Numdifftools Documentation

Release 0.9.39

Per A Brodtkorb and John D'Errico

Jun 10, 2019

Contents:

This is the documentation of **Numdifftools** version 0.9.39 released Jun 10, 2019.

Bleeding edge available at: https://github.com/pbrod/numdifftools.

Official releases are available at: http://pypi.python.org/pypi/Numdifftools.

CHAPTER 1

Introduction

1.1 What is numdifftools?

Numdifftools is a suite of tools written in _Python to solve automatic numerical differentiation problems in one or more variables. Finite differences are used in an adaptive manner, coupled with a Richardson extrapolation methodology to provide a maximally accurate result. The user can configure many options like; changing the order of the method or the extrapolation, even allowing the user to specify whether complex-step, central, forward or backward differences are used.

The methods provided are:

- **Derivative**: Compute the derivatives of order 1 through 10 on any scalar function.
- **directionaldiff**: Compute directional derivative of a function of n variables
- **Gradient**: Compute the gradient vector of a scalar function of one or more variables.
- **Jacobian**: Compute the Jacobian matrix of a vector valued function of one or more variables.
- **Hessian**: Compute the Hessian matrix of all 2nd partial derivatives of a scalar function of one or more variables.
- **Hessdiag**: Compute only the diagonal elements of the Hessian matrix

All of these methods also produce error estimates on the result.

Numdifftools also provide an easy to use interface to derivatives calculated with in _AlgoPy. Algopy stands for Algorithmic Differentiation in Python. The purpose of AlgoPy is the evaluation of higher-order derivatives in the *forward* and *reverse* mode of Algorithmic Differentiation (AD) of functions that are implemented as Python programs.

1.2 How the documentation is organized

Numdifftools has a lot of documentation. A high-level overview of how it's organized will help you know where to look for certain things:

- *Tutorials* take you by the hand through a series of steps to load a CDF container and explore its contents or to construct a new dataset and validate it. Start here if you're new to numdifftools.

- *Topic guides* discuss key topics and concepts at a fairly high level and provide useful background information and explanation.
- *Reference guides* contain technical reference for APIs and other aspects of numdifftools' machinery. They describe how it works and how to use it but assume that you have a basic understanding of key concepts.
- *How-to guides* are recipes. They guide you through the steps involved in addressing key problems and use-cases. They are more advanced than tutorials and assume some knowledge of how numdifftools works.

CHAPTER 2

Tutorials

The pages in this section of the documentation are aimed at the newcomer to numdifftools. They're designed to help you get started quickly, and show how easy it is to work with numdifftools as a developer who wants to customise it and get it working according to their own requirements.

These tutorials take you step-by-step through some key aspects of this work. They're not intended to explain the *topics in depth*, or provide *reference material*, but they will leave you with a good idea of what it's possible to achieve in just a few steps, and how to go about it.

Once you're familiar with the basics presented in these tutorials, you'll find the more in-depth coverage of the same topics in the How-to section.

The tutorials follow a logical progression, starting from installation of numdifftools and the creation of a brand new project, and build on each other, so it's recommended to work through them in the order presented here.

2.1 Install guide

Before you can use numdifftools, you'll need to get it installed. This guide will guide you through a simple installation that'll work while you walk through the introduction.

2.1.1 Install Python

Being a Python library, numdifftools requires Python. Preferably you ned version 3.4 or newer, but you get the latest version of Python at https://www.python.org/downloads/.

You can verify that Python is installed by typing `python` from the command shell; you should see something like:

```
Python 3.6.3 (64-bit)| (default, Oct 15 2017, 03:27:45)
[MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

`pip` is the Python installer. Make sure yours is up-to-date, as earlier versions can be less reliable:

```
$ pip install --upgrade pip
```

2.1.2 Dependencies

Numdifftools requires numpy 1.9 or newer, scipy 0.8 or newer, and Python 2.7 or 3.3 or newer. This tutorial assumes you are using Python 3. Optionally you may also want to install Algopy 0.4 or newer and statsmodels 0.6 or newer in order to be able to use their easy to use interfaces to their derivative functions.

2.1.3 Install numdifftools

To install numdifftools simply type in the 'command' shell:

```
$ pip install numdifftools
```

to get the lastest stable version. Using pip also has the advantage that all requirements are automatically installed.

2.1.4 Verifying installation

To verify that numdifftools can be seen by Python, type `python` from your shell. Then at the Python prompt, try to import numdifftools:

```
>>> import numdifftools as nd
>>> print(nd.__version__)
0.9.39
```

To test if the toolbox is working correctly paste the following in an interactive python prompt:

```
nd.test('--doctest-module')
```

If the result show no errors, you now have installed a fully functional toolbox. Congratulations!

2.1.5 That's it!

That's it – you can now *move onto the getting started tutorial*

2.2 Getting started

2.2.1 The derivative

How does numdifftools.Derivative work in action? A simple nonlinear function with a well known derivative is e^x. At $x = 0$, the derivative should be 1.

```
>>> import numdifftools as nd
>>> f = nd.Derivative(exp, full_output=True)
>>> val, info = f(0)
>>> allclose(val, 1)
True
```

```
>>> allclose(info.error_estimate, 5.28466160e-14)
True
```

A second simple example comes from trig functions. The first four derivatives of the sine function, evaluated at $x = 0$, should be respectively $[cos(0), -sin(0), -cos(0), sin(0)]$, or $[1, 0, -1, 0]$.

```
>>> from numpy import sin, allclose
>>> import numdifftools as nd
>>> df = nd.Derivative(sin, n=1)
```

```
>>> allclose(df(0), 1.)
True
```

```
>>> ddf = nd.Derivative(sin, n=2)
>>> allclose(ddf(0), 0.)
True
```

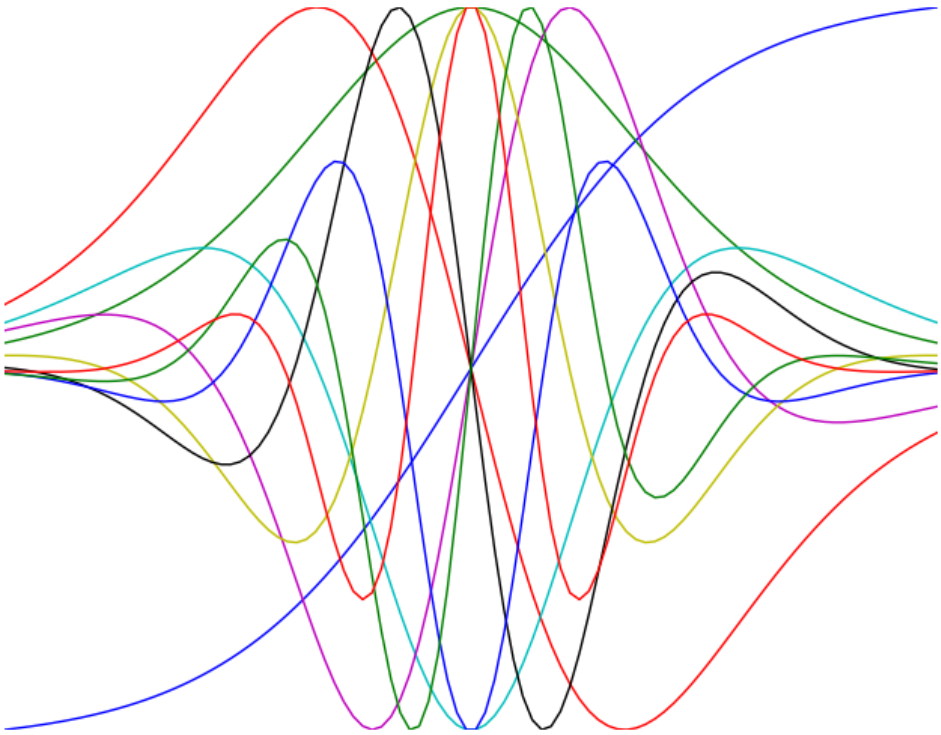
```
>>> dddf = nd.Derivative(sin, n=3)
>>> allclose(dddf(0), -1.)
True
```

```
>>> ddddf = nd.Derivative(sin, n=4)
>>> allclose(ddddf(0), 0.)
True
```

Visualize high order derivatives of the tanh function

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-2, 2, 100)
>>> for i in range(10):
...     df = nd.Derivative(np.tanh, n=i)
...     y = df(x)
...     h = plt.plot(x, y/np.abs(y).max())
```

plt.show()



2.2.2 Gradient and Hessian estimation

Estimation of the gradient vector (numdifftools.Gradient) of a function of multiple variables is a simple task, requiring merely repeated calls to numdifftools.Derivative. Likewise, the diagonal elements of the hessian matrix are merely pure second partial derivatives of a function. numdifftools.Hessdiag accomplishes this task, again calling numdifftools.Derivative multiple times. Efficient computation of the off-diagonal (mixed partial derivative) elements of the Hessian matrix uses a scheme much like that of numdifftools.Derivative, then Richardson extrapolation is used to improve a set of second order finite difference estimates of those mixed partials.

Multivariate calculus examples

Typical usage of the gradient and Hessian might be in optimization problems, where one might compare an analytically derived gradient for correctness, or use the Hessian matrix to compute confidence interval estimates on parameters in a maximum likelihood estimation.

Gradients and Hessians

```
>>> import numpy as np
>>> def rosen(x): return (1-x[0])**2 + 105.*(x[1]-x[0]**2)**2
```

Gradient of the Rosenbrock function at [1,1], the global minimizer

```
>>> grad = nd.Gradient(rosen)([1, 1])
```

The gradient should be zero (within floating point noise)

```
>>> allclose(grad, 0)
True
```

The Hessian matrix at the minimizer should be positive definite

```
>>> H = nd.Hessian(rosen)([1, 1])
```

The eigenvalues of H should be positive

```
>>> li, U = np.linalg.eig(H)
>>> li>0
array([ True,  True], dtype=bool)
```

Gradient estimation of a function of 5 variables

```
>>> f = lambda x: np.sum(x**2)
>>> grad = nd.Gradient(f)(np.r_[1, 2, 3, 4, 5])
>>> allclose(grad, [  2.,   4.,   6.,   8.,  10.])
True
```

Simple Hessian matrix of a problem with 3 independent variables

```
>>> f = lambda x: x[0] + x[1]**2 + x[2]**3
>>> H = nd.Hessian(f)([1, 2, 3])
>>> allclose(H, np.diag([0, 2, 18]))
True
```

A semi-definite Hessian matrix

```
>>> H = nd.Hessian(lambda xy: np.cos(xy[0] - xy[1]))([0, 0])
```

one of these eigenvalues will be zero (approximately)

```
>>> np.abs(np.linalg.eig(H)[0]) < 1e-12
array([ True, False], dtype=bool)
```

Directional derivatives

The directional derivative will be the dot product of the gradient with the (unit normalized) vector. This is of course possible to do with numdifftools and you could do it like this for the Rosenbrock function at the solution, x0 = [1,1]:

```
>>> v = np.r_[1, 2]/np.sqrt(5)
>>> x0 = [1, 1]
>>> directional_diff = np.dot(nd.Gradient(rosen)(x0), v)
```

This should be zero.

```
>>> allclose(directional_diff, 0)
True
```

Ok, its a trivial test case, but it easy to compute the directional derivative at other locations:

```
>>> v2 = np.r_[1, -1]/np.sqrt(2)
>>> x2 = [2, 3]
>>> directionaldiff = np.dot(nd.Gradient(rosen)(x2), v2)
>>> allclose(directionaldiff, 743.87633380824832)
True
```

There is a convenience function $nd.directionaldiff$ that also takes care of the direction normalization:

```
>>> v = [1, -1]
>>> x0 = [2, 3]
>>> directional_diff = nd.directionaldiff(rosen, x0, v)
>>> np.allclose(directional_diff, 743.87633380824832)
True
```

Jacobian matrix

Jacobian matrix of a scalar function is just the gradient

```
>>> jac = nd.Jacobian(rosen)([2, 3])
>>> grad = nd.Gradient(rosen)([2, 3])
>>> allclose(jac, grad)
True
```

Jacobian matrix of a linear system will reduce to the design matrix

```
>>> A = np.random.rand(5,3)
>>> b = np.random.rand(5)
>>> fun = lambda x: np.dot(x, A.T) - b
>>> x = np.random.rand(3)
>>> jac = nd.Jacobian(fun)(x)
```

This should be essentially zero at any location x

```
>>> allclose(jac - A, 0)
True
```

The jacobian matrix of a nonlinear transformation of variables evaluated at some arbitrary location [-2, -3]

```
>>> fun = lambda xy: np.r_[xy[0]**2, np.cos(xy[0] - xy[1])]
>>> jac = nd.Jacobian(fun)([-2, -3])
>>> np.allclose(jac, [[-4.,  0.],
...                   [-0.84147098,  0.84147098]])
True
```

2.3 Conclusion

numdifftools.Derivative is an a adaptive scheme that can compute the derivative of arbitrary (well behaved) functions. It is reasonably fast as an adaptive method. Many options have been provided for the user who wishes the ultimate amount of control over the estimation.

2.4 What to read next

So you've read all the *introductory material* and have decided you'd like to keep using numdifftools. We've only just scratched the surface with this intro.

So what's next?

Well, we've always been big fans of learning by doing. At this point you should know enough to start a project of your own and start fooling around. As you need to learn new tricks, come back to the documentation.

We've put a lot of effort into making numdifftools's documentation useful, easy to read and as complete as possible. The rest of this document explains more about how the documentation works so that you can get the most out of it.

2.4.1 Finding documentation

Numdifftools got a *lot* of documentation, so finding what you need can sometimes be tricky. A few good places to start are the search and the genindex.

Or you can just browse around!

2.4.2 How the documentation is organized

Numdifftools main documentation is broken up into "chunks" designed to fill different needs:

- The *introductory material* is designed for people new to numdifftools. It doesn't cover anything in depth, but instead gives a hands on overview of how to use numdifftools.
- The *topic guides*, on the other hand, dive deep into individual parts of numdifftools from a theoretical perspective.
- We've written a set of *how-to guides* that answer common "How do I . . . ?" questions.
- The guides and how-to's don't cover every single class, function, and method available in numdifftools – that would be overwhelming when you're trying to learn. Instead, details about individual classes, functions, methods, and modules are kept in the *reference*. This is where you'll turn to find the details of a particular function or whatever you need.

2.4.3 How documentation is updated

Just as the numdifftools code base is developed and improved on a daily basis, our documentation is consistently improving. We improve documentation for several reasons:

- To make content fixes, such as grammar/typo corrections.

- To add information and/or examples to existing sections that need to be expanded.
- To document numdifftools features that aren't yet documented. (The list of such features is shrinking but exists nonetheless.)
- To add documentation for new features as new features get added, or as numdifftools APIs or behaviors change.

In plain text

For offline reading, or just for convenience, you can read the numdifftools documentation in plain text.

If you're using an official release of numdifftools, the zipped package (tarball) of the code includes a `docs/` directory, which contains all the documentation for that release.

If you're using the development version of numdifftools (aka the master branch), the `docs/` directory contains all of the documentation. You can update your Git checkout to get the latest changes.

One low-tech way of taking advantage of the text documentation is by using the Unix `grep` utility to search for a phrase in all of the documentation. For example, this will show you each mention of the phrase "max_length" in any numdifftools document:

```
$ grep -r max_length /path/to/numdifftools/docs/
```

As HTML, locally

You can get a local copy of the HTML documentation following a few easy steps:

- numdifftools's documentation uses a system called Sphinx to convert from plain text to HTML. You'll need to install Sphinx by either downloading and installing the package from the Sphinx website, or with `pip`:

  ```
  $ pip install Sphinx
  ```

- Then, just use the included `Makefile` to turn the documentation into HTML:

  ```
  $ cd path/to/numdifftools/docs
  $ make html
  ```

  You'll need GNU Make installed for this.

  If you're on Windows you can alternatively use the included batch file:

  ```
  $ cd path\to\numdifftools\docs
  $ make.bat html
  ```

- The HTML documentation will be placed in `docs/_build/html`.

Using pydoc

The pydoc module automatically generates documentation from Python modules. The documentation can be presented as pages of text on the console, served to a Web browser, or saved to HTML files.

For modules, classes, functions and methods, the displayed documentation is derived from the docstring (i.e. the __doc__ attribute) of the object, and recursively of its documentable members. If there is no docstring, pydoc tries to obtain a description from the block of comment lines just above the definition of the class, function or method in the source file, or at the top of the module (see inspect.getcomments()).

The built-in function help() invokes the online help system in the interactive interpreter, which uses pydoc to generate its documentation as text on the console. The same text documentation can also be viewed from outside the Python interpreter by running pydoc as a script at the operating system's command prompt. For example, running

```
$ pydoc numdifftools
```

at a shell prompt will display documentation on the numdifftools module, in a style similar to the manual pages shown by the Unix man command. The argument to pydoc can be the name of a function, module, or package, or a dotted reference to a class, method, or function within a module or module in a package. If the argument to pydoc looks like a path (that is, it contains the path separator for your operating system, such as a slash in Unix), and refers to an existing Python source file, then documentation is produced for that file.

You can also use pydoc to start an HTTP server on the local machine that will serve documentation to visiting Web browsers. For example, running

```
$ pydoc -b
```

will start the server and additionally open a web browser to a module index page. Each served page has a navigation bar at the top where you can Get help on an individual item, Search all modules with a keyword in their synopsis line, and go to the Module index, Topics and Keywords pages. To quit the server just type

```
$ quit
```

See also:

LibCdf is 100% Python, so if you're new to Python, you might want to start by getting an idea of what the language is like. Below we have given some pointers to some resources you can use to get acquainted with the language.

If you're new to programming entirely, you might want to start with this list of Python resources for non-programmers

If you already know a few other languages and want to get up to speed with Python quickly, we recommend Dive Into Python. If that's not quite your style, there are many other books about Python.

CHAPTER 3

How-to guides

These guides presuppose some familiarity with Numdifftools. They cover some of the same territory as the /introduction/index, but in more detail.

3.1 Set-up

3.2 Using core functionality

3.3 Creating new functionality

3.4 Contributing

CHAPTER 4

Topics guides

This section explains and analyses some key concepts in numdifftools. It's less concerned with explaining *how to do things* than with helping you understand *how it works*.

4.1 Introduction derivative estimation

The general problem of differentiation of a function typically pops up in three ways in Python.

- The symbolic derivative of a function.
- Compute numerical derivatives of a function defined only by a sequence of data points.
- Compute numerical derivatives of a analytically supplied function.

Clearly the first member of this list is the domain of the symbolic toolbox SymPy, or some set of symbolic tools. Numerical differentiation of a function defined by data points can be achieved with the function gradient, or perhaps by differentiation of a curve fit to the data, perhaps to an interpolating spline or a least squares spline fit.

The third class of differentiation problems is where Numdifftools is valuable. This document will describe the methods used in Numdifftools and in particular the Derivative class.

4.2 Numerical differentiation of a general function of one variable

Surely you recall the traditional definition of a derivative, in terms of a limit.

$$f'(x) = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta} \tag{4.1}$$

For small δ, the limit approaches $f'(x)$. This is a one-sided approximation for the derivative. For a fixed value of δ, this is also known as a finite difference approximation (a forward difference.) Other approximations for the derivative are also available. We will see the origin of these approximations in the Taylor series expansion of a

function $f(x)$ around some point x_0.

$$f(x_0+\delta) = f(x_0) + \delta f'(x_0) + \frac{\delta^2}{2}f''(x_0) + \frac{\delta^3}{6}f^{(3)}(x_0) + \\ \frac{\delta^4}{24}f^{(4)}(x_0) + \frac{\delta^5}{120}f^{(5)}(x_0) + \frac{\delta^6}{720}f^{(6)}(x_0) + ... \tag{4.2}$$

Truncate the series in (**??**) to the first three terms, divide by δ and rearrange yields the forward difference approximation (**??**):

$$f'(x_0) = \frac{f(x_0+\delta) - f(x_0)}{\delta} - \frac{\delta}{2}f''(x_0) - \frac{\delta^2}{6}f'''(x_0) + ... \tag{4.3}$$

When δ is small, δ^2 and any higher powers are vanishingly small. So we tend to ignore those higher powers, and describe the approximation in (**??**) as a first order approximation since the error in this approximation approaches zero at the same rate as the first power of δ.[1] The values of $f''(x_0)$ and $f'''(x_0)$, while unknown to us, are fixed constants as δ varies.

Higher order approximations arise in the same fashion. The central difference (**??**) is a second order approximation.

$$f'(x_0) = \frac{f(x_0+\delta) - f(x_0-\delta)}{2\delta} - \frac{\delta^2}{3}f'''(x_0) + ... \tag{4.4}$$

4.3 Unequally spaced finite difference rules

While most finite difference rules used to differentiate a function will use equally spaced points, this fails to be appropriate when one does not know the final spacing. Adaptive quadrature rules can succeed by subdividing each sub-interval as necessary. But an adaptive differentiation scheme must work differently, since differentiation is a point estimate. Derivative generates a sequence of sample points that follow a log spacing away from the point in question, then it uses a single rule (generated on the fly) to estimate the desired derivative. Because the points are log spaced, the same rule applies at any scale, with only a scale factor applied.

4.4 Odd and even transformations of a function

Returning to the Taylor series expansion of $f(x)$ around some point x_0, an even function[2] around x_0 must have all the odd order derivatives vanish at x_0. An odd function has all its even derivatives vanish from its expansion. Consider the derived functions $f_{odd}(x)$ and $f_{even}(x)$.

$$f_{odd}(x) = \frac{f(x_0+x) - f(x_0-x)}{2} \tag{4.5}$$

$$f_{even}(x) = \frac{f(x_0+x) - 2f(x_0) + f(x_0-x)}{2} \tag{4.6}$$

The Taylor series expansion of $f_{odd}(x)$ around zero has the useful property that we have killed off any even order terms, but the odd order terms are identical to $f(x)$, as expanded around x_0.

$$f_{odd}(\delta) = \delta f'(x_0) + \frac{\delta^3}{6}f^{(3)}(x_0) + \frac{\delta^5}{120}f^{(5)}(x_0) + \frac{\delta^7}{5040}f^{(7)}(x_0) + ... \tag{4.7}$$

Likewise, the Taylor series expansion of $f_{even}(x)$ has no odd order terms or a constant term, but other even order terms that are identical to $f(x)$.

[1] We would normally write these additional terms using O() notation, where all that matters is that the error term is $O(\delta)$ or perhaps $O(\delta^2)$, but explicit understanding of these error terms will be useful in the Richardson extrapolation step later on.

[2] An even function is one which expresses an even symmetry around a given point. An even symmetry has the property that $f(x) = f(-x)$. Likewise, an odd function expresses an odd symmetry, wherein $f(x) = -f(-x)$.

$$f_{even}(\delta) = \frac{\delta^2}{2} f^{(2)}(x_0) + \frac{\delta^4}{24} f^{(4)}(x_0) + \frac{\delta^6}{720} f^{(6)}(x_0) + \frac{\delta^8}{40320} f^{(8)}(x_0) + ... \tag{4.8}$$

The point of these transformations is we can rather simply generate a higher order approximation for any odd order derivatives of $f(x)$ by working with $f_{odd}(x)$. Even order derivatives of $f(x)$ are similarly generated from $f_{even}(x)$. For example, a second order approximation for $f'(x_0)$ is trivially written in (5.2) as a function of δ.

$$f'(x_0; \delta) = \frac{f_{odd}(\delta)}{\delta} - \frac{\delta^2}{6} f^{(3)}(x_0) \tag{4.9}$$

We can do better rather simply, so why not? (5.3) shows a fourth order approximation for $f'(x_0)$.

$$f'(x_0; \delta) = \frac{8 f_{odd}(\delta) - f_{odd}(2\delta)}{6\delta} + \frac{\delta^4}{30} f^{(5)}(x_0) \tag{4.10}$$

Again, the next non-zero term (**??**) in that expansion has a higher power of δ on it, so we would normally ignore it since the lowest order neglected term should dominate the behavior for small δ.

$$\frac{\delta^6}{252} f^{(7)}(x_0) \tag{4.11}$$

Derivative uses similar approximations for all derivatives of f up to any order. Of course, it is not always possible for evaluation of a function on both sides of a point, as central difference rules will require. In these cases, you can specify forward or backward difference rules as appropriate. You can also specify to use the complex step derivative, which we will outline in the next section.

4.5 Complex step derivative

The derivation of the complex-step derivative approximation is accomplished by replacing δ in (**??**) with a complex step ih:

$$f(x_0 + ih) = f(x_0) + ihf'(x_0) - \frac{h^2}{2} f''(x_0) - \frac{ih^3}{6} f^{(3)}(x_0) + \frac{h^4}{24} f^{(4)}(x_0) + \\ \frac{ih^5}{120} f^{(5)}(x_0) - \frac{h^6}{720} f^{(6)}(x_0) - ... \tag{4.12}$$

Taking only the imaginary parts of both sides gives

$$\Im(f(x_0 + ih)) = hf'(x_0) - \frac{h^3}{6} f^{(3)}(x_0) + \frac{h^5}{120} f^{(5)}(x_0) - ... \tag{4.13}$$

Dividing with h and rearranging yields:

$$f'(x_0) = \Im(f(x_0 + ih))/h + \frac{h^2}{6} f^{(3)}(x_0) - \frac{h^4}{120} f^{(5)}(x_0) + ... \tag{4.14}$$

Terms with order h^2 or higher can safely be ignored since the interval h can be chosen up to machine precision without fear of rounding errors stemming from subtraction (since there are not any). Thus to within second-order the complex-step derivative approximation is given by:

$$f'(x_0) = \Im(f(x_0 + ih))/h \tag{4.15}$$

Next, consider replacing the step δ in (**??**) with the complex step $i^{\frac{1}{2}}h$:

$$f_{even}(i^{\frac{1}{2}}h) = \frac{ih^2}{2} f^{(2)}(x_0) - \frac{h^4}{24} f^{(4)}(x_0) - \frac{ih^6}{720} f^{(6)}(x_0) + \\ \frac{h^8}{40320} f^{(8)}(x_0) + \frac{ih^{10}}{3628800} f^{(10)}(x_0) - ... \tag{4.16}$$

Similarly dividing with $h^2/2$ and taking only the imaginary components yields:

$$f^{(2)}(x_0) = \Im\left(2\, f_{even}(i^{\frac{1}{2}}h)\right)/h^2 + \frac{h^4}{360}f^{(6)}(x_0) - \frac{h^8}{1814400}f^{(10)}(x_0)... \tag{4.17}$$

This approximation is still subject to difference errors, but the error associated with this approximation is proportional to h^4. Neglecting these higher order terms yields:

$$f^{(2)}(x_0) = 2\Im\left(f_{even}(i^{\frac{1}{2}}h)\right)/h^2 = \Im(f(x_0 + i^{\frac{1}{2}}h) + f(x_0 - i^{\frac{1}{2}}h))/h^2 \tag{4.18}$$

See [LaiCrassidisCheng2005] and [Ridout2009] for more details. The complex-step derivative in numdifftools.Derivative has truncation error $O(\delta^4)$ for both odd and even order derivatives for $n > 1$. For $n = 1$ the truncation error is on the order of $O(\delta^2)$, so truncation error can be eliminated by choosing steps to be very small. The first order complex-step derivative avoids the problem of round-off error with small steps because there is no subtraction. However, the function to differentiate needs to be analytic. This method does not work if it does not support complex numbers or involves non-analytic functions such as e.g.: abs, max, min. For this reason the *central* method is the default method.

4.6 High order derivative

So how do we construct these higher order approximation formulas? Here we will deomonstrate the principle by computing the 6'th order central approximation for the first-order derivative. In order to do so we simply set $f_{odd}(\delta)$ equal to its 3-term Taylor expansion:

$$f_{odd}(\delta) = \sum_{i=0}^{2} \frac{\delta^{2i+1}}{(2i+1)!} f^{(2i+1)}(x_0) \tag{4.19}$$

By inserting three different stepsizes into (**??**), eg $\delta, \delta/2, \delta/4$, we get a set of linear equations:

$$\begin{bmatrix} 1 & \frac{1}{3!} & \frac{1}{5!} \\ \frac{1}{2} & \frac{1}{3!\,2^3} & \frac{1}{5!\,2^5} \\ \frac{1}{4} & \frac{1}{3!\,4^3} & \frac{1}{5!\,4^5} \end{bmatrix} \begin{bmatrix} \delta f'(x_0) \\ \delta^3 f^{(3)}(x_0) \\ \delta^5 f^{(5)}(x_0) \end{bmatrix} = \begin{bmatrix} f_{odd}(\delta) \\ f_{odd}(\delta/2) \\ f_{odd}(\delta/4) \end{bmatrix} \tag{4.20}$$

The solution of these equations are simply:

$$\begin{bmatrix} \delta f'(x_0) \\ \delta^3 f^{(3)}(x_0) \\ \delta^5 f^{(5)}(x_0) \end{bmatrix} = \frac{1}{3} \begin{bmatrix} \frac{1}{15} & \frac{-8}{3} & \frac{256}{15} \\ -8 & 272 & -512 \\ 512 & -5120 & 8192 \end{bmatrix} \begin{bmatrix} f_{odd}(\delta) \\ f_{odd}(\delta/2) \\ f_{odd}(\delta/4) \end{bmatrix} \tag{4.21}$$

The first row of (**??**) gives the coefficients for 6'th order approximation. Looking at at row two and three, we see also that this gives the 6'th order approximation for the 3'rd and 5'th order derivatives as bonus. Thus this is also a general method for obtaining high order differentiation rules. As previously noted these formulas have the additional benefit of beeing applicable to any scale, with only a scale factor applied.

4.7 Richardson extrapolation methodology applied to derivative estimation

Some individuals might suggest that the above set of approximations are entirely adequate for any sane person. Can we do better?

Suppose we were to generate several different estimates of the approximation in (**??**) for different values of δ at a fixed x_0. Thus, choose a single δ, estimate a corresponding resulting approximation to $f'(x_0)$, then do the same for $\delta/2$. If we assume that the error drops off linearly as $\delta \to 0$, then it is a simple matter to extrapolate this process to a zero step size. Our lack of knowledge of $f''(x_0)$ is irrelevant. All that matters is δ is small enough that the linear term dominates so we can ignore the quadratic term, therefore the error is purely linear.

$$f'(x_0) = \frac{f(x_0 + \delta) - f(x_0)}{\delta} - \frac{\delta}{2} f''(x_0) \tag{4.22}$$

The linear extrapolant for this interval halving scheme as $\delta \to 0$ is given by:

$$f_0^{'} = 2f_{\delta/2}^{'} - f_{\delta}^{'} \tag{4.23}$$

Since I've always been a big fan of convincing myself that something will work before I proceed too far, lets try this out in Python. Consider the function e^x. Generate a pair of approximations to $f'(0)$, once at δ of 0.1, and the second approximation at $1/2$ that value. Recall that $\frac{d(e^x)}{dx} = e^x$, so at x = 0, the derivative should be exactly 1. How well will we do?

```
>>> from numpy import exp, allclose
>>> f = exp
>>> dx = 0.1
>>> df1 = (f(dx) - f(0))/dx
>>> allclose(df1, 1.05170918075648)
True
```

```
>>> df2 = (f(dx/2) - f(0))/(dx/2)
>>> allclose(df2, 1.02542192752048)
True
```

```
>>> allclose(2*df2 - df1, 0.999134674284488)
True
```

In fact, this worked very nicely, reducing the error to roughly 1 percent of our initial estimates. Should we be surprised at this reduction? Not if we recall that last term in (**??**). We saw there that the next term in the expansion was $O(\delta^2)$. Since δ was 0.1 in our experiment, that 1 percent number makes perfect sense.

The Richardson extrapolant in (**??**) assumed a linear process, with a specific reduction in δ by a factor of 2. Assume the two term (linear + quadratic) residual term in (**??**), evaluating our approximation there with a third value of δ. Again, assume the step size is cut in half again. The three term Richardson extrapolant is given by:

$$f_0' = \frac{1}{3}f_{\delta}' - 2f_{\delta/2}' + \frac{8}{3}f_{\delta/4}' \tag{4.24}$$

A quick test in Python yields much better results yet.

```
>>> from numpy import exp, allclose
>>> f = exp
>>> dx = 0.1
```

```
>>> df1 = (f(dx) - f(0))/dx
>>> allclose(df1,  1.05170918075648)
True
```

```
>>> df2 = (f(dx/2) - f(0))/(dx/2)
>>> allclose(df2, 1.02542192752048)
True
```

```
>>> df3 = (f(dx/4) - f(0))/(dx/4)
>>> allclose(df3, 1.01260482097715)
True
```

```
>>> allclose(1./3*df1 - 2*df2 + 8./3*df3, 1.00000539448361)
True
```

Again, Derivative uses the appropriate multiple term Richardson extrapolants for all derivatives of f up to any order[3]. This, combined with the use of high order approximations for the derivatives, allows the use of quite large step sizes. See [LynessMoler1966] and [LynessMoler1969]. How to compute the multiple term Richardson extrapolants will be elaborated further in the next section.

[3] For practical purposes the maximum order of the derivative is between 4 and 10 depending on the function to differentiate and also the method used in the approximation.

4.8 Multiple term Richardson extrapolants

We shall now indicate how we can calculate the multiple term Richardson extrapolant for $f_{odd}(\delta)/\delta$ by rearranging (**??**):

$$\frac{f_{odd}(\delta)}{\delta} = f'(x_0) + \sum_{i=1}^{\infty} \frac{\delta^{2i}}{(2i+1)!} f^{(2i+1)}(x_0) \tag{4.25}$$

This equation has the form

$$\phi(\delta) = L + a_0\delta^2 + a_1\delta^4 + a_2\delta^6 + ... \tag{4.26}$$

where L stands for $f'(x_0)$ and $\phi(\delta)$ for the numerical differentiation formula $f_{odd}(\delta)/\delta$.

By neglecting higher order terms ($a_3\delta^8$) and inserting three different stepsizes into (**??**), eg $\delta, \delta/2, \delta/4$, we get a set of linear equations:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & \frac{1}{2^2} & \frac{1}{2^4} \\ 1 & \frac{1}{4^2} & \frac{1}{4^4} \end{bmatrix} \begin{bmatrix} L \\ \delta^2 a_0 \\ \delta^4 a_1 \end{bmatrix} = \begin{bmatrix} \phi(\delta) \\ \phi(\delta/2) \\ \phi(\delta/4) \end{bmatrix} \tag{4.27}$$

The solution of these equations are simply:

$$\begin{bmatrix} L \\ \delta^2 a_0 \\ \delta^4 a_1 \end{bmatrix} = \frac{1}{45} \begin{bmatrix} 1 & -20 & 64 \\ -20 & 340 & -320 \\ 64 & -320 & 256 \end{bmatrix} \begin{bmatrix} \phi(\delta) \\ \phi(\delta/2) \\ \phi(\delta/4) \end{bmatrix} \tag{4.28}$$

The first row of (**??**) gives the coefficients for Richardson extrapolation scheme.

4.9 Uncertainty estimates for Derivative

We can view the Richardson extrapolation step as a polynomial curve fit in the step size parameter δ. Our desired extrapolated value is seen as simply the constant term coefficient in that polynomial model. Remember though, this polynomial model (see (5.3) and (**??**)) has only a few terms in it with known non-zero coefficients. That is, we will expect a constant term a_0, a term of the form $a_1\delta^4$, and a third term $a_2\delta^6$.

A neat trick to compute the statistical uncertainty in the estimate of our desired derivative is to use statistical methodology for that error estimate. While I do appreciate that there is nothing truly statistical or stochastic in this estimate, the approach still works nicely, providing a very reasonable estimate in practice. A three term Richardson-like extrapolant, then evaluated at four distinct values for δ, will yield an estimate of the standard error of the constant term, with one spare degree of freedom. The uncertainty is then derived by multiplying that standard error by the appropriate percentile from the Students-t distribution.

```
>>> import scipy.stats as ss
>>> allclose(ss.t.cdf(12.7062047361747, 1), 0.975)
True
```

This critical level will yield a two-sided confidence interval of 95 percent.

These error estimates are also of value in a different sense. Since they are efficiently generated at all the different scales, the particular spacing which yields the minimum predicted error is chosen as the best derivative estimate. This has been shown to work consistently well. A spacing too large tends to have large errors of approximation due to the finite difference schemes used. But a too small spacing is bad also, in that we see a significant amplification of least significant fit errors in the approximation. A middle value generally seems to yield quite good results. For example, Derivative will estimate the derivative of e^x automatically. As we see, the final overall spacing used was 0.0078125.

```
>>> import numdifftools as nd
>>> from numpy import exp, allclose
>>> f = nd.Derivative(exp, full_output=True)
>>> val, info = f(1)
>>> allclose(val, 2.71828183)
True
>>> allclose(info.error_estimate, 6.927791673660977e-14)
True
>>> allclose(info.final_step, 0.0078125)
True
```

However, if we force the step size to be artificially large, then approximation error takes over.

```
>>> f = nd.Derivative(exp, step=1, full_output=True)
>>> val, info = f(1)
>>> allclose(val, 3.19452805)
True
>>> allclose(val-exp(1), 0.47624622)
True
>>> allclose(info.final_step, 1)
True
```

And if the step size is forced to be too small, then we see noise dominate the problem.

```
>>> f = nd.Derivative(exp, step=1e-10, full_output=True)
>>> val, info = f(1)
>>> allclose(val, 2.71828093)
True
>>> allclose(val - exp(1), -8.97648138e-07)
True
>>> allclose(info.final_step, 1.0000000e-10)
True
```

Numdifftools, like Goldilocks in the fairy tale bearing her name, stays comfortably in the middle ground.

CHAPTER 5

Reference

Technical reference material that details functions, modules, and objects included in numdifftools, describing what they are and what they do.

.

5.1 Numdifftools summary

5.1.1 numdifftools.core module

Derivative(fun[, step, method, order, n, ...])	Calculate n-th derivative with finite difference approximation
Gradient(fun[, step, method, order, n, ...])	Calculate Gradient with finite difference approximation
Jacobian(fun[, step, method, order, n, ...])	Calculate Jacobian with finite difference approximation
Hessdiag(f[, step, method, order, full_output])	Calculate Hessian diagonal with finite difference approximation
Hessian(f[, step, method, order, full_output])	Calculate Hessian with finite difference approximation
directionaldiff(f, x0, vec, **options)	Return directional derivative of a function of n variables

numdifftools.core.Derivative

class Derivative(*fun*, *step=None*, *method='central'*, *order=2*, *n=1*, *full_output=False*, ***step_options*)

Calculate n-th derivative with finite difference approximation

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

step [float, array-like or StepGenerator object, optional] Defines the spacing used in the approximation. Default is MinStepGenerator(base_step=step, step_ratio=None,

num_extrap=0, **step_options)

if step or method in in ['complex', 'multicomplex'], otherwise

MaxStepGenerator(step_ratio=None, num_extrap=14, **step_options)

The results are extrapolated if the StepGenerator generate more than 3 steps.

method [{'central', 'complex', 'multicomplex', 'forward', 'backward'}] defines the method used in the approximation

order [int, optional] defines the order of the error term in the Taylor approximation used. For 'central' and 'complex' methods, it must be an even number.

n [int, optional] Order of the derivative.

full_output [bool, optional] If *full_output* is False, only the derivative is returned. If *full_output* is True, then (der, r) is returned *der* is the derivative, and *r* is a Results object.

****step_options:** options to pass on to the XXXStepGenerator used.

Returns

der [ndarray] array of derivatives

See also:

Gradient

Hessian

Notes

Complex methods are usually the most accurate provided the function to differentiate is analytic. The complex-step methods also requires fewer steps than the other methods and can work very close to the support of a function. The complex-step derivative has truncation error O(steps**2) for *n=1* and O(steps**4) for *n* larger, so truncation error can be eliminated by choosing steps to be very small. Especially the first order complex-step derivative avoids the problem of round-off error with small steps because there is no subtraction. However, this method fails if fun(x) does not support complex numbers or involves non-analytic functions such as e.g.: abs, max, min. Central difference methods are almost as accurate and has no restriction on type of function. For this reason the 'central' method is the default method, but sometimes one can only allow evaluation in forward or backward direction.

For all methods one should be careful in decreasing the step size too much due to round-off errors.

Higher order approximation methods will generally be more accurate, but may also suffer more from numerical problems. First order methods is usually not recommended.

References

Ridout, M.S. (2009) Statistical applications of the complex-step method of numerical differentiation. The American Statistician, 63, 66-74

K.-L. Lai, J.L. Crassidis, Y. Cheng, J. Kim (2005), New complex step derivative approximations with application to second-order kalman filtering, AIAA Guidance, Navigation and Control Conference, San Francisco, California, August 2005, AIAA-2005-5944.

Lyness, J. M., Moler, C. B. (1966). Vandermonde Systems and Numerical Differentiation. *Numerische Mathematik*.

Lyness, J. M., Moler, C. B. (1969). Generalized Romberg Methods for Integrals of Derivatives. *Numerische Mathematik*.

Examples

```
>>> import numpy as np
>>> import numdifftools as nd
```

1'st derivative of exp(x), at x == 1

```
>>> fd = nd.Derivative(np.exp)
>>> np.allclose(fd(1), 2.71828183)
True
```

```
>>> d2 = fd([1, 2])
>>> np.allclose(d2, [ 2.71828183,  7.3890561 ])
True
```

```
>>> def f(x):
...     return x**3 + x**2
```

```
>>> df = nd.Derivative(f)
>>> np.allclose(df(1), 5)
True
>>> ddf = nd.Derivative(f, n=2)
>>> np.allclose(ddf(1), 8)
True
```

Methods

__call__	(callable with the following parameters:) x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.

__init__(*self*, *fun*, *step=None*, *method='central'*, *order=2*, *n=1*, *full_output=False*, ***step_options*)

Methods

__init__(self, fun[, step, method, order, ...])	
set_richardson_rule(self, step_ratio[, ...])	

numdifftools.core.Gradient

class Gradient(*fun*, *step=None*, *method='central'*, *order=2*, *n=1*, *full_output=False*, ***step_options*)

Calculate Gradient with finite difference approximation

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

step [float, array-like or StepGenerator object, optional] Defines the spacing used in the approximation. Default is MinStepGenerator(base_step=step, step_ratio=None,

num_extrap=0, **step_options)

if step or method in in ['complex', 'multicomplex'], otherwise

MaxStepGenerator(step_ratio=None, num_extrap=14, **step_options)

The results are extrapolated if the StepGenerator generate more than 3 steps.

method [{'central', 'complex', 'multicomplex', 'forward', 'backward'}] defines the method used in the approximation

order [int, optional] defines the order of the error term in the Taylor approximation used. For 'central' and 'complex' methods, it must be an even number.

full_output [bool, optional] If *full_output* is False, only the derivative is returned. If *full_output* is True, then (der, r) is returned *der* is the derivative, and *r* is a Results object.

****step_options:** options to pass on to the XXXStepGenerator used.

Returns

grad [array] gradient

See also:

Derivative*, *Hessian*, *Jacobian

Notes

Complex methods are usually the most accurate provided the function to differentiate is analytic. The complex-step methods also requires fewer steps than the other methods and can work very close to the support of a function. The complex-step derivative has truncation error O(steps**2) for *n=1* and O(steps**4) for *n* larger, so truncation error can be eliminated by choosing steps to be very small. Especially the first order complex-step derivative avoids the problem of round-off error with small steps because there is no subtraction. However, this method fails if fun(x) does not support complex numbers or involves non-analytic functions such as e.g.: abs, max, min. Central difference methods are almost as accurate and has no restriction on type of function. For this reason the 'central' method is the default method, but sometimes one can only allow evaluation in forward or backward direction.

For all methods one should be careful in decreasing the step size too much due to round-off errors.

Higher order approximation methods will generally be more accurate, but may also suffer more from numerical problems. First order methods is usually not recommended.

If x0 is an n x m array, then fun is assumed to be a function of n * m variables.

References

Ridout, M.S. (2009) Statistical applications of the complex-step method of numerical differentiation. The American Statistician, 63, 66-74

K.-L. Lai, J.L. Crassidis, Y. Cheng, J. Kim (2005), New complex step derivative approximations with application to second-order kalman filtering, AIAA Guidance, Navigation and Control Conference, San Francisco, California, August 2005, AIAA-2005-5944.

Lyness, J. M., Moler, C. B. (1966). Vandermonde Systems and Numerical Differentiation. *Numerische Mathematik*.

Lyness, J. M., Moler, C. B. (1969). Generalized Romberg Methods for Integrals of Derivatives. *Numerische Mathematik*.

Examples

```
>>> import numpy as np
>>> import numdifftools as nd
>>> fun = lambda x: np.sum(x**2)
>>> dfun = nd.Gradient(fun)
>>> np.allclose(dfun([1,2,3]), [ 2.,  4.,  6.])
True
```

At [x,y] = [1,1], compute the numerical gradient # of the function sin(x-y) + y*exp(x)

```
>>> sin = np.sin; exp = np.exp
>>> x, y = 1, 1
>>> z = lambda xy: sin(xy[0]-xy[1]) + xy[1]*exp(xy[0])
>>> dz = nd.Gradient(z)
>>> dz_dx, dz_dy = dz([x, y])
>>> np.allclose([dz_dx, dz_dy],
...             [ 3.7182818284590686, 1.7182818284590162])
True
```

At the global minimizer (1,1) of the Rosenbrock function, # compute the gradient. It should be essentially zero.

```
>>> rosen = lambda x : (1-x[0])**2 + 105.*(x[1]-x[0]**2)**2
>>> grad_rosen = nd.Gradient(rosen)
>>> df_dx, df_dy = grad_rosen([x, y])
>>> np.allclose([df_dx, df_dy], [0, 0])
True
```

Methods

__call__	(callable with the following parameters:) x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.

__init__(*self*, *fun*, *step=None*, *method='central'*, *order=2*, *n=1*, *full_output=False*, ***step_options*)

Methods

__init__(self, fun[, step, method, order, ...])	
set_richardson_rule(self, step_ratio[, ...])	

numdifftools.core.Jacobian

class Jacobian(*fun*, *step=None*, *method='central'*, *order=2*, *n=1*, *full_output=False*, ***step_options*)

Calculate Jacobian with finite difference approximation

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

step [float, array-like or StepGenerator object, optional] Defines the spacing used in the approximation. Default is MinStepGenerator(base_step=step, step_ratio=None,

num_extrap=0, **step_options)

if step or method in in ['complex', 'multicomplex'], otherwise

MaxStepGenerator(step_ratio=None, num_extrap=14, **step_options)

The results are extrapolated if the StepGenerator generate more than 3 steps.

method [{'central', 'complex', 'multicomplex', 'forward', 'backward'}] defines the method used in the approximation

order [int, optional] defines the order of the error term in the Taylor approximation used. For 'central' and 'complex' methods, it must be an even number.

full_output [bool, optional] If *full_output* is False, only the derivative is returned. If *full_output* is True, then (der, r) is returned *der* is the derivative, and *r* is a Results object.

****step_options:** options to pass on to the XXXStepGenerator used.

Returns

jacob [array] Jacobian

See also:

Derivative*, *Hessian*, *Gradient

Notes

Complex methods are usually the most accurate provided the function to differentiate is analytic. The complex-step methods also requires fewer steps than the other methods and can work very close to the support of a function. The complex-step derivative has truncation error O(steps**2) for *n=1* and O(steps**4) for *n* larger, so truncation error can be eliminated by choosing steps to be very small. Especially the first order complex-step derivative avoids the problem of round-off error with small steps because there is no subtraction. However, this method fails if fun(x) does not support complex numbers or involves non-analytic functions such as e.g.: abs, max, min. Central difference methods are almost as accurate and has no restriction on type of function. For this reason the 'central' method is the default method, but sometimes one can only allow evaluation in forward or backward direction.

For all methods one should be careful in decreasing the step size too much due to round-off errors.

Higher order approximation methods will generally be more accurate, but may also suffer more from numerical problems. First order methods is usually not recommended.

If fun returns a 1d array, it returns a Jacobian. If a 2d array is returned by fun (e.g., with a value for each observation), it returns a 3d array with the Jacobian of each observation with shape xk x nobs x xk. I.e., the Jacobian of the first observation would be [:, 0, :]

References

Ridout, M.S. (2009) Statistical applications of the complex-step method of numerical differentiation. The American Statistician, 63, 66-74

K.-L. Lai, J.L. Crassidis, Y. Cheng, J. Kim (2005), New complex step derivative approximations with application to second-order kalman filtering, AIAA Guidance, Navigation and Control Conference, San Francisco, California, August 2005, AIAA-2005-5944.

Lyness, J. M., Moler, C. B. (1966). Vandermonde Systems and Numerical Differentiation. *Numerische Mathematik*.

Lyness, J. M., Moler, C. B. (1969). Generalized Romberg Methods for Integrals of Derivatives. *Numerische Mathematik*.

Examples

```
>>> import numdifftools as nd
```

#(nonlinear least squares)

```
>>> xdata = np.arange(0,1,0.1)
>>> ydata = 1+2*np.exp(0.75*xdata)
>>> fun = lambda c: (c[0]+c[1]*np.exp(c[2]*xdata) - ydata)**2
```

```
>>> np.allclose(fun([1, 2, 0.75]).shape,  (10,))
True
```

```
>>> jfun = nd.Jacobian(fun)
>>> val = jfun([1, 2, 0.75])
>>> np.allclose(val, np.zeros((10,3)))
True
```

```
>>> fun2 = lambda x : x[0]*x[1]*x[2]**2
>>> jfun2 = nd.Jacobian(fun2)
>>> np.allclose(jfun2([1.,2.,3.]), [[18., 9., 12.]])
True
```

```
>>> fun3 = lambda x : np.vstack((x[0]*x[1]*x[2]**2, x[0]*x[1]*x[2]))
>>> jfun3 = nd.Jacobian(fun3)
>>> np.allclose(jfun3([1., 2., 3.]), [[18., 9., 12.], [6., 3., 2.]])
True
>>> np.allclose(jfun3([4., 5., 6.]), [[180., 144., 240.], [30., 24., 20.]])
True
>>> np.allclose(jfun3(np.array([[1.,2.,3.]]).T), [[ 18.,   9.,  12.],
...                                               [ 6.,   3.,   2.]])
True
```

Methods

__call__(callable with the following parameters:) x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.	

__init__(*self*, *fun*, *step=None*, *method='central'*, *order=2*, *n=1*, *full_output=False*, ***step_options*)

Methods

__init__(self, fun[, step, method, order, ...])	
set_richardson_rule(self, step_ratio[, ...])	

numdifftools.core.Hessdiag

class Hessdiag(*f*, *step=None*, *method='central'*, *order=2*, *full_output=False*, ***step_options*)

Calculate Hessian diagonal with finite difference approximation

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

step [float, array-like or StepGenerator object, optional] Defines the spacing used in the approximation. Default is MinStepGenerator(base_step=step, step_ratio=None,

num_extrap=0, **step_options)

if step or method in in ['complex', 'multicomplex'], otherwise

MaxStepGenerator(step_ratio=None, num_extrap=14, **step_options)

The results are extrapolated if the StepGenerator generate more than 3 steps.

method [{'central', 'complex', 'multicomplex', 'forward', 'backward'}] defines the method used in the approximationorder : int, optional defines the order of the error term in the Taylor approximation used. For 'central' and 'complex' methods, it must be an even number.

full_output [bool, optional] If *full_output* is False, only the derivative is returned. If *full_output* is True, then (der, r) is returned *der* is the derivative, and *r* is a Results object.

****step_options:** options to pass on to the XXXStepGenerator used.

Returns

hessdiag [array] hessian diagonal

See also:

Derivative*, *Hessian*, *Jacobian*, *Gradient

Notes

Complex methods are usually the most accurate provided the function to differentiate is analytic. The complex-step methods also requires fewer steps than the other methods and can work very close to the support of a function. The complex-step derivative has truncation error O(steps**2) for *n=1* and O(steps**4) for *n* larger, so truncation error can be eliminated by choosing steps to be very small. Especially the first order complex-step derivative avoids the problem of round-off error with small steps because there is no subtraction. However, this method fails if fun(x) does not support complex numbers or involves non-analytic functions such as e.g.: abs, max, min. Central difference methods are almost as accurate and has no restriction on type of function. For this reason the 'central' method is the default method, but sometimes one can only allow evaluation in forward or backward direction.

For all methods one should be careful in decreasing the step size too much due to round-off errors.

Higher order approximation methods will generally be more accurate, but may also suffer more from numerical problems. First order methods is usually not recommended.

References

Ridout, M.S. (2009) Statistical applications of the complex-step method of numerical differentiation. The American Statistician, 63, 66-74

K.-L. Lai, J.L. Crassidis, Y. Cheng, J. Kim (2005), New complex step derivative approximations with application to second-order kalman filtering, AIAA Guidance, Navigation and Control Conference, San Francisco, California, August 2005, AIAA-2005-5944.

Lyness, J. M., Moler, C. B. (1966). Vandermonde Systems and Numerical Differentiation. *Numerische Mathematik*.

Lyness, J. M., Moler, C. B. (1969). Generalized Romberg Methods for Integrals of Derivatives. *Numerische Mathematik*.

Examples

```
>>> import numpy as np
>>> import numdifftools as nd
>>> fun = lambda x : x[0] + x[1]**2 + x[2]**3
>>> Hfun = nd.Hessdiag(fun, full_output=True)
>>> hd, info = Hfun([1,2,3])
>>> np.allclose(hd, [0.,   2.,  18.])
True
```

```
>>> np.all(info.error_estimate < 1e-11)
True
```

Methods

__call__	(callable with the following parameters:) x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.

__init__ (*self*, *f*, *step=None*, *method='central'*, *order=2*, *full_output=False*, ***step_options*)

Methods

__init__(self, f[, step, method, order, ...])	
set_richardson_rule(self, step_ratio[, ...])	

numdifftools.core.Hessian

class Hessian (*f*, *step=None*, *method='central'*, *order=2*, *full_output=False*, ***step_options*)

Calculate Hessian with finite difference approximation

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

step [float, array-like or StepGenerator object, optional] Defines the spacing used in the approximation. Default is MinStepGenerator(base_step=step, step_ratio=None,

num_extrap=0, **step_options)

if step or method in in ['complex', 'multicomplex'], otherwise

MaxStepGenerator(step_ratio=None, num_extrap=14, **step_options)

The results are extrapolated if the StepGenerator generate more than 3 steps.

method [{'central', 'complex', 'multicomplex', 'forward', 'backward'}] defines the method used in the approximation

full_output [bool, optional] If *full_output* is False, only the derivative is returned. If *full_output* is True, then (der, r) is returned *der* is the derivative, and *r* is a Results object.

****step_options:** options to pass on to the XXXStepGenerator used.

Returns

hess [ndarray] array of partial second derivatives, Hessian

See also:

Derivative, *Hessian*

Notes

Complex methods are usually the most accurate provided the function to differentiate is analytic. The complex-step methods also requires fewer steps than the other methods and can work very close to the support of a function. The complex-step derivative has truncation error O(steps**2) for *n=1* and O(steps**4)

for *n* larger, so truncation error can be eliminated by choosing steps to be very small. Especially the first order complex-step derivative avoids the problem of round-off error with small steps because there is no subtraction. However, this method fails if fun(x) does not support complex numbers or involves non-analytic functions such as e.g.: abs, max, min. Central difference methods are almost as accurate and has no restriction on type of function. For this reason the 'central' method is the default method, but sometimes one can only allow evaluation in forward or backward direction.

For all methods one should be careful in decreasing the step size too much due to round-off errors.

Computes the Hessian according to method as: 'forward' (5.1), 'central' (5.2) and 'complex' (5.3):

$$((f(x + d_j e_j + d_k e_k) - f(x + d_j e_j)))/(d_j d_k) \tag{5.1}$$

$$((f(x + d_j e_j + d_k e_k) - f(x + d_j e_j - d_k e_k)) - (f(x - d_j e_j + d_k e_k) - f(x - d_j e_j - d_k e_k))/(4 d_j d_k) \tag{5.2}$$

$$imag(f(x + i d_j e_j + d_k e_k) - f(x + i d_j e_j - d_k e_k))/(2 d_j d_k) \tag{5.3}$$

where e_j is a vector with element j is one and the rest are zero and d_j is a scalar spacing $steps_j$.

References

Ridout, M.S. (2009) Statistical applications of the complex-step method of numerical differentiation. The American Statistician, 63, 66-74

K.-L. Lai, J.L. Crassidis, Y. Cheng, J. Kim (2005), New complex step derivative approximations with application to second-order kalman filtering, AIAA Guidance, Navigation and Control Conference, San Francisco, California, August 2005, AIAA-2005-5944.

Lyness, J. M., Moler, C. B. (1966). Vandermonde Systems and Numerical Differentiation. *Numerische Mathematik*.

Lyness, J. M., Moler, C. B. (1969). Generalized Romberg Methods for Integrals of Derivatives. *Numerische Mathematik*.

Examples

```
>>> import numpy as np
>>> import numdifftools as nd
```

Rosenbrock function, minimized at [1,1]

```
>>> rosen = lambda x : (1.-x[0])**2 + 105*(x[1]-x[0]**2)**2
>>> Hfun = nd.Hessian(rosen)
>>> h = Hfun([1, 1])
>>> h
array([[ 842., -420.],
       [-420.,  210.]])
```

cos(x-y), at (0,0)

```
>>> cos = np.cos
>>> fun = lambda xy : cos(xy[0]-xy[1])
>>> Hfun2 = nd.Hessian(fun)
>>> h2 = Hfun2([0, 0])
```

```
>>> h2
array([[-1.,  1.],
       [ 1., -1.]])
```

Methods

__call__(callable with the following parameters:) x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.	

__init__ (*self*, *f*, *step=None*, *method='central'*, *order=2*, *full_output=False*, ***step_options*)

Methods

__init__(self, f[, step, method, order, ...])	
set_richardson_rule(self, step_ratio[, ...])	

numdifftools.core.directionaldiff

directionaldiff (*f*, *x0*, *vec*, ***options*)

Return directional derivative of a function of n variables

Parameters

f: function analytical function to differentiate.

x0: array vector location at which to differentiate 'f'. If x0 is an nXm array, then 'f' is assumed to be a function of n*m variables.

vec: array vector defining the line along which to take the derivative. It should be the same size as x0, but need not be a vector of unit length.

****options:** optional arguments to pass on to Derivative.

Returns

dder: scalar estimate of the first derivative of 'f' in the specified direction.

See also:

Derivative

Gradient

Examples

At the global minimizer (1,1) of the Rosenbrock function, compute the directional derivative in the direction [1 2]

```
>>> import numpy as np
>>> import numdifftools as nd
>>> vec = np.r_[1, 2]
>>> rosen = lambda x: (1-x[0])**2 + 105*(x[1]-x[0]**2)**2
>>> dd, info = nd.directionaldiff(rosen, [1, 1], vec, full_output=True)
>>> np.allclose(dd, 0)
True
```

```
>>> np.abs(info.error_estimate)<1e-14
True
```

5.1.2 Step generators

BasicMaxStepGenerator(base_step, step_ratio, ...)		Generates a sequence of steps of decreasing magnitude
BasicMinStepGenerator(base_step, step_ratio, ...)		Generates a sequence of steps of decreasing magnitude
MinStepGenerator([base_step, step_ratio, ...])		Generates a sequence of steps
MaxStepGenerator([base_step, step_ratio, ...])		Generates a sequence of steps

numdifftools.step_generators.BasicMaxStepGenerator

class BasicMaxStepGenerator(*base_step*, *step_ratio*, *num_steps*, *offset=0*)

Generates a sequence of steps of decreasing magnitude

where steps = base_step * step_ratio ** (-i + offset)

for i=0, 1,.., num_steps-1.

Parameters

base_step [float, array-like.] Defines the start step, i.e., maximum step

step_ratio [real scalar.] Ratio between sequential steps generated. Note: Ratio > 1

num_steps [scalar integer.] defines number of steps generated.

offset [real scalar, optional, default 0] offset to the base step

Examples

```
>>> from numdifftools.step_generators import BasicMaxStepGenerator
>>> step_gen = BasicMaxStepGenerator(base_step=2.0, step_ratio=2,
...                                  num_steps=4)
>>> [s for s in step_gen()]
[2.0, 1.0, 0.5, 0.25]
```

__init__(*self*, *base_step*, *step_ratio*, *num_steps*, *offset=0*)

Methods

__init__(self, base_step, step_ratio, num_steps)	

numdifftools.step_generators.BasicMinStepGenerator

class BasicMinStepGenerator(*base_step*, *step_ratio*, *num_steps*, *offset=0*)

Generates a sequence of steps of decreasing magnitude

where steps = base_step * step_ratio ** (i + offset), i=num_steps-1,... 1, 0.

Parameters

base_step [float, array-like.] Defines the end step, i.e., minimum step

step_ratio [real scalar.] Ratio between sequential steps generated. Note: Ratio > 1

num_steps [scalar integer.] defines number of steps generated.

offset [real scalar, optional, default 0] offset to the base step

Examples

```
>>> from numdifftools.step_generators import BasicMinStepGenerator
>>> step_gen = BasicMinStepGenerator(base_step=0.25, step_ratio=2,
...                                  num_steps=4)
>>> [s for s in step_gen()]
[2.0, 1.0, 0.5, 0.25]
```

__init__(*self*, *base_step*, *step_ratio*, *num_steps*, *offset=0*)

Methods

__init__(self, base_step, step_ratio, num_steps)

numdifftools.step_generators.MinStepGenerator

class MinStepGenerator(*base_step=None*, *step_ratio=2.0*, *num_steps=None*, *step_nom=None*, *offset=0*, *num_extrap=0*, *use_exact_steps=True*, *check_num_steps=True*, *scale=None*)

Generates a sequence of steps

where steps = step_nom * base_step * step_ratio ** (i + offset)

for i = num_steps-1,... 1, 0.

Parameters

base_step [float, array-like, optional] Defines the minimum step, if None, the value is set to EPS**(1/scale)

step_ratio [real scalar, optional, default 2] Ratio between sequential steps generated. Note: Ratio > 1 If None then step_ratio is 2 for n=1 otherwise step_ratio is 1.6

num_steps [scalar integer, optional, default min_num_steps + num_extrap] defines number of steps generated. It should be larger than min_num_steps = (n + order - 1) / fact where fact is 1, 2 or 4 depending on differentiation method used.

step_nom [default maximum(log(1+|x|), 1)] Nominal step where x is supplied at runtime through the __call__ method.

offset [real scalar, optional, default 0] offset to the base step

num_extrap [scalar integer, default 0] number of points used for extrapolation

check_num_steps [boolean, default True] If True make sure num_steps is larger than the minimum required steps.

use_exact_steps [boolean, default True] If true make sure exact steps are generated

scale [real scalar, optional] scale used in base step. If not None it will override the default computed with the default_scale function.

__init__(*self*, *base_step=None*, *step_ratio=2.0*, *num_steps=None*, *step_nom=None*, *offset=0*, *num_extrap=0*, *use_exact_steps=True*, *check_num_steps=True*, *scale=None*)

Methods

__init__(self[, base_step, step_ratio, ...])	
step_generator_function(self, x[, method, ...])	

numdifftools.step_generators.MaxStepGenerator

class MaxStepGenerator(*base_step=2.0*, *step_ratio=2.0*, *num_steps=15*, *step_nom=None*, *offset=0*, *num_extrap=0*, *use_exact_steps=False*, *check_num_steps=True*, *scale=500*)

Generates a sequence of steps

where steps = step_nom * base_step * step_ratio ** (-i + offset)

for i = 0, 1, ..., num_steps-1.

Parameters

base_step [float, array-like, default 2.0] Defines the maximum step, if None, the value is set to EPS**(1/scale)

step_ratio [real scalar, optional, default 2] Ratio between sequential steps generated. Note: Ratio > 1 If None then step_ratio is 2 for n=1 otherwise step_ratio is 1.6

num_steps [scalar integer, optional, default min_num_steps + num_extrap] defines number of steps generated. It should be larger than min_num_steps = (n + order - 1) / fact where fact is 1, 2 or 4 depending on differentiation method used.

step_nom [default maximum(log(1+|x|), 1)] Nominal step where x is supplied at runtime through the __call__ method.

offset [real scalar, optional, default 0] offset to the base step

num_extrap [scalar integer, default 0] number of points used for extrapolation

check_num_steps [boolean, default True] If True make sure num_steps is larger than the minimum required steps.

use_exact_steps [boolean, default True] If true make sure exact steps are generated

scale [real scalar, default 500] scale used in base step.

__init__(*self*, *base_step=2.0*, *step_ratio=2.0*, *num_steps=15*, *step_nom=None*, *offset=0*, *num_extrap=0*, *use_exact_steps=False*, *check_num_steps=True*, *scale=500*)

Methods

__init__(self[, base_step, step_ratio, ...])	
step_generator_function(self, x[, method, ...])	

5.1.3 numdifftools.extrapolation module

convolve(sequence, rule, **kwds)	Wrapper around scipy.ndimage.convolve1d that allows complex input.
Dea([limexp])	Extrapolate a slowly convergent sequence
dea3(v0, v1, v2[, symmetric])	Extrapolate a slowly convergent sequence
Richardson([step_ratio, step, order, num_terms])	Extrapolates as sequence with Richardsons method

numdifftools.extrapolation.convolve

convolve(*sequence*, *rule*, ***kwds*)
Wrapper around scipy.ndimage.convolve1d that allows complex input.

numdifftools.extrapolation.Dea

class Dea(*limexp=3*)
Extrapolate a slowly convergent sequence

LIMEXP is the maximum number of elements the epsilon table data can contain. The epsilon table is stored in the first (LIMEXP+2) entries of EPSTAB.

Notes

List of major variables:

E0,E1,E2,E3 - DOUBLE PRECISION The 4 elements on which the computation of a new element in the epsilon table is based.

NRES - INTEGER Number of extrapolation results actually generated by the epsilon algorithm in prior calls to the routine.

NEWELM - INTEGER Number of elements to be computed in the new diagonal of the epsilon table. The condensed epsilon table is computed. Only those elements needed for the computation of the next diagonal are preserved.

RES - DOUBLE PREISION New element in the new diagonal of the epsilon table.

ERROR - DOUBLE PRECISION An estimate of the absolute error of RES. Routine decides whether RESULT=RES or RESULT=SVALUE by comparing ERROR with abserr from the previous call.

RES3LA - DOUBLE PREISION Vector of DIMENSION 3 containing at most the last 3 results.

__init__(*self*, *limexp=3*)

Methods

__init__(self[, limexp])

numdifftools.extrapolation.dea3

dea3(*v0*, *v1*, *v2*, *symmetric=False*)
Extrapolate a slowly convergent sequence

Parameters

v0, v1, v2 [array-like] 3 values of a convergent sequence to extrapolate

Returns

result [array-like] extrapolated value

abserr [array-like] absolute error estimate

See also:

dea

Notes

DEA3 attempts to extrapolate nonlinearly to a better estimate of the sequence's limiting value, thus improving the rate of convergence. The routine is based on the epsilon algorithm of P. Wynn, see *[1]*.

References

[1]

Examples

integrate sin(x) from 0 to pi/2

```
>>> import numpy as np
>>> import numdifftools as nd
>>> Ei= np.zeros(3)
>>> linfun = lambda i : np.linspace(0, np.pi/2., 2**(i+5)+1)
>>> for k in np.arange(3):
...     x = linfun(k)
...     Ei[k] = np.trapz(np.sin(x),x)
>>> [En, err] = nd.dea3(Ei[0], Ei[1], Ei[2])
>>> truErr = np.abs(En-1.)
>>> np.all(truErr < err)
True
>>> np.allclose(En, 1)
True
>>> np.all(np.abs(Ei-1)<1e-3)
True
```

numdifftools.extrapolation.Richardson

class Richardson(*step_ratio=2.0*, *step=1*, *order=1*, *num_terms=2*)

Extrapolates as sequence with Richardsons method

Notes

Suppose you have series expansion that goes like this

L = f(h) + a0 * h^p_0 + a1 * h^p_1+ a2 * h^p_2 + ...

where p_i = order + step * i and f(h) -> L as h -> 0, but f(0) != L.

If we evaluate the right hand side for different stepsizes h we can fit a polynomial to that sequence of approximations. This is exactly what this class does.

Examples

```
>>> import numpy as np
>>> import numdifftools as nd
>>> n = 3
>>> Ei = np.zeros((n,1))
>>> h = np.zeros((n,1))
>>> linfun = lambda i : np.linspace(0, np.pi/2., 2**(i+5)+1)
>>> for k in np.arange(n):
...     x = linfun(k)
...     h[k] = x[1]
```

```
...     Ei[k] = np.trapz(np.sin(x),x)
>>> En, err, step = nd.Richardson(step=1, order=1)(Ei, h)
>>> truErr = np.abs(En-1.)
>>> np.all(truErr < err)
True
>>> np.all(np.abs(Ei-1)<1e-3)
True
>>> np.allclose(En, 1)
True
```

__init__(*self*, *step_ratio=2.0*, *step=1*, *order=1*, *num_terms=2*)

Methods

__init__(self[, step_ratio, step, order, ...])
extrapolate(self, sequence, steps)
rule(self[, sequence_length])

5.1.4 numdifftools.limits module

CStepGenerator([base_step, step_ratio, ...])	Generates a sequence of steps
Limit(fun[, step, method, order, full_output])	Compute limit of a function at a given point
Residue(f[, step, method, order, ...])	Compute residue of a function at a given point

numdifftools.limits.CStepGenerator

class CStepGenerator(*base_step=None*, *step_ratio=4.0*, *num_steps=None*, *step_nom=None*, *offset=0*, *scale=1.2*, *use_exact_steps=True*, *path='radial'*, *dtheta=0.39269908169872414*, ***kwds*)

Generates a sequence of steps

where

steps = base_step * step_nom * (exp(1j*dtheta) * step_ratio) ** (i + offset)

for i = 0, 1, ..., num_steps-1

Parameters

base_step [float, array-like, default None] Defines the minimum step, if None, the value is set to EPS**(1/scale)

step_ratio [real scalar, optional, default 4.0] Ratio between sequential steps generated.

num_steps [scalar integer, optional,] defines number of steps generated. If None the value is 2 * int(round(16.0/log(abs(step_ratio)))) + 1

step_nom [default maximum(log(1+|x|), 1)] Nominal step where x is supplied at runtime through the __call__ method.

offset [real scalar, optional, default 0] offset to the base step

use_exact_steps [boolean] If true make sure exact steps are generated

scale [real scalar, default 1.2] scale used in base step.

path ['spiral' or 'radial'] Specifies the type of path to take the limit along.

dtheta: real scalar If the path is spiral it will follow an exponential spiral into the limit, with angular steps at dtheta radians.

__init__(*self*, *base_step=None*, *step_ratio=4.0*, *num_steps=None*, *step_nom=None*, *offset=0*, *scale=1.2*, *use_exact_steps=True*, *path='radial'*, *dtheta=0.39269908169872414*, ***kwds*)

Methods

__init__(self[, base_step, step_ratio, ...])	
step_generator_function(self, x[, method, ...])	

numdifftools.limits.Limit

class Limit(*fun*, *step=None*, *method='above'*, *order=4*, *full_output=False*, ***options*)

Compute limit of a function at a given point

Parameters

fun [callable] function fun(z, **args*, ***kwds*) to compute the limit for z->z0. The function, fun, is assumed to return a result of the same shape and size as its input, *z*.

step: float, complex, array-like or StepGenerator object, optional Defines the spacing used in the approximation. Default is CStepGenerator(base_step=step, **options)

method [{'above', 'below'}] defines if the limit is taken from *above* or *below*

order: positive scalar integer, optional. defines the order of approximation used to find the specified limit. The order must be member of [1 2 3 4 5 6 7 8]. 4 is a good compromise.

full_output: bool If true return additional info.

options: options to pass on to CStepGenerator

Returns

limit_fz: array like estimated limit of f(z) as z –> z0

info: Only given if full_output is True and contains the following:

error estimate: ndarray 95 % uncertainty estimate around the limit, such that abs(limit_fz - lim z->z0 f(z)) < error_estimate

final_step: ndarray final step used in approximation

Notes

Limit computes the limit of a given function at a specified point, z0. When the function is evaluable at the point in question, this is a simple task. But when the function cannot be evaluated at that location due to a singularity, you may need a tool to compute the limit. *Limit* does this, as well as produce an uncertainty estimate in the final result.

The methods used by *Limit* are Richardson extrapolation in a combination with Wynn's epsilon algorithm which also yield an error estimate. The user can specify the method order, as well as the path into z0. z0 may be real or complex. *Limit* uses a proportionally cascaded series of function evaluations, moving away from your point of evaluation along a path along the real line (or in the complex plane for complex z0 or step.) The *step_ratio* is the ratio used between sequential steps. The sign of step allows you to specify a limit from above or below. Negative values of step will cause the limit to be taken approaching z0 from below.

A smaller *step_ratio* means that *Limit* will take more function evaluations to evaluate the limit, but the result will potentially be less accurate. The *step_ratio* MUST be a scalar larger than 1. A value in the range [2,100] is recommended. 4 seems a good compromise.

```
>>> import numpy as np
>>> from numdifftools.limits import Limit
>>> def f(x): return np.sin(x)/x
>>> lim_f0, err = Limit(f, full_output=True)(0)
>>> np.allclose(lim_f0, 1)
True
>>> np.allclose(err.error_estimate, 1.77249444610966e-15)
True
```

Compute the derivative of cos(x) at x == pi/2. It should be -1. The limit will be taken as a function of the differential parameter, dx.

```
>>> x0 = np.pi/2;
>>> def g(x): return (np.cos(x0+x)-np.cos(x0))/x
>>> lim_g0, err = Limit(g, full_output=True)(0)
>>> np.allclose(lim_g0, -1)
True
>>> err.error_estimate < 1e-14
True
```

Compute the residue at a first order pole at z = 0 The function 1./(1-exp(2*z)) has a pole at z == 0. The residue is given by the limit of z*fun(z) as z –> 0. Here, that residue should be -0.5.

```
>>> def h(z): return -z/(np.expm1(2*z))
>>> lim_h0, err = Limit(h, full_output=True)(0)
>>> np.allclose(lim_h0, -0.5)
True
>>> err.error_estimate < 1e-14
True
```

Compute the residue of function 1./sin(z)**2 at z = 0. This pole is of second order thus the residue is given by the limit of z**2*fun(z) as z –> 0.

```
>>> def g(z): return z**2/(np.sin(z)**2)
>>> lim_gpi, err = Limit(g, full_output=True)(0)
>>> np.allclose(lim_gpi, 1)
True
>>> err.error_estimate < 1e-14
True
```

A more difficult limit is one where there is significant subtractive cancellation at the limit point. In the following example, the cancellation is second order. The true limit should be 0.5.

```
>>> def k(x): return (x*np.exp(x)-np.expm1(x))/x**2
>>> lim_k0,err = Limit(k, full_output=True)(0)
>>> np.allclose(lim_k0, 0.5)
True
>>> err.error_estimate < 1.0e-8
True
```

```
>>> def h(x): return  (x-np.sin(x))/x**3
>>> lim_h0, err = Limit(h, full_output=True)(0)
>>> np.allclose(lim_h0, 1./6)
True
>>> err.error_estimate < 1e-8
True
```

__init__(*self*, *fun*, *step=None*, *method='above'*, *order=4*, *full_output=False*, ***options*)

Methods

`__init__`(self, fun[, step, method, order, ...])	
`limit`(self, x, *args, **kwds)	

numdifftools.limits.Residue

class Residue (*f*, *step=None*, *method='above'*, *order=None*, *pole_order=1*, *full_output=False*, ***options*)

Compute residue of a function at a given point

Parameters

fun [callable] function fun(z, **args*, ***kwds*) to compute the Residue at z=z0. The function, fun, is assumed to return a result of the same shape and size as its input, *z*.

step: float, complex, array-like or StepGenerator object, optional Defines the spacing used in the approximation. Default is CStepGenerator(base_step=step, **options)

method [{'above', 'below'}] defines if the limit is taken from *above* or *below*

order: positive scalar integer, optional. defines the order of approximation used to find the specified limit. The order must be member of [1 2 3 4 5 6 7 8]. 4 is a good compromise.

pole_order [scalar integer] specifies the order of the pole at z0.

full_output: bool If true return additional info.

options: options to pass on to CStepGenerator

Returns

res_fz: array like estimated residue, i.e., limit of f(z)*(z-z0)**pole_order as z –> z0 When the residue is estimated as approximately zero,

the wrong order pole may have been specified.

info: namedtuple, Only given if full_output is True and contains the following:

error estimate: ndarray 95 % uncertainty estimate around the residue, such that abs(res_fz - lim z->z0 f(z)*(z-z0)**pole_order) < error_estimate Large uncertainties here suggest that the wrong order pole was specified for f(z0).

final_step: ndarray final step used in approximation

Notes

Residue computes the residue of a given function at a simple first order pole, or at a second order pole.

The methods used by residue are polynomial extrapolants, which also yield an error estimate. The user can specify the method order, as well as the order of the pole.

z0 - scalar point at which to compute the residue. z0 may be real or complex.

See the document DERIVEST.pdf for more explanation of the algorithms behind the parameters of Residue. In most cases, the user should never need to specify anything other than possibly the PoleOrder.

Examples

A first order pole at z = 0

```
>>> from numdifftools.limits import Residue
>>> def f(z): return -1./(np.expm1(2*z))
>>> res_f, info = Residue(f, full_output=True)(0)
>>> np.allclose(res_f, -0.5)
True
>>> info.error_estimate < 1e-14
True
```

A second order pole around z = 0 and z = pi >>> def h(z): return 1.0/np.sin(z)**2 >>> res_h, info = Residue(h, full_output=True, pole_order=2)([0, np.pi]) >>> np.allclose(res_h, 1) True >>> (info.error_estimate < 1e-10).all() True

__init__(*self*, *f*, *step=None*, *method='above'*, *order=None*, *pole_order=1*, *full_output=False*, ***options*)

Methods

__init__(self, f[, step, method, order, ...])
limit(self, x, *args, **kwds)

5.1.5 numdifftools.multicomplex module

Bicomplex(z1, z2)	Creates an instance of a Bicomplex object.

numdifftools.multicomplex.Bicomplex

class Bicomplex(*z1*, *z2*)

Creates an instance of a Bicomplex object. zeta = z1 + j*z2, where z1 and z2 are complex numbers.

__init__(*self*, *z1*, *z2*)

Methods

__init__(self, z1, z2)
arccos(self)
arccosh(self)
arcsin(self)
arcsinh(self)
arctan(self)
arctanh(self)
arg_c(self)
arg_c1p(self)
asarray(other)
conjugate(self)
cos(self)
cosh(self)
cot(self)
coth(self)
csc(self)
csch(self)
dot(self, other)
exp(self)

Continued on next page

Table 5.20 – continued from previous page

exp2(self)	
expm1(self)	
flat(self, index)	
log(self)	
log10(self)	
log1p(self)	
log2(self)	
logaddexp(self, other)	
logaddexp2(self, other)	
mat2bicomp(arr)	
mod_c(self)	Complex modulus
norm(self)	
sec(self)	
sech(self)	
sin(self)	
sinh(self)	
sqrt(self)	
tan(self)	
tanh(self)	

5.1.6 numdifftools.nd_algopy module

Derivative(fun[, n, method, full_output])	Calculate n-th derivative with Algorithmic Differentiation method
Gradient(fun[, n, method, full_output])	Calculate Gradient with Algorithmic Differentiation method
Jacobian(fun[, n, method, full_output])	Calculate Jacobian with Algorithmic Differentiation method
Hessdiag(f[, method, full_output])	Calculate Hessian diagonal with Algorithmic Differentiation method
Hessian(f[, method, full_output])	Calculate Hessian with Algorithmic Differentiation method
directionaldiff(f, x0, vec, **options)	Return directional derivative of a function of n variables

numdifftools.nd_algopy.Derivative

class Derivative(*fun*, *n=1*, *method='forward'*, *full_output=False*)

Calculate n-th derivative with Algorithmic Differentiation method

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

n [int, optional] Order of the derivative.

method [string, optional {'forward', 'reverse'}] defines method used in the approximation

Returns

der [ndarray] array of derivatives

See also:

Gradient

Hessdiag

Hessian

Jacobian

Notes

Algorithmic differentiation is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.

References

Sebastian F. Walter and Lutz Lehmann 2013, "Algorithmic differentiation in Python with AlgoPy", in Journal of Computational Science, vol 4, no 5, pp 334 - 344, http://www.sciencedirect.com/science/article/pii/S1877750311001013

https://en.wikipedia.org/wiki/Automatic_differentiation

Examples

1'st and 2'nd derivative of exp(x), at x == 1

```
>>> import numpy as np
>>> import numdifftools.nd_algopy as nda
>>> fd = nda.Derivative(np.exp)               # 1'st derivative
>>> np.allclose(fd(1), 2.718281828459045)
True
>>> fd5 = nda.Derivative(np.exp, n=5)          # 5'th derivative
>>> np.allclose(fd5(1), 2.718281828459045)
True
```

1'st derivative of x^3+x^4, at x = [0,1]

```
>>> fun = lambda x: x**3 + x**4
>>> fd3 = nda.Derivative(fun)
>>> np.allclose(fd3([0,1]), [ 0.,  7.])
True
```

Methods

__call__(callable with the following parameters:) x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.	

__init__(*self*, *fun*, *n=1*, *method='forward'*, *full_output=False*)

Methods

__init__(self, fun[, n, method, full_output])	
computational_graph(self, x, *args, **kwds)	

numdifftools.nd_algopy.Gradient

class Gradient (*fun*, *n=1*, *method='forward'*, *full_output=False*)

Calculate Gradient with Algorithmic Differentiation method

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

method [string, optional {'forward', 'reverse'}] defines method used in the approximation

Returns

grad [array] gradient

See also:

Derivative

Jacobian

Hessdiag

Hessian

Notes

Algorithmic differentiation is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.

References

Sebastian F. Walter and Lutz Lehmann 2013, "Algorithmic differentiation in Python with AlgoPy", in Journal of Computational Science, vol 4, no 5, pp 334 - 344, http://www.sciencedirect.com/science/article/pii/S1877750311001013

https://en.wikipedia.org/wiki/Automatic_differentiation

Examples

```
>>> import numdifftools.nd_algopy as nda
>>> fun = lambda x: np.sum(x**2)
>>> df = nda.Gradient(fun, method='reverse')
>>> np.allclose(df([1,2,3]), [ 2.,  4.,  6.])
True
```

#At [x,y] = [1,1], compute the numerical gradient #of the function sin(x-y) + y*exp(x)

```
>>> sin = np.sin; exp = np.exp
>>> z = lambda xy: sin(xy[0]-xy[1]) + xy[1]*exp(xy[0])
>>> dz = nda.Gradient(z)
>>> grad2 = dz([1, 1])
>>> np.allclose(grad2, [ 3.71828183,  1.71828183])
True
```

#At the global minimizer (1,1) of the Rosenbrock function, #compute the gradient. It should be essentially zero.

```
>>> rosen = lambda x : (1-x[0])**2 + 105.*(x[1]-x[0]**2)**2
>>> rd = nda.Gradient(rosen)
>>> grad3 = rd([1,1])
>>> np.allclose(grad3, [ 0.,  0.])
True
```

Methods

__call__(callable with the following parameters:) x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.	

__init__(*self*, *fun*, *n=1*, *method='forward'*, *full_output=False*)

Methods

__init__(self, fun[, n, method, full_output])	
computational_graph(self, x, *args, **kwds)	

numdifftools.nd_algopy.Jacobian

class Jacobian(*fun*, *n=1*, *method='forward'*, *full_output=False*)

Calculate Jacobian with Algorithmic Differentiation method

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

method [string, optional {'forward', 'reverse'}] defines method used in the approximation

Returns

jacob [array] Jacobian

See also:

Derivative

Gradient

Hessdiag

Hessian

Notes

Algorithmic differentiation is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.

References

Sebastian F. Walter and Lutz Lehmann 2013, "Algorithmic differentiation in Python with AlgoPy", in Journal of Computational Science, vol 4, no 5, pp 334 - 344, http://www.sciencedirect.com/science/article/pii/S1877750311001013

https://en.wikipedia.org/wiki/Automatic_differentiation

Examples

```
>>> import numdifftools.nd_algopy as nda
```

#(nonlinear least squares)

```
>>> xdata = np.arange(0,1,0.1)
>>> ydata = 1+2*np.exp(0.75*xdata)
>>> fun = lambda c: (c[0]+c[1]*np.exp(c[2]*xdata) - ydata)**2
```

Jfun = nda.Jacobian(fun) # Todo: This does not work Jfun([1,2,0.75]).T # should be numerically zero array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])

```
>>> Jfun2 = nda.Jacobian(fun, method='reverse')
>>> res = Jfun2([1,2,0.75]).T # should be numerically zero
>>> np.allclose(res,
...                 [[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
...                  [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
...                  [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
True
```

```
>>> f2 = lambda x : x[0]*x[1]*x[2]**2
>>> Jfun2 = nda.Jacobian(f2)
>>> np.allclose(Jfun2([1., 2., 3.]), [[ 18., 9., 12.]])
True
```

```
>>> Jfun21 = nda.Jacobian(f2, method='reverse')
>>> np.allclose(Jfun21([1., 2., 3.]), [[ 18., 9., 12.]])
True
```

```
>>> def fun3(x):
...     n = int(np.prod(np.shape(x[0])))
...     out = nda.algopy.zeros((2, n), dtype=x)
...     out[0] = x[0]*x[1]*x[2]**2
...     out[1] = x[0]*x[1]*x[2]
...     return out
>>> Jfun3 = nda.Jacobian(fun3)
```

```
>>> np.allclose(Jfun3([1., 2., 3.]), [[[18., 9., 12.]], [[6., 3., 2.]]])
True
>>> np.allclose(Jfun3([4., 5., 6.]), [[[180., 144., 240.]],
...                                   [[30., 24., 20.]]])
True
>>> np.allclose(Jfun3(np.array([[1.,2.,3.], [4., 5., 6.]]).T),
...              [[[18.,    0.,    9.,    0.,   12.,    0.],
...                [0.,  180.,    0.,  144.,    0.,  240.]],
...               [[6.,    0.,    3.,    0.,    2.,    0.],
...                [0.,   30.,    0.,   24.,    0.,   20.]]])
True
```

Methods

__call__(callable with the following parameters:)	x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.

__init__(*self*, *fun*, *n=1*, *method='forward'*, *full_output=False*)

Methods

__init__(self, fun[, n, method, full_output])
computational_graph(self, x, *args, **kwds)

numdifftools.nd_algopy.Hessdiag

class Hessdiag(*f*, *method='forward'*, *full_output=False*)

Calculate Hessian diagonal with Algorithmic Differentiation method

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

method [string, optional {'forward', 'reverse'}] defines method used in the approximation

Returns

hessdiag [ndarray] Hessian diagonal array of partial second order derivatives.

See also:

Derivative

Gradient

Jacobian

Hessian

Notes

Algorithmic differentiation is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.

References

Sebastian F. Walter and Lutz Lehmann 2013, "Algorithmic differentiation in Python with AlgoPy", in Journal of Computational Science, vol 4, no 5, pp 334 - 344, http://www.sciencedirect.com/science/article/pii/S1877750311001013

https://en.wikipedia.org/wiki/Automatic_differentiation

Examples

```
>>> import numdifftools.nd_algopy as nda
```

Rosenbrock function, minimized at [1,1]

```
>>> rosen = lambda x : (1.-x[0])**2 + 105*(x[1]-x[0]**2)**2
>>> Hfun = nda.Hessdiag(rosen)
>>> h = Hfun([1, 1]) #  h =[ 842, 210]
>>> np.allclose(h, [ 842.,  210.])
True
```

cos(x-y), at (0,0)

```
>>> cos = np.cos
>>> fun = lambda xy : cos(xy[0]-xy[1])
>>> Hfun2 = nda.Hessdiag(fun)
>>> h2 = Hfun2([0, 0]) # h2 = [-1, -1]
>>> np.allclose(h2, [-1., -1.])
True
```

```
>>> Hfun3 = nda.Hessdiag(fun, method='reverse')
>>> h3 = Hfun3([0, 0]) # h2 = [-1, -1];
>>> np.allclose(h3, [-1., -1.])
True
```

Methods

__call__(callable with the following parameters:) x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.	

__init__(*self*, *f*, *method='forward'*, *full_output=False*)

Methods

__init__(self, f[, method, full_output])	
computational_graph(self, x, *args, **kwds)	

numdifftools.nd_algopy.Hessian

class Hessian(*f*, *method='forward'*, *full_output=False*)

Calculate Hessian with Algorithmic Differentiation method

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

method [string, optional {'forward', 'reverse'}] defines method used in the approximation

Returns

hess [ndarray] array of partial second derivatives, Hessian

See also:

Derivative

Gradient

Jacobian

Hessdiag

Notes

Algorithmic differentiation is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.

References

Sebastian F. Walter and Lutz Lehmann 2013, "Algorithmic differentiation in Python with AlgoPy", in Journal of Computational Science, vol 4, no 5, pp 334 - 344, http://www.sciencedirect.com/science/article/pii/S1877750311001013

https://en.wikipedia.org/wiki/Automatic_differentiation

Examples

```
>>> import numdifftools.nd_algopy as nda
```

Rosenbrock function, minimized at [1,1]

```
>>> rosen = lambda x : (1.-x[0])**2 + 105*(x[1]-x[0]**2)**2
>>> Hf = nda.Hessian(rosen)
>>> h = Hf([1, 1]) #  h =[ 842 -420; -420, 210];
>>> np.allclose(h, [[ 842., -420.],
...                 [-420.,  210.]])
True
```

cos(x-y), at (0,0)

```
>>> cos = np.cos
>>> fun = lambda xy : cos(xy[0]-xy[1])
>>> Hfun2 = nda.Hessian(fun)
>>> h2 = Hfun2([0, 0]) # h2 = [-1 1; 1 -1]
>>> np.allclose(h2, [[-1.,  1.],
...                  [ 1., -1.]])
True
```

```
>>> Hfun3 = nda.Hessian(fun, method='reverse')
>>> h3 = Hfun3([0, 0]) # h2 = [-1, 1; 1, -1];
>>> np.allclose(h3, [[-1.,  1.],
...                  [ 1., -1.]])
True
```

Methods

__call__(callable with the following parameters:) x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.	

__init__(*self*, *f*, *method='forward'*, *full_output=False*)

Methods

__init__(self, f[, method, full_output])	
computational_graph(self, x, *args, **kwds)	

numdifftools.nd_algopy.directionaldiff

directionaldiff(*f*, *x0*, *vec*, ***options*)

Return directional derivative of a function of n variables

Parameters

fun: callable analytical function to differentiate.

x0: array vector location at which to differentiate fun. If x0 is an nxm array, then fun is assumed to be a function of n*m variables.

vec: array vector defining the line along which to take the derivative. It should be the same size as x0, but need not be a vector of unit length.

****options:** optional arguments to pass on to Derivative.

Returns

dder: scalar estimate of the first derivative of fun in the specified direction.

See also:

Derivative

Gradient

Examples

At the global minimizer (1,1) of the Rosenbrock function, compute the directional derivative in the direction [1 2]

```
>>> import numpy as np
>>> import numdifftools as nd
>>> vec = np.r_[1, 2]
>>> rosen = lambda x: (1-x[0])**2 + 105*(x[1]-x[0]**2)**2
>>> dd, info = nd.directionaldiff(rosen, [1, 1], vec, full_output=True)
>>> np.allclose(dd, 0)
True
>>> np.abs(info.error_estimate)<1e-14
True
```

5.1.7 numdifftools.nd_scipy module

Gradient(fun[, step, method, order, bounds, ...])	Calculate Gradient with finite difference approximation
Jacobian(fun[, step, method, order, bounds, ...])	Calculate Jacobian with finite difference approximation

numdifftools.nd_scipy.Gradient

class Gradient (*fun*, *step=None*, *method='central'*, *order=2*, *bounds=(-inf, inf)*, *sparsity=None*)

Calculate Gradient with finite difference approximation

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

step [float, optional] Stepsize, if None, optimal stepsize is used, i.e., x * _EPS for method==‘complex‘ x * _EPS**(1/2) for method==‘forward‘ x * _EPS**(1/3) for method==‘central‘.

method [{‘central’, ‘complex’, ‘forward’}] defines the method used in the approximation.

See also:

Hessian, *Jacobian*

Examples

```
>>> import numpy as np
>>> import numdifftools.nd_scipy as nd
>>> fun = lambda x: np.sum(x**2)
>>> dfun = nd.Gradient(fun)
>>> np.allclose(dfun([1,2,3]), [ 2.,  4.,  6.])
True
```

At [x,y] = [1,1], compute the numerical gradient # of the function sin(x-y) + y*exp(x)

```
>>> sin = np.sin; exp = np.exp
>>> z = lambda xy: sin(xy[0]-xy[1]) + xy[1]*exp(xy[0])
>>> dz = nd.Gradient(z)
>>> grad2 = dz([1, 1])
>>> np.allclose(grad2, [ 3.71828183,  1.71828183])
True
```

At the global minimizer (1,1) of the Rosenbrock function, # compute the gradient. It should be essentially zero.

```
>>> rosen = lambda x : (1-x[0])**2 + 105.*(x[1]-x[0]**2)**2
>>> rd = nd.Gradient(rosen)
>>> grad3 = rd([1,1])
>>> np.allclose(grad3,[0, 0], atol=1e-7)
True
```

__init__ (*self*, *fun*, *step=None*, *method='central'*, *order=2*, *bounds=(-inf, inf)*, *sparsity=None*)

Methods

__init__(self, fun[, step, method, order, ...])	

numdifftools.nd_scipy.Jacobian

class Jacobian (*fun*, *step=None*, *method='central'*, *order=2*, *bounds=(-inf, inf)*, *sparsity=None*)
Calculate Jacobian with finite difference approximation

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

step [float, optional] Stepsize, if None, optimal stepsize is used, i.e., x * _EPS for method==‘complex‘ x * _EPS**(1/2) for method==‘forward‘ x * _EPS**(1/3) for method==‘central‘.

method [{‘central’, ‘complex’, ‘forward’}] defines the method used in the approximation.

Examples

```
>>> import numdifftools.nd_scipy as nd
```

#(nonlinear least squares)

```
>>> xdata = np.arange(0,1,0.1)
>>> ydata = 1+2*np.exp(0.75*xdata)
>>> fun = lambda c: (c[0]+c[1]*np.exp(c[2]*xdata) - ydata)**2
>>> np.allclose(fun([1, 2, 0.75]).shape, (10,))
True
>>> dfun = nd.Jacobian(fun)
>>> np.allclose(dfun([1, 2, 0.75]), np.zeros((10,3)))
True
```

```
>>> fun2 = lambda x : x[0]*x[1]*x[2]**2
>>> dfun2 = nd.Jacobian(fun2)
>>> np.allclose(dfun2([1.,2.,3.]), [[18., 9., 12.]])
True
```

```
>>> fun3 = lambda x : np.vstack((x[0]*x[1]*x[2]**2, x[0]*x[1]*x[2]))
```

TODO: The following does not work: der3 = nd.Jacobian(fun3)([1., 2., 3.]) np.allclose(der3, . . . [[18., 9., 12.], [6., 3., 2.]]) True np.allclose(nd.Jacobian(fun3)([4., 5., 6.]), . . . [[180., 144., 240.], [30., 24., 20.]]) True

np.allclose(nd.Jacobian(fun3)(np.array([[1.,2.,3.], [4., 5., 6.]]).T), . . . [[[18., 180.], . . . [9., 144.], . . . [12., 240.]], . . . [[6., 30.], . . . [3., 24.], . . . [2., 20.]]]) True

__init__ (*self*, *fun*, *step=None*, *method='central'*, *order=2*, *bounds=(-inf, inf)*, *sparsity=None*)

Methods

__init__(self, fun[, step, method, order, . . .])	

5.1.8 numdifftools.nd_statsmodels module

Hessian(fun[, step, method, order])	Calculate Hessian with finite difference approximation
Jacobian(fun[, step, method, order])	Calculate Jacobian with finite difference approximation

numdifftools.nd_statsmodels.Hessian

class Hessian(*fun*, *step=None*, *method='central'*, *order=None*)
Calculate Hessian with finite difference approximation

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

step [float, optional] Stepsize, if None, optimal stepsize is used, i.e., x * _EPS**(1/3) for method==‘forward‘, *complex* or *central2* x * _EPS**(1/4) for method==‘central‘.

method [{‘central’, ‘complex’, ‘forward’, ‘backward’}] defines the method used in the approximation.

See also:

Jacobian, *Gradient*

Examples

```
>>> import numpy as np
>>> import numdifftools.nd_statsmodels as nd
```

Rosenbrock function, minimized at [1,1]

```
>>> rosen = lambda x : (1.-x[0])**2 + 105*(x[1]-x[0]**2)**2
>>> Hfun = nd.Hessian(rosen)
>>> h = Hfun([1, 1])
>>> np.allclose(h, [[ 842., -420.], [-420.,  210.]])
True
```

cos(x-y), at (0,0)

```
>>> cos = np.cos
>>> fun = lambda xy : cos(xy[0]-xy[1])
>>> Hfun2 = nd.Hessian(fun)
>>> h2 = Hfun2([0, 0])
>>> np.allclose(h2, [[-1.,  1.], [ 1., -1.]])
True
```

__init__(*self*, *fun*, *step=None*, *method='central'*, *order=None*)

Methods

__init__(self, fun[, step, method, order])

numdifftools.nd_statsmodels.Jacobian

class Jacobian(*fun*, *step=None*, *method='central'*, *order=None*)
Calculate Jacobian with finite difference approximation

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

step [float, optional] Stepsize, if None, optimal stepsize is used, i.e., x * _EPS for method==‘complex‘ x * _EPS**(1/2) for method==‘forward‘ x * _EPS**(1/3) for method==‘central‘.

method [{'central', 'complex', 'forward', 'backward'}] defines the method used in the approximation.

Examples

```
>>> import numdifftools.nd_statsmodels as nd
```

#(nonlinear least squares)

```
>>> xdata = np.arange(0,1,0.1)
>>> ydata = 1+2*np.exp(0.75*xdata)
>>> fun = lambda c: (c[0]+c[1]*np.exp(c[2]*xdata) - ydata)**2
>>> np.allclose(fun([1, 2, 0.75]).shape, (10,))
True
>>> dfun = nd.Jacobian(fun)
>>> np.allclose(dfun([1, 2, 0.75]), np.zeros((10,3)))
True
```

```
>>> fun2 = lambda x : x[0]*x[1]*x[2]**2
>>> dfun2 = nd.Jacobian(fun2)
>>> np.allclose(dfun2([1.,2.,3.]), [[18., 9., 12.]])
True
```

```
>>> fun3 = lambda x : np.vstack((x[0]*x[1]*x[2]**2, x[0]*x[1]*x[2]))
>>> np.allclose(nd.Jacobian(fun3)([1., 2., 3.]),
...             [[18., 9., 12.], [6., 3., 2.]])
True
>>> np.allclose(nd.Jacobian(fun3)([4., 5., 6.]),
...             [[180., 144., 240.], [30., 24., 20.]])
True
```

```
>>> np.allclose(nd.Jacobian(fun3)(np.array([[1.,2.,3.], [4., 5., 6.]]).T),
...             [[[  18.,  180.],
...               [   9.,  144.],
...               [  12.,  240.]],
...              [[   6.,   30.],
...               [   3.,   24.],
...               [   2.,   20.]]])
True
```

__init__(*self*, *fun*, *step=None*, *method='central'*, *order=None*)

Methods

__init__(self, fun[, step, method, order])

5.2 Numdifftools package details

5.2.1 Subpackages

numdifftools.tests package

Submodules

numdifftools.tests.conftest module

Dummy conftest.py for numdifftools.

If you don't know what this is for, just leave it empty. Read more about conftest.py under: https://pytest.org/latest/plugins.html

numdifftools.tests.test_extrapolation module

class TestExtrapolation
Bases: object

setup_method(*self*)

test_dea3_on_trapz_sin(*self*)

test_dea_on_trapz_sin(*self*)

test_epsal(*self*)

test_richardson(*self*)

class TestRichardson
Bases: object

setup_method(*self*)

test_order_step_combinations(*self*)

numdifftools.tests.test_hessian module

numdifftools.tests.test_limits module

Created on 28. aug. 2015

@author: pab

class TestCStepGenerator
Bases: object

static test_default_base_step()

static test_default_generator()

static test_fixed_base_step()

class TestLimit
Bases: object

test_derivative_of_cos(*self*)

test_difficult_limit(*self*)

test_residue_1_div_1_minus_exp_x(*self*)

test_sinx_div_x(*self*)

class TestResidue
Bases: object

test_residue_1_div_1_minus_exp_x(*self*)

test_residue_1_div_sin_x2(*self*)

numdifftools.tests.test_multicomplex module

Created on 22. apr. 2015

@author: pab

class TestBicomplex
Bases: object

static test_add()

static test_arccos()

static test_arcsin()

static test_arg_c()

static test_assign()

test_conjugate(*self*)

static test_cos()

static test_der_abs()

static test_der_arccos()

static test_der_arccosh()

static test_der_arctan()

static test_der_cos()

static test_der_log()

static test_division()

static test_dot()

static test_eq()

test_flat(*self*)

static test_ge()

static test_gt()

test_init(*self*)

static test_le()

static test_lt()

static test_multiplication()

test_neg(*self*)

test_norm(*self*)

static test_pow()

test_repr(*self*)

static test_rpow()

static test_rsub()

test_shape(*self*)

static test_sub()

static test_subsref()

class TestDerivative
Bases: object

static test_all_first_derivatives()

static test_all_second_derivatives()

numdifftools.tests.test_nd_algopy module

class TestDerivative
Bases: object

static test_derivative_cube()
Test for Issue 7

static test_derivative_exp()

static test_derivative_on_log()

test_derivative_on_sinh(*self*)

static test_derivative_sin()

static test_directional_diff()

static test_fun_with_additional_parameters()
Test for issue #9

static test_high_order_derivative_cos()

class TestGradient
Bases: object

static test_on_scalar_function()

class TestHessdiag
Bases: object

static test_forward()

static test_reverse()

class TestHessian
Bases: object

static test_hessian_cos_x_y__at_0_0()

test_run_hamiltonian(*self*)

class TestJacobian
Bases: object

static test_issue_25()

static test_on_matrix_valued_function()

static test_on_scalar_function()

test_on_vector_valued_function(*self*)

static test_scalar_to_vector()

numdifftools.tests.test_numdifftools module

Test functions for numdifftools module

class TestDerivative
Bases: object

test_backward_derivative_on_sinh(*self*)

test_central_and_forward_derivative_on_log(*self*)

static test_default_scale()

static test_derivative_cube()
Test for Issue 7

static test_derivative_exp()

static test_derivative_of_cos_x()

static test_derivative_sin()

static test_derivative_with_step_options()

static test_directional_diff()

static test_fun_with_additional_parameters()
Test for issue #9

static test_high_order_derivative_cos()

test_infinite_functions(*self*)

class TestGradient
Bases: object

static test_directional_diff()

static test_gradient()

class TestHessdiag
Bases: object

test_complex(*self*)

test_default_step(*self*)

test_fixed_step(*self*)

class TestHessian
Bases: object

test_complex_hessian_issue_35(*self*)

static test_hessian_cos_x_y_at_0_0()

test_run_hamiltonian(*self*)

class TestJacobian
Bases: object

static test_issue_25()

static test_issue_27a()
Test for memory-error

static test_issue_27b()

static test_on_matrix_valued_function()

static test_on_scalar_function()

static test_on_vector_valued_function()

static test_scalar_to_vector()

class TestRichardson
Bases: object

static test_central_forward_backward()

static test_complex()

numdifftools.tests.test_numdifftools_docstrings module

Module contents

5.2.2 Submodules

Core module

numerical differentiation functions:

Derivative, Gradient, Jacobian, and Hessian

Author: Per A. Brodtkorb Created: 01.08.2008 Copyright: (c) pab 2008 Licence: New BSD

dea3 (*v0*, *v1*, *v2*, *symmetric=False*)

Extrapolate a slowly convergent sequence

Parameters

v0, v1, v2 [array-like] 3 values of a convergent sequence to extrapolate

Returns

result [array-like] extrapolated value

abserr [array-like] absolute error estimate

See also:

dea

Notes

DEA3 attempts to extrapolate nonlinearly to a better estimate of the sequence's limiting value, thus improving the rate of convergence. The routine is based on the epsilon algorithm of P. Wynn, see *[1]*.

References

[1]

Examples

integrate sin(x) from 0 to pi/2

```
>>> import numpy as np
>>> import numdifftools as nd
>>> Ei= np.zeros(3)
>>> linfun = lambda i : np.linspace(0, np.pi/2., 2**(i+5)+1)
>>> for k in np.arange(3):
...     x = linfun(k)
...     Ei[k] = np.trapz(np.sin(x),x)
>>> [En, err] = nd.dea3(Ei[0], Ei[1], Ei[2])
>>> truErr = np.abs(En-1.)
>>> np.all(truErr < err)
True
>>> np.allclose(En, 1)
True
>>> np.all(np.abs(Ei-1)<1e-3)
True
```

class Derivative(*fun*, *step=None*, *method='central'*, *order=2*, *n=1*, *full_output=False*, ***step_options*)

Bases: numdifftools.limits._Limit

Calculate n-th derivative with finite difference approximation

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

step [float, array-like or StepGenerator object, optional] Defines the spacing used in the approximation. Default is MinStepGenerator(base_step=step, step_ratio=None,

num_extrap=0, **step_options)

if step or method in in ['complex', 'multicomplex'], otherwise

MaxStepGenerator(step_ratio=None, num_extrap=14, **step_options)

The results are extrapolated if the StepGenerator generate more than 3 steps.

method [{'central', 'complex', 'multicomplex', 'forward', 'backward'}] defines the method used in the approximation

order [int, optional] defines the order of the error term in the Taylor approximation used. For 'central' and 'complex' methods, it must be an even number.

n [int, optional] Order of the derivative.

full_output [bool, optional] If *full_output* is False, only the derivative is returned. If *full_output* is True, then (der, r) is returned *der* is the derivative, and *r* is a Results object.

****step_options:** options to pass on to the XXXStepGenerator used.

Returns

der [ndarray] array of derivatives

See also:

Gradient

Hessian

Notes

Complex methods are usually the most accurate provided the function to differentiate is analytic. The complex-step methods also requires fewer steps than the other methods and can work very close to the support of a function. The complex-step derivative has truncation error O(steps**2) for *n=1* and O(steps**4) for *n* larger, so truncation error can be eliminated by choosing steps to be very small. Especially the first order complex-step derivative avoids the problem of round-off error with small steps because there is no subtraction. However, this method fails if fun(x) does not support complex numbers or involves non-analytic functions such as e.g.: abs, max, min. Central difference methods are almost as accurate and has no restriction on type of function. For this reason the 'central' method is the default method, but sometimes one can only allow evaluation in forward or backward direction.

For all methods one should be careful in decreasing the step size too much due to round-off errors.

Higher order approximation methods will generally be more accurate, but may also suffer more from numerical problems. First order methods is usually not recommended.

References

Ridout, M.S. (2009) Statistical applications of the complex-step method of numerical differentiation. The American Statistician, 63, 66-74

K.-L. Lai, J.L. Crassidis, Y. Cheng, J. Kim (2005), New complex step derivative approximations with application to second-order kalman filtering, AIAA Guidance, Navigation and Control Conference, San Francisco, California, August 2005, AIAA-2005-5944.

Lyness, J. M., Moler, C. B. (1966). Vandermonde Systems and Numerical Differentiation. *Numerische Mathematik*.

Lyness, J. M., Moler, C. B. (1969). Generalized Romberg Methods for Integrals of Derivatives. *Numerische Mathematik*.

Examples

```
>>> import numpy as np
>>> import numdifftools as nd
```

1'st derivative of exp(x), at x == 1

```
>>> fd = nd.Derivative(np.exp)
>>> np.allclose(fd(1), 2.71828183)
True
```

```
>>> d2 = fd([1, 2])
>>> np.allclose(d2, [ 2.71828183,  7.3890561 ])
True
```

```
>>> def f(x):
...     return x**3 + x**2
```

```
>>> df = nd.Derivative(f)
>>> np.allclose(df(1), 5)
True
>>> ddf = nd.Derivative(f, n=2)
>>> np.allclose(ddf(1), 8)
True
```

Methods

__call__(callable with the following parameters:) x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.	

class info

Bases: `tuple`

count()

error_estimate

Alias for field number 0

final_step

Alias for field number 1

index

Alias for field number 2

n

!! processed by numpydoc !!

set_richardson_rule(*self*, *step_ratio*, *num_terms=2*)

step
!! processed by numpydoc !!

class Jacobian(*fun*, *step=None*, *method='central'*, *order=2*, *n=1*, *full_output=False*, ***step_options*)
Bases: *numdifftools.core.Derivative*

Calculate Jacobian with finite difference approximation

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

step [float, array-like or StepGenerator object, optional] Defines the spacing used in the approximation. Default is MinStepGenerator(base_step=step, step_ratio=None,

num_extrap=0, **step_options)

if step or method in in ['complex', 'multicomplex'], otherwise

MaxStepGenerator(step_ratio=None, num_extrap=14, **step_options)

The results are extrapolated if the StepGenerator generate more than 3 steps.

method [{'central', 'complex', 'multicomplex', 'forward', 'backward'}] defines the method used in the approximation

order [int, optional] defines the order of the error term in the Taylor approximation used. For 'central' and 'complex' methods, it must be an even number.

full_output [bool, optional] If *full_output* is False, only the derivative is returned. If *full_output* is True, then (der, r) is returned *der* is the derivative, and *r* is a Results object.

****step_options:** options to pass on to the XXXStepGenerator used.

Returns

jacob [array] Jacobian

See also:

Derivative*, *Hessian*, *Gradient

Notes

Complex methods are usually the most accurate provided the function to differentiate is analytic. The complex-step methods also requires fewer steps than the other methods and can work very close to the support of a function. The complex-step derivative has truncation error O(steps**2) for *n=1* and O(steps**4) for *n* larger, so truncation error can be eliminated by choosing steps to be very small. Especially the first order complex-step derivative avoids the problem of round-off error with small steps because there is no subtraction. However, this method fails if fun(x) does not support complex numbers or involves non-analytic functions such as e.g.: abs, max, min. Central difference methods are almost as accurate and has no restriction on type of function. For this reason the 'central' method is the default method, but sometimes one can only allow evaluation in forward or backward direction.

For all methods one should be careful in decreasing the step size too much due to round-off errors.

Higher order approximation methods will generally be more accurate, but may also suffer more from numerical problems. First order methods is usually not recommended.

If fun returns a 1d array, it returns a Jacobian. If a 2d array is returned by fun (e.g., with a value for each observation), it returns a 3d array with the Jacobian of each observation with shape xk x nobs x xk. I.e., the Jacobian of the first observation would be [:, 0, :]

References

Ridout, M.S. (2009) Statistical applications of the complex-step method of numerical differentiation. The American Statistician, 63, 66-74

K.-L. Lai, J.L. Crassidis, Y. Cheng, J. Kim (2005), New complex step derivative approximations with application to second-order kalman filtering, AIAA Guidance, Navigation and Control Conference, San Francisco, California, August 2005, AIAA-2005-5944.

Lyness, J. M., Moler, C. B. (1966). Vandermonde Systems and Numerical Differentiation. *Numerische Mathematik*.

Lyness, J. M., Moler, C. B. (1969). Generalized Romberg Methods for Integrals of Derivatives. *Numerische Mathematik*.

Examples

```
>>> import numdifftools as nd
```

#(nonlinear least squares)

```
>>> xdata = np.arange(0,1,0.1)
>>> ydata = 1+2*np.exp(0.75*xdata)
>>> fun = lambda c: (c[0]+c[1]*np.exp(c[2]*xdata) - ydata)**2
>>> np.allclose(fun([1, 2, 0.75]).shape,  (10,))
True
```

```
>>> jfun = nd.Jacobian(fun)
>>> val = jfun([1, 2, 0.75])
>>> np.allclose(val, np.zeros((10,3)))
True
```

```
>>> fun2 = lambda x : x[0]*x[1]*x[2]**2
>>> jfun2 = nd.Jacobian(fun2)
>>> np.allclose(jfun2([1.,2.,3.]), [[18., 9., 12.]])
True
```

```
>>> fun3 = lambda x : np.vstack((x[0]*x[1]*x[2]**2, x[0]*x[1]*x[2]))
>>> jfun3 = nd.Jacobian(fun3)
>>> np.allclose(jfun3([1., 2., 3.]), [[18., 9., 12.], [6., 3., 2.]])
True
>>> np.allclose(jfun3([4., 5., 6.]), [[180., 144., 240.], [30., 24., 20.]])
True
>>> np.allclose(jfun3(np.array([[1.,2.,3.]]).T), [[ 18.,   9.,  12.],
...                                               [ 6.,   3.,   2.]])
True
```

Methods

__call__(callable with the following parameters:) x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.	

class info

Bases: `tuple`

count()

error_estimate
Alias for field number 0

final_step
Alias for field number 1

index
Alias for field number 2

n
!! processed by numpydoc !!

set_richardson_rule(*self*, *step_ratio*, *num_terms=2*)

step
!! processed by numpydoc !!

class Gradient(*fun*, *step=None*, *method='central'*, *order=2*, *n=1*, *full_output=False*, ***step_options*)

Bases: numdifftools.core.Jacobian

Calculate Gradient with finite difference approximation

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

step [float, array-like or StepGenerator object, optional] Defines the spacing used in the approximation. Default is MinStepGenerator(base_step=step, step_ratio=None,

num_extrap=0, **step_options)

if step or method in in ['complex', 'multicomplex'], otherwise

MaxStepGenerator(step_ratio=None, num_extrap=14, **step_options)

The results are extrapolated if the StepGenerator generate more than 3 steps.

method [{'central', 'complex', 'multicomplex', 'forward', 'backward'}] defines the method used in the approximation

order [int, optional] defines the order of the error term in the Taylor approximation used. For 'central' and 'complex' methods, it must be an even number.

full_output [bool, optional] If *full_output* is False, only the derivative is returned. If *full_output* is True, then (der, r) is returned *der* is the derivative, and *r* is a Results object.

****step_options:** options to pass on to the XXXStepGenerator used.

Returns

grad [array] gradient

See also:

Derivative, ***Hessian***, ***Jacobian***

Notes

Complex methods are usually the most accurate provided the function to differentiate is analytic. The complex-step methods also requires fewer steps than the other methods and can work very close to the support of a function. The complex-step derivative has truncation error O(steps**2) for *n=1* and O(steps**4) for *n* larger, so truncation error can be eliminated by choosing steps to be very small. Especially the first order complex-step derivative avoids the problem of round-off error with small steps because there is no subtraction. However, this method fails if fun(x) does not support complex numbers or involves non-analytic

functions such as e.g.: abs, max, min. Central difference methods are almost as accurate and has no restriction on type of function. For this reason the 'central' method is the default method, but sometimes one can only allow evaluation in forward or backward direction.

For all methods one should be careful in decreasing the step size too much due to round-off errors.

Higher order approximation methods will generally be more accurate, but may also suffer more from numerical problems. First order methods is usually not recommended.

If x0 is an n x m array, then fun is assumed to be a function of n * m variables.

References

Ridout, M.S. (2009) Statistical applications of the complex-step method of numerical differentiation. The American Statistician, 63, 66-74

K.-L. Lai, J.L. Crassidis, Y. Cheng, J. Kim (2005), New complex step derivative approximations with application to second-order kalman filtering, AIAA Guidance, Navigation and Control Conference, San Francisco, California, August 2005, AIAA-2005-5944.

Lyness, J. M., Moler, C. B. (1966). Vandermonde Systems and Numerical Differentiation. *Numerische Mathematik*.

Lyness, J. M., Moler, C. B. (1969). Generalized Romberg Methods for Integrals of Derivatives. *Numerische Mathematik*.

Examples

```
>>> import numpy as np
>>> import numdifftools as nd
>>> fun = lambda x: np.sum(x**2)
>>> dfun = nd.Gradient(fun)
>>> np.allclose(dfun([1,2,3]), [ 2.,  4.,  6.])
True
```

At [x,y] = [1,1], compute the numerical gradient # of the function sin(x-y) + y*exp(x)

```
>>> sin = np.sin; exp = np.exp
>>> x, y = 1, 1
>>> z = lambda xy: sin(xy[0]-xy[1]) + xy[1]*exp(xy[0])
>>> dz = nd.Gradient(z)
>>> dz_dx, dz_dy = dz([x, y])
>>> np.allclose([dz_dx, dz_dy],
...             [ 3.7182818284590686, 1.7182818284590162])
True
```

At the global minimizer (1,1) of the Rosenbrock function, # compute the gradient. It should be essentially zero.

```
>>> rosen = lambda x : (1-x[0])**2 + 105.*(x[1]-x[0]**2)**2
>>> grad_rosen = nd.Gradient(rosen)
>>> df_dx, df_dy = grad_rosen([x, y])
>>> np.allclose([df_dx, df_dy], [0, 0])
True
```

Methods

__call__(callable with the following parameters:) x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.	

class info
Bases: `tuple`

count()

error_estimate
Alias for field number 0

final_step
Alias for field number 1

index
Alias for field number 2

n
!! processed by numpydoc !!

set_richardson_rule(*self*, *step_ratio*, *num_terms=2*)

step
!! processed by numpydoc !!

class Hessian(*f*, *step=None*, *method='central'*, *order=2*, *full_output=False*, ***step_options*)
Bases: *numdifftools.core.Hessdiag*

Calculate Hessian with finite difference approximation

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

step [float, array-like or StepGenerator object, optional] Defines the spacing used in the approximation. Default is MinStepGenerator(base_step=step, step_ratio=None,

num_extrap=0, **step_options)

if step or method in in ['complex', 'multicomplex'], otherwise

MaxStepGenerator(step_ratio=None, num_extrap=14, **step_options)

The results are extrapolated if the StepGenerator generate more than 3 steps.

method [{'central', 'complex', 'multicomplex', 'forward', 'backward'}] defines the method used in the approximation

full_output [bool, optional] If *full_output* is False, only the derivative is returned. If *full_output* is True, then (der, r) is returned *der* is the derivative, and *r* is a Results object.

****step_options:** options to pass on to the XXXStepGenerator used.

Returns

hess [ndarray] array of partial second derivatives, Hessian

See also:

Derivative, *Hessian*

Notes

Complex methods are usually the most accurate provided the function to differentiate is analytic. The complex-step methods also requires fewer steps than the other methods and can work very close to the support of a function. The complex-step derivative has truncation error O(steps**2) for *n=1* and O(steps**4) for *n* larger, so truncation error can be eliminated by choosing steps to be very small. Especially the first order complex-step derivative avoids the problem of round-off error with small steps because there is no subtraction. However, this method fails if fun(x) does not support complex numbers or involves non-analytic functions such as e.g.: abs, max, min. Central difference methods are almost as accurate and has no restriction on type of function. For this reason the 'central' method is the default method, but sometimes one can only allow evaluation in forward or backward direction.

For all methods one should be careful in decreasing the step size too much due to round-off errors.

Computes the Hessian according to method as: 'forward' (5.1), 'central' (5.2) and 'complex' (5.3):

$$((f(x + d_j e_j + d_k e_k) - f(x + d_j e_j)))/(d_j d_k) \tag{5.4}$$

$$((f(x + d_j e_j + d_k e_k) - f(x + d_j e_j - d_k e_k)) - (f(x - d_j e_j + d_k e_k) - f(x - d_j e_j - d_k e_k))/(4 d_j d_k) \tag{5.5}$$

$$imag(f(x + i d_j e_j + d_k e_k) - f(x + i d_j e_j - d_k e_k))/(2 d_j d_k) \tag{5.6}$$

where e_j is a vector with element j is one and the rest are zero and d_j is a scalar spacing $steps_j$.

References

Ridout, M.S. (2009) Statistical applications of the complex-step method of numerical differentiation. The American Statistician, 63, 66-74

K.-L. Lai, J.L. Crassidis, Y. Cheng, J. Kim (2005), New complex step derivative approximations with application to second-order kalman filtering, AIAA Guidance, Navigation and Control Conference, San Francisco, California, August 2005, AIAA-2005-5944.

Lyness, J. M., Moler, C. B. (1966). Vandermonde Systems and Numerical Differentiation. *Numerische Mathematik*.

Lyness, J. M., Moler, C. B. (1969). Generalized Romberg Methods for Integrals of Derivatives. *Numerische Mathematik*.

Examples

```
>>> import numpy as np
>>> import numdifftools as nd
```

Rosenbrock function, minimized at [1,1]

```
>>> rosen = lambda x : (1.-x[0])**2 + 105*(x[1]-x[0]**2)**2
>>> Hfun = nd.Hessian(rosen)
>>> h = Hfun([1, 1])
>>> h
array([[ 842., -420.],
       [-420.,  210.]])
```

cos(x-y), at (0,0)

```
>>> cos = np.cos
>>> fun = lambda xy : cos(xy[0]-xy[1])
>>> Hfun2 = nd.Hessian(fun)
>>> h2 = Hfun2([0, 0])
```

```
>>> h2
array([[-1.,  1.],
       [ 1., -1.]])
```

Methods

__call__(callable with the following parameters:) x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.	

class info
Bases: `tuple`

count()

error_estimate
Alias for field number 0

final_step
Alias for field number 1

index
Alias for field number 2

n
!! processed by numpydoc !!

order
!! processed by numpydoc !!

set_richardson_rule(*self*, *step_ratio*, *num_terms=2*)

step
!! processed by numpydoc !!

class Hessdiag(*f*, *step=None*, *method='central'*, *order=2*, *full_output=False*, ***step_options*)
Bases: *numdifftools.core.Derivative*

Calculate Hessian diagonal with finite difference approximation

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

step [float, array-like or StepGenerator object, optional] Defines the spacing used in the approximation. Default is MinStepGenerator(base_step=step, step_ratio=None,

num_extrap=0, **step_options)

if step or method in in ['complex', 'multicomplex'], otherwise

MaxStepGenerator(step_ratio=None, num_extrap=14, **step_options)

The results are extrapolated if the StepGenerator generate more than 3 steps.

method [{'central', 'complex', 'multicomplex', 'forward', 'backward'}] defines the method used in the approximationorder : int, optional defines the order of the error term in the Taylor approximation used. For 'central' and 'complex' methods, it must be an even number.

full_output [bool, optional] If *full_output* is False, only the derivative is returned. If *full_output* is True, then (der, r) is returned *der* is the derivative, and *r* is a Results object.

****step_options:** options to pass on to the XXXStepGenerator used.

Returns

hessdiag [array] hessian diagonal

See also:

Derivative, *Hessian*, *Jacobian*, *Gradient*

Notes

Complex methods are usually the most accurate provided the function to differentiate is analytic. The complex-step methods also requires fewer steps than the other methods and can work very close to the support of a function. The complex-step derivative has truncation error O(steps**2) for *n=1* and O(steps**4) for *n* larger, so truncation error can be eliminated by choosing steps to be very small. Especially the first order complex-step derivative avoids the problem of round-off error with small steps because there is no subtraction. However, this method fails if fun(x) does not support complex numbers or involves non-analytic functions such as e.g.: abs, max, min. Central difference methods are almost as accurate and has no restriction on type of function. For this reason the 'central' method is the default method, but sometimes one can only allow evaluation in forward or backward direction.

For all methods one should be careful in decreasing the step size too much due to round-off errors.

Higher order approximation methods will generally be more accurate, but may also suffer more from numerical problems. First order methods is usually not recommended.

References

Ridout, M.S. (2009) Statistical applications of the complex-step method of numerical differentiation. The American Statistician, 63, 66-74

K.-L. Lai, J.L. Crassidis, Y. Cheng, J. Kim (2005), New complex step derivative approximations with application to second-order kalman filtering, AIAA Guidance, Navigation and Control Conference, San Francisco, California, August 2005, AIAA-2005-5944.

Lyness, J. M., Moler, C. B. (1966). Vandermonde Systems and Numerical Differentiation. *Numerische Mathematik*.

Lyness, J. M., Moler, C. B. (1969). Generalized Romberg Methods for Integrals of Derivatives. *Numerische Mathematik*.

Examples

```
>>> import numpy as np
>>> import numdifftools as nd
>>> fun = lambda x : x[0] + x[1]**2 + x[2]**3
>>> Hfun = nd.Hessdiag(fun, full_output=True)
>>> hd, info = Hfun([1,2,3])
>>> np.allclose(hd, [0.,   2.,  18.])
True
```

```
>>> np.all(info.error_estimate < 1e-11)
True
```

Methods

__call__(callable with the following parameters:) x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.	

class info
Bases: `tuple`

count()

error_estimate
Alias for field number 0

final_step
Alias for field number 1

index
Alias for field number 2

n
!! processed by numpydoc !!

set_richardson_rule(*self*, *step_ratio*, *num_terms=2*)

step
!! processed by numpydoc !!

class MinStepGenerator(*base_step=None*, *step_ratio=2.0*, *num_steps=None*, *step_nom=None*, *offset=0*, *num_extrap=0*, *use_exact_steps=True*, *check_num_steps=True*, *scale=None*)
Bases: `object`

Generates a sequence of steps

where steps = step_nom * base_step * step_ratio ** (i + offset)

for i = num_steps-1,... 1, 0.

Parameters

base_step [float, array-like, optional] Defines the minimum step, if None, the value is set to EPS**(1/scale)

step_ratio [real scalar, optional, default 2] Ratio between sequential steps generated. Note: Ratio > 1 If None then step_ratio is 2 for n=1 otherwise step_ratio is 1.6

num_steps [scalar integer, optional, default min_num_steps + num_extrap] defines number of steps generated. It should be larger than min_num_steps = (n + order - 1) / fact where fact is 1, 2 or 4 depending on differentiation method used.

step_nom [default maximum(log(1+|x|), 1)] Nominal step where x is supplied at runtime through the __call__ method.

offset [real scalar, optional, default 0] offset to the base step

num_extrap [scalar integer, default 0] number of points used for extrapolation

check_num_steps [boolean, default True] If True make sure num_steps is larger than the minimum required steps.

use_exact_steps [boolean, default True] If true make sure exact steps are generated

scale [real scalar, optional] scale used in base step. If not None it will override the default computed with the default_scale function.

base_step
!! processed by numpydoc !!

min_num_steps
!! processed by numpydoc !!

num_steps
!! processed by numpydoc !!

scale
!! processed by numpydoc !!

step_generator_function(*self*, *x*, *method='forward'*, *n=1*, *order=2*)

step_nom
!! processed by numpydoc !!

step_ratio
!! processed by numpydoc !!

class MaxStepGenerator(*base_step=2.0*, *step_ratio=2.0*, *num_steps=15*, *step_nom=None*, *offset=0*, *num_extrap=0*, *use_exact_steps=False*, *check_num_steps=True*, *scale=500*)

Bases: *numdifftools.step_generators.MinStepGenerator*

Generates a sequence of steps

where steps = step_nom * base_step * step_ratio ** (-i + offset)

for i = 0, 1, . . . , num_steps-1.

Parameters

base_step [float, array-like, default 2.0] Defines the maximum step, if None, the value is set to EPS**(1/scale)

step_ratio [real scalar, optional, default 2] Ratio between sequential steps generated. Note: Ratio > 1 If None then step_ratio is 2 for n=1 otherwise step_ratio is 1.6

num_steps [scalar integer, optional, default min_num_steps + num_extrap] defines number of steps generated. It should be larger than min_num_steps = (n + order - 1) / fact where fact is 1, 2 or 4 depending on differentiation method used.

step_nom [default maximum(log(1+|x|), 1)] Nominal step where x is supplied at runtime through the __call__ method.

offset [real scalar, optional, default 0] offset to the base step

num_extrap [scalar integer, default 0] number of points used for extrapolation

check_num_steps [boolean, default True] If True make sure num_steps is larger than the minimum required steps.

use_exact_steps [boolean, default True] If true make sure exact steps are generated

scale [real scalar, default 500] scale used in base step.

base_step
!! processed by numpydoc !!

min_num_steps
!! processed by numpydoc !!

num_steps
!! processed by numpydoc !!

scale
!! processed by numpydoc !!

step_generator_function(*self*, *x*, *method='forward'*, *n=1*, *order=2*)

step_nom
!! processed by numpydoc !!

step_ratio
!! processed by numpydoc !!

class Richardson(*step_ratio=2.0*, *step=1*, *order=1*, *num_terms=2*)
Bases: object

Extrapolates as sequence with Richardsons method

Notes

Suppose you have series expansion that goes like this

L = f(h) + a0 * h^p_0 + a1 * h^p_1+ a2 * h^p_2 + ...

where p_i = order + step * i and f(h) -> L as h -> 0, but f(0) != L.

If we evaluate the right hand side for different stepsizes h we can fit a polynomial to that sequence of approximations. This is exactly what this class does.

Examples

```
>>> import numpy as np
>>> import numdifftools as nd
>>> n = 3
>>> Ei = np.zeros((n,1))
>>> h = np.zeros((n,1))
>>> linfun = lambda i : np.linspace(0, np.pi/2., 2**(i+5)+1)
>>> for k in np.arange(n):
...     x = linfun(k)
...     h[k] = x[1]
...     Ei[k] = np.trapz(np.sin(x),x)
>>> En, err, step = nd.Richardson(step=1, order=1)(Ei, h)
>>> truErr = np.abs(En-1.)
>>> np.all(truErr < err)
True
>>> np.all(np.abs(Ei-1)<1e-3)
True
>>> np.allclose(En, 1)
True
```

extrapolate(*self*, *sequence*, *steps*)

rule(*self*, *sequence_length=None*)

directionaldiff(*f*, *x0*, *vec*, ***options*)
Return directional derivative of a function of n variables

Parameters

f: function analytical function to differentiate.

x0: array vector location at which to differentiate 'f'. If x0 is an nXm array, then 'f' is assumed to be a function of n*m variables.

vec: array vector defining the line along which to take the derivative. It should be the same size as x0, but need not be a vector of unit length.

****options:** optional arguments to pass on to Derivative.

Returns

dder: scalar estimate of the first derivative of 'f' in the specified direction.

See also:

Derivative

Gradient

Examples

At the global minimizer (1,1) of the Rosenbrock function, compute the directional derivative in the direction [1 2]

```
>>> import numpy as np
>>> import numdifftools as nd
>>> vec = np.r_[1, 2]
>>> rosen = lambda x: (1-x[0])**2 + 105*(x[1]-x[0]**2)**2
>>> dd, info = nd.directionaldiff(rosen, [1, 1], vec, full_output=True)
>>> np.allclose(dd, 0)
True
>>> np.abs(info.error_estimate)<1e-14
True
```

Extrapolation module

Created on 28. aug. 2015

@author: pab

class Dea(*limexp=3*)

Bases: object

Extrapolate a slowly convergent sequence

LIMEXP is the maximum number of elements the epsilon table data can contain. The epsilon table is stored in the first (LIMEXP+2) entries of EPSTAB.

Notes

List of major variables:

E0,E1,E2,E3 - DOUBLE PRECISION The 4 elements on which the computation of a new element in the epsilon table is based.

NRES - INTEGER Number of extrapolation results actually generated by the epsilon algorithm in prior calls to the routine.

NEWELM - INTEGER Number of elements to be computed in the new diagonal of the epsilon table. The condensed epsilon table is computed. Only those elements needed for the computation of the next diagonal are preserved.

RES - DOUBLE PREISION New element in the new diagonal of the epsilon table.

ERROR - DOUBLE PRECISION An estimate of the absolute error of RES. Routine decides whether RESULT=RES or RESULT=SVALUE by comparing ERROR with abserr from the previous call.

RES3LA - DOUBLE PREISION Vector of DIMENSION 3 containing at most the last 3 results.

limexp

!! processed by numpydoc !!

class EpsAlg(*limexp=3*)

Bases: object

Extrapolate a slowly convergent sequence

This implementaion is from [R9678172c97e0-1]

References

class Richardson(*step_ratio=2.0*, *step=1*, *order=1*, *num_terms=2*)
Bases: object

Extrapolates as sequence with Richardsons method

Notes

Suppose you have series expansion that goes like this

L = f(h) + a0 * h^p_0 + a1 * h^p_1+ a2 * h^p_2 + ...

where p_i = order + step * i and f(h) -> L as h -> 0, but f(0) != L.

If we evaluate the right hand side for different stepsizes h we can fit a polynomial to that sequence of approximations. This is exactly what this class does.

Examples

```
>>> import numpy as np
>>> import numdifftools as nd
>>> n = 3
>>> Ei = np.zeros((n,1))
>>> h = np.zeros((n,1))
>>> linfun = lambda i : np.linspace(0, np.pi/2., 2**(i+5)+1)
>>> for k in np.arange(n):
...     x = linfun(k)
...     h[k] = x[1]
...     Ei[k] = np.trapz(np.sin(x),x)
>>> En, err, step = nd.Richardson(step=1, order=1)(Ei, h)
>>> truErr = np.abs(En-1.)
>>> np.all(truErr < err)
True
>>> np.all(np.abs(Ei-1)<1e-3)
True
>>> np.allclose(En, 1)
True
```

extrapolate(*self*, *sequence*, *steps*)

rule(*self*, *sequence_length=None*)

convolve(*sequence*, *rule*, ***kwds*)
Wrapper around scipy.ndimage.convolve1d that allows complex input.

dea3(*v0*, *v1*, *v2*, *symmetric=False*)
Extrapolate a slowly convergent sequence

Parameters

v0, v1, v2 [array-like] 3 values of a convergent sequence to extrapolate

Returns

result [array-like] extrapolated value

abserr [array-like] absolute error estimate

See also:

dea

Notes

DEA3 attempts to extrapolate nonlinearly to a better estimate of the sequence's limiting value, thus improving the rate of convergence. The routine is based on the epsilon algorithm of P. Wynn, see *[1]*.

References

[1]

Examples

integrate sin(x) from 0 to pi/2

```
>>> import numpy as np
>>> import numdifftools as nd
>>> Ei= np.zeros(3)
>>> linfun = lambda i : np.linspace(0, np.pi/2., 2**(i+5)+1)
>>> for k in np.arange(3):
...     x = linfun(k)
...     Ei[k] = np.trapz(np.sin(x),x)
>>> [En, err] = nd.dea3(Ei[0], Ei[1], Ei[2])
>>> truErr = np.abs(En-1.)
>>> np.all(truErr < err)
True
>>> np.allclose(En, 1)
True
>>> np.all(np.abs(Ei-1)<1e-3)
True
```

dea_demo()

```
>>> from numdifftools.extrapolation import dea_demo
>>> dea_demo()
NO. PANELS      TRAP. APPROX          APPROX W/EA           abserr
    1           0.78539816            0.78539816            0.78539816
    2           0.94805945            0.94805945            0.97596771
    4           0.98711580            0.99945672            0.21405856
    8           0.99678517            0.99996674            0.00306012
   16           0.99919668            0.99999988            0.00115259
   32           0.99979919            1.00000000            0.00057665
   64           0.99994980            1.00000000            0.00003338
  128           0.99998745            1.00000000            0.00000012
  256           0.99999686            1.00000000            0.00000000
  512           0.99999922            1.00000000            0.00000000
 1024           0.99999980            1.00000000            0.00000000
 2048           0.99999995            1.00000000            0.00000000
```

epsalg_demo()

```
>>> from numdifftools.extrapolation import epsalg_demo
>>> epsalg_demo()
NO. PANELS      TRAP. APPROX          APPROX W/EA           abserr
    1           0.78539816            0.78539816            0.21460184
    2           0.94805945            0.94805945            0.05194055
    4           0.98711580            0.99945672            0.00054328
    8           0.99678517            0.99996674            0.00003326
   16           0.99919668            0.99999988            0.00000012
```

32	0.99979919	1.00000000	0.00000000
64	0.99994980	1.00000000	0.00000000
128	0.99998745	1.00000000	0.00000000
256	0.99999686	1.00000000	0.00000000
512	0.99999922	1.00000000	0.00000000

max_abs(*a1*, *a2*)

Fornberg finite difference approximations

class Taylor(*fun*, *n=1*, *r=0.0061*, *num_extrap=3*, *step_ratio=1.6*, ***kwds*)

Bases: object

Return Taylor coefficients of complex analytic function using FFT

Parameters

fun [callable] function to differentiate

z0 [real or complex scalar at which to evaluate the derivatives]

n [scalar integer, default 1] Number of taylor coefficents to compute. Maximum number is 100.

r [real scalar, default 0.0061] Initial radius at which to evaluate. For well-behaved functions, the computation should be insensitive to the initial radius to within about four orders of magnitude.

num_extrap [scalar integer, default 3] number of extrapolation steps used in the calculation

step_ratio [real scalar, default 1.6] Initial grow/shrinking factor for finding the best radius.

max_iter [scalar integer, default 30] Maximum number of iterations

min_iter [scalar integer, default max_iter // 2] Minimum number of iterations before the solution may be deemed degenerate. A larger number allows the algorithm to correct a bad initial radius.

full_output [bool, optional] If *full_output* is False, only the coefficents is returned (default). If *full_output* is True, then (coefs, status) is returned

Returns

coefs [ndarray] array of taylor coefficents

status: Optional object into which output information is written: degenerate: True if the algorithm was unable to bound the error iterations: Number of iterations executed function_count: Number of function calls final_radius: Ending radius of the algorithm failed: True if the maximum number of iterations was reached error_estimate: approximate bounds of the rounding error.

Notes

This module uses the method of Fornberg to compute the Taylor series coefficients of a complex analytic function along with error bounds. The method uses a Fast Fourier Transform to invert function evaluations around a circle into Taylor series coefficients and uses Richardson Extrapolation to improve and bound the estimate. Unlike real-valued finite differences, the method searches for a desirable radius and so is reasonably insensitive to the initial radius-to within a number of orders of magnitude at least. For most cases, the default configuration is likely to succeed.

Restrictions

The method uses the coefficients themselves to control the truncation error, so the error will not be properly bounded for functions like low-order polynomials whose Taylor series coefficients are nearly zero. If the

error cannot be bounded, degenerate flag will be set to true, and an answer will still be computed and returned but should be used with caution.

References

[1] Fornberg, B. (1981). Numerical Differentiation of Analytic Functions. ACM Transactions on Mathematical Software (TOMS), 7(4), 512-526. http://doi.org/10.1145/355972.355979

Examples

Compute the first 6 taylor coefficients 1 / (1 - z) expanded round z0 = 0: >>> import numdifftools.fornberg as ndf >>> import numpy as np >>> c, info = ndf.Taylor(lambda x: 1./(1-x), n=6, full_output=True)(z0=0) >>> np.allclose(c, np.ones(8)) True >>> np.all(info.error_estimate < 1e-9) True >>> (info.function_count, info.iterations, info.failed) == (144, 18, False) True

derivative (*fun*, *z0*, *n=1*, ***kwds*)

Calculate n-th derivative of complex analytic function using FFT

Parameters

fun [callable] function to differentiate

z0 [real or complex scalar at which to evaluate the derivatives]

n [scalar integer, default 1] Number of derivatives to compute where 0 represents the value of the function and n represents the nth derivative. Maximum number is 100.

r [real scalar, default 0.0061] Initial radius at which to evaluate. For well-behaved functions, the computation should be insensitive to the initial radius to within about four orders of magnitude.

max_iter [scalar integer, default 30] Maximum number of iterations

min_iter [scalar integer, default max_iter // 2] Minimum number of iterations before the solution may be deemed degenerate. A larger number allows the algorithm to correct a bad initial radius.

step_ratio [real scalar, default 1.6] Initial grow/shrinking factor for finding the best radius.

num_extrap [scalar integer, default 3] number of extrapolation steps used in the calculation

full_output [bool, optional] If *full_output* is False, only the derivative is returned (default). If *full_output* is True, then (der, status) is returned *der* is the derivative, and *status* is a Results object.

Returns

der [ndarray] array of derivatives

status: Optional object into which output information is written. Fields: degenerate: True if the algorithm was unable to bound the error iterations: Number of iterations executed function_count: Number of function calls final_radius: Ending radius of the algorithm failed: True if the maximum number of iterations was reached error_estimate: approximate bounds of the rounding error.

This module uses the method of Fornberg to compute the derivatives of a

complex analytic function along with error bounds. The method uses a

Fast Fourier Transform to invert function evaluations around a circle into

Taylor series coefficients, uses Richardson Extrapolation to improve

and bound the estimate, then multiplies by a factorial to compute the

derivatives. Unlike real-valued finite differences, the method searches for

a desirable radius and so is reasonably insensitive to the initial

radius-to within a number of orders of magnitude at least. For most cases,

the default configuration is likely to succeed.

Restrictions

The method uses the coefficients themselves to control the truncation

error, so the error will not be properly bounded for functions like

low-order polynomials whose Taylor series coefficients are nearly zero.

If the error cannot be bounded, degenerate flag will be set to true, and

an answer will still be computed and returned but should be used with

caution.

References

[1] Fornberg, B. (1981). Numerical Differentiation of Analytic Functions. ACM Transactions on Mathematical Software (TOMS), 7(4), 512-526. http://doi.org/10.1145/355972.355979

Examples

To compute the first five derivatives of 1 / (1 - z) at z = 0: Compute the first 6 taylor derivatives of 1 / (1 - z) at z0 = 0: >>> import numdifftools.fornberg as ndf >>> import numpy as np >>> def fun(x): ... return 1./(1-x) >>> c, info = ndf.derivative(fun, z0=0, n=6, full_output=True) >>> np.allclose(c, [1, 1, 2, 6, 24, 120, 720, 5040]) True >>> np.all(info.error_estimate < 1e-9*c.real) True >>> (info.function_count, info.iterations, info.failed) == (144, 18, False) True

fd_derivative (*fx*, *x*, *n=1*, *m=2*)

Return the n'th derivative for all points using Finite Difference method.

Parameters

fx [vector] function values which are evaluated on x i.e. fx[i] = f(x[i])

x [vector] abscissas on which fx is evaluated. The x values can be arbitrarily spaced but must be distinct and len(x) > n.

n [scalar integer] order of derivative.

m [scalar integer] defines the stencil size. The stencil size is of 2 * mm + 1 points in the interior, and 2 * mm + 2 points for each of the 2 * mm boundary points where mm = n // 2 + m.

fd_derivative evaluates an approximation for the n'th derivative of the

vector function f(x) using the Fornberg finite difference method.

Restrictions: 0 < n < len(x) and 2*mm+2 <= len(x)

See also:

fd_weights

Examples

```
>>> import numpy as np
>>> import numdifftools.fornberg as ndf
>>> x = np.linspace(-1, 1, 25)
>>> fx = np.exp(x)
>>> df = ndf.fd_derivative(fx, x, n=1)
>>> np.allclose(df, fx)
True
```

fd_weights (*x*, *x0=0*, *n=1*)

Return finite difference weights for the n'th derivative.

Parameters

x [vector] abscissas used for the evaluation for the derivative at x0.

x0 [scalar] location where approximations are to be accurate

n [scalar integer] order of derivative. Note for n=0 this can be used to evaluate the interpolating polynomial itself.

See also:

fd_weights_all

Examples

```
>>> import numpy as np
>>> import numdifftools.fornberg as ndf
>>> x = np.linspace(-1, 1, 5) * 1e-3
>>> w = ndf.fd_weights(x, x0=0, n=1)
>>> df = np.dot(w, np.exp(x))
>>> np.allclose(df, 1)
True
```

fd_weights_all (*x*, *x0=0*, *n=1*)

Return finite difference weights for derivatives of all orders up to n.

Parameters

x [vector, length m] x-coordinates for grid points

x0 [scalar] location where approximations are to be accurate

n [scalar integer] highest derivative that we want to find weights for

Returns

weights [array, shape n+1 x m] contains coefficients for the j'th derivative in row j (0 <= j <= n)

See also:

fd_weights

Notes

The x values can be arbitrarily spaced but must be distinct and len(x) > n.

The Fornberg algorithm is much more stable numerically than regular vandermonde systems for large values of n.

References

B. Fornberg (1998) "Calculation of weights_and_points in finite difference formulas", SIAM Review 40, pp. 685-691.

http://www.scholarpedia.org/article/Finite_difference_method

richardson(*vals*, *k*, *c=None*)
Richardson extrapolation with parameter estimation

richardson_parameter(*vals*, *k*)

taylor(*fun*, *z0=0*, *n=1*, *r=0.0061*, *num_extrap=3*, *step_ratio=1.6*, ***kwds*)
Return Taylor coefficients of complex analytic function using FFT

Parameters

fun [callable] function to differentiate

z0 [real or complex scalar at which to evaluate the derivatives]

n [scalar integer, default 1] Number of taylor coefficents to compute. Maximum number is 100.

r [real scalar, default 0.0061] Initial radius at which to evaluate. For well-behaved functions, the computation should be insensitive to the initial radius to within about four orders of magnitude.

num_extrap [scalar integer, default 3] number of extrapolation steps used in the calculation

step_ratio [real scalar, default 1.6] Initial grow/shrinking factor for finding the best radius.

max_iter [scalar integer, default 30] Maximum number of iterations

min_iter [scalar integer, default max_iter // 2] Minimum number of iterations before the solution may be deemed degenerate. A larger number allows the algorithm to correct a bad initial radius.

full_output [bool, optional] If *full_output* is False, only the coefficents is returned (default). If *full_output* is True, then (coefs, status) is returned

Returns

coefs [ndarray] array of taylor coefficents

status: Optional object into which output information is written: degenerate: True if the algorithm was unable to bound the error iterations: Number of iterations executed function_count: Number of function calls final_radius: Ending radius of the algorithm failed: True if the maximum number of iterations was reached error_estimate: approximate bounds of the rounding error.

Notes

This module uses the method of Fornberg to compute the Taylor series coefficents of a complex analytic function along with error bounds. The method uses a Fast Fourier Transform to invert function evaluations around a circle into Taylor series coefficients and uses Richardson Extrapolation to improve and bound the estimate. Unlike real-valued finite differences, the method searches for a desirable radius and so is reasonably insensitive to the initial radius-to within a number of orders of magnitude at least. For most cases, the default configuration is likely to succeed.

Restrictions

The method uses the coefficients themselves to control the truncation error, so the error will not be properly bounded for functions like low-order polynomials whose Taylor series coefficients are nearly zero. If the error cannot be bounded, degenerate flag will be set to true, and an answer will still be computed and returned but should be used with caution.

References

[1] Fornberg, B. (1981). Numerical Differentiation of Analytic Functions. ACM Transactions on Mathematical Software (TOMS), 7(4), 512-526. http://doi.org/10.1145/355972.355979

Examples

Compute the first 6 taylor coefficients 1 / (1 - z) expanded round z0 = 0: >>> import numdifftools.fornberg as ndf >>> import numpy as np >>> c, info = ndf.taylor(lambda x: 1./(1-x), z0=0, n=6, full_output=True) >>> np.allclose(c, np.ones(8)) True >>> np.all(info.error_estimate < 1e-9) True >>> (info.function_count, info.iterations, info.failed) == (144, 18, False) True

Limits module

Created on 27. aug. 2015

@author: pab Author: John D'Errico e-mail: woodchips@rochester.rr.com Release: 1.0 Release date: 5/23/2008

class CStepGenerator(*base_step=None*, *step_ratio=4.0*, *num_steps=None*, *step_nom=None*, *offset=0*, *scale=1.2*, *use_exact_steps=True*, *path='radial'*, *dtheta=0.39269908169872414*, ***kwds*)

Bases: *numdifftools.step_generators.MinStepGenerator*

Generates a sequence of steps

where

steps = base_step * step_nom * (exp(1j*dtheta) * step_ratio) ** (i + offset)

for i = 0, 1, . . . , num_steps-1

Parameters

base_step [float, array-like, default None] Defines the minimum step, if None, the value is set to EPS**(1/scale)

step_ratio [real scalar, optional, default 4.0] Ratio between sequential steps generated.

num_steps [scalar integer, optional,] defines number of steps generated. If None the value is 2 * int(round(16.0/log(abs(step_ratio)))) + 1

step_nom [default maximum(log(1+|x|), 1)] Nominal step where x is supplied at runtime through the __call__ method.

offset [real scalar, optional, default 0] offset to the base step

use_exact_steps [boolean] If true make sure exact steps are generated

scale [real scalar, default 1.2] scale used in base step.

path ['spiral' or 'radial'] Specifies the type of path to take the limit along.

dtheta: real scalar If the path is spiral it will follow an exponential spiral into the limit, with angular steps at dtheta radians.

dtheta
!! processed by numpydoc !!

num_steps
!! processed by numpydoc !!

step_ratio
!! processed by numpydoc !!

class Limit (*fun*, *step=None*, *method='above'*, *order=4*, *full_output=False*, ***options*)

Bases: numdifftools.limits._Limit

Compute limit of a function at a given point

Parameters

fun [callable] function fun(z, **args*, ***kwds*) to compute the limit for z->z0. The function, fun, is assumed to return a result of the same shape and size as its input, *z*.

step: float, complex, array-like or StepGenerator object, optional Defines the spacing used in the approximation. Default is CStepGenerator(base_step=step, **options)

method [{'above', 'below'}] defines if the limit is taken from *above* or *below*

order: positive scalar integer, optional. defines the order of approximation used to find the specified limit. The order must be member of [1 2 3 4 5 6 7 8]. 4 is a good compromise.

full_output: bool If true return additional info.

options: options to pass on to CStepGenerator

Returns

limit_fz: array like estimated limit of f(z) as z –> z0

info: Only given if full_output is True and contains the following:

error estimate: ndarray 95 % uncertainty estimate around the limit, such that abs(limit_fz - lim z->z0 f(z)) < error_estimate

final_step: ndarray final step used in approximation

Notes

Limit computes the limit of a given function at a specified point, z0. When the function is evaluable at the point in question, this is a simple task. But when the function cannot be evaluated at that location due to a singularity, you may need a tool to compute the limit. *Limit* does this, as well as produce an uncertainty estimate in the final result.

The methods used by *Limit* are Richardson extrapolation in a combination with Wynn's epsilon algorithm which also yield an error estimate. The user can specify the method order, as well as the path into z0. z0 may be real or complex. *Limit* uses a proportionally cascaded series of function evaluations, moving away from your point of evaluation along a path along the real line (or in the complex plane for complex z0 or step.) The *step_ratio* is the ratio used between sequential steps. The sign of step allows you to specify a limit from above or below. Negative values of step will cause the limit to be taken approaching z0 from below.

A smaller *step_ratio* means that *Limit* will take more function evaluations to evaluate the limit, but the result will potentially be less accurate. The *step_ratio* MUST be a scalar larger than 1. A value in the range [2,100] is recommended. 4 seems a good compromise.

```
>>> import numpy as np
>>> from numdifftools.limits import Limit
>>> def f(x): return np.sin(x)/x
>>> lim_f0, err = Limit(f, full_output=True)(0)
>>> np.allclose(lim_f0, 1)
True
>>> np.allclose(err.error_estimate, 1.77249444610966e-15)
True
```

Compute the derivative of cos(x) at x == pi/2. It should be -1. The limit will be taken as a function of the differential parameter, dx.

```
>>> x0 = np.pi/2;
>>> def g(x): return (np.cos(x0+x)-np.cos(x0))/x
>>> lim_g0, err = Limit(g, full_output=True)(0)
>>> np.allclose(lim_g0, -1)
True
>>> err.error_estimate < 1e-14
True
```

Compute the residue at a first order pole at z = 0 The function 1./(1-exp(2*z)) has a pole at z == 0. The residue is given by the limit of z*fun(z) as z –> 0. Here, that residue should be -0.5.

```
>>> def h(z): return -z/(np.expm1(2*z))
>>> lim_h0, err = Limit(h, full_output=True)(0)
>>> np.allclose(lim_h0, -0.5)
True
>>> err.error_estimate < 1e-14
True
```

Compute the residue of function 1./sin(z)**2 at z = 0. This pole is of second order thus the residue is given by the limit of z**2*fun(z) as z –> 0.

```
>>> def g(z): return z**2/(np.sin(z)**2)
>>> lim_gpi, err = Limit(g, full_output=True)(0)
>>> np.allclose(lim_gpi, 1)
True
>>> err.error_estimate < 1e-14
True
```

A more difficult limit is one where there is significant subtractive cancellation at the limit point. In the following example, the cancellation is second order. The true limit should be 0.5.

```
>>> def k(x): return (x*np.exp(x)-np.expm1(x))/x**2
>>> lim_k0,err = Limit(k, full_output=True)(0)
>>> np.allclose(lim_k0, 0.5)
True
>>> err.error_estimate < 1.0e-8
True
```

```
>>> def h(x): return  (x-np.sin(x))/x**3
>>> lim_h0, err = Limit(h, full_output=True)(0)
>>> np.allclose(lim_h0, 1./6)
True
>>> err.error_estimate < 1e-8
True
```

limit (*self*, *x*, **args*, ***kwds*)

class Residue (*f*, *step=None*, *method='above'*, *order=None*, *pole_order=1*, *full_output=False*, ***options*)

Bases: *numdifftools.limits.Limit*

Compute residue of a function at a given point

Parameters

fun [callable] function fun(z, **args*, ***kwds*) to compute the Residue at z=z0. The function, fun, is assumed to return a result of the same shape and size as its input, *z*.

step: float, complex, array-like or StepGenerator object, optional Defines the spacing used in the approximation. Default is CStepGenerator(base_step=step, **options)

method [{'above', 'below'}] defines if the limit is taken from *above* or *below*

order: positive scalar integer, optional. defines the order of approximation used to find the specified limit. The order must be member of [1 2 3 4 5 6 7 8]. 4 is a good compromise.

pole_order [scalar integer] specifies the order of the pole at z0.

full_output: bool If true return additional info.

options: options to pass on to CStepGenerator

Returns

res_fz: array like estimated residue, i.e., limit of f(z)*(z-z0)**pole_order as z –> z0 When the residue is estimated as approximately zero,

the wrong order pole may have been specified.

info: namedtuple, Only given if full_output is True and contains the following:

error estimate: ndarray 95 % uncertainty estimate around the residue, such that abs(res_fz - lim z->z0 f(z)*(z-z0)**pole_order) < error_estimate Large uncertainties here suggest that the wrong order pole was specified for f(z0).

final_step: ndarray final step used in approximation

Notes

Residue computes the residue of a given function at a simple first order pole, or at a second order pole.

The methods used by residue are polynomial extrapolants, which also yield an error estimate. The user can specify the method order, as well as the order of the pole.

z0 - scalar point at which to compute the residue. z0 may be real or complex.

See the document DERIVEST.pdf for more explanation of the algorithms behind the parameters of Residue. In most cases, the user should never need to specify anything other than possibly the PoleOrder.

Examples

A first order pole at z = 0

```
>>> from numdifftools.limits import Residue
>>> def f(z): return -1./(np.expm1(2*z))
>>> res_f, info = Residue(f, full_output=True)(0)
>>> np.allclose(res_f, -0.5)
True
>>> info.error_estimate < 1e-14
True
```

A second order pole around z = 0 and z = pi >>> def h(z): return 1.0/np.sin(z)**2 >>> res_h, info = Residue(h, full_output=True, pole_order=2)([0, np.pi]) >>> np.allclose(res_h, 1) True >>> (info.error_estimate < 1e-10).all() True

Numdifftools Algopy module

Numdifftools.nd_algopy

This module provide an easy to use interface to derivatives calculated with AlgoPy. Algopy stands for Algorithmic Differentiation in Python.

The purpose of AlgoPy is the evaluation of higher-order derivatives in the forward and reverse mode of Algorithmic Differentiation (AD) of functions that are implemented as Python programs. Particular focus are functions that

contain numerical linear algebra functions as they often appear in statistically motivated functions. The intended use of AlgoPy is for easy prototyping at reasonable execution speeds. More precisely, for a typical program a directional derivative takes order 10 times as much time as time as the function evaluation. This is approximately also true for the gradient.

Algoritmic differentiation

Algorithmic differentiation (AD) is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.

Algorithmic differentiation is not:

Symbolic differentiation, nor Numerical differentiation (the method of finite differences). These classical methods run into problems: symbolic differentiation leads to inefficient code (unless carefully done) and faces the difficulty of converting a computer program into a single expression, while numerical differentiation can introduce round-off errors in the discretization process and cancellation. Both classical methods have problems with calculating higher derivatives, where the complexity and errors increase. Finally, both classical methods are slow at computing the partial derivatives of a function with respect to many inputs, as is needed for gradient-based optimization algorithms. Algoritmic differentiation solves all of these problems.

References

Sebastian F. Walter and Lutz Lehmann 2013, "Algorithmic differentiation in Python with AlgoPy", in Journal of Computational Science, vol 4, no 5, pp 334 - 344, http://www.sciencedirect.com/science/article/pii/S1877750311001013

https://en.wikipedia.org/wiki/Automatic_differentiation

https://pythonhosted.org/algopy/index.html

class Derivative(*fun*, *n=1*, *method='forward'*, *full_output=False*)

Bases: `numdifftools.nd_algopy._Derivative`

Calculate n-th derivative with Algorithmic Differentiation method

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

n [int, optional] Order of the derivative.

method [string, optional {'forward', 'reverse'}] defines method used in the approximation

Returns

der [ndarray] array of derivatives

See also:

Gradient

Hessdiag

Hessian

Jacobian

Notes

Algorithmic differentiation is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.

References

Sebastian F. Walter and Lutz Lehmann 2013, "Algorithmic differentiation in Python with AlgoPy", in Journal of Computational Science, vol 4, no 5, pp 334 - 344, http://www.sciencedirect.com/science/article/pii/S1877750311001013

https://en.wikipedia.org/wiki/Automatic_differentiation

Examples

1'st and 2'nd derivative of exp(x), at x == 1

```
>>> import numpy as np
>>> import numdifftools.nd_algopy as nda
>>> fd = nda.Derivative(np.exp)               # 1'st derivative
>>> np.allclose(fd(1), 2.718281828459045)
True
>>> fd5 = nda.Derivative(np.exp, n=5)          # 5'th derivative
>>> np.allclose(fd5(1), 2.718281828459045)
True
```

1'st derivative of x^3+x^4, at x = [0,1]

```
>>> fun = lambda x: x**3 + x**4
>>> fd3 = nda.Derivative(fun)
>>> np.allclose(fd3([0,1]), [ 0.,  7.])
True
```

Methods

__call__(callable with the following parameters:) x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.	

class Gradient(*fun*, *n=1*, *method='forward'*, *full_output=False*)

Bases: numdifftools.nd_algopy._Derivative

Calculate Gradient with Algorithmic Differentiation method

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

method [string, optional {'forward', 'reverse'}] defines method used in the approximation

Returns

grad [array] gradient

See also:

Derivative

Jacobian

Hessdiag

Hessian

Notes

Algorithmic differentiation is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.

References

Sebastian F. Walter and Lutz Lehmann 2013, "Algorithmic differentiation in Python with AlgoPy", in Journal of Computational Science, vol 4, no 5, pp 334 - 344, http://www.sciencedirect.com/science/article/pii/S1877750311001013

https://en.wikipedia.org/wiki/Automatic_differentiation

Examples

```
>>> import numdifftools.nd_algopy as nda
>>> fun = lambda x: np.sum(x**2)
>>> df = nda.Gradient(fun, method='reverse')
>>> np.allclose(df([1,2,3]), [ 2.,  4.,  6.])
True
```

#At [x,y] = [1,1], compute the numerical gradient #of the function sin(x-y) + y*exp(x)

```
>>> sin = np.sin; exp = np.exp
>>> z = lambda xy: sin(xy[0]-xy[1]) + xy[1]*exp(xy[0])
>>> dz = nda.Gradient(z)
>>> grad2 = dz([1, 1])
>>> np.allclose(grad2, [ 3.71828183,  1.71828183])
True
```

#At the global minimizer (1,1) of the Rosenbrock function, #compute the gradient. It should be essentially zero.

```
>>> rosen = lambda x : (1-x[0])**2 + 105.*(x[1]-x[0]**2)**2
>>> rd = nda.Gradient(rosen)
>>> grad3 = rd([1,1])
>>> np.allclose(grad3, [ 0.,  0.])
True
```

Methods

__call__(callable with the following parameters:) x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.	

class Hessdiag(*f*, *method='forward'*, *full_output=False*)

Bases: *numdifftools.nd_algopy.Hessian*

Calculate Hessian diagonal with Algorithmic Differentiation method

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

method [string, optional {'forward', 'reverse'}] defines method used in the approximation

Returns

hessdiag [ndarray] Hessian diagonal array of partial second order derivatives.

See also:

Derivative

Gradient

Jacobian

Hessian

Notes

Algorithmic differentiation is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.

References

Sebastian F. Walter and Lutz Lehmann 2013, "Algorithmic differentiation in Python with AlgoPy", in Journal of Computational Science, vol 4, no 5, pp 334 - 344, http://www.sciencedirect.com/science/article/pii/S1877750311001013

https://en.wikipedia.org/wiki/Automatic_differentiation

Examples

```
>>> import numdifftools.nd_algopy as nda
```

Rosenbrock function, minimized at [1,1]

```
>>> rosen = lambda x : (1.-x[0])**2 + 105*(x[1]-x[0]**2)**2
>>> Hfun = nda.Hessdiag(rosen)
>>> h = Hfun([1, 1]) #  h =[ 842, 210]
>>> np.allclose(h, [ 842.,  210.])
True
```

cos(x-y), at (0,0)

```
>>> cos = np.cos
>>> fun = lambda xy : cos(xy[0]-xy[1])
>>> Hfun2 = nda.Hessdiag(fun)
>>> h2 = Hfun2([0, 0]) # h2 = [-1, -1]
>>> np.allclose(h2, [-1., -1.])
True
```

```
>>> Hfun3 = nda.Hessdiag(fun, method='reverse')
>>> h3 = Hfun3([0, 0]) # h2 = [-1, -1];
>>> np.allclose(h3, [-1., -1.])
True
```

Methods

__call__(callable with the following parameters:) x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.	

class Hessian(*f*, *method='forward'*, *full_output=False*)

Bases: numdifftools.nd_algopy._Derivative

Calculate Hessian with Algorithmic Differentiation method

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

method [string, optional {'forward', 'reverse'}] defines method used in the approximation

Returns

hess [ndarray] array of partial second derivatives, Hessian

See also:

Derivative

Gradient

Jacobian

Hessdiag

Notes

Algorithmic differentiation is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.

References

Sebastian F. Walter and Lutz Lehmann 2013, "Algorithmic differentiation in Python with AlgoPy", in Journal of Computational Science, vol 4, no 5, pp 334 - 344, http://www.sciencedirect.com/science/article/pii/S1877750311001013

https://en.wikipedia.org/wiki/Automatic_differentiation

Examples

```
>>> import numdifftools.nd_algopy as nda
```

Rosenbrock function, minimized at [1,1]

```
>>> rosen = lambda x : (1.-x[0])**2 + 105*(x[1]-x[0]**2)**2
>>> Hf = nda.Hessian(rosen)
>>> h = Hf([1, 1]) #  h =[ 842 -420; -420, 210];
>>> np.allclose(h, [[ 842., -420.],
...                 [-420.,  210.]])
True
```

cos(x-y), at (0,0)

```
>>> cos = np.cos
>>> fun = lambda xy : cos(xy[0]-xy[1])
>>> Hfun2 = nda.Hessian(fun)
>>> h2 = Hfun2([0, 0]) # h2 = [-1 1; 1 -1]
>>> np.allclose(h2, [[-1.,  1.],
...                  [ 1., -1.]])
True
```

```
>>> Hfun3 = nda.Hessian(fun, method='reverse')
>>> h3 = Hfun3([0, 0]) # h2 = [-1, 1; 1, -1];
>>> np.allclose(h3, [[-1.,  1.],
...                  [ 1., -1.]])
True
```

Methods

__call__(callable with the following parameters:) x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.	

class Jacobian(*fun*, *n=1*, *method='forward'*, *full_output=False*)

Bases: *numdifftools.nd_algopy.Gradient*

Calculate Jacobian with Algorithmic Differentiation method

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

method [string, optional {'forward', 'reverse'}] defines method used in the approximation

Returns

jacob [array] Jacobian

See also:

Derivative

Gradient

Hessdiag

Hessian

Notes

Algorithmic differentiation is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.

References

Sebastian F. Walter and Lutz Lehmann 2013, "Algorithmic differentiation in Python with AlgoPy", in Journal of Computational Science, vol 4, no 5, pp 334 - 344, http://www.sciencedirect.com/science/article/pii/S1877750311001013

https://en.wikipedia.org/wiki/Automatic_differentiation

Examples

```
>>> import numdifftools.nd_algopy as nda
```

#(nonlinear least squares)

```
>>> xdata = np.arange(0,1,0.1)
>>> ydata = 1+2*np.exp(0.75*xdata)
>>> fun = lambda c: (c[0]+c[1]*np.exp(c[2]*xdata) - ydata)**2
```

Jfun = nda.Jacobian(fun) # Todo: This does not work Jfun([1,2,0.75]).T # should be numerically zero array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])

```
>>> Jfun2 = nda.Jacobian(fun, method='reverse')
>>> res = Jfun2([1,2,0.75]).T # should be numerically zero
>>> np.allclose(res,
...                 [[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
...                  [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
...                  [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
True
```

```
>>> f2 = lambda x : x[0]*x[1]*x[2]**2
>>> Jfun2 = nda.Jacobian(f2)
>>> np.allclose(Jfun2([1., 2., 3.]), [[ 18., 9., 12.]])
True
```

```
>>> Jfun21 = nda.Jacobian(f2, method='reverse')
>>> np.allclose(Jfun21([1., 2., 3.]), [[ 18., 9., 12.]])
True
```

```
>>> def fun3(x):
...     n = int(np.prod(np.shape(x[0])))
...     out = nda.algopy.zeros((2, n), dtype=x)
...     out[0] = x[0]*x[1]*x[2]**2
...     out[1] = x[0]*x[1]*x[2]
...     return out
>>> Jfun3 = nda.Jacobian(fun3)
```

```
>>> np.allclose(Jfun3([1., 2., 3.]), [[[18., 9., 12.]], [[6., 3., 2.]]])
True
>>> np.allclose(Jfun3([4., 5., 6.]), [[[180., 144., 240.]],
...                                   [[30., 24., 20.]]])
True
>>> np.allclose(Jfun3(np.array([[1.,2.,3.], [4., 5., 6.]]).T),
...              [[[18.,    0.,    9.,    0.,   12.,    0.],
...                [0.,  180.,    0.,  144.,    0.,  240.]],
...               [[6.,    0.,    3.,    0.,    2.,    0.],
...                [0.,   30.,    0.,   24.,    0.,   20.]]])
True
```

Methods

__call__(callable with the following parameters:) x : array_like value at which function derivative is evaluated args : tuple Arguments for function *fun*. kwds : dict Keyword arguments for function *fun*.	

directionaldiff (*f*, *x0*, *vec*, ***options*)

Return directional derivative of a function of n variables

Parameters

fun: callable analytical function to differentiate.

x0: array vector location at which to differentiate fun. If x0 is an nxm array, then fun is assumed to be a function of n*m variables.

vec: array vector defining the line along which to take the derivative. It should be the same size as x0, but need not be a vector of unit length.

****options:** optional arguments to pass on to Derivative.

Returns

dder: scalar estimate of the first derivative of fun in the specified direction.

See also:

Derivative

Gradient

Examples

At the global minimizer (1,1) of the Rosenbrock function, compute the directional derivative in the direction [1 2]

```
>>> import numpy as np
>>> import numdifftools as nd
>>> vec = np.r_[1, 2]
>>> rosen = lambda x: (1-x[0])**2 + 105*(x[1]-x[0]**2)**2
>>> dd, info = nd.directionaldiff(rosen, [1, 1], vec, full_output=True)
>>> np.allclose(dd, 0)
True
>>> np.abs(info.error_estimate)<1e-14
True
```

Numdifftools Scipy module

class Gradient (*fun*, *step=None*, *method='central'*, *order=2*, *bounds=(-inf, inf)*, *sparsity=None*)
Bases: *numdifftools.nd_scipy.Jacobian*

Calculate Gradient with finite difference approximation

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

step [float, optional] Stepsize, if None, optimal stepsize is used, i.e., x * _EPS for method==‘complex‘ x * _EPS**(1/2) for method==‘forward‘ x * _EPS**(1/3) for method==‘central‘.

method [{‘central’, ‘complex’, ‘forward’}] defines the method used in the approximation.

See also:

Hessian, *Jacobian*

Examples

```
>>> import numpy as np
>>> import numdifftools.nd_scipy as nd
>>> fun = lambda x: np.sum(x**2)
>>> dfun = nd.Gradient(fun)
>>> np.allclose(dfun([1,2,3]), [ 2.,  4.,  6.])
True
```

At [x,y] = [1,1], compute the numerical gradient # of the function sin(x-y) + y*exp(x)

```
>>> sin = np.sin; exp = np.exp
>>> z = lambda xy: sin(xy[0]-xy[1]) + xy[1]*exp(xy[0])
>>> dz = nd.Gradient(z)
>>> grad2 = dz([1, 1])
>>> np.allclose(grad2, [ 3.71828183,  1.71828183])
True
```

At the global minimizer (1,1) of the Rosenbrock function, # compute the gradient. It should be essentially zero.

```
>>> rosen = lambda x : (1-x[0])**2 + 105.*(x[1]-x[0]**2)**2
>>> rd = nd.Gradient(rosen)
>>> grad3 = rd([1,1])
>>> np.allclose(grad3,[0, 0], atol=1e-7)
True
```

class Jacobian (*fun*, *step=None*, *method='central'*, *order=2*, *bounds=(-inf, inf)*, *sparsity=None*)
Bases: numdifftools.nd_scipy._Common

Calculate Jacobian with finite difference approximation

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

step [float, optional] Stepsize, if None, optimal stepsize is used, i.e., x * _EPS for method==‘complex‘ x * _EPS**(1/2) for method==‘forward‘ x * _EPS**(1/3) for method==‘central‘.

method [{‘central’, ‘complex’, ‘forward’}] defines the method used in the approximation.

Examples

```
>>> import numdifftools.nd_scipy as nd
```

#(nonlinear least squares)

```
>>> xdata = np.arange(0,1,0.1)
>>> ydata = 1+2*np.exp(0.75*xdata)
>>> fun = lambda c: (c[0]+c[1]*np.exp(c[2]*xdata) - ydata)**2
>>> np.allclose(fun([1, 2, 0.75]).shape, (10,))
True
>>> dfun = nd.Jacobian(fun)
>>> np.allclose(dfun([1, 2, 0.75]), np.zeros((10,3)))
True
```

```
>>> fun2 = lambda x : x[0]*x[1]*x[2]**2
>>> dfun2 = nd.Jacobian(fun2)
>>> np.allclose(dfun2([1.,2.,3.]), [[18., 9., 12.]])
True
```

```
>>> fun3 = lambda x : np.vstack((x[0]*x[1]*x[2]**2, x[0]*x[1]*x[2]))
```

TODO: The following does not work: der3 = nd.Jacobian(fun3)([1., 2., 3.]) np.allclose(der3, ... [[18., 9., 12.], [6., 3., 2.]]) True np.allclose(nd.Jacobian(fun3)([4., 5., 6.]), ... [[180., 144., 240.], [30., 24., 20.]]) True

np.allclose(nd.Jacobian(fun3)(np.array([[1.,2.,3.], [4., 5., 6.]]).T), ... [[[18., 180.], ... [9., 144.], ... [12., 240.]], ... [[6., 30.], ... [3., 24.], ... [2., 20.]]]) True

Numdifftools Statsmodels module

Numdifftools.nd_statsmodels

This module provides an easy to use interface to derivatives calculated with statsmodels.numdiff.

class Gradient (*fun*, *step=None*, *method='central'*, *order=None*)

Bases: *numdifftools.nd_statsmodels.Jacobian*

Calculate Gradient with finite difference approximation

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

step [float, optional] Stepsize, if None, optimal stepsize is used, i.e., x * _EPS for method==‘complex‘ x * _EPS**(1/2) for method==‘forward‘ x * _EPS**(1/3) for method==‘central‘.

method [{‘central’, ‘complex’, ‘forward’, ‘backward’}] defines the method used in the approximation.

See also:

Hessian, ***Jacobian***

Examples

```
>>> import numpy as np
>>> import numdifftools.nd_statsmodels as nd
>>> fun = lambda x: np.sum(x**2)
```

```
>>> dfun = nd.Gradient(fun)
>>> np.allclose(dfun([1,2,3]), [ 2.,  4.,  6.])
True
```

At [x,y] = [1,1], compute the numerical gradient # of the function sin(x-y) + y*exp(x)

```
>>> sin = np.sin; exp = np.exp
>>> z = lambda xy: sin(xy[0]-xy[1]) + xy[1]*exp(xy[0])
>>> dz = nd.Gradient(z)
>>> grad2 = dz([1, 1])
>>> np.allclose(grad2, [ 3.71828183,  1.71828183])
True
```

At the global minimizer (1,1) of the Rosenbrock function, # compute the gradient. It should be essentially zero.

```
>>> rosen = lambda x : (1-x[0])**2 + 105.*(x[1]-x[0]**2)**2
>>> rd = nd.Gradient(rosen)
>>> grad3 = rd([1,1])
>>> np.allclose(grad3,[0, 0])
True
```

class Hessian (*fun*, *step=None*, *method='central'*, *order=None*)

Bases: `numdifftools.nd_statsmodels._Common`

Calculate Hessian with finite difference approximation

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

step [float, optional] Stepsize, if None, optimal stepsize is used, i.e., x * _EPS**(1/3) for method=='forward', *complex* or *central2* x * _EPS**(1/4) for method=='central'.

method [{'central', 'complex', 'forward', 'backward'}] defines the method used in the approximation.

See also:

Jacobian*, *Gradient

Examples

```
>>> import numpy as np
>>> import numdifftools.nd_statsmodels as nd
```

Rosenbrock function, minimized at [1,1]

```
>>> rosen = lambda x : (1.-x[0])**2 + 105*(x[1]-x[0]**2)**2
>>> Hfun = nd.Hessian(rosen)
>>> h = Hfun([1, 1])
>>> np.allclose(h, [[ 842., -420.], [-420.,  210.]])
True
```

cos(x-y), at (0,0)

```
>>> cos = np.cos
>>> fun = lambda xy : cos(xy[0]-xy[1])
>>> Hfun2 = nd.Hessian(fun)
>>> h2 = Hfun2([0, 0])
>>> np.allclose(h2, [[-1.,  1.], [ 1., -1.]])
True
```

n

!! processed by numpydoc !!

class Jacobian (*fun*, *step=None*, *method='central'*, *order=None*)

Bases: `numdifftools.nd_statsmodels._Common`

Calculate Jacobian with finite difference approximation

Parameters

fun [function] function of one array fun(x, **args*, ***kwds*)

step [float, optional] Stepsize, if None, optimal stepsize is used, i.e., x * _EPS for method==‘complex‘ x * _EPS**(1/2) for method==‘forward‘ x * _EPS**(1/3) for method==‘central‘.

method [{‘central’, ‘complex’, ‘forward’, ‘backward’}] defines the method used in the approximation.

Examples

```
>>> import numdifftools.nd_statsmodels as nd
```

#(nonlinear least squares)

```
>>> xdata = np.arange(0,1,0.1)
>>> ydata = 1+2*np.exp(0.75*xdata)
>>> fun = lambda c: (c[0]+c[1]*np.exp(c[2]*xdata) - ydata)**2
>>> np.allclose(fun([1, 2, 0.75]).shape, (10,))
True
>>> dfun = nd.Jacobian(fun)
>>> np.allclose(dfun([1, 2, 0.75]), np.zeros((10,3)))
True
```

```
>>> fun2 = lambda x : x[0]*x[1]*x[2]**2
>>> dfun2 = nd.Jacobian(fun2)
>>> np.allclose(dfun2([1.,2.,3.]), [[18., 9., 12.]])
True
```

```
>>> fun3 = lambda x : np.vstack((x[0]*x[1]*x[2]**2, x[0]*x[1]*x[2]))
>>> np.allclose(nd.Jacobian(fun3)([1., 2., 3.]),
...             [[18., 9., 12.], [6., 3., 2.]])
True
>>> np.allclose(nd.Jacobian(fun3)([4., 5., 6.]),
...             [[180., 144., 240.], [30., 24., 20.]])
True
```

```
>>> np.allclose(nd.Jacobian(fun3)(np.array([[1.,2.,3.], [4., 5., 6.]]).T),
...             [[[  18.,  180.],
...               [   9.,  144.],
...               [  12.,  240.]],
...              [[   6.,   30.],
...               [   3.,   24.],
...               [   2.,   20.]]])
True
```

approx_fprime (*x*, *f*, *epsilon=None*, *args=()*, *kwargs=None*, *centered=True*)

Gradient of function, or Jacobian if function fun returns 1d array

Parameters

x [array] parameters at which the derivative is evaluated

fun [function] *fun(*((x,)+args), **kwargs)* returning either one value or 1d array

epsilon [float, optional] Stepsize, if None, optimal stepsize is used. This is _EPS**(1/2)*x for *centered* == False and _EPS**(1/3)*x for *centered* == True.

args [tuple] Tuple of additional arguments for function *fun*.

kwargs [dict] Dictionary of additional keyword arguments for function *fun*.

centered [bool] Whether central difference should be returned. If not, does forward differencing.

Returns

grad [array] gradient or Jacobian

Notes

If fun returns a 1d array, it returns a Jacobian. If a 2d array is returned by fun (e.g., with a value for each observation), it returns a 3d array with the Jacobian of each observation with shape xk x nobs x xk. I.e., the Jacobian of the first observation would be [:, 0, :]

CHAPTER 6

Changelog

6.1 Version 0.9.39 Jun 10, 2019

Robert Parini (1):

- Fix issue #43: numpy future warning

6.2 Version 0.9.38 Jun 10, 2019

Andrew Nelson (1):

- MAINT: special.factorial instead of misc.factorial

Dougal J. Sutherland (1):

- include LICENSE.txt in distributions

Per A Brodtkorb (140):

- Adjusted runtime for hypothesis tests to avoid failure and fixed pep8 failures.
- Fixed a bug in setup.cfg
- Replaced valarray function with numpy.full in step_generators.py
- Added try except on import of algopy
- Updated the badges used in the README.rst
- Replaced numpy.testing.Tester with pytest.
- Removed dependence on pyscaffold.
- Simplified setup.py and setup.cfg
- Updated .travis.yml configuration.
- Reorganized the documentation.
- Ongoing work to simplify the classes.
- Replaced unittest with pytest.
- Added finite_difference.py

- replaced , with .
- Reverted to coverage=4.3.4
- New attempt
- Fixed conflicting import
- Missing import of EPS
- Added missing FD_RULES = {}
- Removed pinned coverage, removed dependence on pyscaffold
- Updated .travis.yml and .appveyor.yml
- Replaced conda channel omnia with conda-forge
- Removed commented out code. Set pyqt=5 in appveyor.yml
- Updated codeclimate checks
- Dropped support for python 3.3 and 3.4. Added support for python 3.6, 3.7
- Simplified code.
- Pinned IPython==5.0 in order to make the testserver not crash.
- Added line_profiler to appveyor.yml
- Removed line_profiler from requirements.txt
- Fix issue #37: Unable to install on Python 2.7
- Added method='backward' to nd_statsmodels.py
- Skip test_profile_numdifftools_profile_hessian and TestDoProfile
- Added missing import of warnings
- Added tests for the scripts from profile_numdifftools.py, _find_default_scale.py and run_benchmark.py.
- Added reason to unittest.skipIf
- Added line_profiler to requirements.
- misssing import of warnings fixed.
- Renamed test so it comes last, because I suspect this test mess up the coverage stats.
- Reordered the tests.
- Added more tests.
- Cleaned up _find_default_scale.py
- Removed link to depsy
- Reverted: install of cython and pip install setuptools
- Disabled sonar-scanner -X for python 3.5 because it crashes.
- Reverted [options.packages.find] to exclude tests again
- Added cython and reverted to pip install setuptools
- Updated sphinx to 1.6.7
- Try to install setuptools with conda instead.
- Added hypothesis and pytest to requirements.readthedocs.txt
- Set version of setuptools==37.0
- Added algopy, statsmodels and numpy to requirements.readthedocs.txt

- Restricted sphinx in the hope that the docs will be generated.
- Removed exclusion of tests/ directory from test coverage.
- Added dependencies into setup.cfg
- Readded six as dependency
- Refactored and removed commented out code.
- Fixed a bug in the docstring example: Made sure the shape passed on to zeros is an integer.
- Fixed c_abs so it works with algopy on python 3.6.
- Fixed flaky test and made it more robust.
- Fixed bug in .travis.yml
- Refactored the taylor function into the Taylor class in order to simplify the code.
- Fixed issue #35 and added tests
- Attempt to simplify complexity
- Made doctests more robust
- Updated project path
- Changed install of algopy
- Fixed small bugs
- Updated docstrings
- Changed Example and Reference to Examples and References in docstrings to comply with numpydoc-style.
- Renamed CHANGES.rst to CHANGELOG.rst
- Renamed source path
- Hack due to a bug in algopy or changed behaviour.
- Small fix.
- Try to reduce complexity
- Reduced cognitive complexity of min_num_steps
- Simplified code in Jacobian
- Merge branch 'master' of https://github.com/pbrod/numdifftools
- Fixed issue #34 Licence clarification.
- Locked coverage=4.3.4 due to a bug in coverage that cause code-climate test-reporter to fail.
- Added script for finding default scale
- updated from sonarcube to sonarcloud
- Made sure shape is an integer.
- Refactored make_step_generator into a step property
- Issue warning message to the user when setting the order to something different than 1 or 2 in Hessian.
- Updated example in Gradient.
- Reverted –timid option to coverage because it took too long time to run.
- Reverted –pep8 option
- pep8 + added –timid in .travis.yml coverage run in order to to increase missed coverage.
- Refactored taylor to reduce complexity

- No support for python 3.3. Added python 3.6
- Fixed a small bug and updated test.
- Removed unneccasarry perenthesis. Reduced the complexity of do_profile
- Made python3 compatible
- Removed assert False
- Made unittests more forgiving.
- updated doctest in nd_scipy.py and profiletools.py install line_profiler on travis
- Made python 3 compatible
- Updated tests
- Added test_profiletools.py and capture_stdout_and_stderr in testing.py
- Optimized numdifftools.core.py for speed: fd_rules are now only computed once.
- Only keeping html docs in the distribution.
- Added doctest and updated .pylintrc and requirements.txt
- Reduced time footprint on tests in the hope that it will pass on Travis CI.
- Prefer static methods over instance methods
- Better memory handling: This fixes issue #27
- Added statsmodels to requirements.txt
- Added nd_statsmodels.py
- Simplified input
- Merge branch 'master' of https://github.com/pbrod/numdifftools
- Updated link to the documentation.

Robert Parini (4):

- Avoid RuntimeWarning in _get_logn
- Allow fd_derivative to take complex valued functions

solarjoe (1):

- doc: added nd.directionaldiff example

6.3 Version 0.9.20, Jan 11, 2017

Per A Brodtkorb (1):

- Updated the author email address in order for twine to be able to upload to pypi.

6.4 Version 0.9.19, Jan 11, 2017

Per A Brodtkorb (1):

- Updated setup.py in a attempt to get upload to pypi working again.

6.5 Version 0.9.18, Jan 11, 2017

Per A Brodtkorb (38):

- Updated setup
- Added import statements in help header examples.
- Added more rigorous tests using hypothesis.
- Forced to use wxagg backend
- Moved import of matplotlib.pyplot to main in order to avoid import error on travis.
- Added fd_derivative function
- Updated references.
- Attempt to automate sonarcube analysis
- Added testcoverage to sonarqube and codeclimate
- Simplified code
- Added .pylintrc + pep8
- Major change in api: class member variable self.f changed to self.fun
- Fixes issue #25 (Jacobian broken since 0.9.15)

6.6 Version 0.9.17, Sep 8, 2016

Andrew Fowlie (1):

- Fix ReadTheDocs link as mentioned in #21

Per A Brodtkorb (79):

- Added test for MinMaxStepgenerator
- Removed obsolete docs from core.py
- Updated appveyor.yml
- Fixed sign in inverse matrix
- Simplified code
- Added appveyor badge + synchronised info.py with README.rst.
- Removed plot in help header
- Added Programming Language :: Python :: 3.5
- Simplified code
- Renamed bicomplex to Bicomplex
- Simplified example_functions.py
- **Moved MinStepGenerator, MaxStepGeneretor and MinMaxStepGenerator to step_generators.py**
  - Unified the step generators
  - Moved step_generator tests to test_step_generators.py
  - Major simplification of step_generators.py
- Removed duplicated code + pep8
- Moved fornberg_weights to fornberg.py + added taylor and derivative

- Fixed print statement
- Replace xrange with range
- Added examples + made computation more robust.
- Made 'backward' and alias for 'reverse' in nd_algopy.py
- Expanded the tests + added test_docstrings to testing.py
- Replace string interpolation with format()
- Removed obsolete parameter
- Smaller start radius for Fornberg method
- Simplified "n" and "order" properties
- Simplified default_scale
- Removed unecessary parenthesis and code. pep8
- Fixed a bug in Dea + small refactorings.
- Added test for EpsAlg
- Avoid mutable default args and prefer static methods over instance-meth.
- Refactored to reduce cyclomatic complexity
- Changed some instance methods to static methods
- Renamed non-pythonic variable names
- Turned on xvfb (X Virtual Framebuffer) to imitate a display.
- Added extra test for Jacobian
- Replace lambda function with a def
- Removed unused import
- Added test for epsalg
- Fixed test_scalar_to_vector
- Updated test_docstrings

6.7 Version 0.9.15, May 10, 2016

Cody (2):

- Migrated % string formating
- Migrated % string formating

Per A Brodtkorb (28):

- Updated README.rst + setup.cfg
- Replaced instance methods with static methods +pep8
- Merge branch 'master' of https://github.com/pbrod/numdifftools
- Fixed a bug: replaced missing triple quote
- Added depsy badge
- added .checkignore for quantificode
- Added .codeclimate.yml
- Fixed failing tests

- Changed instance methods to static methods
- Made untyped exception handlers specific
- Replaced local function with a static method
- Simplified tests
- Removed duplicated code Simplified _Derivative._get_function_name
- exclude tests from testclimate
- Renamed test_functions.py to example_functions.py Added test_example_functions.py

Per A. Brodtkorb (2):

- Merge pull request #17 from pbrod/autofix/wrapped2_to3_fix
- Merge pull request #18 from pbrod/autofix/wrapped2_to3_fix-0

pbrod (17):

- updated conf.py
- added numpydoc>=0.5, sphinx_rtd_theme>=0.1.7 to setup_requires if sphinx
- updated setup.py
- added requirements.readthedocs.txt
- Updated README.rst with info about how to install it using conda in an anaconda package.
- updated conda install description
- Fixed number of arguments so it does not differs from overridden '_default_base_step' method
- Added codecov to .travis.yml
- Attempt to remove coverage of test-files
- Added directionaldiff function in order to calculate directional derivatives. Fixes issue #16. Also added supporting tests and examples to the documentation.
- Fixed isssue #19 multiple observations mishandled in Jacobian
- Moved rosen function into numdifftools.testing.py
- updated import of rosen function from numdifftools.testing
- Simplified code + pep8 + added TestResidue
- Updated readme.rst and replaced string interpolation with format()
- Cleaned Dea class + pep8
- Updated references for Wynn extrapolation method.

6.8 Version 0.9.14, November 10, 2015

pbrod (53):

- Updated documentation of setup.py
- Updated README.rst
- updated version
- Added more documentation
- Updated example
- Added .landscape.yml updated .coveragerc, .travis.yml

- Added coverageall to README.rst.
- updated docs/index.rst
- Removed unused code and added tests/test_extrapolation.py
- updated tests
- Added more tests
- Readded c_abs c_atan2
- Removed dependence on wheel, numpydoc>=0.5 and sphinx_rtd_theme>=0.1.7 (only needed for building documentation)
- updated conda path in .travis.yml
- added omnia channel to .travis.yml
- Added conda_recipe files Filtered out warnings in limits.py

6.9 Version 0.9.13, October 30, 2015

pbrod (21):

- Updated README.rst and CHANGES.rst.
- updated Limits.
- Made it possible to differentiate complex functions and allow zero'th order derivative.
- BUG: added missing derivative order, n to Gradient, Hessian, Jacobian.
- Made test more robust.
- Updated structure in setup according to pyscaffold version 2.4.2.
- Updated setup.cfg and deleted duplicate tests folder.
- removed unused code.
- Added appveyor.yml.
- Added required appveyor install scripts
- Fixed bug in appveyor.yml.
- added wheel to requirements.txt.
- updated appveyor.yml.
- Removed import matplotlib.

Justin Lecher (1):

- Fix min version for numpy.

kikocorreoso (1):

- fix some prints on run_benchmark.py to make it work with py3

6.10 Version 0.9.12, August 28, 2015

pbrod (12):

- Updated documentation.
- Updated version in conf.py.
- Updated CHANGES.rst.

- Reimplemented outlier detection and made it more robust.
- Added limits.py with tests.
- Updated main tests folder.
- Moved Richardson and dea3 to extrapolation.py.
- Making a new release in order to upload to pypi.

6.11 Version 0.9.11, August 27, 2015

pbrod (2):

- Fixed sphinx-build and updated docs.
- Fixed issue #9 Backward differentiation method fails with additional parameters.

6.12 Version 0.9.10, August 26, 2015

pbrod (7):

- Fixed sphinx-build and updated docs.
- Added more tests to nd_algopy.
- Dropped support for Python 2.6.

6.13 Version 0.9.4, August 26, 2015

pbrod (7):

- Fixed sphinx-build and updated docs.

6.14 Version 0.9.3, August 23, 2015

Paul Kienzle (1):

- more useful benchmark plots.

pbrod (7):

- Fixed bugs and updated docs.
- Major rewrite of the easy to use interface to Algopy.
- Added possibility to calculate n'th order derivative not just for n=1 in nd_algopy.
- Added tests to the easy to use interface to algopy.

6.15 Version 0.9.2, August 20, 2015

pbrod (3):

- Updated documentation
- Added parenthesis to a call to the print function
- Made the test less strict in order to pass the tests on Travis for python 2.6 and 3.2.

6.16 Version 0.9.1, August 20,2015

Christoph Deil (1):

- Fix Sphinx build

pbrod (47):

- **Total remake of numdifftools with slightly different call syntax.**
  - Can compute derivatives of order up to 10-14 depending on function and method used.
  - Updated documentation and tests accordingly.
  - Fixed a bug in dea3.
  - Added StepsGenerator as an replacement for the adaptive option.
  - Added bicomplex class for testing the complex step second derivative.
  - Added fornberg_weights_all for computing optimal finite difference rules in a stable way.
  - Added higher order complex step derivative methods.

6.17 Version 0.7.7, December 18, 2014

pbrod (35):

- Got travis-ci working in order to run the tests automatically.
- Fixed bugs in Dea class.
- Fixed better error estimate for the Hessian.
- Fixed tests for python 2.6.
- Adding tests as subpackage.
- Restructerd folders of numdifftools.

6.18 Version 0.7.3, December 17, 2014

pbrod (5):

- Small cosmetic fixes.
- pep8 + some refactorings.
- Simplified code by refactoring.

6.19 Version 0.6.0, February 8, 2014

pbrod (20):

- Update and rename README.md to README.rst.
- Simplified call to Derivative: removed step_fix.
- Deleted unused code.
- Simplified and Refactored. Now possible to choose step_num=1.
- Changed default step_nom from max(abs(x0), 0.2) to max(log2(abs(x0)), 0.2).
- pep8ified code and made sure that all tests pass.

6.20 Version 0.5.0, January 10, 2014

pbrod (9):

- Updated the examples in Gradient class and in info.py.
- Added test for vec2mat and docstrings + cosmetic fixes.
- Refactored code into private methods.
- Fixed issue #7: Derivative(fun)(numpy.ones((10,5)) * 2) failed.
- Made print statements compatible with python 3.

6.21 Version 0.4.0, May 5, 2012

pbrod (1)

- Fixed a bug for inf and nan values.

6.22 Version 0.3.5, May 19, 2011

pbrod (1)

- Fixed a bug for inf and nan values.

6.23 Version 0.3.4, Feb 24, 2011

pbrod (11)

- Made automatic choice for the stepsize more robust.
- Added easy to use interface to the algopy and scientificpython modules.

6.24 Version 0.3.1, May 20, 2009

pbrod (4)

- First version of numdifftools published on google.code

CHAPTER 7

Contributors

- Per A. Brodtkorb <per.andreas.brodtkorb (at) gmail.com>
- John D'Errico <woodchips (at) rochester.rr.com>

CHAPTER 8

License

```
Copyright (c) 2009-2018, Per A. Brodtkorb, John D'Errico
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this
  list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer in the documentation
  and/or other materials provided with the distribution.

* Neither the name of the copyright holders nor the names of its
  contributors may be used to endorse or promote products derived from
  this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

CHAPTER 9

Acknowledgments

The numdifftools package was originally a translation of an adaptive numerical differentiation toolbox written in Matlab by John D'Errico [DErrico2006].

Numdifftools has as of version 0.9 been extended with some of the functionality found in the statsmodels.tools.numdiff module written by Josef Perktold [Perktold2014].

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

CHAPTER 11

Bibliography

This is a collection of various bibliographic items referenced in the documentation.

Bibliography

[1] C. Brezinski and M. Redivo Zaglia (1991) “Extrapolation Methods. Theory and Practice”, North-Holland.

[1] C. Brezinski and M. Redivo Zaglia (1991) “Extrapolation Methods. Theory and Practice”, North-Holland.

[1] C. Brezinski and M. Redivo Zaglia (1991) “Extrapolation Methods. Theory and Practice”, North-Holland.

[LynessMoler1966] Lyness, J. M., Moler, C. B. (1966). Vandermonde Systems and Numerical Differentiation. *Numerische Mathematik*.

[LynessMoler1969] Lyness, J. M., Moler, C. B. (1969). Generalized Romberg Methods for Integrals of Derivatives. *Numerische Mathematik*.

[DErrico2006] D’Errico, J. R. (2006), Adaptive Robust Numerical Differentiation http://www.mathworks.com/matlabcentral/fileexchange/13490-adaptive-robust-numerical-differentiation

[Perktold2014] Perktold, J (2014), numdiff package http://statsmodels.sourceforge.net/0.6.0/_modules/statsmodels/tools/numdiff.html

[LaiCrassidisCheng2005] K.-L. Lai, J.L. Crassidis, Y. Cheng, J. Kim (2005), New complex step derivative approximations with application to second-order kalman filtering, AIAA Guidance, *Navigation and Control Conference*, San Francisco, California, August 2005, AIAA-2005-5944.

[Ridout2009] Ridout, M.S. (2009) Statistical applications of the complex-step method of numerical differentiation. *The American Statistician*, 63, 66-74

[NAG] *NAG Library*. NAG Fortran Library Document: D04AAF

Python Module Index

n

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

R

S

T