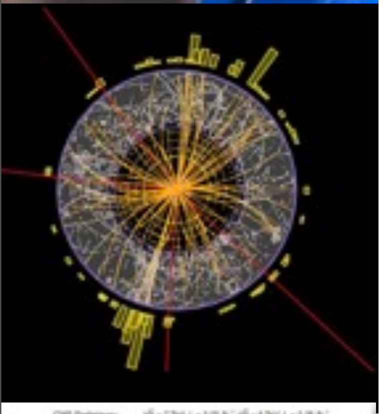
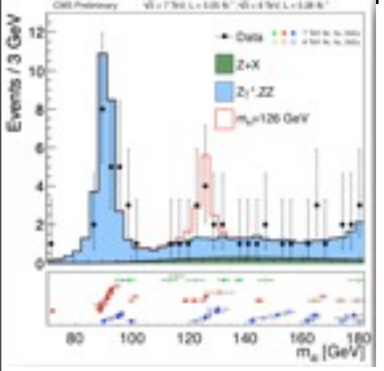
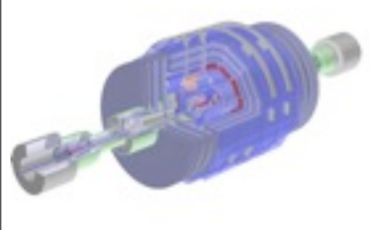






New ROOT TFormula class

Lorenzo Moneta (PH-SFT),
Maciej Zimnoch (University of Wroclaw, GSoC)
presented by
Fons Rademakers (PH-SFT)

October 15, 2013

Outline

- `TFormula` class in ROOT
  - Introduction
  - Current functionality and limitations
- New `TFormula` class
  - *Developed by Maciej Zimnoch (Google Summer of Code student 2013)*
  - Using Cling to evaluate expressions
- Current Status
  - Performance tests
- Future improvements
  - Auto-Differentiation
- Conclusions

Introduction

- **`TFormula`** class in ROOT
  - Class for evaluating mathematical functions provided as expression strings
  - ROOT function class (**`TF1`**) derives from **`TFormula`**
    - Uses **`TFormula`** constructs for making functions from string
  - **`TF1`** is used for fitting and for plotting functions in ROOT
- Examples of **`TFormula`** constructs:
  - Simple functions

```
TFormula(“f1”, “sin(x)” );

TFormula(“f2”, “x^2+2” );
```

  - Composition of functions

```
TFormula(“f3”, “f1 + f2” );
```

Formula Constructs

– Function with parameters:

```
TFormula(“f”, “[0] * sin( x * [1] )” );
```

– Using predefined functions:

```
TFormula(“f”, “gaus”);
```

- **gaus** is equivalent to: `[0]*TMath::Gaus(x,[1],[2])`

```
TFormula(“f”, “pol2”);
```

- **pol2** is equivalent to: `[0] + [1]* x + [2]*x^2`

– Using any library function:

```
TFormula(“f”, ”ROOT::Math::chisquared_pdf(x,[0])”);
```

Current Limitations

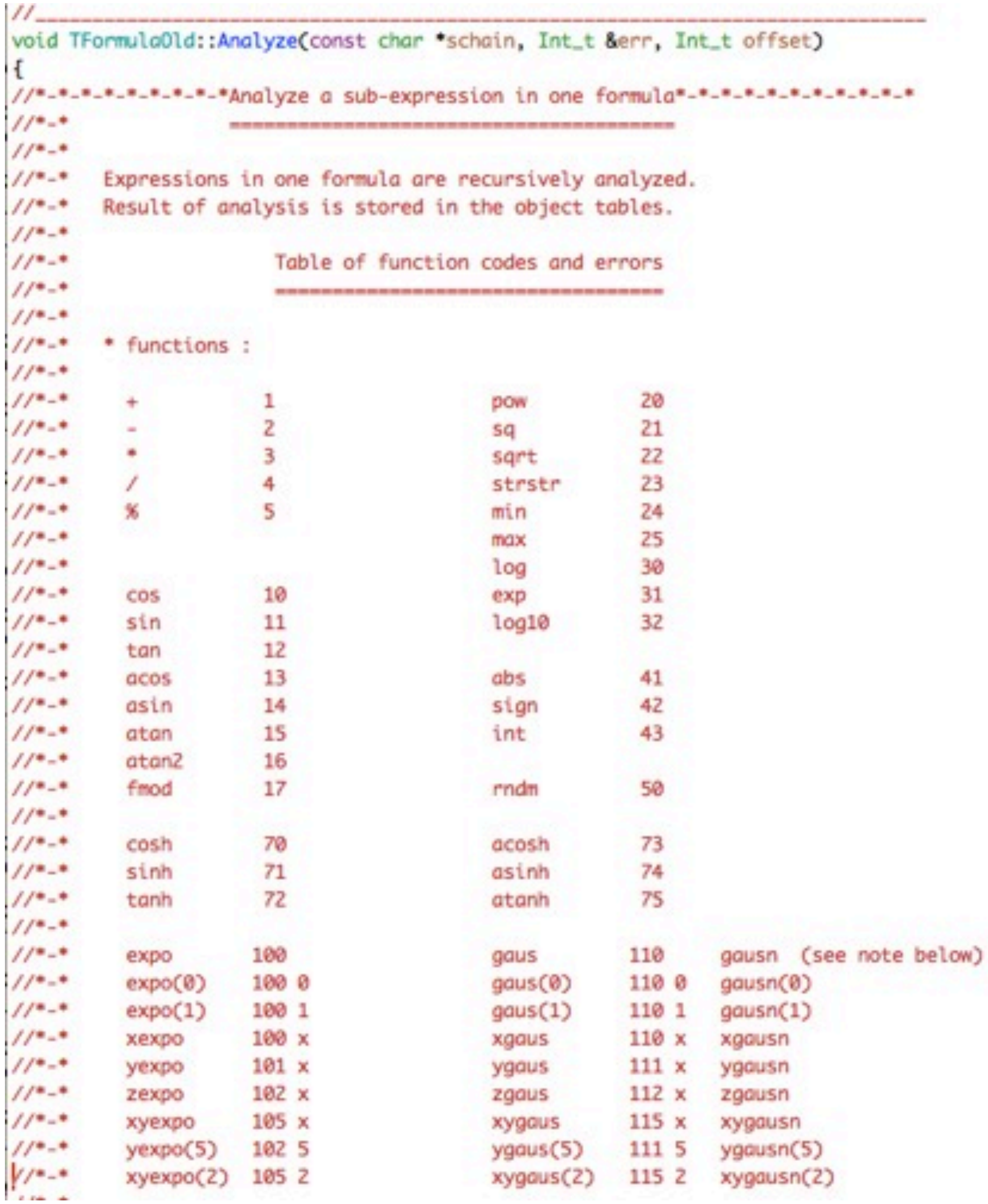
- **`TFormula`** contains customized code to parse the expression string and evaluate it
  - Custom parser (does not use CINT C/C++ parser)
    - CINT too slow to evaluate functions
  - Has been optimized for speed
    - Used for fitting
    - Used in TTreeFormula to query TTree's
  - Parsing expressions is complex
    - Several 1000's lines of code
  - Very difficult to extend and maintain
    - E.g. adding new C++11 syntax

Current Limitations (2)

- Dictionary (CINT and now Cling) is used for functions from the library
  - E.g. functions from TMath or ROOT::Math
  - Slow to execute since it function call is wrapped in interpreted code
  - **`TFormula`** defines some pre-defined functions:
    - **`gaus, polN, expo, landau`**
    - Used in the formula as compiled code
    - E.g. “**`gaus`**” is much faster than “**`[0] * TMath::Gaus(x,[1],[2] )`**”
  - Pre-defined functions add extra-complexity in the code
    - Difficult to add new ones

Current Limitations (3)

- Do we really need this customized parsing code in TFormula?



```
//______________________________________________________________________________
void TFormulaOld::Analyze(const char *schain, Int_t &err, Int_t offset)
{
//*-*-*-*-*-*-*-*-*-*Analyze a sub-expression in one formula*-*-*-*-*-*-*-*-*-*-*
//*-*                ========================================
//*-*
//*-*  Expressions in one formula are recursively analyzed.
//*-*  Result of analysis is stored in the object tables.
//*-*
//*-*                  Table of function codes and errors
//*-*                  ==================================
//*-*
//*-*  * functions :
//*-*
//*-*    +           1                   pow         20
//*-*    -           2                   sq          21
//*-*    *           3                   sqrt        22
//*-*    /           4                   strstr      23
//*-*    %           5                   min         24
//*-*                                    max         25
//*-*                                    log         30
//*-*    cos         10                  exp         31
//*-*    sin         11                  log10       32
//*-*    tan         12
//*-*    acos        13                  abs         41
//*-*    asin        14                  sign        42
//*-*    atan        15                  int         43
//*-*    atan2       16
//*-*    fmod        17                  rndm        50
//*-*
//*-*    cosh        70                  acosh       73
//*-*    sinh        71                  asinh       74
//*-*    tanh        72                  atanh       75
//*-*
//*-*    expo       100                  gaus       110    gausn  (see note below)
//*-*    expo(0)    100 0                gaus(0)    110 0  gausn(0)
//*-*    expo(1)    100 1                gaus(1)    110 1  gausn(1)
//*-*    xexpo      100 x                xgaus      110 x  xgausn
//*-*    yexpo      101 x                ygaus      111 x  ygausn
//*-*    zexpo      102 x                zgaus      112 x  zgausn
//*-*    xyexpo     105 x                xygaus     115 x  xygausn
//*-*    yexpo(5)   102 5                ygaus(5)   111 5  ygausn(5)
//*-*    xyexpo(2)  105 2                xygaus(2)  115 2  xygausn(2)
```

New TFormula

- Uses Cling available in ROOT 6
- Replace old parser with the JIT provided by Cling
  - A real C++ interpreter
  - More confident in correctness of results when using a real compiler
  - Better detection of syntax errors
  - Reduce substantially the code size
- Maintain the old functionality for backward compatibility
- Extensible
  - Easy to add new functions
  - Use different variables names than x,y,z
- Scale to large and complex expressions

How Does it Work

- **`TFormula`** creates a C/C++ functions which is passed to Cling

```
TFormula(“f”, “[0] * sin( x * [1] )” );
```

```
Double_t TF__f(Double_t *x,Double_t *p)
{
          return p[0]*TMath::Sin(x[0]*p[1]) ;
}
```

- The created function is now compiled on the fly using the JIT of Cling

Evaluation of TFormula

- Faster evaluation: it is compiled code!
  - No need to have a dedicated parser to analyze and compile the code as in old **`TFormula`**
- JIT compilation is done at initialization time, not when evaluating the expression
- The created function is evaluated using its function pointer, which can be retrieved via the ROOT interpreter interface
  - Very small overhead compared to calling the function via MethodCall interface
- Pre-defined functions (**`gaus`**) and library functions (**`TMath::Gaus`**) are treated in the same way

TFormula Parsing (Initizalization)

- **`TFormula`** parsing is now limited to clean up the input expression:
  - interpreting parameter names
    - `[0]` → `par[0]`
  - interpreting variables
    - `x,y,z` → `x[0], x[1], x[2]`
  - translating pre-defined expressions
    - `gaus(0)` → `par[0]*TMath::Gaus(x[0],par[1],par[2])`
- Check validity of expression
- Create the C/C++ function for Cling
- Code is reduced substantially

Advantages of New TFormula

- More flexible code
  - Easy to add new pre-defined functions as shortcuts for user convenience
    - Just one line of code to change to include the translation symbol corresponding to the pre-defined function
      - I.e. what “**`myfunc`**” is translated to in C++
  - One can add meaningful parameters directly in the expression
    - `TFormula f(“f”, “A * sin( x * B )” );`
    - `f.SetParameter(“A”,1); f.SetParameter(“B”,2);`
  - Function dimension is not limited to 4
    - One can define functions with several variables
    - Use `f.AddVariable(..)`

Performance Tests

- Tests of new **`TFormula`**:
  - Formula with 4 variables and 1000 parameters:
    - Initialization:
      - Time = 0.1 seconds
    - 1 million evaluations:
      - Time = 2.1 seconds
- Using Old **`TFormula`**:
  - Using the same expression (1000 parameters)
    - Initialization
      - Time = 1.2 seconds
    - Evaluation:
      - Fails to evaluate such large expressions
      - Does not scale for such large formula

Performance Test (2)

- Test of evaluation of *new vs old* **`TFormula`**
  - Time for 1 evaluation (in ns)

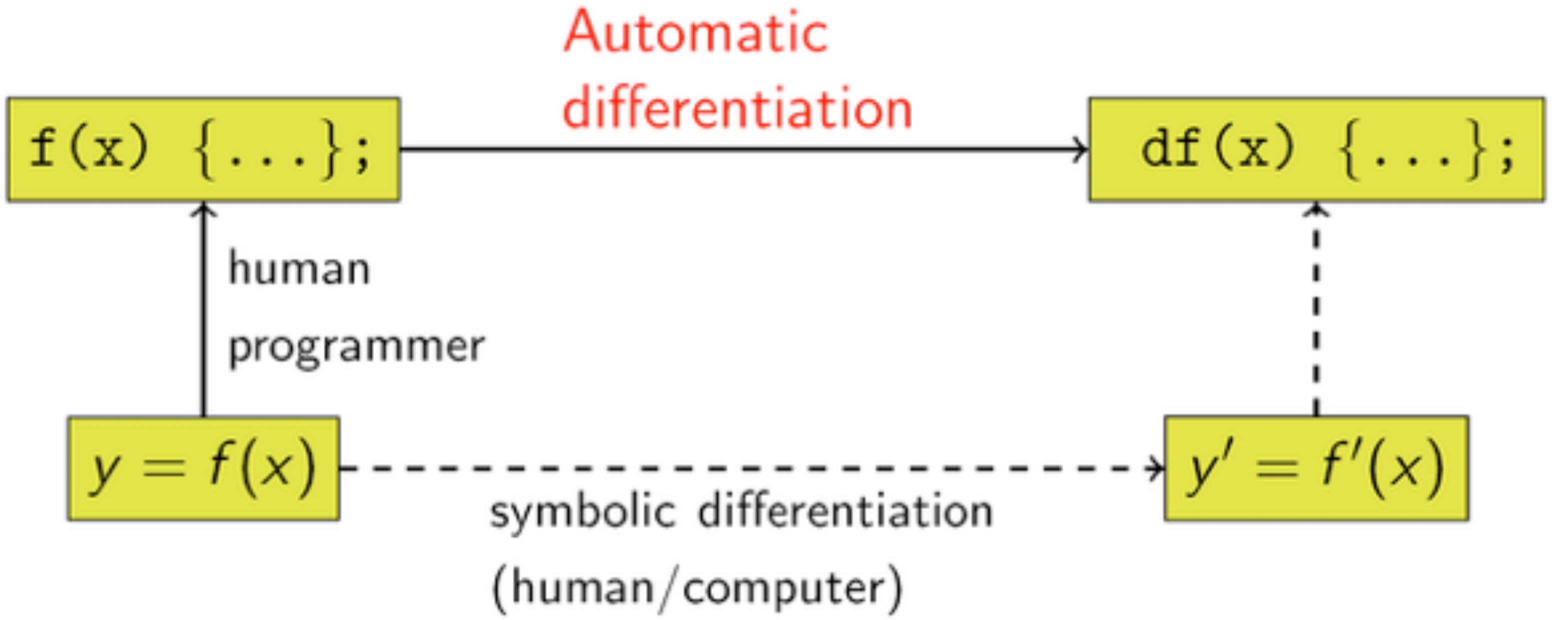
Expression type	New TFormula v5.99	Old TFormula v5.34
predefined functions **`gaus(0) + gaus(3)`**	60 ns	65 ns
interpreting expression **`"TMath::Gaus(....)"`**	65 ns	400 ns
formula functions **`"exp(-0.5*(x-[1])/[2])^2)..."`**	60 ns	200 ns
compiled functions **`double f(double*x,double*p) { return TMath::Gaus(...).. }`**	50 ns	50 ns

Current Status

- New **`TFormula`** class is available on github
  - https://github.com/lmoneta/root/tree/tformula
- It has already been integrated in **`TF1`** and it can be used for fitting and plotting functions
  - Several remaining issues are being fixed
- Will soon be integrated in the ROOT master
  - Still working on improving:
    - Adding more pre-defined functions
    - Better interface to add new variables and new parameter names in expression

Further Improvements

- Computing Derivatives inside **`TFormula`** using Auto-Differentiation (AD)



Automatic differentiation

`f(x) {...};`

`df(x) {...};`

human programmer

$y = f(x)$

$y' = f'(x)$

symbolic differentiation (human/computer)

  – AD allows to compute precisely and efficiently the function derivatives
    - Reduces numerical error and reduces computation cost compared to numerical derivatives
  – Extremely useful for fitting/minimization

Automatic Differentiation

- Auto-Differentiation requires to transform the semantic of a program (function)
  - Works by combining values of basic operations using the derivative chain rule

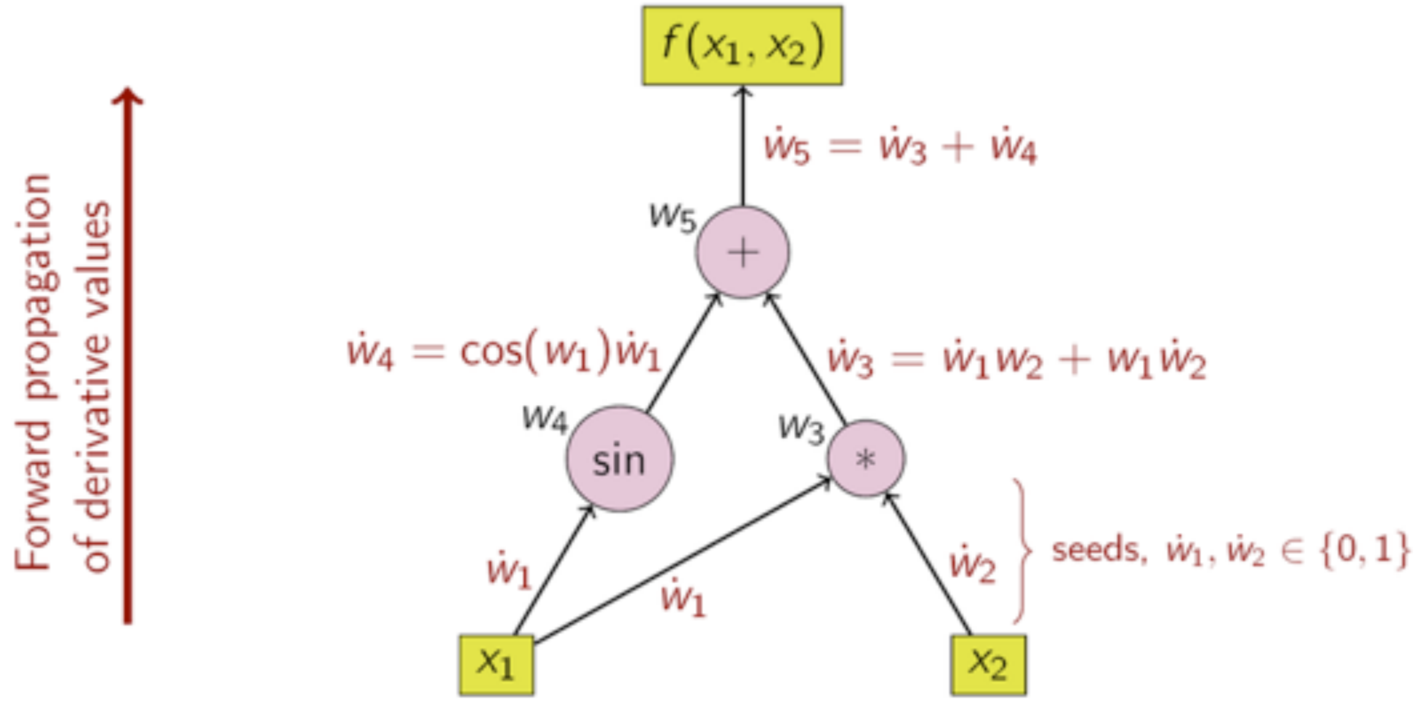
$$f(x) = g(h(x))$$

$$f'(x) = g'(h(x)) \cdot h'(x)$$

$$\frac{\partial f}{\partial x} = \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial x}$$

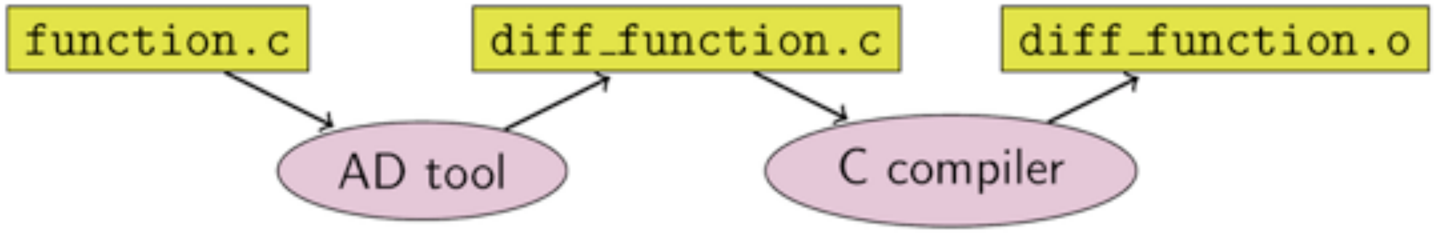
$$f(x_1, x_2) = x_1 x_2 + \sin(x_1)$$

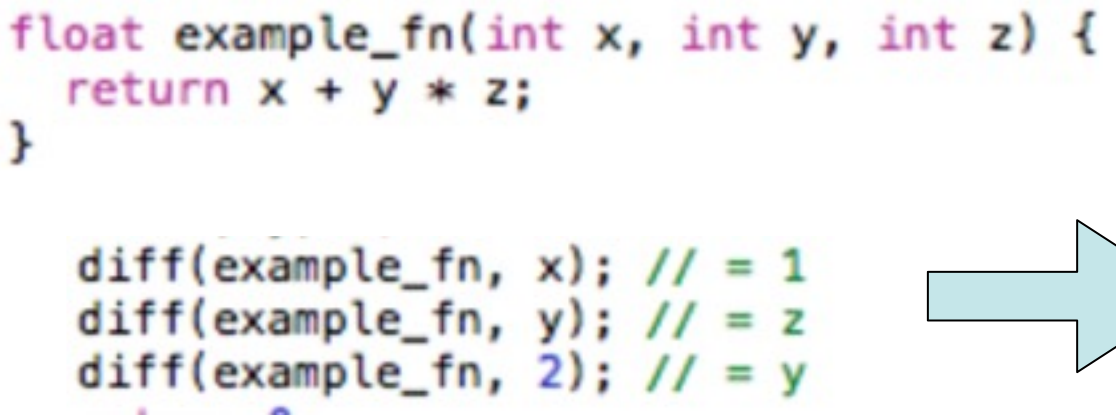
Forward propagation



AD Prototype in Cling

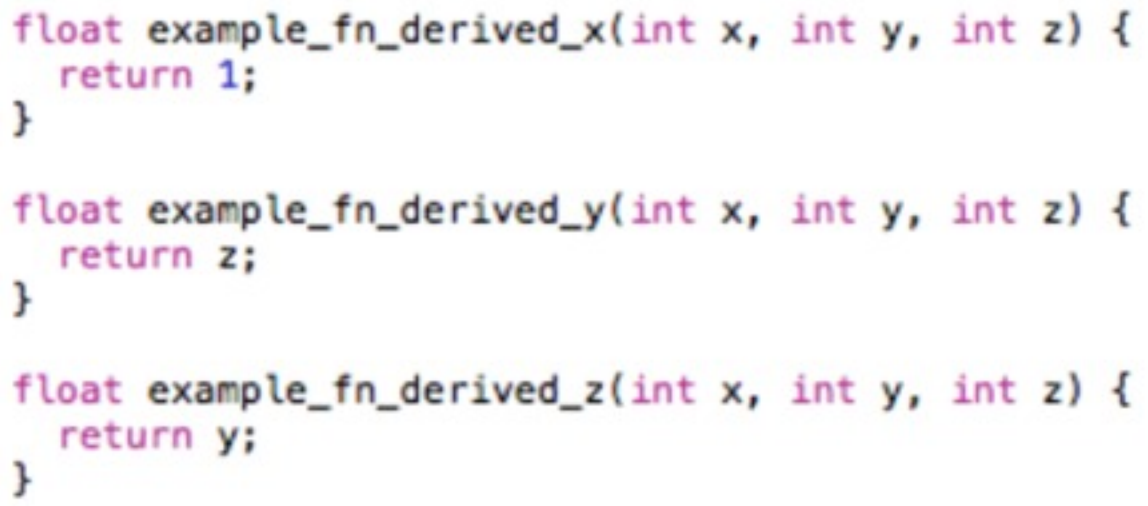
- Prototype implementation of AD as a plug-in for Cling
  - *Developed by Violeta Ilieva (Google Summer of Code 2013 student from Princeton University) and Vassil Vassilev (CERN PH-SFT)*
  - Uses forward propagation and source code transformation
  - Uses Cling to parse code and create derivative functions





```
float example_fn(int x, int y, int z) {
  return x + y * z;
}

  diff(example_fn, x); // = 1
  diff(example_fn, y); // = z
  diff(example_fn, 2); // = y
```



```
float example_fn_derived_x(int x, int y, int z) {
  return 1;
}

float example_fn_derived_y(int x, int y, int z) {
  return z;
}

float example_fn_derived_z(int x, int y, int z) {
  return y;
}
```

Conclusions

- The new `TFormula` leverages the Cling functionality
  - Will be integrated in ROOT 6
  - Provides a more robust and faster evaluation of functions
  - Scales to very large expressions
  - Can be used for building the parametric functions for fitting
  - Integration with `TTreeFormula` and RooFit in the pipeline
- Integrate Auto-Differentiation developments
  - Very interesting and useful technique which can be integrated in ROOT and RooFit to compute derivatives of functions
  - Will speed-up minimization (fitting) of very complex functions
    - E.g. models used at LHC (Higgs combination model)
- Thanks to Google for allocating us GSoC students!