```
"""Convenience functions built on top of `grad`."""
from __future__ import absolute_import
import itertools as it

import autograd.numpy as np
from autograd.core import grad, getval
from six.moves import map


def multigrad(fun, argnums=0):
    """Takes gradients wrt multiple arguments simultaneously."""
    original_fun = fun
    def combined_arg_fun(multi_arg, *args, **kwargs):
        extra_args_list = list(args)
        for argnum_ix, arg_ix in enumerate(argnums):
            extra_args_list[arg_ix] = multi_arg[argnum_ix]
        return original_fun(*extra_args_list, **kwargs)
    gradfun = grad(combined_arg_fun, argnum=0)
    def gradfun_rearranged(*args, **kwargs):
        multi_arg = tuple([args[i] for i in argnums])
        return gradfun(multi_arg, *args, **kwargs)
    return gradfun_rearranged

def grad_and_aux(fun, argnum=0):
    """Builds a function that returns the gradient of the first output and the
    (unmodified) second output of a function that returns two outputs."""
    def grad_and_aux_fun(*args, **kwargs):
        saved_aux = []
        def return_val_save_aux(*args, **kwargs):
            val, aux = fun(*args, **kwargs)
            saved_aux.append(aux)
            return val
        gradval = grad(return_val_save_aux, argnum)(*args, **kwargs)
        return gradval, saved_aux[0]

    return grad_and_aux_fun

def value_and_grad(fun, argnum=0):
    """Returns a function that returns both value and gradient. Suitable for use
    in scipy.optimize"""
    def double_val_fun(*args, **kwargs):
```

```
        val = fun(*args, **kwargs)
        return val, getval(val)
    gradval_and_val = grad_and_aux(double_val_fun, argnum)

    def value_and_grad_fun(*args, **kwargs):
        gradval, val = gradval_and_val(*args, **kwargs)
        return val, gradval

    return value_and_grad_fun

def elementwise_grad(fun, argnum=0):
    """Like `jacobian`, but produces a function which computes just the diagonal
    of the Jacobian, and does the computation in one pass rather than in a loop.
    Note: this is only valid if the Jacobian is diagonal. Only arrays are
    currently supported."""
    def sum_output(*args, **kwargs):
        return np.sum(fun(*args, **kwargs))
    return grad(sum_output, argnum=argnum)

def jacobian(fun, argnum=0):
    """Returns a function that computes the Jacobian of `fun`. If the input to
    `fun` has shape (in1, in2, ...) and the output has shape (out1, out2, ...)
    then the Jacobian has shape (out1, out2, ..., in1, in2, ...).    Only arrays
    are currently supported."""
    # TODO: consider adding this to `autograd.grad`. We could avoid repeating
    # the forward pass every time.
    def jac_fun(*args, **kwargs):
        arg_in = args[argnum]
        output = fun(*args, **kwargs)
        assert isinstance(getval(arg_in), np.ndarray), "Must have array input"
        assert isinstance(getval(output), np.ndarray), "Must have array output"
        jac = np.zeros(output.shape + arg_in.shape)
        input_slice = (slice(None),) * len(arg_in.shape)
        for idxs in it.product(*list(map(range, output.shape))):
            scalar_fun = lambda *args, **kwargs : fun(*args, **kwargs)[idxs]
            jac[idxs + input_slice] = grad(scalar_fun, argnum=argnum)(*args, **kwargs)
        return jac
    return jac_fun

def hessian_vector_product(fun, argnum=0):
    """Builds a function that returns the exact Hessian-vector product.
```

```
    The returned function has arguments (*args, vector, **kwargs), and takes
    roughly 4x as long to evaluate as the original function."""
    fun_grad = grad(fun, argnum)
    def vector_dot_grad(*args, **kwargs):
        args, vector = args[:-1], args[-1]
        return np.dot(vector, fun_grad(*args, **kwargs))
    return grad(vector_dot_grad, argnum)    # Grad wrt original input.

def hessian(fun, argnum=0):
    """Returns a function that computes the exact Hessian.
    The Hessian is computed by calling hessian_vector_product separately for
    each row.    For a function with N inputs, this takes roughly 4N times as
    long as a single evaluation of the original function."""
    hvp = hessian_vector_product(fun, argnum)
    def hessian_fun(*args, **kwargs):
        arg_in = args[argnum]
        directions = np.eye(arg_in.size)    # axis-aligned directions.
        hvp_list = [hvp(*(args+(direction,)), **kwargs) for direction in directions]
        return np.array(hvp_list)
    return hessian_fun
```