Chumpy Documentation

Release 1.0

Matthew Loper

July 02, 2014

CONTENTS

CHAPTER

ONE

WHAT DOES CHUMPY DO?

Chumpy is a Python-based framework designed to handle the auto-differentiation problem, which is to evalute an expression and its derivatives with respect to its inputs, with the use of the chain rule. Auto-differentiation is neither purely symbolic, nor "differentation by finite differences."

Chumpy is intended to be "Numpy made differentiable:" it supports mostly the same interface as that supported by the popular Numpy library for scientific computing. Many alternative autodifferentiation frameworks exist; Chumpy was written for ease of use. By allowing the transparent, concise specification and optimization of objectives, we hope that Chumpy can help people solve problems. Other frameworks should be considered if second-order derivatives or compilation (to CPU or GPU) is important.

Both Numpy and Chumpy allow working with arrays: specifically creating, assigning, operating on, and doing linear algebra with arrays. At a high-level, Chumpy makes construction and local minimization of objectives easier. Specifically, it provides:

- Easy *problem construction* by using Numpy's application interface
- Easy *access to derivatives* via autodifferentiation
- Easy *local optimization methods* (12 of them: most of which use the derivatives)

Better Numpy tutorials are available elsewhere, but hopefully this will serve to introduce you to some of what is possible, both in Numpy and Chumpy. Much of this tutorial will be by example.

(Disclaimer: Chumpy relies heavily on Numpy and Scipy, and would not exist without them.)

1.1 Easy problem construction

Consider the construction of this quadratic function:

$$f(x, y, A) = x^T A x + y^2$$

In Numpy, the energy f can be expressed as follows, with some arbitrary initializations given to $\{x, y, A\}$:

```
import numpy as np
x, y, A = np.array([10,20,30]), np.array([5]), np.eye(3)
z = x.T.dot(A).dot(x)
f = z + y**2
```

In Chumpy, it can be expressed in the same way:

```
import chumpy as ch
x, y, A = ch.array([10,20,30]), ch.array([5]), ch.eye(3)
z = x.T.dot(A).dot(x)
f = z + y**2
```

But whereas Numpy is operating on (and assigning to) arrays in memory, Chumpy internally builds structures to facilitate differentiation and optimization.

Much more complicated functions can be expressed with both Numpy and Chumpy, including vectors, matrices, tensors, selection, and linear algebra. Over 100 functions in Numpy are also available and differentiated through in Chumpy, including SVD, least squares, determinants, tensor multiplication, and matrix inversion. Functions not yet available in Chumpy include eigenvalue decomposition, QR decomposition, and Kronecker products.

1.2 Easy access to derivatives

Now let us compute $\frac{\partial f}{\partial x}$. With Chumpy, this requires only one additional line:

```
df_x = f.dr_wrt(x)
```

Derivatives with respect to the other variables (intermediate and otherwise, vectors and matrices) are also easy to compute:

```
df_y = f.dr_wrt(y)
df_A = f.dr_wrt(A)
df_z = f.dr_wrt(z)
```

Of course, for this example, computing derivatives by hand is easy. The problem comes when more complex energies (or combinations of energies) is required; differentiation for each minor change to the objective, by hand, is not always a good use of time.

Depending on the constructed function, the Jacobian may take three forms:

- A 2D SciPy sparse matrix
- A 2D Numpy dense array
- "None", if an object is not differentiable with respect to the requested object

Answers to related questions (such as "why is the Jacobian 2D", or "why is the Jacobian sometimes dense and sometimes sparse") are available in Chapter 2.

1.3 Easy local optimization

Local optimization methods can benefit greatly from having built-in derivatives. Performing local optimization of f is also designed to be easy, and requires one more line:

```
ch.minimize(f, [x, y], method='bfgs')
```

Note that the independent variables here are not required to be $\{x, y\}$, but can be any subset of $\{x, y, A\}$.

That function can use one of 12 methods for optimization, each of which is implemented by SciPy's minimize method, and which can be specified by the inclusion of a 'method' parameter to ch.minimize. Those methods include Nelder-Mead, Powell, CG, BFGS, Newton-CG, Anneal, L-BFGS-B, TNC, COBYLA, SLSQP, trust-ncg. To be clear: with the exception of "dogleg", optimization is "outsourced" to SciPy, but a simple interface is provided so that function and derivative callbacks don't require explicit definition.

This also avoids the packing and unpacking (of both parameters and residuals) often necessary in the construction of callbacks for nonlinear optimization.

1.4 Numpy is still useful!

First: Chumpy uses Numpy a lot under the hood, and wouldn't exist without it. Even so, one might wonder: can I replace all my Numpy code with Chumpy code? Will everything just work and be better?

Here are some considerations:

- Chumpy always casts integers to floating-point, because it depends on values changing smoothly. Using Chumpy to store indices doesn't make much sense, so indices are best kept as Numpy arrays.
- Although Chumpy supports some interesting functions (svd, tensorinv, inv, lstsq), it certainly does not support all. Eigenvalue related functions, for example, are not yet differentiated through with Chumpy.
- Because it behaves more functionally (as in functional programming) than Numpy, results may be different.
- Complex numbers are not supported.

At a more abstract level, if you don't want derivatives and you don't want to optimize an objective, using Chumpy probably doesn't make sense.

CHAPTER

TWO

ADVANCED TOPICS

First, we will state some technical characteristics of Chumpy. Next we'll show a way of visualizing the structure of an expression. Finally, we'll introduce some examples of how to define (rather than compose) differentiable primitives with Chumpy.

2.1 Basic Characteristics

Chumpy is primarily a forward-mode, Jacobian producing autodifferentiation framework. Reverse-mode is partially supported, but much slower than it should be because some basic functions need implementation. Values and Jacobians are computed on demand, and recomputed in a lazy fashion: when assignment at the leaves of an expression tree occurs, caches in dependent parents are invalidated. Second-order and higher-order derivatives are not supported.

Its origin as a forward-mode framework comes partially from its use for Gauss-Newton Hessian approximation in least-squares nonlinear optimization, where the Jacobian is required. Reverse mode is "simulated" by Jacobian multiplication where not yet explicitly implemented.

Numpy is more procedural than Chumpy. In Numpy, operations on arrays result in new arrays that don't retain references to operands. Chumpy, on the other hand, constructs a tree out of the expressions you give it, and the values of some objects may change when others do. For example:

```
x = ch.array(5)
y = ch.array([5,7])
z = x + y
x[0] = 6 # This changes "z" in Chumpy, but not in Numpy!
```

This can be considered good or bad, but (given all the talk of Numpy compatibility) it must be considered as a "gotcha" in the case that Numpy and Chumpy code are expected to behave in the same way.

One question that may arise is why the Jacobian is always 2D. The answer is that sparse libraries available in Python do not have good support for sparse tensors. This means that if the input (or output) of a function is multidimensional, the rows (or columns) correspond to values as they would be unwrapped by row, i.e. row-major order.

One might also wonder why both dense and sparse matrices are needed. If Chumpy only returned sparse or dense matrices, this would restrict performance: sparse and dense matrices are fast in different scenarios. For example, differentiating through the addition of two vectors produces Jacobians that take the form of the identity, so instead of returning a contiguous dense Jacobian with mostly zeros, a sparse matrix is returned in this case. But interesting expressions are typically composed of many operations. Currently, the rule is that if any dense matrices are encountered in a chain, the resulting Jacobian becomes dense, but other heuristics are possible.

2.2 Visualizing your Terms

As expressions become more complicated, you may want ways of visualizing their structure. One simple method for doing so is as follows (note: this expression is nowhere near complicated, but is useful for illustration):

```
a, b, c, d = Ch(5), Ch(6), Ch(7), Ch(8)
y = a+b/(c*d)
y.show_tree() # pops up a window
```

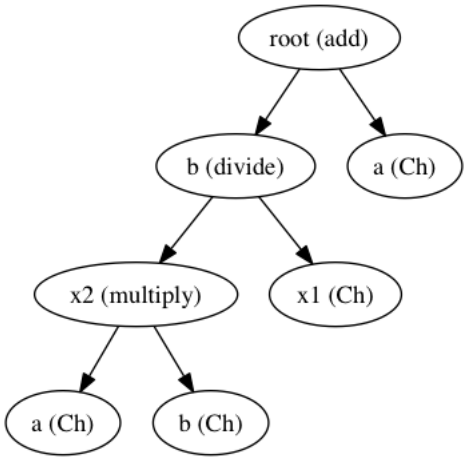

Figure 2.1: An expression tree.

Because Python doesn't track variable names well, the names in the tree are unrelated to the names you gave them. This makes it more difficult to decipher the tree. For that reason, you may wish to label parts of the expression, and show it again:

```
a.label = 'cats'
b.label = 'dogs'
c.label = 'kittens'
d.label = 'goats'
y.show_tree()
```

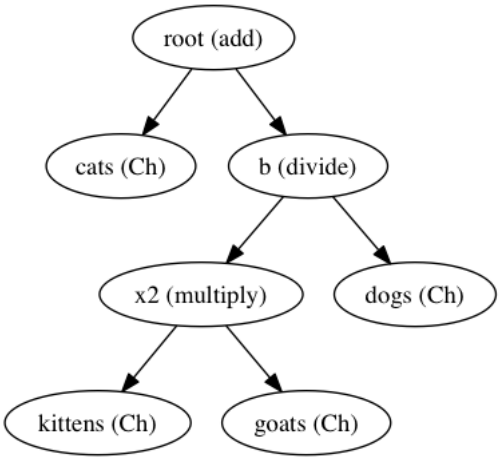

Figure 2.2: An expression tree with labeled items.

2.3 Implementing Chumpy Objects

Many Chumpy primitives already exist (for addition, subtraction, matrix multiplication, on and on), but you may wish to define your own: for efficiency or because it can't be created by composing existing primitives.

Composing expressions from the bottom up is not always feasible or desirable. For that reason, you may wish to define your own classes. For example, let us imagine for the moment that the Sin primitive does not exist. Here is an implementation, which supports scalars, vectors and arrays:

```
from chumpy import Ch
import scipy.sparse as sp
class Sin(Ch):
    dterms = ('x',)

    def compute_r(self):
        return np.sin(self.x.r)

    def compute_dr_wrt(self, wrt):
        if wrt is self.x:
            values = np.cos(self.x.r)
            return sp.diags([values.ravel()], [0])
```

Doing this required subclassing Ch, defining a tuple of all the terms we're differentiable with respect to, and defining methods that compute values and derivatives. And one may use it as follows:

```
x1 = Ch(10)
result = Sin(x1) # or "result = Sin(x=x1)"
print result
print result.dr_wrt(x1)
```

Some classes may also have terms that they're not differentiable with respect to. Although Chumpy already allows selection, let us imagine it does not. The following example allows the specification of values and indices to select, and returns the unraveled, selected input:

```
from chumpy import Ch
import scipy.sparse as sp
class Select(Ch):

    dterms = ('x',)
    terms = ('indices',)

    def compute_r(self):
        return self.x.r.ravel()[self.indices]

    def compute_dr_wrt(self, wrt):
        return sp.csc_matrix(((np.arange(self.x.size), self.indices), np.ones(self.indices.size)))
```

To summarize, our 'dterms' contains strings with attributes we declare ourselves to be differentiable with respect to, and 'terms' includes strings with attributes we're not differentiable with respect to. Assignment to anything in 'terms' or 'dterms' invalidates cache for an object and its parents.

And it can be used as follows:

```
x1 = ch.arange(20)/2.
selected = Select(x=x1, indices=[1,2,5])
print selected # prints .5, 1, 2.5
selected.indices = [10,20]
print selected # prints 5, 10
```

```
print selected.dr_wrt(selected.indices) # prints None
print selected.dr_wrt(selected.x) # returns sparse matrix
```

2.4 Caching helpers

One of the tenets of Chumpy is lazy execution. For that reason, some facilities have been added to allow computation to happen only when absolutely necessary. Some of these happen behind the scenes; for example, the result and derivatives of a Chumpy object are cached until one or more of its terms or dterms is changed.

But there are also some facilities directly exposed to the user.

The first is the use of the "depends_on" decorator. It exposes a property which is only recomputed on demand if one or more of its dependencies is changed; each dependency is indicated by its string name. For example, in the last case, the use of np.ones during every call implies repeated memory allocation for a vector whose contents may not change often. We can change it as follows:

```
from chumpy import Ch
import scipy.sparse as sp
class Select(Ch):

    dterms = ('x',)
    terms = ('indices',)

    def compute_r(self):
        return self.x.r.ravel()[self.indices]

    @depends_on('indices')
    def ones(self):
        return np.ones(self.indices.size)

    def compute_dr_wrt(self, wrt):
        return sp.csc_matrix(((np.arange(self.x.size), self.indices), self.ones))
```

Usage is the same as before.

Another is the use of the "on_changed' method, which offers a different style. Say you have a hidden value that you only want to recompute when an external term is changed, and only right before `compute_r` or `compute_dr_wrt` is called. Here we use it for the same purpose as before: to recompute something when necessary:

```
from chumpy import Ch
import scipy.sparse as sp
class Select(Ch):

    dterms = ('x',)
    terms = ('indices',)

    def compute_r(self):
        return self.x.r.ravel()[self.indices]

    def on_changed(self, which):
        if 'indices' in which: # which is always a nonempty list of strings
            self.ones = np.ones(self.indices.size)

    def compute_dr_wrt(self, wrt):
        return sp.csc_matrix(((np.arange(self.x.size), self.indices), self.ones))
```

Note that on_changed is called only right before `compute_r` or `compute_dr_wrt`, not right when something is changed.