Lab 15

Symbolic and Automatic Differentiation in Python

Lab Objective: *Python is good for more than just analysis of numerical data. There are several packages available which allow symbolic and automatic computation in Python, two of which are SympPy and autograd. This lab should teach you to do some basic manipulations in SymPy and autograd as well as their applications in differentiation. Keep in mind, however, that this introduction is not comprehensive.*

SymPy

SymPy is designed to be a fully featured computer algebra system in Python. It is actually used for much of the symbolic computation in Sage. An example of what we mean by *symbolic computation* is the following:

```
>>> import sympy as sy
>>> x = sy.symbols('x')             # Define a symbolic variable.
>>> sy.expand((x+1)**10)            # Expand an expression with that variable.
x**10 + 10*x**9 + 45*x**8 + 120*x**7 + 210*x**6 + 252*x**5 + 210*x**4 + 120*x**3 ↩
    + 45*x**2 + 10*x + 1
```

When used properly, this package can simplify large amounts of algebra for you. This can be incredibly useful in a wide variety of situations. As you may have guessed, such packages generally have a wide variety of features. The official documentation for SymPy is found at `http://sympy.org/en/index.html`.

Some Convenient Tools

SymPy includes a simplified plotting wrapper around Matplotlib.

```
>>> x = sy.symbols('x')
>>> expr = sy.sin(x) * sy.exp(x)
>>> sy.plot(expr, (x, -3, 3))
```

which plots $\sin(x)e^x$ for values of x from -3 to 3.

SymPy also has several nice options for printing equations. If you want to get a rough idea of what the equation looks like, you can use `sy.pprint()`. It can interface with the IPython Notebook to display the formula more clearly as well. If you are using the IPython Notebook, you can enable pretty printing by loading the extension that comes with SymPy. In SymPy 7.2 this is done as follows:

```
%load_ext sympy.interactive.ipythonprinting
```

In SymPy 7.3 it is:

```
import sympy as sy
sy.init_printing()
```

SymPy has many more useful features. Figure 15.1 shows a screenshot of SymPy's special printing in the IPython notebook. If at some point you need to write a formula in LaTeX, the function `sy.latex()` can convert a SymPy symbolic expression to LaTeX for you.

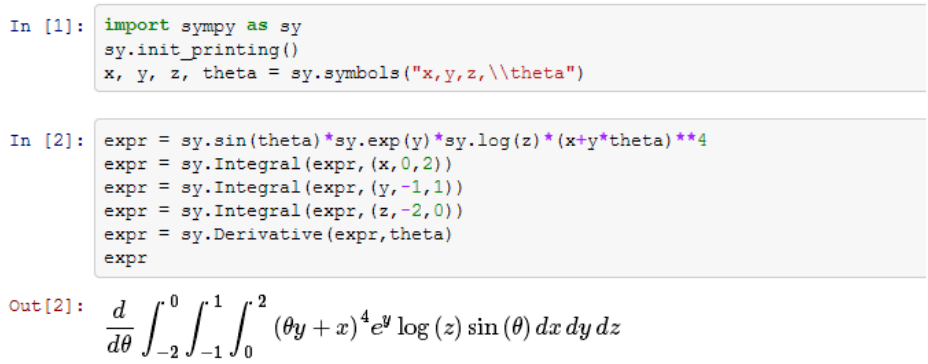

Figure 15.1: A screenshot showing how SymPy can interface with the IPython Notebook to display equations nicely.

Basic Number Types

SymPy has some good built in datatypes which can be used to represent rational numbers and arbitrary precision floating point numbers. Arbitrary precision floating point operations are supported through the package `mpmath`. These can be useful if you need to do computation to a very high precision, or to avoid possible overflow error in computation. They are, however, much more costly to compute.

You can declare a rational number $\frac{a}{b}$ using `sy.Rational(a, b)`. A real number r of precision n can be declared using $sy.Float(r, n)$.

A nice example of the use of these datatypes is the following function which computes π to the nth digit.

```
def mypi(n):
    #calculates pi to n decimal points.
    tot = sy.Rational(0, 1)
    term = 1
```

```
    bound = sy.Rational(1, 10)**(n+1)
    i = 0
    while bound <= term:
        term = 6 * sy.Rational(sy.factorial(2*i), 4**i*(sy.factorial(i))**2*(2*i↩
            +1)*2**(2*i+1))
        tot += term
        i += 1
    return sy.Float(tot, n)
```

This function works by evaluating the Taylor Series for $6 \arcsin\left(\frac{1}{2}\right)$. We used a rather crude error estimate to ensure that we were close enough to break the loop.

Problem 1. Write a function `myexp` that takes an integer n as a parameter and evaluates e to the nth digit, using an approach similar to the code given above. Use the series

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

Use the same condition as above to determine when to break your while loop.

Symbolic Manipulation

In SymPy, you need to declare symbolic objects before you use them. To define a symbol x, we write `x = sy.symbols('x')`. This can also be used to define multiple variables at once, as in `x, y, z = sy.symbols('x,y,z')`. The string form of each variable on the right is used for showing expressions that involve the variable. You will need to be careful about capitalization and that last 's' in the name of the function. Calling `sy.Symbol()` will allow you to create a single symbolic variable, but `sy.symbol` is a submodule and cannot be called at all.

SymPy can be used to solve difficult expressions for given variables. Not everything can be solved, but SymPy's algorithms are pretty good and can often save a great deal of time. We consider the following equation:

$$\frac{w}{w-x} + \frac{x}{x-y} + \frac{y}{y-z} + \frac{z}{z-w} = 0$$

Say we need an explicit solution for a given variable. Since the expression is symmetrical, it doesn't really matter which one, but we can solve this in the following way:

```
import sympy as sy
w, x, y, z = sy.symbols('w, x, y, z')
expr = w/(w-x) + x/(x-y) + y/(y-z) + z/(z-w)
sy.solve(expr, w)
```

In any variable, this expression is quadratic, but each coefficient will depend on the other 3 variables. This would be a terrible pain to do by hand, but SymPy can take care of that for us. It is worth noting that SymPy cannot do everything, but if we keep in mind what we are doing and are familiar with the tools it has, it can make

complex algebraic operations a great deal faster. We should also note that `solve()` returns a list of expressions. If we want an expression to work with, we must take an item from the list.

In this particular example, you will notice that we only used a symbolic expression and not a full equation. What SymPy did was set the expression equal to zero, solve for the variable we wanted, then return the result. SymPy also supports equation objects (and inequalities), but this approach is often easier. If you need to declare equations, they can be declared using something like `equation = sy.Eq(x, y)` which represents the equation $x = y$

Problem 2. Use SymPy to solve the equation $y = e^x + x$ for x. The SymPy syntax for e^x is simpy `sy.exp(x)`, provided that x has been initialized as a symbol. The answer will be in terms of the Lambert W function, which is a special function available in most major symbolic math libraries. It is included in both SciPy and SymPy and is defined as the inverse of $y = xe^x$.

SymPy can also be used to expand and simplify different symbolic expressions. As an example we will simplify the expression

$$\frac{wx^2y^2 - wx^2 - wy^2 + w - x^2y^2z + 2x^2y^2 + x^2z - 2x^2 + y^2z - 2y^2 - z + 2}{wxy - wx - wy + w - xyz + 2xy + xz - 2x + yz - 2y - z + 2}$$

```
>>> w, x, y, z=sy.symbols('w, x, y, z')
>>> expr = (w*x**2*y**2 - w*x**2 - w*y**2 + w - x**2*y**2*z + 2*x**2*y**2 + x**2*↩
     z - 2*x**2 + y**2*z - 2*y**2 - z + 2)/(w*x*y - w*x - w*y + w - x*y*z + 2*x*y↩
      + x*z - 2*x + y*z - 2*y - z + 2)
>>> expr.simplify()
x*y + x + y + 1
```

`simplify()` is the general simplification method for a SymPy expression. It can be called as a function from the module, i.e. as `sy.simplify()` or it can be used as a method for an object as in the example above. You can also tell SymPy to do more specific types of simplification, for example if you want to factor an expression, you can use `factor()`. If you want to tell it to expand an expression you can use `expand()`. To focus purely on simplifying the trigonometric aspects of the expression you can use `trigsimp()`. If you want SymPy to cancel variable expressions in the numerator and denominator of all rational sub-expressions, use `cancel()`. There are several other kinds of algebraic manipulations you can do in SymPy, see the documentation for a more comprehensive list. Many of these more important functions in SymPy are also available as methods to expressions. This is the case with all of the above examples.

Be sure to be careful when writing out symbolic expressions. Python floating point numbers are still floating point numbers and Python integers are still integers. A simple example is the expression `(3/4)*sy.sin(x)` which evaluates to 0, and should probably have been written as `3*sy.sin(x)/4` or `sy.Rational(3,4)*sy.sin(x)`. Be careful about using floating points as well. The expression `(3./4.)*sy.sin(x)` will evaluate to `.75*sy.sin(x)`, and the floating point calculations will carry through to the rest of your symbolic work with that expression.

Another useful feature is substitution. Substitution can be done using the `subs()` method of an expression. You can substitute numbers and variables in for variables and even expressions. For example, if you want to see what an expression looks like if x is set to zero, you can use:

```
>>> x, y = sy.symbols('x,y')
>>> expr = sy.sin(x)*sy.cos(x)*sy.exp(y)*(x**3+y)**4
>>> expr.subs(x, 0)
0
```

where `expr` is the expression you have already defined. Note that none of these operations modify the expression in place. They return a modified version of the expression, but do not actually change the original.

Substitution also can be used (to some extent) to substitute one expression for another. For example, if you want to apply the double angle identity to replace products of sines and cosines, you could use the following:

```
>>> expr.subs(sy.sin(x) * sy.cos(x), sy.sin(2*x)/2)
(x**3 + y)**4*exp(y)*sin(2*x)/2
```

If you want to eliminate higher powers of a variable in an expression you can use something like:

```
>>> expr.subs(x**3, 0)
y**4*exp(y)*sin(x)*cos(x)
```

which will eliminate all terms of the expression involving x^3. At present time this will not eliminate terms involving x^4 or higher powers of x that are not divisible by 3.

If you want to substitute values in for multiple variables, you can either use a dictionary or a list of tuples for each variable and value.

```
>>> expr.subs({x:1.,y:2.})
272.113412745844
>>> expr.subs([(x,1.),(y,2.)])
272.113412745844
```

Calculus in SymPy

SymPy can also be used to take limits, integrals, and derivatives. Again, this can be very helpful when doing things that would be difficult to do by hand. For example, the following equation takes the 20th partial derivative with respect to x of

$$\prod_{i=1}^{23}(x+iy)$$

```
x, y, i = sy.symbols('x, y, i')
expr = sy.product((x+i*y), (i, 1, 23))
expr = expr.expand()
expr.diff(x, 20)
```

We can also integrate difficult things, for example, if we want to integrate $e^x \sin(x) \sinh(x)$, this can be done with one line in SymPy.

```
sy.Integral(sy.sin(x) * sy.exp(x) * sy.sinh(x), x).doit()
```

Notice the `.doit()` method. This tells SymPy to evaluate all derivatives, integrals, limits, etc. inside the expression.

Derivatives can be taken using the `sy.Derivative()` function, or the `.diff()` method of a SymPy expression. This can be done like this:

```
>>> x, y = sy.symbols('x,y')
>>> expr = sy.sin(x)*sy.cos(x)*sy.exp(y)*(x**3+y)**4
>>> sy.Derivative(expr,x).doit()
12*x**2*(x**3 + y)**3*exp(y)*sin(x)*cos(x) - (x**3 + y)**4*exp(y)*sin(x)**2 + (x↩
     **3 + y)**4*exp(y)*cos(x)**2
```

or, equivalently, like this:

```
>>> expr.diff(x)
12*x**2*(x**3 + y)**3*exp(y)*sin(x)*cos(x) - (x**3 + y)**4*exp(y)*sin(x)**2 + (x↩
     **3 + y)**4*exp(y)*cos(x)**2
```

You can find the 3rd derivative by doing the following:

```
expr.diff(x, 3)
```

The second argument is the number of derivatives to take. If it is omitted, one derivative will be taken. You can also pass additional variables as arguments, for example, if we want to find $\frac{\partial}{\partial x}\left(\frac{\partial}{\partial y}\sin(xy)\right)$, we can do it in this way:

```
expr = sy.sin(x*y)
expr.diff(x,y)
```

These tricks also work with integrals, with the exception of taking multiple integrals in the same variable. You can integrate two different variables along two different bounds, as in:

```
sy.integrate(y**2*x**2, (x, -1, 1), (y, -1, 1))
```

and you can integrate with respect to multiple variables, as in:

```
sy.integrate(y**2 * x**2, x, y)
```

Problem 3. Use SymPy to symbolically evaluate

$$\int_0^\infty \sin\left(x^2\right) dx$$

In SymPy, positive infinity is represented by the object `sy.oo`

Problem 4. Use SymPy to calculate the derivate of $e^{\sin(\cos(x))}$ at $x = 1$. Time how long it takes to compute this derivative using SymPy as well as centered difference quotients and calculate the error for each approximation.

You can also use SymPy to solve some sorts of basic ordinary differential equations. This will solve the equation $y_{xx} - 2 * y_x + y = \sin(x)$

```
x = sy.symbols('x')
f = sy.Function('f')
eq = sy.Eq(f(x).diff(x, 2) - 2*f(x).diff(x) + f(x), sy.sin(x))
sy.dsolve(eq)
```

or, equivalently,

```
x = sy.symbols('x')
f = sy.Function('f')
expr = f(x).diff(x, 2) - 2*f(x).diff(x) + f(x) - sy.sin(x)
sy.dsolve(expr)
```

Problem 5. Use SymPy to solve the following differential equation:

$$y_{xxxxxx} + 3y_{xxxx} + 3y_{xx} + y = \\ x^{10}e^x + x^{11}\sin(x) + x^{12}e^x\sin(x) - x^{13}\cos(2x) + x^{14}e^x\cos(3x)$$

You may recall from your last class on differential equations that this sort of problem is solved by the method of undetermined coefficients. Imagine how terrible this would be to do by hand!

SymPy can also be used to compute the Jacobian of a matrix using the `.jacobian()` method, which takes in either a list or a matrix of the variables. The Jacobian of $f(x, y) = \begin{bmatrix} x^2 \\ x + y \end{bmatrix}$ is found by doing the following:

```
x,y = sy.symbols('x,y')
F = sy.Matrix([x**2,x+y])
F.jacobian([x,y])
```

In addition, SymPy includes several integral transforms, such as the Laplace, Fourier, Sine, and Cosine Transforms. SymPy also allows you to do simple separation of variables on PDEs, Taylor Series, Laurent Series, Fourier Series, and many, *many* other things.

Autograd

Autograd is a package that allows you to automatically differentiate Python and NumPy code. Unlike SymPy which has many diverse applications, autograd is used solely in differentiation. Autograd is very useful in machine learning.

Autograd is installed by running the following command in the terminal:

```
pip install autograd
```

The following code computes the derivative of $e^{\sin(\cos(x))}$ at $x = 1$ using autograd.

```
>>> from autograd import grad
>>> import autograd.numpy as np
>>> g = lambda x: np.exp(np.sin(np.cos(x)))
>>> grad_g = grad(g)
>>> grad_g(1.)
-1.20697770398
```

There are a couple of things to note from this example. `autograd.numpy` is a thinly-wrapped NumPy. Also, `grad()` returns a function that computes the gradient of your original function. This new function which returns the gradient accepts the same parameters as the original function.

When there are multiple variables, the parameter `argnum` allows you to specify with respect to which variable you are computing the gradient.

```
>>> f = lambda x,y: 3*x*y + 2*y - x
>>> grad_f = grad(f, argnum=0) #gradient with respect to the first variable
>>> grad_f(.25,.5)
0.5
>>> grad_f = grad(f, argnum=1) #gradient with respect to the second variable
>>> grad_fun(.25,.5)
2.75
```

Finding the gradient with respect to multiple variables can by done using `multigrad()` by specifying which variables in the `argnums` parameter.

```
>>> grad_fun = autograd.multigrad(function, argnums=[0,1])
>>> grad_fun(.25,.5)
(0.5, 2.75)
```

Problem 6. Use autograd to compute the derivative of $f(x) = \ln\sqrt{\sin(\sqrt{x})}$ at $x = \frac{\pi}{4}$. Time how long it takes to compute this derivative using autograd, SymPy, and the centered difference quotient. Calculate the error for each approximation.

Computing the Jacobian is also possible using autograd. The following shows how to find the Jacobian of $f(x, y) = \begin{bmatrix} x^2 \\ x + y \end{bmatrix}$ evaluated at (1,1):

```
from autograd import jacobian
f = lambda x: np.array([x[0]**2, x[0]+y[0]])
jacobian_f = jacobian(f)
jacobian_f(np.array([1.,1.]))
```

It is important to remember that if you use `grad()` the output must be a scalar, whereas using `jacobian()` allows you to compute the gradient of a vector.

Problem 7.

Let $f : \mathbb{R}^2 \to \mathbb{R}^2$ be defined by

$$f(x, y) = \begin{bmatrix} e^x \sin(y) + y^3 \\ 3y - \cos(x) \end{bmatrix}$$

Find the Jacobian function using SymPy and autograd. Time how long it takes to compute each Jacobian at $(x, y) = (1, 1)$.

To learn more about autograd visit `https://github.com/HIPS/autograd`.