PYAF

Une interface python pour AFF3CT

Romain Tajan

e-mail : romain.tajan@ims-bordeaux.fr

Plan

1 Introduction

2 Méthodologie

3 Évolutions de l'ergonomie

4 Nouvelle feuille de route

Bilan de la dernière journée

PYAF : une interface python pour AFF3CT

- philosophie : écrire du python ↦ exécuter du C++
- création et diffusion de modules personnalisés décrits en python/C++

Feuille de route

1. augmentation du volume de modules accessibles
2. amélioration de l'ergonomie
   - ▹ simplifier l'installation
   - ▹ simplifier l'importation
   - ▹ simplifier l'utilisation
3. amélioration de la gestion des modules personnalisés
4. amélioration du compilateur (boucles, conditions, control flow...)
5. amélioration de la gestion du GIL (Global Interpreter Lock)

Bilan de la dernière journée

PYAF : une interface python pour AFF3CT

- philosophie : écrire du python ↦ exécuter du C++
- création et diffusion de modules personnalisés décrits en python/C++

Feuille de route

1. augmentation du volume de modules accessibles
2. amélioration de l'ergonomie
   - ▹ simplifier l'installation
   - ▹ simplifier l'importation
   - ▹ simplifier l'utilisation
3. amélioration de la gestion des modules personnalisés
4. amélioration du compilateur (boucles, conditions, control flow...)
5. amélioration de la gestion du GIL (Global Interpreter Lock)

Bilan de la dernière journée

PYAF : une interface python pour AFF3CT

- philosophie : écrire du python ↦ exécuter du C++
- création et diffusion de modules personnalisés décrits en python/C++

Feuille de route

1. augmentation du volume de modules accessibles
2. amélioration de l'ergonomie
   - ▹ simplifier l'installation
   - ▹ simplifier l'importation
   - ▹ simplifier l'utilisation
3. amélioration de la gestion des modules personnalisés
4. amélioration du compilateur (boucles, conditions, control flow...)
5. amélioration de la gestion du GIL (Global Interpreter Lock)

Bilan de la dernière journée

PYAF : une interface python pour AFF3CT

- philosophie : écrire du python ↦ exécuter du C++
- création et diffusion de modules personnalisés décrits en python/C++

Feuille de route

1. augmentation du volume de modules accessibles
2. amélioration de l'ergonomie
   - simplifier l'installation
   - simplifier l'importation
   - simplifier l'utilisation
3. amélioration de la gestion des modules personnalisés
4. amélioration du compilateur (boucles, conditions, control flow...)
5. amélioration de la gestion du GIL (Global Interpreter Lock)

Bilan de la dernière journée

PYAF : une interface python pour AFF3CT

- philosophie : écrire du python ↦ exécuter du C++
- création et diffusion de modules personnalisés décrits en python/C++

Feuille de route

1. augmentation du volume de modules accessibles
2. amélioration de l'ergonomie
   - ▹ simplifier l'installation
   - ▹ simplifier l'importation
   - ▹ simplifier l'utilisation
3. amélioration de la gestion des modules personnalisés
4. amélioration du compilateur (boucles, conditions, control flow...)
5. amélioration de la gestion du GIL (Global Interpreter Lock)

Bilan de la dernière journée

PYAF : une interface python pour AFF3CT

- philosophie : écrire du python ↦ exécuter du C++
- création et diffusion de modules personnalisés décrits en python/C++

Feuille de route

1. augmentation du volume de modules accessibles
2. amélioration de l'ergonomie
   - simplifier l'installation
   - simplifier l'importation
   - simplifier l'utilisation
3. amélioration de la gestion des modules personnalisés
4. amélioration du compilateur (boucles, conditions, control flow...)
5. amélioration de la gestion du GIL (Global Interpreter Lock)

Bilan de la dernière journée

PYAF : une interface python pour AFF3CT

- philosophie : écrire du python ↦ exécuter du C++
- création et diffusion de modules personnalisés décrits en python/C++

Feuille de route

1. augmentation du volume de modules accessibles
2. amélioration de l'ergonomie
   - ▹ simplifier l'installation
   - ▹ simplifier l'importation
   - ▹ simplifier l'utilisation
3. amélioration de la gestion des modules personnalisés
4. amélioration du compilateur (boucles, conditions, control flow...)
5. amélioration de la gestion du GIL (Global Interpreter Lock)

Bilan de la dernière journée

PYAF : une interface python pour AFF3CT

- philosophie : écrire du python ↦ exécuter du C++
- création et diffusion de modules personnalisés décrits en python/C++

Feuille de route

1. augmentation du volume de modules accessibles
2. amélioration de l'ergonomie
   - ▹ simplifier l'installation
   - ▹ simplifier l'importation
   - ▹ simplifier l'utilisation
3. amélioration de la gestion des modules personnalisés
4. amélioration du compilateur (boucles, conditions, control flow...)
5. amélioration de la gestion du GIL (Global Interpreter Lock)

Bilan de la dernière journée

PYAF : une interface python pour AFF3CT

- philosophie : écrire du python ↦ exécuter du C++
- création et diffusion de modules personnalisés décrits en python/C++

Feuille de route

1. augmentation du volume de modules accessibles
2. amélioration de l'ergonomie
   - ▹ simplifier l'installation
   - ▹ simplifier l'importation
   - ▹ simplifier l'utilisation
3. amélioration de la gestion des modules personnalisés
4. amélioration du compilateur (boucles, conditions, control flow...)
5. amélioration de la gestion du GIL (Global Interpreter Lock)

Bilan de la dernière journée

PYAF : une interface python pour AFF3CT

- philosophie : écrire du python ↦ exécuter du C++
- création et diffusion de modules personnalisés décrits en python/C++

Feuille de route

1. augmentation du volume de modules accessibles
2. amélioration de l'ergonomie
   - ▹ simplifier l'installation
   - ▹ simplifier l'importation
   - ▹ simplifier l'utilisation
3. amélioration de la gestion des modules personnalisés
4. amélioration du compilateur (boucles, conditions, control flow...)
5. amélioration de la gestion du GIL (Global Interpreter Lock)

Bilan de la dernière journée

PYAF : une interface python pour AFF3CT

- philosophie : écrire du python ↦ exécuter du C++
- création et diffusion de modules personnalisés décrits en python/C++

Feuille de route

1. augmentation du volume de modules accessibles
2. amélioration de l'ergonomie
   - ▹ simplifier l'installation
   - ▹ simplifier l'importation
   - ▹ simplifier l'utilisation
3. amélioration de la gestion des modules personnalisés
4. amélioration du compilateur (boucles, conditions, control flow...)
5. amélioration de la gestion du GIL (Global Interpreter Lock)

Bilan de la dernière journée

PYAF : une interface python pour AFF3CT

- philosophie : écrire du python ↦ exécuter du C++
- création et diffusion de modules personnalisés décrits en python/C++

Feuille de route

1. augmentation du volume de modules accessibles
2. amélioration de l'ergonomie
   - ▹ simplifier l'installation
   - ▹ simplifier l'importation
   - ▹ simplifier l'utilisation
3. amélioration de la gestion des modules personnalisés
4. amélioration du compilateur (boucles, conditions, control flow...)
5. amélioration de la gestion du GIL (Global Interpreter Lock)

Bilan de la dernière journée

PYAF : une interface python pour AFF3CT

- philosophie : écrire du python ↦ exécuter du C++
- création et diffusion de modules personnalisés décrits en python/C++

Feuille de route

1. augmentation du volume de modules accessibles
2. amélioration de l'ergonomie
   - ▹ simplifier l'installation
   - ▹ simplifier l'importation
   - ▹ simplifier l'utilisation
3. amélioration de la gestion des modules personnalisés
4. amélioration du compilateur (boucles, conditions, control flow...)
5. amélioration de la gestion du GIL (Global Interpreter Lock)

Bilan de la dernière journée

PYAF : une interface python pour AFF3CT

- philosophie : écrire du python ↦ exécuter du C++
- création et diffusion de modules personnalisés décrits en python/C++

Feuille de route

1. augmentation du volume de modules accessibles
2. amélioration de l'ergonomie
   - ▹ simplifier l'installation
   - ▹ simplifier l'importation
   - ▹ simplifier l'utilisation
3. amélioration de la gestion des modules personnalisés
4. amélioration du compilateur (boucles, conditions, control flow...)
5. amélioration de la gestion du GIL (Global Interpreter Lock)

Bilan de la dernière journée

PYAF : une interface python pour AFF3CT

- philosophie : écrire du python ↦ exécuter du C++
- création et diffusion de modules personnalisés décrits en python/C++

Feuille de route

1. augmentation du volume de modules accessibles (AFF3CT-core)
2. amélioration de l'ergonomie
   - ▹ simplifier l'installation
   - ▹ simplifier l'importation
   - ▹ simplifier l'utilisation
3. amélioration de la gestion des modules personnalisés
4. amélioration du compilateur (boucles, conditions, control flow...)
5. amélioration de la gestion du GIL (Global Interpreter Lock)

Objectifs de cette présentation

1. Retour illustré sur la méthodologie
2. Avancées concernant l'ergonomie
   - Se rapprocher de la syntaxe de python
   - Utilisation avec NumPy
   - Simplifier installation / importation
3. Présentation d'une nouvelle feuille de route

Plan

1 Introduction

2 Méthodologie

3 Évolutions de l'ergonomie

4 Nouvelle feuille de route

Méthodologie

pybind11

1. Mature : utilisée dans d'autres projets (PyTorch, GNU Radio,...)
2. Permet d'exposer des classes C++ à python (et vice/versa)
   - Conserve les liens d'héritage
   - Choix des attributs/méthodes exposées
   - Étendre des classes C++ avec des méthodes écrites en python

Exposer les classes d'AFF3CT :

1. `aff3ct::module::Module`, ou dérivés
2. `aff3ct::runtime::Task`
3. `aff3ct::runtime::Socket`
4. `aff3ct::runtime::Sequence`
5. `aff3ct::runtime::Pipeline`

Méthodologie

pybind11

1. Mature : utilisée dans d'autres projets (PyTorch, GNU Radio,...)
2. Permet d'exposer des classes C++ à python (et vice/versa)
   - Conserve les liens d'héritage
   - Choix des attributs/méthodes exposées
   - Étendre des classes C++ avec des méthodes écrites en python

Exposer les classes d'AFF3CT :

1. `aff3ct::module::Module`, ou dérivés
2. `aff3ct::runtime::Task`
3. `aff3ct::runtime::Socket`
4. `aff3ct::runtime::Sequence`
5. `aff3ct::runtime::Pipeline`

Méthodologie

pybind11

1. Mature : utilisée dans d'autres projets (PyTorch, GNU Radio,...)
2. Permet d'exposer des classes C++ à python (et vice/versa)
   - Conserve les liens d'héritage
   - Choix des attributs/méthodes exposées
   - Étendre des classes C++ avec des méthodes écrites en python

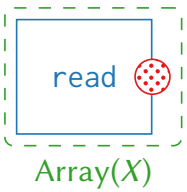
Exposer les classes d'AFF3CT :

1. `aff3ct::module::Module`, ou dérivés
2. `aff3ct::runtime::Task`
3. `aff3ct::runtime::Socket`
4. `aff3ct::runtime::Sequence`
5. `aff3ct::runtime::Pipeline`

```
import aff3ct
...
```

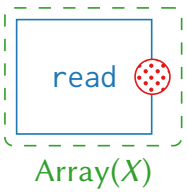
```
#include<aff3ct.hpp>
...
```

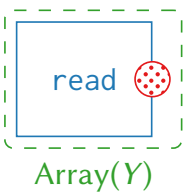
```python
import aff3ct
...

X     = Array_float32                    ([1,2,3])
```



```cpp
#include<aff3ct.hpp>
...

Array   <float                                  > X    ({1,2,3});
```

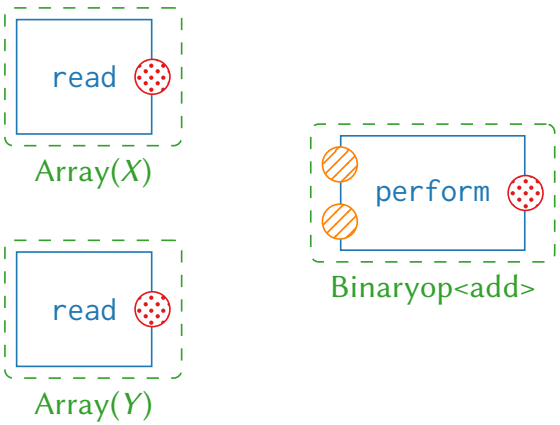
```
import aff3ct
...

X     = Array_float32                    ([1,2,3])
Y     = Array_float32                    ([4,5,6])
```





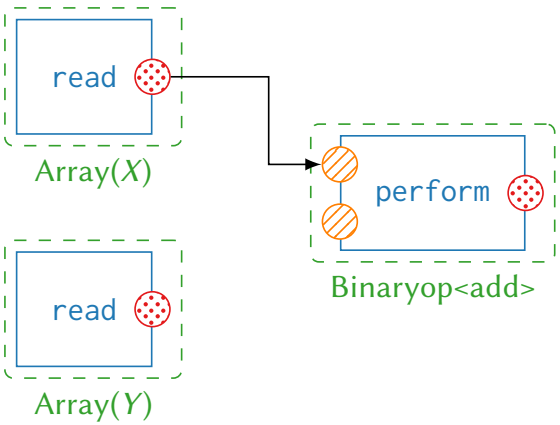
```
#include<aff3ct.hpp>
...

Array   <float                                > X    ({1,2,3});
Array   <float                                > Y    ({4,5,6});
```

```
import aff3ct
...

X    = Array_float32                ([1,2,3])
Y    = Array_float32                ([4,5,6])
plus = Binaryop_add_float32_float32(      3)
```

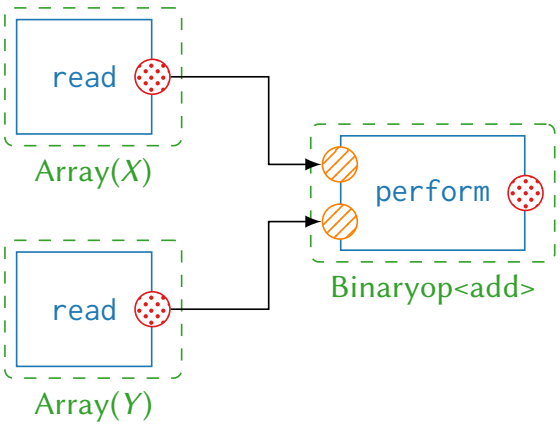



```
#include<aff3ct.hpp>
...

Array    <float                                > X    ({1,2,3});
Array    <float                                > Y    ({4,5,6});
Binaryop<float,float,bop_add<float,float>> plus(      3);
```

```
import aff3ct
...

X     = Array_float32                ([1,2,3])
Y     = Array_float32                ([4,5,6])
plus  = Binaryop_add_float32_float32(      3)

plus["perform::in0"] = X["read::data"]
```

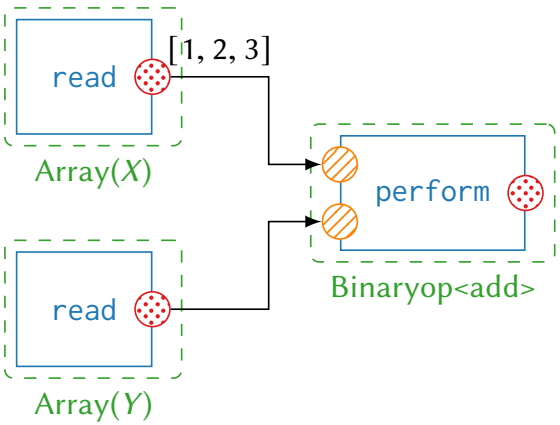


```
#include<aff3ct.hpp>
...

Array    <float                                > X    ({1,2,3});
Array    <float                                > Y    ({4,5,6});
Binaryop<float,float,bop_add<float,float>> plus(      3);

plus["perform::in0"] = X["read::data"];
```



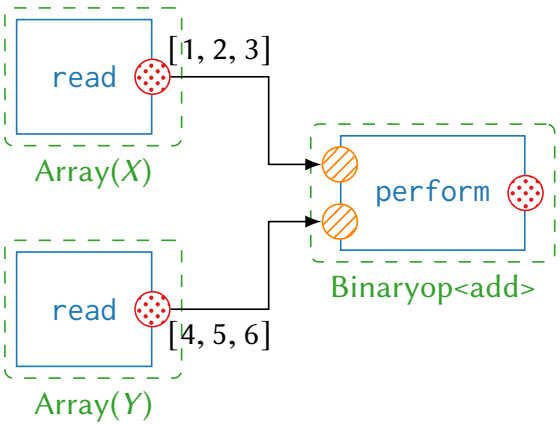

```python
import aff3ct
...

X    = Array_float32                ([1,2,3])
Y    = Array_float32                ([4,5,6])
plus = Binaryop_add_float32_float32(      3)

plus["perform::in0"] = X["read::data"]
plus["perform::in1"] = Y["read::data"]
```

```cpp
#include<aff3ct.hpp>
...

Array    <float                                > X    ({1,2,3});
Array    <float                                > Y    ({4,5,6});
Binaryop<float,float,bop_add<float,float>> plus(      3);

plus["perform::in0"] = X["read::data"];
plus["perform::in1"] = Y["read::data"];
```



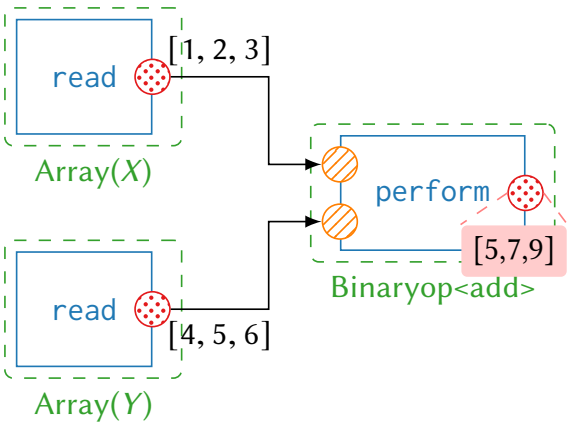

```python
import aff3ct
...

X    = Array_float32                ([1,2,3])
Y    = Array_float32                ([4,5,6])
plus = Binaryop_add_float32_float32(      3)

plus["perform::in0"] = X["read::data"]
plus["perform::in1"] = Y["read::data"]

X   ["read"   ].exec()
```

```cpp
#include<aff3ct.hpp>
...

Array    <float                                  > X    ({1,2,3});
Array    <float                                  > Y    ({4,5,6});
Binaryop<float,float,bop_add<float,float>> plus(      3);

plus["perform::in0"] = X["read::data"];
plus["perform::in1"] = Y["read::data"];

X   ("read"   ).exec();
```




```python
import aff3ct
...

X    = Array_float32                ([1,2,3])
Y    = Array_float32                ([4,5,6])
plus = Binaryop_add_float32_float32(      3)

plus["perform::in0"] = X["read::data"]
plus["perform::in1"] = Y["read::data"]

X   ["read"   ].exec()
Y   ["read"   ].exec()
```

```cpp
#include<aff3ct.hpp>
...

Array    <float                                  > X    ({1,2,3});
Array    <float                                  > Y    ({4,5,6});
Binaryop<float,float,bop_add<float,float>> plus(      3);

plus["perform::in0"] = X["read::data"];
plus["perform::in1"] = Y["read::data"];

X   ("read"   ).exec();
Y   ("read"   ).exec();
```




```python
import aff3ct
...

X    = Array_float32                ([1,2,3])
Y    = Array_float32                ([4,5,6])
plus = Binaryop_add_float32_float32(      3)

plus["perform::in0"] = X["read::data"]
plus["perform::in1"] = Y["read::data"]

X   ["read"   ].exec()
Y   ["read"   ].exec()
plus["perform"].exec()
```

```cpp
#include<aff3ct.hpp>
...

Array   <float                                  > X    ({1,2,3});
Array   <float                                  > Y    ({4,5,6});
Binaryop<float,float,bop_add<float,float>> plus(      3);

plus["perform::in0"] = X["read::data"];
plus["perform::in1"] = Y["read::data"];

X   ("read"   ).exec();
Y   ("read"   ).exec();
plus("perform").exec();
```

Plan

1 Introduction

2 Méthodologie

3 Évolutions de l'ergonomie

4 Nouvelle feuille de route

Ergonomie (socket) : état des lieux

1. DSEL AFF3CT ⇒ "orienté" sur les tâches
2. NumPy, PyTorch, . . . ⇒ "orientés" sur les données
   - même exemple, PyTorch → 4 lignes
   - PyTorch crée aussi le graphe des tâches (calcul de la rétro-propagation)
   - debug plus intuitif (pour un traiteur de signaux)

Utilisateur pourrait manipuler les sockets comme des données

Ergonomie (socket) : état des lieux

1. DSEL AFF3CT ⇒ "orienté" sur les tâches
2. NumPy, PyTorch, . . . ⇒ "orientés" sur les données
   - même exemple, PyTorch → 4 lignes
   - PyTorch crée aussi le graphe des tâches (calcul de la rétro-propagation)
   - debug plus intuitif (pour un traiteur de signaux)

Utilisateur pourrait manipuler les sockets comme des données

Ergonomie (socket) : état des lieux

1. DSEL AFF3CT ⇒ "orienté" sur les tâches
2. NumPy, PyTorch, ... ⇒ "orientés" sur les données
   - même exemple, PyTorch → 4 lignes
   - PyTorch crée aussi le graphe des tâches (calcul de la rétro-propagation)
   - debug plus intuitif (pour un traiteur de signaux)

Utilisateur pourrait manipuler les sockets comme des données

Ergonomie (socket) : état des lieux

1. DSEL AFF3CT ⇒ "orienté" sur les tâches
2. NumPy, PyTorch, ... ⇒ "orientés" sur les données
   - même exemple, PyTorch → 4 lignes
   - PyTorch crée aussi le graphe des tâches (calcul de la rétro-propagation)
   - debug plus intuitif (pour un traiteur de signaux)

Utilisateur pourrait manipuler les sockets comme des données

Ergonomie (socket) : état des lieux

1. DSEL AFF3CT ⇒ "orienté" sur les tâches
2. NumPy, PyTorch, ... ⇒ "orientés" sur les données
   - même exemple, PyTorch → 4 lignes
   - PyTorch crée aussi le graphe des tâches (calcul de la rétro-propagation)
   - debug plus intuitif (pour un traiteur de signaux)

Utilisateur pourrait manipuler les sockets comme des données

Ergonomie (socket) : état des lieux

1. DSEL AFF3CT ⇒ "orienté" sur les tâches
2. NumPy, PyTorch, ... ⇒ "orientés" sur les données
   - même exemple, PyTorch → 4 lignes
   - PyTorch crée aussi le graphe des tâches (calcul de la rétro-propagation)
   - debug plus intuitif (pour un traiteur de signaux)

Utilisateur pourrait manipuler les sockets comme des données

Ergonomie (socket) : actions menées

1. `x = aff3ct.array([1.,2.,3.], dtype=aff3ct.float32)`
   `y = aff3ct.array([4.,5.,6.], dtype=aff3ct.float32)`
   - un module `Array_<dtype>` instancié
   - la tâche `read` est exécutée
   - la socket `read::data` retournée à l'utilisateur
2. `z = x + y`
   Surcharge des opérateurs binaires Socket
   - `__add__` (`__sub__`, `__mul__`, `__le__` ...)
   - un module `Binaryop` instancié (type et taille déduits des sockets),
   - le bind est réalisé, la tâche `perform` est exécutée
   - la socket `perform::out` est retournée à l'utilisateur
3. Surcharge des opérateurs unaires `__abs__`, `__neg__`...
4. Surcharge des réductions `__sum__`, `__prod__`, ...
5. Surcharge de `__str__` et `__repr__` (affichage des données)
6. Surcharge des accès aux éléments/slices `__getitem__` (`s0 = s1[0::2]`), `__setitem__` (`s1[0::2] = s0`)
7. Ajout de méthode numpy : renvoie les données comme `np.array`

Ergonomie (socket) : actions menées

1. `x = aff3ct.array([1.,2.,3.], dtype=aff3ct.float32)`
   `y = aff3ct.array([4.,5.,6.], dtype=aff3ct.float32)`
   - un module `Array_<dtype>` instancié
   - la tâche `read` est exécutée
   - la socket `read::data` retournée à l'utilisateur
2. `z = x + y`
   Surcharge des opérateurs binaires Socket
   - `__add__` (`__sub__`, `__mul__`, `__le__` ...)
   - un module `Binaryop` instancié (type et taille déduits des sockets),
   - le bind est réalisé, la tâche `perform` est exécutée
   - la socket `perform::out` est retournée à l'utilisateur
3. Surcharge des opérateurs unaires `__abs__`, `__neg__`...
4. Surcharge des réductions `__sum__`, `__prod__`, ...
5. Surcharge de `__str__` et `__repr__` (affichage des données)
6. Surcharge des accès aux éléments/slices `__getitem__` (`s0 = s1[0::2]`), `__setitem__` (`s1[0::2] = s0`)
7. Ajout de méthode numpy : renvoie les données comme `np.array`

Ergonomie (socket) : actions menées

1. `x = aff3ct.array([1.,2.,3.], dtype=aff3ct.float32)`
   `y = aff3ct.array([4.,5.,6.], dtype=aff3ct.float32)`
   - un module `Array_<dtype>` instancié
   - la tâche `read` est exécutée
   - la socket `read::data` retournée à l'utilisateur
2. `z = x + y`
   Surcharge des opérateurs binaires Socket
   - `__add__` (`__sub__`, `__mul__`, `__le__` ...)
   - un module `Binaryop` instancié (type et taille déduits des sockets),
   - le bind est réalisé, la tâche `perform` est exécutée
   - la socket `perform::out` est retournée à l'utilisateur
3. Surcharge des opérateurs unaires `__abs__`, `__neg__`...
4. Surcharge des réductions `__sum__`, `__prod__`, ...
5. Surcharge de `__str__` et `__repr__` (affichage des données)
6. Surcharge des accès aux éléments/slices `__getitem__` (`s0 = s1[0::2]`), `__setitem__` (`s1[0::2] = s0`)
7. Ajout de méthode numpy : renvoie les données comme `np.array`

Ergonomie (socket) : actions menées

1. `x = aff3ct.array([1.,2.,3.], dtype=aff3ct.float32)`
   `y = aff3ct.array([4.,5.,6.], dtype=aff3ct.float32)`
   - un module `Array_<dtype>` instancié
   - la tâche `read` est exécutée
   - la socket `read::data` retournée à l'utilisateur
2. `z = x + y`
   Surcharge des opérateurs binaires Socket
   - `__add__` (`__sub__`, `__mul__`, `__le__` ...)
   - un module `Binaryop` instancié (type et taille déduits des sockets),
   - le bind est réalisé, la tâche `perform` est exécutée
   - la socket `perform::out` est retournée à l'utilisateur
3. Surcharge des opérateurs unaires `__abs__`, `__neg__`...
4. Surcharge des réductions `__sum__`, `__prod__`, ...
5. Surcharge de `__str__` et `__repr__` (affichage des données)
6. Surcharge des accès aux éléments/slices `__getitem__` (`s0 = s1[0::2]`), `__setitem__` (`s1[0::2] = s0`)
7. Ajout de méthode numpy : renvoie les données comme `np.array`

Ergonomie (socket) : actions menées

1. `x = aff3ct.array([1.,2.,3.], dtype=aff3ct.float32)`
   `y = aff3ct.array([4.,5.,6.], dtype=aff3ct.float32)`
   - un module `Array_<dtype>` instancié
   - la tâche `read` est exécutée
   - la socket `read::data` retournée à l'utilisateur
2. `z = x + y`
   Surcharge des opérateurs binaires Socket
   - `__add__` (`__sub__`, `__mul__`, `__le__` ...)
   - un module `Binaryop` instancié (type et taille déduits des sockets),
   - le bind est réalisé, la tâche `perform` est exécutée
   - la socket `perform::out` est retournée à l'utilisateur
3. Surcharge des opérateurs unaires `__abs__`, `__neg__`...
4. Surcharge des réductions `__sum__`, `__prod__`, ...
5. Surcharge de `__str__` et `__repr__` (affichage des données)
6. Surcharge des accès aux éléments/slices `__getitem__` (`s0 = s1[0::2]`), `__setitem__` (`s1[0::2] = s0`)
7. Ajout de méthode numpy : renvoie les données comme `np.array`

Ergonomie (socket) : actions menées

1. `x = aff3ct.array([1.,2.,3.], dtype=aff3ct.float32)`
   `y = aff3ct.array([4.,5.,6.], dtype=aff3ct.float32)`
   - un module `Array_<dtype>` instancié
   - la tâche `read` est exécutée
   - la socket `read::data` retournée à l'utilisateur
2. `z = x + y`
   Surcharge des opérateurs binaires Socket
   - `__add__` (`__sub__`, `__mul__`, `__le__` ...)
   - un module `Binaryop` instancié (type et taille déduits des sockets),
   - le bind est réalisé, la tâche `perform` est exécutée
   - la socket `perform::out` est retournée à l'utilisateur
3. Surcharge des opérateurs unaires `__abs__`, `__neg__`...
4. Surcharge des réductions `__sum__`, `__prod__`, ...
5. Surcharge de `__str__` et `__repr__` (affichage des données)
6. Surcharge des accès aux éléments/slices `__getitem__` (`s0 = s1[0::2]`), `__setitem__` (`s1[0::2] = s0`)
7. Ajout de méthode numpy : renvoie les données comme `np.array`

Ergonomie (socket) : actions menées

1. `x = aff3ct.array([1.,2.,3.], dtype=aff3ct.float32)`
   `y = aff3ct.array([4.,5.,6.], dtype=aff3ct.float32)`
   - un module `Array_<dtype>` instancié
   - la tâche `read` est exécutée
   - la socket `read::data` retournée à l'utilisateur
2. `z = x + y`
   Surcharge des opérateurs binaires Socket
   - `__add__` (`__sub__`, `__mul__`, `__le__` ...)
   - un module `Binaryop` instancié (type et taille déduits des sockets),
   - le bind est réalisé, la tâche `perform` est exécutée
   - la socket `perform::out` est retournée à l'utilisateur
3. Surcharge des opérateurs unaires `__abs__`, `__neg__`...
4. Surcharge des réductions `__sum__`, `__prod__`, ...
5. Surcharge de `__str__` et `__repr__` (affichage des données)
6. Surcharge des accès aux éléments/slices `__getitem__` (`s0 = s1[0::2]`), `__setitem__` (`s1[0::2] = s0`)
7. Ajout de méthode numpy : renvoie les données comme `np.array`

Ergonomie (socket) : actions menées

1. `x = aff3ct.array([1.,2.,3.], dtype=aff3ct.float32)`
   `y = aff3ct.array([4.,5.,6.], dtype=aff3ct.float32)`
   - un module `Array_<dtype>` instancié
   - la tâche `read` est exécutée
   - la socket `read::data` retournée à l'utilisateur
2. `z = x + y`
   Surcharge des opérateurs binaires Socket
   - `__add__` (`__sub__`, `__mul__`, `__le__` ...)
   - un module `Binaryop` instancié (type et taille déduits des sockets),
   - le bind est réalisé, la tâche `perform` est exécutée
   - la socket `perform::out` est retournée à l'utilisateur
3. Surcharge des opérateurs unaires `__abs__`, `__neg__`...
4. Surcharge des réductions `__sum__`, `__prod__`, ...
5. Surcharge de `__str__` et `__repr__` (affichage des données)
6. Surcharge des accès aux éléments/slices `__getitem__` (`s0 = s1[0::2]`), `__setitem__` (`s1[0::2] = s0`)
7. Ajout de méthode numpy : renvoie les données comme `np.array`

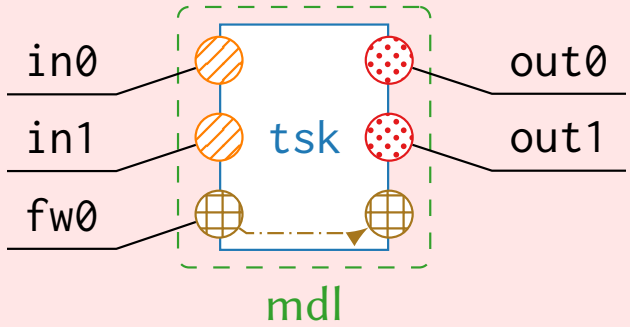
Ergonomie (socket) : actions menées

1. `x = aff3ct.array([1.,2.,3.], dtype=aff3ct.float32)`
   `y = aff3ct.array([4.,5.,6.], dtype=aff3ct.float32)`
   - un module `Array_<dtype>` instancié
   - la tâche `read` est exécutée
   - la socket `read::data` retournée à l'utilisateur
2. `z = x + y`
   Surcharge des opérateurs binaires Socket
   - `__add__` (`__sub__`, `__mul__`, `__le__` ...)
   - un module `Binaryop` instancié (type et taille déduits des sockets),
   - le bind est réalisé, la tâche `perform` est exécutée
   - la socket `perform::out` est retournée à l'utilisateur
3. Surcharge des opérateurs unaires `__abs__`, `__neg__`...
4. Surcharge des réductions `__sum__`, `__prod__`, ...
5. Surcharge de `__str__` et `__repr__` (affichage des données)
6. Surcharge des accès aux éléments/slices `__getitem__ (s0 = s1[0::2])`, `__setitem__ (s1[0::2] = s0)`
7. Ajout de méthode `numpy` : renvoie les données comme `np.array`

Ergonomie (socket) : actions menées

1. `x = aff3ct.array([1.,2.,3.], dtype=aff3ct.float32)`
   `y = aff3ct.array([4.,5.,6.], dtype=aff3ct.float32)`
   - un module `Array_<dtype>` instancié
   - la tâche `read` est exécutée
   - la socket `read::data` retournée à l'utilisateur
2. `z = x + y`
   Surcharge des opérateurs binaires Socket
   - `__add__` (`__sub__`, `__mul__`, `__le__` ...)
   - un module `Binaryop` instancié (type et taille déduits des sockets),
   - le bind est réalisé, la tâche `perform` est exécutée
   - la socket `perform::out` est retournée à l'utilisateur
3. Surcharge des opérateurs unaires `__abs__`, `__neg__`...
4. Surcharge des réductions `__sum__`, `__prod__`, ...
5. Surcharge de `__str__` et `__repr__` (affichage des données)
6. Surcharge des accès aux éléments/slices `__getitem__` (`s0 = s1[0::2]`), `__setitem__` (`s1[0::2] = s0`)
7. Ajout de méthode numpy : renvoie les données comme `np.array`

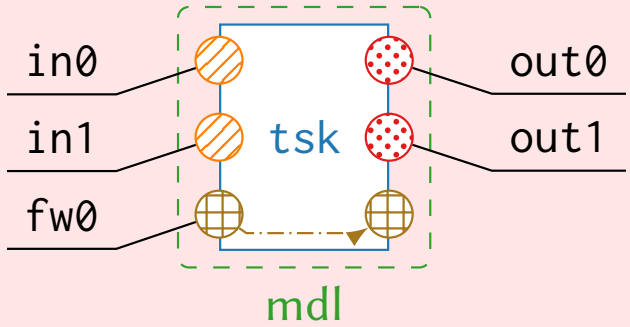
Ergonomie (tâches)



Utilisateur pourrait manipuler les sockets comme des méthodes.

1. `y0, y1 = mdl['tsk'](x0, x1, fw0 = x2)`
   Surcharge de `__call__`
   - `x0`, `x1`, `x2` sont des sockets de sorties (obtenues par ailleurs),
   - on relie `x0` $\mapsto$ `tsk::in0`, `x1` $\mapsto$ `tsk::in1` et `x2` $\mapsto$ `tsk::fw0`
   - la tâche `tsk` est exécutée
   - les sockets `tsk::out1` et `tsk::out2` retournées à l'utilisateur dans `y0` et `y1`
2. Appeler `mdl["tsk"]` avec `x0`...`x2` des `np.ndarray` ? ✓
3. Idem mais `y0` et `y1` aussi `np.ndarray` ? ✓
   - `y0, y1 = mdl['tsk'](x0, x1, fw0 = x2, raw_data=True)`

Ergonomie (tâches)



Utilisateur pourrait manipuler les sockets comme des méthodes.

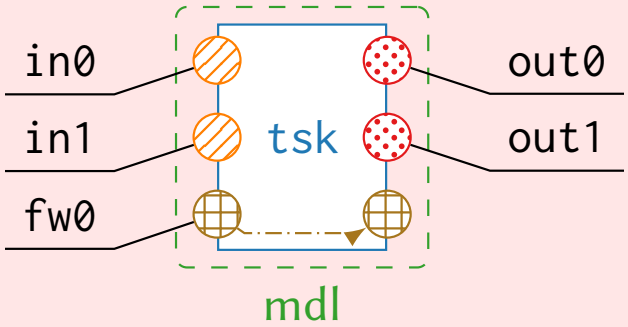
1. `y0, y1 = mdl['tsk'](x0, x1, fw0 = x2)`

   Surcharge de `__call__`

   - `x0`, `x1`, `x2` sont des sockets de sorties (obtenues par ailleurs),
   - on relie `x0` $\mapsto$ `tsk::in0`, `x1` $\mapsto$ `tsk::in1` et `x2` $\mapsto$ `tsk::fw0`
   - la tâche `tsk` est exécutée
   - les sockets `tsk::out1` et `tsk::out2` retournées à l'utilisateur dans `y0` et `y1`

2. Appeler `mdl["tsk"]` avec `x0`...`x2` des `np.ndarray` ? ✓
3. Idem mais `y0` et `y1` aussi `np.ndarray` ? ✓
   - `y0, y1 = mdl['tsk'](x0, x1, fw0 = x2, raw_data=True)`

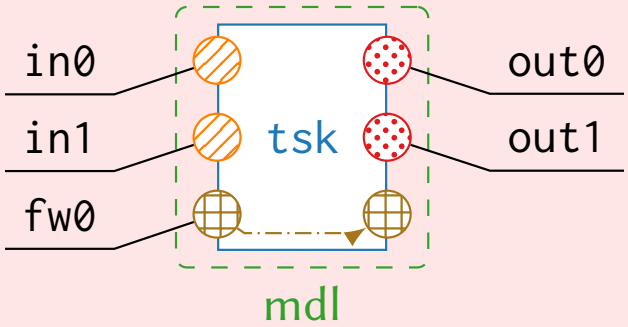
Ergonomie (tâches)



Utilisateur pourrait manipuler les sockets comme des méthodes.

1. `y0, y1 = mdl['tsk'](x0, x1, fw0 = x2)`
   Surcharge de `__call__`
   - `x0`, `x1`, `x2` sont des sockets de sorties (obtenues par ailleurs),
   - on relie `x0` $\mapsto$ `tsk::in0`, `x1` $\mapsto$ `tsk::in1` et `x2` $\mapsto$ `tsk::fw0`
   - la tâche `tsk` est exécutée
   - les sockets `tsk::out1` et `tsk::out2` retournées à l'utilisateur dans `y0` et `y1`
2. Appeler `mdl["tsk"]` avec `x0`...`x2` des `np.ndarray` ? ✓
3. Idem mais `y0` et `y1` aussi `np.ndarray` ? ✓
   - `y0, y1 = mdl['tsk'](x0, x1, fw0 = x2, raw_data=True)`

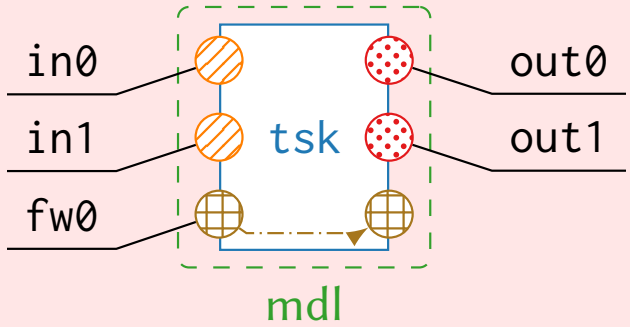
Ergonomie (tâches)



Utilisateur pourrait manipuler les sockets comme des méthodes.

1. `y0, y1 = mdl['tsk'](x0, x1, fw0 = x2)`
   Surcharge de `__call__`
   - `x0`, `x1`, `x2` sont des sockets de sorties (obtenues par ailleurs),
   - on relie `x0` ↦ `tsk::in0`, `x1` ↦ `tsk::in1` et `x2` ↦ `tsk::fw0`
   - la tâche `tsk` est exécutée
   - les sockets `tsk::out1` et `tsk::out2` retournées à l'utilisateur dans `y0` et `y1`
2. Appeler `mdl["tsk"]` avec `x0`...`x2` des `np.ndarray` ? ✓
3. Idem mais `y0` et `y1` aussi `np.ndarray` ? ✓
   - `y0, y1 = mdl['tsk'](x0, x1, fw0 = x2, raw_data=True)`

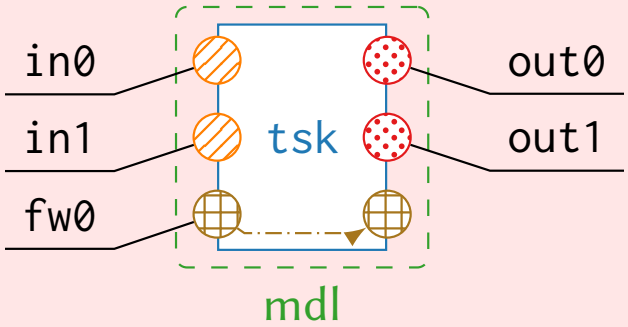
Ergonomie (tâches)



Utilisateur pourrait manipuler les sockets comme des méthodes.

1. `y0, y1 = mdl['tsk'](x0, x1, fw0 = x2)`
   Surcharge de `__call__`
   - `x0`, `x1`, `x2` sont des sockets de sorties (obtenues par ailleurs),
   - on relie `x0` $\mapsto$ `tsk::in0`, `x1` $\mapsto$ `tsk::in1` et `x2` $\mapsto$ `tsk::fw0`
   - la tâche `tsk` est exécutée
   - les sockets `tsk::out1` et `tsk::out2` retournées à l'utilisateur dans `y0` et `y1`
2. Appeler `mdl["tsk"]` avec `x0`...`x2` des `np.ndarray` ? ✓
3. Idem mais `y0` et `y1` aussi `np.ndarray` ? ✓
   - `y0, y1 = mdl['tsk'](x0, x1, fw0 = x2, raw_data=True)`

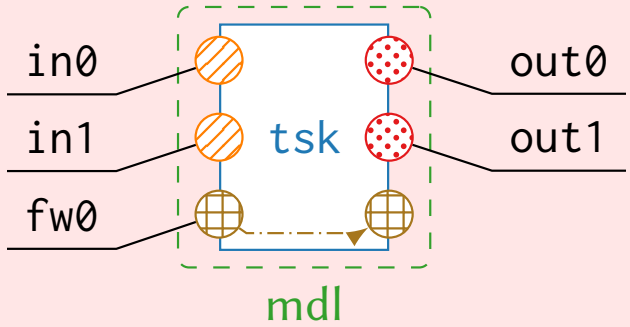
Ergonomie (tâches)



Utilisateur pourrait manipuler les sockets comme des méthodes.

1. `y0, y1 = mdl['tsk'](x0, x1, fw0 = x2)`
   Surcharge de `__call__`
   - `x0`, `x1`, `x2` sont des sockets de sorties (obtenues par ailleurs),
   - on relie `x0` ↦ `tsk::in0`, `x1` ↦ `tsk::in1` et `x2` ↦ `tsk::fw0`
   - la tâche `tsk` est exécutée
   - les sockets `tsk::out1` et `tsk::out2` retournées à l'utilisateur dans `y0` et `y1`
2. Appeler `mdl["tsk"]` avec `x0`...`x2` des `np.ndarray` ? ✓
3. Idem mais `y0` et `y1` aussi `np.ndarray` ? ✓
   - `y0, y1 = mdl['tsk'](x0, x1, fw0 = x2, raw_data=True)`

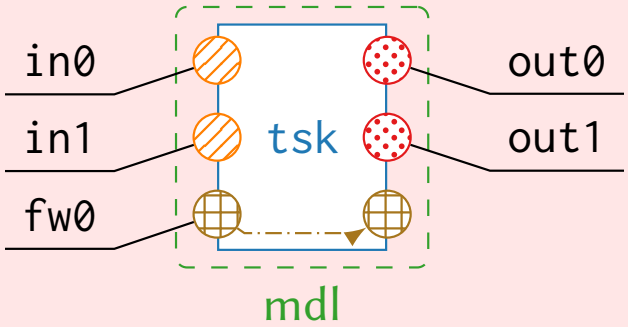
Ergonomie (tâches)



Utilisateur pourrait manipuler les sockets comme des méthodes.

1. `y0, y1 = mdl['tsk'](x0, x1, fw0 = x2)`
   Surcharge de `__call__`
   - `x0`, `x1`, `x2` sont des sockets de sorties (obtenues par ailleurs),
   - on relie `x0` ↦ `tsk::in0`, `x1` ↦ `tsk::in1` et `x2` ↦ `tsk::fw0`
   - la tâche `tsk` est exécutée
   - les sockets `tsk::out1` et `tsk::out2` retournées à l'utilisateur dans `y0` et `y1`
2. Appeler `mdl["tsk"]` avec `x0`...`x2` des `np.ndarray` ? ✓
3. Idem mais `y0` et `y1` aussi `np.ndarray` ? ✓
   - `y0, y1 = mdl['tsk'](x0, x1, fw0 = x2, raw_data=True)`

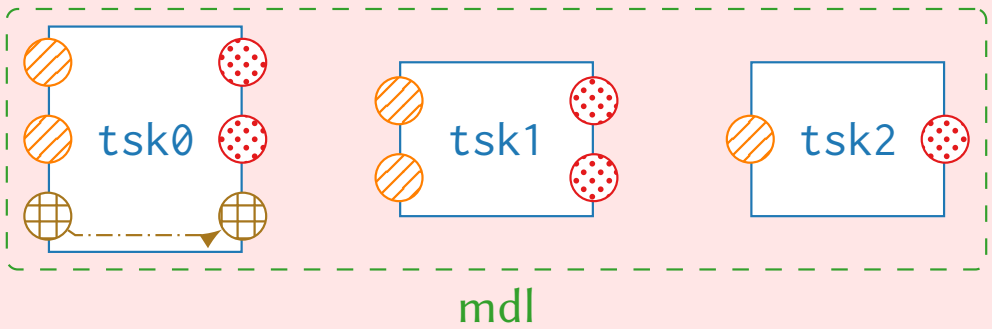
Ergonomie (tâches)



Utilisateur pourrait manipuler les sockets comme des méthodes.

1. `y0, y1 = mdl['tsk'](x0, x1, fw0 = x2)`
   Surcharge de `__call__`
   - `x0`, `x1`, `x2` sont des sockets de sorties (obtenues par ailleurs),
   - on relie `x0` ↦ `tsk::in0`, `x1` ↦ `tsk::in1` et `x2` ↦ `tsk::fw0`
   - la tâche `tsk` est exécutée
   - les sockets `tsk::out1` et `tsk::out2` retournées à l'utilisateur dans `y0` et `y1`
2. Appeler `mdl["tsk"]` avec `x0`...`x2` des `np.ndarray` ? ✓
3. Idem mais `y0` et `y1` aussi `np.ndarray` ? ✓
   - `y0, y1 = mdl['tsk'](x0, x1, fw0 = x2, raw_data=True)`

Ergonomie (modules)



Quid des modules ?

1. `mdl.tsk0`
   Les tâches ⇔ attributs des modules (automatiquement)
   (Les sockets ⇔ attributs des tâches - automatiquement)
   - Surcharge de `__getattr__` et `__setattr__` (bind)
   - Surcharge de `__dir__` permet l'auto-complétion
   - `y0, y1 = mdl.tsk0(x0, x1, fw0 = x2)`
2. `y0, y1 = mdl(x0, x1, fw0 = x2)`
   Surcharge de `__call__` pour les modules : `mdl(...)` ⇔ `mdl.tasks[0](...)`

Ergonomie (modules)

Et les séquences / pipelines dans tout ça ?

1. L'utilisateur a toujours accès à la tâche d'une socket avec l'attribut `task`
   Il peut donc créer une séquence ou un pipeline comme avant.
2. `seq = aff3ct.Sequence.from_socket(s)`
   Fonctionnalité qui crée la une séquence permettant de "remplir" la socket `s`.
3. Pipeline non abordé (encore)

Ergonomie (modules)

Et les séquences / pipelines dans tout ça ?

1. L'utilisateur a toujours accès à la tâche d'une socket avec l'attribut `task`
   Il peut donc créer une séquence ou un pipeline comme avant.
2. `seq = aff3ct.Sequence.from_socket(s)`
   Fonctionnalité qui crée la une séquence permettant de "remplir" la socket `s`.
3. Pipeline non abordé (encore)

Ergonomie (modules)

Et les séquences / pipelines dans tout ça ?

1. L'utilisateur a toujours accès à la tâche d'une socket avec l'attribut `task`
   Il peut donc créer une séquence ou un pipeline comme avant.
2. `seq = aff3ct.Sequence.from_socket(s)`
   Fonctionnalité qui crée la une séquence permettant de "remplir" la socket `s`.
3. Pipeline non abordé (encore)

Ergonomie (modules)

Et les séquences / pipelines dans tout ça ?

1. L'utilisateur a toujours accès à la tâche d'une socket avec l'attribut `task`
   Il peut donc créer une séquence ou un pipeline comme avant.
2. `seq = aff3ct.Sequence.from_socket(s)`
   Fonctionnalité qui crée la une séquence permettant de "remplir" la socket `s`.
3. Pipeline non abordé (encore)

Installer AFF3CT pour Python

Avant...

```
git clone --recursive https://github.com/aff3ct/py_aff3ct.git
# Construire AFF3CT
# ... 4 lignes
make -j4

# Construire la documentation d'AFF3CT
# ... 4 lignes
doxygen Doxyfile

# Configurer PYAF, construire librairie AFF3CT pour python
# ... 6 lignes
make -j4
```

Maintenant

```
git clone --recursive https://github.com/aff3ct/pyaf-core.git
pip install ./pyaf-core
```

Installer AFF3CT pour Python

Avant...

```
git clone --recursive https://github.com/aff3ct/py_aff3ct.git
# Construire AFF3CT
# ... 4 lignes
make -j4

# Construire la documentation d'AFF3CT
# ... 4 lignes
doxygen Doxyfile

# Configurer PYAF, construire librairie AFF3CT pour python
# ... 6 lignes
make -j4
```

Maintenant

```
git clone --recursive https://github.com/aff3ct/pyaf-core.git
pip install ./pyaf-core
```

Installer AFF3CT pour Python

Avant...

```
git clone --recursive https://github.com/aff3ct/py_aff3ct.git
# Construire AFF3CT
# ... 4 lignes
make -j4

# Construire la documentation d'AFF3CT
# ... 4 lignes
doxygen Doxyfile

# Configurer PYAF, construire librairie AFF3CT pour python
# ... 6 lignes
make -j4
```

Maintenant

```
git clone --recursive https://github.com/aff3ct/pyaf-core.git
pip install ./pyaf-core
```

Installer AFF3CT pour Python

Avant...

```
git clone --recursive https://github.com/aff3ct/py_aff3ct.git
# Construire AFF3CT
# ... 4 lignes
make -j4

# Construire la documentation d'AFF3CT
# ... 4 lignes
doxygen Doxyfile

# Configurer PYAF, construire librairie AFF3CT pour python
# ... 6 lignes
make -j4
```

Maintenant

```
git clone --recursive https://github.com/aff3ct/pyaf-core.git
pip install ./pyaf-core
```

Installer AFF3CT pour Python

Avant...

```
git clone --recursive https://github.com/aff3ct/py_aff3ct.git
# Construire AFF3CT
# ... 4 lignes
make -j4

# Construire la documentation d'AFF3CT
# ... 4 lignes
doxygen Doxyfile

# Configurer PYAF, construire librairie AFF3CT pour python
# ... 6 lignes
make -j4
```

Maintenant

```
git clone --recursive https://github.com/aff3ct/pyaf-core.git
pip install ./pyaf-core
```

Installer AFF3CT pour Python

Avant...

```
git clone --recursive https://github.com/aff3ct/py_aff3ct.git
# Construire AFF3CT
# ... 4 lignes
make -j4

# Construire la documentation d'AFF3CT
# ... 4 lignes
doxygen Doxyfile

# Configurer PYAF, construire librairie AFF3CT pour python
# ... 6 lignes
make -j4
```

Maintenant

```
git clone --recursive https://github.com/aff3ct/pyaf-core.git
pip install ./pyaf-core
```

Importer AFF3CT dans Python

Avant...

```python
import sys
sys.path.insert(0, '../../build/lib')
# Il faut connaitre le chemin de la librairie

import py_aff3ct as aff3ct
```

Maintenant

```python
import aff3ct
```

Importer AFF3CT dans Python

Avant...

```
import sys
sys.path.insert(0, '../../build/lib')
# Il faut connaitre le chemin de la librairie

import py_aff3ct as aff3ct
```

Maintenant

```
import aff3ct
```

Bilan

1. Utilisation de `pybind11` pour exposer les classes `Module`, `Task`, `Socket` ...
2. Amélioration de l'ergonomie → écrire du python
   - Sockets ↦ données
   - Interface "fluide" avec NumPy
   - Tâches ↦ fonctions/méthodes
3. Simplification de la procédure d'installation
4. Intégration continue (CI) et tests unitaires `pytest`
5. Interface documentée
6. Aide pour les instances : `aff3ct.help(mdl/tsk/sck)` ou `help(mdl/tsk/sck)`

Bilan

1. Utilisation de `pybind11` pour exposer les classes `Module`, `Task`, `Socket` ...
2. Amélioration de l'ergonomie → écrire du python
   - Sockets ↦ données
   - Interface "fluide" avec NumPy
   - Tâches ↦ fonctions/méthodes
3. Simplification de la procédure d'installation
4. Intégration continue (CI) et tests unitaires `pytest`
5. Interface documentée
6. Aide pour les instances : `aff3ct.help(mdl/tsk/sck)` ou `help(mdl/tsk/sck)`

Bilan

1. Utilisation de `pybind11` pour exposer les classes `Module`, `Task`, `Socket` ...
2. Amélioration de l'ergonomie → écrire du python
   - Sockets ↦ données
   - Interface "fluide" avec NumPy
   - Tâches ↦ fonctions/méthodes
3. Simplification de la procédure d'installation
4. Intégration continue (CI) et tests unitaires `pytest`
5. Interface documentée
6. Aide pour les instances : `aff3ct.help(mdl/tsk/sck)` ou `help(mdl/tsk/sck)`

Bilan

1. Utilisation de `pybind11` pour exposer les classes `Module`, `Task`, `Socket` ...
2. Amélioration de l'ergonomie → écrire du python
   - Sockets ↦ données
   - Interface "fluide" avec NumPy
   - Tâches ↦ fonctions/méthodes
3. Simplification de la procédure d'installation
4. Intégration continue (CI) et tests unitaires `pytest`
5. Interface documentée
6. Aide pour les instances : `aff3ct.help(mdl/tsk/sck)` ou `help(mdl/tsk/sck)`

Plan

1 Introduction

2 Méthodologie

3 Évolutions de l'ergonomie

4 Nouvelle feuille de route

Nouvelle feuille de route

1. AFF3CT-core → AFF3CT
2. Améliorer l'installation
   - `pip install aff3ct`
   - Déploiement de versions pré-compilées sur PyPI, conda
   - Utilisation de `cibuildwheel`
3. Amélioration de l'intégration continue (CI) et tests unitaires
   - Actuellement utilisation de `pytest` (CI pour `python3.9` uniquement.)
   - Utilisation de `tox` pour le test de plusieurs versions de python
4. Amélioration de l'interface avec les IDE (VSCode, Spyder)
5. Amélioration de la gestion des boucles, conditions

Nouvelle feuille de route

1. AFF3CT-core → AFF3CT
2. Améliorer l'installation
   - `pip install aff3ct`
   - Déploiement de versions pré-compilées sur PyPI, conda
   - Utilisation de `cibuildwheel`
3. Amélioration de l'intégration continue (CI) et tests unitaires
   - Actuellement utilisation de `pytest` (CI pour `python3.9` uniquement.)
   - Utilisation de `tox` pour le test de plusieurs versions de python
4. Amélioration de l'interface avec les IDE (VSCode, Spyder)
5. Amélioration de la gestion des boucles, conditions

Nouvelle feuille de route

1. AFF3CT-core → AFF3CT
2. Améliorer l'installation
   - `pip install aff3ct`
   - Déploiement de versions pré-compilées sur PyPI, conda
   - Utilisation de `cibuildwheel`
3. Amélioration de l'intégration continue (CI) et tests unitaires
   - Actuellement utilisation de `pytest` (CI pour `python3.9` uniquement.)
   - Utilisation de `tox` pour le test de plusieurs versions de python
4. Amélioration de l'interface avec les IDE (VSCode, Spyder)
5. Amélioration de la gestion des boucles, conditions

Nouvelle feuille de route

1. AFF3CT-core → AFF3CT
2. Améliorer l'installation
   - `pip install aff3ct`
   - Déploiement de versions pré-compilées sur PyPI, conda
   - Utilisation de `cibuildwheel`
3. Amélioration de l'intégration continue (CI) et tests unitaires
   - Actuellement utilisation de `pytest` (CI pour `python3.9` uniquement.)
   - Utilisation de `tox` pour le test de plusieurs versions de python
4. Amélioration de l'interface avec les IDE (VSCode, Spyder)
5. Amélioration de la gestion des boucles, conditions

Nouvelle feuille de route

1. AFF3CT-core → AFF3CT
2. Améliorer l'installation
   - `pip install aff3ct`
   - Déploiement de versions pré-compilées sur PyPI, conda
   - Utilisation de `cibuildwheel`
3. Amélioration de l'intégration continue (CI) et tests unitaires
   - Actuellement utilisation de `pytest` (CI pour `python3.9` uniquement.)
   - Utilisation de `tox` pour le test de plusieurs versions de python
4. Amélioration de l'interface avec les IDE (VSCode, Spyder)
5. Amélioration de la gestion des boucles, conditions

Merci pour votre attention !