Milestone 1

Group Name: Exponentials

Introduction

Automatic Differentiation (AD) is a numerical method used to evaluate derivatives of functions. AD is superior to classical computer-based differentiation procedures such as finite difference method in that AD is accurate up to machine precision and does not require the user to “tune” parameters such as step-size. AD is also much more efficient than symbolic differentiation.

This project implements the forward mode of AD using elementary operator override methods. In particular, our library allows users to evaluate the gradient/Jacobian of multivariable scalar/vector functions using AD up to machine precision. We also provide flexibility in computing either the entire Jacobian matrix J or its action on a particular vector p, i.e. Jp. The latter option is beneficial when the entire J is not required in application: computation of Jp is much more efficient due to the way we implement AD.

Background

Automatic differentiation (AD) is a method for evaluating derivatives of functions. The method is fundamentally different from other computational differentiation approaches which use approximative finite difference methods or symbolic differentiation, and it overcomes limitations of both approaches. AD requires shorter computation times than symbolic differentiation and computes derivatives up to machine precision while finite difference methods are prone to rounding errors or have limited robustness.

AD has a forward mode and a reverse mode. Here we will only describe the forward mode.

Let the function of interest be $f : R^n \rightarrow R^m$, consisting of elementary functions ϕ_i. Then, forward AD computes $J_f(c) \cdot p$ where $J_f(c)$ (abbr. J) is the Jacobian matrix of function f evaluated at c and p is arbitrary vectors. Note that the entire Jacobian J may

be evaluated by computing the i-th column of J as $J_f(c) \cdot e_i$ where e_i denotes a vector with i-th entry set to 1 and 0 otherwise.

Essentially, f is understood as a composition of elementary functions such as `sin, cos, exp`. The program stores the analytic derivative of these elementary functions and the differentiation of f is obtained by applying the chain rule successively. The exact procedure may be understood conceptually in terms of a computational graph. Each node of the graph stores two real scalars: value and derivative. The forward AD initiates at n nodes, each representing a variable in f: the i-th node's value and derivative are set to be `c[i]` and `p[i]`. New nodes are then "composed" in exact order the sub-terms of f are composed, starting from n variables, from elementary functions.

To illustrate, let's denote a new node as s_{k+1} and old nodes as $s^{(i)}{}_k$. The value of a new node is simply the elementary function applied to value of old node(s) :

$s_{k+1}[value] \leftarrow \phi_i(s_k[value])$ or

$s_{k+1}[value] \leftarrow \phi_i(s_k{}^{(1)}[value], s_k{}^{(2)}[value])$

Whereas the derivative is calculated by applying the chain rule to the accumulated value and derivative fields

$s_{k+1}[derivative] \leftarrow \phi_i'(s_k[value]) s_k[derivative]$ or

$s_{k+1}[derivative] \leftarrow \partial/\partial x\ \phi_i(s_k{}^{(1)}[value], s_k{}^{(2)}[value]) \cdot s_k{}^{(1)}[derivative]]$

$+ \partial/\partial y\ \phi_i(s_k{}^{(1)}[value], s_k{}^{(2)}[value]) \cdot s_k{}^{(2)}[derivative]]$

New nodes are generated automatically by virtue of each elementary function and overridden to evaluate the value and derivative as above; and the program should "trace out" the entire computational graph by simply evaluating f in the natural arithmetic order.

How to Use AutoJac

The user will interact with our package as with other more standard python libraries such as sklearn, autograd etc.. In particular, they will import the following two modules:

```
>>> from AutoJac import ADiff as AD # ADiff is the user interface module
>>> import AutoJac.ElemFunc as EM # EM contains exp(), sin(), cos() etc.
```

And then, users will define the function of interest as follows:

```
>>> def myfunc(x,y):
...     f=[x*y+EF.sin(x),x+y+EF.sin(x*y)]
...     return f
```

where `x,y` can be any other variable names, number of variables and dimension of vector function is arbitrary.

Note that the program is robust enough for function handles of various forms (e.g. input below is completely equivalent to input above):

```
>>>def myfunc(x,y):
...     f1=x
...     f2=EF.sin(x)
...     f3=10
...     f4=x+y+EF.sin(x*y)+10
...     return [y*f1+f2, -(f3-f4)]
```

Users will instantiate a `ADiff` object (`ADiff` is the interface class under module `AD`):

```
>>> f_obj = AD.ADiff(myfunc) # Instantiate the interface
```

Then users will have the option to evaluate the entire Jacobian J at any point c:

```
>>> c = [1,2] # Jacobian evaluated at c=[1,2]
>>> res = f_obj.Jac(c)
```

The output `res` is a dictionary with keys: `val, diff. val` is the value of the function evaluated at `c, diff` is Jacobian matrix:

```
>>> res
{
'value': [2.841, 3.909]
'diff': [[2.540, 1.000],
         [0.168, 0.584]],
}
```

Or they can just evaluate `Jp` for any vector `p`. This method is more efficient than computing `J` and compute `Jp` again via matrix multiplication:

```
>>> p = [2,3]
>>> res = f_obj.pJac(c, p) % Outputs the vector Jp and value of function at c
```

The output is a dictionary where `val`is the value of function at `c` and `diff` is `Jp`:

```
>>> res
{
'value': [2.841, 3.909]
'diff':  [8.081, 2.087]
}
```

Time permitting, we will implement a corresponding GUI. In that case, there is no need for the user to explicitly instantiate an AD object.

Software Organization

Directory Structure

```
AutoJac/
       AD/
         __init__.py
         AutoDiff.py
         ADiff.py
         ElemFunc.py
       .travis.yml
       LICENSE.txt
       requirements.txt
       README.md
       setup.py
       tests/
            AutoJactests.py
```

At the root of the project, we will include general files such as `README.md`, `.travis.yml`, `requirements.txt`, and packaging and distribution files.

The AD sub-directory will include the files that implement automatic differentiation:

- `ADiff` module: implement `ADiff` class, which is user-interface
- `AutoDiff` module: implement `AutoDiff` class, which overloads built-in python functions
- `ElemFunc` module: implement elementary functions such as `sin, cos, exp`

A sub-directory tests contains the files that test the rest of the code.

Modules and Basic Functionalities

AutoDiff class overloads python built-in functions. Other elementary functions `sin`, `sqrt`, `log`, and `exp` are defined in ElemFunc class. We also require an ADiff class, which serves as a user interface: it receives and stores user inputs, and assembles results from AutoDiff, ElemFunc to form output of various sorts (e.g. entire Jacobian J, or Jp or function values).

Testing

Our test suite will live under a subdirectory `tests`. We will indeed use continuous integration tools such as TravisCI and CodeCov. A quick summary, similar to HW4, will be included in the README.md file. We will use doctests for our tests. In addition, we will include acceptance tests for every user interface functionality and unit tests for every operator overloaded and every elementary function implemented. Our tests will check individual functionalities as well as functions that require various functionalities at different layers. Lastly, our tests will check for the handling of unexpected user behavior.

Packaging and Distribution

We choose to package and distribute our software so that the onboarding time for a standard python user is minimal. We will package our software using the pip package manager since it is standard, all python users have it, and simple to distribute.

We will distribute our package through PyPI. We will simply follow the standard workflow using the setuptool. This procedure is given in [1], which includes creating setup.py, LICENSE, README.md, generating and uploading distribution archives.

The distribution method above will require users to have environments which include the corresponding dependencies used in our software. To alleviate this constraint, we will also distribute our software through Docker. We will push our software as an image to dockerhub. From there, users can simply pull our image, run a container of the image, and easily use the software. This method of distribution targets python users who are familiar with docker and dockerhub.

Lastly, we will write a detailed README including our dependencies and the intended user interface. The user interface explanation will benefit any user who uses our software. By providing our dependencies (including versions), python users who are not familiar with pip, can clone the repository to use our software. This is also beneficial to us in the case of users who would like to contribute to the software.

Implementation

Use Cases

We support vector or scalar functions as input. Our interface provides choices of outputs as either an entire Jacobian matrix or the product of a Jacobian matrix and a vector where the latter is computationally efficient. Note that the implementation is based on the idea of a seed vector p at the end of lecture 10.

Core Data Structure

We use only Python Lists, Dictionaries and Class structures for the development of the forward mode:

- We use List object to store matrices and vectors instead of third party data structures such as numpy array. Since no matrix computation is involved in the forward mode, we opt to avoid unnecessary dependencies on external libraries.
- We define the output of our computation as a Dictionary where we store the value of the function and derivative/gradient/Jacobian of the function.
- See next section for our use of class structure and their purposes.

Classes to implement, Core Flow

AutoDiff class overloads python built-in functions. Other elementary functions `sin`, `sqrt`, `log`, and `exp` are defined in ElemFunc class. We also require a ADiff class, which serves as a user interface: it receives and stores user inputs, and assembles results from AutoDiff, ElemFunc to form output of various sorts (e.g. entire Jacobian J, or Jp or function values).

The core flow of the program (implemented within interface `ADiff.pJac`) to evaluate `Jp` is as follows (entire `J` is easily implemented by calling this procedure with `p` set as `[1,0,0,..], [0,1,0,0,...]...`):

Step 1. Receive function of interest from user:

```
>>> def myfunc(x,y):
...     f=[x*y+EF.sin(x),x+y+EF.sin(x*y)]
...     return f
```

Step 2. Receive `c, p` from user

Step 3. Count number of variables of the function using

```
>>> varNum = myfunc.__code__.co_argcount
```

Step 4. Construct `varNum` of `AutoDiff` instances and put them in a list, each initialized with `p[i], c[i]`

```
>>> varList=[AutoDiff(c[i], p[i]) for i in range (varNum)]
```

Step 5. Pass this list as arguments to `myfunc`.

```
>>> myfunc(*varList)
```

Notice that we simply construct AutoDiff class instances based on number and order of inputs of myfunc. It is more natural to let user pass their function as a function handle instead of string. Also, this design saves the trouble of parsing variable names and associated exception handling.

Attributes for Classes

AutoDiff Class

The `AutoDiff` class is analogous to `AutoDiffToy` class in HW4. The idea is that each (i-th) variable in the input function is to be initialized as an instance of the AutoDiff class with seed `p[i]` and variable values `c[i]`.

The constructor of `AutoDiff` receives two inputs, which are primary instance variables of this class:

- `val`: a **scalar** real number initialized as `c[i]`, representing the "value column" in the evaluation trace;
- `der`: a **scalar** real number initialized as `p[i]`, representing the "directional derivative column" in the evaluation trace.

Suppose we are interested in the directional derivative `Jp` of a function for any vector `p` evaluated at `c`. Function variables `x, y` are initialized as instances of this class as follows.

```
>>> p=[2,3]
>>> c=[1,2]

>>> x=AutoDiff(c[0],p[0])
>>> y=AutoDiff(c[1],p[1])
```

This is exactly what happens in Step 3 in `ADiff.pJac` outlined above. Indeed, the i-th entry in `f_obj` below is an `AutoDiff` instance with `val` being the i-th entry of the function value at `c` and `der` being the i-th entry of `Jp`.

```
>>> f_obj=[x*y+EF.sin(x),x+y+EF.sin(xy)]
```

Class methods of `AutoDiff` class are Dunder methods overloading python built-in elementary operations. In particular, each of these methods **returns a new `AutoDiff` instance** with the appropriate `val, der` computed according to the following table, which are just elementary derivatives with the chain rule. **Note that the format of the pseudo-code below is: `AutoDiff(val, der)` where `val, der` are the appropriate `val, der` fields based on input `self, other` for each overloaded method to return, analogous to HW4.** We differentiate implementation of input of AutoDiff type or real scalar type. This is done by the Duck Typing method covered in lecture and HW4.

	Return when other is AutoDiff type	Return when other is real scalar
`__add__(self, other)`	`AutoDiff(self.val+other.val, self.der+other.der)`	`AutoDiff(self.val+other, self.der)`
`__radd__(self, other)`	N/A	”
`__sub__(self, other)`	`AutoDiff(self.val-other.val, self.der-other.der)`	`AutoDiff(self.val-other, self.other)`
`__rsub__(self,other)`	N/A	`AutoDiff(other-self.val,-self.der)`
`__mul__(self, other)`	`AutoDiff(self.val*other.val, self.val*other.der+self.der*other.val)`	`AutoDiff(self.val*other, self.der*other)`
`__rmul__(self,other)`	N/A	”
`__truediv__(self, other)`	`AutoDiff(self.val/other.val, (self.der*other.val-other.der*self.val)/(other.val**2))`	`AutoDiff(self.val/other, self.der/other)`
`__rtruediv__(self, other)`	N/A	`AutoDiff(other/self.val,-other*self.der/(self.val**2))`

*Note that this table is not complete: we have omitted more complicated expressions such as __pow__(), __rpow__(). The idea is the same.

ElemFunc Class

The other elementary functions such as `exp, sin, cos` are implemented in ElemFunc Classes -- all these functions return a new `AutoDiff` object same as the Dunder methods, with corresponding `val` and `der`. They can handle either real or AutoDiff type input with duck typing, similar to Duder methods.

	`x is AutoDiff object`	`x is real number`
`sin(x)`	`AutoDiff(math.sin(x.val), math.cos(x.val)*x.der)`	`AutoDiff(math.sin(x),0)`
`cos(x)`	`AutoDiff(math.cos(x.val), -math.sin(x.val)*x.der)`	`AutoDiff(math.cos(x),0)`
`exp(x)`	`AutoDiff(math.exp(x.val), math.exp(x.val)*x.der)`	`AutoDiff(math.exp(x),0)`

* Again this table is not complete.

ADiff Class

This class mainly serves as a user-interface. The constructor receives one input, which are instance variable of this class:

- `func:` a user defined function

```
>>> def myfunc(x,y):
...     f=[x*y+EF.sin(x),x+y+EF.sin(x*y)]
...     return f
>>> f.obj = ADiff(myfunc)
```

There are two methods in this class:

- `Jac(c):` Return a dictionary with value of the function at `c` and entire Jacobian `J` at `c`
- `pJac(c,p):` Return a dictionary with value of function at `c` and Jacobian acting on `p`, i.e. `Jp`

The implementation of `pJac` is given in previous section (Classes to implement, core flow). Jac fills i-th column of Jacobian matrix by calling upon `pJac` with `p` set to be a vector with 1 at i-th entry and 0 otherwise.

External Dependencies

We will rely on math and pytest (for testing). For advanced features, we will likely use numpy and are likely to evaluate our results to those implemented in standard library such as scikit-learn.

Implementing elementary functions like `sin, sqrt, log,` and `exp`

The Python built-in arithmetic operators such as “+”, “-”, “**” are implemented by overriding corresponding Dunder methods. Others such as `sin, sqrt, log,` and `exp` are implemented in Elemfunc Class. See previous sections for detail.