Tests, retour d'expérience dans le logiciel Siconos

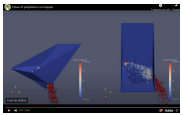
Franck Pérignon

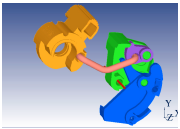
Séminaire rdatadev, 30 juin 2021

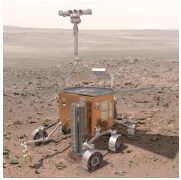
Contexte : le logiciel Siconos

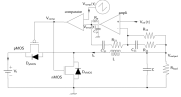
Modélisation et simulation de systèmes dynamiques non-lisses

https://nonsmooth.gricad-pages.univ-grenoble-alpes.fr/siconos/index.html









- De nombreux domaines d'application : mécanique (contacts, frottements), électronique, robotique, contrôle ...
- ... et les utilisateurs (et développeurs) qui vont avec !
  - Des habitudes et des contextes d'utilisation différents,
  - des manières d'exploiter le soft très variables (taille et types de problèmes ...),
  - sur plusieurs types de plates-formes/OS (Linux, MacOs, clusters de calcul et même des tentatives Windows 😉),
  - code utilisé pour la recherche (et donc en évolution constante) et par des industriels.

Quelques caractéristiques du code

- Gros code (plus de 400000 lignes), “vieux” ($>$ 15 ans)
- Plusieurs langages (C++/C/Python/Fortran, pilotage via cmake)
- Pas mal de dépendances plus ou moins pénibles (boost, blas-lapack, hdf5, suitesparse, swig, bullet-physics ...)
- Utilisation ‘standalone’ (via un programme Python ou C++) ou comme noyau de calcul, interfacé avec d'autres logiciel

Bref, que de bonnes raisons pour avoir pas mal de problèmes ... et donc mettre en place un environnement de "tests" pour, entre autres :

- Vérifier/valider, produire un code robuste et portable (augmenter la crédibilité des résultats, inciter à utiliser notre code), tester/anticiper des contextes d'utilisation ...
- Faciliter les interactions et les développements au quotidien.

Aujourd'hui : un retour d'expérience sur ce qui est fait dans le logiciel Siconos.

- Quels tests ?
  Unitaires, fonctionnels, de non-régression, d'intégration, ...
- Comment ?

Analyse statique du code

- 'Tester' le code : utiliser un éditeur "intelligent", capable de vérifier automatiquement, à la volée,la syntaxe, les types etc
- Passage "régulier" des outils d'analyse statique sur le code (pylint, ...)
- Des asserts en C++ ou python (en mode debug)

👉 quelques premiers tests faciles avec des outils qui apportent beaucoup relativement à l'effort demandé pour les mettre en place.

Tests unitaires, de non-régression

De "petits" tests pour vérifier le bon fonctionnement des méthodes, classes etc (appels des constructeurs, des opérateurs ...)
☞ on code et on vérifie directement que ça fonctionne.

En pratique dans Siconos :

- Un cadre pour piloter des suites de tests
  - framework cppunit pour le C++ (*https://freedesktop.org/wiki/Software/cppunit/*)
  - pytest pour Python (*https://docs.pytest.org/en/6.2.x/*)
  - fait maison pour le C ...
- Dans ces cadres, on écrit des fonctions "test", vérifiant un minimum de choses (en théorie une seule ...) qui renvoient vrai ou faux.
- CMake gère le process de création/exécution (ctest) des tests ainsi que la "publication" (cdash) des résultats

Exemple, fonction test en python

Très simple : il suffi d'écrire une liste de fonctions nommées test_..., pytest fait le reste.

```
def test_LagrangianLinearTIDS():
  ball = K.LagrangianLinearTIDS(q, v, mass)
  assert np.allclose(ball.q(), q, rtol=tol, atol=tol)
  assert np.allclose(ball.velocity(), v, rtol=tol, atol=tol)
  assert np.allclose(ball.mass(), mass, rtol=tol, atol=tol)
  ball.setFExtPtr(weight)
  assert np.allclose(ball.fExt(), weight, rtol=tol, atol=tol)
```

Exemple, fonction test en C++, la syntaxe cppunit

Implémentation

```
// Copy from a sparse
void SiconosVectorTest::testConstructor5()
{
  std::cout << "--> Test: constructor 5." <<std::endl;
  SP::SiconosVector v(new SiconosVector(*sv));
  CPPUNIT_ASSERT_EQUAL_MESSAGE("testConstructor5 : ", v->size() == sv->size(), true);
  CPPUNIT_ASSERT_EQUAL_MESSAGE("testConstructor5 : ", v->num() == Siconos::SPARSE, true);
  std::cout << "--> Constructor 5 test ended with success." <<std::endl;
}
```

Déclaration d'une "suite"

```
class SiconosVectorTest : public CppUnit::TestFixture
{
  // Name of the tests suite
  CPPUNIT_TEST_SUITE(SiconosVectorTest);
  // Declare all tests
  CPPUNIT_TEST(testConstructor0);
```

Exemple, fonction test en C

Des routines pour piloter des "collections" de tests.
Cas d'usage : une série de jeux de données à tester pour une série de solveurs.

- Pour chaque cas, un fichier C définit la liste de fichiers d'entrée (hdf5 ou ascii) et les paramètres des solveurs à tester
- A partir de ces infos et d'un modèle générique, CMake génère les fichiers C qui seront compilés pour créer la pile de tests

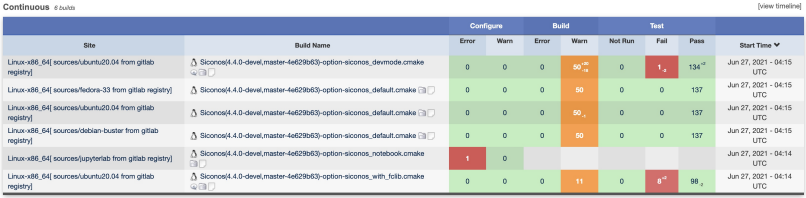
Pilotage via des fonctions et macros cmake/ctest

```
begin_tests(src/FrictionContact/test DEPS "${suitesparse}")
new_tests_collection(
  DRIVER fc_test_collection.c.in FORMULATION fc3d COLLECTION TEST_NSGS_COLLECTION_1
  EXTRA_SOURCES data_collection_1.c test_nsgs_1.c)
new_tests_collection(
  DRIVER fc_test_collection.c.in FORMULATION fc3d COLLECTION TEST_NSGS_COLLECTION_2
  EXTRA_SOURCES data_collection_2.c test_nsgs_1.c)
```

Visualisation des résultats des tests

Serveur CDash *http://siconos-dashboard.univ-grenoble-alpes.fr:8080/index.php?project=siconos*

- Une machine virtuelle (Nova),
- Installation/déploiement via une image docker fournie par CDash,
- Publication automatique des résultats de tests, géré par cmake (transferts de fichiers xml générés par ctest)
- Config : un fichier CTestConfig.cmake + fonctions 'ctest'



démo ...

Concrètement ?

Une petite démo

```
# inventaire des tests (fonc deps, options, \ldots)
# et préparation de la compilation/link des tests + ajout dans la pile
cmake -DWITH_TESTING=ON ...

# compilation/link des tests
make

#  Exécution et génération de rapports (xml)
make test
...
```

ou

```
ctest
# Publication vers un serveur cdash
make experimental
```

Tests fonctionnels

Une chaîne de traitement plus complexe, modélisation/simulation complète de divers problèmes.

- Un projet gitlab à part
  https://gricad-gitlab.univ-grenoble-alpes.fr/nonsmooth/siconos-tutorials
- Une batterie de cas tests, d'exemples complets, disponibles pour les utilisateurs.
- Des fichiers de référence pour valider les résultats

Pilotage : cmake / ctest / cdash

Résumons

- pas mal d'écriture de code pour les tests 😞,
- gestion complexe (mais automatique !) de la pile de tests grâce à cmake,
- discipline nécessaire pour écrire les tests au fur et à mesure des développements,
- vraiment utile, apporte un certain confort de développement,
- rapport de tests très complets (debug !),
- nécessite des valeurs/des fichiers de référence à jour.

Il manque une dernière étape indispensable : l'intégration continue.

Intégration continue

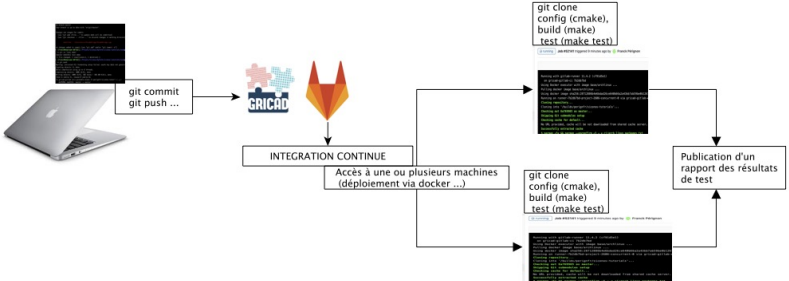
Vérifier systématiquement l'impact de toute modification du code source sur le fonctionnement, les performances etc.

Du point de vue des tests, les objectifs sont :

- d'automatiser la chaine de tests,
- de tester différents contextes (OS et configuration du code).

Concrètement :

- deux projets gitlab (siconos et siconos-tutorials)
- à chaque **push** vers le serveur, des tâches pré-définies par les développeurs sont exécutées.



Intégration continue dans Siconos, les étapes possibles

1. Génération d'images docker (une par OS/ensemble de dépendances),
2. Sauvegarde de ces images dans les registries du projet gitlab,
3. Jobs de configuration/build/tests (ctest) pour différentes configurations avec publication vers le serveur cdash (un pipeline par OS/config (image docker)).
4. Installation de siconos dans un container, génération/sauvegarde d'images "prêtes à l'emploi" ...
5. ... utilisées pour un pipeline de tests de simus complètes (siconos-tutorials).

Comment ?

Un fichier yaml de description de templates de jobs et de règles. Par exemple :

```yaml
.siconos-build:
  extends: .siconos-ctest
  variables:
    ctest_mode: Build
    #GIT_STRATEGY: none
  stage: build
  artifacts:
    paths:
      - build
    expire_in: 2 days

.examples-rules:
  rules:
    - if: $CI_COMMIT_MESSAGE =~ /\[with examples\]/
      when: always
    - when: never
```

Un fichier .gitlab-ci.yml, listant les jobs (basés sur les templates et les règles)
Par exemple

```
debian-buster:build:
variables:
    IMAGE_NAME: $CI_REGISTRY_IMAGE/sources/debian-buster
    cdash_submit: 1
    user_file: $CI_PROJECT_DIR/$siconos_confs/siconos_default.cmake
extends:
  - .siconos-build
  - .full-ci-rules
needs: ["debian-buster:configure"]
```

Un fichier cmake pour décrire la conf (ctest_driver_install_siconos.cmake) et un fichier à exécuter par chaque job (ctest_siconos.sh)

```
  ctest -S ${CI_PROJECT_DIR}/ci_gitlab/ctest_driver_install_siconos.cmake -Dmodel=$ctest_bui
   -DUSER_FILE=$user_file -DALLOW_PARALLEL_BUILD=$allow_parallel_build -DCDASH_SUBMIT=$cdash
    -V -DCTEST_MODE=${ctest_mode}
```

- Où tournent les jobs ? Dans des containers docker, sur des runners gitlab (machines virtuelles Nova ...)
- CI conditionnelle (branche, mots clés dans les commits, manuelle ...)
- Exploitation des variables d'environnement gitlab-ci (*https://docs.gitlab.com/ee/ci/variables/predefined_variables.html*)
- Connexion entre les projet (bridge) : sous certaines conditions, la CI du projet siconos déclenche celle du projet siconos-tutorials

Bilan / avis

- Fastidieux à mettre en place, difficiles à maintenir ...
- ... mais un garde-fou indispensable et très utile pour les développeurs (bon outil de debug).
- Rassurant pour les utilisateurs.
- Automatisation relativement facile à mettre en place via gitlab-ci (ou github/Travis)
- Tests, CI, automatisation : un pas vers une recherche reproductible
  (pub 💻 *https://reproduct2021.sciencesconf.org/*)
- Il faut le faire dès le début et au fur et à mesure des développements !
- Il faut exploiter au maximum les outils disponible (CTest, CDash, gitlab-ci, ...)
- Attention à l'impact de la CI

(re-pub 💻 : *https://ecoinfo.cnrs.fr/2020/11/20/plaquette-je-code-les-bonnes-pratiques-en-eco-conception-de-service-numerique-a-destination-des-developpeurs-de-logiciels/*)