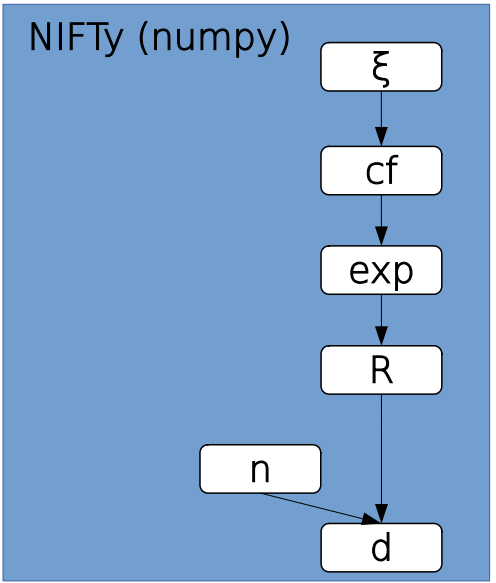
Extending NIFTy and Jax

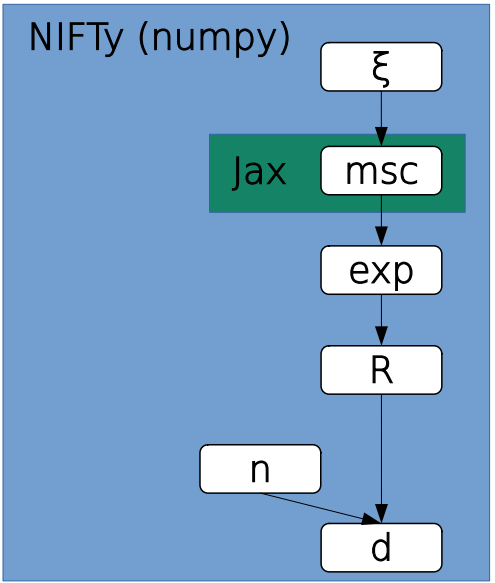
Jakob Roth, MPA Garching

November 21, 2023

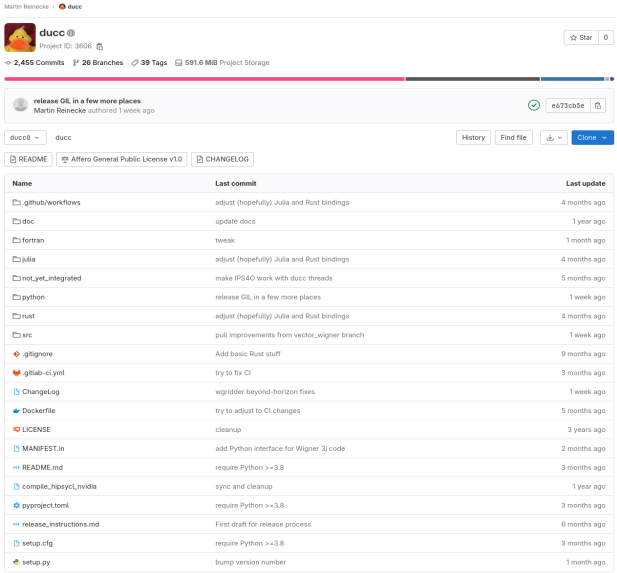
NIFTy Model

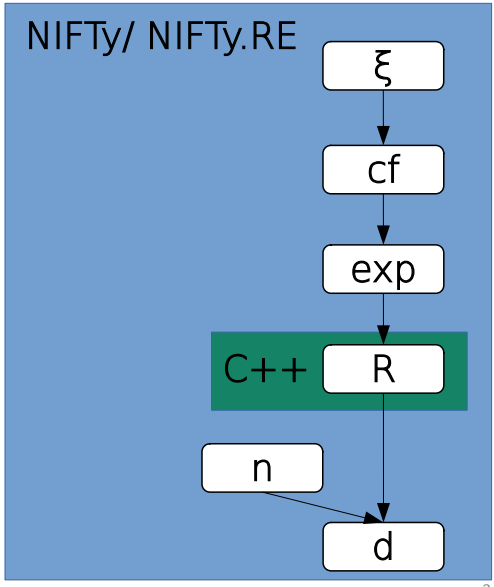


NIFTy Model



NIFTy/ NIFTy.Re Model





Interfaces

- Jax to NIFTy
- C++ to NIFTy
- C++ to NIFTy.RE
- Tensorflow in Jax

Jax to NIFTy

```python
class JaxOperator(Operator):
  """Wrap a jax function as nifty operator.

  Parameters
  ----------
  domain : DomainTuple or MultiDomain
    Domain of the operator.

  target : DomainTuple or MultiDomain
    Target of the operator.

  func : callable
    The jax function that is evaluated by
    the operator.
  """
  def __init__(self, domain, target, func):
```

Why is it that easy?

- NIFTy and Jax both in Python
- Jax as built in AD

→ JaxOperator connects Jax AD with NIFTy AD.

Jax to NIFTy

NIFTy

```
y = op(x)

# for linear op
y = op.adjoint(x)

# for non-linear op
jac = op(lin).jac
dy = jac(dx)
dx = jac.adjoint(dy)
```

Jax

```
y = op(x)

# for linear op
x = linear_transpose(op, x)(y)[0]

# for non-linear op
y, dy = jvp(op, [x], [dx])
y, op_vjp = vjp(op, x)
dx = op_vjp(dy)
```

Jax to NIFTy

```python
class JaxLinearOperator(LinearOperator):
    def __init__(self, domain, target, func):
        self._domain = makeDomain(domain)
        self._target = makeDomain(target)
        self._func = func
        self._func_T = jax.linear_transpose(func, inp)
        self._capability = self.TIMES | self.ADJOINT_TIMES

    def apply(self, x, mode):
        if mode == self.TIMES:
            fx = self._func(x.val)
            return makeField(self._target, fx)
        fx = self._func_T(x.val)
        return makeField(self._domain, fx)
```

Jax to NIFTy

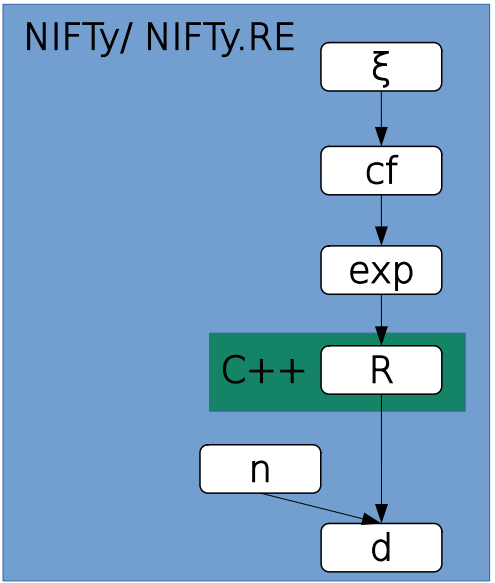
```
class JaxOperator(Operator):
  def __init__(self, domain, target, func):
    ...

    def apply(self, x):
      if is_linearization(x):
        res = self._func(x.val.val)
        bwd = partial(self._bwd, x.val.val)
        fwd = partial(self._fwd, x.val.val)
        jac = JaxLinearOperator(domain, target, fwd, bwd)
        return x.new(makeField(target, res), jac)

    res = self._func(x.val)
    return makeField(target, res)
```

C++ to NIFTy

- Bind C++ functions to python: e.g. with pybind11
- No AD in C++: Additionally code Jacobian and Jacobian adjoint



C++ to NIFTy: Summary

C++:

- Apply Operator
- Apply Jacobian
- Apply adjoint Jacobian

pybind11:

- Expose C++ functions to python

NIFTy:

- Interface to NIFTy
- Wrap C++ computations as a nifty operator

Examples: $y = x^2$

```
class SquareOperator {
  public:
    SquareOperator() {}
    py::array apply(const py::array &inp) const {
      auto pyout = inp * inp;
      return pyout;
    }

    Linearization apply_with_jac(const py::array &d) {
    // Implementation of vjp and vjp

      return Linearization(apply(d), jvp, vjp);
    }
};
```

pybind11

- "pybind11 is a lightweight header-only library that exposes C++ types in Python and vice versa, mainly to create Python bindings of existing C++ code."

https://github.com/pybind/pybind11

```cpp
#include <pybind11/numpy.h>
#include <pybind11/pybind11.h>

// Implementation of SquareOperator and Linearization
// ...

PYBIND11_MODULE(cpp_for_nifty, m) {
    py::class_<SquareOperator>(m, "SquareOperator")
        .def(py::init<>())
        .def("apply", &SquareOperator::apply)
        .def("apply_with_jac", &SquareOperator::apply_with_jac);

    py::class_<Linearization>(m, "Linearization")
        .def(py::init<const py::array &, function<py::array(const py::array &)>,
                      function<py::array(const py::array &)>>())
        .def("position", &Linearization::position)
        .def("jac_times", &Linearization::jac_times)
        .def("jac_adjoint_times", &Linearization::jac_adjoint_times);
}
```

C++ to NIFTy

```python
import numpy as np
import nifty8 as ift


class Pybind11Operator(ift.Operator):
  def __init__(self, .., pb11_op):
    ...
    self._op = pb11_op()

  def apply(self, x):
    if ift.is_linearization(x):
      lin = self._op.apply_with_jac(x.val.val)
      jac = Pybind11LinearOperator(...,  lin)
      return x.new(pos.position(), jac)
    return ift.makeField(self.target, self._op.apply(x.val))

# init op
from cpp_for_nifty import SquareOperator as sq_op_cpp
sq_op = Pybind11Operator(..., sq_op_cpp)
```

C++ to NIFTy: Summary

C++:

- Apply Operator
- Apply Jacobian
- Apply adjoint Jacobian

pybind11:

- Expose C++ functions to python

NIFTy:

- Interface to NIFTy
- Wrap C++ computations as a nifty operator

C++ to Jax: Summary

C++:

- Apply Operator
- Apply Jacobian
- Apply adjoint Jacobian

pybind11:

- Expose C++ functions to python

NIFTy:

- Interface to Jax
- Wrap C++ computations as a Jax primitive

Jax primitive

- jvp
- vjp
- vmap
- jit
- CPU/ GPU backend

C++ to Jax

```
def op(...):

def _op_abstract_eval(...): # for jit

def _op_lowering(...): # for jit

def _op_jvp(...): # for jvp

def _op_transpose(...): # for vjp

def _op_batch(...): # for vmap
```

Tensorflow in Jax

- Jax and Tensorflow both based on XLA
- *jax2tf* module enables interoperability
- Note: At the moment not all jax transforms possible with interfaced tf code
- `https://github.com/google/jax/tree/main/jax/experimental/jax2tf`
- `https://www.tensorflow.org/guide/jax2tf`

Summary

- Interfacing with other programming langues possible
- Binding C++ code for python common practice
- Manual implementation of *jvp* and *vjp* necessary
- DUCC library with many helper functions for handling n-dim arrays in C++:
  `https://gitlab.mpcdf.mpg.de/mtr/ducc`
- Interfacing Jax also possible, but a bit experimental
- Two examples for interfacing NIFTy and Jax:
  `https://gitlab.mpcdf.mpg.de/jroth/extending-jax-and-nifty`