Mon code python sur PLMLab : du script au paquet installable

Séminaire Infomath, Paris, 4 avril 2024

Matthieu Boileau

Sources | Diaporama HTML | Diaporama PDF

Situation banale d'un laboratoire de mathématiques

Pour résoudre un problème, un chercheur a écrit un script python, généralement sous forme de notebook Jupyter. Il souhaite :

- en faire une publication,
- le développer en équipe,
- le mettre à disposition de collaborateurs.

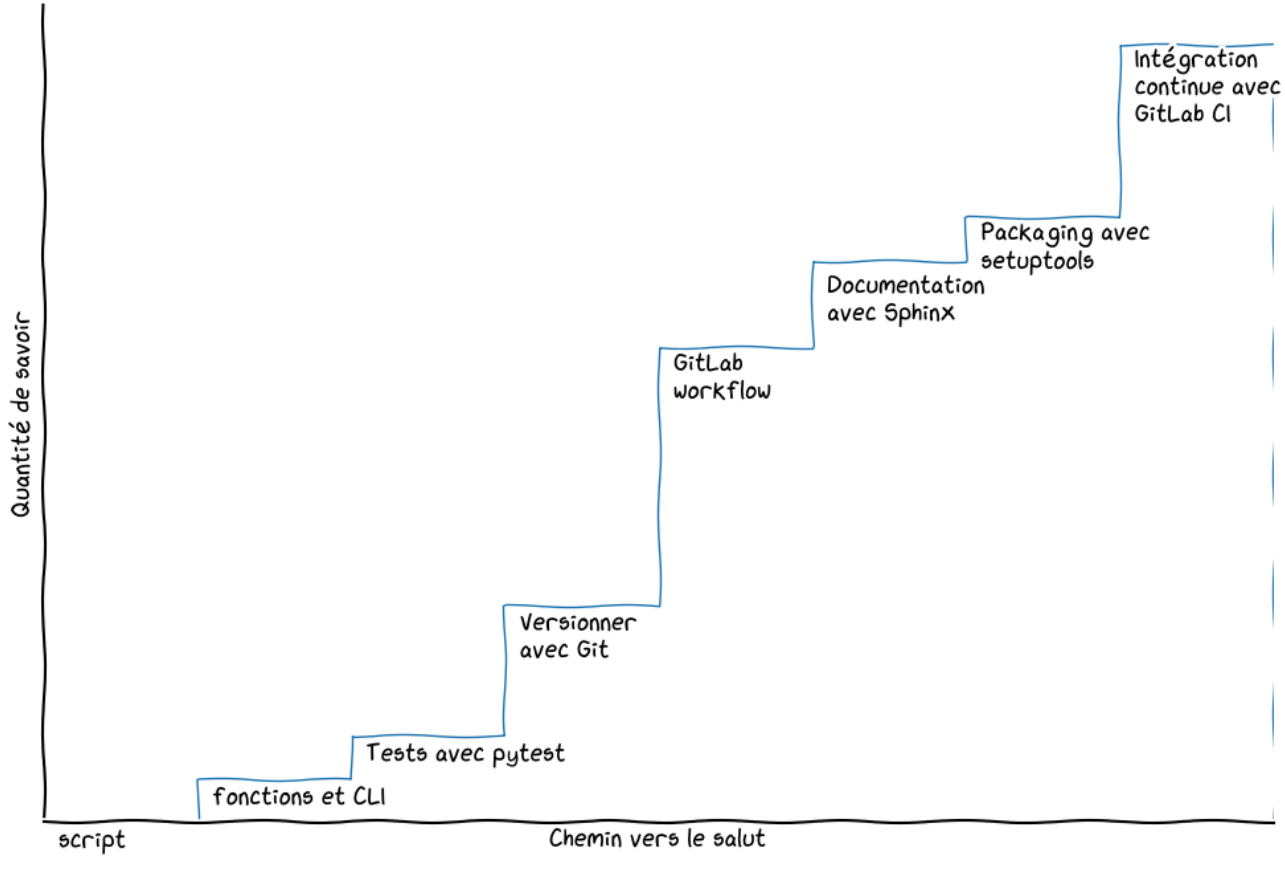
Ce script contient plein d'idées géniales, mais...

- il est monolithique : aucune modularité
- il mélange code et données :
    - si on veut changer les données d'entrée, il faut changer le code donc multiplier les versions
    - les données produites se retrouvent dans les sources du code
- il n'est validé que par son auteur : pas de relecture, pas de tests, pas de documentation

Cet exposé propose un chemin qui mène du script à un paquet python testé, documenté, installable et publié.

L'escalier de la compétence

Ce chemin peut être vu comme un escalier où chaque marche correspond à une compétence à acquérir :



Exemple : un notebook qui calcule la propagation d'une onde linéaire en 1D

linewave.ipynb

Première étape : découpage en fonctions et CLI

In [1]:
```
from IPython.display import Code
Code(filename='linewave/linewave.py')
```

Out[1]:

```
"""Solve the 1D wave equation using the leap-frog scheme"""
import argparse

import numpy as np
import matplotlib.pyplot as plt


c = 1.0  # Wave speed


def sinus(x, t):
    """Return the analytical solution of the 1D wave equation"""
    return np.sin(2 * np.pi * (x - c * t))


def compute_wave(L=1.0, T=100.0, CFL=0.99, N=40):
    """Compute the solution of the 1D wave equation using the leap-fro
g scheme"""
    # Discretization
    x, dx = np.linspace(0, L, N, endpoint=False, retstep=True)
    dt = CFL * dx / c  # Time step

    # Initialize arrays
    un = np.empty_like(x)
    unm1 = np.empty_like(x)
    unp1 = np.empty_like(x)

    # Set initial solution
    un = sinus(x, 0.0)
    unm1 = sinus(x, -dt)

    # Leap-frog scheme

    t = 0.0
    while t < T:
        t += dt
        unp1 = -unm1 + 2 * un + CFL**2 * (np.roll(un, 1) - 2 * un + n
```

```
p.roll(un, -1))
        # Exchange array references for avoiding a copy
        unm1, un, unp1 = un, unp1, unm1

    return t, x, un


def L2_error(t, x, u):
    """Compute the L2 error norm"""
    return np.linalg.norm(u - sinus(x, t)) / np.linalg.norm(sinus(x,
t))


def plot(t, x, u):
    """Plot the solution"""
    plt.plot(x, u, "o", label=f"t = {t:.2f}")
    plt.plot(x, sinus(x, t), label="Analytical")
    plt.title(f"Leap-frog solution with N = {len(x)}")
    plt.xlabel("x")
    plt.ylabel("u")
    plt.legend()
```

```
rid points")
    args = parser.parse_args()
    t, x, u = compute_wave(**vars(args))
    plot(t, x, u)
    print(f"L2 error norm: {L2_error(t, x, u):.3e}")


if __name__ == "__main__":
    main()
```

On peut désormais exécuter le script en ligne de commande. On affiche l'aide :

In [2]:
```
%run linewave/linewave.py --help
```

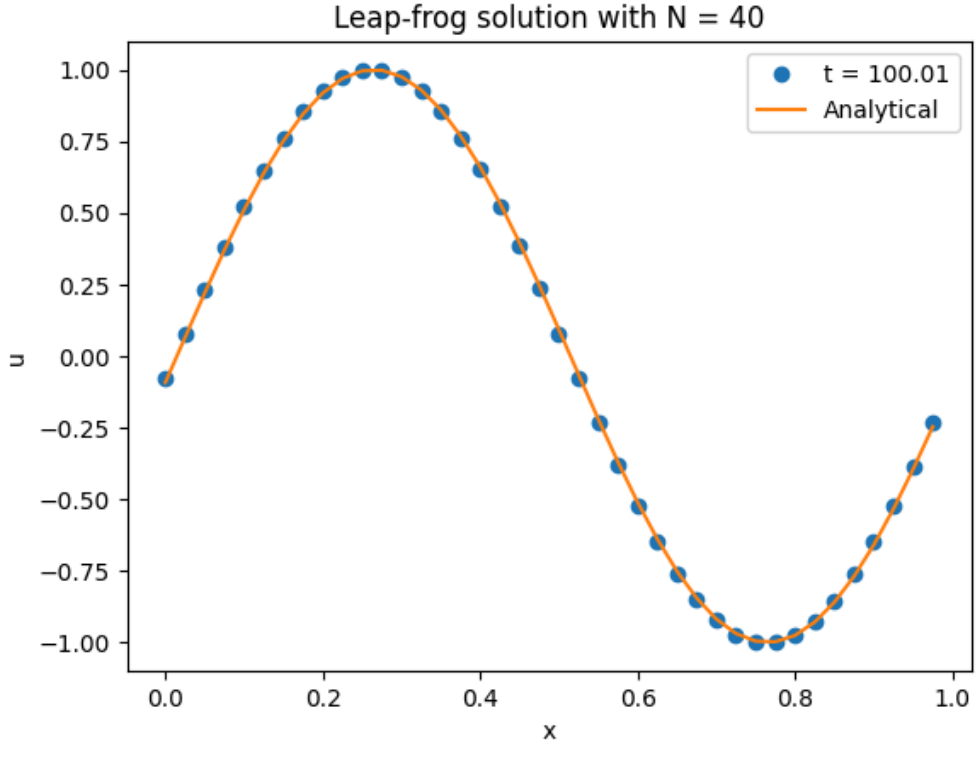
```
usage: linewave.py [-h] [--L L] [--T T] [--CFL CFL] [--N N]

Solve the 1D wave equation using the leap-frog scheme

options:
  -h, --help  show this help message and exit
  --L L       Length of the domain (default: 1.0)
  --T T       Final time (default: 100.0)
  --CFL CFL   CFL number (default: 0.99)
  --N N       Number of grid points (default: 40)
```

Et l'exécuter avec ses valeurs par défaut :

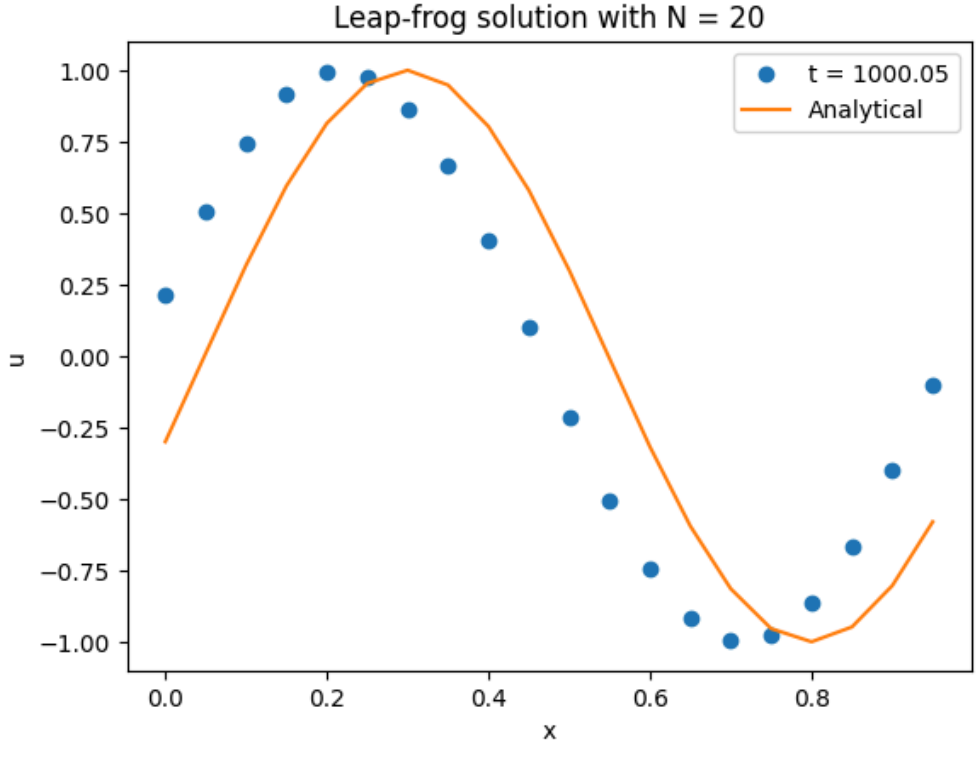
In [3]:
```
%run linewave/linewave.py
```




```
L2 error norm: 1.289e-02
```

Ou avec d'autres paramètres :

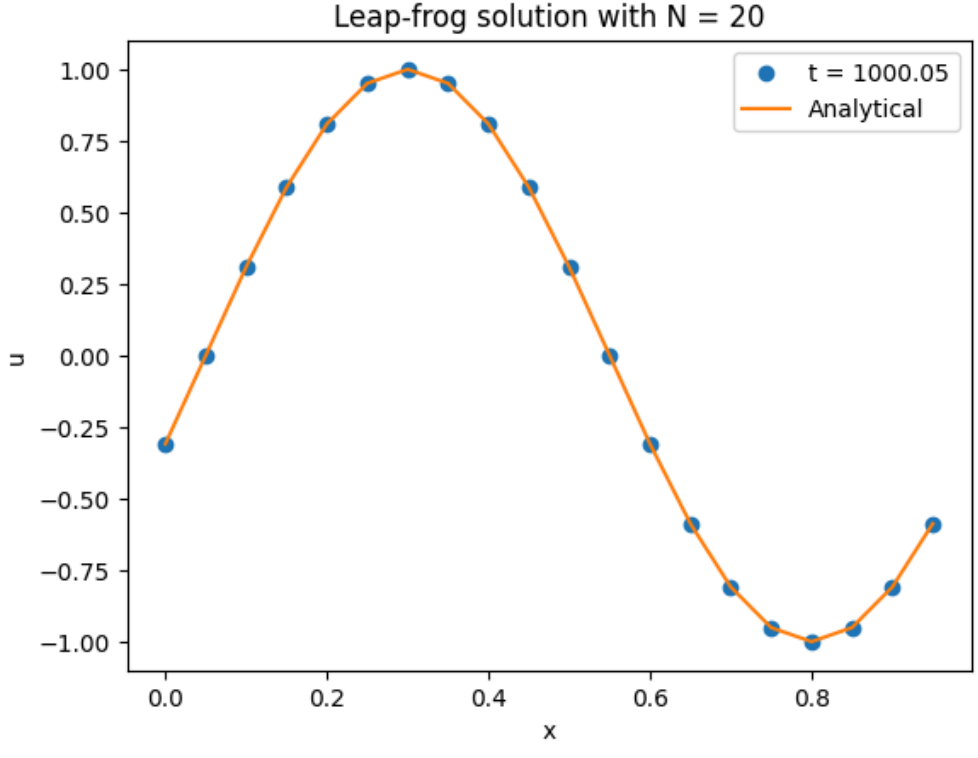
In [4]:
```
%run linewave/linewave.py --T 1000 --N 20
```




```
L2 error norm: 5.134e-01
```

Dans cette configuration 1D, la méthode est exacte pour CFL = 1 :

In [5]:
```
%run linewave/linewave.py --T 1000 --N 20 --CFL 1.
```




```
L2 error norm: 2.273e-09
```

Tests unitaires avec pytest

On se contente de tester la fonction `compute_wave()` en écrivant un fichier nommé `test_linewave.py` :

In [6]:
```
Code(filename='test_linewave.py')
```

Out[6]:
```
from linewave import linewave


def test_linewave():
    t, x, u = linewave.compute_wave(T=50, N=80, L=2.)
    assert x.max() == 2. - 2. / 80
    assert t >= 50
    assert linewave.L2_error(t, x, u) < 0.01
```

In [7]:
```
!pytest test_linewave.py
```

```
============================= test session starts ==============
================
platform darwin -- Python 3.11.7, pytest-8.1.1, pluggy-1.4.0
rootdir: /Users/boileau/Documents/Conf/2024/Infomath/script2paqu
et
plugins: anyio-4.3.0
collected 1 item

test_linewave.py .
[100%]

============================== 1 passed in 0.21s ===============
================
```

GitLab & Co

À ce stade, on prend un **ÉNORME** raccourci : on utilise cookiecutter pour créer un projet gitlab avec un squelette de paquet python :

```
cookiecutter https://plmlab.math.cnrs.fr/mboileau/cookiecutter.git --
directory python
```

Cette commande :

- crée un projet python contenant :
    - les sources,
    - les tests,
    - la documentation Sphinx,
    - un fichier `LICENSE`,
    - un fichier `README.md`,
    - un fichier `CHANGELOG.md`,
    - le fichier `pyproject.toml` qui décrit le projet python,
    - le fichier `.gitlab-ci.yml` qui décrit la pipeline d'intégration continue
- publie ce projet sur gitlab à l'adresse https://plmlab.math.cnrs.fr/mboileau/linewave

On copie notre code dans le squelette :

In [ ]:
```
!cp linewave/linewave.py ../linewave/src/linewave/
!cp test_linewave.py ../linewave/tests/
```

Ce qu'il reste à faire

- installer localement en mode éditable
- vérifier que les tests passent
- construire la documentation automatique
- pousser la nouvelle version sur gitlab.

Installation

En mode éditable depuis la racine des sources :

```
pip install -e .
```

On vérifie la version installée :

```
python -c 'import linewave;print(linewave.__version__)'
```

Exécution des tests unitaires

On installe les dépendances des tests :

```
pip install -e ".[test]"
```

On les exécute avec la commande :

```
pytest
```

Doc avec sphinx

Sphinx est le générateur natif de documentation de python.

Notre template cookiecutter contient les fichiers essentiels de sphinx de :

```
docs
├── Makefile  # pour construire sous Linux
├── make.bat  # pour construire sous Windows
└── source
    ├── conf.py  # la configuration de sphinx
    ├── index.md  # la page d'accueil de la doc
    └── installation.md  # une page d'installation
```

On installe les dépendances de documentation avec

```
pip install -e ".[doc]"
```

On construit la doc avec :

```
sphinx-build -b html docs/source/ docs/_build/
```

On ouvre le fichier `docs/_build/index.html`.

Première release sur GitLab

Le travail en local est terminé :

✅ installer localement en mode éditable
✅ vérifier que les tests passent
✅ construire la documentation automatique

On pousse la nouvelle version sur gitlab. Sur GitLab, dans Code > Tags, on crée un nouveau tag nommé `v0.1.0` qui déclenche le job gitlab-ci produisant la release du même nom.

Conclusion

- On a vu comment passer d'un script python à un paquet testé et documenté en continu, et installable par exemple via la commande :

```
pip install
git+ssh://git@plmlab.math.cnrs.fr/mboileau/linewave.git@v0.1.0
```

Conclusion

- On a vu comment passer d'un script python à un paquet testé et documenté en continu, et installable par exemple via la commande :

```
pip install
git+ssh://git@plmlab.math.cnrs.fr/mboileau/linewave.git@v0.1.0
```

- Un grand nombre d'étapes peuvent être automatisées avec cookiecutter
- La quantité de code ajouté est assez faible

MAIS cette apparente facilité peut être trompeuse car elle nécessite de maîtriser :

- le suivi de version avec git,
- l'environnement d'une forge gitlab,
- les principes du workflow gitlab-ci,
- les bases du packaging, des tests et de la documentation en python.

Les dernières marches à gravir

- référencer son code sur Software Heritage : facile !
- publier sur https://pypi.org/ : facile !
- rendre son code reproductible : difficile...

Une référence à retenir : https://learn.scientific-python.org/development/