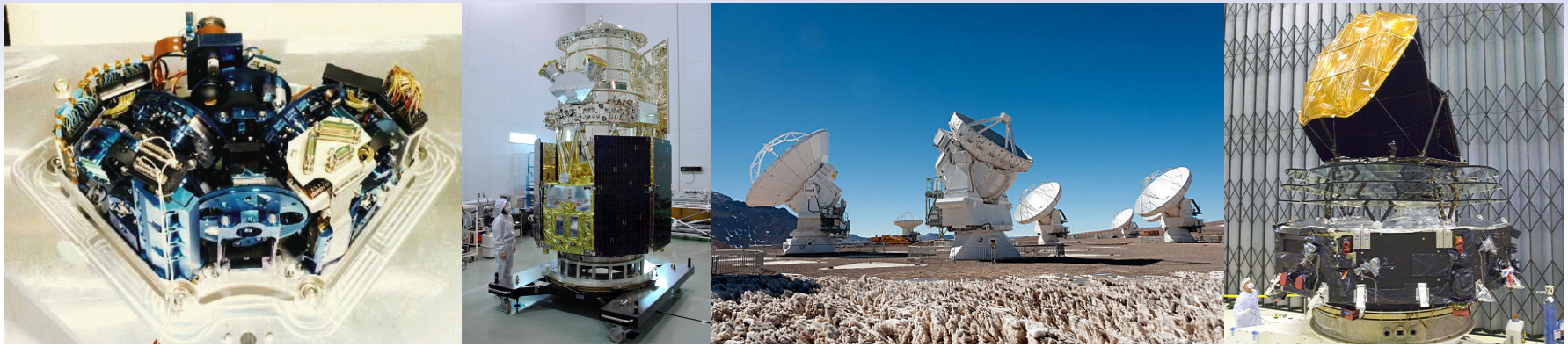
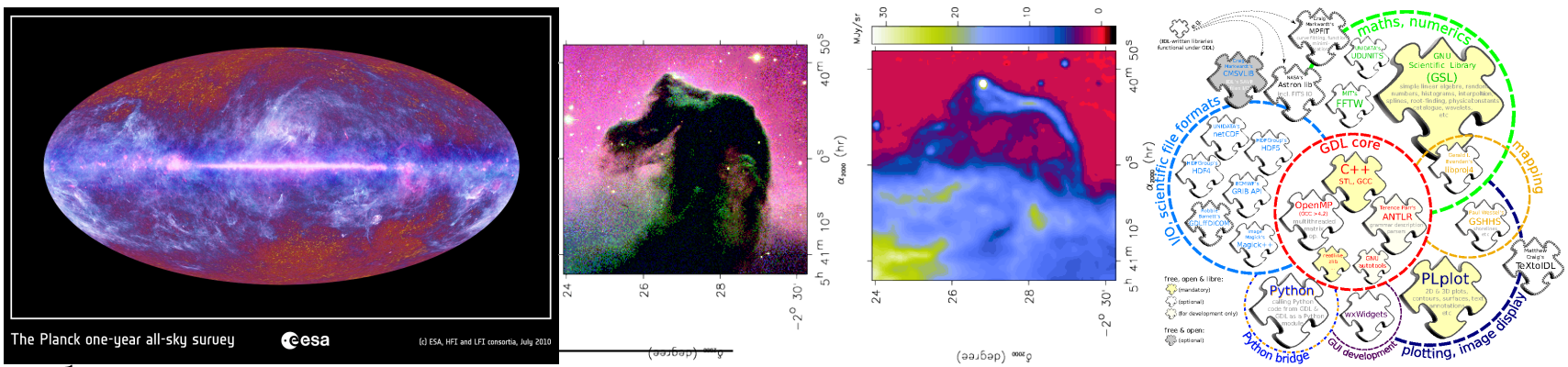


Alain Coulais[1]

LERMA (UMR 8112), CNRS (INSU) et Observatoire de Paris
& AIM (UMR 7158) au SAp, CEA

27 Novembre 2018

GDL (Gnu Data Language) et ses tests



[1] alain.coulais@obspm.fr

Quelques billes

Nous ne faisons pas d'informatique théorique

Les codes pro en astro sont souvent sans suite de test ... et donneront *parfois* des résultats différents sur des machines différentes

Rarement il y a une démarche qualité en amont (critère : *ça marche* [sic] sur le jeu de données qui nous intéresse ...)
Et peu de traçabilité ! (emploi pas si fréquent de SVN ou Git)

Je suis là pour vous écouter ! Attraper des idées, des démarches, du vocabulaire !

Présentation rapide d'IDL

IDL est un vieux langage interprété (1977), pas *moderne* du tout !
Grand nombre de procédures et fonctions (FFT, plot, INVERT() ....)
IDL est extrèmement efficace pour traiter les données en Astronomie : séries temporelles, images 2D, cubes 3D

IDL est remarquable par sa concision.

Des codes, même gros, écris en syntaxe IDL il y a 10, 20 ans, 30 ans continuent de tourner sans changer une virgule
Très large base de librairies tierces (AstroLib, Coyote, MPfit packagés dans Debian Astro)

A quoi comparer IDL ? Python, R, Matlab, Scilab, Octave ...

GNU Data Language, clone libre d'IDL

GDL est un clone libre d'IDL (il existe un autre clone : FL)
Existe depuis 2004, d'abord sur SourceForge, désormais sur GitHub (2018) https://github.com/gnudatalanguage/gdl/

Version 0.9.9 (Nov. 2018)

GDL est développé par une poignée de contributeurs *bénévoles* qui ne sont pas nécessairement de vrais informaticiens ou programmeurs (formation sur le tas ... chercheurs en astronomie, en géophysique ...)

Une très large fraction des fonctions de base d'IDL sont disponibles, suffisamment pour faire tourner des codes de plusieurs milliers ou dizaines de milliers de lignes (en syntaxe IDL)

Un langage, 3 compilateurs !

En fait, on se retrouve dans une situation un peu semblable à des compilateurs (C, Fortran) : un langage (la syntaxe IDL) et trois compilateurs : IDL, GDL et FL

Un (gros) code C/C++ compilé avec GCC, Clang & icc, on se sent mieux ! (Mm̂e si ce n'est pas une garantie !). Et qui donne les mêmes résultats sur une suite de tests, encore mieux !

Donc oui, avoir GDL apporte un plus aux codes en IDL !

Et je montrerai à la fin que oui, on peut tester IDL aussi :)

GNU Data Language, clone libre d'IDL

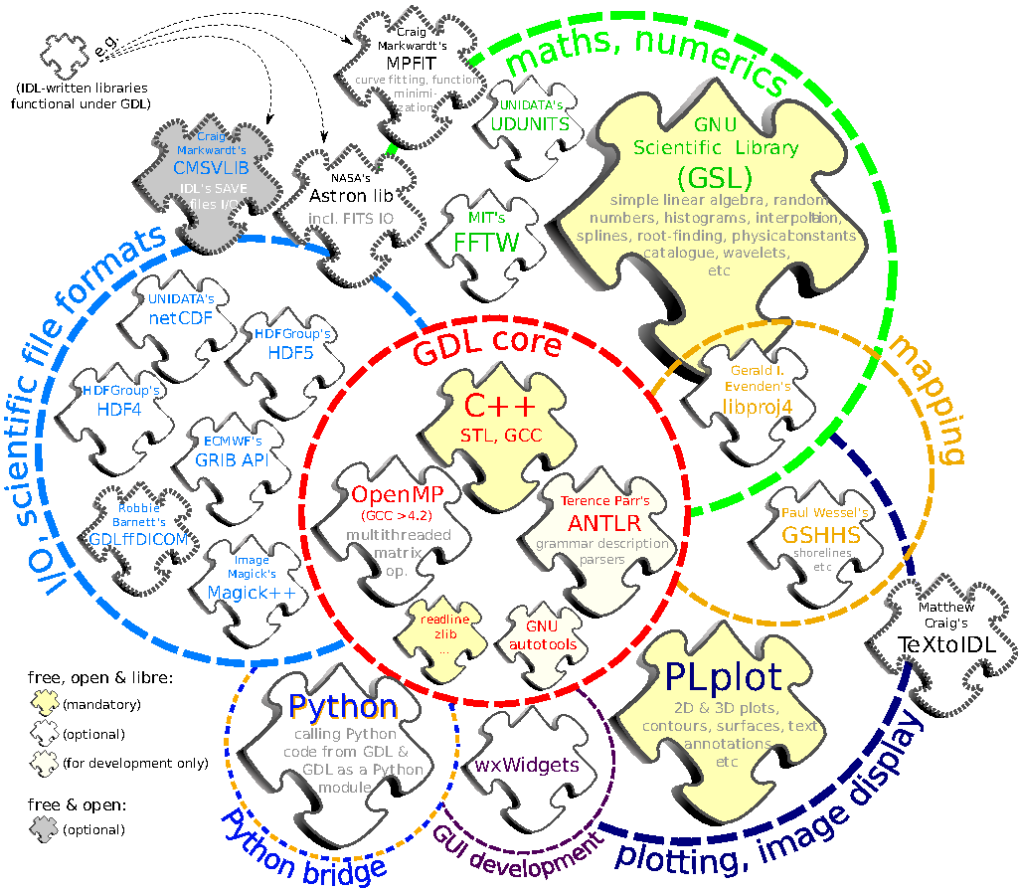


Figure : Dépendances de GDL à des librairies libres tierces: PLplot, FFTw, GSL, OpenMP, Eigen3, GSHHG, NetCDF, HDF ...

Planck beams; pipeline Nika2; Corono MIRI JWST (Danielski & al., 2018) ...

Des tests dans GDL

215 000 lignes de code en c++, le parser utilise Antlr
15 types différents (+ null)

`make check`

- tests de compilation (un code en syntaxe IDL doit être compilable en GDL)
- tests unitaires
- tests numériques
- tests OpenMP
- tests *macroscopiques*

Une bonne nouvelle : on utilise des libs. tierces largement testés : FFTw, Eigen3, la GSL, plplot ...

Les tests numériques dans GDL

Ce n'est pas nécessairement le plus difficile. (on peut poser le problème, on sait ou on va.)

Une très bonne référence, qui a donné des idées : la suite de test dans la GNU GSL

Je trouve que les interactions entre les mot-clefs sont plus compliqués à tester que les tests numériques. Certaines pro/func du langage ont des dizaines de keywords ...

Les tests OpenMP dans GDL

Alors là, on rentre dans un univers pas totalement sous contrôle ! (c'est un truc qui m'effraie dans les gros codes : on finit par *moyenner* (cacher) les effets des bugs !)

On sait jouer, si nécessaire, avec le nombre de cœurs actifs

On a pris une approche statistique :
on tire N (grand, $>$ 100 000) points sur une distribution de bruit (gaussien ou uniforme selon les cas) et on vérifie à la fin que le résultat est bien consistant.
(tri SORT(), localisation WHERE(), TOTAL(), ...)

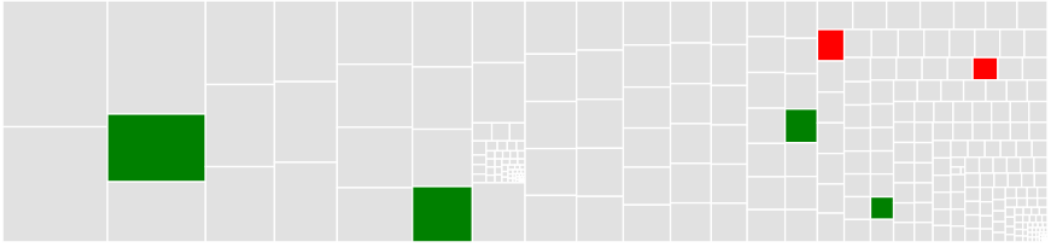
On a du MPI, hors sujet ici.

Taux de couverture des tests

Depuis que nous sommes dans GitHub, au printemps 2018, grâce à CodeCov, nous savons que notre suite de test couvre environ 43% du code source (la partie en C++)
Avec mes étudiants, on espérait augmenter cette couverture l'été 2018, on a été pris par des activités plus amusantes ! Alors même que c'était dans les sujets des stages !

Codecov Report

Merging #520 into master will **increase** coverage by `0.01%`.
The diff coverage is `89.65%`.



```
@@            Coverage Diff             @@
##           master     #520      +/-   ##
==========================================
+ Coverage   43.76%   43.77%   +0.01%
==========================================
  Files         295      295
  Lines       95757    95761       +4
```

Tests macroscopiques : pipelines

Si on regarde le taux de couverture (faible) et la complexité du code, on pourrait se dire que c'est sans espoir.
Mais en fait, comme on a trois interpréteurs (IDL, GDL, FL), on utilise des *gros* codes (pipelines) sur des jeux de données pour tester en parallèle un code *sous contrôle* sur de très nombreux cas:

- les tests sur les I/O
- les tests sur les formats et libs. de format (fits, netcdf, XDR, ...)
- la validité des calculs numériques
- la qualité des sorties graphiques
- temps de calcul
- ...

Si deux figures sont trop différentes, il y a un problème (erreur numérique ?)
Si les échelles sont trop différentes, il y a un problème (erreur graphique ?)
Si temps de calcul très différent, on va vite trouver la fonctionnalité en retrait
...
Ne pas oublier qu'on a un interpréteur : on s'arrète ou on veut, avec la main sur les données ...

Les tests du langage de référence !

Et ça marche ! On trouve des bugs, des erreurs dans la doc. ou des techniques sous-optimales dans le langage officiel (IDL)

- Erreur numérique : INTERPOLATE(/double) introduit dans IDL 8.2.3
- Méthode sous optimale : QROMO()
- Erreur de documentation l'ensemble des paramètres d'entrée des functions mathématiques d'IDL ont été revus entre IDL 8.2 et IDL 8.4

```
IDL82> print, beseli([3], [1.,2,2,2])
3.95337 0.00000 48529.1 48529.1
IDL82> print, beseli(3, [1.,2,2,2])
3.95337 2.24521 2.24521 2.24521

IDL84> print, beseli([3], [1.,2,2,2])
3.95337
IDL84> print, beseli(3, [1.,2,2,2])
3.95337 2.24521 2.24521 2.24521
```

(• *feature* : test insuffisant des inputs : INTERPOL() ne vérifie pas la monotonie des entrées ! GDL le teste.)

Exemple de notre suite de test

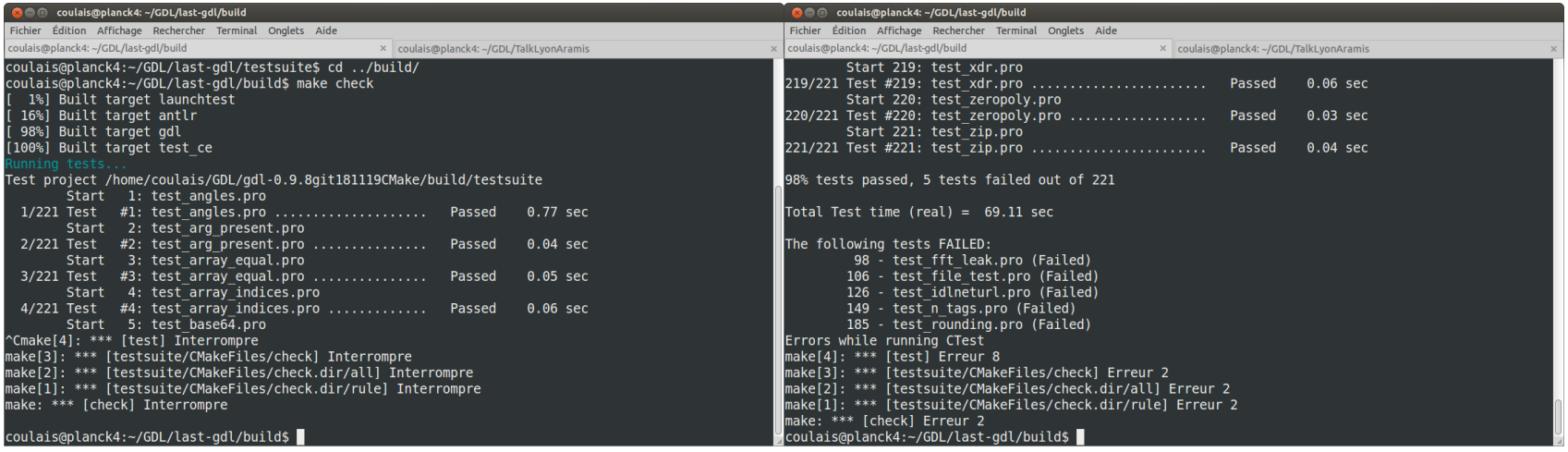


Figure : 221 tests *élémentaires* ici (dépend des options de compilation); certains redondants; on sait que certains ne passent pas (par ex. `test_idlneturl.pro` sans réseau !)

Tests via GitHub (Continuous integration)

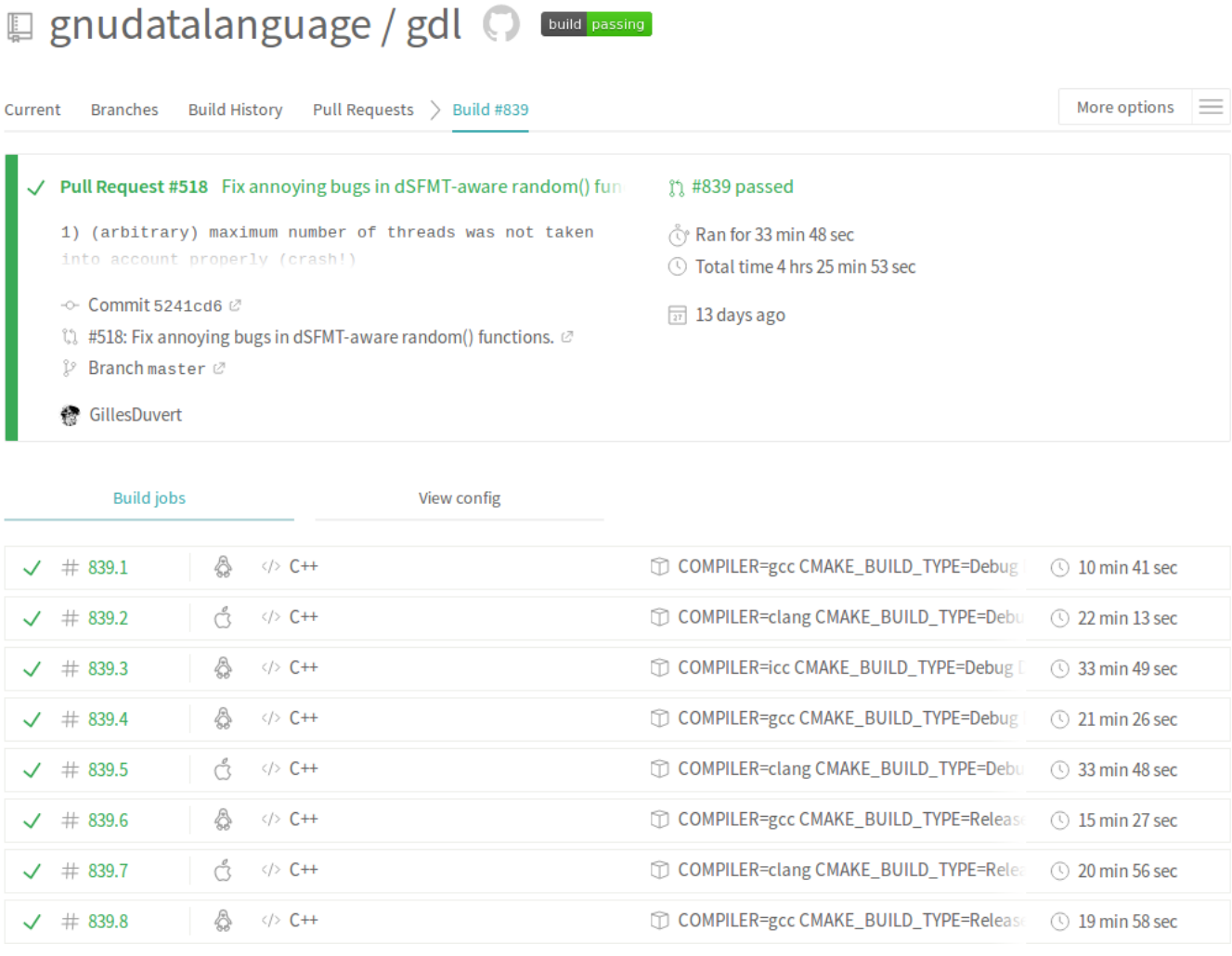
Nous faisons tester via GitHub :

- sur Travis (CI continuous integration) :
  - Ubuntu & CentOS, OSX
  - chacun des trois compilo (gcc, icc, clang)
  - pour les *third party libs.*, une version *a minima* et une version *full* (eg : avec et sans Eigen3, FFTw, ReadLine, ...)
- sur Appveyor
  - la version MS-win

Inconvénients : lent (environ 1h pour avoir une sortie complète), problème fréquent de paquet(s) manquant(s).

Etrangement, nos VMs restent utiles !

Travis sur GitHub (Continuous integration)



Difficultés

• Code source hétérogène: Notre code a presque 15 ans d'âge, et les contributeurs n'ont pas tous le même niveau, loin de là.

• Multi-plateforme : Nous visons un code OK sous Linux, *BSD, OSX & MS-win.
Les évolutions internes à OSX sont *fatiguantes* ! (depuis 10.6 ...). Quand à Brew, j'ai l'impression qu'il change de règles tous les mois !

• multi-compilos : OK pour GCC, Clang & icc Je ne vous dis pas la quantité de `#ifdef` dans certains endroits du code ...

• les librairies systèmes ne se comportent pas toujours à l'identique (BSD et Linux ne pourraient pas se mettre d'accord ?!)

Démarche

Accepter la complémentarité des profils : un codeur rapide, malin, ne sera pas nécessairement un bon testeur !

S'assurer dès l'ajout d'une nouvelle fonctionnalité qu'elle a des tests associés : il plus facile de faire revenir qlq1 sur un code frais que 6 mois ou 2 ans plus tard (heu, j'ai fait quoi, là ?!)

Vérifier dès tout changement dans le code que : (1) des tests couvrent ce changement, sinon chercher à en ajouter (2) essayer d'imaginer des effets de bord (si changement il y a, c'est sans doute qu'il y avait un problème ...)

L'expérience montre que peu de personnes font régulièrement tourner la suite de test ! (on fait confiance aux packageurs, on se fait confiance ...)

Test sur benchmark : diagnostic

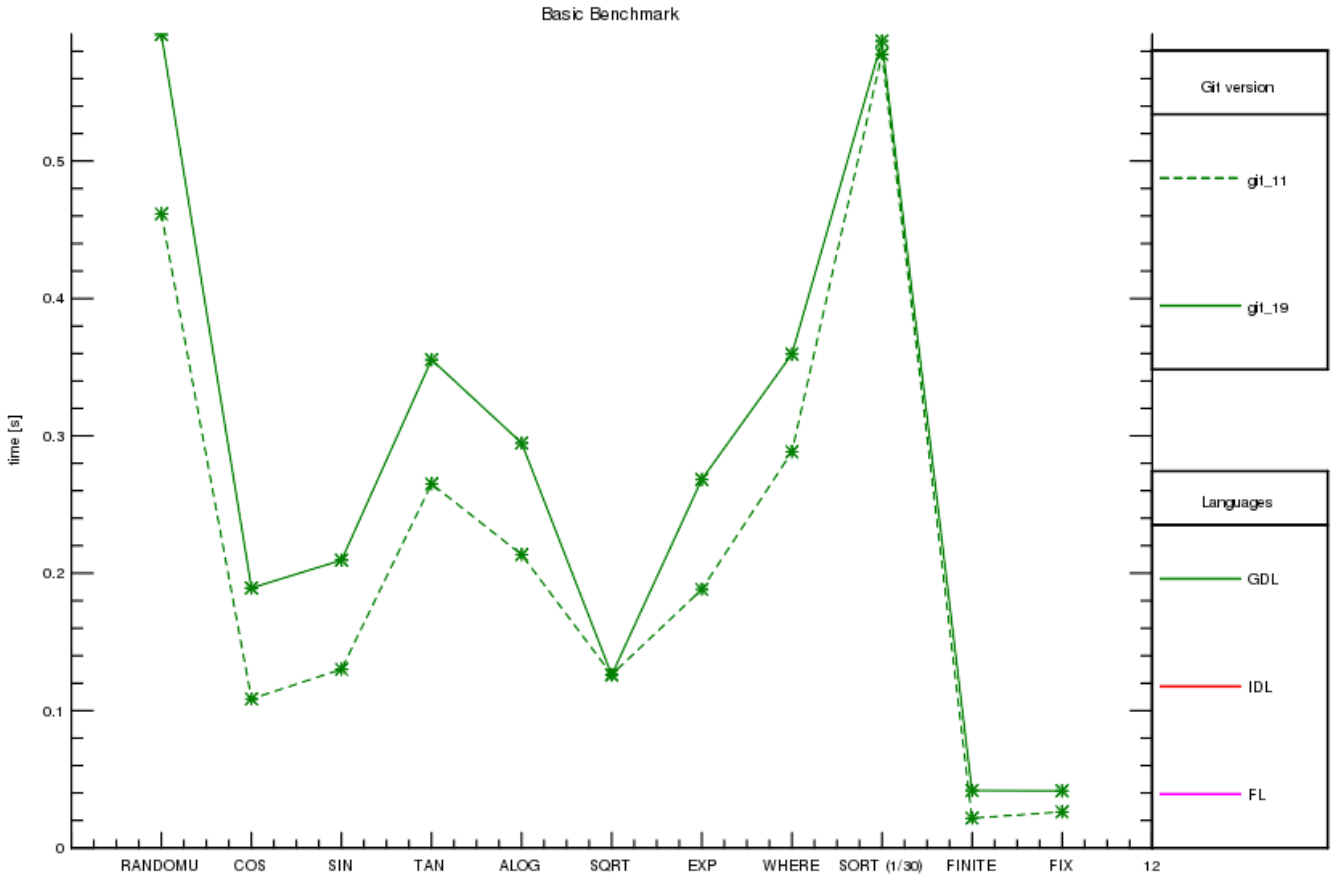
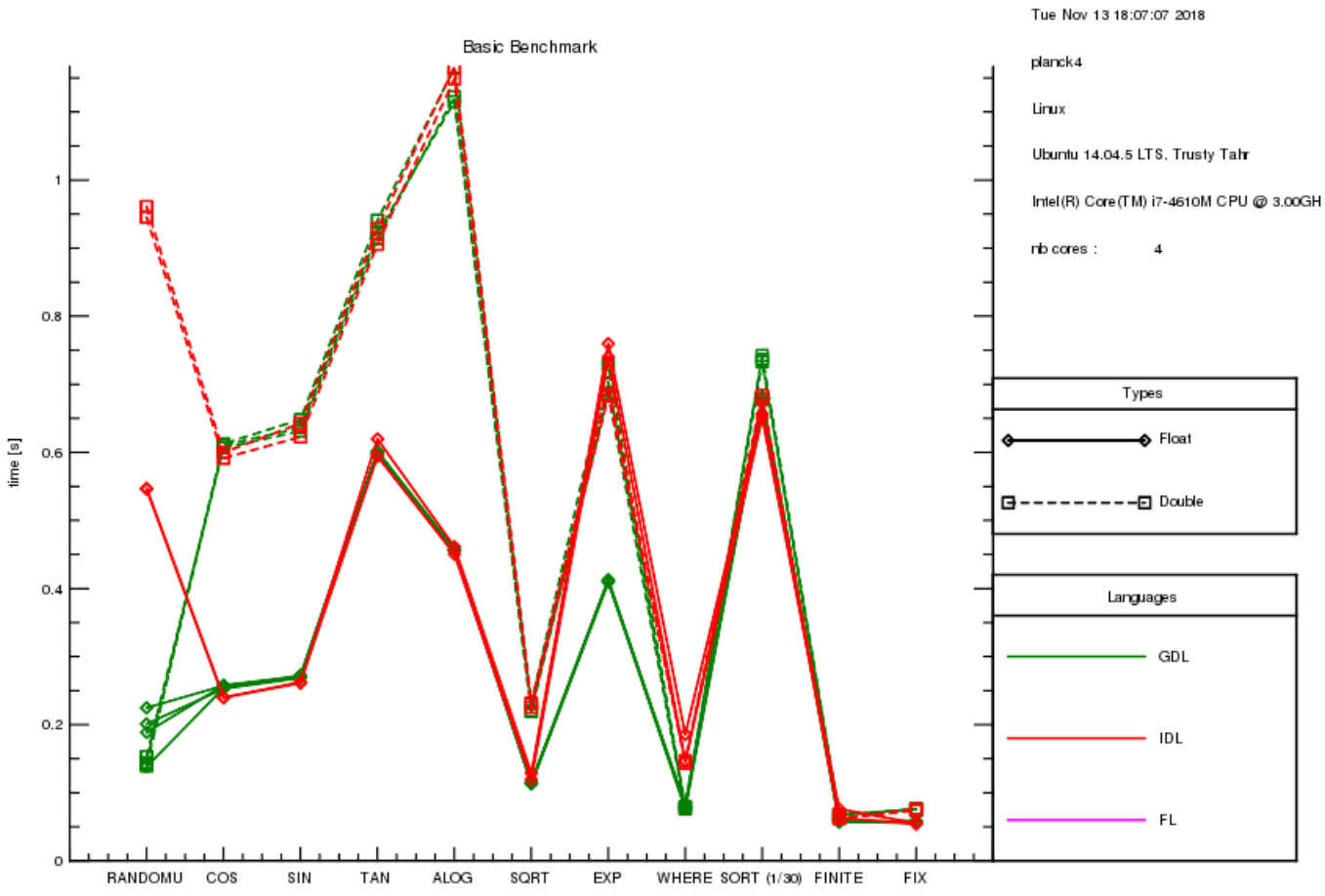


Figure : Regression de performance entre le 11 et le 19 Juin, le tout au milieu d'un commit avec environ 50 fichiers modifiés ...

Encore fallait-il avoir un indicateur de benchmark !

Test sur benchmark : correction



Anecdote

Je suis actuellement embarqué dans un simulateur (en python) d'un instrument d'un gros projet spatial, dans une équipe délocalisée d'une dizaine de personnes.

Instruction du chef de projet avant la sortie de la 1.0 :

Des tests unitaires ne passent plus ? Les supprimer !

Commentaire de mon collègue hier au labo, un peu écœuré (il a besoin de cas un peu inattendus du similateur pour dimensionner le pipeline) :

Chaque fois que je démarre le simulateur je trouve un bug

Conclusion

- **Sans suite de test, le projet GDL serait mort depuis longtemps**
- Les tests ne couvrent que 43% du code :(
- Plus la couverture en tests du code augmente, moins nous avons de retour (à taux de téléchargement $\sim$ constant)
- La complexité des bugs a sérieusement augmenté, il est devenu très rare d'avoir des retours sur des bugs triviaux.
- Les utilisateurs sont aussi nos testeurs ! (la plupart le savent !)
- Les tests sur les benchmarks ne sont que ponctuels. Pas simples à faire proprement, rarement pertinents sur les VM de test.

Un sentiment de roche de Sisyphe ...