Geant4 Python Interface

Koichi Murakami
KEK / CRC

Table of Contents

- Introduction
- Technical Aspects
- Practical Aspects
  - Installation
  - How to use
  - How to expose your applications
- Summary

Introduction

Python

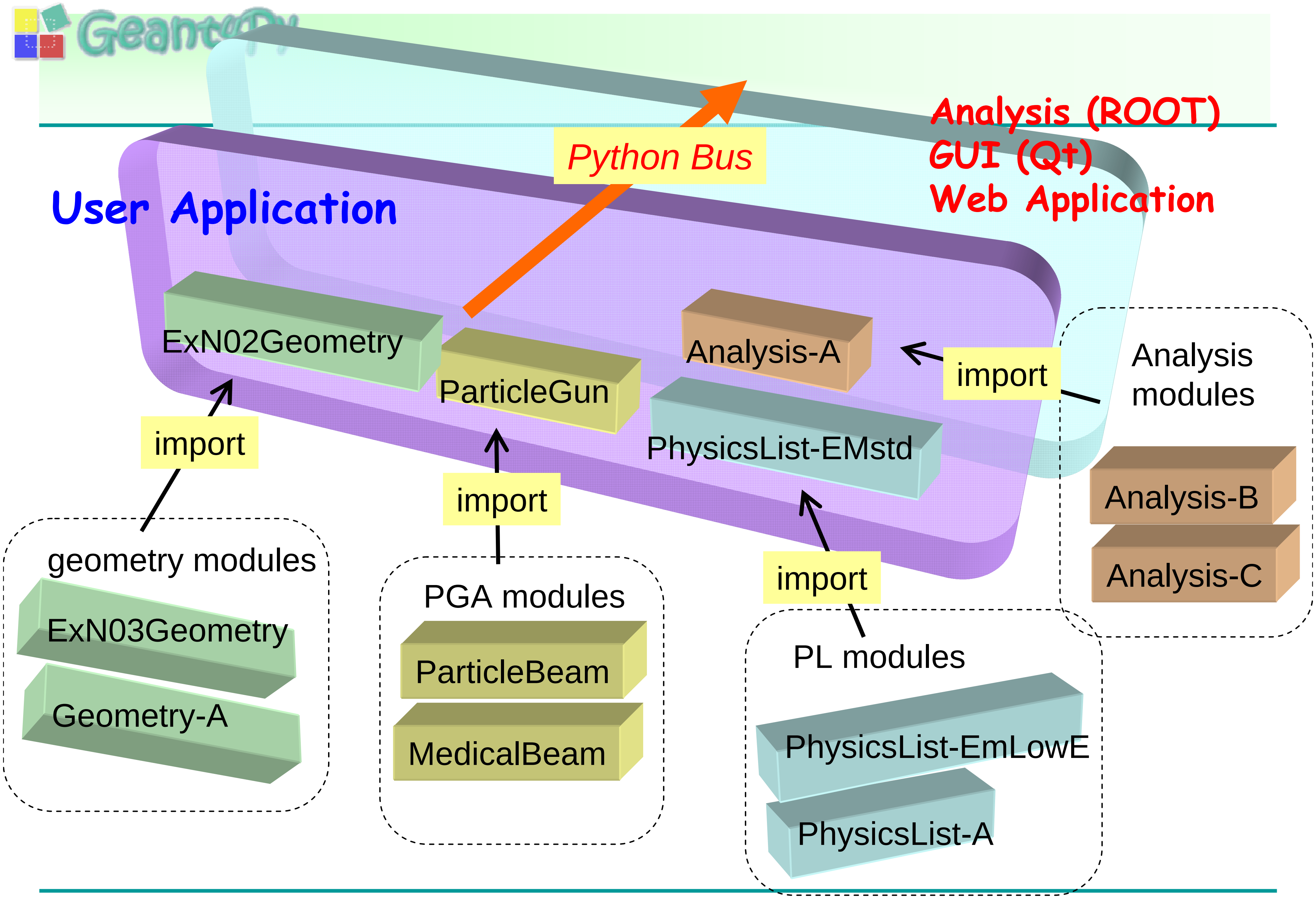
- Shell Environment
  - front end shell
  - script language
- Programming Language
  - easy to program
    - Scripting language is much easier than C++.
  - supporting Object-Oriented programming
  - providing multi-language binding (C-API/JYTHON)
  - dynamic binding
    - modularization
    - software component bus
- Runtime Performance
  - slower than compiled codes, but not so slow.
  - Performance could be tunable between speed and interactivity.

Motivation of Geant4-Python Bridge

- Missing functionalities of current Geant4 UI
  - more powerful scripting environment
    - flow control, variables, arithmetic operation
  - direct object handling for G4XXX classes
    - only limited "manager-like" classes can be exposed via G4UImessenger.
- Python is the most promising technological choice in terms of
  - A powerful scripting language
    - Python can work as a interactive front end.
    - Modularization of user classes with dynamic loading scheme
      - *DetectorConstruction, PhysicsList, PrimaryGeneratorAction, UserAction-s*
      - It helps avoid code duplication.
  - Software Component Bus
    - C++ objects can be exposed to python.
    - Interconnectivity with many Python external modules,
      - analysis tools (ROOT/AIDA), web interface,...

Modular Approach and Software Component Bus

- Modularizing, combining, and using components
  - Material
    - predefined materials (NIST materials, …)
  - Geometry
    - loading predefined geometries (native C++/GDML/…)
  - Physics list
    - EM, Hadron, Ion, …
  - Detector response
    - Calorimeter, Tracker, ..
  - Analysis packages
    - ROOT, HBOOK, AIDA, …
  - Visualization
  - GUI
    - Qt, Tk, …
  - Web applications


Geant4Py
Python Bus
Analysis (ROOT)
GUI (Qt)
Web Application
User Application
ExN02Geometry
ParticleGun
Analysis-A
PhysicsList-EMstd
import
import
import
import
import
Analysis modules
Analysis-B
Analysis-C
geometry modules
ExN03Geometry
Geometry-A
PGA modules
ParticleBeam
MedicalBeam
PL modules
PhysicsList-EmLowE
PhysicsList-A

Technical Aspects

Conceptual Design of Geant4Py

- *"Natural Pythonization"* of Geant4
  - not specific to particular applications
  - There are no invention of new conceptual ideas and terminologies!
  - keeping compatibility with the current UI scheme
  - exposing secure methods only
    - avoiding to expose *"kernel-internal-control"* methods
  - minimal dependencies of external packages
    - only depending on *Boost-Python C++ Library*, which is a common, well-established and freely available library.

Geant4Py Module Structure

- Python package name : "**Geant4**"
  - It consists of a collection of modules same as Geant4 directory structure.
    - *run/event/particle/geometry/track/...*

```
# __init__.py
from G4run import *
from G4event import *
from G4event import *
...
```

  - including CLHEP components typedef-ed as G4XXX, like
    - G4ThreeVector, G4RotationMatrix, ...
    - Units definition (mm, cm, kg, ...)

- From users side,

```
>>> import Geant4
>>> from Geant4 import *
```

Name Policy in Python Side

- Names of classes as well as methods are same as used in Geant4.

```
>>> gRunManager= Geant4.G4RunManager()
>>> gRunManager.BeamOn(10)
```

  - This makes it easy to translate from C++ to Python, and vice versa.

- As an exception, pure singleton class, which has no public constructor, like G4UImanager, can not be exposed in Boost-Python. So, necessary members of such classes are exposed directly in the Geant4 namespace.

```
>>> Geant4.gApplyUIcommand("/run/beamOn")
>>> Geant4.gGetCurrentValues("/run/verbose")
>>> Geant4.gStartUISession()
```

  - Each name is still similar to a corresponding method.

Geant4Py

What is/isnot Exposed to Python

What is exposed:

- Classes for main Geant4 flow control
  - G4RunManager, G4UImanager, G4UIterminal
- Some Utility classes
  - G4String, G4ThreeVector, G4RotationMatrix, ...
- Classes of base classes of user actions
  - G4UserDetetorConstruction, G4UserPhysicsList,
  - G4UserXXXAction (PrimaryGenerator, Run, Event, Stepping,...)
  - can be inherited in Python side
- Classes having information to be analyzed
  - G4Step, G4Track, G4StepPoint, G4ParticleDefinition, ...
- Classes for construction user inputs
  - G4ParticleGun, G4Box, G4PVPlacement, ...

What is not exposed:

- NOT all methods are exposed.
  - only safe methods (getting internal information) are exposed.
- Out of Scope
  - implementation of physics processes
  - implementation of internal control flows
  - It just ends in deterioration of performance.

Expose with Boost-Python

```
#include <boost/python.hpp>
#include "G4Step.hh"
using namespace boost::python;

void export_G4Step()
{
  class_<G4Step, G4Step*>("G4Step", "step class")
    .def("GetTrack",                  &G4Step::GetTrack,
         return_value_policy<reference_existing_object>())
    .def("GetPreStepPoint",           &G4Step::GetPreStepPoint,
         return_internal_reference<>())
    .def("GetPostStepPoint",          &G4Step::GetPostStepPoint,
         return_internal_reference<>())
    .def("GetTotalEnergyDeposit",     &G4Step::GetTotalEnergyDeposit)
    .def("GetStepLength",             &G4Step::GetStepLength)
    .def("GetDeltaPosition",          &G4Step::GetDeltaPosition)
    .def("GetDeltaTime",              &G4Step::GetDeltaTime)
    .def("GetDeltaMomentum",          &G4Step::GetDeltaMomentum)
    .def("GetDeltaEnergy",            &G4Step::GetDeltaEnergy)
    ;
}
```

Global Variables/Functions

- Some global variables/functions starting with "**g**“ are predefined.
  - Singleton objects / methods of pure singleton classes / static member functions;
    - `gRunManager`
    - `gVisManager`
    - `gApplyUIcommand()`
    - `gGetCurrentValues()`
    - `gStartUISession()`
  - *gRunManager* and *gVisManager* are taken care not so as to be doubly instantiated, so that users do not have to take any more care about the timing of object instantiation in python side.
  - All of visualization drivers (OpenGL, VRML, DAWN, ...) are automatically registered. So users are now free from implementation of VisManager.

Global Variabales

- gRunManager
- gEventManager
- gStackManager
- gTrackingManager
- gStateManager
- gTransportationManager
- gParticleTable
- gProcessTable
- gNistManager
- gVisManager
- gMaterialTable
- gElementTable
- gApplyUICommand()
- gGetCurrentValues()
- gStartUISession()
- gControlExecute()

Bridge to G4UImanager

- Geant4Py provides a bridge to G4UImanager.
  - Keeping compatibility with current usability
- UI Commands
  - **`gApplyUICommand("/xxx/xxx")`** allows to execute any G4UI commands.
  - Current values can be obtained by **`gGetCurrentValues("/xxx/xxx")`**.
- Existing G4 macro files can be reused.
  - **`gControlExecute("macro_file_name")`**
- Front end shell can be activated from Python
  - **`gStartUISession()`** starts G4UIsession.

Trace of Geant4 Version

- "G4VERSION_NUMBER" will be introduced in `global/management/include/version.hh` (8.0) for identifying the each G4 version number.
  - "setenv G4VERSION_NUMBER" is required for versions (7.0/7.0.p1/7.1/7.1.p1).

```
// Numbering rule for "G4VERSION_NUMBER":
// - The number is consecutive (i.e. 711) as an integer.
// - The meaning of each digit is as follows;
//
//   711
//   |--> major version number
//    |--> minor version number
//     |--> patch number

#ifndef G4VERSION_NUMBER
#define G4VERSION_NUMBER  711
#endif

#ifndef G4VERSION_TAG
#define G4VERSION_TAG "$Name:  $"
#endif

static const G4String G4Version = "$Name:  $";
static const G4String G4Date    = "(25-Oct-2005)";
```

Site-module package

- We will also provide site-module package as pre-defined components.
  - Material
    - sets of pre-defined materials
      - NIST materials via G4NistManager
  - Geometry
    - "exN03" geometry as pre-defined geometry
    - *"EZgeometry"*
      - provides functionalities for easy geometry setup (applicable to target experiments)
  - Physics List
    - pre-defined physics lists
    - (easy access to cross sections, stopping powers, ... via *G4EmCalculator)*
  - Primary Generator Action
    - particle gun / particle beam
  - Sensitive Detector
    - calorimeter type / tracker type
- They can be used just by importing modules.
- They can be combined and connected to higher application layers (Analysis / GUI components).

Practical Aspects

Installation

How to use

How to expose

Software Requirements

- All libraries should be compiled in shared libraries.
- Python
- BOOST-Python
  - *1.32, latest*
- Geant4
  - *7.0 or later*
  - should be built in "global" and "shared" libraries.
  - All header files should be collected into $(G4INSTALL)/include by "make includes"
- CLHEP
  - *1.9.1.1 or later*
  - building shared objects is supported since version 1.9.
- Platforms
  - Linux system is ok.
    - SUSE Linux 9.3 is a development environment.
      - It is the easiest way to go, because Boost C++ library is preinstalled.
    - Scientific Linux (SL3/SL4) is checked for well working.
  - Mac OSX has some troubles with
    - Boost-python
    - Creating shared library of CLHEP1.9.xx
  - Win32-Cygwin is bad idea. Win32-VC could be better.

Why do you have to use shared libraries?

- Linking against static libraries results in multiple or incomplete copies of a library
  - What happens :
    - libXXX.a is exposed to both “foo” and “bar”;

```
>>> import foo
>>> import bar
>>> bar.set_spam(42)
>>> print foo.get_spam()
7
```

    - Objects in static library will be locally copied!!
- You have to use shared libraries!!

Boost-Python

- Boost-python is a part of Boost C++ libray.
  - http://www.boost.org/
  - "bjam" is also required to build Boost package.
  - Some recent Linux distributions include Boost packages.
    - included in SuSE 9.3, Fedra Core.
    - SuSE10.0 does not Boost-Python. So you have to reinstall manually.
  - To build and install Boost,
    - `bjam --prefix="xxx/xxx" "-sTOOLS=gcc" "-sBUILD=release" install`

Geant4 global shared library

- Environment variables
  - You can co-work with "normal" granular static libraries.

```
setenv G4BUILD_SHARED =1
setenv G4LIB = G4INSTALL/slib
setenv G4TMP = G4INSTALL/tmp-slib
```

- How to build library

```
> make global
> make includes
```

- Global libraries are required because Geant4Py does not know which granular libraries are used in your application.
  - dynamically linked

How to Build Geant4Py

- Some environment variables are required to install Geant4Py modules.

```
Minimal set of environment variables are:
+-----------------------------------------------------------------+
 G4PY_INSTALL         | install directory of Geant4Py package
 G4VERSION_NUMBER     | Geant4 version in three digits (eg. 700)
                      | required only for 7.0/7.0.p01/7.1
 G4SYSTEM*            | system type
 G4INSTALL*           | install directory of Geant4
 CLHEP_BASE_DIR*      | install diretory of CLHEP
+-----------------------------------------------------------------+
 G4LIB_BUILD_SHARED*| must be defined (true) if you will build
                      | your own G4 applications
+-----------------------------------------------------------------+
(*) used also in Geant4
```

Optional environment variables:

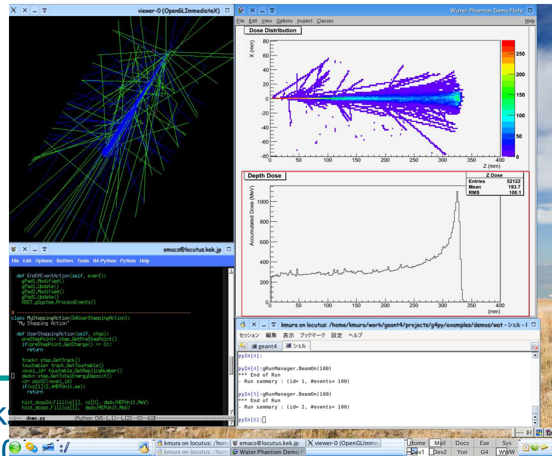
```
+-------------------+---------------------------------------+
 PYTHON_INCDIR      | Python include director
                    | [/usr/include/python]
 BOOST_INCDIR       | Boost include directory
                    | [/usr/include/(boost)]
 BOOST_LIBDIR       | Boost library directory  [/usr/lib]
 BOOST_PYTHON_LIB   | library name of Boost-python
                    | [boost_python]
+-------------------+---------------------------------------+
```

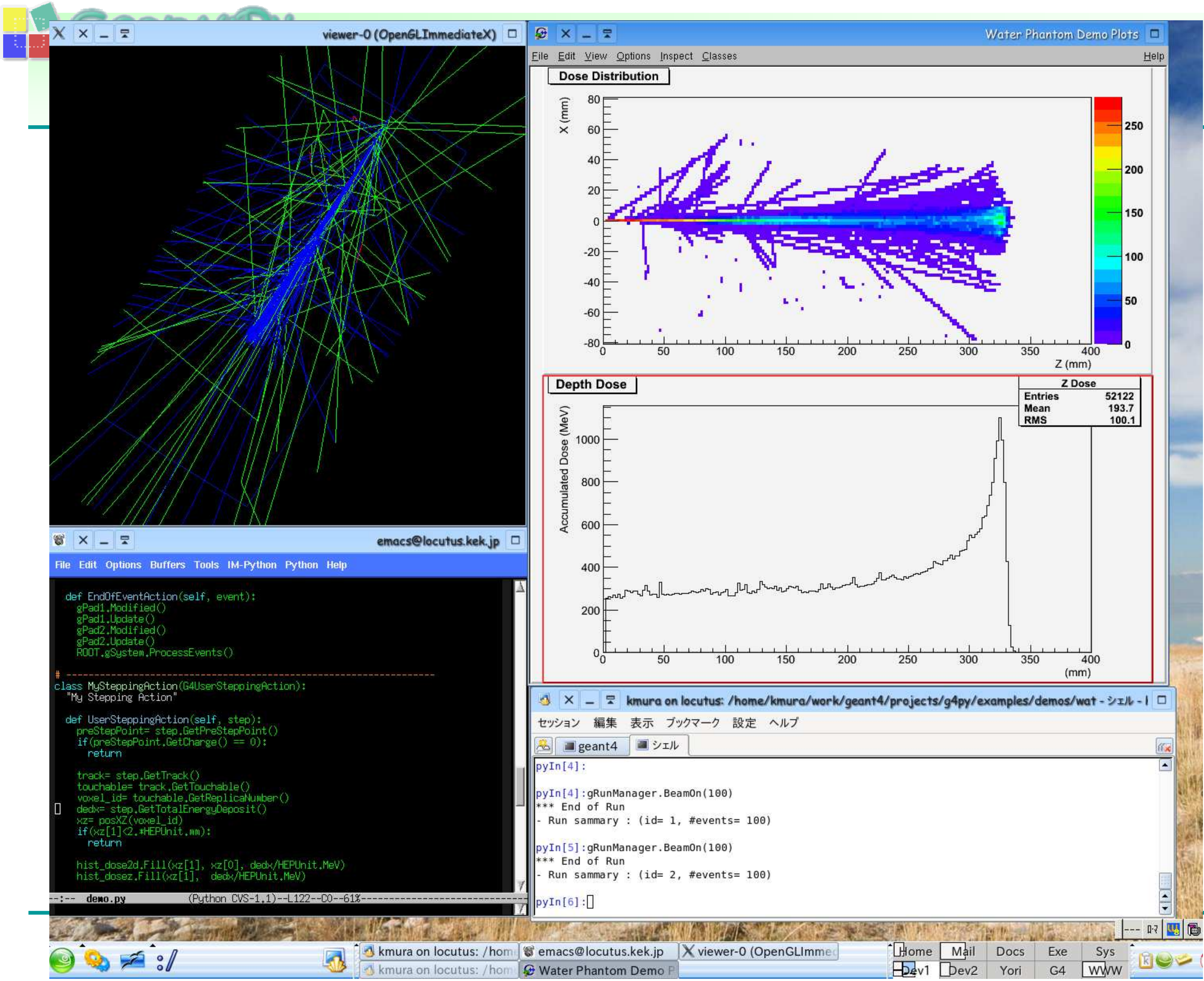
- Environment variables for visualization, like G4VIS_USE_XXX, are avaiale. Note that it is required to define "G4VIS_USE_OPENGLX" to expose OpenGL visualization.
- After setting envrionment variables, then just type as

```
> cd $G4PY_INSTALL/source
> make install
> cd $G4PY_INSTALL/site-modules/source
> make install
```

A Medical Application Example

- Several examples of using Python interface are/will be presented.
- An example of "water phantom dosimetry"
  - *This demo program shows that a Geant4 application well coworks with ROOT on the Python front end.*
- You can look features of;
  - dose calculation in a water phantom
  - Python implementation of sensitive detector
  - Python overloading of user actions
  - on-line histogramming with ROOT
  - visualization




viewer-0 (OpenGLImmediateX)
Water Phantom Demo Plots
File Edit View Options Inspect Classes Help
Dose Distribution
X (mm)
Z (mm)
Depth Dose
Z Dose
Entries 52122
Mean 193.7
RMS 100.1
Accumulated Dose (MeV)
(mm)
emacs@locutus.kek.jp
File Edit Options Buffers Tools IM-Python Python Help
def EndOfEventAction(self, event):
gPad1.Modified()
gPad1.Update()
gPad2.Modified()
gPad2.Update()
ROOT.gSystem.ProcessEvents()
class MySteppingAction(G4UserSteppingAction):
"My Stepping Action"
def UserSteppingAction(self, step):
preStepPoint= step.GetPreStepPoint()
if(preStepPoint.GetCharge() == 0):
return
track= step.GetTrack()
touchable= track.GetTouchable()
voxel_id= touchable.GetReplicaNumber()
dedx= step.GetTotalEnergyDeposit()
xz= posXZ(voxel_id)
if(xz[1]<2.*HEPUnit.mm):
return
hist_dose2d.Fill(xz[1], xz[0], dedx/HEPUnit.MeV)
hist_dosez.Fill(xz[1], dedx/HEPUnit.MeV)
--:-- demo.py (Python CVS-1.1)--L122--C0--61%
kmura on locutus: /home/kmura/work/geant4/projects/g4py/examples/demos/wat - シェル
セッション 編集 表示 ブックマーク 設定 ヘルプ
geant4 シェル
pyIn[4]:
pyIn[4]:gRunManager.BeamOn(100)
*** End of Run
- Run sammary : (id= 1, #events= 100)
pyIn[5]:gRunManager.BeamOn(100)
*** End of Run
- Run sammary : (id= 2, #events= 100)
pyIn[6]:

An Example of Exposure of Users' Application

- Uses' existing applications are easily exposed to python following a simple prescription of Boost-Python manner.

```
BOOST_PYTHON_MODULE(demo_wp){
  class_<MyApplication>("MyApplication", "my application")
    .def("Configure", &MyApplication::Configure)
  ;
  class_<MyMaterials>("MyMaterials", "my material")
    .def("Construct", &MyMaterials::Construct)
  ;
  class_<MyDetectorConstruction, MyDetectorConstruction*,
  bases<G4VUserDetectorConstruction> >
  ("MyDetectorConstruction", "my detector")
  ;
  class_<MyPhysicsList, MyPhysicsList*,
  bases<G4VUserPhysicsList> >
  ("MyPhysicsList", "my physics list")
  ;
}
```

Example of A Python Script

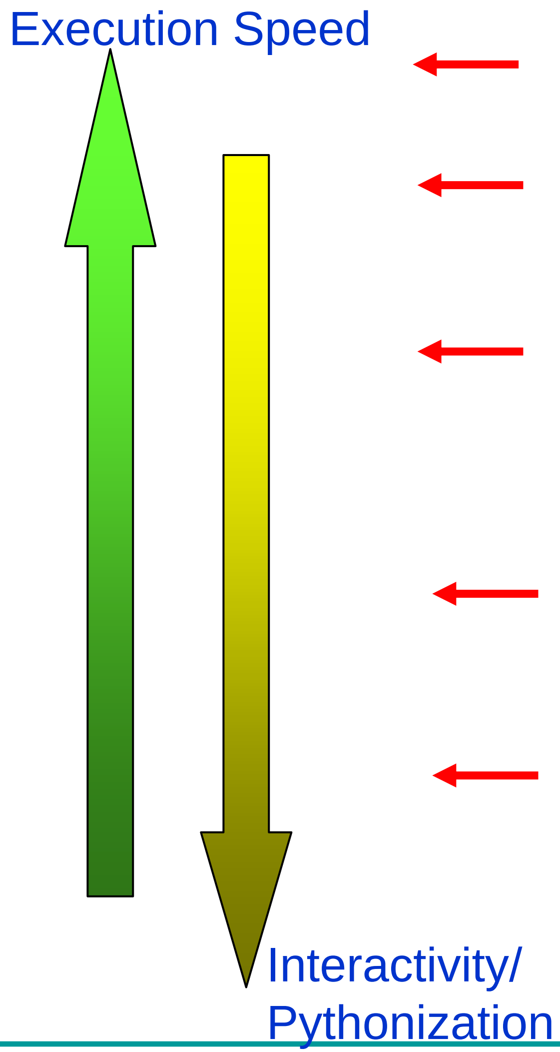
```
from Geant4 import *
import demo_wp    # module of a user G4 application
import ROOT

#  ------------------------------------------------
class ScoreSD(G4VSensitiveDetector):
  "SD for score voxels“
  def __init__(self):
    G4VSensitiveDetector.__init__(self, "ScoreVoxel")
  def ProcessHits(self, step, rohist):
    preStepPoint= step.GetPreStepPoint()
    if(preStepPoint.GetCharge() == 0):
      return
    track= step.GetTrack()
    touchable= track.GetTouchable()
    voxel_id= touchable.GetReplicaNumber()
    dedx= step.GetTotalEnergyDeposit()
    xz= posXZ(voxel_id)
    hist_dose2d.Fill(xz[1], xz[0], dedx/MeV)
    hist_dosez.Fill(xz[1],  dedx/MeV)
```

```
# user detector construction (C++)
myDC= demo_wp.MyDetectorConstruction()
gRunManager.SetUserInitialization(myDC)

# user physics list (C++)
myPL= demo_wp.MyPhysicsList()
gRunManager.SetUserInitialization(myPL)

# user P.G.A (Python)
myPGA= MyPrimaryGeneratorAction()
gRunManager.SetUserAction(myPGA)

# setting particle gun
pg= myPGA.particleGun
pg.SetParticleByName("proton")
pg.SetParticleEnergy(230.*MeV)
pg.SetParticleMomentumDirection(G4ThreeVector(0., 0., 1.))
pg.SetParticlePosition(G4ThreeVector(0.,0.,-20.)*cm)

gRunManager.Initialize()

gApplyUICommand("/control/execute vis.mac")
gRunManager.BeamOn(100)
```

Various Levels of Pythonization

- Various level of pythonized application can be realized.
  - It is completely up to users!
- There are two metrics;
  - Execution Speed
    - just wrapping current existing applications
  - Interactivity
    - interactive analysis
    - rapid prototyping
    - educational use
- Optimized point depends on what you want to do in Python.
  - no performance loss in case of object controller
  - pay performance penalty to interpretation in stepping actions.

Use-case of Pythonization



- large scale of mass production
- a collection of short productions with various running conditions
- applications having interconnections with other software components
- interactive analysis
- free from compilation
  - rapid prototyping
  - educational uses

Some Comments

- Some C++ has no Python equivalent.
  - pointer arithmetic
    - pointer vs. reference
    - pointer/vector – list/tuple conversion
    - These treatments cannot be automated.
  - memory management
- Memory management is different between C++ and Python.
  - In C++, object life span should be controlled manually.
    - new / delete
  - In Python, objects will be deleted automatically when they have zero reference counts.
  - Take care of object life time!
    - To keep objects beyond functions, they should be global.
- Problem with termination time
  - At termination time, Python session ends with segmentation fault because lifetimes of objects are not correctly managed.
  - Please be patient for a while . Don't worry, nothing is losed.

Resources

- Project Home Page
  - http://www-geant4.kek.jp/projects/Geant4Py/
- CVS view
  - http://www-geant4.kek.jp/projects/Geant4Py/cvs/
- Forum
  - Forums for developer and users
  - http://www-geant4.kek.jp/forum/

Summary

- Python Interface of Geant4 (Geant4Py) has been well designed and implementation is now rapidly on-going.

- Python as a powerful scripting language
  - much better interactivity
    - configuration
    - rapid prototyping
- Python as "Software Component Bus"
  - interconnectivity with various kind of software components.
    - histogramming with ROOT
    - system integration

- We have a plan to commit the beta release into the next December release.
  - "environments/" directory is a suitable position