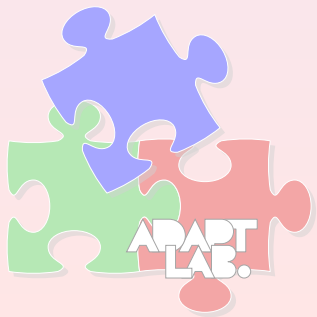
Hydra: Language Independent Library Development

Francesco Bertolotti

Università degli Studi di Milano,
Computer Science Department

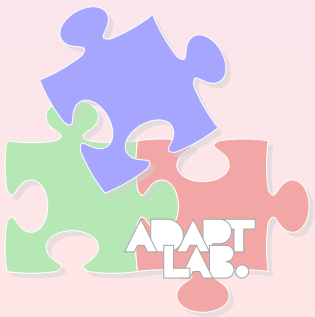
T-LADIES Kick-off, Pisa, July 6th 2022

Joint work with Walter Cazzola



Motivation

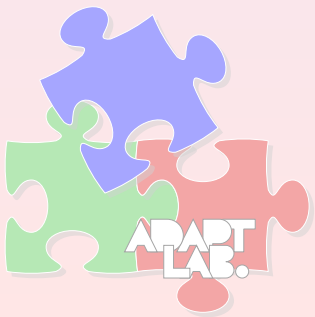
Libraries are one of the most important components of the language ecosystem.



Motivation

Library development takes years of community work. For example:

- Numpy development lasted more than 15 years.
- Pytorch development lasted more than 5 years.



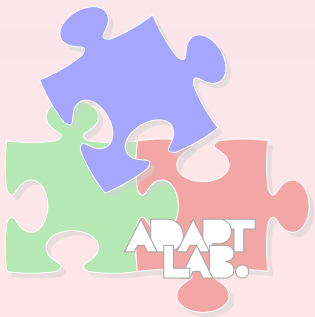
Motivation

Library development takes years of community work. For example:

- Numpy development lasted more than 15 years.
- Pytorch development lasted more than 5 years.

Libraries are available only for a handful of languages. For example:

- Numpy is available only for Python and C.
- Pytorch is available only for Python and C.



Motivation

Library development takes years of community work. For example:

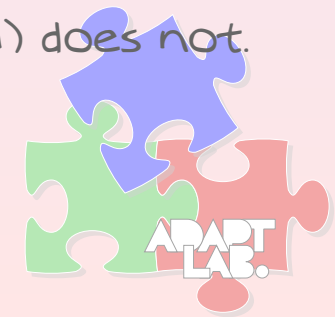
- Numpy development lasted more than 15 years.
- Pytorch development lasted more than 5 years.

Libraries are available only for a handful of languages. For example:

- Numpy is available only for Python and C.
- Pytorch is available only for Python and C.

Libraries from different languages have inconsistent interfaces. For example:

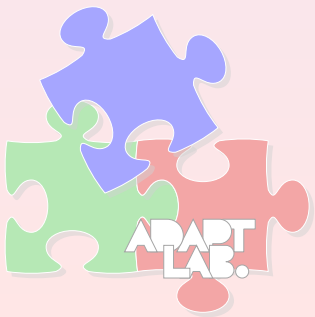
- Python random module handles more distributions than Java Random.
- Python pytorch module has a softmax method, dl4j (Java) does not.



Motivation

Language-Independent Libraries may bring few benefits.

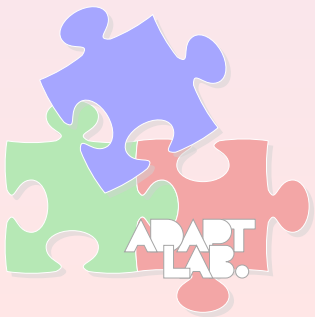
- Available for several languages and potentially new ones.
- Consistent interface between languages.
- Bigger user base: bugs get catched faster.



Problem statement and proposed solution

Problem: Library availability is language dependent

Objective: Rendering library availability language independent with a transpilation infrastructure.

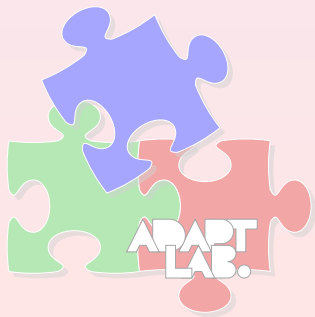


Hydra

Hydra is a one-to-many, source-to-source, transpilation infrastructure.

- one-to-many: One source language is translated to several target languages.
- source-to-source: The code of a source language is transcribed into the target language equivalent.

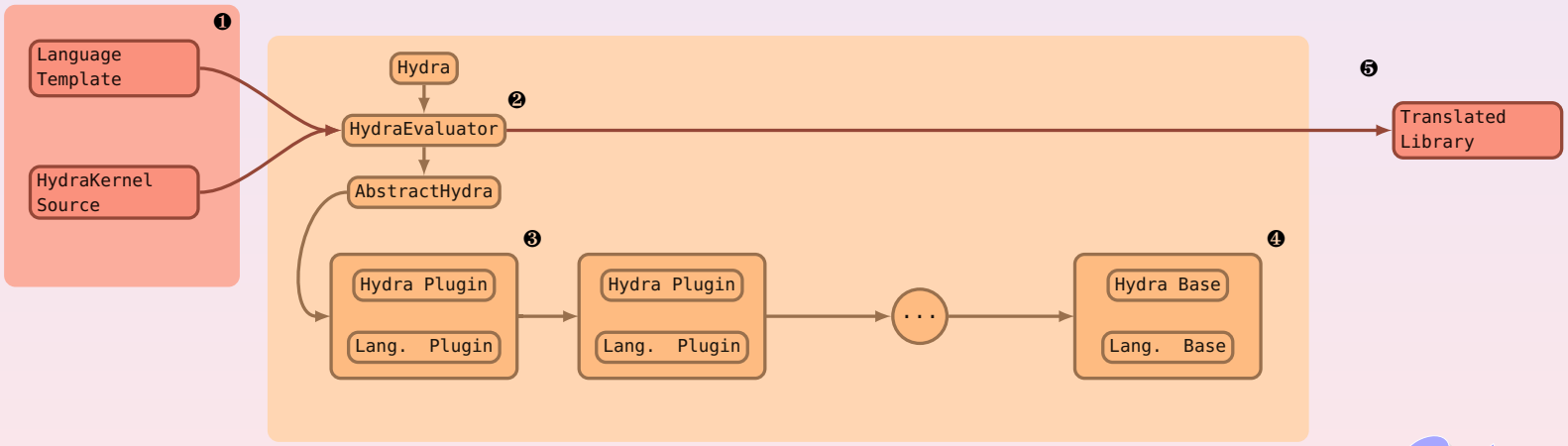
Hydra is designed for language-independent library development.

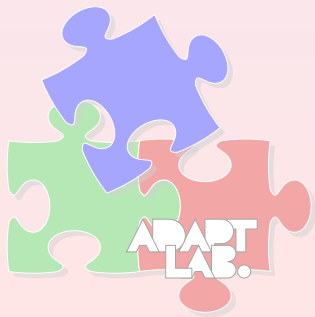


Hydra Components

There are three main components in the Hydra architecture.

- HydraKernel: the source language.
- HydraTemplate: the target language definition mechanism.
- HydraPlugins: the extension mechanism for HydraKernel.



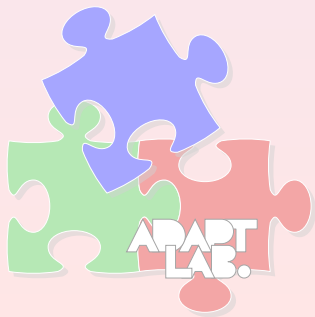


HydraKernel

HydraKernel is a subset of the Scala language.

It supports:

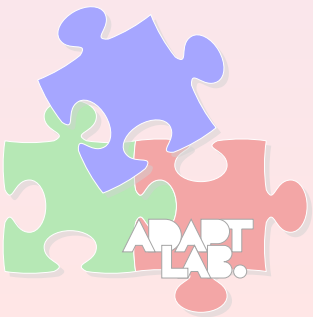
- types: int, float, boolean, void (unit in Scala), string, pointer to the previous.
- arithmetic operations: Addition, subtraction, multiplication, power, and division for int and float types.
- logical operations: and, or, and not on booleans types.
- type conversion: int-to-string, float-to-string, int-to-float.
- string operations: concatenation, length, and equality
- Control-flow statements: if, if-else, and while-do.
- Basic object-orientation abstraction such as classes.



HydraKernel

BubbleSort wrote in HydraKernel.

```
var i: Int = 0
var j: Int = 0
while ( i < array.size) {
  while ( j < array.size) {
    if(array(j) > array(j+1)) {
      swap(array,i,j)
    }
  }
}
```



HydraKernel

HydraKernel BubbleSort translated in Python.

```python
i = 0
j = 0
while i < array.__len__():
  while j < array.__len__():
    if array[j] > array[j + 1]:
      swap(array, i, j)
```

HydraKernel BubbleSort translated in C++.

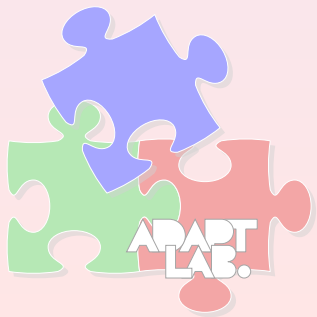
```cpp
int i = 0;
int j = 0;
while (i < array.size()) {
  while(j < array.size()) {
    if(array[j] > array[j + 1]) {
      swap(array, i, j)
    }
  }
}
```

Code Conditioning Directives

If needed, HydraKernel supports code conditioning directives (CCD).

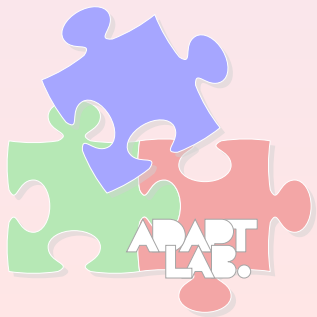
CCDs are preprocessing directives expressed through comments.

CCDs can change the code generation according to language features.

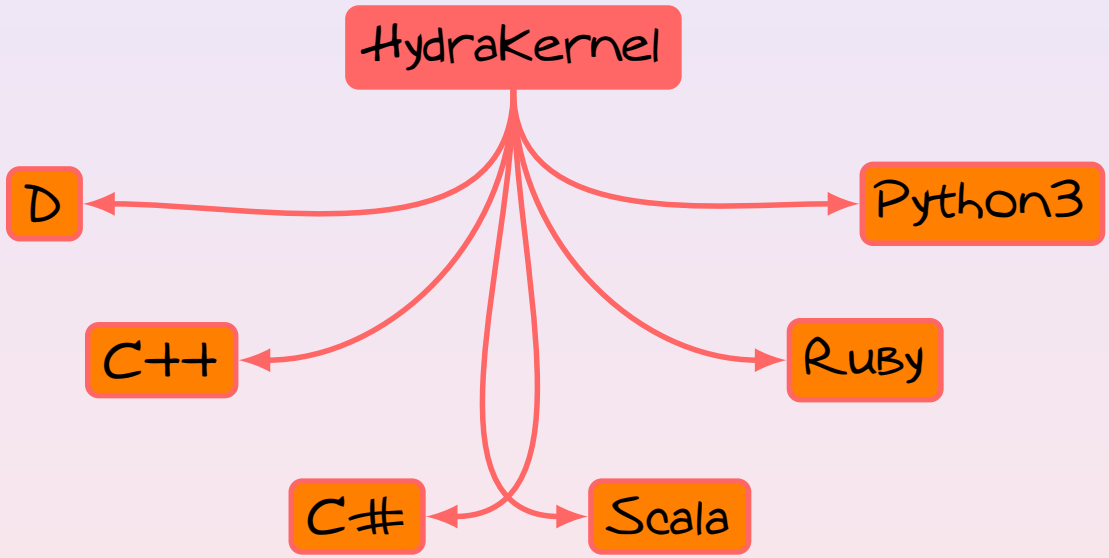


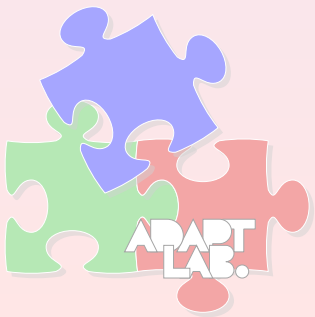
Code Conditioning Directives


```
/* IF STATIC_ARITH */
def add(op1: T, op2: T): T = return new T(add_(op1,op2))
def sub(op1: T, op2: T): T = return new T(sub_(op1,op2))
def mul(op1: T, op2: T): T = return new T(mul_(op1,op2))
def div(op1: T, op2: T): T = return new T(div_(op1,op2))
/* ELSE */
def add(op: T): T = return new T(add_(T.this.value,op))
def sub(op: T): T = return new T(sub_(T.this.value,op))
def mul(op: T): T = return new T(mul_(T.this.value,op))
def div(op: T): T = return new T(div_(T.this.value,op))
/* ENDIF */
```



HydraKernel

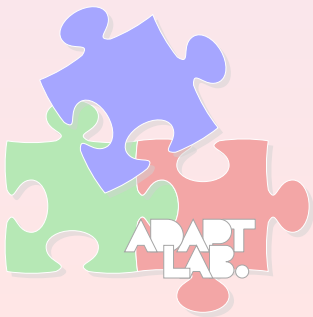




HydraTemplate

HydraTemplates are one component of the extension mechanism of Hydra.

HydraTemplates define language features in a declarative manner.

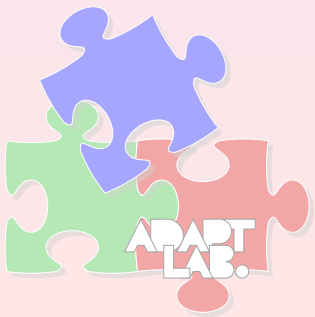


Hydra Template

By overriding:

```
def while_do(cond: String, body: String) :
  String = s"while $cond:\n${indent(body)}\n"
def if_def(cond: String, body: String) :
  String = s"if $cond:\n${indent(body)}\n"
```

- It specialize the `while_do` pattern for Python language.
- It specialize the `if_def` pattern for Python language.

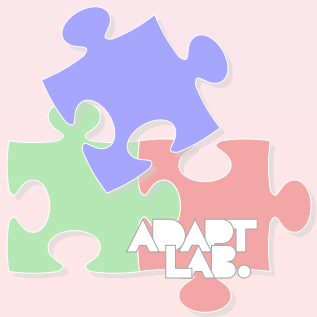


HydraTemplate

HydraTemplates should be application independent.

HydraTemplates can be specialized through inheritance.

Inherited HydraTemplates can be application-dependent.



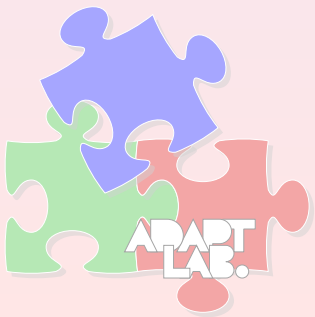
HydraPlugin

HydraPlugins are the second component of the extension mechanism.

HydraPlugins are made of two components:

- An HydraModule.
- A LanguageModule.

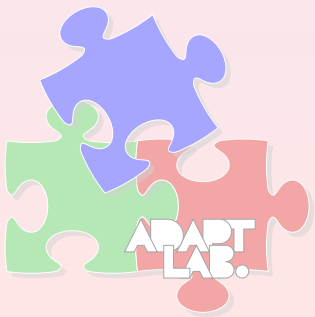
For example, HydraBase is the HydraPlugin responsible for HydraKernel core language features such as while-loops and if



HydraPlugin

HydraPlugins can be used to add new features to the HydraKernel Language.

HydraTemplates specialize the pattern for a given language.



HydraBase Module

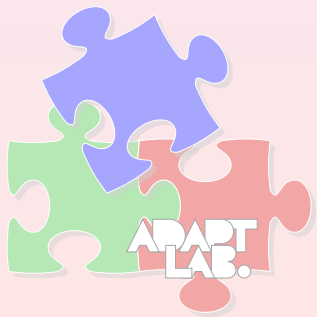
The HydraModule is responsible for identifying nodes of interest in the HydraKernel AST.

For example,

```
def translate(tree: Tree, parent_tree: Tree): String = {
  tree match {
    // ...
    case If(condition, block, Literal(Constant(()))) =>
      lang.if_def(condition, end_block(block))
    // ...
  }
}
```

It uses Scala internals to identify `if_def` nodes.

It translates the nodes using the HydraModule.



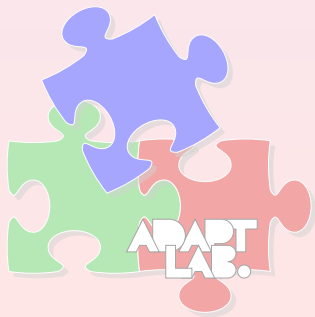
HydraBase Language

The HydraLanguage is responsible for transcribing nodes of interest into target language strings.

For example,

```
// ...
def while_do(cond: String, body: String): String
def if_def(cond: String, block: String): String
// ...
```

while_do and if_def are completely delegated to HydraTemplates.

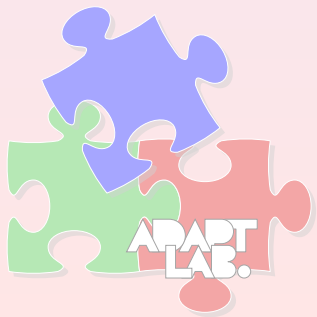


HydraNativeCalls Plugin

HydraNativeCalls plugin is responsible for handling the Foreign Function Interface (FFI).

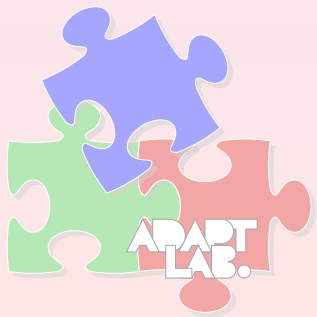
Its HydraModule identifies any extension to the `com.sun.jna.`**`Library`**.

Its HydraLanguage transcribes classes extending `com.sun.jna.`**`Library`** to the target language equivalent.



HydraNativeCalls Module


```
abstract override def translate(tree: Tree, parent_tree: Tree): String = {
  // ...
  tree match {
    // ...
    // if class extends com.sun.jna.Library
    case ClassDef(_, name, List(), Template(List(parent), _, native_funs))
         if parent.toString == "com.sun.jna.Library" =>
      // transcribe class into FFI eqv for target language
    // ...
  }
}
```



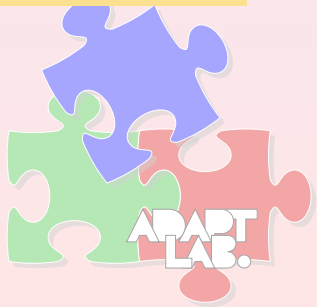
HydraNativeCalls Language

NativePlugin needs to know how a null pointer in the target language is declared.

```
// ...
val null_pointer_value: String
/// ...
```

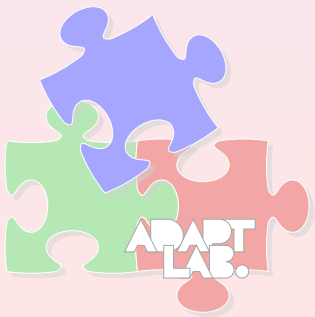
A LanguageTemplate fills the language-dependent knowledge. For example the Python3 template:

```
// ...
val null_pointer_value: String = "None"
// ...
```



Hydra Workflow

1. Develop the library with any language.
2. Expose a C API for your library.
3. Write bindings using HydraKernel.
4. Translate the bindings to any language.
5. If needed use add new templates or new plugins.

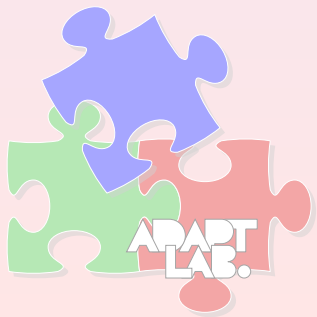


Case study: Numpy

Numpy is a popular library for scientific computing.

Numpy is available only for Python and C.

We would like to have a Numpy-like library for other languages.



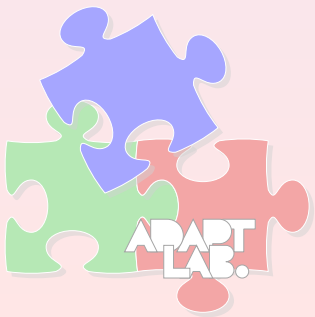
MiniNumpy

MiniNumpy is a Numpy subset written in HydraKernel.

MiniNumpy builds upon Numpy and Python C API.

We ported MiniNumpy for:

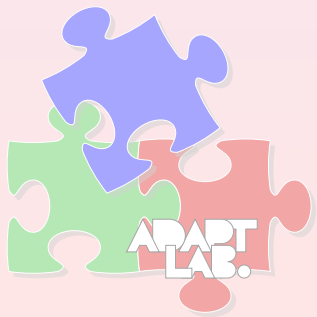
- D,
- C++,
- C#,
- Scala,
- Ruby,
- Python3.



MiniNumpy

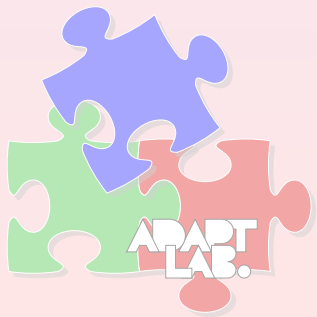
MiniNumpy supports:

- Some array allocation routines: arange, linespace, zeros, and ones.
- Some array manipulation routines: reshape, repeat.
- Some array operations: addition, subtraction, division, and multiplication.
- And, classic indexing routines.



MiniNumpy HydraKernel


```
object Numpy {
  // ...
  def zeros(shape:Array[NativeInt], type:Int): NpyArray = {
    return new NpyArray(c_zeros((new Shape(shape)).getPointer(),
                                type.toNativeInt))
  }
  def ones(shape:Array[NativeInt], type:Int): NpyArray = {
    return new NpyArray(c_ones((new Shape(shape)).getPointer(),
                           type.toNativeInt))
  }
  // ...
}
```



MiniNumpy Generated

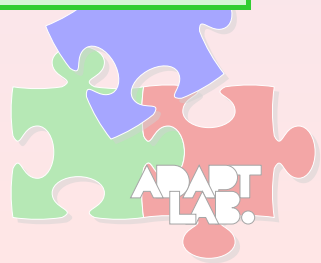
MiniNumpy for C++.

```
NpyArray zeros(std::vector<int> shape, int npytype) {
    return (NpyArray(c_zeros(Shape(shape).getPointer(), npytype)));
}
NpyArray ones(std::vector<int> shape, int npytype) {
    return (NpyArray(c_ones(Shape(shape).getPointer(), npytype)));
}
```

MiniNumpy for Python.

```
def zeros(shape, npytype):
  return NpyArray(c_zeros(Shape(shape).getPointer(), npytype))

def ones(shape, npytype):
  return NpyArray(c_ones(Shape(shape).getPointer(), npytype))
```



MiniNumpy Mandelbrot

MiniNumpy Mandelbrot in Python.

```
size = 1024

x = np.linspace(-2, 1, size, np.FLOAT64)
      .reshape((1, size))
      .repeat(size,0)
y = np.linspace(-1, 1, size, np.FLOAT64)
      .reshape((size, 1))
      .repeat(size,1)

c = x+np.complex_(0,1)*y
z = np.zeros((size, size), np.COMPLEX128)
m = np.ones((size, size), np.BOOLEAN)

for i in range(100):
    z[m] = z[m] * z[m] + c[m]
    m[z.abs() > np.float_(2.0)] = np.bool_(False)
```

Numpy Mandelbrot

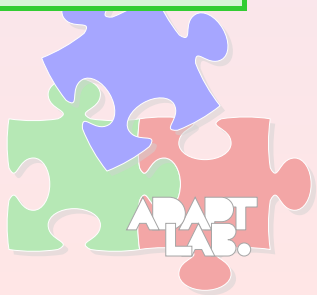
Numpy Mandelbrot in Python.

```
size = 1024

x = np.linspace(-2, 1, size, dtype=np.float64)
      .reshape((1, size))
      .repeat(size,0)
y = np.linspace(-1, 1, size, dtype=np.float64)
      .reshape((size, 1))
      .repeat(size,1)

c = x+1j*y
z = np.zeros((size, size), dtype=np.complex128)
m = np.ones((size, size), dtype=bool)

for i in range(100):
    z[m] = z[m] * z[m] + c[m]
    m[abs(z) > 2.0] = False
```

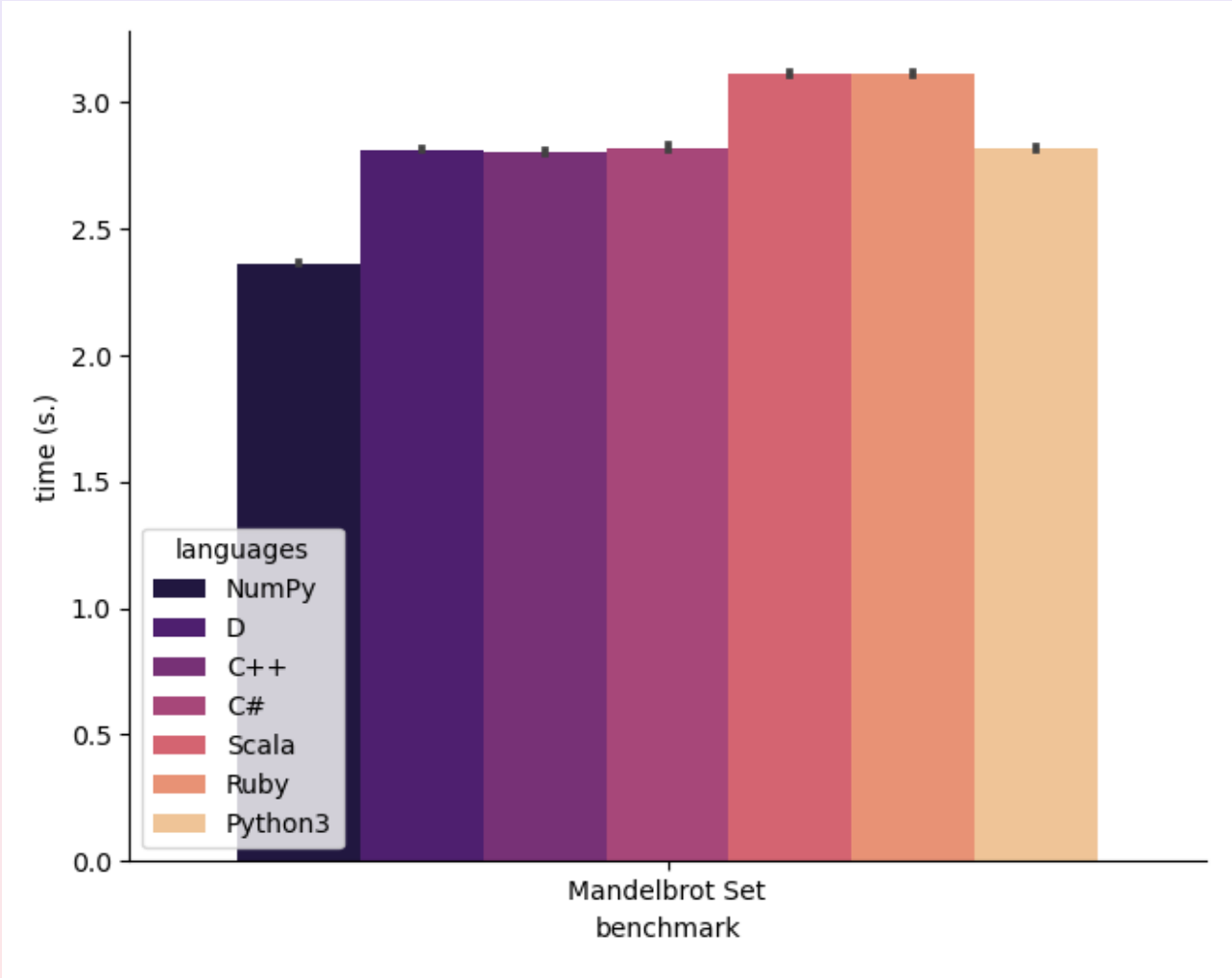


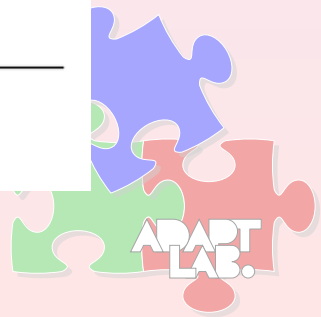
MiniNumpy Mandelbrot

MiniNumpy Mandelbrot in C++.

```
auto size = 1024;
auto x = np::linspace(-2, 1, size, FLOAT64).reshape({1, size})
                                           .repeat(size,0);
auto y = np::linspace(-1, 1, size, FLOAT64).reshape({size, 1})
                                           .repeat(size,1);
auto c = x+np::complex_(0,1)*y;
auto z = np::zeros({size, size}, COMPLEX128);
auto m = np::ones({size, size}, BOOLEAN);
for(int i = 0; i < 100; ++i) {
    z.put(m, z.loc(m) * z.loc(m) + c.loc(m));
    m.put(z.abs() > (np::float_(2.0)), np::bool_(false));
}
```

Benchmark





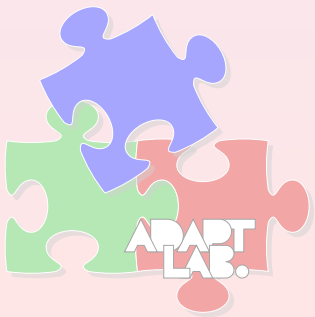
Conclusions

Pros:

- Hydra can deliver language-independent libraries.
- HydraTemplates are reusable.
- HydraTemplates declarative approach renders language extension simple.

Cons:

- HydraKernel lacks many convenient language features.
- HydraKernel debugging is difficult.



Thank you for your attention.