xitorch Documentation

Release 0.4.0.dev0+

Muhammad Firmansyah Kasim

Jan 23, 2023

GETTING STARTED

xitorch (pronounced "*sigh-torch*") is a library based on PyTorch that provides differentiable operations and functionals for scientific computing and deep learning. xitorch provides analytic first and higher order derivatives automatically using PyTorch's autograd engine. It is inspired by SciPy, a popular Python library for scientific computing.

Example operations available in xitorch:

- `xitorch.linalg.symeig`: symetric eigendecomposition for large sparse matrix or implicit linear operator,
- `xitorch.optimize.rootfinder`: multivariate root finder, and
- `xitorch.integrate.solve_ivp`: initial value problem solver or commonly known as ordinary differential equations (ODE) solver.

Why use xitorch:

- contains differentiable functionals;
- provides 1st, 2nd, and higher order gradients of functionals;
- enables the use of functionals in the object-oriented way.

Source code: https://github.com/xitorch/xitorch/

CHAPTER

ONE

INSTALLATION

1.1 Requirements

- python >= 3.6
- pytorch >= 1.6 (install here)

1.2 Installation

In your terminal, type:

```
pip install xitorch
```

Or if you want to install from source, type:

```
git clone https://github.com/xitorch/xitorch/
cd xitorch
pip install -e .
```

CHAPTER TWO

USING FUNCTIONALS

xitorch contains functionals that are commonly used in scientific computing and deep learning, such as rootfinder and initial value problem solver. One advantage of xitorch is that it can provide the first and higher order derivatives of the functional outputs. However, it comes with a cost: the function input to the functionals are restricted to

1. Pure functions (i.e functions with their outputs fully determined by their tensor inputs)
2. Methods of classes derived from `torch.nn.Module`
3. Methods of classes derived from *`xitorch.EditableModule`*
4. Siblings of the above methods

In this example, we will show how to use the functionals in xitorch with above function inputs.

2.1 Pure function as input

Let's say we want to find $\mathbf{x}$ that is a root of the equation

$$\mathbf{0} = \tanh(\mathbf{A}\mathbf{x} + \mathbf{b}) + \mathbf{x}/2$$

where $\mathbf{x}$ and $\mathbf{b}$ are vectors of size $n \times 1$, and $\mathbf{A}$ is a matrix of size $n \times n$. The first step is to write the function with $\mathbf{x}$ as the first argument as well as specifying the known parameters, i.e. $\mathbf{A}$ and $\mathbf{b}$:

```
import torch
def func1(x, A, b):
    return torch.tanh(A @ x + b) + x / 2.0
A = torch.tensor([[1.1, 0.4], [0.3, 0.8]]).requires_grad_()
b = torch.tensor([[0.3], [-0.2]]).requires_grad_()
```

Once the function and parameters have been defined, now we can call the functional with an initial guess of the root.

```
from xitorch.optimize import rootfinder
x0 = torch.zeros((2,1))  # zeros as the initial guess
xroot = rootfinder(func1, x0, params=(A, b))
print(xroot)
```

```
tensor([[-0.2393],
        [ 0.2088]], grad_fn=<_RootFinderBackward>)
```

The function *`xitorch.optimize.rootfinder()`* and most other functionals in xitorch takes the similar argument patterns. It typically starts with the function as the first argument, the parameter of interest as the second argument, then followed by other parameters required by the function.

The output of the functional can be used to calculate the first order and higher order derivatives.

```
dxdA, dxdb = torch.autograd.grad(xroot.sum(), (A, b), create_graph=True)  # first␣
↪derivative
grad2A, grad2b = torch.autograd.grad(dxdA.sum(), (A, b), create_graph=True)  # second␣
↪derivative
print(grad2A)
```

```
tensor([[-0.1431,  0.1084],
        [-0.1720,  0.1303]], grad_fn=<AddBackward0>)
```

2.2 Methods of `torch.nn.Module` as input

Functionals in xitorch can also take methods from `torch.nn.Module` as their inputs, given that all the affecting parameters are listed in `.named_parameters()`.

Let's take the previous problem as an example: finding the root $\mathbf{x}$ to satisfy

$$\mathbf{0} = \tanh(\mathbf{Ax} + \mathbf{b}) + \mathbf{x}/2$$

where now $\mathbf{A}$ and $\mathbf{b}$ are parameters in a `torch.nn.Module`.

```
import torch
class NNModule(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.A = torch.nn.Parameter(torch.tensor([[1.1, 0.4], [0.3, 0.8]]))
        self.b = torch.nn.Parameter(torch.tensor([[0.3], [-0.2]]))

    def forward(self, x):  # also called in __call__
        return torch.tanh(self.A @ x + self.b) + x / 2.0
```

The functional can then be applied similarly with the previous case, but now without additional parameters

```
from xitorch.optimize import rootfinder
module = NNModule()
x0 = torch.zeros((2,1))  # zeros as the initial guess
xroot = rootfinder(module.forward, x0, params=())  # module.forward only takes x
print(xroot)
```

```
tensor([[-0.2393],
        [ 0.2088]], grad_fn=<_RootFinderBackward>)
```

The output of the rootfinder can also be used to calculate the first and higher order derivatives of the module's parameters

```
nnparams = list(module.parameters())  # (A, b)
dxdA, dxdb = torch.autograd.grad(xroot.sum(), nnparams, create_graph=True)  # first␣
↪derivative
grad2A, grad2b = torch.autograd.grad(dxdA.sum(), nnparams, create_graph=True)  #␣
↪second derivative
print(grad2A)
```

```
tensor([[-0.1431,  0.1084],
        [-0.1720,  0.1303]], grad_fn=<AddBackward0>)
```

2.3 Methods of `xitorch.EditableModule` as input

The problem with `torch.nn.Module` classes is that they can only take leaves as the parameters. However, in large scientific simulations, sometimes we want processed variables (non-leaf) as the parameters for efficiency.

To illustrate the use case of *`xitorch.EditableModule`*, let's slightly modify the test case above. We want to find the root $\mathbf{x}$ to satisfy the equation

$$\mathbf{0} = \tanh[(\mathbf{E}^3)\mathbf{x} + \mathbf{b}] + \mathbf{x}/2$$

where $\mathbf{E}^3$ is the matrix power operator. Because the matrix power operand does not depend on $\mathbf{x}$, we should be able to precompute $\mathbf{A} = \mathbf{E}^3$ so we don't have to compute it every time in the function.

To do this with *`xitorch.EditableModule`*, we can write something like

```
import torch
import xitorch
class MyModule(xitorch.EditableModule):
    def __init__(self, E, b):
        self.E = E
        self.A = E @ E @ E
        self.b = b

    def forward(self, x):
        return torch.tanh(self.A @ x + self.b) + x / 2.0

    def getparamnames(self, methodname, prefix=""):
        if methodname == "forward":
            return [prefix+"A", prefix+"b"]
        else:
            raise KeyError()
```

The biggest difference here is that in *`xitorch.EditableModule`*, a method `getparamnames` must be implemented. It returns a list of parameters affecting the outputs of a method in that class. To check if the list of parameters written manually in `getparamnames` is correct, *`xitorch.EditableModule.assertparams()`* can be used.

To use the functional, it is similar to the previous test cases

```
from xitorch.optimize import rootfinder
E = torch.tensor([[1.1, 0.4], [0.3, 0.9]]).requires_grad_()
b = torch.tensor([[0.3], [-0.2]]).requires_grad_()
mymodule = MyModule(E, b)
x0 = torch.zeros((2,1))  # zeros as the initial guess
xroot = rootfinder(mymodule.forward, x0, params=())  # .forward() only takes x
print(xroot)
```

```
tensor([[-0.3132],
        [ 0.3125]], grad_fn=<_RootFinderBackward>)
```

The output can then be used to get the derivatives with respect to direct parameters ($\mathbf{A}$ and $\mathbf{b}$) as well as indirect parameters ($\mathbf{E}$).

```
params = (mymodule.A, mymodule.b, mymodule.E)
dxdA, dxdb, dxdE = torch.autograd.grad(xroot.sum(), params, create_graph=True)  # 1st␣
↪deriv
```

(continues on next page)

(continued from previous page)

```
grad2A, grad2b, gradE = torch.autograd.grad(dxdE.sum(), params, create_graph=True)   #␣
↪2nd deriv
print(grad2A)
```

```
tensor([[-0.3660,  0.3447],
        [-0.4332,  0.4018]], grad_fn=<AddBackward0>)
```

2.4 Siblings of acceptable methods

Suppose that we want to make a new functional that finds a solution for the equation below,

$$\mathbf{y}^2 = \mathbf{f}(\mathbf{y}, \theta).$$

This is equivalent of finding the root of $\mathbf{g}(\mathbf{y}, \theta) = \mathbf{y}^2 - \mathbf{f}(\mathbf{y}, \theta)$. A naive solution would look like below

```
import torch
from xitorch.optimize import rootfinder

def quad_naive_solver(fcn, y, params, **rf_kwargs):  # solve y^2 = f(y,*params)
    def gfcn(y, *params):
        return y*y - fcn(y, *params)
    return rootfinder(gfcn, y, params, **rf_kwargs)
```

The solution above would only work if `fcn` is a pure function because in a pure function, all affecting parameters should be in `params`. However, if `fcn` is a method of `torch.nn.Module` or *`xitorch.EditableModule`*, there might be some object's parameters that are affecting parameters which are not included in `params` (as seen in the previous subsection).

The solution is to use *`xitorch.make_sibling()`* decorator as below

```
import xitorch
from xitorch.optimize import rootfinder

def quad_solver(fcn, y, params, **rf_kwargs):  # solve y^2 = f(y,*params)
    @xitorch.make_sibling(fcn)
    def gfcn(y, *params):
        return y*y - fcn(y, *params)
    return rootfinder(gfcn, y, params, **rf_kwargs)
```

The function *`xitorch.make_sibling()`* makes the decorated function as a sibling of its input function. It means that the decorated function can be seen as another method of the same instance as `fcn.__self__`. It only takes an effect if `fcn` is a method and it doesn't have any effect if `fcn` is a pure function.

Now, let's try our implementations with a method from `torch.nn.Module`.

```
class DummyModule(torch.nn.Module):
    def __init__(self, a):
        super().__init__()
        self.a = a

    def forward(self, y):
        return self.a[0] * y * y + self.a[1] * y + self.a[2]

a = torch.nn.Parameter(torch.tensor([2., 4., -5.]))
```

(continues on next page)

(continued from previous page)

```
module = DummyModule(a)
y0 = torch.zeros((1,), dtype=a.dtype)
ysolve = quad_solver(module.forward, y0, params=())
print(ysolve)
```

```
tensor([1.0000], grad_fn=<_RootFinderBackward>)
```

```
dyda = torch.autograd.grad(ysolve, a, create_graph=True)
# analytically calculated derivative
dyda_true = torch.tensor([-1./6, -1./6, -1./6])
print(dyda, dyda_true)
```

```
(tensor([-0.1667, -0.1667, -0.1667], grad_fn=<AddBackward0>),) tensor([-0.1667, -0.
↪1667, -0.1667])
```

Results matching with the analytically calculated results means that our new functional works! You can see yourself what happens if we use the naive implementation without `xitorch.make_sibling()`.

CHAPTER

THREE

BUILDING A CUSTOM LINEAR OPERATOR

xitorch provides some linear algebra operations that does not need the explicit matrix, such as *xitorch.linalg.solve()* and *xitorch.linalg.symeig()*. To represent the matrix implicitly, base class *xitorch.LinearOperator* should be used to construct user-defined linear operators. To write a LinearOperator class, the method `_mv` (matrix-vector multiplication) must be implemented.

If the LinearOperator is used in xitorch's functional with grad enabled, e.g. *xitorch.linalg.symeig()* or *xitorch.linalg.solve()*, it must have the method `_getparamnames` implemented. `_getparamnames` returns a list of parameters affecting the output, as in *xitorch.EditableModule*

As an example, to write the matrix

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & ... & 0 & a_0 \\ 0 & 0 & ... & a_1 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & a_{n-2} & ... & 0 & 0 \\ a_{n-1} & 0 & ... & 0 & 0 \end{pmatrix}$$

as a LinearOperator, we can write

```
import torch
import xitorch
class MyFlip(xitorch.LinearOperator):
   def __init__(self, a, size):
       super().__init__(shape=(size,size))
       self.a = a

   def _mv(self, x):
       return torch.flip(x, dims=(-1,)) * a

   def _getparamnames(self, prefix=""):
       return [prefix+"a"]

a = torch.arange(1, 6, dtype=torch.float).requires_grad_()
flip = MyFlip(a, 5)
print(flip)
```

```
LinearOperator (MyFlip) with shape (5, 5), dtype = torch.float32, device = cpu
```

With only `_mv` implemented, we can call all matrix operations, including

- `.mv()` (matrix-vector multiplication),
- `.mm()` (matrix-matrix multiplication),
- `.fullmatrix()` (returns the dense representation of the linear operator),

- `.rmv()` (matrix-vector right-multiplication), and
- `.rmm()` (matrix-matrix right-multiplication).

The matrix-matrix multiplication is calculated by batched matrix-vector calculation, while the right-multiplication is performed using the adjoint trick with the help of PyTorch's autograd engine.

```
vec = torch.arange(5, dtype=torch.float)
mat = torch.cat((vec.unsqueeze(-1), 2*vec.unsqueeze(-1)), dim=-1)
print(flip.mv(vec))
```

```
tensor([4., 6., 6., 4., 0.], grad_fn=<MulBackward0>)
```

```
# matrix-vector right-multiplication
print(flip.rmv(vec))
```

```
tensor([20., 12.,  6.,  2.,  0.], grad_fn=<FlipBackward>)
```

```
# matrix-matrix multiplication
print(flip.mm(mat))
```

```
tensor([[ 4.,  8.],
        [ 6., 12.],
        [ 6., 12.],
        [ 4.,  8.],
        [ 0.,  0.]], grad_fn=<SqueezeBackward1>)
```

```
# getting the dense representation
print(flip.fullmatrix())
```

```
tensor([[0., 0., 0., 0., 1.],
        [0., 0., 0., 2., 0.],
        [0., 0., 3., 0., 0.],
        [0., 4., 0., 0., 0.],
        [5., 0., 0., 0., 0.]], grad_fn=<SqueezeBackward1>)
```

The LinearOperator instance can also be used for linear algebra's operations in xitorch, such as `xitorch.linalg.solve()`

```
from xitorch.linalg import solve
mmres = flip.mm(mat)
mat2 = solve(flip, mmres)
print(mat2)
```

```
tensor([[0., 0.],
        [1., 2.],
        [2., 4.],
        [3., 6.],
        [4., 8.]], grad_fn=<LinalgSolveBackward>)
```

CHAPTER

FOUR

DEBUGGING EDITABLEMODULE AND LINEAROPERATOR

If you are implementing *xitorch.EditableModule* or *xitorch.LinearOperator*, how are you sure that your implementation is correct? For example, are parameters listed in getparamnames() method of *xitorch.EditableModule* complete or excessive? Does the implementation of *xitorch.LinearOperator* actually behave like a proper linear operator? We will answer those questions here.

4.1 Checking parameters in xitorch.EditableModule

Let's say we have a class derived from *xitorch.EditableModule*:

```
import torch
import xitorch

class AClass(xitorch.EditableModule):
    def __init__(self, a):
        self.a = a
        self.b = a*a

    def mult(self, x):
        return self.b * x

    def getparamnames(self, methodname, prefix=""):
        if methodname == "mult":
            return [prefix+"a"]  # intentionally wrong
        else:
            raise KeyError()
```

The method getparamnames returns the wrong parameters for method mult above: it returns a while it should be b. To detect the fault, you can use the method assertparams of the classes derived from *xitorch.EditableModule*.

The method assertparams takes a method and its arguments and keyword arguments as the inputs. It raises warnings if it detects missing affecting variables and excessive variables. An example is shown below.

```
a = torch.tensor(2.0).requires_grad_()
x = torch.tensor(0.4).requires_grad_()
A = AClass(a)
A.assertparams(A.mult, x)
```

```
"mult" method check done
```

```
/home/docs/checkouts/readthedocs.org/user_builds/xitorch/envs/latest/lib/
↪python3.7/site-packages/ipykernel_launcher.py:4: UserWarning: getparams␣
↪for AClass.mult does not include: b
  after removing the cwd from sys.path.
/home/docs/checkouts/readthedocs.org/user_builds/xitorch/envs/latest/lib/
↪python3.7/site-packages/ipykernel_launcher.py:4: UserWarning: getparams␣
↪for AClass.mult has excess parameters: a
  after removing the cwd from sys.path.
```

4.2 Is my `LinearOperator` actually a linear operator?

Programmatically, to implement a `LinearOperator`, you just need to implement the matrix-vector multiplication function, `._mv()`. But does the implemented operation behave like a linear operator?

To check if your implementation is correct, you can use the method `.check()` in classes derived from `LinearOperator`. It does not take any input and it will perform several checks which will raise an error if it fails.

Let's take an example of a wrong implementation of a linear operator.

```python
import torch
import xitorch

class WrongLinearOp(xitorch.LinearOperator):
    def __init__(self, a):
        shape = (torch.numel(a), torch.numel(a))
        super().__init__(shape=shape, dtype=a.dtype, device=a.device)
        self.a = a

    def _mv(self, x):
        return self.a * x + 1.0  # not a linear operator

a = torch.tensor(1.2, requires_grad=True)
linop = WrongLinearOp(a)
linop.check()
```

```
/home/docs/checkouts/readthedocs.org/user_builds/xitorch/envs/latest/lib/
↪python3.7/site-packages/ipykernel_launcher.py:15: UserWarning: The linear␣
↪operator check is performed. This might slow down your program.
  from ipykernel import kernelapp as app
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/tmp/ipykernel_223/342784360.py in <module>
     13 a = torch.tensor(1.2, requires_grad=True)
     14 linop = WrongLinearOp(a)
---> 15 linop.check()

~/checkouts/readthedocs.org/user_builds/xitorch/envs/latest/lib/python3.7/site-
↪packages/xitorch-0.4.0.dev0+3327cc0-py3.7.egg/xitorch/_core/linop.py in check(self,␣
↪warn)
    518             msg = "The linear operator check is performed. This might slow␣
↪down your program."
```

(continues on next page)

(continued from previous page)

```
    519             warnings.warn(msg, stacklevel=2)
--> 520         checklinop(self)
    521         print("Check linear operator done")
    522 

~/checkouts/readthedocs.org/user_builds/xitorch/envs/latest/lib/python3.7/site-
↪packages/xitorch-0.4.0.dev0+3327cc0-py3.7.egg/xitorch/_core/linop.py in␣
↪checklinop(linop)
    778     r = 2
    779     for (mv_xshape, mv_yshape) in zip(mv_xshapes, mv_yshapes):
--> 780         runtest("mv", mv_xshape, mv_yshape)
    781         runtest("mm", (*mv_xshape, r), (*mv_yshape, r))
    782 

~/checkouts/readthedocs.org/user_builds/xitorch/envs/latest/lib/python3.7/site-
↪packages/xitorch-0.4.0.dev0+3327cc0-py3.7.egg/xitorch/_core/linop.py in␣
↪runtest(methodname, xshape, yshape)
    749         y2 = fcn(x2)
    750         msg = "Linearity check fails\n%s\n" % str(linop)
--> 751         assert torch.allclose(y2, 1.25 * y), msg
    752         y0 = fcn(0 * x)
    753         assert torch.allclose(y0, y * 0), "Linearity check (with 0) fails\n"␣
↪+ str(linop)

AssertionError: Linearity check fails
LinearOperator (WrongLinearOp) with shape (1, 1), dtype = torch.float32, device = cpu
```

As expected, it raises an error where the check fails (i.e., it is in linearity check). This check should only be done in debugging mode as it takes considerable amount of time.

CHAPTER

FIVE

WRITING A CUSTOM IMPLEMENTATION

New in version 0.2

In every operation and functional in xitorch, there are a few method implementations are available (e.g. `solve_ivp()` has `"rk34"`, `"rk45"`, etc). What if those implementations are not good enough for you? The answer is: just write your own, and you can still take the advantage of higher order differentability provided by xitorch.

For xitorch functionals (i.e. functions that take functions as inputs, such as `rootfinder`, `quad`, etc), the custom implementation will run in `torch.no_grad()` environment, so you don't have to worry about in-place operations, backward instability, etc in your implementation. You can also use non-PyTorch or even non-Python implementation (with appropriate wrapper). This does not apply for operations, such as `Interp1D` and `SQuad`, where the gradient is obtained via backward calculation of the implementation.

To write a custom implementation of a method, it must follow the signature of the functional or operation without `bck_options` and `method` arguments. For example, the signature of `solve_ivp()` is

```
solve_ivp(fcn, ts, y0, params, bck_options, method, **fwd_options)
```

so the signature of your custom implementation should be

```
my_solve_ivp_impl(fcn, ts, y0, params, **fwd_options)
```

Let's take an example of writing the forward Euler step in `solve_ivp()`. The forward Euler step is simply given by

$$\mathbf{y}_{i+1} = \mathbf{y}_i + \mathbf{f}(t_i, \mathbf{y})(t_{i+1} - t_i).$$

The forward Euler can be implemented as below

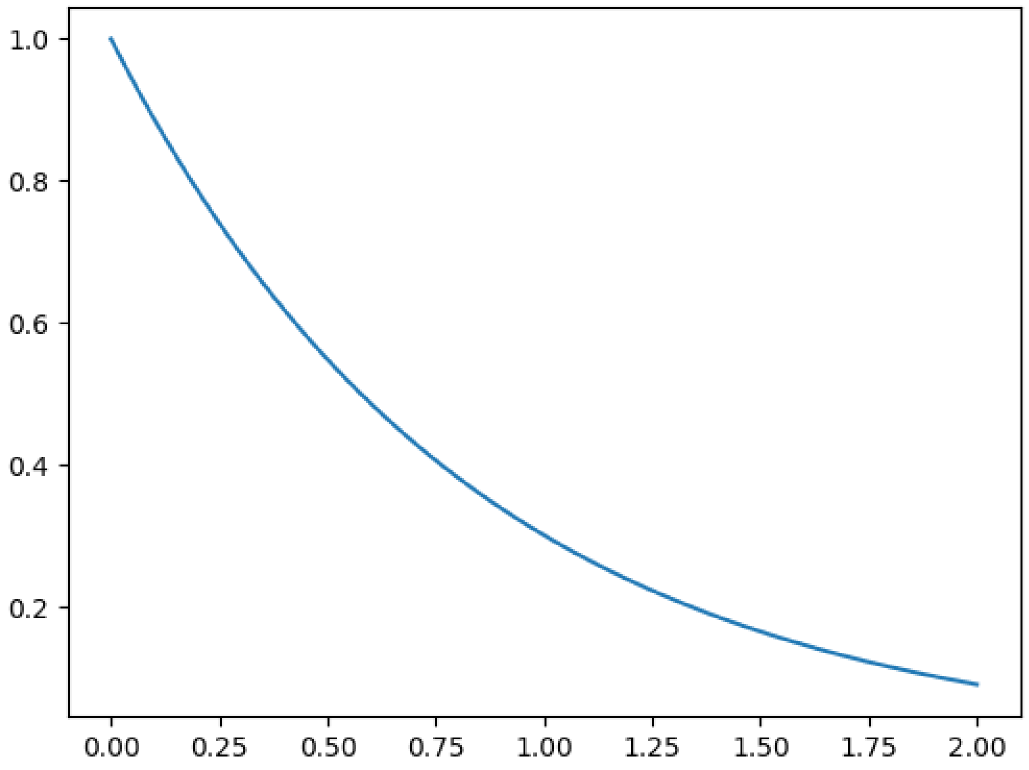
```python
import torch
import matplotlib.pyplot as plt
from xitorch.integrate import solve_ivp

def euler_forward(fcn, ts, y0, params, verbose=False, **unused):
    with torch.no_grad():
        yt = torch.empty((len(ts), *y0.shape), dtype=y0.dtype, device=y0.device)
        yt[0] = y0
        for i in range(len(ts)-1):
            yt[i+1] = yt[i] + (ts[i+1] - ts[i]) * fcn(ts[i], yt[i], *params)
        if verbose:
            print("Done")
        return yt
```

I use `torch.no_grad()` above just to illustrate that the gradient propagation is not needed in the custom implementation. In the example above, all the required arguments are present (i.e. `fcn, ts, y0, params`) plus one additional option for the implementation (i.e. `verbose`). The additional options must have a default value to comply with other implementations.

Now, using the above implementation is straightforward, just put the function above as input to the `method` argument in `solve_ivp()`.

```
fcn = lambda t,y,a: -a*y
ts = torch.linspace(0, 2, 1000, requires_grad=True)
a = torch.tensor(1.2, requires_grad=True)
y0 = torch.tensor(1.0, requires_grad=True)
yt = solve_ivp(fcn, ts, y0, params=(a,), method=euler_forward)  # custom␣
↪implementation
_ = plt.plot(ts.detach(), yt.detach())  # y(t) = exp(-a*t)
```



Although the implementation is written without gradient propagation, xitorch can still propagate the gradient. This is because xitorch uses analytical expression for the backward instead of propagating the gradient through a specific implementation.

```
# first order grad
grad_a, = torch.autograd.grad(yt[-1], a, create_graph=True)
grad_a_true = -ts[-1] * torch.exp(-a*ts[-1])  # dy/da = -t*exp(-a*t)
print(grad_a.data, grad_a_true.data)
```

```
tensor(-0.1809) tensor(-0.1814)
```

```
# second order grad
grad_a2, = torch.autograd.grad(grad_a, a)
grad_a2_true = ts[-1]**2 * torch.exp(-a*ts[-1])  # d2y/da2 = t*t*exp(-a*t)
print(grad_a2.data, grad_a2_true.data)
```

```
tensor(0.3618) tensor(0.3629)
```

We can see that with custom implementation (which does not propagate gradient), it can still calculate the first and second order gradients. The small discrepancy above is due to the imperfect calculation of Euler forward method.

CHAPTER

SIX

HOW TO CONTRIBUTE

There are a lot of ways to contribute to xitorch:

- Report bugs or request features via xitorch Github page
- Improving documentations
- Reporting and fixing bugs
- Writing a new feature

6.1 Improving documentations

If you found a typo or you think you had a better way to explain things in the documentation, you can directly make a pull request to our Github page.

If you would like to see or write a tutorial on a certain things, please raise an issue to the issue page.

6.2 Reporting and fixing bugs

If you find a bug, please report it to the Github page with a simple example on how to reproduce the bug. You can fix the bug you reported or the bugs reported by others.

6.3 Writing a new feature

If you want to implement a new feature, please raise an issue on the Github page before starting your work. It will be reviewed if the feature is suitable for xitorch. If approved, you are welcome to make a pull request. We can also give pointers on how to implement the feature if you don't know where to start.

CHAPTER

SEVEN

IMPLEMENTATION AND MATH NOTES

7.1 Derivatives of `xitorch.linalg.symeig`

Author: Muhammad Firmansyah Kasim (2020)

7.1.1 Problem

The function *`xitorch.linalg.symeig`* decomposes a linear operator to its k smallest or largest eigenvectors and eigenvalues,

$$\mathbf{AX} = \mathbf{MXE}$$

where $\mathbf{A}, \mathbf{M}$ are symmetric $n \times n$ linear operators that act as the inputs of the function. The outputs: $\mathbf{X}$ is an $n \times k$ matrix containing the eigenvectors on its column, and $\mathbf{E}$ is a $k \times k$ diagonal matrix containing the corresponding eigenvalues.

The linear operators $\mathbf{A}$ and $\mathbf{M}$ have parameters that their elements depend on, which are denoted by θ_A and θ_M, respectively. In this case, we only consider 1 parameters for each linear operator. Extending it to multiple parameters for one linear operator can be done trivially because the obtained expression will be similar to other parameters.

In this page, we will derive the expression for backward derivative (a.k.a. the vector-Jacobian product) of the linear operators parameters: $\overline{\theta_A} \equiv \partial\mathcal{L}/\partial\theta_A$ and $\overline{\theta_M} \equiv \partial\mathcal{L}/\partial\theta_M$ as functions of $\overline{\mathbf{X}} \equiv \partial\mathcal{L}/\partial\mathbf{X}$ and $\overline{\mathbf{E}} \equiv \partial\mathcal{L}/\partial\mathbf{E}$ for a loss value $\mathcal{L}$. One challenge is that we only have implicit linear operators $\mathbf{A}$ and $\mathbf{M}$ where they are expressed by their matrix-vector multiplication and right-multiplications without explicit representation on their matrix elements. Another challenge is that only k eigenpairs are available, so calculations involving full eigenvectors and eigenvalues cannot be used.

This derivation assumes the eigenvalues are all unique. Cases with degenerate eigenvalues are treated differently.

7.1.2 Forward derivative of a single eigenpair

Let's start with the eigendecomposition expression for one eigenvector and eigenvalue,

$$\mathbf{Ax} = \lambda\mathbf{Mx}, \tag{7.1}$$

where the eigenvector is normalized,

$$\mathbf{x}^T\mathbf{Mx} = 1. \tag{7.2}$$

Applying first order derivative to the equations above we obtain,

$$\mathbf{A}'\mathbf{x} + \mathbf{Ax}' = \lambda'\mathbf{Mx} + \lambda\mathbf{M}'\mathbf{x} + \lambda\mathbf{Mx}' \tag{7.3}$$

and

$$\mathbf{x}^T\mathbf{M}'\mathbf{x} + 2\mathbf{x}^T\mathbf{M}\mathbf{x}' = 0. \tag{7.4}$$

Applying $\mathbf{x}^T$ on both sides of equation (7.3), we obtain

$$\mathbf{x}^T\mathbf{A}'\mathbf{x} + \mathbf{x}^T\mathbf{A}\mathbf{x}' = \lambda'\mathbf{x}^T\mathbf{M}\mathbf{x} + \lambda\mathbf{x}^T\mathbf{M}'\mathbf{x} + \lambda\mathbf{x}^T\mathbf{M}\mathbf{x}'. \tag{7.5}$$

Substituting $\mathbf{x}^T\mathbf{M}\mathbf{x}$ from equation (7.2) and $\mathbf{x}^T\mathbf{A}$ from the transposed equation (7.1), we get the derivative of the eigenvalue,

$$\lambda' = \mathbf{x}^T(\mathbf{A}' - \lambda\mathbf{M}')\mathbf{x}. \tag{7.6}$$

To obtain the derivative of the eigenvector, we substitute (7.6) to (7.3) and rearrange it to obtain,

$$(\mathbf{A} - \lambda\mathbf{M})\mathbf{x}' = -(\mathbf{I} - \mathbf{M}\mathbf{x}\mathbf{x}^T)(\mathbf{A}' - \lambda\mathbf{M}')\mathbf{x} \tag{7.7}$$

The matrix $(\mathbf{A} - \lambda\mathbf{M})$ is not a full rank matrix, so when multiplied to $\mathbf{x}'$, some of its component is lost. To solve this, we split $\mathbf{x}'$ into 2 components, orthogonal ($\mathbf{x}'_{\mathbf{M}}$) and parallel ($\mathbf{x}'_{-\mathbf{M}}$):

$$\mathbf{x}' = \mathbf{x}'_{\mathbf{M}} + \mathbf{x}'_{-\mathbf{M}} \tag{7.8}$$

where

$$\begin{aligned} \left(\mathbf{I} - \mathbf{x}\mathbf{x}^T\mathbf{M}\right)\mathbf{x}'_{\mathbf{M}} &= \mathbf{x}'_{\mathbf{M}} \\ \left(\mathbf{I} - \mathbf{x}\mathbf{x}^T\mathbf{M}\right)\mathbf{x}'_{-\mathbf{M}} &= \mathbf{0}. \end{aligned} \tag{7.9}$$

Simple arrangement of the equations above yields

$$\begin{aligned} \mathbf{x}\mathbf{x}^T\mathbf{M}\mathbf{x}'_{\mathbf{M}} &= \mathbf{0} \\ \mathbf{x}'_{-\mathbf{M}} &= \mathbf{x}\mathbf{x}^T\mathbf{M}\mathbf{x}'_{-\mathbf{M}}. \end{aligned} \tag{7.10}$$

Using the equations (7.10) in equation (7.4) and (7.7) produces

$$\begin{aligned} \mathbf{x}^T\mathbf{M}\mathbf{x}'_{-\mathbf{M}} &= -\frac{1}{2}\mathbf{x}^T\mathbf{M}'\mathbf{x} \\ (\mathbf{A} - \lambda\mathbf{M})\mathbf{x}'_{\mathbf{M}} &= -(\mathbf{I} - \mathbf{M}\mathbf{x}\mathbf{x}^T)(\mathbf{A}' - \lambda\mathbf{M}')\mathbf{x}. \end{aligned} \tag{7.11}$$

Multiplying the first equation above with $\mathbf{x}$ and using the second equation from (7.10), we obtain,

$$\mathbf{x}'_{-\mathbf{M}} = -\frac{1}{2}\mathbf{x}\mathbf{x}^T\mathbf{M}'\mathbf{x}. \tag{7.12}$$

Moving the matrix $(\mathbf{A} - \lambda\mathbf{M})$ on the second equation of (7.11) to the right hand side gives us

$$\mathbf{x}'_{\mathbf{M}} = -(\mathbf{I} - \mathbf{x}\mathbf{x}^T\mathbf{M})(\mathbf{A} - \lambda\mathbf{M})^+(\mathbf{I} - \mathbf{M}\mathbf{x}\mathbf{x}^T)(\mathbf{A}' - \lambda\mathbf{M}')\mathbf{x}, \tag{7.13}$$

where the symbol $\mathbf{C}^+$ indicates the pseudo-inverse of the matrix. The additional term $(\mathbf{I} - \mathbf{x}\mathbf{x}^T\mathbf{M})$ is to make sure the result is orthogonal. The calculation of the pseudo-inverse can be obtained using standard linear equation solver.

To summarize, the forward derivatives are given by

$$\begin{aligned} \lambda' &= \mathbf{x}^T(\mathbf{A}' - \lambda\mathbf{M}')\mathbf{x}. \\ \mathbf{x}' &= -\frac{1}{2}\mathbf{x}\mathbf{x}^T\mathbf{M}'\mathbf{x} - (\mathbf{I} - \mathbf{x}\mathbf{x}^T\mathbf{M})(\mathbf{A} - \lambda\mathbf{M})^+(\mathbf{I} - \mathbf{M}\mathbf{x}\mathbf{x}^T)(\mathbf{A}' - \lambda\mathbf{M}')\mathbf{x}. \end{aligned}$$

7.1.3 Backward derivative

From the forward derivatives, it is relatively straightforward to get the backward derivatives. Using the relation

$$\mathbf{P}' = \mathbf{Q}\mathbf{R}'\mathbf{S} \implies \overline{\mathbf{R}} = \mathbf{Q}^T\overline{\mathbf{P}}\mathbf{S}^T,$$

we get

$$\begin{aligned}\overline{\mathbf{A}} &= \mathbf{x}\mathbf{x}^T\overline{\lambda} - (\mathbf{I} - \mathbf{x}\mathbf{x}^T\mathbf{M})(\mathbf{A} - \lambda\mathbf{M})^+(\mathbf{I} - \mathbf{M}\mathbf{x}\mathbf{x}^T)\overline{\mathbf{x}}\mathbf{x}^T \\ \overline{\mathbf{M}} &= -\mathbf{x}\mathbf{x}^T\lambda\overline{\lambda} - \frac{1}{2}\mathbf{x}\mathbf{x}^T\overline{\mathbf{x}}\mathbf{x}^T + \lambda(\mathbf{I} - \mathbf{x}\mathbf{x}^T\mathbf{M})(\mathbf{A} - \lambda\mathbf{M})^+(\mathbf{I} - \mathbf{M}\mathbf{x}\mathbf{x}^T)\overline{\mathbf{x}}\mathbf{x}^T.\end{aligned}$$

For cases with multiple eigenpairs, the contributions should be summed from all eigenvalues and eigenvectors,

$$\begin{aligned}\overline{\mathbf{A}} &= \mathbf{X}\overline{\mathbf{E}}\mathbf{X}^T - \overline{\mathbf{Y}}\mathbf{X}^T \\ \overline{\mathbf{M}} &= \mathbf{X}\mathbf{E}\overline{\mathbf{E}}\mathbf{X}^T - \frac{1}{2}\mathbf{X}(\mathbf{I}\circ\mathbf{X}^T\overline{\mathbf{X}})\mathbf{X}^T + \overline{\mathbf{Y}}\mathbf{E}\mathbf{X}^T.\end{aligned} \tag{7.14}$$

where $\circ$ indicates element-wise multiplication and

$$\begin{aligned}\overline{\mathbf{Y}} &= \overline{\mathbf{V}} - \mathbf{X}\left(\mathbf{I}\circ\mathbf{X}^T\mathbf{M}\overline{\mathbf{V}}\right) \\ \overline{\mathbf{V}} &: \text{solve } \mathbf{A}\overline{\mathbf{V}} - \mathbf{M}\overline{\mathbf{V}}\mathbf{E} = \overline{\mathbf{X}} - \mathbf{M}\mathbf{X}\left(\mathbf{I}\circ\mathbf{X}^T\overline{\mathbf{X}}\right).\end{aligned} \tag{7.15}$$

Given the gradient of each elements in the linear operator, the gradient with respect to the parameters of $\mathbf{A}$ and $\mathbf{M}$ are

$$\begin{aligned}\overline{\theta_A} &= \mathrm{tr}\left(\overline{\mathbf{A}}^T\frac{\partial\mathbf{A}}{\partial\theta_A}\right) \\ \overline{\theta_M} &= \mathrm{tr}\left(\overline{\mathbf{M}}^T\frac{\partial\mathbf{M}}{\partial\theta_M}\right)\end{aligned}$$

or more conveniently written as

$$\begin{aligned}\overline{\theta_A} &= \mathrm{tr}\left[(\mathbf{X}\overline{\mathbf{E}} - \overline{\mathbf{Y}})^T\frac{\partial(\mathbf{A}\mathbf{X})}{\partial\theta_A}\right] \\ \overline{\theta_M} &= \mathrm{tr}\left[\left(\mathbf{X}\mathbf{E}\overline{\mathbf{E}} - \frac{1}{2}\mathbf{X}(\mathbf{I}\circ\mathbf{X}^T\overline{\mathbf{X}}) + \overline{\mathbf{Y}}\mathbf{E}\right)^T\frac{\partial(\mathbf{M}\mathbf{X})}{\partial\theta_M}\right].\end{aligned}$$

In PyTorch, the terms above can be calculated by propagating the gradient from $\mathbf{AX}$ or $\mathbf{MX}$ with initial gradient given on the left term, e.g. $(\mathbf{X}\overline{\mathbf{E}} - \overline{\mathbf{Y}})$ for $\overline{\theta_A}$.

CHAPTER

EIGHT

XITORCH

8.1 EditableModule

class xitorch.**EditableModule**

`EditableModule` is a base class to enable classes that it inherits be converted to pure functions for higher order derivatives purpose.

abstract getparamnames(*methodname: str*, *prefix: str = ''*) → List[str]

This method should list tensor names that affect the output of the method with name indicated in `methodname`. If the `methodname` is not on the list in this function, it should raise `KeyError`.

Parameters

- **methodname** (*str*) – The name of the method of the class.
- **prefix** (*str*) – The prefix to be appended in front of the parameters name. This usually contains the dots.

Returns Sequence of name of parameters affecting the output of the method.

Return type Sequence of string

Raises **KeyError** – If the list in this function does not contain `methodname`.

Example

```
>>> class A(xitorch.EditableModule):
...     def __init__(self, a):
...         self.b = a*a
...
...     def mult(self, x):
...         return self.b * x
...
...     def getparamnames(self, methodname, prefix=""):
...         if methodname == "mult":
...             return [prefix+"b"]
...         else:
...             raise KeyError()
```

getuniqueparams(*methodname: str*, *onlyleaves: bool = False*) → List[torch.Tensor]

Returns the list of unique parameters involved in the method specified by *methodname*.

Parameters

- **methodname** (*str*) – Name of the method where the returned parameters play roles.

- **onlyleaves** (*bool*) – If True, only returns leaf tensors. Otherwise, returns all tensors.

Returns List of tensors that are involved in the specified method of the object.

Return type list of tensors

assertparams (*method*, **args*, ***kwargs*)

Perform a rigorous check on the implemented `getparamnames` in the class for a given method and its arguments as well as keyword arguments. It raises warnings if there are missing or excess parameters in the `getparamnames` implementation.

Parameters

- **method** (*callable method*) – The method of this class to be tested
- ***args** – Arguments of the method
- ****kwargs** – Keyword arguments of the method

Example

```
>>> class AClass(xitorch.EditableModule):
...     def __init__(self, a):
...         self.a = a
...         self.b = a*a
...
...     def mult(self, x):
...         return self.b * x
...
...     def getparamnames(self, methodname, prefix=""):
...         if methodname == "mult":
...             return [prefix+"a"]  # intentionally wrong
...         else:
...             raise KeyError()
>>> a = torch.tensor(2.0).requires_grad_()
>>> x = torch.tensor(0.4).requires_grad_()
>>> A = AClass(a)
>>> A.assertparams(A.mult, x)
<...>:1: UserWarning: getparams for AClass.mult does not include: b
  A.assertparams(A.mult, x)
<...>:1: UserWarning: getparams for AClass.mult has excess parameters: a
  A.assertparams(A.mult, x)
"mult" method check done
```

8.2 LinearOperator

class xitorch.**LinearOperator** (**args*, ***kwargs*)

`LinearOperator` is a base class designed to behave as a linear operator without explicitly determining the matrix. This `LinearOperator` should be able to operate as batched linear operators where its shape is `(B1, B2, ..., Bb, p, q)` with `B*` as the (optional) batch dimensions.

For a user-defined class to behave as `LinearOperator`, it must use `LinearOperator` as one of the parent and it has to have `._mv()` method implemented and `._getparamnames()` if used in xitorch's functionals with torch grad enabled.

classmethod m (*mat: torch.Tensor*, *is_hermitian: Optional[bool] = None*)

Class method to wrap a matrix into `LinearOperator`.

Parameters

- **mat** (*torch.Tensor*) – Matrix to be wrapped in the `LinearOperator`.
- **is_hermitian** (*bool or None*) – Indicating if the matrix is Hermitian. If `None`, the symmetry will be checked. If supplied as a bool, there is no check performed.

Returns Linear operator object that represents the matrix.

Return type *LinearOperator*

Example

```
>>> mat = torch.rand(1,3,1,2)  # 1x2 matrix with (1,3) batch dimensions
>>> linop = xitorch.LinearOperator.m(mat)
>>> print(linop)
MatrixLinearOperator with shape (1, 3, 1, 2):
   tensor([[[[0.1117, 0.8158]],

            [[0.2626, 0.4839]],

            [[0.6765, 0.7539]]]])
```

abstract _getparamnames(*prefix: str = ''*) → List[str]
List the self's parameters that affecting the `LinearOperator`. This is for the derivative purpose.

Parameters prefix (*str*) – The prefix to be appended in front of the parameters name. This usually contains the dots.

Returns List of parameter names (including the prefix) that affecting the `LinearOperator`.

Return type list of str

abstract _mv(*x: torch.Tensor*) → torch.Tensor
Abstract method to be implemented for matrix-vector multiplication. Required for all `LinearOperator` objects.

_rmv(*x: torch.Tensor*) → torch.Tensor
Abstract method to be implemented for transposed matrix-vector multiplication. Optional. If not implemented, it will use the adjoint trick to compute `.rmv()`. Usually implemented for efficiency reasons.

_mm(*x: torch.Tensor*) → torch.Tensor
Abstract method to be implemented for matrix-matrix multiplication. If not implemented, then it uses batched version of matrix-vector multiplication. Usually this is implemented for efficiency reasons.

_rmm(*x: torch.Tensor*) → torch.Tensor
Abstract method to be implemented for transposed matrix-matrix multiplication. If not implemented, then it uses batched version of transposed matrix-vector multiplication. Usually this is implemented for efficiency reasons.

mv(*x: torch.Tensor*) → torch.Tensor
Apply the matrix-vector operation to vector `x` with shape `(...,q)`. The batch dimensions of `x` need not be the same as the batch dimensions of the `LinearOperator`, but it must be broadcastable.

Parameters x (*torch.tensor*) – The vector with shape `(...,q)` where the linear operation is operated on

Returns y – The result of the linear operation with shape `(...,p)`

Return type torch.tensor

mm(*x: torch.Tensor*) → torch.Tensor
Apply the matrix-matrix operation to matrix `x` with shape `(...,q,r)`. The batch dimensions of `x` need not be the same as the batch dimensions of the `LinearOperator`, but it must be broadcastable.

Parameters **x** (*torch.tensor*) – The matrix with shape `(...,q,r)` where the linear operation is operated on.

Returns **y** – The result of the linear operation with shape `(...,p,r)`

Return type torch.tensor

rmv(*x: torch.Tensor*) → torch.Tensor
Apply the matrix-vector adjoint operation to vector `x` with shape `(...,p)`, i.e. `A^H x`. The batch dimensions of `x` need not be the same as the batch dimensions of the `LinearOperator`, but it must be broadcastable.

Parameters **x** (*torch.tensor*) – The vector of shape `(...,p)` where the adjoint linear operation is operated at.

Returns **y** – The result of the adjoint linear operation with shape `(...,q)`

Return type torch.tensor

rmm(*x: torch.Tensor*) → torch.Tensor
Apply the matrix-matrix adjoint operation to matrix `x` with shape `(...,p,r)`, i.e. `A^H X`. The batch dimensions of `x` need not be the same as the batch dimensions of the `LinearOperator`, but it must be broadcastable.

Parameters **x** (*torch.Tensor*) – The matrix of shape `(...,p,r)` where the adjoint linear operation is operated on.

Returns **y** – The result of the adjoint linear operation with shape `(...,q,r)`.

Return type torch.Tensor

property H
Returns a LinearOperator representing the Hermite / transposed of the self LinearOperator.

Returns The Hermite / transposed LinearOperator

Return type *LinearOperator*

matmul(*b: xitorch._core.linop.LinearOperator*, *is_hermitian: bool = False*)
Returns a LinearOperator representing *self @ b*.

Parameters

- **b** (`LinearOperator`) – Other linear operator
- **is_hermitian** (*bool*) – Flag to indicate if the resulting LinearOperator is Hermitian.

Returns LinearOperator representing *self @ b*

Return type *LinearOperator*

check(*warn: Optional[bool] = None*) → None
Perform checks to make sure the `LinearOperator` behaves as a proper linear operator.

Parameters **warn** (*bool or None*) – If `True`, then raises a warning to the user that the check might slow down the program. This is to remind the user to turn off the check when not in a debugging mode. If `None`, it will raise a warning if it runs not in a debug mode, but will be silent if it runs in a debug mode.

Raises

- **RuntimeError** – Raised if an error is raised when performing linear operations of the object (e.g. calling `.mv()`, `.mm()`, etc)
- **AssertionError** – Raised if the linear operations do not behave as proper linear operations. (e.g. not scaling linearly)

8.3 make_sibling

`xitorch.`**`make_sibling`**`(`**pfuncs*`)` → Callable[[Callable], xitorch._core.pure_function.PureFunction]

Used as a decor to mark the decorated function as a sibling method of the input `pfunc`. Sibling method is a method that is virtually belong to the same object, but behaves differently. Changing the state of the decorated function will also change the state of `pfunc` and its other siblings.

8.4 Packer

class `xitorch.`**`Packer`**`(`*obj: Any*`)`

Packer is an object that could extract the tensors in a structure and rebuild the structure from the given tensors. This object preserves the structure of the object by performing the deepcopy of the object, except for the tensor.

Parameters **obj** (*Any*) – Any structure object that contains tensors.

Example

```
>>> a = torch.tensor(1.0)
>>> obj = {
...     "a": a,
...     "b": a * 3,
...     "c": a,
... }
>>> packer = xitorch.Packer(obj)
>>> tensors = packer.get_param_tensor_list()
>>> print(tensors)
[tensor(1.), tensor(3.)]
>>> new_tensors = [torch.tensor(2.0), torch.tensor(4.0)]
>>> new_obj = packer.construct_from_tensor_list(new_tensors)
>>> print(new_obj)
{'a': tensor(2.), 'b': tensor(4.), 'c': tensor(2.)}
```

`get_param_tensor_list``(`*unique: bool = True*`)` → List[torch.Tensor]

Returns the list of tensors contained in the object. It will traverse down the object via elements for list, values for dictionary, or `__dict__` for object that has `__dict__` attribute.

Parameters **unique** (*bool*) – If True, then only returns the unique tensors. Otherwise, duplicates can also be returned.

Returns List of tensors contained in the object.

Return type list of torch.Tensor

`get_param_tensor``(`*unique: bool = True*`)` → Optional[torch.Tensor]

Returns the tensor parameters as a single tensor. This can be used, for example, if there are multiple parameters to be optimized using `xitorch.optimize.minimize`.

Parameters **`unique`** (*`bool`*) – If True, then only returns the tensor from unique tensors list. Otherwise, duplicates can also be returned.

Returns The parameters of the object in a single tensor or None if there is no tensor contained in the object.

Return type torch.Tensor or None

`construct_from_tensor_list`(*tensors: List[torch.Tensor]*, *unique: bool = True*) → Any
Construct the object from the tensor list and returns the object structure with the new tensors. Executing this does not change the state of the Packer object.

Parameters

- **`tensors`** (*`list of torch.Tensor`*) – The tensor parameters to be filled into the object.
- **`unique`** (*`bool`*) – Indicating if the tensor list `tensors` is from the unique parameters of the object.

Returns A new object with the same structure as the input to `__init__` object except the tensor is changed according to `tensors`.

Return type Any

`construct_from_tensor`(*a: torch.Tensor*, *unique: bool = True*) → Any
Construct the object from the single tensor (i.e. it is the parameters tensor merged into a single tensor) and returns the object structure with the new tensor. Executing this does not change the state of the Packer object.

Parameters

- **`a`** (*`torch.Tensor`*) – The single tensor parameter to be filled.
- **`unique`** (*`bool`*) – Indicating if the tensor `a` is from the unique parameters of the object.

Returns A new object with the same structure as the input to `__init__` object except the tensor is changed according to `a`.

Return type Any

CHAPTER

NINE

XITORCH.OPTIMIZE

9.1 rootfinder

`xitorch.optimize.`**`rootfinder`**(*fcn: Callable[[...], torch.Tensor], y0: torch.Tensor, params: Sequence[Any] = [], bck_options: Mapping[str, Any] = {}, method: Optional[Union[str, Callable]] = None, **fwd_options*) → torch.Tensor

Solving the rootfinder equation of a given function,

$$\mathbf{0} = \mathbf{f}(\mathbf{y}, \theta)$$

where $\mathbf{f}$ is a function that can be non-linear and produce output of the same shape of $\mathbf{y}$, and θ is other parameters required in the function. The output of this block is $\mathbf{y}$ that produces the $\mathbf{0}$ as the output.

Parameters

- **`fcn`** (*`callable`*) – The function $\mathbf{f}$ with output tensor `(*ny)`
- **`y0`** (*`torch.tensor`*) – Initial guess of the solution with shape `(*ny)`
- **`params`** (*`list`*) – Sequence of any other parameters to be put in `fcn`
- **`bck_options`** (*`dict`*) – Method-specific options for the backward solve (see *`xitorch.linalg.solve()`*)
- **`method`** (*`str or callable or None`*) – Rootfinder method. If None, it will choose `"broyden1"`.
- **`**fwd_options`** – Method-specific options (see method section)

Returns The solution which satisfies $\mathbf{0} = \mathbf{f}(\mathbf{y}, \theta)$ with shape `(*ny)`

Return type torch.tensor

Example

```
>>> def func1(y, A):  # example function
...     return torch.tanh(A @ y + 0.1) + y / 2.0
>>> A = torch.tensor([[1.1, 0.4], [0.3, 0.8]]).requires_grad_()
>>> y0 = torch.zeros((2,1))  # zeros as the initial guess
>>> yroot = rootfinder(func1, y0, params=(A,))
>>> print(yroot)
tensor([[-0.0459],
        [-0.0663]], grad_fn=<_RootFinderBackward>)
```

method="broyden1"

```
rootfinder(..., method="broyden1", *, alpha=None, uv0=None, max_rank=None, 
↪maxiter=None, f_tol=None, f_rtol=None, x_tol=None, x_rtol=None, line_
↪search=True, verbose=False, custom_terminator=None)
```

Solve the root finder or linear equation using the first Broyden method[1]. It can be used to solve minimization by finding the root of the function's gradient.

References

Keyword Arguments

- **alpha** (*float or None*) – The initial guess of inverse Jacobian is `- alpha * I + u v^T`.
- **uv0** (*tuple of tensors or str or None*) – The initial guess of inverse Jacobian is `- alpha * I + u v^T`. If `"svd"`, then it uses 1-rank svd to obtain `u` and `v`. If None, then `u` and `v` are zeros.
- **max_rank** (*int or None*) – The maximum rank of inverse Jacobian approximation. If `None`, it is `inf`.
- **maxiter** (*int or None*) – Maximum number of iterations, or inf if it is set to None.
- **f_tol** (*float or None*) – The absolute tolerance of the norm of the output `f`.
- **f_rtol** (*float or None*) – The relative tolerance of the norm of the output `f`.
- **x_tol** (*float or None*) – The absolute tolerance of the norm of the input `x`.
- **x_rtol** (*float or None*) – The relative tolerance of the norm of the input `x`.
- **line_search** (*bool or str*) – Options to perform line search. If `True`, it is set to `"armijo"`.
- **verbose** (*bool*) – Options for verbosity

method="broyden2"

```
rootfinder(..., method="broyden2", *, alpha=None, uv0=None, max_rank=None, 
↪maxiter=None, f_tol=None, f_rtol=None, x_tol=None, x_rtol=None, line_
↪search=True, verbose=False, custom_terminator=None)
```

Solve the root finder or linear equation using the second Broyden method[2]. It can be used to solve minimization by finding the root of the function's gradient.

[1] B.A. van der Rotten, PhD thesis, "A limited memory Broyden method to solve high-dimensional systems of nonlinear equations". Mathematisch Instituut, Universiteit Leiden, The Netherlands (2003). https://web.archive.org/web/20161022015821/http://www.math.leidenuniv.nl/scripties/Rotten.pdf

[2] B.A. van der Rotten, PhD thesis, "A limited memory Broyden method to solve high-dimensional systems of nonlinear equations". Mathematisch Instituut, Universiteit Leiden, The Netherlands (2003). https://web.archive.org/web/20161022015821/http://www.math.leidenuniv.nl/scripties/Rotten.pdf

References

Keyword Arguments

- **alpha** (*float or None*) – The initial guess of inverse Jacobian is `- alpha * I + u v^T`.
- **uv0** (*tuple of tensors or str or None*) – The initial guess of inverse Jacobian is `- alpha * I + u v^T`. If `"svd"`, then it uses 1-rank svd to obtain `u` and `v`. If None, then `u` and `v` are zeros.
- **max_rank** (*int or None*) – The maximum rank of inverse Jacobian approximation. If `None`, it is `inf`.
- **maxiter** (*int or None*) – Maximum number of iterations, or inf if it is set to None.
- **f_tol** (*float or None*) – The absolute tolerance of the norm of the output `f`.
- **f_rtol** (*float or None*) – The relative tolerance of the norm of the output `f`.
- **x_tol** (*float or None*) – The absolute tolerance of the norm of the input `x`.
- **x_rtol** (*float or None*) – The relative tolerance of the norm of the input `x`.
- **line_search** (*bool or str*) – Options to perform line search. If `True`, it is set to `"armijo"`.
- **verbose** (*bool*) – Options for verbosity

method="linearmixing"

```
rootfinder(..., method="linearmixing", *, alpha=None, maxiter=None, f_
↪tol=None, f_rtol=None, x_tol=None, x_rtol=None, line_search=True,␣
↪verbose=False)
```

Solve the root finding problem by approximating the inverse of Jacobian to be a constant scalar.

Keyword Arguments

- **alpha** (*float or None*) – The initial guess of inverse Jacobian is `-alpha * I`.
- **maxiter** (*int or None*) – Maximum number of iterations, or inf if it is set to None.
- **f_tol** (*float or None*) – The absolute tolerance of the norm of the output `f`.
- **f_rtol** (*float or None*) – The relative tolerance of the norm of the output `f`.
- **x_tol** (*float or None*) – The absolute tolerance of the norm of the input `x`.
- **x_rtol** (*float or None*) – The relative tolerance of the norm of the input `x`.
- **line_search** (*bool or str*) – Options to perform line search. If `True`, it is set to `"armijo"`.
- **verbose** (*bool*) – Options for verbosity

9.2 equilibrium

`xitorch.optimize.equilibrium`(*fcn: Callable[[...], torch.Tensor]*, *y0: torch.Tensor*, *params: Sequence[Any] = []*, *bck_options: Mapping[str, Any] = {}*, *method: Optional[Union[str, Callable]] = None*, ***fwd_options*) → torch.Tensor

Solving the equilibrium equation of a given function,

$$\mathbf{y} = \mathbf{f}(\mathbf{y}, \theta)$$

where $\mathbf{f}$ is a function that can be non-linear and produce output of the same shape of $\mathbf{y}$, and θ is other parameters required in the function. The output of this block is $\mathbf{y}$ that produces the same $\mathbf{y}$ as the output.

Parameters

- **fcn** (*callable*) – The function $\mathbf{f}$ with output tensor `(*ny)`
- **y0** (*torch.tensor*) – Initial guess of the solution with shape `(*ny)`
- **params** (*list*) – Sequence of any other parameters to be put in `fcn`
- **bck_options** (*dict*) – Method-specific options for the backward solve (see `xitorch.linalg.solve()`)
- **method** (*str or None*) – Rootfinder method. If None, it will choose `"broyden1"`.
- ****fwd_options** – Method-specific options (see method section)

Returns The solution which satisfies $\mathbf{y} = \mathbf{f}(\mathbf{y}, \theta)$ with shape `(*ny)`

Return type torch.tensor

Example

```
>>> def func1(y, A):  # example function
...     return torch.tanh(A @ y + 0.1) + y / 2.0
>>> A = torch.tensor([[1.1, 0.4], [0.3, 0.8]]).requires_grad_()
>>> y0 = torch.zeros((2,1))  # zeros as the initial guess
>>> yequil = equilibrium(func1, y0, params=(A,))
>>> print(yequil)
tensor([[ 0.2313],
        [-0.5957]], grad_fn=<_RootFinderBackward>)
```

Note:

- This is a direct implementation of finding the root of $\mathbf{g}(\mathbf{y}, \theta) = \mathbf{y} - \mathbf{f}(\mathbf{y}, \theta)$

method="broyden1"

```
equilibrium(..., method="broyden1", *, alpha=None, uv0=None, max_rank=None,␣
↪maxiter=None, f_tol=None, f_rtol=None, x_tol=None, x_rtol=None, line_
↪search=True, verbose=False, custom_terminator=None)
```

Solve the root finder or linear equation using the first Broyden method[1]. It can be used to solve minimization by finding the root of the function's gradient.

References

Keyword Arguments

- **alpha** (*float or None*) – The initial guess of inverse Jacobian is `- alpha * I + u v^T`.
- **uv0** (*tuple of tensors or str or None*) – The initial guess of inverse Jacobian is `- alpha * I + u v^T`. If `"svd"`, then it uses 1-rank svd to obtain `u` and `v`. If None, then `u` and `v` are zeros.
- **max_rank** (*int or None*) – The maximum rank of inverse Jacobian approximation. If `None`, it is `inf`.
- **maxiter** (*int or None*) – Maximum number of iterations, or inf if it is set to None.
- **f_tol** (*float or None*) – The absolute tolerance of the norm of the output `f`.
- **f_rtol** (*float or None*) – The relative tolerance of the norm of the output `f`.
- **x_tol** (*float or None*) – The absolute tolerance of the norm of the input `x`.
- **x_rtol** (*float or None*) – The relative tolerance of the norm of the input `x`.
- **line_search** (*bool or str*) – Options to perform line search. If `True`, it is set to `"armijo"`.
- **verbose** (*bool*) – Options for verbosity

method="broyden2"

```
equilibrium(..., method="broyden2", *, alpha=None, uv0=None, max_rank=None,␣
↪maxiter=None, f_tol=None, f_rtol=None, x_tol=None, x_rtol=None, line_
↪search=True, verbose=False, custom_terminator=None)
```

Solve the root finder or linear equation using the second Broyden method[2]. It can be used to solve minimization by finding the root of the function's gradient.

References

Keyword Arguments

- **alpha** (*float or None*) – The initial guess of inverse Jacobian is `- alpha * I + u v^T`.
- **uv0** (*tuple of tensors or str or None*) – The initial guess of inverse Jacobian is `- alpha * I + u v^T`. If `"svd"`, then it uses 1-rank svd to obtain `u` and `v`. If None, then `u` and `v` are zeros.
- **max_rank** (*int or None*) – The maximum rank of inverse Jacobian approximation. If `None`, it is `inf`.

[1] B.A. van der Rotten, PhD thesis, "A limited memory Broyden method to solve high-dimensional systems of nonlinear equations". Mathematisch Instituut, Universiteit Leiden, The Netherlands (2003). https://web.archive.org/web/20161022015821/http://www.math.leidenuniv.nl/scripties/Rotten.pdf

[2] B.A. van der Rotten, PhD thesis, "A limited memory Broyden method to solve high-dimensional systems of nonlinear equations". Mathematisch Instituut, Universiteit Leiden, The Netherlands (2003). https://web.archive.org/web/20161022015821/http://www.math.leidenuniv.nl/scripties/Rotten.pdf

- **maxiter** (*int or None*) – Maximum number of iterations, or inf if it is set to None.
- **f_tol** (*float or None*) – The absolute tolerance of the norm of the output `f`.
- **f_rtol** (*float or None*) – The relative tolerance of the norm of the output `f`.
- **x_tol** (*float or None*) – The absolute tolerance of the norm of the input `x`.
- **x_rtol** (*float or None*) – The relative tolerance of the norm of the input `x`.
- **line_search** (*bool or str*) – Options to perform line search. If `True`, it is set to `"armijo"`.
- **verbose** (*bool*) – Options for verbosity

method="linearmixing"

```
equilibrium(..., method="linearmixing", *, alpha=None, maxiter=None, f_
↪tol=None, f_rtol=None, x_tol=None, x_rtol=None, line_search=True,␣
↪verbose=False)
```

Solve the root finding problem by approximating the inverse of Jacobian to be a constant scalar.

Keyword Arguments

- **alpha** (*float or None*) – The initial guess of inverse Jacobian is `-alpha * I`.
- **maxiter** (*int or None*) – Maximum number of iterations, or inf if it is set to None.
- **f_tol** (*float or None*) – The absolute tolerance of the norm of the output `f`.
- **f_rtol** (*float or None*) – The relative tolerance of the norm of the output `f`.
- **x_tol** (*float or None*) – The absolute tolerance of the norm of the input `x`.
- **x_rtol** (*float or None*) – The relative tolerance of the norm of the input `x`.
- **line_search** (*bool or str*) – Options to perform line search. If `True`, it is set to `"armijo"`.
- **verbose** (*bool*) – Options for verbosity

9.3 minimize

`xitorch.optimize.`**minimize**(*fcn: Callable[[…], torch.Tensor], y0: torch.Tensor, params: Sequence[Any] = [], bck_options: Mapping[str, Any] = {}, method: Union[str, Callable] = None, **fwd_options*) → torch.Tensor

Solve the unbounded minimization problem:

$$\mathbf{y}^* = \arg\min_{\mathbf{y}} f(\mathbf{y}, \theta)$$

to find the best $\mathbf{y}$ that minimizes the output of the function f.

Parameters

- **fcn** (*callable*) – The function to be optimized with output tensor with 1 element.
- **y0** (*torch.tensor*) – Initial guess of the solution with shape `(*ny)`
- **params** (*list*) – Sequence of any other parameters to be put in `fcn`
- **bck_options** (*dict*) – Method-specific options for the backward solve (see `xitorch.linalg.solve()`)

- **method** (*str or callable or None*) – Minimization method. If None, it will choose "broyden1".
- ****fwd_options** – Method-specific options (see method section)

Returns The solution of the minimization with shape (*ny)

Return type torch.tensor

Example

```
>>> def func1(y, A):  # example function
...     return torch.sum((A @ y)**2 + y / 2.0)
>>> A = torch.tensor([[1.1, 0.4], [0.3, 0.8]]).requires_grad_()
>>> y0 = torch.zeros((2,1))  # zeros as the initial guess
>>> ymin = minimize(func1, y0, params=(A,))
>>> print(ymin)
tensor([[-0.0519],
        [-0.2684]], grad_fn=<_RootFinderBackward>)
```

method="broyden1"

```
minimize(..., method="broyden1", *, alpha=None, uv0=None, max_rank=None,␣
↪maxiter=None, f_tol=None, f_rtol=None, x_tol=None, x_rtol=None, line_
↪search=True, verbose=False, custom_terminator=None)
```

Solve the root finder or linear equation using the first Broyden method[1]. It can be used to solve minimization by finding the root of the function's gradient.

References

Keyword Arguments

- **alpha** (*float or None*) – The initial guess of inverse Jacobian is `- alpha * I + u v^T`.
- **uv0** (*tuple of tensors or str or None*) – The initial guess of inverse Jacobian is `- alpha * I + u v^T`. If "svd", then it uses 1-rank svd to obtain u and v. If None, then u and v are zeros.
- **max_rank** (*int or None*) – The maximum rank of inverse Jacobian approximation. If None, it is inf.
- **maxiter** (*int or None*) – Maximum number of iterations, or inf if it is set to None.
- **f_tol** (*float or None*) – The absolute tolerance of the norm of the output f.
- **f_rtol** (*float or None*) – The relative tolerance of the norm of the output f.
- **x_tol** (*float or None*) – The absolute tolerance of the norm of the input x.
- **x_rtol** (*float or None*) – The relative tolerance of the norm of the input x.
- **line_search** (*bool or str*) – Options to perform line search. If True, it is set to "armijo".

[1] B.A. van der Rotten, PhD thesis, "A limited memory Broyden method to solve high-dimensional systems of nonlinear equations". Mathematisch Instituut, Universiteit Leiden, The Netherlands (2003). https://web.archive.org/web/20161022015821/http://www.math.leidenuniv.nl/scripties/Rotten.pdf

- **verbose** (*bool*) – Options for verbosity

method="broyden2"

```
minimize(..., method="broyden2", *, alpha=None, uv0=None, max_rank=None,␣
↪maxiter=None, f_tol=None, f_rtol=None, x_tol=None, x_rtol=None, line_
↪search=True, verbose=False, custom_terminator=None)
```

Solve the root finder or linear equation using the second Broyden method[2]. It can be used to solve minimization by finding the root of the function's gradient.

References

Keyword Arguments

- **alpha** (*float or None*) – The initial guess of inverse Jacobian is `- alpha * I + u v^T`.
- **uv0** (*tuple of tensors or str or None*) – The initial guess of inverse Jacobian is `- alpha * I + u v^T`. If `"svd"`, then it uses 1-rank svd to obtain `u` and `v`. If None, then `u` and `v` are zeros.
- **max_rank** (*int or None*) – The maximum rank of inverse Jacobian approximation. If `None`, it is `inf`.
- **maxiter** (*int or None*) – Maximum number of iterations, or inf if it is set to None.
- **f_tol** (*float or None*) – The absolute tolerance of the norm of the output `f`.
- **f_rtol** (*float or None*) – The relative tolerance of the norm of the output `f`.
- **x_tol** (*float or None*) – The absolute tolerance of the norm of the input `x`.
- **x_rtol** (*float or None*) – The relative tolerance of the norm of the input `x`.
- **line_search** (*bool or str*) – Options to perform line search. If `True`, it is set to `"armijo"`.
- **verbose** (*bool*) – Options for verbosity

method="linearmixing"

```
minimize(..., method="linearmixing", *, alpha=None, maxiter=None, f_tol=None,␣
↪f_rtol=None, x_tol=None, x_rtol=None, line_search=True, verbose=False)
```

Solve the root finding problem by approximating the inverse of Jacobian to be a constant scalar.

Keyword Arguments

- **alpha** (*float or None*) – The initial guess of inverse Jacobian is `-alpha * I`.
- **maxiter** (*int or None*) – Maximum number of iterations, or inf if it is set to None.
- **f_tol** (*float or None*) – The absolute tolerance of the norm of the output `f`.
- **f_rtol** (*float or None*) – The relative tolerance of the norm of the output `f`.
- **x_tol** (*float or None*) – The absolute tolerance of the norm of the input `x`.

[2] B.A. van der Rotten, PhD thesis, "A limited memory Broyden method to solve high-dimensional systems of nonlinear equations". Mathematisch Instituut, Universiteit Leiden, The Netherlands (2003). https://web.archive.org/web/20161022015821/http://www.math.leidenuniv.nl/scripties/Rotten.pdf

- **x_rtol** (*float or None*) – The relative tolerance of the norm of the input `x`.
- **line_search** (*bool or str*) – Options to perform line search. If `True`, it is set to `"armijo"`.
- **verbose** (*bool*) – Options for verbosity

method="gd"

```
minimize(..., method="gd", *, step=0.001, gamma=0.9, maxiter=1000, f_tol=0.0,␣
↪f_rtol=1e-08, x_tol=0.0, x_rtol=1e-08, verbose=False)
```

Vanilla gradient descent with momentum. The stopping conditions use OR criteria. The update step is following the equations below.

$$\mathbf{v}_{t+1} = \gamma \mathbf{v}_t - \eta \nabla_{\mathbf{x}} f(\mathbf{x}_t)$$
$$\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{v}_{t+1}$$

Keyword Arguments

- **step** (*float*) – The step size towards the steepest descent direction, i.e. η in the equations above.
- **gamma** (*float*) – The momentum factor, i.e. γ in the equations above.
- **maxiter** (*int*) – Maximum number of iterations.
- **f_tol** (*float or None*) – The absolute tolerance of the output `f`.
- **f_rtol** (*float or None*) – The relative tolerance of the output `f`.
- **x_tol** (*float or None*) – The absolute tolerance of the norm of the input `x`.
- **x_rtol** (*float or None*) – The relative tolerance of the norm of the input `x`.

method="adam"

```
minimize(..., method="adam", *, step=0.001, beta1=0.9, beta2=0.999, eps=1e-08,
↪ maxiter=1000, f_tol=0.0, f_rtol=1e-08, x_tol=0.0, x_rtol=1e-08,␣
↪verbose=False)
```

Adam optimizer by Kingma & Ba (2015). The stopping conditions use OR criteria. The update step is following the equations below.

$$\mathbf{g}_t = \nabla_{\mathbf{x}} f(\mathbf{x}_{t-1})$$
$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t$$
$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2$$
$$\hat{\mathbf{m}}_t = \mathbf{m}_t/(1-\beta_1^t)$$
$$\hat{\mathbf{v}}_t = \mathbf{v}_t/(1-\beta_2^t)$$
$$\mathbf{x}_t = \mathbf{x}_{t-1} - \alpha\hat{\mathbf{m}}_t/(\sqrt{\hat{\mathbf{v}}_t} + \epsilon)$$

Keyword Arguments

- **step** (*float*) – The step size towards the descent direction, i.e. α in the equations above.
- **beta1** (*float*) – Exponential decay rate for the first moment estimate.
- **beta2** (*float*) – Exponential decay rate for the first moment estimate.
- **eps** (*float*) – Small number to prevent division by 0.

- **maxiter** (*int*) – Maximum number of iterations.
- **f_tol** (*float or None*) – The absolute tolerance of the output `f`.
- **f_rtol** (*float or None*) – The relative tolerance of the output `f`.
- **x_tol** (*float or None*) – The absolute tolerance of the norm of the input `x`.
- **x_rtol** (*float or None*) – The relative tolerance of the norm of the input `x`.

CHAPTER

TEN

XITORCH.INTEGRATE

10.1 quad

`xitorch.integrate.`**`quad`**(*fcn: Union[Callable[[...], torch.Tensor], Callable[[...], Sequence[torch.Tensor]]], xl: Union[float, int, torch.Tensor], xu: Union[float, int, torch.Tensor], params: Sequence[Any] = [], bck_options: Mapping[str, Any] = {}, method: Optional[Union[str, Callable]] = None, **fwd_options*) → Union[torch.Tensor, Sequence[torch.Tensor]]

Calculate the quadrature:

$$y = \int_{x_l}^{x_u} f(x, \theta)\ \mathrm{d}x$$

Parameters

- **`fcn`** (*`callable`*) – The function to be integrated. Its output must be a tensor with shape `(*nout)` or list of tensors.
- **`xl`** (*`float, int or 1-element torch.Tensor`*) – The lower bound of the integration.
- **`xu`** (*`float, int or 1-element torch.Tensor`*) – The upper bound of the integration.
- **`params`** (*`list`*) – Sequence of any other parameters for the function `fcn`.
- **`bck_options`** (*`dict`*) – Options for the backward quadrature method.
- **`method`** (*`str or callable or None`*) – Quadrature method. If None, it will choose `"leggauss"`.
- **`**fwd_options`** – Method-specific options (see method section).

Returns The quadrature results with shape `(*nout)` or list of tensors.

Return type torch.tensor or a list of tensors

`method="leggauss"`

```
quad(..., method="leggauss", *, n=100)
```

Performing 1D integration using Legendre-Gaussian quadrature

Keyword Arguments **`n`** (*`int`*) – The number of integration points.

10.2 solve_ivp

`xitorch.integrate.`**`solve_ivp`**(*fcn: Union[Callable[[. . .], torch.Tensor], Callable[[. . .], Sequence[torch.Tensor]]], ts: torch.Tensor, y0: torch.Tensor, params: Sequence[Any] = [], bck_options: Mapping[str, Any] = {}, method: Optional[Union[str, Callable]] = None, **fwd_options*) → Union[torch.Tensor, Sequence[torch.Tensor]]

Solve the initial value problem (IVP) or also commonly known as ordinary differential equations (ODE), where given the initial value $\mathbf{y_0}$, it then solves

$$\mathbf{y}(t) = \mathbf{y_0} + \int_{t_0}^{t} \mathbf{f}(t', \mathbf{y}, \theta)\ \mathrm{d}t'$$

Although the original `solve_ivp` does not accept batched `ts`, it can be batched using functorch's `vmap` (only for explicit solver, though, e.g. `rk38`, `rk4`, and `euler`). Adaptive steps cannot be vmapped at the moment.

Parameters

- **`fcn`** (*`callable`*) – The function that represents dy/dt. The function takes an input of a single time `t` and tensor `y` with shape `(*ny)` and produce $\mathrm{d}\mathbf{y}/\mathrm{d}t$ with shape `(*ny)`. The output of the function must be a tensor with shape `(*ny)` or a list of tensors.
- **`ts`** (*`torch.tensor`*) – The time points where the value of *y* will be returned. It must be monotonically increasing or decreasing. It is a tensor with shape `(nt,)`.
- **`y0`** (*`torch.tensor`*) – The initial value of `y`, i.e. `y(t[0]) == y0`. It is a tensor with shape `(*ny)` or a list of tensors.
- **`params`** (*`list`*) – Sequence of other parameters required in the function.
- **`bck_options`** (*`dict`*) – Options for the backward solve_ivp method. If not specified, it will take the same options as fwd_options.
- **`method`** (*`str or callable or None`*) – Initial value problem solver. If None, it will choose `"rk45"`.
- **`**fwd_options`** – Method-specific option (see method section below).

Returns The values of `y` for each time step in `ts`. It is a tensor with shape `(nt,*ny)` or a list of tensors

Return type torch.tensor or a list of tensors

`method="rk45"`

```
solve_ivp(..., method="rk45", *, atol=1e-08, rtol=1e-05)
```

Perform the adaptive Runge-Kutta steps with order 4 and 5.

Keyword Arguments

- **`atol`** (*`float`*) – The absolute error tolerance in deciding the steps
- **`rtol`** (*`float`*) – The relative error tolerance in deciding the steps

`method="rk23"`

```
solve_ivp(..., method="rk23", *, atol=1e-08, rtol=1e-05)
```

Perform the adaptive Runge-Kutta steps with order 2 and 3.

Keyword Arguments

- **atol** (*float*) – The absolute error tolerance in deciding the steps
- **rtol** (*float*) – The relative error tolerance in deciding the steps

method="rk4"

```
solve_ivp(..., method="rk4")
```

Perform the Runge-Kutta steps of order 4 with a fixed step size.

method="rk38"

```
solve_ivp(..., method="rk38")
```

method="euler"

```
solve_ivp(..., method="euler")
```

10.3 mcquad

`xitorch.integrate.`**`mcquad`**(*ffcn: Union[Callable[[…], torch.Tensor], Callable[[…], Sequence[torch.Tensor]]], log_pfcn: Callable[[…], torch.Tensor], x0: torch.Tensor, fparams: Sequence[Any] = [], pparams: Sequence[Any] = [], bck_options: Mapping[str, Any] = {}, method: Optional[Union[str, Callable]] = None, **fwd_options*) → Union[torch.Tensor, Sequence[torch.Tensor]]

Performing monte carlo quadrature to calculate the expectation value:

$$\mathbb{E}_p[f] = \frac{\int f(\mathbf{x}, \theta_f) p(\mathbf{x}, \theta_p)\ \mathrm{d}\mathbf{x}}{\int p(\mathbf{x}, \theta_p)\ \mathrm{d}\mathbf{x}}$$

Parameters

- **ffcn** (*Callable*) – The function with to be integrated. Its outputs is a tensor or a list of tensors. To call the function: `ffcn(x, *fparams)`
- **log_pfcn** (*Callable*) – The natural logarithm of the probability function. The output should be a one-element tensor. To call the function: `log_pfcn(x, *pparams)`
- **x0** (*torch.Tensor*) – Tensor with any size as the initial position. The call `ffcn(x0, *fparams)` must work.
- **fparams** (*list*) – Sequence of any other parameters for `ffcn`.
- **pparams** (*list*) – Sequence of any other parameters for `gfcn`.
- **bck_options** (*dict*) – Options for the backward mcquad operation. Unspecified fields will be taken from `fwd_options`.
- **method** (*str or callable or None*) – Monte Carlo quadrature method. If None, it will choose `"mh"`.
- ****fwd_options** (*dict*) – Method-specific options (see method section below).

Returns The expectation values of the function `ffcn` over the space of `x`. If the output of `ffcn` is a list, then this is also a list.

Return type torch.Tensor or a list of torch.Tensor

method="mh"

```
mcquad(..., method="mh", *, nsamples=10000, nburnout=5000, step_size=1.0)
```

Perform Metropolis-Hasting steps to collect samples

Keyword Arguments

- **nsamples** (*int*) – The number of samples to be collected
- **nburnout** (*int*) – The number of initial steps to be performed before collecting samples
- **step_size** (*float*) – The size of the steps to be taken

method="mhcustom"

```
mcquad(..., method="mhcustom", *, nsamples=10000, nburnout=5000, custom_
↪step=None)
```

Perform Metropolis sampling using custom_step

Keyword Arguments

- **nsamples** (*int*) – The number of samples to be collected
- **nburnout** (*int*) – The number of initial steps to be performed before collecting samples
- **custom_step** (*callable or None*) – Callable with call signature `custom_step(x, *pparams)` to produce the next samples (already decided whether to accept or not). This argument is **required**. If `None`, it will raise an error

10.4 SQuad

class xitorch.integrate.**SQuad**(*x: torch.Tensor*, *method: Optional[Union[str, Callable]] = None*, ***fwd_options*)

SQuad (Sampled QUADrature) is a class for quadrature performed with a fixed samples at given points. Mathematically, it does the integration

$$\mathbf{z}(x) = \int_{x_0}^{x} \mathbf{y}(x')\ \mathrm{d}x$$

where $\mathbf{y}(x)$ is the interpolated function from a given sample.

Parameters

- **x** (*torch.Tensor*) – The positions where the samples are given. It is a 1D tensor with shape `(nx,)`.
- **method** (*str or callable or None*) – The integration method. If None, it will choose `"cspline"`.
- ****fwd_options** – Method-specific options (see method section below)

method="cspline"

```
SQuad(..., method="cspline", *, bc_type="natural")
```

Perform integration of given sampled values by assuming it is interpolated with cubic spline[1]. It is simply

$$S = \sum_{i=0}^{N-2} \left[\frac{1}{2}(y_i + y_{i+1}) + \frac{1}{12}(y_i' - y_{i+1}')(x_{i+1} - x_i)^2\right]$$

Keyword Arguments **bc_type** (*str*) – Boundary condition. See *xitorch.interpolate.Interp1D* with `"cspline"` method for details.

References

method="trapz"

```
SQuad(..., method="trapz")
```

Perform integration with trapezoidal rule. It is simply

$$S = \sum_{i=0}^{N-2} \frac{1}{2}(y_i + y_{i+1})$$

cumsum(*y: torch.Tensor*, *dim: int = - 1*) → torch.Tensor
Perform the cumulative integration of the samples **y** over the specified dimension.

Parameters

- **y** (*torch.Tensor*) – The value of samples. The size of `y` at `dim` must be equal to the length of `x`.
- **dim** (*int*) – The dimension where the cumulative integration is performed.

Returns The cumulative integrated values with the same shape as `y`.

Return type torch.Tensor

integrate(*y: torch.Tensor*, *dim: int = - 1*, *keepdim: bool = False*) → torch.Tensor
Perform the full integration of the samples **y** over the specified dimension.

Parameters

- **y** (*torch.Tensor*) – The value of samples. The size of `y` at `dim` must be equal to the length of `x`, i.e. `(..., nx, ...)`.
- **dim** (*int*) – The dimension where the integration is performed.
- **keepdim** (*bool*) – Option to not discard the integrated dimension. If `True`, the integrated dimension size will be 1.

Returns The integrated values.

Return type torch.Tensor

[1] Mark H. Holmes, "Connections Between Cubic Splines and Quadrature Rules" (eq. 8), The American Mathematical Monthly, Volume 121, Issue 8, 2014.

CHAPTER

ELEVEN

XITORCH.LINALG

11.1 solve

`xitorch.linalg.`**`solve`**(*A: xitorch._core.linop.LinearOperator*, *B: torch.Tensor*, *E: Optional[torch.Tensor] = None*, *M: Optional[xitorch._core.linop.LinearOperator] = None*, *bck_options: Mapping[str, Any] = {}*, *method: Optional[Union[str, Callable]] = None*, ***fwd_options*) → torch.Tensor

Performing iterative method to solve the equation

$$\mathbf{A}\mathbf{X} = \mathbf{B}$$

or

$$\mathbf{A}\mathbf{X} - \mathbf{M}\mathbf{X}\mathbf{E} = \mathbf{B}$$

where $\mathbf{E}$ is a diagonal matrix. This function can also solve batched multiple inverse equation at the same time by applying $\mathbf{A}$ to a tensor $\mathbf{X}$ with shape `(...,na,ncols)`. The applied $\mathbf{E}$ are not necessarily identical for each column.

Parameters

- **`A`** (`xitorch.LinearOperator`) – A linear operator that takes an input `X` and produce the vectors in the same space as `B`. It should have the shape of `(*BA, na, na)`
- **`B`** (*`torch.Tensor`*) – The tensor on the right hand side with shape `(*BB, na, ncols)`
- **`E`** (*`torch.Tensor or None`*) – If a tensor, it will solve $\mathbf{A}\mathbf{X} - \mathbf{M}\mathbf{X}\mathbf{E} = \mathbf{B}$. It will be regarded as the diagonal of the matrix. Otherwise, it just solves $\mathbf{A}\mathbf{X} = \mathbf{B}$ and `M` is ignored. If it is a tensor, it should have shape of `(*BE, ncols)`.
- **`M`** (`xitorch.LinearOperator` *`or None`*) – The transformation on the `E` side. If `E` is `None`, then this argument is ignored. If E is not `None` and `M` is `None`, then `M=I`. If LinearOperator, it must be Hermitian with shape `(*BM, na, na)`.
- **`bck_options`** (*`dict`*) – Options of the iterative solver in the backward calculation.
- **`method`** (*`str or callable or None`*) – The method of linear equation solver. If `None`, it will choose `"cg"` or `"bicgstab"` based on the matrices symmetry. *Note*: default method will be changed quite frequently, so if you want future compatibility, please specify a method.
- **`**fwd_options`** – Method-specific options (see method below)

Returns The tensor $\mathbf{X}$ that satisfies $\mathbf{A}\mathbf{X} - \mathbf{M}\mathbf{X}\mathbf{E} = \mathbf{B}$.

Return type torch.Tensor

method="cg"

```
solve(..., method="cg", *, posdef=None, precond=None, max_niter=None, rtol=1e-
↪06, atol=1e-08, eps=1e-12, resid_calc_every=10, verbose=False)
```

Solve the linear equations using Conjugate-Gradient (CG) method.

Keyword Arguments

- **posdef** (*bool or None*) – Indicating if the operation $\mathbf{AX} - \mathbf{MXE}$ a positive definite for all columns and batches. If None, it will be determined by power iterations.
- **precond** (LinearOperator *or None*) – LinearOperator for the preconditioning. If None, no preconditioner is applied.
- **max_niter** (*int or None*) – Maximum number of iteration. If None, it is set to `int(1.5 * A.shape[-1])`
- **rtol** (*float*) – Relative tolerance for stopping condition w.r.t. norm of B
- **atol** (*float*) – Absolute tolerance for stopping condition w.r.t. norm of B
- **eps** (*float*) – Substitute the absolute zero in the algorithm's denominator with this value to avoid nan.
- **resid_calc_every** (*int*) – Calculate the residual in its actual form instead of substitution form with this frequency, to avoid rounding error accummulation. If your linear operator has bad numerical precision, set this to be low. If 0, then never calculate the residual in its actual form.
- **verbose** (*bool*) – Verbosity of the algorithm.

method="bicgstab"

```
solve(..., method="bicgstab", *, posdef=None, precond_l=None, precond_r=None,␣
↪max_niter=None, rtol=1e-06, atol=1e-08, eps=1e-12, verbose=False, resid_
↪calc_every=10)
```

Solve the linear equations using stabilized Biconjugate-Gradient method.

Keyword Arguments

- **posdef** (*bool or None*) – Indicating if the operation $\mathbf{AX} - \mathbf{MXE}$ a positive definite for all columns and batches. If None, it will be determined by power iterations.
- **precond_l** (LinearOperator *or None*) – LinearOperator for the left preconditioning. If None, no preconditioner is applied.
- **precond_r** (LinearOperator *or None*) – LinearOperator for the right preconditioning. If None, no preconditioner is applied.
- **max_niter** (*int or None*) – Maximum number of iteration. If None, it is set to `int(1.5 * A.shape[-1])`
- **rtol** (*float*) – Relative tolerance for stopping condition w.r.t. norm of B
- **atol** (*float*) – Absolute tolerance for stopping condition w.r.t. norm of B
- **eps** (*float*) – Substitute the absolute zero in the algorithm's denominator with this value to avoid nan.

- **resid_calc_every** (*int*) – Calculate the residual in its actual form instead of substitution form with this frequency, to avoid rounding error accummulation. If your linear operator has bad numerical precision, set this to be low. If 0, then never calculate the residual in its actual form.
- **verbose** (*bool*) – Verbosity of the algorithm.

method="exactsolve"

```
solve(..., method="exactsolve")
```

Solve the linear equation by contructing the full matrix of LinearOperators.

Warning:

- As this method construct the linear operators explicitly, it might requires a large memory.

method="broyden1"

```
solve(..., method="broyden1", *, alpha=None, uv0=None, max_rank=None,␣
↪maxiter=None, f_tol=None, f_rtol=None, x_tol=None, x_rtol=None, line_
↪search=True, verbose=False, custom_terminator=None)
```

Solve the root finder or linear equation using the first Broyden method[1]. It can be used to solve minimization by finding the root of the function's gradient.

References

Keyword Arguments

- **alpha** (*float or None*) – The initial guess of inverse Jacobian is `- alpha * I + u v^T`.
- **uv0** (*tuple of tensors or str or None*) – The initial guess of inverse Jacobian is `- alpha * I + u v^T`. If `"svd"`, then it uses 1-rank svd to obtain `u` and `v`. If None, then `u` and `v` are zeros.
- **max_rank** (*int or None*) – The maximum rank of inverse Jacobian approximation. If `None`, it is `inf`.
- **maxiter** (*int or None*) – Maximum number of iterations, or inf if it is set to None.
- **f_tol** (*float or None*) – The absolute tolerance of the norm of the output `f`.
- **f_rtol** (*float or None*) – The relative tolerance of the norm of the output `f`.
- **x_tol** (*float or None*) – The absolute tolerance of the norm of the input `x`.
- **x_rtol** (*float or None*) – The relative tolerance of the norm of the input `x`.
- **line_search** (*bool or str*) – Options to perform line search. If `True`, it is set to `"armijo"`.
- **verbose** (*bool*) – Options for verbosity

[1] B.A. van der Rotten, PhD thesis, "A limited memory Broyden method to solve high-dimensional systems of nonlinear equations". Mathematisch Instituut, Universiteit Leiden, The Netherlands (2003). https://web.archive.org/web/20161022015821/http://www.math.leidenuniv.nl/scripties/Rotten.pdf

method="scipy_gmres"

```
solve(..., method="scipy_gmres", *, min_eps=1e-09, max_niter=None)
```

Using SciPy's gmres method to solve the linear equation.

Keyword Arguments

- **min_eps** (*float*) – Relative tolerance for stopping conditions
- **max_niter** (*int or None*) – Maximum number of iterations. If `None`, default to twice of the number of columns of `A`.

method="gmres"

```
solve(..., method="gmres", *, posdef=None, max_niter=None, rtol=1e-06,␣
↪atol=1e-08, eps=1e-12)
```

Solve the linear equations using Generalised minial residual method.

Keyword Arguments

- **posdef** (*bool or None*) – Indicating if the operation $\mathbf{AX} - \mathbf{MXE}$ a positive definite for all columns and batches. If None, it will be determined by power iterations.
- **max_niter** (*int or None*) – Maximum number of iteration. If None, it is set to `int(1.5 * A.shape[-1])`
- **rtol** (*float*) – Relative tolerance for stopping condition w.r.t. norm of B
- **atol** (*float*) – Absolute tolerance for stopping condition w.r.t. norm of B
- **eps** (*float*) – Substitute the absolute zero in the algorithm's denominator with this value to avoid nan.

11.2 symeig

`xitorch.linalg.`**`symeig`**(*A: xitorch._core.linop.LinearOperator*, *neig: Optional[int] = None*, *mode: str = 'lowest'*, *M: Optional[xitorch._core.linop.LinearOperator] = None*, *bck_options: Mapping[str, Any] = {}*, *method: Optional[Union[str, Callable]] = None*, ***fwd_options*) → Tuple[torch.Tensor, torch.Tensor]

Obtain `neig` lowest eigenvalues and eigenvectors of a linear operator,

$$\mathbf{AX} = \mathbf{MXE}$$

where $\mathbf{A}$, $\mathbf{M}$ are linear operators, $\mathbf{E}$ is a diagonal matrix containing the eigenvalues, and $\mathbf{X}$ is a matrix containing the eigenvectors. This function can handle derivatives for degenerate cases by setting non-zero `degen_atol` and `degen_rtol` in the backward option using the expressions in[1].

Parameters

- **A** (`xitorch.LinearOperator`) – The linear operator object on which the eigenpairs are constructed. It must be a Hermitian linear operator with shape `(*BA, q, q)`
- **neig** (*int or None*) – The number of eigenpairs to be retrieved. If `None`, all eigenpairs are retrieved

[1] Muhammad F. Kasim, "Derivatives of partial eigendecomposition of a real symmetric matrix for degenerate cases". arXiv:2011.04366 (2020) https://arxiv.org/abs/2011.04366

- **mode** (*str*) – "lowest" or "uppermost"/"uppest". If "lowest", it will take the lowest neig eigenpairs. If "uppest", it will take the uppermost neig.
- **M** (xitorch.LinearOperator) – The transformation on the right hand side. If None, then M=I. If specified, it must be a Hermitian with shape (*BM, q, q).
- **bck_options** (*dict*) – Method-specific options for *solve()* which used in backpropagation calculation with some additional arguments for computing the backward derivatives:
  - degen_atol (float or None): Minimum absolute difference between two eigenvalues to be treated as degenerate. If None, it is torch.finfo(dtype).eps**0.6. If 0.0, no special treatment on degeneracy is applied. (default: None)
  - degen_rtol (float or None): Minimum relative difference between two eigenvalues to be treated as degenerate. If None, it is torch.finfo(dtype).eps**0.4. If 0.0, no special treatment on degeneracy is applied. (default: None)

  Note: the default values of degen_atol and degen_rtol are going to change in the future. So, for future compatibility, please specify the specific values.
- **method** (*str or callable or None*) – Method for the eigendecomposition. If None, it will choose "exacteig".
- ****fwd_options** – Method-specific options (see method section below).

Returns It will return eigenvalues and eigenvectors with shapes respectively (*BAM, neig) and (*BAM, na, neig), where *BAM is the broadcasted shape of *BA and *BM.

Return type tuple of tensors (eigenvalues, eigenvectors)

References

method="exacteig"

```
symeig(..., method="exacteig")
```

Eigendecomposition using explicit matrix construction. No additional option for this method.

Warning:

- As this method construct the linear operators explicitly, it might requires a large memory.

method="davidson"

```
symeig(..., method="davidson", *, max_niter=1000, nguess=None, v_init="randn",
↪ max_addition=None, min_eps=1e-06, verbose=False)
```

Using Davidson method for large sparse matrix eigendecomposition[2].

Parameters

- **max_niter** (*int*) – Maximum number of iterations
- **v_init** (*str*) – Mode of the initial guess ("randn", "rand", "eye")
- **max_addition** (*int or None*) – Maximum number of new guesses to be added to the collected vectors. If None, set to neig.

[2] P. Arbenz, "Lecture Notes on Solving Large Scale Eigenvalue Problems" http://people.inf.ethz.ch/arbenz/ewp/Lnotes/chapter12.pdf

- **min_eps** (*float*) – Minimum residual error to be stopped
- **verbose** (*bool*) – Option to be verbose

References

11.3 svd

`xitorch.linalg.`**`svd`**(*A: xitorch._core.linop.LinearOperator*, *k: Optional[int] = None*, *mode: str = 'uppest'*, *bck_options: Mapping[str, Any] = {}*, *method: Optional[Union[str, Callable]] = None*, ***fwd_options*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Perform the singular value decomposition (SVD):

$$\mathbf{A} = \mathbf{U\Sigma V}^H$$

where $\mathbf{U}$ and $\mathbf{V}$ are semi-unitary matrix and $\mathbf{\Sigma}$ is a diagonal matrix containing real non-negative numbers. This function can handle derivatives for degenerate singular values by setting non-zero `degen_atol` and `degen_rtol` in the backward option using the expressions in[1].

Parameters

- **A** (`xitorch.LinearOperator`) – The linear operator to be decomposed. It has a shape of `(*BA, m, n)` where `(*BA)` is the batched dimension of `A`.
- **k** (*int or None*) – The number of decomposition obtained. If `None`, it will be `min(*A.shape[-2:])`
- **mode** (*str*) – `"lowest"` or `"uppermost"`/`"uppest"`. If `"lowest"`, it will take the lowest `k` decomposition. If `"uppest"`, it will take the uppermost `k`.
- **bck_options** (*dict*) – Method-specific options for *`solve()`* which used in backpropagation calculation with some additional arguments for computing the backward derivatives:
  - `degen_atol` (`float` or None): Minimum absolute difference between two singular values to be treated as degenerate. If None, it is `torch.finfo(dtype).eps**0.6`. If 0.0, no special treatment on degeneracy is applied. (default: None)
  - `degen_rtol` (`float` or None): Minimum relative difference between two singular values to be treated as degenerate. If None, it is `torch.finfo(dtype).eps**0.4`. If 0.0, no special treatment on degeneracy is applied. (default: None)

  Note: the default values of `degen_atol` and `degen_rtol` are going to change in the future. So, for future compatibility, please specify the specific values.
- **method** (*str or callable or None*) – Method for the svd (same options for *`symeig()`*). If `None`, it will choose `"exacteig"`.
- ****fwd_options** – Method-specific options (see method section below).

Returns It will return `u, s, vh` with shapes respectively `(*BA, m, k)`, `(*BA, k)`, and `(*BA, k, n)`.

Return type tuple of tensors (u, s, vh)

[1] Muhammad F. Kasim, "Derivatives of partial eigendecomposition of a real symmetric matrix for degenerate cases". arXiv:2011.04366 (2020) https://arxiv.org/abs/2011.04366

Note: It is a naive implementation of symmetric eigendecomposition of `A.H @ A` or `A @ A.H` (depending which one is cheaper)

References

`method="exacteig"`

```
svd(..., method="exacteig")
```

Eigendecomposition using explicit matrix construction. No additional option for this method.

Warning:

- As this method construct the linear operators explicitly, it might requires a large memory.

`method="davidson"`

```
svd(..., method="davidson", *, M=None, max_niter=1000, nguess=None, v_init=
↪"randn", max_addition=None, min_eps=1e-06, verbose=False)
```

Using Davidson method for large sparse matrix eigendecomposition[2].

Parameters

- **max_niter** (*int*) – Maximum number of iterations
- **v_init** (*str*) – Mode of the initial guess (`"randn"`, `"rand"`, `"eye"`)
- **max_addition** (*int or None*) – Maximum number of new guesses to be added to the collected vectors. If None, set to `neig`.
- **min_eps** (*float*) – Minimum residual error to be stopped
- **verbose** (*bool*) – Option to be verbose

References

[2] P. Arbenz, "Lecture Notes on Solving Large Scale Eigenvalue Problems" http://people.inf.ethz.ch/arbenz/ewp/Lnotes/chapter12.pdf

CHAPTER

TWELVE

XITORCH.INTERPOLATE

12.1 Interp1D

class xitorch.interpolate.**Interp1D**(*x: torch.Tensor, y: Optional[torch.Tensor] = None, method: Optional[Union[str, Callable]] = None, assume_sorted: bool = False, **fwd_options*)

1D interpolation class. When initializing the class, the *x* must be specified and *y* can be specified during initialization or later.

Parameters

- **x** (*torch.Tensor*) – The position of known values in tensor with shape `(..., nr)`
- **y** (*torch.Tensor or None*) – The values at the given position with shape `(..., nr)`. If `None`, it must be supplied during `__call__`
- **method** (*str or callable or None*) – Interpolation method. If None, it will choose `"cspline"`.
- **assume_sorted** (*bool*) – Assume x is sorted monotonically increasing. If False, then it sorts the input x and y first before doing the interpolation.
- ****fwd_options** – Method-specific options (see method section below)

Note: Batched `x` and `xq` is only implemented if there is no extrapolation involved.

method="cspline"

```
Interp1D(..., method="cspline", *, y=None, bc_type=None, extrap=None)
```

Perform 1D cubic spline interpolation for non-uniform `x`[1][2].

Keyword Arguments

- **bc_type** (*str or None*) – Boundary condition:
  - `"not-a-knot"`: The first and second segments are the same polynomial
  - `"natural"`: 2nd grad at the boundaries are 0
  - `"clamped"`: 1st grad at the boundaries are 0
  - `"periodic"`: periodic boundary condition (*new in version 0.2*)

[1] SplineInterpolation on Wikipedia, https://en.wikipedia.org/wiki/Spline_interpolation#Algorithm_to_find_the_interpolating_cubic_spline)

[2] Carl de Boor, "A Practical Guide to Splines", Springer-Verlag, 1978.

If `None`, it will choose `"not-a-knot"`

- **`extrap`** (*`int, float, 1-element torch.Tensor, str, or None`*) – Extrapolation option:
  - `int`, `float`, or 1-element `torch.Tensor`: it will pad the extrapolated values with the specified values
  - `"mirror"`: the extrapolation values are mirrored
  - `"periodic"`: periodic boundary condition. `y[...,0] == y[...,-1]` must be fulfilled for this condition.
  - `"bound"`: fill in the extrapolated values with the left or right bound values.
  - `"nan"`: fill the extrapolated values with nan
  - callable: apply this extrapolation function with the extrapolated positions and use the output as the values
  - `None`: choose the extrapolation based on the `bc_type`. These are the pairs:
    - `"clamped"`: `"mirror"`
    - other: `"nan"`

  Default: `None`

References

`method="linear"`

```
Interp1D(..., method="linear", *, y=None, extrap=None)
```

Perform 1D linear interpolation for non-uniform `x`.

Keyword Arguments **`extrap`** (*`int, float, 1-element torch.Tensor, str, or None`*) – Extrapolation option:

- `int`, `float`, or 1-element `torch.Tensor`: it will pad the extrapolated values with the specified values
- `"mirror"`: the extrapolation values are mirrored
- `"periodic"`: periodic boundary condition. `y[...,0] == y[...,-1]` must be fulfilled for this condition.
- `"bound"`: fill in the extrapolated values with the left or right bound values.
- `"nan"`: fill the extrapolated values with nan
- callable: apply this extrapolation function with the extrapolated positions and use the output as the values
- `None`: choose the extrapolation based on the `bc_type`. These are the pairs:
  - `"clamped"`: `"mirror"`
  - other: `"nan"`

Default: `None`

`__call__`(*xq: torch.Tensor*, *y: Optional[torch.Tensor] = None*) → torch.Tensor

Parameters

- **xq** (*torch.Tensor*) – The position of query points with shape `(..., nrq)`.
- **y** (*torch.Tensor or None*) – The values at the given position with shape `(..., nr)`. If `y` has been specified during `__init__` and also specified here, the value of `y` given here will be ignored. If no `y` ever specified, then it will raise an error.

Returns The interpolated values with shape `(..., nrq)`.

Return type torch.Tensor

CHAPTER THIRTEEN

INDICES AND TABLES

- genindex
- search

INDEX