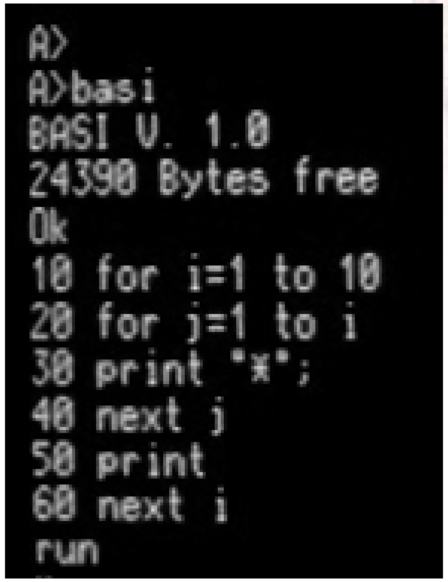
The Role of Interpreters in High Performance Computing



```
A>
A>basi
BASI V. 1.0
24390 Bytes free
Ok
10 for i=1 to 10
20 for j=1 to i
30 print "x";
40 next j
50 print
60 next i
run
```

ACAT 2008

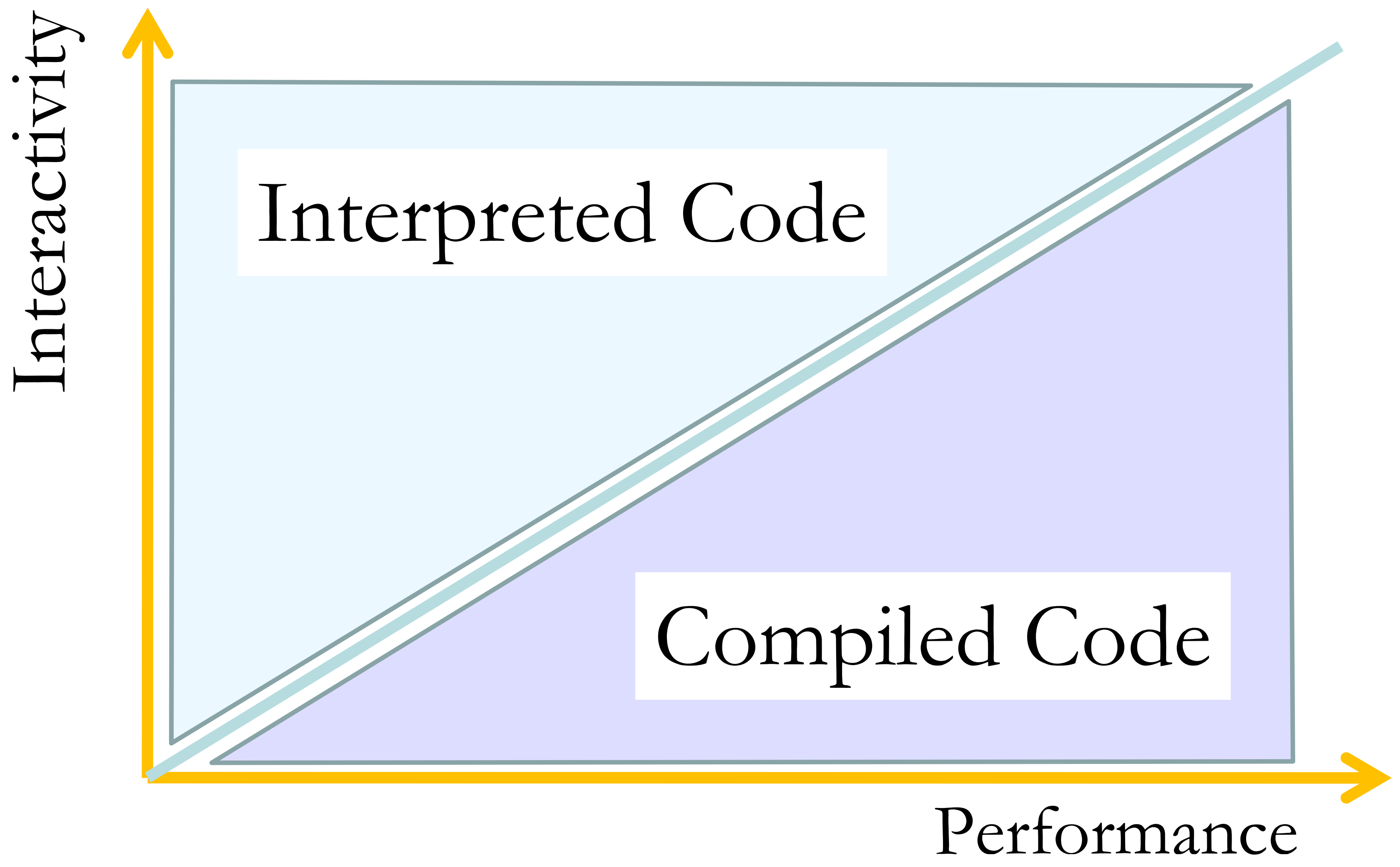
Axel Naumann (CERN), Philippe Canal (Fermilab)

Applications

Wide range:

- Job management: submission, error control
- Gluing programs and configurations
- "Volatile" algorithms subject to change / part of the configuration

Traditional Area of Use



From Text…

Analyses subject to change

- Different cuts, parameters
- Different input / output

(More or less) easy to configure using text files:

```
JetETMin: 12
NJetsMin: 2
```

```
<JetETMin value="12"/>
<NJetsMin value="2"/>
```

or databases

… To Code: Volatile Algorithms

Changes to algorithms themselves

Especially during development

» two jets and one muon each

» three jets and two muons anywhere

» no isolated muon

```
TriggerFlags.doMuon=False
EFMissingET_Met.Tools = \
   [EFMissingETFromFEBHeader()]
```

Configuration not trivial!

Algorithms As Configuration

Acknowledge physicists' reality:

- Refining analyses is asymptotic process
- Programs and algorithms change
- Often tens or hundreds of optimization steps before target algorithm is found
- **Almost** the same:
  - » background analysis vs. signal analysis
  - » trigger A vs. trigger B

Why Use Interpreters?

Slower than compiled code

Difficult to quantify:

- nested loops `foreach event { foreach muon {...`
- calls into libraries `hist.Draw()`
- virtual functions, etc.

Usually O(1)-O(10) slower than compiled code

Interpreters ***can not*** replace compiled code!

Why Use Interpreters?

Slower than compiled code

Not integrated well with reconstruction software

Can be unreliable

Not part of the build system

Difficult to debug

Not convincing! And yet…

Compiled vs. Interpreter

Compiled:

usually many packages need changes by *regular physicists* as opposed to release managers

Interpreter:

helps localize changes,

modular algorithmic test bed

Interpreter Advantage: Localized

Compiled: distributed changes

Interpreter: localized changes

- Easier to track (CVS / SVN)
- Less side effects
- Feeling of control over software
- Eases communication / validation of algorithms

Interpreter Advantage: Data Access

Easier access to data:

- Interpreters can analyze data format, interpret code
- Hide data details irrelevant for analysis:

  `vector` – `hash_map` – `list`? Who cares!

  ```
  foreach electron {...
  ```

- Framework provides job setup transparently

  ```
  MyAnalysis(const Event& event)
  ```

Interpreter Advantage: Agility

Interpreter boosts users' agility

```
virtual int Run() {
  if (NJetsCut(postzerojets))
    VJetsPlots(postZeroJetPlots);
}
```

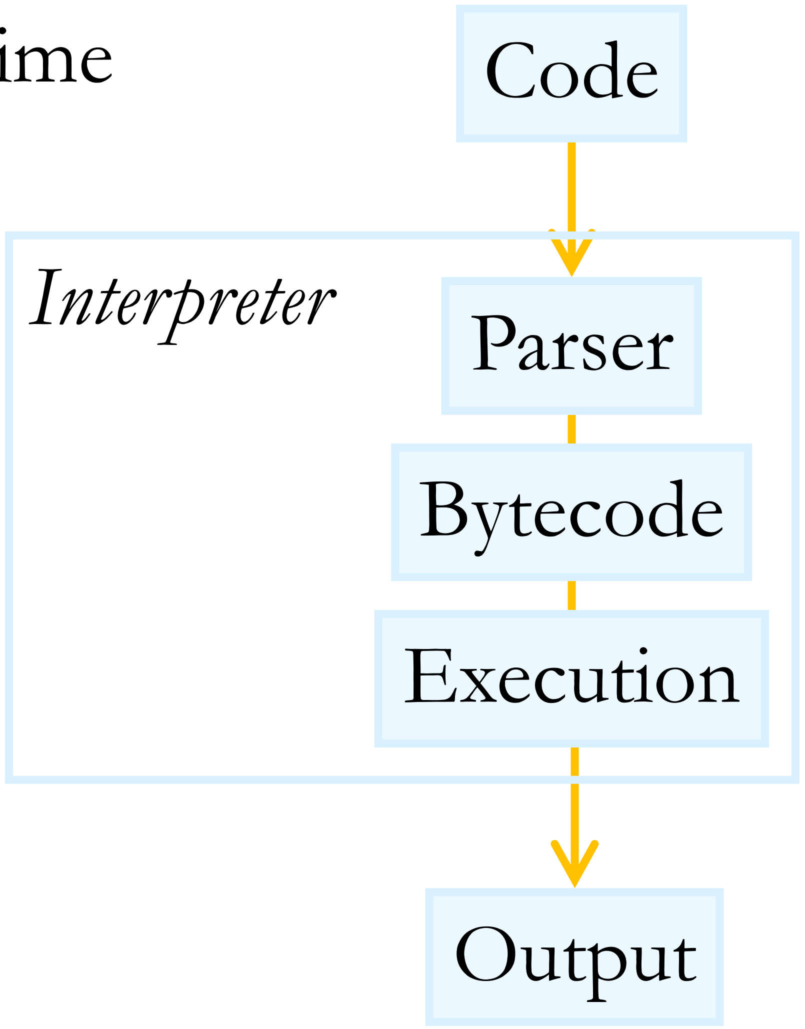
compared to configuration file

```
+postZeroJets.Run: NJetsCut(postzerojets) \
                   VJetsPlots(postZeroJetPlots)
```

- more expressiveness
- thus higher threshold for recompilation of the framework

Ideal Interpreter

1. Fast, e.g. compile just-in-time
2. No errors introduced: quality of all ingredients



Ideal Interpreter

3. Smooth transition to compiled code:
   With compiler or conversion to compiled language
4. Straight-forward use: known / easy language
   Possible extensions with conversion to e.g. C++

```
foreach electron in tree.Electrons
```

```
vector<Electron>* ve = 0;
tree->SetBranchAddress("Electrons", ve);
tree->GetBranch("Electrons")->GetEntry(ev);
for (int i=0; i<ve.size(); ++i) {
  Electron* electron = ve[i];
```

Common Languages

- Job management: shell (bash)
- Glue: shell, python
- Algorithms: python, C++, custom

Focus on volatile algorithms.

Common Interpreter Options: Custom

Even though not interpreted as interpreter:

Parameters

```
postzerojets.nJetsMin: 0
postzerojets.nJetsMax: 0

+postZeroJets.Run: NJetsCut(postzerojets) \
                   VJetsPlots(postZeroJetPlots)

postzerojets.JetBranch: %{VJets.GoodJet_Branch}
```

Algorithm

Common Interpreter Options: Python

- Distinct interpreter language
- Interface to ROOT
- Rigid style: easy to learn, easy to read, easy to communicate

```
h1f = TH1F('h1f', 'Test', 200, 0, 10)
h1f.SetFillColor(45)
h1f.FillRandom('sqroot', 10000)
h1f.Draw()
```

Python: Abstraction

Real power is abstraction:

- can do without types:

```
h1f = TH1F(...)
```

- can loop without knowing collection:

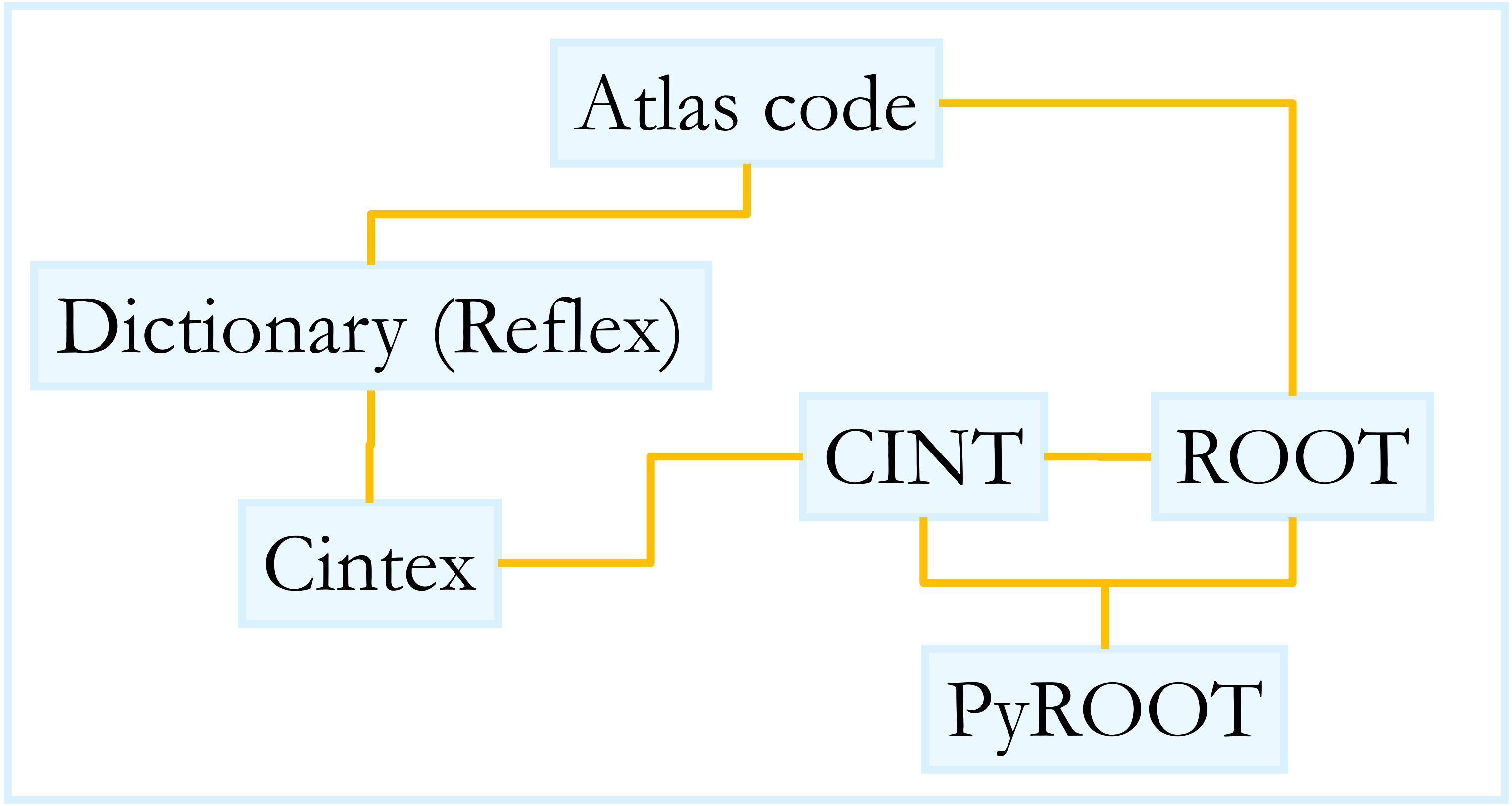
```
for event in events:
  muons = event.Muons
  for muon in muons:
    print muon.pt()
```

Major weakness:

compile time errors become runtime errors

PyROOT: The Maze

ROOT's python interface for e.g. Atlas:



Common Interpreter Options: CINT

- C++ should be prerequisite to data analysis anyway – interpreter often used for first steps
- Seamless integration with C++ software, e.g. ROOT itself
- Can migrate code to framework!
- Rapid edit/run cycles compared to framework

```
void draw() {
  TH1F* h1 = new TH1F(...);
  h1->Draw();
}
```

Common Interpreter Options: CINT

- Forgiving: automatic #includes, library loading

```
// load libHist.so
#include "TH1.h"

void draw() {
  TH1F* h1 = new TH1F(...);
  h1->Draw();
}
```

Major issue: correctness

Common Interpreter Options: CINT

Covers large parts of ISO C++:
templates, virtual functions, etc.

>15 years of development!

Can be invoked from compiled code:

```
gROOT->ProcessLine("new Klass(12)");
```

Or from prompt, e.g. on a whole C++ file:

```
root [0] .L MyCode.cxx
```

CINT And Libraries

Call into library:

```
TH1F* h1 = new TH1F(...);
h1->Draw();
```

Even custom library:

```
root [0] gSystem->Load("Klass.so")
root [1] Klass* k = Klass::Gimme()
root [2] k->Say()
```

Knows what `"Klass"` is!

Translates `"Klass::Gimme()"` into a call!

CINT And Dictionaries

CINT must know available types, functions

Extracted by special CINT run from library's headers (alternatives exist)

Often provided by experiments' build system

Also prerequisite for data storage, see "Data and C++" tomorrow at 3pm

CINT And ACLiC

The plus in

```
root [0] .L MyCode.cxx+
```

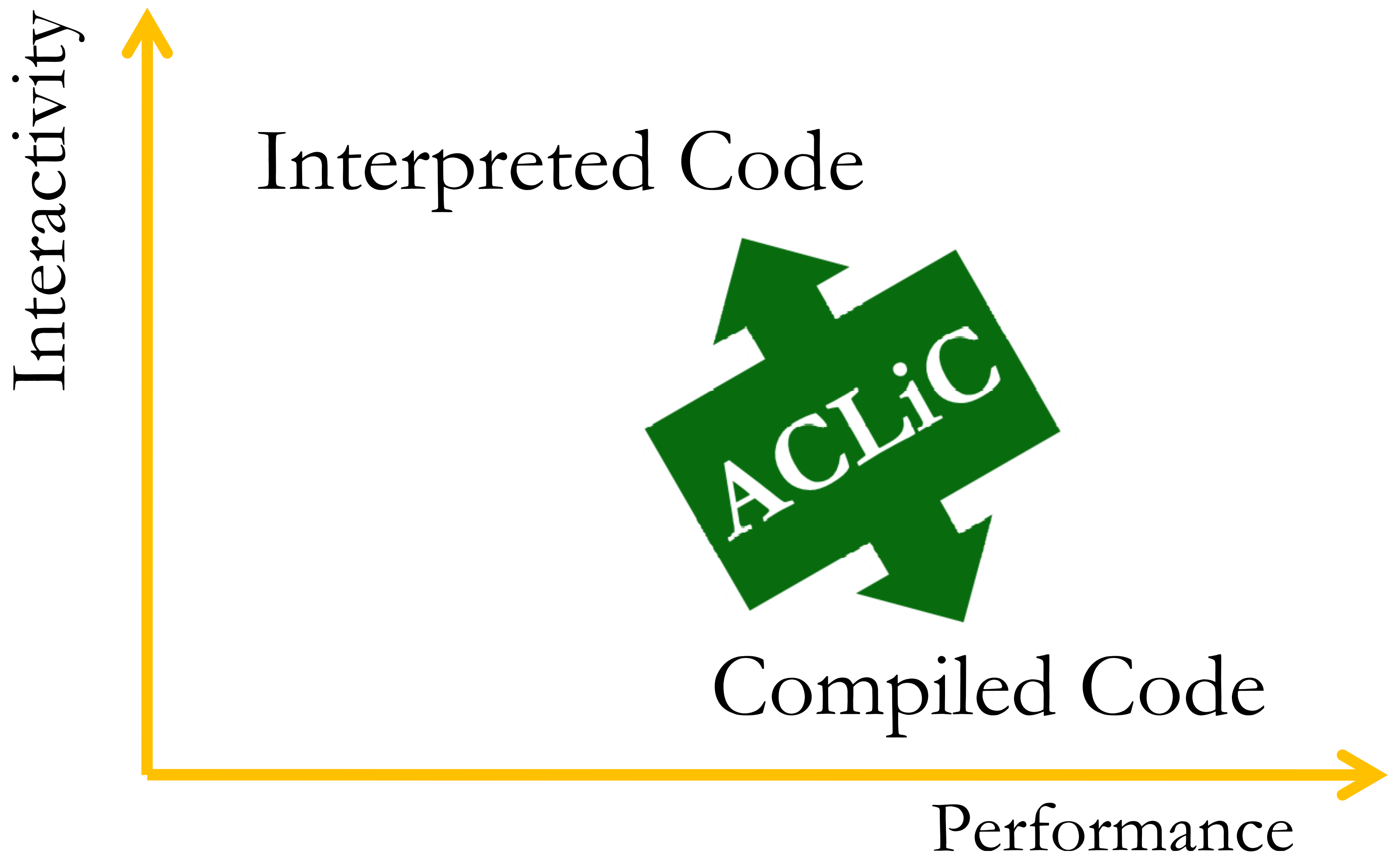
Invokes:

- dictionary generator
- compiler
- linker
- with dependency tracking!

Any platform, any compiler, with any libraries!

Trivial transition from interpreted to compiled!

Less Walls With ACLiC



LLVM

Alternative to CINT based on LLVM?

See plenary talk by ***Chris Lattner*** tomorrow morning

LLVM is much more than a compiler

Modular design, allows us to hook e.g. into

- output of parser,
- language-independent code representation (IR)

Offers JIT, bytecode interpreter…

Summary: Interpreters

Wide spectrum of applications and solutions

Python and CINT are widespread and reasonable options with different use cases

1. Must not replace compiled analysis!

2. No good reason to invent custom configuration language: don't forget the algorithms!

Summary: C++ Interpreter

CINT has known issues

Transparent transition between interpreted and compiled world: huge benefit

We have many years of experience, several options for improving it

Keep interpreted C++ on stage!