ÉCOLE POLYTECHNIQUE DE L'UNIVERSITÉ DE NANTES
DÉPARTEMENT D'INFORMATIQUE

RAPPORT DE RECHERCHE ET DÉVELOPPEMENT

Intégration d'un interprète Python pour DGtal

Florent GUILLEMOT & Gwenn MEYNIER

30 novembre 2014

encadré par Nicolas NORMAND

coordinateur : Jean-Pierre GUÉDON

POLYTECH NANTES

UNIVERSITÉ DE NANTES

Intégration d'un interprète Python pour DGtal

Florent GUILLEMOT & Gwenn MEYNIER

Résumé

Ce rapport présente le projet de recherche et développement dont le but était de mettre en place un interpréteur Python pour la bibliothèque de géométrie discrète DGtal. Pour cela, nous avons étudié les différentes solutions d'encapsulation du C++, notamment SWIG et Boost.Python. L'une des contraintes est de ne pas ajouter de code à maintenir au sein de la bibliothèque DGtal. Nous avons donc opté pour l'amélioration d'un générateur d'interface pour SWIG. En effet, le support des modèles n'est pas optimal, et DGTal fait une utilisation importante de ce concept. Nous avons proposé d'utiliser le compilateur Clang afin de pouvoir extraire les informations caractérisant les modèles et pouvoir générer l'interface qui permettra leur utilisation avec Python. Durant la phase de développement, nous avons réalisé une interface SWIG pour quelques classes de DGtal, et avons remarqué que le projet comporte des éléments qui ne sont pas compatibles avec le fonctionnement de SWIG, et certaines modifications ont été nécessaires. Un script d'analyse du code de DGtal a également été produit, générant des interfaces pouvant êtres utilisées par SWIG, mais ne gérant pas la création de modèles C++.

Remerciements

Nous remercions Nicolas Normand pour son accompagnement tout au long du projet, ainsi que Jean-Pierre Guédon, coordinateur du projet.

Table des matières

1

Introduction

Cette section présente le projet d'intégration d'un interpréteur Python pour DGtal. DGtal étant développé en C++, la recherche a été portée sur les interfaces entre ces deux langages.

1.1 Intégration d'un interpréteur Python pour DGtal

DGtal est une bibliothèque de géométrie algorithmique développée en C++ à l'aide de la bibliothèque Boost. Afin d'utiliser les éléments de DGtal, les développeurs doivent développer des modules C++ qu'ils vont compiler et exécuter. Cette solution présente plusieurs problèmes de simplicité d'utilisation. De plus, Python devient un langage de plus en plus utilisé dans le monde scientifique [Lan07]. L'utilisation de Python comme langage de script afin d'utiliser les éléments de DGtal, de façon dynamique ou non, permettrait de résoudre certains de ces problèmes. Et cela permettrait aussi de démocratiser l'usage de DGtal dans le monde scientifique.

1.2 Objectifs poursuivis

Notre objectif va être de rendre tous les objets de DGtal utilisables facilement dans Python. Il faudra également que notre système soit résistant aux mises à jours du code de DGtal afin de ne pas produire de code supplémentaire à maintenir pour les personnes travaillant sur ce projet. L'objectif est donc d'étudier les différentes méthodes existantes afin de mettre en place un tel système, et de s'assurer qu'elles remplissent les fonctions nécessaires.

1.3 Travail réalisé

Nous avons porté nos recherches vers les outils existants en terme d'encapsulation de code C++ pour le langage Python. Les deux principales solutions sont SWIG et Boost.Python. d'autres solutions tels que CTypes ou Clang ont également été explorées.

Le problème principal rencontré est l'utilisation des

modèles C++. En effet, DGtal utilise de manière importante les modèles, qui sont des structures n'ayant pas d'existence si elle ne sont pas instanciées, et qui génèrent des classes à la compilation du projet. Ce système ne pose pas de problème lors de l'utilisation de la bibliothèque en C++ directement, mais pose problème pour notre projet.

Une fois choisi l'utilisation de SWIG, combinés au compilateur Clang afin de générer les interfaces SWIG automatiquement, nous avons développé un script d'analyse du code de DGtal et manuellement développé des interfaces SWIG pour encapsuler certaines parties du projet.

1.4 Contribution

Le résultat de ce projet est un script d'analyse du code de DGtal, générant des interfaces pouvant etre utilisées par SWIG afin de générer du code d'interface pouvant etre compilé en bibliothèque utilisable par python. Ce script ne gère pas la génération de modèles C++, et ne couvre donc pas ce type de structure. Nous avons également généré des interfaces manuellement, pouvant etre utilisées afin de manipuler des classes issus de modèles en python. Certaines modifications ont également dues etre effectuées sur le code de DGtal, afin de le rendre compatible avec l'analyseur de SWIG. Ces modifications n'alterent pas le fonctionnement initial de DGtal.

1.5 Plan de l'étude

Tout d'abord, dans la première phase du projet, nous avons étudié l'architecture globale de DGtal, puis nous nous sommes renseigné sur les différentes techniques existante pour interfacer du C++ avec du Python. Enfin, nous avons assemblé ces connaissances pour formuler des propositions innovantes pour résoudre le problème. Dans la deuxième phase du projet, nous avons développé un script d'analyse du code de DGtal générant des fichiers d'interface pour SWIG, et en parallèle, nous avons créé des interfaces fonctionelles pour une partie des classes de DGtal.

2

Présentation de DGtal

Dans cette partie, nous allons vous présenter DGtal dans sa version 0.8. Le site officiel du projet est situé à l'adresse suivante : http://dgtal.org/.

DGtal est à la fois un outil et une bibliothèque d'algorithme de géométrie discrète. Elle prend la forme d'une bibliothèque C++ et d'un ensemble d'outils appelés DGtalTools.

Le projet réunit un nombre important de structures de données et d'algorithmes, dans le but d'aider les chercheurs et étudiants à appréhender plus facilement la géométrie numérique et de pouvoir tester, implémenter et diffuser des algorithmes ou d'autres idées.

2.1 Structure du projet

DGtal utilise comme structure de base le *concept*, qui est un ensemble de types reliés à des interfaces. Ces concepts sont situés dans divers *packages*, en fonction de leur domaine. L'architecture de la version 0.8 de DGtal est la suivante :

Base
: comporte plusieurs concepts de base et classes utilitaires pouvant être utilisées dans les autres paquets.

Kernel
: rassemble des concepts de base susceptibles d'être utilisés par la plupart des éléments de la géométrie discrète.

Arithmetic
: contient, comme son nom l'indique, des algorithmes et concepts arithmétiques. Il comporte des structures de fractions ou des algorithmes de manipulation d'entiers.

Geometry
: couvre un large champ de concepts et est séparé en plusieurs modules distincts. Sont inclus les modules de courbes, surfaces, volumes et un ensemble d'outils plus généraux.

Shape
: définit des concepts de formes en deux ou trois dimensions, dont certaines sont pré-paramétrés, tels

que les cercles ou les arcs.

Topology
contient des algorithmes et concepts liés à la topologie, utilisés dans d'autres paquets.

DEC
(*Discrete Exterior Calculus*) permet certaines manipulations sur des opérateurs linéaires et des vecteurs.

Graph
fournit des concepts et classes autour de l'utilisation de graphes.

Mathematical
contient divers concepts mathématiques tels que l'analyse de polynômes ou de statistiques.

Image
Différents algorithmes et méthodes sur les images.

IO
contient diverses classes utilitaires qui permettent de manipuler les fichiers et d'effectuer un rendu à l'utilisateur.

2.2 Exemple d'utilisation de DGtal

Nous allons prendre pour exemple un code utilisant la bibliothèque DGtal, ainsi que le code des classes utilisées.

Nous prendrons un exemple simple : l'algorithme d'approximation de fraction. Cet algorithme recherche la plus proche fraction pour un nombre décimal donné. Il utilise la classe Fraction, qui est elle-même une classe interne de LighterSternBrocot, contenu dans le paquet arithmétique.

Voici l'algorithme utilisé dans cet exemple :

Listing 2.1 – Code de l'algorithme d'approximation

```
1
2  long double epsilon = 1e-14;
3  long double number0 = strtold( argv[ 1 ], 0 );
4  long double number = number0;
5  ASSERT( number >= 0.0 );
6  Fraction f;
7  OutputIterator itback = std::back_inserter( f
       );
8  Quotient i = 0;
9  while ( true )
10   {
11     long double int_part = floorl( number );
12     Quotient u = NumberTraits<long double>::
           castToInt64_t( int_part );
13     *itback++ = std::make_pair( u, i++ );
14     long double approx =
15       ( (long double) NumberTraits<Integer>::
             castToDouble( f.p() ) )
16       / ( (long double) NumberTraits<Integer
             >::castToDouble( f.q() ) );
17     std::cout << "z = " << f.p() << " / " << f
           .q()
18               << " =~ " << std::setprecision(
                     16 ) << approx << std::endl;
19     number -= int_part;
20     if ( ( (number0 - epsilon ) < approx )
21          && ( approx < (number0 + epsilon ) )
                ) break;
22     number = 1.0 / number;
23   }
```

La classe DGtal utilisée ici est LighterSternBrocot<Integer, Quotient, StdMapRebinder> : :Fraction, re-

nommée ici en *Fraction*.

3

État de l'art

Il sera détaillé ici l'état de l'art, où nous vous présenterons des solutions d'interfaçage entre C++ et Python, qui ont déjà été étudié pour d'autres projets pouvant être comparés au notre. Les principales solutions trouvées utilisent les bibliothèques Boost.Python ou SWIG.

3.1 Utilisation de SWIG

Cette partie a été rédigée en utilisant la documentation officielle de SWIG [comc].

3.1.1 Présentation

SWIG (Simplified Wrapper and Interface Generator) est un logiciel de développement pour créer des interfaces sur les programmes C et C++. Originalement développé en 1995, SWIG a d'abord été utilisé par les scientifiques de la division physique théorique du laboratoire national de Los Alamos pour construire des interfaces utilisateur pour les codes de simulation qui s'exécutaient sur le super-ordinateur Connexion Machine 5 [1].

SWIG est maintenant un outil qui permet de transformer du code C++ ou C en une multitude d'autres langages comme Python, GoLang, Java, etc. Nous nous sommes intéressés à sa capacité de créer des modules à partir de C++ pour Python.

Tout d'abord, nous allons voir son fonctionnement à l'aide d'un exemple simple.

Listing 3.1 – Code C++ d'exemple

```
1 #include <iostream>
2
3 template <class T>
4 class HelloWorld
5 {
6 public:
7         inline
8         void sayHello() {
9                 std::cout << "Hello world" <<
                      std::endl;
```

1. http://swig.sourceforge.net/Doc3.0/SWIGDocumentation.html#Preface_nn2

```
10          }
11          inline
12          T multiply(T x,T y)
13          {
14                  return x*y;
15          }
16
17  };
```

Listing 3.2 – Interface SWIG

```
1  %module myexample
2  %{
3  #include "myexample.h"
4  %}
5  %include "myexample.h"
6  %template(IntHello) HelloWorld<int>;
```

Listing 3.3 – Code Python de test

```
1  from myexample import IntHello
2
3  test = IntHello()
4  test.sayHello()
5  print test.multiply(4,5)
```

Listing 3.4 – Commandes nécessaires avec un système Linux pour exécuter l'exemple

```
1  #!/bin/bash
2
3  # Compilation
4  swig3.0 -c++ -python myexample.i
5  g++ -O2 -fPIC -c myexample_wrap.cxx -I/usr/
       include/python2.7
6  g++ -shared myexample_wrap.o -o _myexample.so
7
8  # exécution de l'exemple
9  python test.py
```

Le fonctionnement de SWIG est tel que :

1. On écrit son code comme pour 3.1.
2. On écrit le fichier interface 3.2 qui va permettre à SWIG de comprendre le code C++.
3. On compile le code avec SWIG et g++ pour créer une extension Python ligne 4 à 6 ici 3.4.
4. On peut utiliser le code compilé à comme une extension Python 3.3.

Présentation du fonctionnement interne

Cette partie est basée sur la documentation de SWIG[comc].

Les sources de SWIG sont organisées de la façon suivante :

CParse contient les fonctions de parsing du code en C et en lex/yacc.

DOH gère l'allocation mémoire, l'accès aux fichiers et les conteneurs génériques.

Include contient un fichier de configuration et de définitions.

Modules contient les fichier cxx effectuant la génération de code pour les langages de script.

Preprocessor contient des méthodes de pré-processeur pour SWIG.

Swig contient le cœur de la bibliothèque.

SWIG contient son propre pré-processeur pour les fichiers d'interface. On trouve les directives classiques du C tels que les *includes* (qui sont protégés contre l'autoinclusion) et les macros, mais également des directives propres aux langages utilisés.

Afin d'effectuer des échanges de variables (via paramètre et retour de fonctions par exemple), SWIG effectue des conversions entres les types suivants ceux disponibles pour le langage cible. Il est possible de surcharger ces conversions grâce à la directive de pré-processeur SWIG *typemap*.

Le programme commence par analyser la ligne de commande afin de déterminer la langage qui va être utilisé pour englober le c++. Un objet adéquat va donc être créé et va analyser la partie de la ligne de commande spécifique au langage. Ensuite, Swig lance le pré-processeur et appelle la méthode parse() de l'objet de langage afin de générer les fichiers nécessaires à l'utilisation de la bibliothèque.

Swig utilise un système de types de base pour gérer les variables dynamiques pour les langages de scripts tels que perl ou Python. Ce système s'appelle DOH et à pour but d'être simple, sans hiérarchie compliquée de classe. Ces types sont organisés de telle sortes à ce qu'ils utilisent le typage et la gestion de mémoire dynamique (avec ramasse miettes). Les types compris sont :

String Chaîne de caractères avec différentes méthodes et gestion de fichiers.

List Liste d'objets DOH.

Hash dictionnaire d'associations clé-valeur.

File interface pour la structure File* de C.

Void interface pour un pointeur vide de C.

Afin d'englober du code C++, SWIG génère une surcouche écrite en C basique qui pourra être utilisée par le langage cible. Le code généré sera bien dissocié du code original afin de ne pas interférer avec.

3.1.2 Analyse

Intérêts de l'utilisation de SWIG

L'intérêt principal pour utiliser SWIG est qu'une fois que les interfaces ont été écrites, il est très facile de rendre le code compatible avec un autre langage.

Limites de l'utilisation de SWIG

La principale limite est qu'il faut écrire un fichier d'interface pour chaque classe du code de DGtal. Un autre inconvénient est qu'il faut spécifier le type de l'instance du modèle cf 3.2 ligne 6. Cette limitation peut être contournée en instanciant tous les types possibles de modèles. Cela entraîne des objets avec le nom "type+Nom de le la classe C++". Si l'on choisit cette méthode, il faudra établir une convention de nommage.

Certaines particularités de C++ ne sont pas gérées par SWIG (bien que cette liste se réduise de plus en plus). On peut notamment citer la surcharge d'opérateurs spécifique (new, delete...).

3.2 Utilisation de Boost.Python

3.2.1 Présentation

La bibliothèque Boost.Python permet de compiler une bibliothèque. Celle ci sera ensuite utilisable par un interpréteur Python comme un module. Pour réaliser cela, il faut utiliser le système suivant :

Listing 3.5 – Code C++ d'exemple

```
1  #include <iostream>
2
3  template <class T>
4  class HelloWorld
5  {
6  public:
7          inline
8          void sayHello() {
9                  std::cout << "Hello world" <<
                       std::endl;
10         }
11         inline
12         T multiply(T x,T y)
13         {
14                 return x*y;
15         }
16 };
```

Listing 3.6 – Interface Boost.Python

```
1  #include <boost/python.hpp>
2  #include "myexample.h"
3
4  BOOST_PYTHON_MODULE(myexample)
5  {
6          using namespace boost::python;
7          class_< HelloWorld< int > >( "IntHello
               " )
8                  .def(
9                          "multiply"
10                         , (int ( ::HelloWorld<
                               int>::* )( int,int
                                ) )( &::
                               HelloWorld< int
                               >::multiply )
11                         , ( arg("x"), arg("y")
                                ) )
12                 .def(
13                         "sayHello"
14                         , (void ( ::HelloWorld
                               <int>::* )(  ) )(
                               &::HelloWorld< int
                                >::sayHello ) );
15 }
```

Listing 3.7 – Commandes nécessaires avec un système Linux pour exécuter l'exemple

```
1  #!/bin/bash
2
3  # compilation de l'extension
4  g++ -g -shared -fPIC -I/usr/include/python2.7
       myexample.cpp -lpython2.7 -lboost_python -
       o myexample.so
5
6  # exécution du test
7  python test.py
```

Listing 3.8 – Code Python de test

```
1  from myexample import IntHello
2
3  test = IntHello()
4  test.sayHello()
5  print test.multiply(4,5)
```

Pour utiliser Boost.Python, il faut effectuer les manipulations suivantes :

1. Tout d'abord, il faut écrire le code source c++ comme l'exemple 3.5.

2. Ensuite, il faut écrire l'interface Boost.Python comme pour 3.6.

3. Enfin, il faut compiler l'extension avec la commande 3.7

Présentation du fonctionnement interne

Cette partie a été rédigée en utilisant la documentation de Boost.Python [Abr] et en analysant le code source. Lorsqu'une classe ou une structure est déclarée en C++, un nouvel objet python sera créé et ajouté dans un registre. Plusieurs paramètres peuvent être appliqués afin de copier tous les attributs de cette classe ou de créer la méthode __init__ (le constructeur). En cas de copie de valeur (argument ou retour de fonctions) des modèles C++ se chargent d'effectuer la conversion en fonction du sens et du type. Dans le cas de C++ vers python, un certain nombre de convertisseurs (pour les types de base, notamment) peuvent être utilisé, et, dans le cas échéant, un convertisseur "universel", copiant les attributs et méthodes sera utilisé. Un système similaire est effectué dans le sens inverse.

3.2.2 Analyse

Intérêts de l'utilisation de Boost.Python

La bibliothèque DGtal est basée sur Boost, ce qui est un avantage en terme de cohérence pour le projet. Il existe également un système de parsing de code, py++, qui permet de générer les interfaces automatiquement.

Limites de l'utilisation de Boost.Python

Pour créer l'extension Python, il faut coder une interface. Boost.Python présente la même limitation que SWIG, à savoir qu'il n'est pas possible instancier dynamiquement un modèle. En effet, la seule méthode possible est d'utiliser un alias de la classe ou de la fonction qui permet de sélectionner la bonne instance de modèle.

3.3 Utilisation de Py++

3.3.1 Présentation

L'outil Py++ permet de générer le code de l'interface Boost.Python

Listing 3.9 – Code C++ d'exemple

```
1 #include <iostream>
2
3 void hello_world() {
4         std::cout << "Hello world" << std::
              endl;
5 }
```

Listing 3.10 – Interface Boost.Python générée par Py++

```
#include "boost/python.hpp"
#include "myexample.h"

namespace bp = boost::python;

BOOST_PYTHON_MODULE(pyplusplus){
    { //::hello_world

        typedef void ( *
            hello_world_function_type )(  );

        bp::def(
            "hello_world"
            , hello_world_function_type( &::
                hello_world ) );

    }
}
```

Listing 3.11 – Commandes nécessaires avec un système Linux pour exécuter l'exemple en plus de l'utilisation de Py++

```
#!/bin/bash

# compilation de l'extension
g++ -g -shared -fPIC -I/usr/include/python2.7
    myexample.cpp -lpython2.7 -lboost_python -
    o pyplusplus.so

# exécution du test
python test.py
```

Listing 3.12 – Code Python de test

```
import pyplusplus

pyplusplus.hello_world()
```

Le fonctionnement de Py++ est similaire à l'utilisation de Boost.Python. L'étape pendant laquelle il est nécessaire d'écrire une interface est remplacé par l'utilisation de Py++ qui va générer automatiquement ce fichier. Py++ est disponible sous la forme d'une interface graphique web ou Qt. 3.1.

3.3.2 Analyse

Intérêts de l'utilisation de Py++

Il n'y a aucun code supplémentaire à écrire, il suffit de lancer l'outil. Cela permet de ne pas avoir de code à maintenir, car il suffira de relancer le générateur à chaque modification de structure du code.

Limites de l'utilisation de Py++

Il faut relancer Py++ à chaque modification pour actualiser le code modifié. Il ne détecte pas les modèles qui n'ont pas été instanciées. Ceci est due au fait que Py++ utilise GCCXML pour extraire la structure du code. Or il s'avère que le projet GCCXML n'a pas besoin de modèles non instanciées [2].

2. https://gccxml.github.io/HTML/FAQ.html

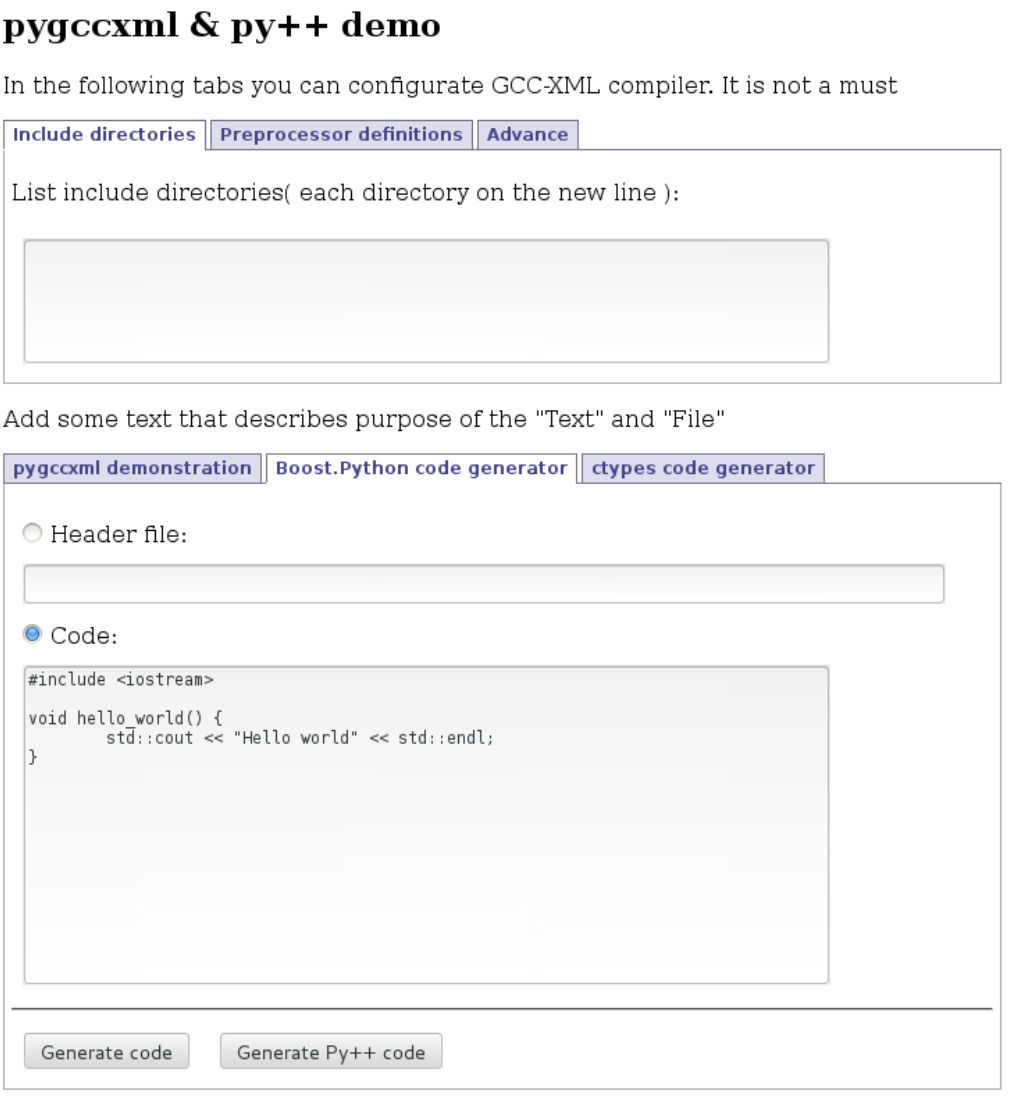


FIGURE 3.1 – Interface web de Py++

3.4 Utilisation des CTypes

Cette partie a été écrite en utilisant la documentation officielle des CTypes[comb].

3.4.1 Présentation

Ctypes est une bibliothèque de fonction étrangères pour Python. Cela permet d'avoir des types de données compatible avec le langage C et également d'appeler des fonctions présentes dans les DLLs ou bibliothèques partagées. Cette bibliothèque peut être utilisée comme interface entre ces bibliothèques et du code Python. Il est donc possible de charger directement une bibliothèque existante avec ce système.

L'article [Guy08] donne un exemple de génération automatique de code utilisant les CTypes avec les bibliothèques h2xml et xml2py. Malheureusement, nous n'avons pas pu reproduire ces résultats.

On allons donc vous présenter un exemple qui a été écrit manuellement :

Listing 3.13 – Exemple C++ utilisé

```
1 #include <iostream>
2
3 extern "C" void sayHello() {
4         std::cout << "Hello world" << std::
              endl;
5 }
```

Listing 3.14 – Commandes utilisées pour exécuter l'exemple

```
1 #!/bin/bash
2
3 # compilation de la librairie
4 g++ -g -Wall -shared -fPIC myexample.cpp -o
      myexample.so
5
6 # exécution du test
7 python test.py
```

Listing 3.15 – Test Python des Ctypes

```
1 import ctypes
2
3 testlib = ctypes.CDLL('./myexample.so')
4 testlib.sayHello()
```

À la ligne 3 de 3.13, on remarque qu'il y a l'ajout de *extern "C"*. Cet ajout est obligatoire pour de la compilateur n'ajoute pas de décorations sur les méthodes. Pour utiliser les CTypes, il faut tout d'abord compiler le code C++ sous la forme d'une extension. Puis, on peut utiliser l'extension comme 3.15. La variable *testlib* contient toutes les méthodes de l'extension et elles sont utilisables comme un objet.

3.4.2 Analyse

Intérêts de l'utilisation des CTypes

Il s'agit de la solution la plus rapide et la plus simple en terme de création et d'éxécution du code. En effet, il n'est pas nécessaire d'écrire une interface entre le code C++ et Python. À l'aide des CTypes, on peut exécuter directement les fonctions présentes dans une bibliothèque partagée.

Limites de l'utilisation des CTypes

Les CTypes ne supportent pas l'instanciation de modèles. De plus pour rendre utilisable les fonctions, il est nécessaire d'ajouter *extern "C"* sur le prototype des fonctions. Il n'est aussi pas possible d'avoir un support des namespaces ou des classes. En effet, toutes les fonctions doivent être accessible directement à partir de la bibliothèque. Les CTypes auraient pu être intéressant si DGtal avait été écrite en C car les CTypes ont été initialement pensés pour être utilisés en C.

3.5 Utilisation de l'arbre de syntaxe C++

Cette partie a été rédigée à l'aide de l'article [Sze14].

3.5.1 Présentation

L'arbre de syntaxe aussi appelé AST *Abstract Syntax Tree* permet d'obtenir la structure du code C++. Nous avons extrait cette structure avec le compilateur Clang, nous avons choisi ce compilateur car il représente également les modèles non instanciés. De plus Clang est un compilateur C/C++ qui ce veut plus personnalisable que les équivalents tel que GCC [coma]. Grâce à cet arbre, nous pouvons générer automatiquement les interfaces nécessaires à l'utilisation de SWIG ou Boost.Python.

Listing 3.16 – Code Python nécessaire à la génération de l'AST

```python
#!/usr/bin/python
# vim: set fileencoding=utf-8
import clang.cindex
import asciitree
import sys

def node_children(node):
    return (c for c in node.get_children() if c
        .location.file.name == sys.argv[1])

def print_node(node):
    text = node.spelling or node.displayname
    kind = str(node.kind)[str(node.kind).index(
        '.')+1:]
    return '{} {}'.format(kind, text)

if len(sys.argv) != 2:
    print("Usage: dump_ast.py [header file name
        ]")
    sys.exit()

clang.cindex.Config.set_library_file('/usr/lib
    /llvm-3.6/lib/libclang.so')
index = clang.cindex.Index.create()
translation_unit = index.parse(sys.argv[1], ['
    -x', 'c++', '-std=c++11', '-
    D__CODE_GENERATOR__'])
print(asciitree.draw_tree(translation_unit.
    cursor, node_children, print_node))
```

Listing 3.17 – Code d'exemple c++ avec une modèle

```cpp
#include <iostream>

template <class T>
class HelloWorld
{
public:
        inline
```

```
        void sayHello() {
                std::cout << "Hello world" <<
                    std::endl;
        }
        inline
        T multiply(T x,T y)
        {
                return x*y;
        }
};
```

Listing 3.18 – AST généré

```
TRANSLATION_UNIT ../boostpython/myexample.h
  +--CLASS_TEMPLATE HelloWorld
     +--TEMPLATE_TYPE_PARAMETER T
     +--CXX_ACCESS_SPEC_DECL
     +--CXX_METHOD sayHello
     |  +--COMPOUND_STMT
     |     +--CALL_EXPR operator<<
     |        +--CALL_EXPR operator<<
     |        |  +--DECL_REF_EXPR cout
     |        |  |  +--NAMESPACE_REF std
     |        |  +--UNEXPOSED_EXPR operator<<
     |        |  |  +--DECL_REF_EXPR operator
          <<
     |        |  +--UNEXPOSED_EXPR
     |        |     +--STRING_LITERAL "Hello
          world"
     |        +--UNEXPOSED_EXPR operator<<
     |        |  +--DECL_REF_EXPR operator<<
     |        +--UNEXPOSED_EXPR endl
     |           +--DECL_REF_EXPR endl
     |              +--NAMESPACE_REF std
     +--CXX_METHOD multiply
        +--TYPE_REF T
        +--PARM_DECL x
        |  +--TYPE_REF T
        +--PARM_DECL y
        |  +--TYPE_REF T
        +--COMPOUND_STMT
           +--RETURN_STMT
              +--BINARY_OPERATOR
                 +--DECL_REF_EXPR x
                 +--DECL_REF_EXPR y
```

Grâce au script 3.16, on peut générer l'arbre de syntaxe 3.18 de l'exemple 3.17.

Intérêts de l'utilisation de l'arbre de syntaxe C++

La combinaison de l'arbre syntaxique et du compilateur Clang, nous permettrais de créer un outil. Cet outil pourrait générer automatiquement le code d'interface nécessaire à l'utilisation de SWIG ou Boost.Python.

Limites de l'utilisation de l'arbre de syntaxe C++

L'utilisation de l'arbre de syntaxe C++ ne règle pas le problème des modèles. En effet, il faut toujours les instancier pour pouvoir les utiliser.

3.6 Récapitulatif

Nous avons préparé un tableau récapitulatif 3.1 qui contient les différents critères permettant de différencier les solutions d'interfaçage de C++ en Python. Nous avons fusionné Py++ et Boost.Python car il s'agit de la même solution à savoir Boost.Python. Nous n'avons pas inclus la solution qui parlait de l'utilisation de l'arbre syntaxique C++ car elle n'est pas comparable avec les autres.

	SWIG	Boost.Python	CTypes
Peut convertir en d'autres langages que Python	√		
Possède des générateurs d'interface	√	√	
Documentation complète sur tous les éléments du langage C++	√		
Exhaustivité de la documentation	Bonne	Légère pour les fonctions avancées	Légère
Support du C++	Quasi complet	Total	Aucun (support du C)
Support des modèles	Via instanciation	Via instanciation	Non

TABLE 3.1 – Tableau comparatif des avantages des interfaceurs de C++ en Python

3.7 Conclusion

À l'issue de ce travail de recherche bibliographique, il apparaît que plusieurs propositions peuvent servir de base à la résolution de notre problème.

Il semble se détacher un certain nombre de directions privilégiées que nous allons exploiter en priorité dans nos propositions. Celles-ci font l'objet de l'étude du chapitre suivant.

4

Proposition

A l'aide de notre état de l'art, nous avons pu formuler la proposition suivante :

4.1 Utiliser l'arbre syntaxique (AST) donné par le compilateur Clang pour générer les interfaces nécessaires à SWIG

Lors de notre état de l'art, nous avons identifié trois méthodes utilisables pour générer des modules Python : SWIG, Boost.Python et CTypes. Nous écartons d'emblée CTypes car son utilisation est orientée vers le langage C, et, bien qu'il soit possible de développer une sur-couche afin de prendre en compte les éléments du C++, cela serait trop coûteux en temps et peu intéressant par rapport à ce qu'offre les autres solutions.

SWIG et Boost.Python nécessitent l'écriture d'une interface pour faire le lien entre le code Python et le code C++. La principale différence entre SWIG et Boost.Python est que SWIG est plus à jour et permet de générer du code autre que Python une fois l'interface écrite. La documentation est également plus à jour et plus précise. Ce sont ces éléments qui ont motivé notre décision d'opter pour SWIG.

Étant donné que la solution proposée ne doit pas ajouter de code supplémentaire à gérer, nous allons mettre en place une solution d'automatisation de génération d'interface. Nous avons comparé différentes méthodes pour extraire la structure du code. Nous avons trouvé deux méthodes utilisables à savoir GCCXML et Clang. Nous avons sélectionné Clang car il extrait également la structure des modèles non instanciés.

Il reste à régler le problème des modèles. En effet, il faut connaître le type attribué à leurs arguments avant de les instancier, nous proposons d'intégrer les types les plus susceptibles d'êtres utilisés dans la documentation Doxygen de la classe. Ainsi, nous pourrons récupérer ces types afin de générer les classes correspondant aux modèles.

5

Expérimentations et résultats

5.1 Interfaçage de DGtal avec SWIG

La première étape dans la réalisation de notre proposition était de réaliser des premiers essais d'interfaçage grâce à SWIG. Nous avons donc sélectionné des classes de DGtal pour créer manuellement des interfaces SWIG et utiliser lesdites classes en Python.

5.1.1 Réalisation

Nous avons centré nos efforts sur plusieurs classes, parmi lesquelles les classes SpaceND (DGtal/kernel/SpaceND.h) et LighterSternBrocot (DGtal/arithmetic/LighterSternBrocot.h).

La première classe qui a été interfacée est la classe SpaceND, qui représente un espace de dimension N. Il s'agit d'un modèle C++, qui a été interfacé en fixant le paramètre qui correspond à la dimension de l'espace. Le nombre choisi a été de 3 car il correspond aux utilisations classiques de la classe. Nous avons sélectionné cette classe car elle possède peu de dépendances au noyau de DGtal.

La seconde classe LighterSternBrocot est une classe utilisé dans les exemples de DGtal (examples/arithmetic/*). L'idée était de pouvoir réaliser un de ces exemples.

5.1.2 Difficultés

Pour la classe LighterSternBrocot, nous avons rencontré plusieurs erreurs en essayant de gérer la classe. La première était une erreur liées aux macros Boost, qui a nécessité l'inclusion dans l'interface SWIG des fichiers entêtes des parties de Boost en question.

Une fois cette erreur réglée, une autre ligne a posé un problème que nous n'avons pas pu résoudre convenablement. En effet pour fonctionner, le modèle LighterSternBrocot demande 3 paramètres : TInteger, TQuotient et TMap. Les deux premiers sont des nombres tandis que le dernier est une classe de la même structure que StdMapRebinder 5.1. C'est là que le problème se pose.

Listing 5.1 – Implémentation de la classe StdMapRebinder

```
1  struct StdMapRebinder
2  {
3    template <typename Key, typename Value>
4    struct Rebinder {
5      typedef std::map<Key, Value> Type;
6    };
7  };
```

Comme on peut le voir dans le code, il s'agit d'une imbrication de structure. Ceci rend le code non interprétable par SWIG. Cette classe étant utilisée par LighterSternBrocot uniquement, nous avons donc proposé une modification de la structure de celle-ci, en nous basant sur des exemples tels que ceux décrits dans cette réponse de Stackoverflow [Ter]. Pour résoudre ce problème, nous avons donc réécrit la classe 5.2 en supprimant le paramètre TMap et en utilisant directement le modèle *std :map*.

Listing 5.2 – Modifications de la classe LighterSternBrocot

```
1  template <typename TInteger, typename
       TQuotient>
2  class LighterSternBrocot
3  {
4  public:
5    typedef TInteger Integer;
6    typedef TQuotient Quotient;
7    typedef LighterSternBrocot<TInteger,
         TQuotient> Self;
8
9  //    BOOST_CONCEPT_ASSERT(( concepts::
       CInteger< Integer > ));
10
11   struct Node;
12   typedef typename std::map<Quotient, Node*>
         MapQuotientToNode;
13
14 public:
```

Cette modification n'a pas eu beaucoup d'incidences sur DGtal car nous avons pu arriver à une version compilable de DGtal assez rapidement.

Enfin, une erreur est survenue lorsque nous avons essayé d'encapsuler la classe interne "Fraction", située dans la classe LighterSternBrocot. En effet, le code encapsulant généré par SWIG tente d'appeler le constructeur sans paramètre d'une des classes appelée par Fraction. Or, ce constructeur est protégé, ce qui génère une erreur à la compilation. Nous avons contacté la communauté de développement de SWIG sur Github à propos de ce problème mais notre message est resté sans réponse. Nous avons pu contourner le problème de manière temporaire afin de pouvoir utiliser ses classes pour la démonstration. Pour l'instant, le moyen de contournement utilisé est la mise en commentaire du système de vérification des paramètres et la récupération directe du retour de la méthode.

5.1.3 Résultats

Nous avons pu utiliser un modèle de DGtal dans Python. Il s'agit de LighterSternBrocot. Pour cela, nous avons utilisé l'interface suivante :

Listing 5.3 – Interface SWIG utilisé pour le modèle LighterSternBrocot

```
1  %module dgtal
2
```

```
3  %feature("flatnested");
4
5  %{
6
7  #include "/home/florent/projet/DGtal/src/DGtal
       /arithmetic/LighterSternBrocot.h"
8  using namespace DGtal;
9  %}
10
11 %include "/home/florent/projet/DGtal/src/DGtal
       /arithmetic/LighterSternBrocot.h"
12
13 %template (LightSternBrocot) DGtal::
       LighterSternBrocot<int,int>;
14 %rename (Fraction) LightSternBrocot::Fraction;
15 %rename (Node) LightSternBrocot::Node;
```

Pour le code 5.3, on remarque à la ligne 3 l'instruction *%feature("flatnested");*, elle permet d'indiquer à SWIG que les classes internes sont vues comme des structures plates. Cette instruction est nécessaire pour des langages comme Python puisque c'est son mode de fonctionnement mais pas pour C# et JAVA qui possèdent une gestion plus avancée des classes internes avec SWIG [comc]. Les lignes 14 et 15 permettent d'utiliser plus facilement les classes Fractions et Node mais ne sont pas nécessaires.

Listing 5.4 – Commandes nécessaires pour l'utilisation de LighterSternBrocot

```
1  #!/bin/bash
2
3  # Parametre
4  dgtalPath=/home/florent/projet/DGtal
5
6  # Suppression des fichier inutiles
7  # rm dgtal_wrap.cxx
8  rm *.so
9  rm *.o
10
11 # compilation
12 swig3.0 -c++ -python -Wall -D_SWIG dgtal.i
13 g++ -O2 -fPIC -c dgtal_wrap.cxx -I/usr/include
       /python2.7 -I$dgtalPath/src/ -I$dgtalPath/
       build/src/
14 g++ -O2 -fPIC -c $dgtalPath/src/DGtal/kernel/
       NumberTraits.cpp -I$dgtalPath/src/ -
       I$dgtalPath/build/src/
15 g++ -O2 -fPIC -c $dgtalPath/src/DGtal/
       arithmetic/LighterSternBrocot.cpp -
       I$dgtalPath/src/ -I$dgtalPath/build/src/
16 g++ -O2 -fPIC -c $dgtalPath/src/DGtal/base/
       Common.cpp -I$dgtalPath/src/ -I$dgtalPath/
       build/src/
17
18 # assemblage de la bibliothèque
19 g++ -shared Common.o LighterSternBrocot.o
       NumberTraits.o dgtal_wrap.o -o _dgtal.so
20
21 # exécution de l'exemple
22 python test.py
```

L'explication du code 5.4 est la suivante. À la ligne 12, SWIG va générer un fichier *dgtal_wrap.cxx* mais comme on l'a vu précédemment ce fichier est mal généré, il faudra donc corriger les erreurs avant d'exécuter les lignes suivantes. Ensuite de la ligne 13 à 16, les différentes parties de DGtal sont compilées. Cette partie a été difficile à trouver puisque les dépendances ne sont pas clairement indiqué. C'est aussi une erreur que nous avons faite au début ne pas tenir compte des dépendances à compiler. Et enfin à la ligne 19, la bibliothèque est assemblé. On peut ensuite s'en servir dans un script Python en effectuant un

import dgtal.

5.2 Script de génération d'interfaces SWIG

Afin d'automatiser le processus d'interfaçage du code C++ avec SWIG, et dans le but de rendre la bibliothèque Python résistante aux mises à jour de la bibliothèque, nous avons développé un script python utilisant une partie du compilateur Clang, afin d'analyser et parcourir le code de DGtal et de gérer les interfaces qui vont elles-mêmes générer le code d'interface.

Suite à une réunion avec M. Normand, il a été décidé plutôt que de détecter des paramètres des modèles spécifiés dans la documentation Doxygen, (qui n'est pas détecté par Clang) nous allions utiliser les fichiers d'exemples afin de générer les modèles appelés par ceux-ci. Cela permet d'avoir les mêmes éléments que si l'exemple était compilé directement avec g++.

5.2.1 Difficultés rencontrées

Nous avons tout d'abord remarqué que la documentation Doxygen n'apparaissait pas dans l'AST de Clang, et qu'il aurait donc fallu passer par un outil extérieur. Il est possible de détecter les commentaires, mais uniquement en parcourant les *tokens* du code (simple séparation du code selon les espaces et retours chariots) ce qui n'est pas intéressant pour l'utilisation qui nous concerne.

Le choix d'utiliser des exemples amenait que l'on pouvait repérer les modèles créés, mais nous avons remarqué que les types des paramètres n'apparaissent pas. Il faut pour cela utiliser un paramètre spécifique, nécessitant la recompilation de Clang et de Cmake, afin d'avoir accès aux modules instanciés par le code. Les limitations techniques de nos outils, ainsi que les limites de temps ne nous ont pas permis d'effectuer ces recompilations, ce qui ne nous a pas permis d'avoir accès a ces modèles, et par là, de générer le script nécessaire à l'automatisation de la génération d'interface.

5.2.2 Résultats

Au final, nous avons un script incomplet par rapport au but original, mais qui répond aux fonctionnalités de base, c'est à dire la génération de fichiers d'interface pour SWIG. Ce script pourra être complété à l'avenir afin de prendre en compte ces paramètres et permettre ainsi la génération automatique des interfaces SWIG pour l'ensemble de DGtal.

5.3 Conclusion

Au terme de la période de développement, nous avons réussi à interfacer une partie du code de DGtal est développé un script d'analyse du code générant des interfaces SWIG. Ce script est à compléter, et le code de DGtal doit continuer à être analysé et interfacé pour s'assurer que la compilation s'effectuerait sans problèmes. Une intégration aux fichiers cmake du projet est envisageable et faciliterait la génération d'interface et la compilation de la bibliothèque python.

6

Conclusion

Nous avons exploré dans ce document les différentes solutions pour utiliser une bibliothèque développée en C++ avec le langage Python. SWIG et Boost.Python permettent de mettre en place ce genre de solution mais impliquent un développement supplémentaire et ne gèrent pas les modèles de façon optimale.

Nous avons proposé une solution en combinant la puissance de SWIG et du compilateur Clang.

A cause d'une difficulté que nous avons sous-estimé, le projet n'a pas pu être complété comme prévu. Nous avons toutefois fourni une piste qui pourrait être reprise pour effectuer cette interface entre python et DGtal, et fourni un rapport qui pourrait être utile dans le cas ou l'on voudrait utiliser SWIG pour d'autres projets conséquents en C++. Il est compliqué d'interfacer un projet déjà établi utilisant des bibliothèques externes et des concepts C++ non triviaux tels que les modèles ou les structures internes complexes. En nous intéressant à 'utilisation de SWIG dans d'autres projets nous avons remarqué le cas de GNURadio, qui a inclus SWIG dés le début de son développement, utilisant python, SWIG et la bibliothèque C++ de manière continue permettant d'intégrer de manière efficace et de fournir un code non problématique pour l'interfaçage avec SWIG.

6.1 Enseignements

Nous avons tiré des enseignements au niveau du déroulement d'un projet de recherche et développement, pour la partie d'étude bibliographique.

Nous connaissions déjà une partie des technologies utilisés, mais pas sous cet angle. Ce projet nous a donc fait monter en compétence sur les interactions entre les technologies de programmation en général et de façon plus précise sur C++ et Python.

Nous avons également appris de l'importance de l'estimation de la complexité d'une tâche, étant donné la difficulté d'intégration de SWIG au projet DGtal, dont nous avons sous-estimé l'importance.

Bibliographie

[Cot03] T.L. COTTOM. “Using SWIG to bind C++ to Python”. In : *Computing in Science Engineering* 5.2 (mar. 2003), p. 88–97. ISSN : 1521-9615. DOI : 10.1109/MCISE.2003.1182968.

[Gen+04] Jacek GENEROWICZ et al. “Reflection-Based Python-C++ Bindings”. In : (oct. 2004).

[Mae+04] T. MAENO et al. “A python extension to the ATLAS online software for the thin gap chamber trigger system”. In : *Nuclear Science, IEEE Transactions on* 51.3 (juin 2004), p. 576–577. ISSN : 0018-9499. DOI : 10.1109/TNS.2004.828505.

[WCL04] S.N. WUTH, R. COETZEE et S.P. LEVITT. “Creating a Python GUI for a C++ image processing library”. In : *AFRICON, 2004. 7th AFRICON Conference in Africa*. T. 2. Sept. 2004, 1203–1206 Vol.2. DOI : 10.1109/AFRICON.2004.1406880.

[Lan07] Hans Petter LANGTANGEN. *Python Scripting for Computational Science*. Simula Research Laboratory et Department of Informatics University of Oslo, 2007.

[Guy08] K. Kloss GUY. “Automatic C Library Wrapping Ctypes from the Trenches”. In : *Open Journal System* 3.3 (2008).

[Sze14] Tamás SZELEI. “Implementing a code generator with libclang”. In : (fév. 2014). DOI : 10.5708/szelei.2014.A.1.

[Abr] Dave ABRAHAMS. *Boost.Python internals*. https://wiki.python.org/moin/boost.python/PeekUnderTheHoodl.

[coma] LLVM COMMUNITY. *Clang description*. http://clang.llvm.org/index.html.

[comb] Python COMMUNITY. *A foreign function library for Python*. https://docs.python.org/2/library/ctypes.html.

[comc] SWIG COMMUNITY. *SWIG Users Manual*. http://www.swig.org/Doc3.0/Contents.html.

[Ter] Evan TERAN. *What are some uses of template template parameters in C++ ?* https://stackoverflow.com/a/213811.

Table des figures

Liste des tableaux

Fiches de lecture

A.1 *Python scripting for computational science*

Ce livre [Lan07] explique les avantages de l'utilisation de Python pour faire du calcul scientifique. Nous nous sommes intéressé à la partie qui explique comment faire appel à des fonctions c++ dans Python.

A.1.1 Résumé

L'auteur nous apprend les techniques déjà existantes pour interfacer du C++ avec Python. Il aborde les techniques suivantes :

1. F2PY, Weave, Instant (avec un exemple)
2. SWIG (avec des exemples)
3. Boost.Python, SCXX, CXX (seulement évoquées)

A.1.2 Analyse

Nous sommes toujours au commencement de nos recherches, nous allons donc étudier en détail les différentes techniques abordées par l'auteur.

A.2 *Reflection-based Python-c++ bindings*

Cet article [Gen+04] traite des deux principales méthodes pour interfacer du c++ avec du Python.

A.2.1 Résumé

Les principaux problèmes pour interfacer du c++ avec du Python sont les suivants :

1. *Objets et conversion de paramètres* : les arguments et les valeurs de retour des fonctions doivent être passé d'un langage à l'autre.
2. *Types manquants ou incomplet* : la signature des fonctions et des classes peut inclure des types qui sont seulement déclarés et pas définis.

3. *Gestion de la mémoire* : Python possède un ramasse miette alors que C++ demande une gestion de la mémoire à la main.
4. *Redéfinition de fonctions c++* : Python n'a pas besoin de ce système étant donné qu'il est typé dynamiquement.
5. *Modèles C++* : Une convention de nommage est requise pour distinguer les modèles.

Il existe deux manière d'interfacer du C++ avec du Python :

1. *Static Wrapping* : Cette technique consiste à écrire du code entre le code Python et c++ pour déclarer la liste des méthodes utilisées avec leurs paramètres. L'inconvénient de cette méthode est qu'il est très long et fastidieux d'écrire ce code. Il existe déjà des bibliothèques qui implémentent cette technique. SWIG et Boost.Python sont les plus populaires.
2. *Dynamic Wrapping* : Cette technique consiste à s'appuyer sur un dictionnaire de type pour exécuter directement sur C++ dynamiquement à l'exécution. L'avantage est qu'il n'y a pas à écrire de code supplémentaire. Cette technique est implémenté par PyLCGDict et PyROOT.

A.2.2 Analyse

Cet article nous a permis d'avancer dans notre réflexion. Nous allons essayer de voir ce que donne les deux techniques (statique et dynamique en pratique).

A.3 *A Python Extension to the ATLAS Online Software for the Thin Gap Chamber Trigger System*

Il s'agit d'un article scientifique [Mae+04] qui utilise montre un exemple pratique de l'utilisation de Python comme passerelle entre applications.

A.3.1 Résumé

Dans cet article, les auteurs ont fait le choix de Boost.Python pour interfacer une application c++ avec Zope un cadriciel web Python. Cela leurs a permis d'avoir accès à ATLAS Online dans leur site internet.

A.3.2 Analyse

On pourrait penser que cet article n'a pas beaucoup d'intérêts. Mais, il faut voir que l'article date de 2004. Il faudra donc vérifier si l'on a encore des innovations à faire dans ce domaine.

A.4 *Using SWIG to bind C++ to Python*

Cet article [Cot03] est un manuel qui explique le fonctionnement de SWIG.

A.4.1 Résumé

Cet article explique en détail les différentes parties du langage C++ qu'il est possible de passer en Python avec

SWIG. SWIG est une passerelle de type statique. Pour utiliser SWIG, il faut écrire un fichier d'interface qui fait l'intermédiaire entre le code C++ et l'interpréteur Python.

Voici la liste des fonctions qu'il est possible de passer de C++ à l'interpréteur Python :

1. *Les fonctions simple*.
2. *La redéfinition de fonctions*.
3. *Les classes C++*.
4. *Les arguments par défaut*.
5. *L'ordre des arguments* : Python est plus flexible que C++, il est possible de choisir l'argument que l'on veut utiliser. En effet, s'il existe une méthode *action(int a=5,int b=8)*, il est possible de l'utiliser en avec uniquement l'argument b tel que *action(b=9)*.
6. *La redéfinition d'opérateur*.
7. *Les modèles C++*.
8. *Les exceptions*.

Il est également possible de d'étendre SWIG. Il est possible de créer des macros pour écrire des fichiers d'interface plus court.

A.4.2 Analyse

Cet article explique bien le fonctionnement de SWIG. Malheureusement, il n'est pas très à jour, il vaut mieux utiliser la documentation de SWIG qui est plus complète.

A.5 *Creating a Python GUI for a C++ Image Processing Library*

Cet article [WCL04] est un retour d'expérience sur l'élaboration d'une interface graphique en Python pour une bibliothèque en C++.

A.5.1 Résumé

L'objectif du travail des chercheurs aillant rédigé l'article était de créer une interface graphique utilisant Python et Tkinter pour la bibliothèque de manipulation d'images IPL98, écrite en C/C++. L'utilisation de Python pour créer l'interface graphique est due à sa simplicité et sa rapidité d'utilisation par rapport aux bibliothèques graphiques C++. Afin de gérer l'interfaçage entre la bibliothèque C++ et Python, l'équipe a utilisé SWIG, qui permet d'utiliser des fonctions C++ en rédigeant des interfaces spécifiques. Il leur est vite apparu qu'interfacer l'intégralité de la librairie IPL98 était très complexe sinon impossible, et ce pour plusieurs raisons :

1. L'interface SWIG, devant être générée manuellement, serait bien trop complexe à maintenir.
2. La disimilarité entre les langages (types de structure, modèles d'objets, etc.) rendrait la bibliothèque Python différente de celle d'origine, ce qui impliquerait de rédiger de la documentation supplémentaire pour la version Python.
3. La gestion de mémoire devient bien plus compliquée lorsque des objets et des pointeurs sont manipulés entre les deux langages.

La solution apportée est d'écrire des fonctions C++ réalisant des tâches spécifiques, et qui seront elles interfacées avec Python. Ainsi, si la totalité de la bibliothèque n'est pas utilisable, via Python, on peut porter des fonctionnalité individuellement par des fonctions C++. Parmi les avantages de cette solution, on peut citer la simplicité de mise en oeuvre, la disparition des problèmes de mémoire, et l'indépendance des différents modules. Cependant, cette solutions présente également des défauts : pas d'accès direct aux objets créé car si ils sont détruits après l'exécution de la fonction C++ appelée par Python, et donc, une plus grande complexité pour les fonctionnalités nécessitant une interaction utilisateur (sélectionner un pixel par exemple)

A.5.2 Analyse

Les besoins exprimés par l'équipe de recherche dans cet article sont similaire à ce de ce projet, car ils nécessitent une interface entre Python et C++.

A.6 Automatic C Library Wrapping - Ctypes from the trenches

cet article [Guy08] utilise un exemple de d'utilisation de wrapping Python avec le framework little CMS.

A.6.1 Résumé

L'auteur souligne l'importance des Ctypes pour effectuer la transformation de c++ à Python. Boost.Python est intéressant à utiliser si l'on veut utiliser une API plus complète en C++ qui reflète également la nature objet du code comme l'héritage. Cython est utile si l'on souhaite améliorer les performances de Python sur certaine partie du code. SWIG est intéressant si l'on souhaite utiliser le code original dans différents langages.

Il existe différents outils pour générer le code nécessaire à utilisation des englobeurs tel que Py++ pour Boost.Python et h2xml.py et xml2py.py pour les CTypes. L'auteur à choisit d'utiliser les CTypes pour 3 raisons :

1. L'ubiquité comme approche de liaison, étant donné que ctype fait partie de la distribution par défaut.
2. Il n'y a pas de compilation du code natif en bibliothèque de nécessaire. De plus ce système se base sur l'installation outil de développement et la bibliothèque d'englobement et vu comme une plateforme indépendante.
3. L'utilisation d'un générateur de code pour automatiser des grandes parties de l'englobement du code rend le système robuste contre les modifications.

À la fin de l'article, l'auteur explique qu'il lui a fallu 15 minutes pour mettre à jour son code lorsque qu'une nouvelle version de *little CMS* est sortie.

A.6.2 Analyse

Cette approche d'utiliser h2xml.py et xml2py.py semble intéressante car elle répond à notre besoin. Nous allons ajouter cette méthode à notre comparatif pour voir la différence avec les autres.

A.7 Implementing a code generator with libclang

Cet article [Sze14] montre qu'il est possible d'utiliser le compilateur clang pour générer le code intermédiaire pour Boost.Python.

A.7.1 Résumé

L'avantage d'utiliser clang pour générer le code intermédiaire est qu'il est facile de mettre à jour l'interface SWIG ou Boost.Python. Dans l'article, l'auteur se concentre sur une classe simple, à laquelle il va générer le code de l'interface avec LLVM. Cette technique est utilisé car le langage C++ manque de système d'introspection évolué. Normalement, pour cette génération, il faut que le projet suive des règles strictes lors de lors du codage mais libclang permet de s'affranchir de cette contrainte.

L'auteur explique ensuite que pour générer l'interface, il extrait l'arbre AST (abstract syntax tree). Cela permet d'avoir toutes les informations sur le code et d'appliquer une transformation aux noeuds. Sur chaque noeuds, qui va représenter une méthode, une classe, une fonction va être appliqué pour générer l'interface.

A.7.2 Analyse

Cette approche est intéressante et elle nous offre une alternative à Py++. On pourrait également utiliser cette méthode pour générer du code SWIG ce qui permettrait de rendre le code utilisable sur d'autres plateformes que Python.

B

Planification

La figure B.1 présente le planning prévisionnel élaboré au début du projet. Nous avons choisi une période de 4 semaines pour l'étude bibliographique car bien que nous ayons des connaissances en C++ et Python, une documentation était nécessaire afin de monter en compétence sur les différentes manières d'interfacer les deux langages. De plus, il faut ajouter la prise de connaissances sur la bibliothèque DGtal. Ensuite, nous avons mis 3 semaines pour l'analyse des solutions car nous anticipions d'en trouver un nombre important, la comparaison nous aurait donc pris du temps. Le planning effectif B.2 montre que nous avons du étendre notre phase de recherche bibliographique. En effet bien que nous n'ayons pas eu de problèmes pour trouver des solutions exploitables, celles-ci se sont avérées anciennes et/ou peu documentées. Cela nous a forcé à fournir un comparatif de solutions moins étendues mais plus récentes.
En ce qui concerne al partie développement, nous avons sous-estimés la charge de travail que représente l'interfaçage du code de DGtal avec SWIG. En effet, cette étape est bien moins triviale que ce que nous avions prévu en étudiant la bibliothèque de manière théorique, et nous a pris un temps important. De la même manière, la génération d"interface via l'analyse de l'arbre AST s'est avérée inefficace pour gérer l'analyse des fichiers d'exemples (qui a été décidée après la réalisation du planning prévisionnel) et a donc engendré des complications à ce niveau ? La réalisation des premières étapes s'étant révélée moins triviales que prévues, les tâches reposants dessus se sont vues abandonnées.

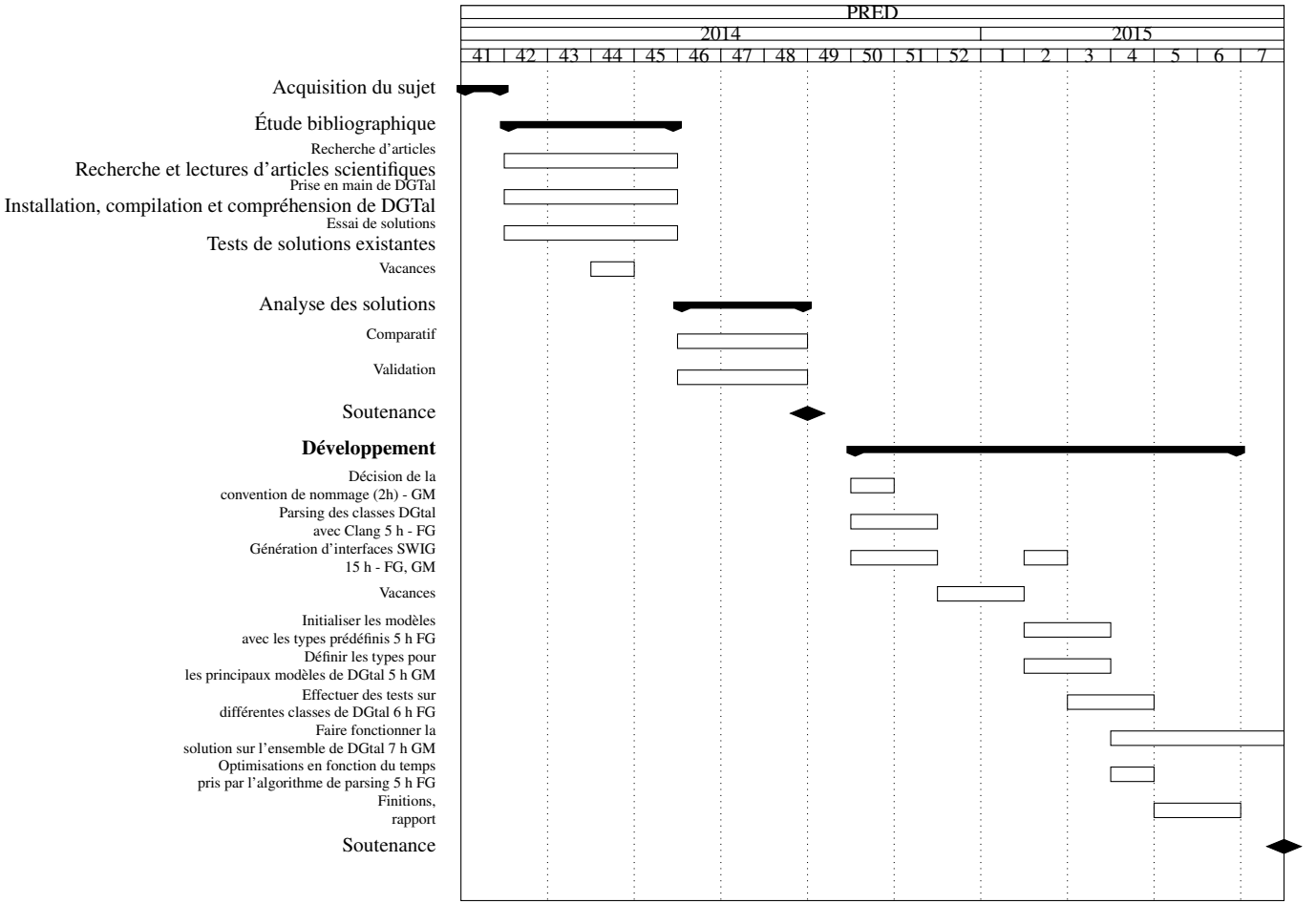


FIGURE B.1 – Planification prévisionnelle

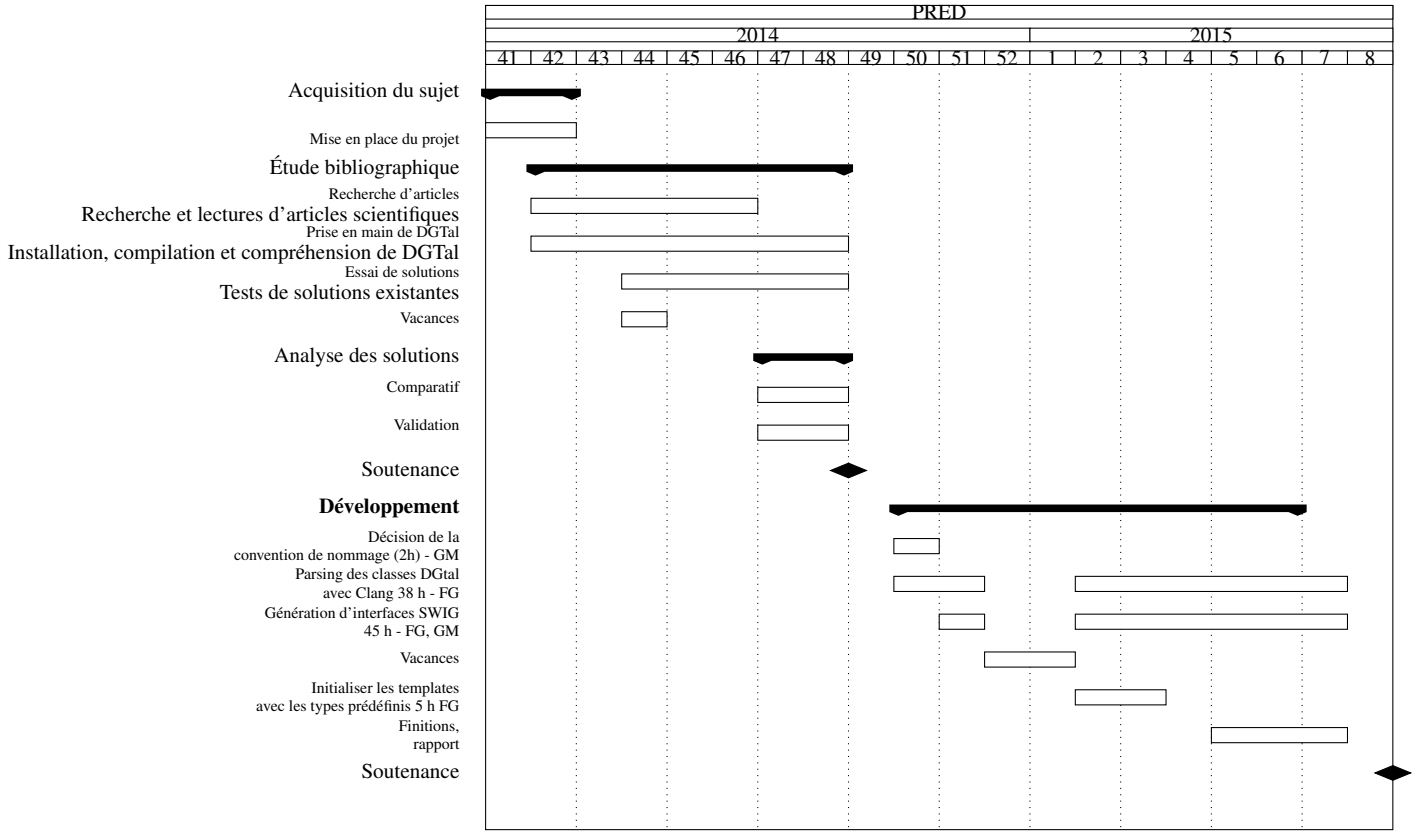
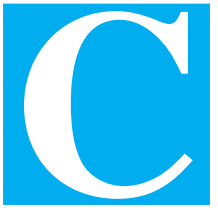


FIGURE B.2 – Planning effectif



Fiches de suivi

Fiche de suivi de la semaine 1
du 6 octobre 2014 au 12 octobre 2014

Temps de travail de Florent GUILLEMOT: 3 h 00 m
Temps de travail de Gwenn MEYNIER: 3 h 00 m
Travail effectué.

- Recherche bibliographique réalisée à 5% ;
- Compilation de DGtal fonctionne mais impossible d'utiliser la librairie ITK ;
- Test de certains exemple de DGtal ;
- Configuration de l'environnement de travail c9.io ;
- Rappel sur la syntaxe de Python ;
- Mise en place d'un git pour le stockage des rapports https ://github.com/CoreFloDev/Pred2014-DGtal.

Travail non effectué.

- Rendre le rapport hebdomadaire.

Échanges avec le commanditaire.

- Réunion de présentation avec Nicolas Normand.

Planification pour la semaine prochaine.

- Travail à avancer sur la bibliographie ;
- Mise en place du rapport final sur github en latex.

Fiche de suivi de la semaine 2
du 13 octobre 2014 au 19 octobre 2014

Temps de travail de Florent GUILLEMOT: 8 h 00 m
Temps de travail de Gwenn MEYNIER: 5 h 00 m
Travail effectué.

- Recherche bibliographique et écriture d'une fiche de lecture ;
- Mise en place du rapport de Pred en latex ;
- Test de différentes techniques de communication entre c++ et Python par l'utilisation de librairies.
- Début de l'analyse de DGtal ;
- Test des exemples de DGtal ;

— Mise en place d'un nouveau système de communication entre binome.

Travail non effectué.

Échanges avec le commanditaire.

— Bref échange de 5 min sur l'avancement.

Planification pour la semaine prochaine.

— Analyse de DGtal ;
— État de l'art des techniques de communication avec Python.

Fiche de suivi de la semaine 3
du 20 octobre 2014 au 26 octobre 2014

Temps de travail de Florent GUILLEMOT: 7 h 00 m
Temps de travail de Gwenn MEYNIER: 6 h 00 m

Travail effectué.

— Recherche et lecture de nouveaux articles scientifiques ;
— Étude d'un exemple de DGtal.

Travail non effectué.

Échanges avec le commanditaire.

— Réunion d'une heure.

Planification pour la semaine prochaine.

— Fiches de lectures pour les articles scientifiques trouvés ;
— Comparatif des différentes méthodes d'interfaçage entre C++ et Python ;
— Planning prévisionnel.

Fiche de suivi de la semaine 4
du 27 octobre 2014 au 2 novembre 2014

Temps de travail de Florent GUILLEMOT: 8 h 00 m
Temps de travail de Gwenn MEYNIER: 6 h 00 m

Travail effectué.

— réalisation des fiches de lecture des articles trouvés.

Travail non effectué.

Échanges avec le commanditaire.

Planification pour la semaine prochaine.

Fiche de suivi de la semaine 5
du 3 novembre 2014 au 9 novembre 2014

Temps de travail de Florent GUILLEMOT: 10 h 00 m
Temps de travail de Gwenn MEYNIER: 8 h 00 m

Travail effectué.

— comparatif des méthodes de création d'extension Python à partir de code C++ ;
— ajout de biblatex dans le document latex ;
— ajout de la coloration du code dans le rapport Latex ;
— réalisation du planning prévisionnel avec un gantt latex.

Travail non effectué.

Échanges avec le commanditaire.

Planification pour la semaine prochaine.

Fiche de suivi de la semaine 6
du 10 novembre 2014 au 16 novembre 2014

Temps de travail de Florent GUILLEMOT: 6 h 30 m
Temps de travail de Gwenn MEYNIER: 8 h 00 m

Travail effectué.

— écriture d'une nouvelle fiche de lecture ;
— comparatif des méthodes d'interfaçage entre C++ et Python complété à 70% ;
— difficultés à faire fonctionner h2xml et xml2py ;
— analyse de la structure de DGtal ;
— commencement de l'intégration d'une classe de DGtal avec boost.Python ;
— structure globale (introduction, conclusion, transitions) du rapport à 25%.

Travail non effectué.

Échanges avec le commanditaire.

— réunion de 1 heure le 14/11.

Planification pour la semaine prochaine.

— analyser le fonctionnement de CLANG pour générer du code d'interface utilisable par boost.Python ou SWIG ;
— faire une version présentable du rapport.

Fiche de suivi de la semaine 7
du 17 novembre 2014 au 23 novembre 2014

Temps de travail de Florent GUILLEMOT: 10 h 00 m
Temps de travail de Gwenn MEYNIER: 10 h 00 m

Travail effectué.

— Intégration d'un modèle à l'exemple de Boost.Python et SWIG ;
— tableau comparatif des méthodes d'interpolation avec Python ;
— lecture d'articles sur les modèles C++ et leur fonctionnement détaillé ;
— rédaction de la présentation de DGtal.

Travail non effectué.

Échanges avec le commanditaire.

Planification pour la semaine prochaine.

— recherches plus approfondies sur CLANG, les Ctype, l'API C Python et les RTTI ;
— fournir une version aboutie du rapport ;
— rédaction de bibliographie sur le fonctionnement interne de SWIG et Boost.Python.

Fiche de suivi de la semaine 8
du 24 novembre 2014 au 30 novembre 2014

Temps de travail de Florent GUILLEMOT: 13 h 00 m

Temps de travail de Gwenn MEYNIER: 13 h 00 m

Travail effectué.

— Rapport final ;
— Planning actualisé ;
— Ajout d'une partie sur les CTypes ;
— Ajout d'une partie sur CLang
— Ajout fiche auto évaluation ;
— Correction des fautes du rapport.

Travail non effectué.

Échanges avec le commanditaire.

— Envoie d'une version préliminaire du rapport le 28/11.

Planification pour la semaine prochaine.

— Diaporama de présentation.

Fiche de suivi de la semaine 9
du 1 décembre 2014 au 7 décembre 2014

Temps de travail de Florent GUILLEMOT: 10 h 00 m
Temps de travail de Gwenn MEYNIER: 10 h 00 m
Travail effectué.

— Préparation de la soutenance ;
— Passage de la soutenance ;
— Validation de la proposition avec le commanditaire.

Travail non effectué.

Échanges avec le commanditaire.

— Réunion de 20 minutes pour présenter la proposition.

Planification pour la semaine prochaine.

— Prototype fonctionnel ;
— Établissement de la convention de nommage pour les classes de DGtal ;
— Réunion avec le commanditaire.

Fiche de suivi de la semaine 10
du 8 décembre 2014 au 14 décembre 2014

Temps de travail de Florent GUILLEMOT: 8 h 00 m
Temps de travail de Gwenn MEYNIER: 8 h 00 m
Travail effectué.

— Début de l'exploitation des données de l'AST pour générer le code SWIG ;
— Recherche de solutions pour extraire des informations en commentaire avec Clang.

Travail non effectué.

— Réunion avec Nicolas Normand (indisponible).

Échanges avec le commanditaire.

Planification pour la semaine prochaine.

— Avancer sur l'exploitation de l'AST ;
— Établir une norme avec Nicolas Normand pour le nommage des templates en Python ;
— Extraire des informations en commentaire avec Clang.

Fiche de suivi de la semaine 11
du 15 décembre 2014 au 21 décembre 2014

Temps de travail de Florent GUILLEMOT: 8 h 00 m
Temps de travail de Gwenn MEYNIER: 8 h 00 m

Travail effectué.

— Avancées de l'exploitation des données de l'AST pour générer le code SWIG.

Travail non effectué.

Échanges avec le commanditaire.

— Réunion avec Nicolas Normand.

Planification pour la semaine prochaine.

— Continuer d'avancer sur l'exploitation de l'AST ;
— Utiliser les exemples pour générer les types d'instance de classe de DGtal.

Fiche de suivi de la semaine 12
du 5 janvier 2015 au 11 janvier 2015

Temps de travail de Florent GUILLEMOT: 4 h 00 m

Temps de travail de Gwenn MEYNIER: 6 h 00 m

Travail effectué.

— Reprise du code après la pause des vacances ;
— Rédaction du compte rendu de réunion ;
— Travail sur un exemple d'interface SWIG sur un template dgtal.

Travail non effectué.

— Prototype fonctionnel.

Échanges avec le commanditaire.

Planification pour la semaine prochaine.

— Avancer sur la création d'un prototype fonctionel.

Fiche de suivi de la semaine 13
du 12 janvier 2015 au 18 janvier 2015

Temps de travail de Florent GUILLEMOT: 8 h 00 m
Temps de travail de Gwenn MEYNIER: 8 h 00 m

Travail effectué.

- Avancement sur le développemnt d'un prototype fonctionnel ;
- Réalisation d'une interface swig pour une classe de dgtal ;
- Développement d'un script pour analyser le code de DGtal et des exemples ;
- Difficultés rencontrés afin d'interfacer le code d'une classe de DGtal.

Travail non effectué.

Échanges avec le commanditaire.

- Réunion planifié pour la semaine prochaine.

Planification pour la semaine prochaine.

- Présentation d'un prototype fonctionnel.

Fiche de suivi de la semaine 14
du 19 janvier 2015 au 25 janvier 2015

Temps de travail de Florent GUILLEMOT: 10 h 00 m
Temps de travail de Gwenn MEYNIER: 10 h 00 m

Travail effectué.

- Avancement dans la création du prototype ;
- Génération d'un premier fichier swig à partir d'un script python ;
- Compilation réussie d'une interface swig avec des sources DGtal.

Travail non effectué.

Échanges avec le commanditaire.

- Réunion avec Nicolas Normand le 20 janvier ;
- Réunion planifié pour la semaine prochaine.

Planification pour la semaine prochaine.

- Prototype fonctionnel.

Fiche de suivi de la semaine 15
du 26 janvier 2015 au 1 février 2015

Temps de travail de Florent GUILLEMOT: 24 h 00 m
Temps de travail de Gwenn MEYNIER: 32 h 00 m

Travail effectué.

- Recherche dans l'exportation des modèles des exemples de DGtal ;
- Compilation de certaines parties de DGtal avec SWIG.

Travail non effectué.

Échanges avec le commanditaire.

- Réunion avec Nicolas Normand le 28 janvier ;

— Réunion planifiée pour la semaine prochaine.

Planification pour la semaine prochaine.

— Prototype fonctionnel ;
— Rapport final ;
— Mise à jour du Gantt ;
— Documentation de l'installation des scripts.

Fiche de suivi de la semaine 16
du 2 février 2015 au 8 février 2015

Temps de travail de Florent GUILLEMOT: 25 h 00 m
Temps de travail de Gwenn MEYNIER: 18 h 00 m

Travail effectué.

— Compilation d'une bibliothèque composée de classes de DGtal utilisable en Python ;
— Rédaction du rapport de projet.

Travail non effectué.

Échanges avec le commanditaire.

Planification pour la semaine prochaine.

— Passage de la soutenance.

Le tableau C.1 récapitule le taux d'avancement du projet. Rappelons que le temps de travail théorique *minimal* correspond au temps indiqué sur la maquette pédagogique auquel on ajoute un strict minimum de 20 % correspondant au travail personnel hors emploi du temps. La partie « haute » de la fourchette correspond à 50 % de temps supplémentaire au titre du travail personnel.

	Temps prévu		Florent GUILLEMOT			Gwenn MEYNIER		
	bas	haut	hebdo.	Σ	%	hebdo.	Σ	%
Semaine	h : m	h : m	h : m	h : m		h : m	h : m	
1	10 : 00	12 : 30	3 : 00	3 : 00	30 (24)	3 : 00	3 : 00	30 (24)
2	20 : 00	25 : 00	8 : 00	11 : 00	55 (44)	5 : 00	8 : 00	40 (32)
3	30 : 00	37 : 30	7 : 00	18 : 00	60 (48)	6 : 00	14 : 00	46 (37)
4	40 : 00	50 : 00	8 : 00	26 : 00	65 (52)	6 : 00	20 : 00	50 (40)
5	50 : 00	62 : 30	10 : 00	36 : 00	72 (57)	8 : 00	28 : 00	56 (44)
6	60 : 00	75 : 00	6 : 30	42 : 30	70 (56)	8 : 00	36 : 00	60 (48)
7	70 : 00	87 : 30	10 : 00	52 : 30	75 (60)	10 : 00	46 : 00	65 (52)
8	80 : 00	100 : 00	13 : 00	65 : 30	81 (65)	13 : 00	59 : 00	73 (59)
9	90 : 00	112 : 30	10 : 00	75 : 30	83 (67)	10 : 00	69 : 00	76 (61)
10	100 : 00	125 : 00	8 : 00	83 : 30	83 (66)	8 : 00	77 : 00	77 (61)
11	110 : 00	137 : 30	8 : 00	91 : 30	83 (66)	8 : 00	85 : 00	77 (61)
12	120 : 00	150 : 00	4 : 00	95 : 30	79 (63)	6 : 00	91 : 00	75 (60)
13	130 : 00	162 : 30	8 : 00	103 : 30	79 (63)	8 : 00	99 : 00	76 (60)
14	140 : 00	175 : 00	10 : 00	113 : 30	81 (64)	10 : 00	109 : 00	77 (62)
15	150 : 00	187 : 30	24 : 00	137 : 30	91 (73)	32 : 00	141 : 00	94 (75)
16	160 : 00	200 : 00	25 : 00	162 : 30	101 (81)	18 : 00	159 : 00	99 (79)

TABLE C.1 – Avancement du projet par rapport au temps de travail théorique minimal (respectivement haut)

Auto-contrôle et auto-évaluation

Cette annexe est *obligatoire*.

La figure D.1 permet d'énumérer un certain nombre de points importants dans les trois composantes du travail :

1. rapport ;
2. présentation orale ;
3. travail de fond ;

ainsi que d'évaluer notre niveau de satisfaction à l'issue de la phase I, composée de trois étapes :

1. étude préalable ;
2. étude bibliographique ;
3. conception générale.

Les points de satisfaction ou d'insatisfaction peuvent être approfondis.

La figure D.2 permet d'énumérer un certain nombre de points importants dans les trois composantes du travail ainsi que d'évaluer notre niveau de satisfaction à l'issue de la phase II, constituée de :

1. la conception détaillée ;
2. la réalisation ;
3. la recette.

Grille d'évaluation du Projet de recherche (et développement)

Phase I : Etude préalable, étude bibliographique et conception générale

	Prénom	Nom	Année
Binome 1	Florent	Guillemot	2014-2015
Binome 2	Gwenn	Meynier	**Date**
			30/11/14
Jury 1	Nicolas	Normand	
Jury 2	Jean-Pierre		
Jury 3			
Jury 4			

Notation	
A	Maîtrise dans l'application du savoir-faire requis
B	Application du savoir-faire requis
C	Insuffisances, lacunes à corriger dans l'application du savoir-faire requis *a minima*
D	Insuffisances flagrantes, inacceptables, voire travail absent

				A	B	C	D	Remarque / Note / Commentaire
Rapport	Organisation	Plan	Equilibre		x			
			Cohérence		x			
		Fluidité	Introductions (partielles)			x		
			Transitions			x		
			Conclusions (partielles)		x			
		Tableaux, figures	Numérotés	x				
			Légendés	x				
			Référencés (non "en ligne")	x				
	Rédaction	Orthographe	Coquilles		x			
			Fautes évitables		x			
			Franglais, jargon	x				
		Rédaction	Aisée		x			
			Absence de plagiat !		x			
	Bibliographie	Références	Suffisantes (nombre, intérêt)	x				
			Pérennes	x				
			Complètes (auteurs, pages..)	x				
			Conséquentes (volume)		x			
		Références dans le texte		x				

Proposition de note haute	16,41
Proposition de note basse	11,41
Proposition de note du jury	**13**

Projection	Organisation	Plan	!!!	x				Nous n'avons pas encore fait le diaporama au moment du rendu.
		Liaisons	!!!		x			Il ne nous est donc pas possible de noter cette partie.
		Numérotation		x				
	Contenu	Informatif			x			
		Concis			x			
		Clair			x			
		Orthographe		x				
		Illustrations			x			
Oral	Présentation	Aisance			x			
		Tenue			x			
		Articulation, compréhension			x			
	Durée	Respect		x				
		Temps de parole équilibré		x				
	Réponses	Pertinence			x			
		Argumentation				x		

Proposition de note haute	15,19
Proposition de note basse	10,37
Proposition de note du jury	**13,5**

Travail	Etude	Bibliographie	Adéquate	x			
			Suffisante		x		
		Cahier des charges	Clair		x		
			Formalisé			x	
		Hypothèses envisagées	Nombre			x	
			Pertinence	x			
			Analyse *a priori*			x	
		Validation	Tableau comparatif		x		
			Choix argumenté(s)	x			
			Faisabilité		x		
	Complexité	Temps consacré			x		
		Résultats obtenus		x			
		Difficulté	Intrinsèque			x	
			Vis-à-vis du binôme	x			
	Annexes	Fiches d'avancement	Régularité		x		
			Détaillées			x	
		Gantt	Prévisionnel et justifications	x			
			Effectifs, erreurs, corrections	x			

Proposition note haute	15,61
Proposition note basse	10,61
Proposition de note du jury	**12**
PROPOSITION DE NOTE (I)	**12,83**

FIGURE D.1 – Points à contrôler à l'issue de la phase I

				A	B	C	D
Rapport	Organisation	Plan	Equilibre	x			
			Cohérence	x			
		Fluidité	Introductions (partielles)		x		
			Transitions		x		
			Conclusions (partielles)		x		
		Tableaux, figures	Numérotés	x			
			Légendés	x			
			Référencés (non "en ligne")	x			
	Rédaction	Orthographe	Coquilles	x			
			Fautes évitables		x		
			Franglais, jargon	x			
		Rédaction	Aisée		x		
			Absence de plagiat !	x			

Proposition de note haute	17,50
Proposition de note basse	12,50

Proposition de note du jury	**14**

			Nous n'avons pas effectué la soutenance à l'heure ou nous publions cette fiche d'évaluation					
Projection	Organisation	Plan		?		x		
		Liaisons		?		x		
		Numérotation		?	x			
	Contenu	Informatif		?			x	
		Concis		?		x		
		Clair		?	x			
		Orthographe		?		x		
		Illustrations		?	x			
Oral	Présentation	Aisance		?		x		
		Tenue		?		x		
		Articulation, compréhension		?		x		
	Durée	Respect		?	x			
		Temps de parole équilibré		?		x		
	Réponses	Pertinence		?		x		
		Argumentation		?		x		

Proposition de note haute	?
Proposition de note basse	?

Proposition de note du jury	**13,5**

Travail	Conception	Générale	Clarté		x		
			Niveau de détail adéquat	x			
			Argumentation	x			
			Validation		x		
		Détaillée	Formalisation adéquate (algorithm			x	
			Validation		x		
	Réalisation	Développement	"Fonctionnalités"		x		
			Volume, environnement...		x		
		Tests (unitaires, d'intégration, fonctionnel, de performance...)				x	
	Recette	Livraison du paquetage				x	
		Redéploiement du paquetage				x	
	Complexité	Temps consacré			x		
		Résultats obtenus			x		
		Difficulté	Intrinsèque	x			
			Vis-à-vis du binôme		x		
	Annexes	Fiches d'avancement	Régularité	x			
			Détaillées			x	
		Gantt	Prévisionnel et justifications	x			
			Effectif, erreurs, corrections		x		

Proposition de note haute	14,86
Proposition de note basse	9,86

Proposition de note du jury	**14**

PROPOSITION DE NOTE (II)	**13,83**

NOTE PROJET	**13,5**

FIGURE D.2 – Points à contrôler à l'issue de la phase II