ipyopt

Gerhard Bräunlich

Jun 06, 2022

CONTENTS

CHAPTER

ONE

GETTING STARTED

Ipopt solves the following class of nonlinear programming problems (NLP):

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^n} f(\boldsymbol{x}) \\ \boldsymbol{g}_l \leq \boldsymbol{g}(\boldsymbol{x}) \leq \boldsymbol{g}_u \in \mathbb{R}^m \\ \boldsymbol{x}_l \leq \boldsymbol{x} \leq \boldsymbol{x}_u \in \mathbb{R}^n \end{aligned}$$

Lets start with a minimal example:

$$\begin{aligned} \boldsymbol{x} \in [-10, 10]^3 \subset \mathbb{R}^3 \\ f(\boldsymbol{x}) = x_0^2 + x_1^2 + x_2^2 \\ 0 \leq g(\boldsymbol{x}) = (x_0 - 1)^2 + x_1^2 + x_2^2 \leq 4 \in \mathbb{R}^1. \end{aligned}$$

That is, $\boldsymbol{x}_l = (-10, -10, -10)$, $\boldsymbol{x}_u = (10, 10, 10)$.

Step 1: Define the problem in Python

We first define the corresponding Python function for `f` and its derivative:

```
def f(x: np_array) -> float:
    out: float = numpy.sum(x**2)
    return out
```

```
def grad_f(x: np_array, out: np_array) -> np_array:
    out[()] = 2.0 * x
    return out
```

Note: For performance reasons, you have to write the value of `grad_f` into the `out` argument of the function (this way we can avoid unnecessary memory allocation).

Next, we define `g` and its derivative:

```
def g(x: np_array, out: np_array) -> np_array:
    """Constraint function: squared distance to (1, 0, ..., 0)"""
    out[0] = numpy.sum((x - _e_x) ** 2)
    return out
```

```
def jac_g(x: np_array, out: np_array) -> np_array:
    out[()] = 2.0 * (x - _e_x)
    return out
```

Here, The Jacobian $\mathrm{D}g = 2(x_0 - 1, x_1, x_2)$, thus the non zero slots (all slots in this case) are `(i,j) = (0,0), (0,1), (0,2)`. Translated into python code:

```
(numpy.array([0, 0, 0]), numpy.array([0, 1, 2]))
```

Note: While in the notation `(i,j) = (0,0), (0,1), (0,2)`, we use index pairs, you have to provide the collections of all `i` components (first tuple field) and the collection of all `j` components (second tuple field) as sparsity info arguments to ipyopt.

For maximal performance, we also can provide the Hessian of the Lagrangian $L(x) = \sigma f(x) + \langle \lambda, g(x) \rangle$, where $\langle \cdot, \cdot \rangle$ denotes the standard scalar product (in this case just the usual multiplication, as we are in $\mathbb{R}^1$). In our case:

$$\mathrm{Hess}\, L = \sigma \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix} + \lambda_0 \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix} = 2(\sigma + \lambda_0) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Therefore, we have nonzero entries at the diagonal, i.e. `(i,j) = (0,0), (1,1), (2,2)`. The corresponding sparsity info argument is

`(numpy.array([0, 1, 2]), numpy.array([0, 1, 2]))` and this is how to compute the non zero entries:

```
def h(
    _x: np_array, lagrange: np_array, obj_factor: float, out: np_array
) -> np_array:
    out[()] = numpy.full((_n,), 2.0 * (obj_factor + lagrange[0]))
    return out
```

Here, `lagrange` corresponds to λ and `obj_factor` to σ. If we don't provide the Hessian, Ipopt will numerically approximate it for us, at the price of some performance loss.

Now, we are ready to define the Python problem:

```
import ipyopt
nlp = ipyopt.Problem(
    n=2,
    x_l=numpy.array([-10.0, -10.0, -10.0]),
    x_u=numpy.array([10.0, 10.0, 10.]),
    m=1,
    g_l=numpy.array([0.0]),
    g_u=numpy.array([4.0]),
    sparsity_indices_jac_g=(numpy.array([0, 0, 0]), numpy.array([0, 1, 2])),
    sparsity_indices_h=(numpy.array([0, 1, 2]), numpy.array([0, 1, 2])),
    f,
    grad_f,
    g,
    jac_g,
    h
)
```

Step 2: Solve the problem

We will use $x_0 = (0.1, 0.1, 0.1)$ as initial guess.

```
x, obj, status = nlp.solve(x0=numpy.array([0.1, 0.1, 0.1]))
```

As a result, we should obtain the solution `x = (0.,0.,0.)`, `obj = f(x) = 0.` and `status = 0` (meaning, that Ipopt found the optimal solution within tolerance).

CHAPTER

TWO

SCIPY OPTIMIZE METHOD

ipyopt also comes with a `ipopt` method for **`scipy.optimize.minimize`_**:

```
result = scipy.optimize.minimize(
    fun=...,
    x0=...,
    jac=...,
    constraints=...,
    method=ipyopt.optimize.ipopt,
    ...
    )
```

Warning: The `ipopt` method differs in some points from the standard scipy methods:

- The argument `jac` is mandatory (explicitly use **`scipy.optimize.approx_fprime`_** if you want to numerically approximate it or see *Automatic symbolic derivatives / code generation* on how to auto differentiate symbolic expressions)
- `hess` is not the Hessian of the objective function `f` but the Hessian of the Lagrangian `L(x) = obj_factor * f(x) + lagrange * g(x)`, where `g` is the constraint residuals.
- The argument `constraints` is mandatory. It is also not a list as in usual scipy optimize methods, but a single `Constraint` instance.

If you are looking for a solution whose API is closer to the usual scipy interface, have a look at cyipopt.

2.1 Module optimize

CHAPTER

THREE

USING PYCAPSULES

Instead of passing pure Python callables to `ipyopt.Problem` which causes some overhead when calling the Python callables from C++, we also can pass C callbacks encapsulated in PyCapsule objects.

> **Warning: Segfaults**
>
> When working with C callbacks inside PyCapsule objects, ipyopt will always assume, that the C callbacks have the correct signature. If this is not the case, expect memory errors and thus crashes.

PyCapsule can be defined in C extensions. For an example on how to do this from scratch, see module.c.

A much more convenient way is to use Cython. Then you don't need to write boilerplate code to create a python module. All you still need to do is to define the objective function, the constraint residuals and their derivatives:

```
import cython
cdef extern from "stdbool.h":
    ctypedef bint bool

cdef api:
    bool f(int n, const double *x, double *obj_value, void* userdata):
        obj_value[0] = x[0] * x[3] * (x[0] + x[1] + x[2]) + x[2]
        return True

    bool grad_f(int n, const double *x, double *out, void *userdata):
        out[0] = x[0] * x[3] + x[3] * (x[0] + x[1] + x[2])
        out[1] = x[0] * x[3]
        out[2] = x[0] * x[3] + 1.0
        out[3] = x[0] * (x[0] + x[1] + x[2])
        return True

    bool g(int n, const double *x, int m, double *out, void *userdata):
        out[0] = x[0] * x[1] * x[2] * x[3]
        out[1] = x[0] * x[0] + x[1] * x[1] + x[2] * x[2] + x[3] * x[3]
        return True

    bool jac_g(int n, const double *x, int m, int n_out, double *out,
                      void *userdata):
        out[0] = x[1] * x[2] * x[3]
        out[1] = x[0] * x[2] * x[3]
        out[2] = x[0] * x[1] * x[3]
        out[3] = x[0] * x[1] * x[2]
```

(continues on next page)

(continued from previous page)

```
        out[4] = 2.0 * x[0]
        out[5] = 2.0 * x[1]
        out[6] = 2.0 * x[2]
        out[7] = 2.0 * x[3]
        out[8] = x[0] * x[1] * x[2] * x[3]
        return True

    bool h(int n, const double *x, double obj_factor, int m,
                   const double *lagrange, int n_out, double *out, void *userdata):
        out[0] = obj_factor * (2 * x[3])
        out[1] = obj_factor * (x[3])
        out[2] = 0
        out[3] = obj_factor * (x[3])
        out[4] = 0
        out[5] = 0
        out[6] = obj_factor * (2 * x[0] + x[1] + x[2])
        out[7] = obj_factor * (x[0])
        out[8] = obj_factor * (x[0])
        out[9] = 0
        out[1] += lagrange[0] * (x[2] * x[3])

        out[3] += lagrange[0] * (x[1] * x[3])
        out[4] += lagrange[0] * (x[0] * x[3])

        out[6] += lagrange[0] * (x[1] * x[2])
        out[7] += lagrange[0] * (x[0] * x[2])
        out[8] += lagrange[0] * (x[0] * x[1])
        out[0] += lagrange[1] * 2
        out[2] += lagrange[1] * 2
        out[5] += lagrange[1] * 2
        out[9] += lagrange[1] * 2
        return True
```

To just in time compile the `pyx`, file, you can use Cython's `pyximport`:

```
import pyximport
pyximport.install(language_level=3)
```

The compiled C extension can then be included. It will contain an attribute `__pyx_capi__` containing the PyCapsules:

```
from hs071_capsules import __pyx_capi__ as capsules

nlp = ipyopt.Problem(
    ...
    capsules["f"],
    capsules["grad_f"],
    capsules["g"],
    capsules["jac_g"],
    capsules["h"],
)
```

CHAPTER

FOUR

AUTOMATIC SYMBOLIC DERIVATIVES / CODE GENERATION

Probably the most convenient way to use ipyopt is to let sympy compute your derivatives automatically. This however is only possible out of the box for simple functions without exotic matrix operations / special functions (i.e. Bessel functions).

At this stage, ipyopt only ships a proof of concept to show that this is possible. Therefore the module `ipyopt.sym_compile` is to be considered experimental.

Its `array_sym` function can be used to declare symbolic vector valued variables to be used in the objective function `f` and the constraint residuals `g`. You then define expressions for `f` and `g`. Currently the variable used in this expression has to be exactly `x`. Choosing a different letter wont work in the current version. The expressions for `f` and `g` are sufficient. `ipyopt.sym_compile` will then be able to

- automatically build derivatives
- generate C code for all expressions
- compile the C code to a python extension
- load the python extension and return the contained PyCapsule objects in a dict.

The returned dict can directly be passed to `ipyopt.Problem`.

Here is Ipopt's `hs071` example reimplemented using this approach:

```
import numpy

import ipyopt
from ipyopt.sym_compile import array_sym, SymNlp

n = 4
m = 2

x = array_sym("x", n)

f = x[0] * x[3] * (x[0] + x[1] + x[2]) + x[2]
g = [x[0] * x[1] * x[2] * x[3], x[0] * x[0] + x[1] * x[1] + x[2] * x[2] + x[3] * x[3]]

c_api = SymNlp(f, g).compile()

nlp = ipyopt.Problem(
    n=n,
    x_l=numpy.ones(n),
    x_u=numpy.full(n, 5.0),
    m=m,
```

(continues on next page)

(continued from previous page)

```
    g_l=numpy.array([25.0, 40.0]),
    g_u=numpy.array([2.0e19, 40.0]),
    **c_api,
)
```

4.1 Module sym_compile

CHAPTER

FIVE

REFERENCE

5.1 Submodules

5.2 Module contents

5.3 ipyopt.optimize module

5.4 ipyopt.sym_compile module