AutoWIG: Automatic generation of `Python` or `R` bindings for `C++` libraries

Pierre Fernique

M2P2, UMR AGAP, Cirad, Inra

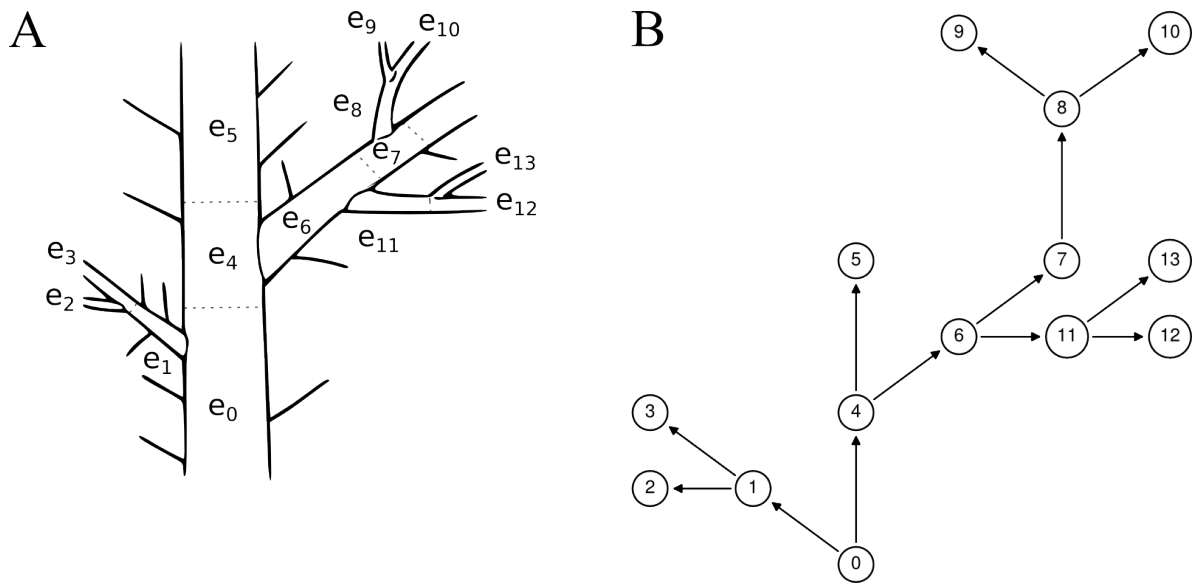
2018/02/19

Introduction

¬ Scientific context

Virtual Plants focuses on plant development and its modulation by environmental and genetic factors:

1. At a macroscopic scale. Each vertex represents a botanical entity and edges encode either the temporal precedence of two botanical entities.



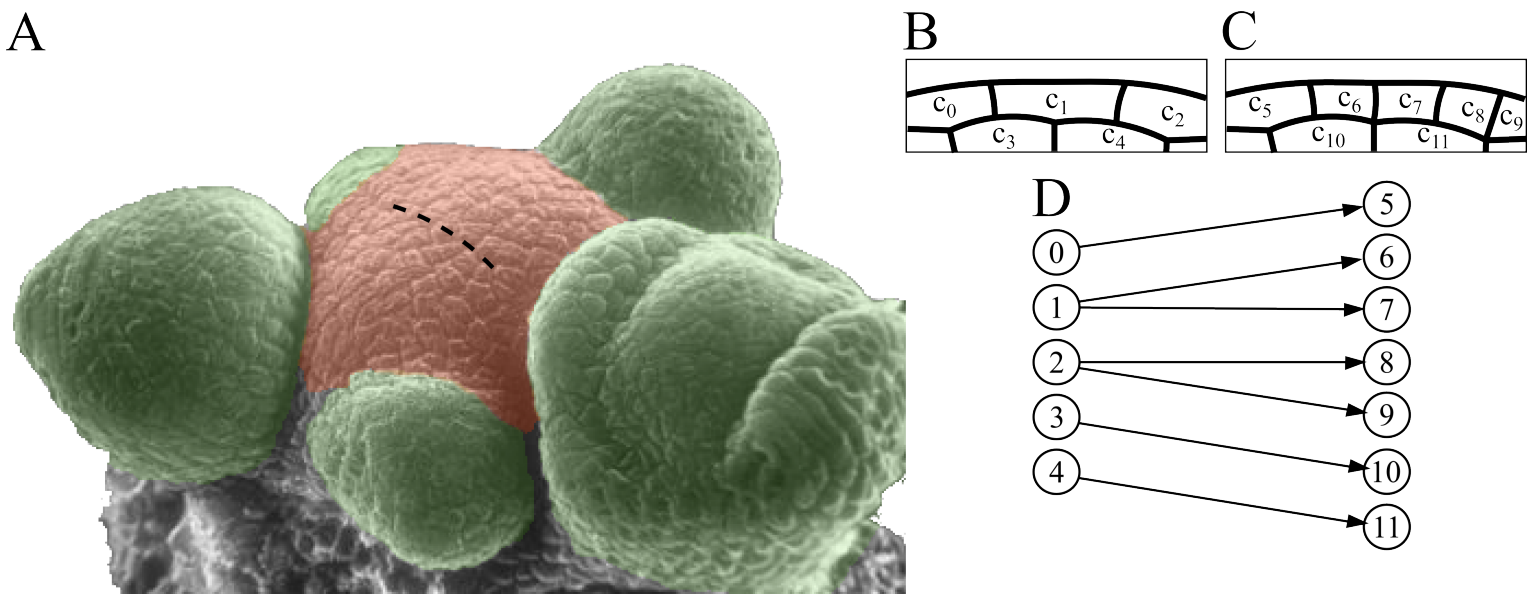
Tree-indexed data extraction from whole plants

Introduction

¬ Scientific context

Virtual Plants focuses on plant development and its modulation by environmental and genetic factors:

2. At a microscopic scale. Each vertex represents a cell and edges encode either the tracking of a cell throughout time or the lineage relationships between parent and child cells.

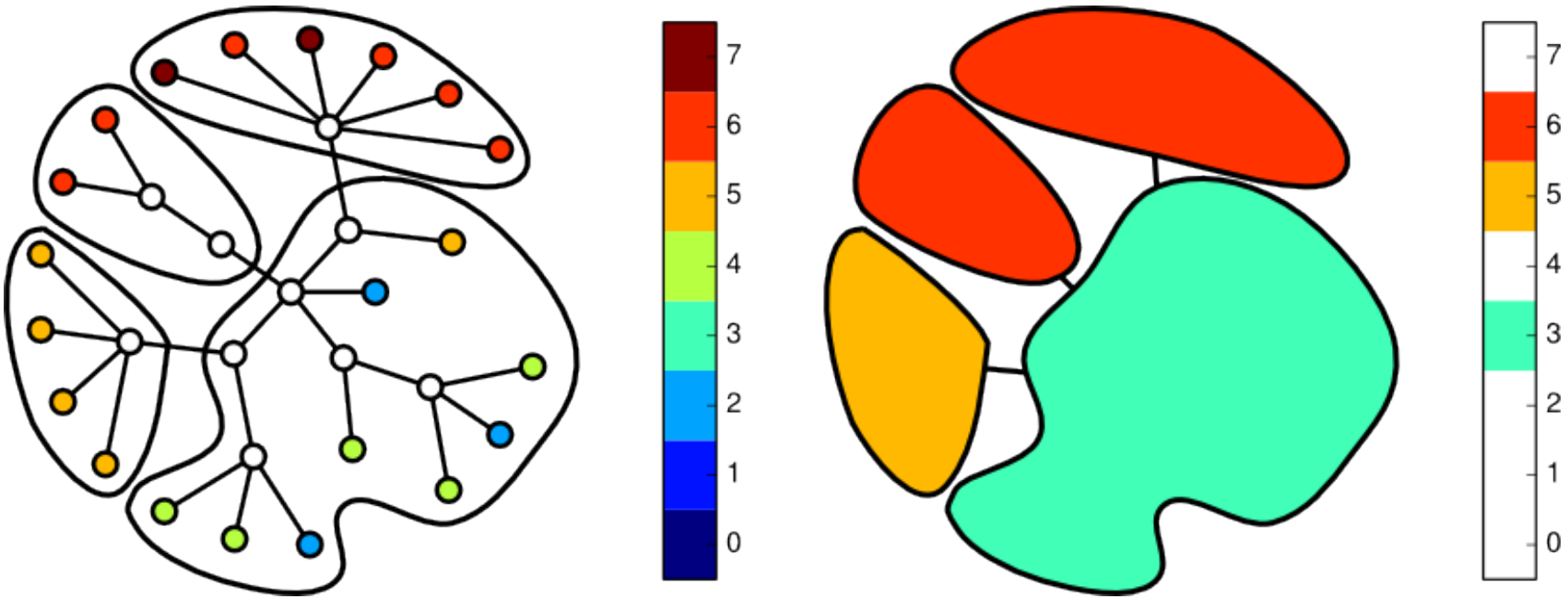


Tree-indexed data extraction from cell lineages

Introduction

¬ Scientific context

M2P2 focuses on statistical analyses for tree-indexed data:



Quotienting model combined with a hidden markov model

Require a programming framework combining

- High-performance,
- Interactivity.

Introduction

¬ Low-level programming languages

Many high-performance scientific libraries are written in low-level programming languages such as:

- `Fortran`,
- `C`,
- `C++`.

Entail the usage of the traditional development chain:

1. edit,
2. compile,
3. execute.

Introduction

¬ Low-level programming languages

Many high-performance scientific libraries are written in low-level programming languages such as:

- `Fortran`,
- `C`,
- `C++`.

Entail the usage of the traditional development chain:

1. edit,
2. compile,
3. execute.

Require a programming framework combining

- High-performance,
- ~~Interactivity~~.

Introduction

¬ Scripting languages

Many interactive scientific packages are written in scripting languages scuh as:

- `Sage` (symbolic mathematics),
- `R` (statistical analyses),
- `Python` (general purposes).

Allow on the fly to:

- explore data,
- test new ideas,
- combine algorithmic approaches.

Introduction

¬ Scripting languages

Many interactive scientific packages are written in scripting languages scuh as:

- `Sage` (symbolic mathematics),
- `R` (statistical analyses),
- `Python` (general purposes).

Allow on the fly to:

- explore data,
- test new ideas,
- combine algorithmic approaches.

Require a programming framework combining

- ~~High-performance~~,
- Interactivity.

Introduction

¬ Python and R bindings for C++ libraries

Python and R are interpreted languages implemented in C.

They provide a C API to allow foreign libraries implemented in:

- C,
- C++,
- Fortran.

Introduction

¬ Python and R bindings for C++ libraries

Python and R are interpreted languages implemented in C.

They provide a C API to allow foreign libraries implemented in:

- C,
- C++,
- Fortran.

Require a programming framework combining

- High-performance,
- Interactivity.

Introduction

¬ Python and R bindings for C++ libraries

Python and R are interpreted languages implemented in C.
They provide a C API to allow foreign libraries implemented in:

- C,
- C++,
- Fortran.

Require a programming framework combining

- High-performance,
- Interactivity.

This C API is designed to be stable but low-level:

- It does not provide support for object-oriented languages.
- Every type and function have to be *manually* wrapped.
- Verbose approach.

This approach is only efficient for exposing few functions and object for developers.

Introduction

¬ Python and R bindings for C++ libraries

Several *semi-automatic* solutions have been proposed to simplify and ease the process of wrapping large C++ libraries:

- **Boost.Python**,
- **Rcpp**.

They depend on meta-programming to provide high-level abstractions:

- registration of classes and inheritance,
- automatic conversion of registered types and classes,
- management of smart pointers,
- C++ object-oriented interface to Python/R objects.

Introduction

¬ Python and R bindings for C++ libraries

```
11  class BinomialDistribution
12  {
13    public:
14      BinomialDistribution(const unsigned int n, const double pi);
15      BinomialDistribution(const BinomialDistribution& binomial);
16
17      double pmf(const unsigned int value) const;
18
19      double get_pi() const;
20      /**
21       * \param pi New probability value
22       * \throws \ref ::ProbabilityError If the new probability value
23       *         is not in the interval \f$\left[0,1\right]\f$ */
24      void set_pi(const double pi);
25
26      unsigned int n;
27
28    protected:
29      double _pi;
30  };
```

Introduction

¬ Python and R bindings for C++ libraries

Several *semi-automatic* solutions have been proposed to simplify and ease the process of wrapping large C++ libraries:

- **Boost.Python**,

```
1  BOOST_PYTHON_MODULE(_binomial) {
2  boost::python::class_< BinomialDistribution >("BinomialDistribution")
3      .def(boost::python::init< const unsigned int, const double >)
4      .def(boost::python::init< const BinomialDistribution& >)
5      .def("pmf", &BinomialDistribution::pmf)
6      .def("get_pi", &BinomialDistribution::get_pi)
7      .def("set_pi", &BinomialDistribution::set_pi, ":Parameter:\n  `pi`
8      .add_property("n", &BinomialDistribution::n)
9      ;
10 }
```

  - Highly similar to C++ headers.

Introduction

¬ Python and R bindings for C++ libraries

Several *semi-automatic* solutions have been proposed to simplify and ease the process of wrapping large C++ libraries:

- **Rcpp**.

```
1  RCPP_MODULE(_binomial) {
2  Rcpp::class_< BinomialDistribution >("BinomialDistribution")
3      .constructor< const unsigned int, const double >()
4      .constructor< const BinomialDistribution& >()
5      .method("pmf", &BinomialDistribution::pmf)
6      .method("get_pi", &BinomialDistribution::get_pi)
7      .method("set_pi", &BinomialDistribution::set_pi, ":Parameter:\n  `p
8      .field("n", &BinomialDistribution::n)
9      ;
10 }
```

  - Highly similar to **Boost.Python**

Introduction

¬ Python and R bindings for C++ libraries

Several *semi-automatic* solutions have been proposed to simplify and ease the process of wrapping large C++ libraries:

- **Boost.Python**,
- **Rcpp**.

```
1  struct BinomialDistribution {
2      BinomialDistribution(const unsigned int n, const double pi);
3      BinomialDistribution(const BinomialDistribution& binomial);
4      double pmf(const unsigned int value) const;
5      double get_pi() const;
6      void set_pi(const double pi);
7      unsigned int n; };
```

- **Highly similar to headers**.
- Lots of knowledge for developers.
- Still cumbersome and error prone.

Requirements

C++ parsing

In order to automatically expose C++ components in Python/R:

- Parsing C++ code implementing the last C++ standard.

```
struct BinomialDistribution {
    BinomialDistribution(const unsigned int n, const double pi);
    BinomialDistribution(const BinomialDistribution& binomial);
    double pmf(const unsigned int value) const;
    double get_pi() const;
    void set_pi(const double pi);
    unsigned int n; };
```

Requirements

C++ parsing

In order to automatically expose C++ components in Python/R:

- Represent C++ constructs in Python/R.

```
BOOST_PYTHON_MODULE(_binomial) {
boost::python::class_< BinomialDistribution >("BinomialDistribution")
    .def(boost::python::init< const unsigned int, const double >)
    .def(boost::python::init< const BinomialDistribution& >)
    .def("pmf", &BinomialDistribution::pmf)
    .def("get_pi", &BinomialDistribution::get_pi)
    .def("set_pi", &BinomialDistribution::set_pi, ":Parameter:\n  `pi`
    .add_property("n", &BinomialDistribution::n)
    ;
}
```

Requirements

C++ parsing

In order to automatically expose C++ components in Python/R:

- Parsing C++ code implementing the last C++ standard.
- Represent C++ constructs in Python/R.

```
>>> from test.binomial import *
>>> binomial = BinomialDistribution(1,.5)
>>> binomial.pmf(0)
0.5
>>> binomial.pmf(1)
0.5
>>> binomial.n = 0
>>> binomial.pmf(0)
1.0
```

Requirements

Documentation

Associate C++ components documentation to their corresponding Python/R components:

- Reduce the redundancy and keep it up-to-date.

```
/**
 * \param pi New probability value
 * \throws \ref ::ProbabilityError If the new probability value
 *         is not in the interval \f£\left[0,1\right]\f£ */
void set_pi(const double pi);
```

Requirements

Documentation

Associate C++ components documentation to their corresponding Python/R components:

- Reduce the redundancy and keep it up-to-date.

```
.def("set_pi", &BinomialDistribution::set_pi, ":Parameter:\n  `pi`
.method("set_pi", &BinomialDistribution::set_pi, ":Parameter:\n  `p
```

Requirements

Documentation

Associate C++ components documentation to their corresponding Python/R components:

- Reduce the redundancy and keep it up-to-date.
- Format it according to language standards.

```
/**
 * \param pi New probability value
 * \throws \ref ::ProbabilityError If the new probability value
 *         is not in the interval \f$\left[0,1\right]\f$ */
void set_pi(const double pi);
```

Requirements

Documentation

Associate C++ components documentation to their corresponding Python/R components:

- Reduce the redundancy and keep it up-to-date.
- Format it according to language standards.

```
>>> help(binomial.set_pi)
:Parameter:
    `pi` (:cpp:any:`double`) - New probability value

:Return Type:
    :cpp:any:`void`

:Raises:
    :py:exc:`test.binomial._binomial.ProbabilityError` - If the new pro
                                                         :math:`\left[0
```

Memory management

- C++ libraries expose in their interfaces either raw pointers, shared pointers or references.
- `Python` handles memory allocation and garbage collection automatically.

Memory management

- C++ libraries expose in their interfaces either raw pointers, shared pointers or references.
- Python handles memory allocation and garbage collection automatically.

Consider the following C++ template function declaration,

```
template<class T> T* ambiguous_function();
```

There is *a priori* no way to know whether the pointer returned should be deleted or not.

Memory management

- C++ libraries expose in their interfaces either raw pointers, shared pointers or references.
- Python handles memory allocation and garbage collection automatically.

Consider the following C++ template function declaration,

```
template<class T> T* ambiguous_function();
```

There is *a priori* no way to know whether the pointer returned should be deleted or not.
Smart pointers overcome these C++ ambiguities.

```
template<class T> std::unique_ptr< T > unambiguous_function();
```

Explicit the fact that the caller takes ownership of the result.

Error management

C++ exceptions need to be consistently managed in Python:

- Handling of C++ exceptions thrown by wrappers.
- Translation into Python errors which preserve information.

```
#include <exception>

class ProbabilityError : public std::exception
{
    /// \brief Compute the exception content
    /// \returns The message "a probability must be in the interval
    ///          [0,1]"
    virtual const char* what() const;
};
```

Error management

C++ exceptions need to be consistently managed in Python:

- Handling of C++ exceptions thrown by wrappers.
- Translation into Python errors which preserve information.

```
>>> binomial.set_pi(1.1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ProbabilityError: a probability must be in the interval [0,1]
```

Methodology

¬ Interactive workflow

Parse

In this step, **AutoWIG** performs:

- a syntactic analysis of headers to produce an AST (**Clang**).
- a semantic analysis of the AST to produce an ASG.

AST & ASG Graph databases within which each C++ component used in headers are represented by a node.

AST Abstract Syntax Tree within which there is no-uniqueness of C++ components.

ASG Abstract Semantic Graph within which there is uniqueness of C++ components.

```
>>> import autowig
>>> asg = autowig.AbstractSemanticGraph()
>>> asg = autowig.parser(asg, ['./test/binomial/binomial.h'],
...                           ['-x', 'c++', '-std=c++11'])
```

Control

In this step, the code introspection using the ASG ensures a consistent wrapping of a C++ library:

- Node removal,
- Simulated refactoring,

```
>>> asg = autowig.controller(asg)
```

Methodology

¬ Interactive workflow

Control

In this step, the code introspection using the ASG ensures a consistent wrapping of a C++ library:

- Node removal,
- Simulated refactoring,
- Set control parameters.

```
>>> asg = autowig.controller(asg)
>>> asg['class ::std::reverse_iterator'].boost_python_export = False
>>> for spc in asg['class ::std::pair'].specializations():
...   if spc.templates[0].is_const:
...     for mtd in spc.methods():
...       if not mtd.is_const:
...         mtd.boost_python_export = False
```

Methodology

¬ Interactive workflow

Generate

In this step, the code introspection using the ASG and predefined rules generates:

- Files containing wapper functions for each component of a C++ library.
- Files defining an user friendly interface for the Python/R bindings.

```
>>> wrappers = autowig.generator(asg,
...                              module='./test/binomial/module.cpp',
...                              decorator=None,
...                              closure=True)
>>> wrappers.write()
>>> autowig.scons('test')
...
scons: done building targets.
```

Methodology

¬ Interactive workflow

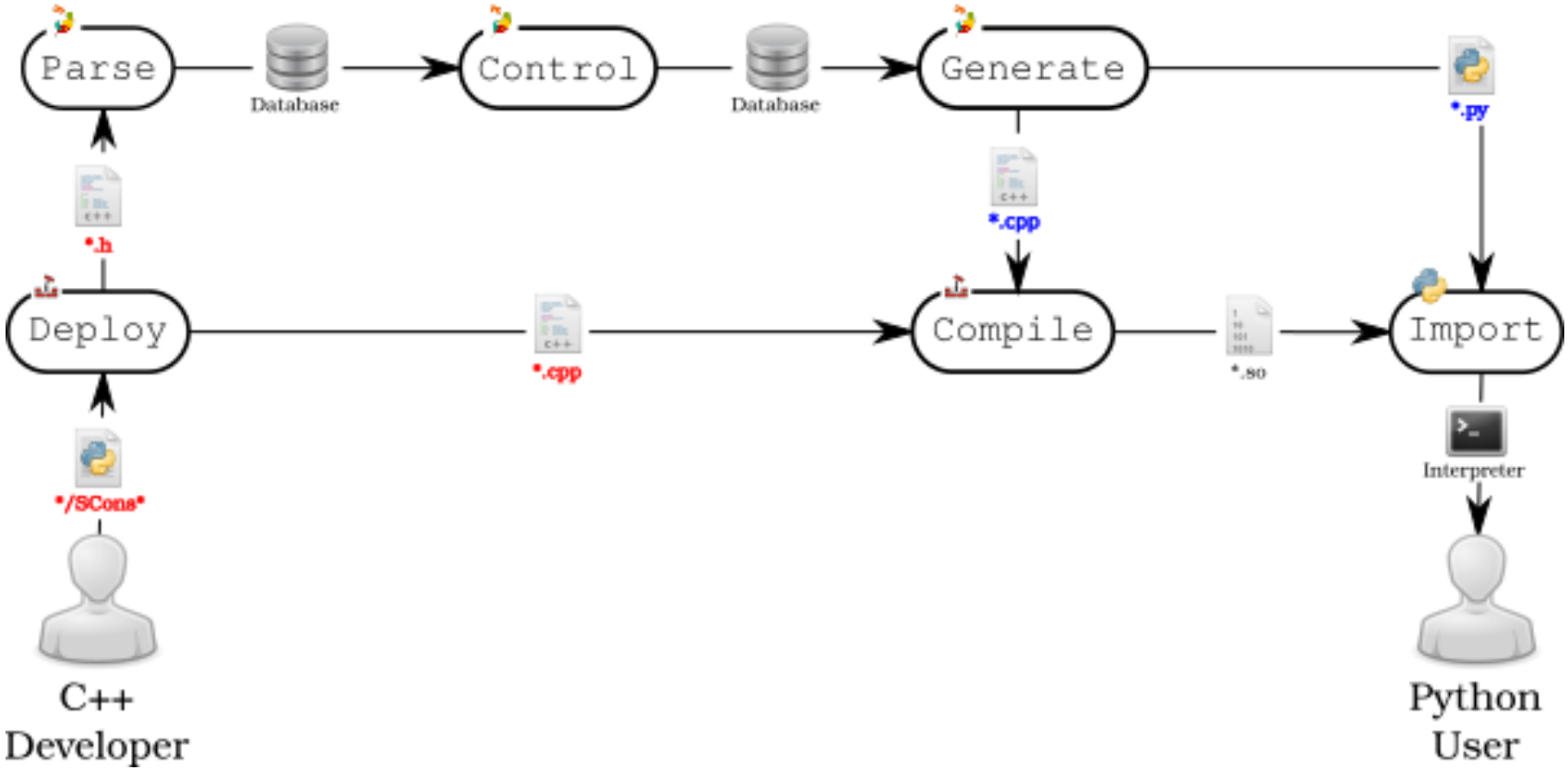
```
>>> from test.binomial import *
>>> binomial = BinomialDistribution(1,.5)
>>> binomial.pmf(0)
0.5
>>> binomial.pmf(1)
0.5
>>> binomial.n = 0
>>> binomial.pmf(0)
1.0
>>> binomial.set_pi(1.1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ProbabilityError: a probability must be in the interval [0,1]
>>> help(binomial.set_pi)
:Parameter:
    `pi` (:cpp:any:`double`) - New probability value

:Return Type:
    :cpp:any:`void`

:Raises:
```

Methodology

¬ SCons workflow



Deployment workflow using **SCons** *and* **AutoWIG** *for generating* `Python` *bindings for C++ libraries.*

Conclusion

Automatic generation of `Python` bindings for `C++` libraries is tested:

STL	36 classes, 203 functions from 31,650 SLOC.
StructureAnalysis	166 classes, 1,995 functions from 176,049 SLOC.
CLang	142 classes, 2,152 functions from 412,362 SLOC.

Conclusion

Automatic generation of `Python` bindings for `C++` libraries is tested:

STL 36 classes, 203 functions from 31,650 SLOC.

StructureAnalysis 166 classes, 1,995 functions from 176,049 SLOC.

CLang 142 classes, 2,152 functions from 412,362 SLOC.

Automatic generation of `R` bindings for `C++` libraries is in development:

- Memory management,
- Enumerations,
- `R` interface.

Conclusion

Automatic generation of `Python` bindings for `C++` libraries is tested:

STL 36 classes, 203 functions from 31,650 SLOC.

StructureAnalysis 166 classes, 1,995 functions from 176,049 SLOC.

CLang 142 classes, 2,152 functions from 412,362 SLOC.

Automatic generation of `R` bindings for `C++` libraries is in development:

- Memory management,
- Enumerations,
- `R` interface.

Differs from others (**SWIG**, **Cython**, **Py++**):

- Access to all steps.
- Allow code introspection.
- Based on a plugin architecture.

See you on:

- `https://github.com/StatisKit/AutoWIG` (code)
- `https://github.com/StatisKit/FP17` (examples)