Documentation sur le développement de DIYABC version 2 en python

Veyssier Julien

2 septembre 2011

Table des matières

1 Introduction **3**
1.1 Objectifs . . . 3
1.2 Choix d'outils . . . 3

2 Conception **3**
2.1 Documentation . . . 3
2.2 Partie graphique . . . 4
2.3 Coeur de l'application . . . 5
2.4 Le modèle historique . . . 6
2.5 Les données génétiques . . . 6
2.6 Les préférences . . . 6
2.7 La génération de la table de référence . . . 6
2.8 Les analyses . . . 7

3 Tests **7**
3.1 Conception . . . 8
3.2 Lancement . . . 8

4 Construction du projet **8**
4.1 Introduction . . . 8
4.2 Windows . . . 9
4.2.1 pyinstaller . . . 10
4.2.2 Inno setup compiler . . . 10
4.2.3 Py2exe . . . 10
4.2.4 Solution choisie . . . 11
4.3 Linux . . . 11
4.3.1 Installer PyQt4 . . . 11
4.3.2 pyinstaller . . . 12
4.3.3 Script de génération du paquet Debian . . . 12
4.4 MacOS 10.6 Snow Leopard . . . 13
4.4.1 Installer PyQt4 . . . 14
4.4.2 pyinstaller . . . 14
4.4.3 Solution choisie . . . 15

5 Améliorations à apporter **15**

1 Introduction

Ce projet est la refonte de DIYABC, initialement programmé en delphi, en PyQt.

1.1 Objectifs

Modulariser le code de sorte à permettre facilement l'ajout de nouveaux blocs, tant graphiques que fonctionnels.

1.2 Choix d'outils

La librairie Qt a été choisie pour sa portabilité, sa flexibilité et sa facilité d'utilisation. Le langage python a été choisi pour sa clarté, sa réutilisabilité et son grand nombre de librairies. QtCreator est utilisé pour la production des objets graphiques.

2 Conception

2.1 Documentation

L'intégralité du code est documenté façon javadoc. Ces docstrings sont donc interprêtables par beaucoup de générateurs de documentation. Je génère une documentation html du code avec Doxygen. Un fichier de configuration de doxygen nommé doxyconfig est versionné. Il se trouve à la racine des sources python, c'est à dire le dossier python_interface.

Une aide à l'utilisateur est intégrée à l'interface. Elle est construite à partir d'un fichier xml lui même généré à partir des sources LaTeXde la notice de DIYABC. La génération de cette documentation est décrite en 4.1 Pendant l'utilisation de l'interface, cette aide est consultable en effectuant un clic droit sur un label ou un bouton. L'identifiant d'un label ou d'un bouton est son "objectName" dans les sources python. Pour écrire le contenu qui sera affiché lors d'un clic droit sur ce label/bouton, il suffit d'insérer dans le source LaTeXde la notice un bloc de la forme :

```
\label{doc_objectName}
La description du bouton ou du label
```

La description s'arrête au prochain délimiteur de section comme un paragraphe, une section, une sous section ...

Voici la liste des labels pour lesquels il me semble utile de fournir une aide :

- versionLabel : numero de version dans la page d'accueil de l'interface.
- projNameLabelValue : nom du project dans l'écran principal d'un projet
- dataFileInfoLabel : informations sur le datafile dans l'écran principal d'un projet
- nbScLabel : nombre de scenarios
- nbHistParamLabel : nombre de paramètres historiques
- nbMicrosatLabel : nombre de microsatellites dans le datafile
- nbSequencesLabel : nombre de sequences dans le datafile
- nbGroupLabel : nombre de groupes de locus dans les données génétiques
- nbSumStatsLabel : nombre de summary stats demandées
- reftableProgressLabel : estmation du temps restant pour la generation de la table de reference
- mmrLabel : mean mutation rate
- ilmrLabel : individual locus mutation rate
- mcpLabel : mean coefficient P
- ilcpLabel : individual locus coefficient P
- msrLabel : mean SNI rate
- ilsrLabel : individual locus SNI rate
- mc1Label : mean coefficient k_c/t
- ilc1Label : individual locus k_c/t
- mc2Label : mean coefficient k_a/g
- ilc2Label : individual locus k_a/g

Et la liste des boutons :

- setHistoricalButton : aller vers l'écran de définition du modèle historique
- setGeneticButton : aller vers l'écran de définition des données génétiques
- runReftableButton : lancer la génération de la reftable en appelant l'exécutable de calcul
- stopReftableButton : arrêter l'exécutable de calcul
- newAnButton : définir une nouvelle analyse

2.2 Partie graphique

La partie graphique statique est partiellement découplée du fonctionnement. Des fichiers .ui sont produits à partir d'un IDE. Toute entité graphique étant succeptible d'être présente plusieurs fois dans l'application est produite indépendament dans un .ui et une classe associée.

L'ui principal est celui de la fenêtre principale. Un ui peut n'être qu'un widget. Un fichier source python produit à partir d'un .ui avec "pyuic" contient une classe non graphique qui crée tous les objets graphiques à l'intérieur de l'objet que l'on veut créer.

Exemple : Création d'une scroll area.

- Création d'un fichier .ui pour QScrollArea avec QtCreator par exemple
- Génération du code python avec pyuic4 :
  ```
  pyuic4 mon.ui > monui.py
  ```
- Instanciation d'un objet QScrollArea (on peut créer une classe qui dérive de QScrollArea ou bien instancier un vrai QScrollArea)
- Appel de la methode setupUi de la classe générée dans monui.py avec l'objet QScrollArea en paramètre
  ```
  from monui import Ui_QScrollArea
  maScrollArea = QScrollArea()
  ui = Ui_QScrollArea()
  ui.setupUi(maScrollArea)
  ```

2.3 Coeur de l'application

La classe principale de l'application dérive de QMainWindow. Elle est instanciée dans le main du fichier qui la contient. La QMainWindow contient une zone à onglet. Chaque onglet est un projet. Un projet est un ensemble de plusieurs informations :

- Un fichier de données
- Un modèle historique
- Des données génétiques
- Une table de référence
- Des analyses

Un projet peut être sauvegardé et chargé par la suite. La sauvegarde consiste en l'écriture de plusieurs fichiers de sauvegarde qui se trouvent dans le dossier du projet.

- conf.hist : modèle historique
- conf.gen : données génétiques
- conf.analysis : sérialisation des analyses
- conf.th : table header (dernière partie du reftable header)

– conf.tmp : entête du reftable header

Ces fichiers sont lu lors du chargement d'un projet pour rétablir son état. Pendant la manipulation d'un projet, un fichier de verrou (.lock) est créé dans le dossier du projet. Il permet de s'assurer que l'on ne manipule pas le même projet avec deux instances de DIYABC. Si l'interface a été arrêtée brutalement, le verrou n'est pas enlevé. Un message d'avertissement expliquant la situation est donc présenté au chargement du projet. On peut outrepasser le verrou si on est sur de ce que l'on fait.

2.4 Le modèle historique

La conception du modèle historique n'est pas très complexe. La gestion des scénarios, de conditions et des paramètres est similaire, du moins en ce qui concerne la GUI. Les méthodes addSc, addCondition et addParamGui ajoutent graphiquement ces entités et les mémorisent dans des attributs de la classe SetHistoricalModel .

2.5 Les données génétiques

La classe SetGenData contient un tableau de locus et une liste de groupes de locus. Pour chaque groupe, elle instancie deux modèles mutationnels et deux ensembles de summary statistics (microsats et sequences). Lorsque l'utilisateur veut définir le modèle mutationnel d'un groupe contenant des locus, le modèle mutationnel correspondant (microsats ou sequences), déjà instancié, est affiché.

2.6 Les préférences

Les préférences sont stockées dans le dossier personnel de l'utilisateur. Elles sont divisées en plusieures catégories chacune représentée par un fichier. Les valeurs sont stockées sous la forme clé → valeur sauf dans le cas des valeurs par défaut qui sont représentées comme dans le reftable header. Les préférences sont appliquées dès leur modification dans la fenêtre. Un clic sur annuler recharge les préférences à partir des fichiers, c'est à dire avant les modifications effectuées.

2.7 La génération de la table de référence

En local Elle s'effectue dans un QThread qui scrute l'avancement dans le fichier progress.txt . Pour stopper l'exécutable de calcul, il faut créer un ficher .stop dans le dossier du projet.

Un signal est prévu en cas de problème comme la fin prématurée de l'exécutable de calcul. La gui l'indique alors dans une popup d'information. Il en est de même pour le calcul des analyses.

Sur le cluster J'ai implémenté un fonctionnalité permettant à l'interface de lancer une génération sur un cluster qui marche avec SGE ou un système compatible. L'interface crée une archive tar, se connecte (par socket) à un serveur écrit en python qui se trouve sur un des noeuds du cluster. Elle transfere l'archive par cette connexion. Le serveur, une fois l'archive réceptionnée, extrait son contenu et exécute le script launch.sh qui s'occupe de lancer les qsub. Chaque job lance l'exécutable de calcul en arrière plan et transfère le fichier d'avancement local au noeud maitre à intervalle de temps régulier. Le serveur fait régulièrement la somme des avancements et la transmet à intervalle régulier à l'interface par le réseau afin qu'elle se mette à jour.

2.8 Les analyses

Nous n'avons pas jugé nécessaire de prévoir l'exécution des analyses sur le cluster. Il faudrait transférer la table de référence et le temps de calcul n'est souvent pas très long, le gain serait peut-être donc négatif.

L'écran principal de définition d'une analyse se trouve ans le fichier defineNewAnalysis.py . En fonction du type d'analyse sélectionné, l'écran suivant est instancié. Pour certains cas, comme l'écran de sélection de scénario, l'écran qui doit succéder est passé en paramètre au constructeur de l'écran. Pour les autres cas, la classe qui implémente l'écran instancie l'écran suivant directement en fonction de l'analyse dont il est question. Chaque écran ajoute les informations qui lui sont propres à l'analyse. Le dernier écran est chargé de donner l'analyse à la classe projet pour qu'elle l'ajoute à la GUI.

Le lancement des analyse est effectué dans un QThread qui scrute l'avancement dans le fichier prévu à cet effet. A chaque avancement, il emet un signal capté par le thread principal qui met à jour la GUI. Une fois l'analyse terminée, c'est la méthode qui capte le signal d'avancement qui appelle la méthode qui termine le thread d'analyse et qui range les résultats dans un dossier du nom de l'analyse.

3 Tests

Les tests sont tous dans le fichier unit_test.py . Ils utilisent le module python unittest.

3.1 Conception

Test général Les opérations de haut niveau sur un projet sont testées dans la méthode testGeneral . Attention, ce test nécessite l'existence d'un projet. Un projet existant étant versionné, ce test est opérationnel mais dans le cas où ce projet n'existe pas ou bien qu'il a été rendu incohérent par une modification à la main, on peut toujours changer les sources du test afin de définir correctement les chemins qui pointent statiquement sur un projet existant.

Test des clics La méthode testAllClicks simule une intéraction avec l'interface graphique. Un arbre des clics possibles est généré pour ensuite former une liste de toutes les séquences de clic possibles. L'arbre de clics permet de connaitre, pour un bouton donné, la liste des boutons candidats (les fils) pour le clic suivant. Il est evidemment possible de trouver des cycles dans les séquences générées. Un paramètre permet de fixer une borne maximum sur le nombre de fois que l'on clique sur un même bouton dans une séquence.

Ce test n'utilise pas d'assertion, il a pour but d'identifier une suite d'opérations provoquant l'arrêt prématuré de l'interface.

3.2 Lancement

Pour effectuer les tests, il suffit de lancer le script python unit_test.py qui contient une procédure principale. Il est conseillé de mettre la constante "debug" à True dans output.py pour éviter les popups d'information qui sont bloquants.

Le premier échec sur une assertion ou plantage provoque l'arrêt des tests.

A la fin des tests, si tout s'est bien passé, un résumé est affiché. L'erreur de segmentation à la toute fin des tests semble inévitable. Elle survient après tous les affichages et n'est donc pas gênante.

4 Construction du projet

4.1 Introduction

Pour construire le projet en vue de le distribuer, on dispose de trois scripts, un pour chaque système, qui vont générer un exécutable, une application ou un paquet. Cet objet généré est transportable et distribuable tel quel avec, pour Windows et MacOS, un dossier contenant les données.

Pour générer un exécutable sous MacOS et Windows, il faut :
– Se trouver sur le système en question
– Posséder python et toutes les librairies nécessaires au programme
– Disposer de pyinstaller

Pas de dépendance particulière pour générer un paquet sous Debian ou Ubuntu.

La génération de l'aide interne est très simple. Pour n'importe lequel de ces systèmes (Mac, Windows, GNU/Linux), il faut générer un xml à partir de la notice LaTeX. Cette opération s'effectue avec "latexml" présent dans les dépôts Debian et Ubuntu.

```
latexml notice.tex > documentation.xml
```

4.2 Windows

Sous windows, plusieurs solutions sont envisageables. La première, plus simple pour l'utilisateur et le packageur : PyInstaller. L'exécutable produit est indépendant et léger.

La deuxième, Inno setup compiler, pour créer un installateur qui effectue une copie de fichier et certaines actions. Deux stratégies sont possibles. On peut copier en un bloc python contenant ses dépendances et diyabc. Le problème est le suivant, python.exe refuse de se lancer s'il n'a pas été installé avec l'installeur officiel Python. Donc il faut tout de même déclencher l'installation de Python en incluant l'installeur dans le setup de diyabc. Sinon on peut déclencher l'installeur de Python, PyQt, Numpy, PyQwt pendant le setup de diyabc ce qui marche à coup sur mais est lourd pour l'utilisateur.

La troisème, py2exe est satisfaisante mais impose l'installation à part de certaines dll.

Pour ces trois solutions j'utilise python2.6 et les librairies "imposées" par PyQwt5 (http ://pyqwt.sourceforge.net/download.html). Il faut installer les versions suivantes des logiciels :
– python-2.6.2.msi
– numpy-1.3.0-win32-superpack-python2.6.exe
– PyQt-Py2.6-gpl-4.5.4-1.exe
– PyQwt5.2.0-Python2.6-PyQt4.5.4-NumPy1.3.0-1.exe

4.2.1 pyinstaller

Anciennes versions La version 1.4 nécessite une version de python <= à 2.5. L'utilisation de pyinstaller est assez simple. Il faut, lancer "python Configure.py" pour que pyinstaller se configure par rapport au système sur lequel il est. Ensuite il faut lancer "python Makespec.py –onefile path_to_python_main_source.py" qui génère le specfile dans un dossier du même nom que notre source python. Ensuite "python Build.py specfile.spec" construit l'exécutable qui se trouvera dans le même dossier que le spec file dans un dossier nommé "dist".

Solution fonctionnelle La version svn est de loin la meilleure solution. Elle marche sans aucun problème sous windows. Son utilisation est simplifiée. Il suffit d'appeler le script pyinstaller.py avec le programme cible en paramètre. Les trois étapes (configure, makespec et build) sont exécutées automatiquement si besoin. Dans le fichier .spec produit, on peut ajouter l'option icon pour personnaliser l'icone de l'exécutable. Utilisation :

```
python pyinstaller.py --onefile -w --icon=path\to\icon.ico main.py
```

4.2.2 Inno setup compiler

http ://www.jrsoftware.org/isdl.php

La création du fichier de config (.iss) est assistée et plutôt aisée. On donne les fichiers à inclure, le dossier de destination et quelques options. Pour plus de détails, consulter le fichier .iss inclut dans le dépot qui est commenté.

4.2.3 Py2exe

Le site de py2exe contient un tuto simple auquel il faut ajouter quelques détails pour que ça fonctionne parfaitement. Le fonctionnement est simple. On écrit un setup.py puis on exécute

```
python setup.py py2exe
```

Cela Crée un dossier Build et Dist. Dans ce dernier se trouve léxecutable ainsi Voici un exemple de setup.py incluant la copie des données nécessaires au programme ainsi que la résolution du problème lié à la librairie SIP :

```
from distutils.core import setup
import py2exe
from glob import glob
import os
```

```
data_files = [("docs", glob(r'docs\*.*')),("docs/accueil_pictures",
                glob(r'docs/accueil_pictures/*.*'))]

setup(console=[{"script":"diyabc.py"}],data_files=data_files,
            options={"py2exe":{"includes":["sip"]}})
```

L'option "window" plutôt que "console" ne fonctionne pas.

4.2.4 Solution choisie

Sous windows, j'utilise pyinstaller (révision 1355). Après avoir renoncé à l'écriture d'un script batch, j'ai découvert que git installait un bash tout à fait fonctionnel. J'ai donc écrit le script de génération d'un exécutable en bash.

Utilisation Ce script est : python_interface/docs/project_builders/windows_generation.sh . Une exécution sans paramètre affiche les paramètres à fournir :

- path_to_pyinstaller.py : chemin vers le script python de pyinstaller
- path_to_icon.ico : chemin vers le fichier ico qui sera l'icone du fichier .exe
- output_path : dossier cible (existant ou non) dans lequel se trouvera l'executable et les données du programme
- path_to_main.py : chemin vers le script python principal (diyabc.py dans notre cas)

Après exécution de ce script de génération, un dossier nommé diyabc-VERSION se trouve dans le dossier spécifié en output. A l'intérieur de ce dossier se trouve l'exécutable ainsi que le dossier docs contenant des images et la documentation.

Fonctionnement Ce script récupère le numéro de version dans le fichier version.txt qui se trouve avec les sources python de l'interface. Il crée ensuite un dossier temporaire où il copie tout le nécessaire à la construction de l'exécutable. Il modifie la variable VERSION dans les sources (diyabc.py). Il lance la génération avec pyinstaller puis copie les données à côté du .exe .

4.3 Linux

4.3.1 Installer PyQt4

PyQt4 et PyQwt5 sont packagés pour toutes les distributions. Aucun problème de ce côté.

4.3.2 pyinstaller

Pyinstaller semble ne pas fonctionner parfaitement sous linux. L'exécutable produit ne marche que sur la machine qui l'a créé. Je suis d'avis qu'il faut plutôt faire un paquet debian et/ou rpm et distribuer le programme de cette façon. C'est donc la solution que j'ai choisi.

4.3.3 Script de génération du paquet Debian

Ce script se situe dans python_interface/docs/project_builders/debian/ .

Utilisation Une exécution du script sans paramètre affiche la liste des paramètres à fournir :

- directory_of_main_source : chemin vers le dossier contenant le script python principal (python_interface dans notre cas)
- main_flag : flag qui détermine si le paquet généré comprendra ou pas la version dans son nom. S'il vaut MAIN, le paquet généré se nommera diyabc. . ..deb, s'il vaut autre chose, le paquet se nommera diyabc-VERSION. . ..deb .

Après exécution de ce script, le paquet debian se trouve dans le dossier courant.

Fonctionnement Ce script utilise le template se trouvant dans le même répertoire. Dans un premier temps, il modifie les fichiers d'information du paquet pour coller avec la version. Ensuite, il copie les sources nécessaires dans l'arborescence du paquet, modifie les fichiers nécessaires aux menus gnome et autres Il écrit enfin le contenu du script de lancement qui sera dans /usr/local/bin.

Dépôt debian Peut-être auront nous par la suite envie de mettre en place un dépôt debian pour distribuer plus facilement DIYABC aux utilisateur de Debian, d'Ubuntu et autres dérivés de Debian. Pour mettre en place un dépôt, il suffit d'avoir un serveur web opérationnel. Un dépôt n'est en fait simplement qu'un ensemble de dossier et de fichiers respectant un certain standart. Il existe plusieurs outils facilitant la création de cette arborescence. J'ai utilisé reprepro pour tester. En résumé, il suffit de créer un dossier qui sera la racine de notre dépôt :

```
mkdir debian
```

Puis d'y créer deux répertoires conf et incoming :

```
mkdir debian/conf
mkdir debian/incoming
```

Il faut ensuite éditer le fichier conf/distributions qui va contenir les types de distributions que l'on veut servir. Notre paquet étant fonctionnel sur toute distribution Linux, nous ne mettrons qu'une distribution dans ce fichier et tout le monde s'adressera au dépôt comme s'il possédait cette distribution. Le contenu du fichier distributions est de la forme :

```
Origin: CBGP
Label: INRA
Suite: stable
Codename: squeeze
Version: 1.0
Architectures: i386 amd64
Components: main
Description: Repository of CBGP softwares
```

Une fois qu'on a un paquet prêt à être ajouté au dépôt, on utilise reprepro pour éviter de créer à la main toute l'arborescence dans le dépôt. L'ajout d'un paquet s'effectue de la manière suivante :

```
reprepro -Vb path/to/my/repo includedeb stable path/to/my/package.deb
```

Voici deux sources de documentation pour aller plus loin avec reprepro : `http://www.isalo.org/wiki.debian-fr/index.php?title=Reprepro` et `http://doc.ubuntu-fr.org/tutoriel/comment_creer_depot` . On peut vouloir signer un dépôt avec une clé GPG ou encore faire un mirroir d'un autre dépôt.

4.4 MacOS 10.6 Snow Leopard

Sous MacOS, on pourrait créer un pkg qui installe python et les dépendances de DIYABC. C'est un peu lourd pour l'utilisateur. Py2app ne fonctionne pas, après avoir compilé un apptemplate pour i386_x64 il subsiste un problème dont l'origine m'est inconnue. Au lancement du .app, "DIYABC error" apparait et je n'ai aucun moyen de savoir pourquoi. J'ai donc retenu pyinstaller qui fonctionne parfaitement.

Sur la machine qui va générer le .app, il faut avoir un environnement de développement complet, à savoir python2.6, PyQt4 et PyQwt5.

4.4.1 Installer PyQt4

Problèmes PyQt n'est pas disponible en binaire sur MacOS. Il faut donc avoir Qt et gcc pour le compiler. Gcc n'est que très difficilement installable sans Xcode. Il faut donc installer Xcode. En suivant à peu près les instructions données à cette adresse :
http ://deanezra.com/2010/05/setting-up-pythonqt-pyqt4-on-mac-os-x-snow-leopard/
, on arrive à bout de cette installation qui est tout de même un gros morceau pour pas grand chose.

Pour la version 4.8.3 de PyQt, je n'ai trouvé d'autre solution que de commenter les 3 lignes qui concernent "designer" pour que la compilation fonctionne. En cas d'erreurs incompréhensibles, bien penser à faire un "make clean" pour effacer les .o déjà compilés qui peuvent provenir d'une compilation précédente mal paramétrée.

Solution La solution est d'utiliser les macports pour installer PyQwt (py26-pyqwt), ce qui installera tout le nécessaire. Il faut tout de même faire attention à numpy qui pose un problème de version de lib vector. La solution est de désinstaller numpy et de l'installer avec l'option "-atlas".

4.4.2 pyinstaller

Les version 1.4 et 1.5 ne fonctionnent pas. Par contre, pyinstaller svn est tout à fait opérationnel avec python2.6. Il faut tout de même effectuer une série d'opérations avant et après le build pour que le .app soit opérationnel.Se rérérer à
http ://diotavelli.net/PyQtWiki/PyInstallerOnMacOSX
Protocole :

- Lancer une première fois "python pyinstaller.py diyabc.py"
- Ajouter à la fin de diyabc.spec :

```
import sys
if sys.platform.startswith(‘‘darwin’’):
    app = BUNDLE(exe,
    appname='DIYABC',
    version='1.0')
```

- Relancer pyinstaller sur le specfile : "python pyinstaller diyabc/diyabc.spec"
- Changer une valeur dans le fichier Info.list
- Copier le contenu du dossier dist/nom_du_projet dans le dossier MacOs du .app

- Copier /Library/Frameworks/QtGui.framework/Versions/4/Resources/qt_menu.nib dans le dossier Ressources du .app
- Copier l'icone dans le dossier Ressources avec le nom "App.icns".

4.4.3 Solution choisie

Utilisation Comme sous windows, j'ai écrit un script bash qui utilise pyinstaller (révision 1355). Ce script se trouve dans python_interface/docs/project_builders/ . Une exécution sans paramètre affiche les paramètres à fournir :

- path_to_pyinstaller.py : chemin vers le script python de pyinstaller
- path_to_icon.icns : chemin vers le fichier icns qui sera l'icone de l'application (.app)
- output_path : dossier cible (existant ou non) dans lequel se trouvera l'application et les données du programme
- path_to_main.py : chemin vers le script python principal (diyabc.py dans notre cas)

Fonctionnement Comme les autres, ce script crée un dossier temporaire où il copie tout le nécessaire à la construction du .app . Il récupère le numero de version dans le fichier version.txt . Il modifie les sources en conséquence. Il lance pyinstaller une première fois pour générer le .spec . Il modifie le .spec pour produire un .app puis lance à nouveau pyinstaller pour générer le .app . Ensuite il effectue les opérations décrites dans la section 4.4.2 afin d'obtenir un .app fonctionnel.

Après exécution de ce script de génération, à l'intérieur du dossier d'output se trouve l'application ainsi que le dossier docs contenant des images et la documentation.

5 Améliorations à apporter

La partie sur le lancement des calculs sur le cluster est à améliorer. Elle n'a pas été testée ni lancé un grand nombre de fois et la finition est à revoir.

Il faudrait aussi effectuer le transfert de la table de référence générée une fois le calcul effectué sur le cluster.

L'interruption de connexion n'est pas gérée, les calculs continuent de tourner sur le cluster même en cas d'arrêt de l'interface mais on ne peut pas se connecter à nouveau pour voir l'avancement.