}

Return generated value by assigning it to the proxy
Writing a PDF – implementing an internal generator

- **Static generator initialization: `initGenerator()`**
  - This function is guaranteed to be call once before each series of `generateEvent()` calls with the same configuration

```c++
void RooBMixDecay::initGenerator(Int_t code)
{
    switch (code) {
    case 2:
    {
        // Calculate the fraction of B0bar events to generate
        Double_t sumInt = RooRealIntegral(...).getVal() ;
        _tagFlav = 1 ; // B0
        Double_t flavInt = RooRealIntegral(...).getVal() ;
        _genFlavFrac = flavInt/sumInt ;
        break ;
    }
    }
}
```

Generator identification code assigned by `getGenerator()`

Store your precalculated values in data members
Writing a convoluted PDF – physics/resolution factorization

- Physics model and resolution model are implemented separately in RooFit
  - Factorization achieved via a common set ‘basis functions’ $f_k$

\[ P(dt,...) = \sum_{k} c_k (...) (f_k (dt,...) \otimes R(dt,...)) \]

- **RooResolutionModel** (physics model)
  - Implements $f_i (dt,...) \otimes R(dt,...)$
  - Also a PDF by itself
  - Declares list of $f_k$ needed

**No magic**: You must still calculate the convolution integral yourself, but factorization enhances modularity & flexibility for end user
Writing a convoluted PDF – class RooConvolutedPdf

- Class declaration

```cpp
class RooBMixDecay : public RooConvolutedPdf {
    public:
    RooBMixDecay(const RooBMixDecay& other, const char* name=0);
    virtual TObject* clone(const char* newname) const;
    virtual ~RooBMixDecay();
    virtual Double_t coefficient(Int_t basisIndex) const;

    protected:
    
};
```

Convolutable PDF classes inherit from **RooConvolutedPdf** instead of **RooAbsPdf**

Implement `coefficient()` instead of `evaluate()`
Class `RooConvolutedPdf` – Constructor implementation

- Constructor must declare all basis functions used

```cpp
RooBMixDecay::RooBMixDecay(const char *name, const char *title,…) :
    RooConvolutedPdf(name,title,model,t), …
{
    // Constructor
    _basisExp = declareBasis("exp(-abs(@0)/@1)",
                              RooArgList(tau)) ,
    _basisCos = declareBasis("exp(-abs(@0)/@1)*cos(@0*@2)",
                              RooArgList(tau,dm)) ;
}
```

- Call `declareBasis()` for each basis function used in this PDF
- Name of basis function is `RooFormulaVar` expression
- @0 is convolution variable, @1..@n are basis function parameters
- Supply basis function parameters here
- Return code assign unique integer code to each declared basis
- Wouter Verkerke, UCSB
Class **RooConvolutedPdf** – Coefficient implementation

- Method `coefficient()` implements all coefficient values

```c++
Double_t RooBMixDecay::coefficient(Int_t basisIndex) const
{
    if (basisIndex==_basisExp) {
        return (1 - _tagFlav*_delMistag) ;
    }

    if (basisIndex==_basisCos) {
        return _mixState*(1-2*_mistag) ;
    }

    return 0 ;
}
```

Requested index is one of the basis function codes returned by `declareBasis()`.

- At this point class is **complete** and **functional**
Class **RooConvolutedPdf** – Analytical integrals

- **You can optionally** advertise and implement **analytical integrals** for your **coefficient functions**
  - Interface similar to analytical integrals in RooAbsReal

- **Advertising coefficient integrals**
  - Method identical to `RooAbsReal::getAnalyticalIntegral()`, just the name is different

```
Int_t getCoefAnalyticalIntegral(RooArgSet& allVars,
                                RooArgSet& analVars) const ;
```

- **Implementing coefficient integrals**
  - Method similar to `RooAbsReal::analyticalIntegral()`
  - One extra argument to identify the coefficient in question

```
Double_t coefAnalyticalIntegral(Int_t coef,
                                 Int_t code) const ;
```
Class **RooConvolutedPdf** – Internal generator implementation

- You can optionally advertise and implement an internal generator for the *unconvoluted* PDF function
  - Methods identical to regular PDF generator implementation

- An internal generator will **greatly accelerate** toyMC generation from a convoluted PDF
  - If both physics PDF and resolution model provide internal generators, then events can be generated as
    \[ x_{P \otimes R} = x_P + x_R \]
    i.e. no convolutions integrals need to be evaluated
  - Only works with internal generator implementations because both \( x_P \) and \( x_R \) must be generated on an unbound domain for this technique to work
    - Accept reject sample doesn’t work on unbound domains
Writing a resolution model – physics/resolution factorization

- Physics model and resolution model are implemented separately in RooFit
  - Factorization achieved via a common set ‘basis functions’ \( f_k \)

\[
P(dt,...) = \sum_k c_k (...) (f_k (dt,...) \otimes R(dt,...))
\]

- Implements \( f_i (dt,...) \otimes R(dt,...) \)
- Also a PDF by itself
- RooResolutionModel

- Implements \( c_k \)
- Declares list of \( f_k \) needed
Writing a resolution model PDF – class **RooResolutionModel**

- Class declaration

```cpp
class RooGaussModel : public RooResolutionModel {
public:
  RooGaussModel(const RooBMixDecay& other, const char* name=0);
  virtual TObject* clone(const char* newname) const ;
  virtual ~RooGaussModel();

protected:
  ...
  ClassDef(RooGaussModel,1) // Gaussian resolution model
};
```

Resolution model classes inherit from **RooResolutionModel** instead of **RooAbsPdf**

- Method **basisCode()** advertises supported basis functions

```cpp
virtual Int_t basisCode(const char* name) const = 0 ;
virtual Double_t evaluate() const ;
```

**evaluate()** returns *regular or convoluted* PDF depending on internal state
Class \textbf{RooResolutionModel} – Advertising basis functions

- Function \texttt{basisCode()} assigns unique integer code to each supported basis function

```cpp
Int_t RooGaussModel::basisCode(const char* name) const
{
    if (!TString("exp(-@0/@1)").CompareTo(name)) return 1;
    if (!TString("exp(@0/@1)").CompareTo(name)) return 2;
    if (!TString("exp(-abs(@0)/@1)").CompareTo(name)) return 3;
    if (!TString("exp(-@0/@1)*sin(@0*@2)").CompareTo(name)) return 4;
    if (!TString("exp(@0/@1)*sin(@0*@2)").CompareTo(name)) return 5;
    if (!TString("exp(-abs(@0)/@1)*sin(@0*@2)").CompareTo(name)) return 6;
    if (!TString("exp(-@0/@1)*cos(@0*@2)").CompareTo(name)) return 7;
    if (!TString("exp(@0/@1)*cos(@0*@2)").CompareTo(name)) return 8;
    if (!TString("exp(-abs(@0)/@1)*cos(@0*@2)").CompareTo(name)) return 9;
    return 0;
}
```

Return 0 if basis function is not supported.
Class **RooResolutionModel** – Implementing **evaluate()**

- **evaluate()** returns both convoluted and unconvoluted PDF value

```cpp
Double_t RooGaussModel::evaluate() const
{
    Int_t code = currentBasisCode() ;

    if (code==0) {
        // return unconvoluted PDF value ;
    }

    if (code==1) {
        // Return PDF convoluted with basis function #1
        // Retrieve basis function paramater value
        Double_t tau = basis().getParameter(1))->getVal() ;
    } 
}
```

**currentBasisCode()** returns the ID of the basis function we’re convoluted with. If zero, not convoluted is requested.
Class **RooResolutionModel** – Implementing **evaluate()**

- **evaluate()** returns both convoluted and unconvoluted PDF value

```cpp
Double_t RooGaussModel::evaluate() const
{
    Int_t code = currentBasisCode() ;
    if (code==0) {
        // return unconvoluted PDF value
    }
    if (code==1) {
        // Return PDF convoluted with basis function #1
        // Retrieve basis function parameter value
        Double_t tau = basis().getParameter(1))->getVal() ;
    }
}
```

- **basis()** returns a reference to the **RooFormulaVar** representing the current basis function
- **getParameter(n)** returns a **RooAbsReal** reference to the **n**th parameter of the function
Class **RooResolutionModel** – Analytical integrals

- Advertising and implementing **analytical integrals** works the **same way as in RooAbsPdf**

```cpp
Int_t RooGaussModel::getAnalyticalIntegral(RooArgSet& allVars, RooArgSet& analVars) const
{
    switch(currentBasisCode()) {
        // Analytical integration capability of raw PDF
        case 0:
            if (matchArgs(allVars, analVars, convVar())) return 1 ;
            break ;
        // Analytical integration capability of convoluted PDF
        case 1:
            if (matchArgs(allVars, analVars, convVar())) return 1 ;
            break ;
    }
}
```

Advertisement and implementation should reflect the ‘current’ convolution indicated by **currentBasisCode()**
Class **RooResolutionModel** – Internal generator implementation

- You can optionally advertise and implement an internal generator for the *unconvoluted* resolution model
  - Methods identical to regular PDF generator implementation
Class **RooAbsGoodnessOfFit** – Goodness of fit

- No time left to write this section... (sorry!)
Debugging

ROOT and gdb/dbx
Finding memory leaks
Tracing function evaluation
Checking integrals & generators
Profiling
Using the system debugger

- Compiled applications linked with RooFit
  - Just use `gdb/dbx <executable>`

- Interactive ROOT
  - You can use gdb/dbx to debug your compiled RooFit class
  - Trick: attach debugger to already running ROOT process
    1. Start interactive ROOT the usual way
    2. In a separate shell on the same host
       attach gdb/dbx to the running ROOT session
    3. Resume running of ROOT
       `gdb> continue`
    4. Execute the code you want to test

```bash
#!/bin/sh
line=`ps -wwfu $USER | grep root.exe | grep -v grep | tail -1`
if [ "$line" = "" ]; then
  echo "No ROOT session running"
  exit 1
fi
set $line
exec gdb $8 $2
```
Finding memory leaks

- **RooTrace** utility keeps track of RooFit object allocation

```cpp
RooTrace::active(kTRUE)

RooRealVar x("x","x",-10,10) ;
RooGaussian g("g","g",x,RooConst(0),RooConst(1)) ;

RooTrace::dump(cout);
List of RooFit objects allocated while trace active:
00086b7118 : RooRealVar - x
00086aa178 : RooArgList - RooRealVar Constants Database
00086b7658 : RooConstVar - 0.000000
00086b7b08 : RooConstVar - 1.000000
00086bc3e8 : RooGaussian - g
```
Finding memory leaks

- You can do incremental leak searches

```cpp
RooTrace::active(kTRUE)

RooRealVar x("x","x",-10,10) ;
RooGaussian g("g","g",x,RooConst(0),RooConst(1)) ;

RooTrace::mark() ; // mark all objects created sofar

RooGaussian g2("g2","g2",x,RooConst(2),RooConst(1)) ;

// Dump only objects created since last mark
RooTrace::dump(cout,kTRUE);
List of RooFit objects allocated while trace active:
00086c8f50 : RooConstVar - 2.000000
00086c9400 :          RooGaussian - g2
5 marked objects suppressed
```
Tracing function evaluation

- When you have many instances of a single class it can be more useful to trace function evaluation with printed messages than via debugger
  - Debugger breakpoint will stop in every instance of your class even if you only want to examine a single instance

- RooFit provides system-wide tracing techniques
  - `RooAbsArg::setVerboseDirty(kTRUE)`
    - Track lazy evaluation logic of RooAbsArg classes
    - May help to understand why your evaluate() doesn’t get called
  - `RooAbsArg::setVerboseEval(Int_t level)`
    - Level 0 – No messages
    - Level 1 – Print one-line message each time a normalization integral is recalculated
    - Level 2 – Print one-line message each time a PDF is recalculated
    - Level 3 – Provide details of convolution integral recalculations
Tracing function evaluation

• And object-specific tracing techniques
  – pdf->setTraceCounter(Int_t n, Bool_t recursive)
  – Prints one-lines messages for the next n times pdf is evaluated
  – If recursive option is set, trace counter is also set for all component PDFs of pdf
  – Useful in fitting/likelihood calculations where is single likelihood evaluation can trigger thousands of PDF evaluations
Checking analytical integrals and internal generators

- Function integrals and PDF event generators both have a numerical backup solution
  - You can use those as a cross check to validate your function/PDF-specific implementation

- Integrals
  - `RooAbsReal::forceNumInt(kTRUE)` will disable the use of any advertised analytical integrals

- Generators
  - `RooAbsPdf::forceNumGen(kTRUE)` will disable the use of any advertised internal PDF generator methods
Profiling

• To run the profiler you must build your test application as a standalone executable
  
  compile & link with \-pg flag

```c
#include "TROOT.h"
#include "TApplication.h"

// Instantiate ROOT system
TROOT root("root", "root");
int main(int argc, char **argv)
{
    // Instantiate graphics event handler
    TApplication app("TAppTest", &argc, argv);

    // User code goes here
}
```

– You cannot have any RooFit classes as global variables
  
  Prior instantiation of TROOT needed, but cannot be guaranteed

– Place your driver executable in the RooFitModels directory and list it as a binary target in the GNUMakefile

Wouter Verkerke, UCSB
Outlook

- New goodness-of-fit calculation classes will be introduced soon (~1 week)
  - Likelihood and ChiSquare as examples.
  - Complete function optimization support for likelihood fitting now generically available for all goodness of fits
  - Built-in support for handling RooSimultaneous PDFs
  - Support for parallel execution on multi-CPU hosts
    - No support from user code needed except prescription to merge partial results (Default implementation adds partial results)