GNU Guix, vers la reproductibilité computationnelle ?

Simon Tournier

Inserm US53 - UAR CNRS 2030
simon.tournier@inserm.fr

Journée BlueHats, 8 novembre 2022

https://hpc.guix.info

Université Paris Cité

Pourquoi j'en suis venu à GNU Guix

≈ 2010 **Thésard**

Développement d'1-2 outils utilisant un gestionnaire de paquets classique
(Simulation numérique `C` et `Fortran` avec Debian / Ubuntu / `apt`)

≈ 2014 **Post-doc**

Développement de 2-3 outils utilisant un gestionnaire de paquets sans droit administrateur
(Simulation numérique `Python` et `C++` avec `conda`)

2016 **Ingénieur. de Recherche**

- Administration d'un *cluster* (`modulefiles`)
- Utilisation de 10+ outils pour un même projet

(Analyse « bioinformatique »)

Question : **pourquoi cela fonctionne-t-il pour Alice et pas pour Bob ? Et vice-versa.**

Le monde est *open*, n'est-ce pas ?

- *open* journal
- *open* data
- *open* source
- *open* science
- *open* etc.

Quel est le problème de reproductibilité dans un contexte scientifique ?
(même si tout devient *open*)

« ordinateur » : traitement automatique de données

Le monde est *open*, n'est-ce pas ?

- *open* journal
- *open* data
- *open* source
- *open* science
- *open* etc.

Quel est le problème de reproductibilité dans un contexte scientifique ?
(même si tout devient *open*)

« ordinateur » : traitement automatique de données $\Longrightarrow$ environnement computationnel

Le monde est *open*, n'est-ce pas ?

- *open* journal
- *open* data
- *open* source
- *open* science
- *open* etc.

Quel est le problème de reproductibilité dans un contexte scientifique ?
(même si tout devient *open*)

« ordinateur » : traitement automatique de données $\Longrightarrow$ environnement computationnel

$$\left(\begin{array}{ccc} \text{« ordinateur »} \approx \text{instrument} & \text{et} & \text{« calcul »} \approx \text{mesure} \\ \text{environnement computationnel} & \leftrightarrow & \text{conditions expérimentales} \end{array} \right)$$

Le monde est *open*, n'est-ce pas ?

- *open* journal
- *open* data
- *open* source
- *open* science
- *open* etc.

Quel est le problème de reproductibilité dans un contexte scientifique ?
(même si tout devient *open*)

« ordinateur » : traitement automatique de données $\Longrightarrow$ environnement computationnel

$$\left(\begin{array}{ccc} \text{« ordinateur »} \approx \text{instrument} & \text{et} & \text{« calcul »} \approx \text{mesure} \\ \text{environnement computationnel} & \leftrightarrow & \text{conditions expérimentales} \end{array} \right)$$

Quel contrôle de l'environnement computationnel ?

Exemple d'un calcul

Listing 1 – Fonction J_0 de Bessel en langage C

```c
#include <stdio.h>
#include <math.h>

int main(){
  printf("%E\n", j0f(0x1.33d152p+1f));
}
```

Alice voit : 5.643440E-08
Carole voit : 5.963430E-08

Pourquoi ? Le code est disponible pourtant.

Établir si le différence est significative ou non est laissé à l'expertise des scientifiques du domaine.

Quelques questions sur ce calcul

```c
#include <stdio.h>
#include <math.h>

int main(){
  printf("%E\n", j0f(0x1.33d152p+1f));
}
```

Alice	voit :	5.643440E-08
Carole	voit :	5.963430E-08

- Quel compilateur ?
- Quelles bibliothèques (`<math.h>`) ?
- Quelles versions ?
- Quelles options de compilation ?

Par exemple : options de constructions (compilation)

Alice et Carole utilisent « GCC à la version 11.2.0 »

```c
#include <stdio.h>
#include <math.h>

int main(){
  printf("%E\n", j0f(0x1.33d152p+1f));
}
```

```
alice@laptop$  gcc bessel.c                    && ./a.out
               5.643440E-08
carole@desktop$ gcc bessel.c -lm -fno-builtin  && ./a.out
               5.963430E-08
```

(Ah, sacré *constant folding*)

Par exemple : options de constructions (compilation)

Alice et Carole utilisent « GCC à la version 11.2.0 »

```c
#include <stdio.h>
#include <math.h>

int main(){
  printf("%E\n", j0f(0x1.33d152p+1f));
}
```

```
alice@laptop$   gcc bessel.c                    && ./a.out
                5.643440E-08
carole@desktop$ gcc bessel.c  -lm -fno-builtin  && ./a.out
                5.963430E-08
```

(Ah, sacré *constant folding*)

Alice et Carole ont fait leur calcul dans **deux environnements computationnels différents**

Par exemple : options de constructions (compilation)

Alice et Carole utilisent « GCC à la version 11.2.0 »

```c
#include <stdio.h>
#include <math.h>

int main(){
  printf("%E\n", j0f(0x1.33d152p+1f));
}
```

```
alice@laptop$   gcc bessel.c                    && ./a.out
                5.643440E-08
carole@desktop$ gcc bessel.c  -lm -fno-builtin  && ./a.out
                5.963430E-08
```

(Ah, sacré *constant folding*)

Alice et Carole ont fait leur calcul dans **deux environnements computationnels différents**

Il faut donc plus que les numéros de versions

Les questions d'un environnement computationnel

- Quelles sont les sources des outils ?
- Quelles sont les outils requis pour la construction ?
- Quelles sont les outils requis pour l'exécution ?
- Comment chaque outil est-il produit ? (récursivement)

Répondre à ces questions signifie **contrôler la variabilité** de l'environnement computationnel

Les questions d'un environnement computationnel

- Quelles sont les sources des outils ?
- Quelles sont les outils requis pour la construction ?
- Quelles sont les outils requis pour l'exécution ?
- Comment chaque outil est-il produit ? (récursivement)

Répondre à ces questions signifie **contrôler la variabilité** de l'environnement computationnel

Comment capturer ces informations ?

(Réponses usuelles : Gestionnaire de paquets (Conda, APT, Brew, ...) ; *Modulefiles* ; Conteneur ; etc.)

Enjeu de la reproductibilité en recherche

D'un point de vue de la « méthode scientifique » :

Tout l'enjeu est le **contrôle de la variabilité**

D'un point de vue de la construction du « savoir scientifique » (un caractère universel ?) :

- Un observateur indépendant doit être capable d'observer le même résultat.
- L'observation doit être pérenne (dans une certaine mesure).

Enjeu de la reproductibilité en recherche

D'un point de vue de la « méthode scientifique » :

Tout l'enjeu est le **contrôle de la variabilité**

D'un point de vue de la construction du « savoir scientifique » (un caractère universel ?) :

- Un observateur indépendant doit être capable d'observer le même résultat.
- L'observation doit être pérenne (dans une certaine mesure).

Dans un monde où (presque) tout est donnée numérique (*data*),

comment refaire plus tard et là-bas ce qui a été fait aujourd'hui et ici ?

(sous entendu avec un « ordinateur »)

Solution(s)

1. gestionnaire de paquets : APT (Debian/Ubuntu), YUM (RedHat), etc.
2. gestionnaire d'environnements : Modulefiles, Conda, etc.
3. conteneur : Docker, Singularity

Guix = #1 + #2 + #3

APT, Yum Difficile de faire coexister plusieurs versions ou revenir en arrière ?

Modulefiles Comment sont-ils maintenus ? (qui les utilise sur son *laptop* ?)

Conda Quelle granularité sur la transparence ? (qui sait comment a été produit PyTorch dans `conda install torch` ? (lien))

Docker Dockerfile basé sur APT, YUM etc.

```
RUN apt-get update && apt-get install
```

Guix est gestionnaire d'environnements sous *stéroïde*

un **gestionnaire de paquets** (comme APT, Yum, etc.)
transactionnel et déclaratif (revenir en arrière, versions concurrentes)
qui produit des ***packs* distribuables** (conteneur Docker ou Singularity)
qui génèrent des ***machines virtuelles* isolées** (*à la* Ansible ou Packer)
sur lequel on construit une distribution Linux (mieux que les autres ? :-))
. . .et aussi une bibliothèque Scheme. . . (extensibilité, que voilà !)

Ce qu'il faut retenir :

la gestion de paquets *fonctionnelle* $\Rightarrow$ reproductibilité

Guix fonctionne sur n'importe quelle distribution Linux

(Facile à essayer. . .)

Sur une *distro externe*

Guix s'installe sur **n'importe quelle distribution** Linux récente.

Il faut les droits administrateur (`root`) pour l'installation.

```
$ cd /tmp
$ wget https://git.savannah.gnu.org/cgit/guix.git/plain/etc/guix-install.sh
$ chmod +x guix-install.sh
$ sudo ./guix-install.sh
```

(Quelques réglages supplémentaires, voir le manuel)

Pour commencer :

```
$ guix help
```

Guix, un gestionnaire de paquets comme les autres !

```
guix search dynamically-typed programming language # 1.
guix show    python                                # 2.
guix install python                                # 3.
guix install python-ipython python-numpy           # 4.
guix remove  python-ipython                        # 5.
guix install python-matplotlib python-scipy        # 6.
```

alias de `guix package`, p. ex. `guix package --install`

Transactionnel

```
guix package --install python                                    # 3.
guix package --install python-ipython python-numpy               # 4.
guix package -r python-ipython -i python-matplotlib python-scipy # 5. & 6.
```

Guix, un gestionnaire de paquets comme les autres ?

- Interface *ligne de commande* comme les autres gestionnaires de paquets
- Installation/suppression sans privilège particulier
- Transactionnel (= pas d'état « *cassé* »)
- *Substituts* binaires (téléchargement d'éléments pré-construits)

15 min, c'est court :-) :

- ~~Les *profils* et leur composition~~
- ~~Gestion déclarative~~
- ~~Environnement isolé à la volée~~
- ~~Génération image Docker~~

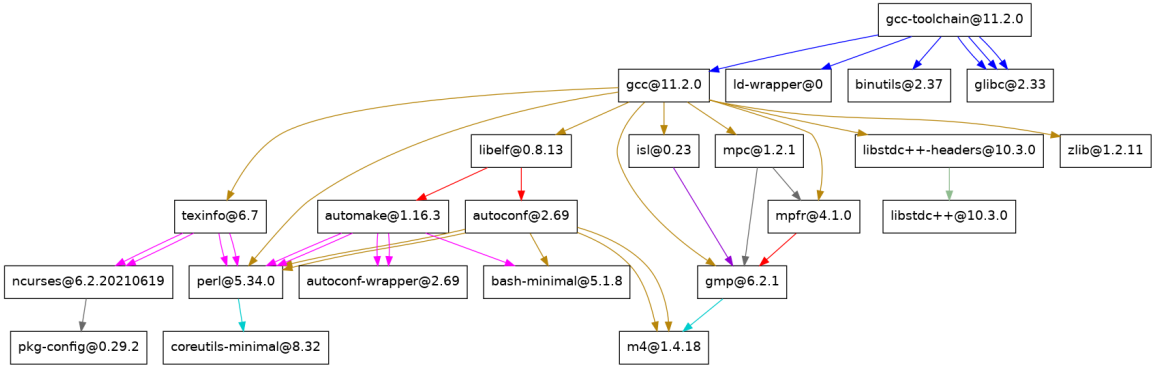
mini-tuto 1h aux JRES – Marseille, 2022 (lien PDF) (lien Vidéo)
https://replay.jres.org/w/3TuYmocHwKtzs7q1VtL1GB

Très bien tout cela, mais en quoi est-ce reproductible ?

Parlons de versions !

(Exemple : Carole collabore avec Alice)

Alice dit « GCC à la version 11.2.0 »



Est-ce la même version de GCC si `mpfr` est à la version 4.0 ?

Graphe complet : 43 ou 104 ou 125 ou 218 nœuds
(suivant ce que l'on considère comme graine binaire du *bootstrap*)

Quelle est ma version de Guix ?

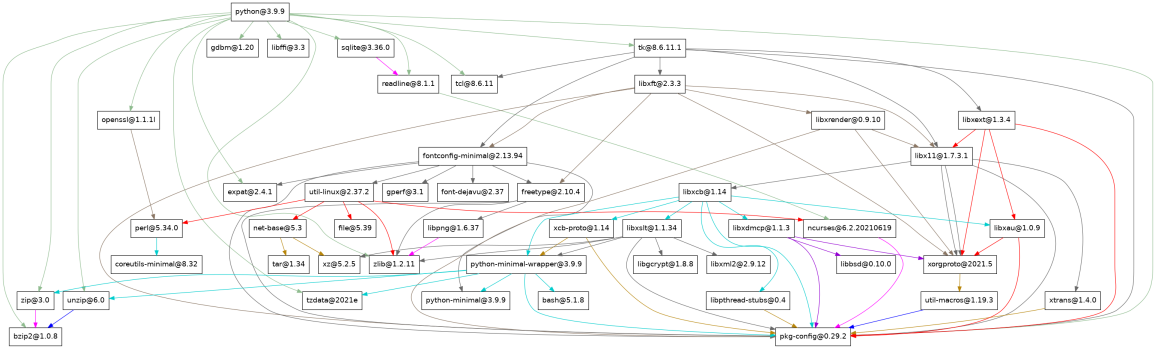
```
$ guix describe
Generation 76 Apr 25 2022 12:44:37 (current)
  guix eb34ff1
    repository URL: https://git.savannah.gnu.org/git/guix.git
    branch: master
    commit: eb34ff16cc9038880e87e1a58a93331fca37ad92

$ guix --version
guix (GNU Guix) eb34ff16cc9038880e87e1a58a93331fca37ad92
```

Un état fixe toute la collection des paquets et de Guix lui-même

(Un état peut contenir plusieurs canaux (*channel* = dépôt Git),
avec des URL, branches ou commits divers et variés)

État = Graphe Acyclique Dirigé (*DAG*)

```
guix graph --max-depth=6 python | dot -Tpng > graph-python.png
```



Graphe complet : Python = 137 nœuds, Numpy = 189, Matplotlib = 915, Scipy = 1439 nœuds

Révision = un graphe spécifique

« GCC à la version 11.2.0 » = un graphe

```
$ guix describe
Generation 76 Apr 25 2022 12:44:37 (current)
  guix eb34ff1
    repository URL: https://git.savannah.gnu.org/git/guix.git
    branch: master
    commit: eb34ff16cc9038880e87e1a58a93331fca37ad92
```

La révision `eb34ff1` capture **tout** le graphe

- Alice dit « j'ai utilisé Guix à la révision `eb34ff1` »
- Carole connaît toutes les informations pour reproduire le même environnement

Collaboration

Alice décrit son environnement :

- la liste des outils avec le fichier `liste-outils.scm`

génère son environnement avec, e.g.,

```
guix shell -m liste-outils.scm
```

Alice décrit son environnement :

- la liste des outils avec le fichier `liste-outils.scm`
- la révision (Guix lui-même et aussi potentiellement les autres canaux) :

```
guix describe -f channels > état-alice.scm
```

génère son environnement avec, e.g.,

```
guix shell -m liste-outils.scm
```

collaborer = partager un environnement computationnel

Alice décrit son environnement :

- la liste des outils avec le fichier `liste-outils.scm`
- la révision (Guix lui-même et aussi potentiellement les autres canaux) :

```
guix describe -f channels > état-alice.scm
```

génère son environnement avec, e.g.,

```
guix shell -m liste-outils.scm
```

collaborer = partager un environnement computationnel ⇒ partager un graphe

Alice décrit son environnement :

- la liste des outils avec le fichier `liste-outils.scm`
- la révision (Guix lui-même et aussi potentiellement les autres canaux) :

```
guix describe -f channels > état-alice.scm
```

génère son environnement avec, e.g.,

```
guix shell -m liste-outils.scm
```

collaborer = partager un environnement computationnel ⇒ partager un graphe

Alice décrit son environnement :

- la liste des outils avec le fichier `liste-outils.scm`
- la révision (Guix lui-même et aussi potentiellement les autres canaux) :

```
guix describe -f channels > état-alice.scm
```

génère son environnement avec, e.g.,

```
guix shell -m liste-outils.scm
```

et **partage ses deux fichiers** : `état-alice.scm` et `liste-outils.scm`.

collaborer = partager un environnement computationnel ⇒ partager un graphe

Alice décrit son environnement :

- la liste des outils avec le fichier `liste-outils.scm`
- la révision (Guix lui-même et aussi potentiellement les autres canaux) :

```
guix describe -f channels > état-alice.scm
```

génère son environnement avec, e.g.,

```
guix shell -m liste-outils.scm
```

et **partage ses deux fichiers** : `état-alice.scm` et `liste-outils.scm`.

Carole génère le même environnement **à partir des deux fichiers** d'Alice,

```
guix time-machine -C état-alice.scm -- shell -m liste-outils.scm
```

collaborer = partager un environnement computationnel ⇒ partager un graphe

Alice décrit son environnement :

- la liste des outils avec le fichier `liste-outils.scm`
- la révision (Guix lui-même et aussi potentiellement les autres canaux) :

```
guix describe -f channels > état-alice.scm
```

génère son environnement avec, e.g.,

```
guix shell -m liste-outils.scm
```

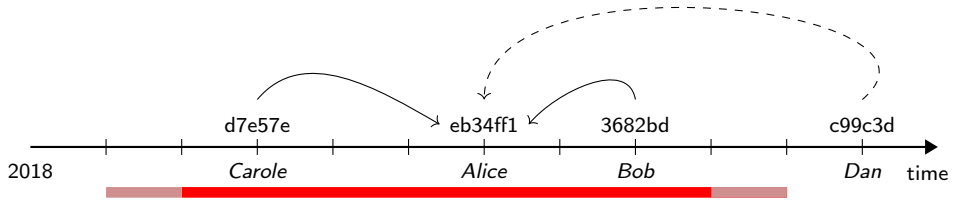
et **partage ses deux fichiers** : `état-alice.scm` et `liste-outils.scm`.

Carole génère le même environnement **à partir des deux fichiers** d'Alice,

```
guix time-machine -C état-alice.scm -- shell -m liste-outils.scm
```

Dan peut donc aussi avoir le même environnement qu'Alice et Carole.

Reproductibilité en arrière, en avant : `guix time-machine`



Pour être reproductible dans le temps, il faut :

- Une préservation de **tous** les codes source ($\approx$ 75% archivés (lien) dans Software Heritage (lien))
- Une *backward* compatibilité du noyau Linux
- Une compatibilité du *hardware* (p. ex. CPU, disque dur (NVMe), etc.)

Quelle est la taille de la fenêtre temporelle avec les 3 conditions satisfaites ?

(À ma connaissance, le projet Guix réalise une expérimentation grandeur nature et quasi-unique depuis sa v1.0 en 2019)

Au final

comment refaire plus tard et là-bas ce qui a été fait aujourd'hui et ici ?

traçabilité et transparence

être capable d'étudier bogue-à-bogue

Guix devrait s'occuper de tout

```
guix time-machine -C channels.scm -- cmd -m manifest.scm
```

si on spécifie

« comment construire » `channels.scm`

« quoi construire » `manifest.scm`

. . . Guix est un *continuum*

(sur n'importe quelle distribution Linux)

un **gestionnaire de paquets** déclaratif	`guix package` (`-m` *manifest*)
temporairement étendu à la volée	`guix shell` (`--container`)
maîtrisant exactement l'*état*	`guix time-machine` (`-C` *channels*)
qui produit des ***packs* distribuables**	`guix pack` (`-f docker`)
qui génèrent des ***machines virtuelles* isolées**	`guix sytem vm`
et aussi une bibliothèque Scheme	`guix repl` (*extensions*)
(la distribution Linux elle-même	`config.scm` (Guix System))

Guix permet un contrôle fin du graphe de configuration sous-jacent

`guix time-machine -C état.scm -- `*commande options* `une-config.scm`

`une-config.scm` est **reproductible** d'une machine à l'autre et dans le temps

En production

Grid'5000		828-nodes	(12,000+ cores, 31 clusters)	(France)
GliCID (CCIPL)	Nantes	392-nodes	(7500+ cores)	(France)
PlaFrIM Inria	Bordeaux	120-nodes	(3000+ cores)	(France)
GriCAD	Grenoble	72-nodes	(1000+ cores)	(France)
Max Delbrück Center	Berlin	250-nodes	+ workstations	(Allemagne)
UMC	Utrecht	68-nodes	(1000+ cores)	(Pays-Bas)
UTHSC Pangenome		11-nodes	(264 cores)	(USA)

(le vôtre ?)

`https://hpc.guix.info`

Toward practical transparent verifiable and long-term reproducible research using Guix (lien)

Des questions ?

guix-science@gnu.org

https://hpc.guix.info/events/2022/café-guix/

Cette présentation est archivée.
(Software Heritage id swh:1:dir:80ed9280e64392434b9c3dd8d4577161295c87cc)

Gestion déclarative (`guix shell python python-numpy --export-manifest`)

déclaratif = fichier de configuration

Un fichier `some-python.scm` peut contenir cette déclaration :

```
(specifications->manifest
 (list
  "python"
  "python-numpy"))
```

```
guix package --manifest=some-python.scm
```

équivalent à

```
guix install python python-numpy
```

Gestion *déclarative* : remarques

Version ? Nous le verrons dans la suite

Langage ? *Domain-Specific Language* (DSL) basé sur Scheme (« langage fonctionnel Lisp »)

- `(Oui (quand (= Lisp parenthèses) (baroque)))`
- Mais **continuum** :
  1. configuration (`manifest`)
  2. définition des paquets (ou services)
  3. extension
  4. le cœur est écrit aussi en Scheme

Guix est **adaptable** à ses besoins

Déclaratif vs Impératif (et non pas Donnée inerte vs Programme)

Programmation déclarative = programmation fonctionnelle ou descriptive (LaTeX) ou logique (Prolog)

Gestion déclarative : exemple de transformation (machine de Goldberg :-) (lien))

```
(define python "python")

(specifications->manifest
 (append
  (list python)
  (map (lambda (pkg)
         (string-append python "-" pkg))
       (list
        "matplotlib"
        "numpy"
        "scipy"))))
```

Guix DSL, *variables*, Scheme et chaîne de caractères.

Exemple état.scm (guix describe -f channels-sans-intro)

```
(list (channel
        (name 'guix)
        (url "https://git.savannah.gnu.org/git/guix.git")
        (branch "master")
        (commit
          "00ff6f7c399670a76efffb91276dea2633cc130c")))
```