```
from __future__ import absolute_import
from __future__ import print_function
import autograd.numpy as np
import itertools as it
from autograd.core import grad, safe_type
from copy import copy
from autograd.numpy.use_gpu_numpy import use_gpu_numpy
from autograd.container_types import ListNode, TupleNode
import six
from six.moves import map
from six.moves import range
from six.moves import zip

if use_gpu_numpy():
    garray_obj = np.garray
    array_types = (np.ndarray, garray_obj)
    EPS, RTOL, ATOL = 1e-4, 1e-2, 1e-2
else:
    garray_obj = ()
    array_types = (np.ndarray,)
    EPS, RTOL, ATOL = 1e-4, 1e-4, 1e-6

def nd(f, *args):
    unary_f = lambda x : f(*x)
    return unary_nd(unary_f, args)

def unary_nd(f, x, eps=EPS):
    if isinstance(x, array_types):
        if np.iscomplexobj(x):
            nd_grad = np.zeros(x.shape) + 0j
        elif isinstance(x, garray_obj):
            nd_grad = np.array(np.zeros(x.shape), dtype=np.gpu_float32)
        else:
            nd_grad = np.zeros(x.shape)
        for dims in it.product(*list(map(range, x.shape))):
            nd_grad[dims] = unary_nd(indexed_function(f, x, dims), x[dims])
        return nd_grad
    elif isinstance(x, tuple):
        return tuple([unary_nd(indexed_function(f, tuple(x), i), x[i])
                      for i in range(len(x))])
    elif isinstance(x, dict):
```

```
        return {k : unary_nd(indexed_function(f, x, k), v) for k, v in six.iteritems(x)}
    elif isinstance(x, list):
        return [unary_nd(indexed_function(f, x, i), v) for i, v in enumerate(x)]
    elif np.iscomplexobj(x):
        result = (f(x +        eps/2) - f(x -        eps/2)) / eps \
            - 1j*(f(x + 1j*eps/2) - f(x - 1j*eps/2)) / eps
        return type(safe_type(x))(result)
    else:
        return type(safe_type(x))((f(x + eps/2) - f(x - eps/2)) / eps)

def indexed_function(fun, arg, index):
    def partial_function(x):
        local_arg = copy(arg)
        if isinstance(local_arg, tuple):
            local_arg = local_arg[:index] + (x,) + local_arg[index+1:]
        elif isinstance(local_arg, list):
            local_arg = local_arg[:index] + [x] + local_arg[index+1:]
        else:
            local_arg[index] = x
        return fun(local_arg)
    return partial_function

def check_equivalent(A, B, rtol=RTOL, atol=ATOL):
    assert base_class(type(A)) is base_class(type(B)),\
        "Types are: {0} and {1}".format(type(A), type(B))
    if isinstance(A, (tuple, list)):
        for a, b in zip(A, B): check_equivalent(a, b)
    elif isinstance(A, dict):
        assert len(A) == len(B)
        for k in A: check_equivalent(A[k], B[k])
    else:
        if isinstance(A, np.ndarray):
            assert A.shape == B.shape, "Shapes are analytic: {0} and numeric: {1}".format(
                A.shape, B.shape)
            assert A.dtype == B.dtype, "Types are    analytic: {0} and numeric: {1}".format(
                A.dtype, B.dtype)

        assert np.allclose(A, B, rtol=rtol, atol=atol), \
            "Diffs are:\n{0}.\nanalytic is:\n{A}.\nnumeric is:\n{B}.".format(A - B, A=A, B=B)

def check_grads(fun, *args):
```

```
    if not args:
        raise Exception("No args given")
    exact = tuple([grad(fun, i)(*args) for i in range(len(args))])
    numeric = nd(fun, *args)
    check_equivalent(exact, numeric)

def to_scalar(x):
    if isinstance(x, list)    or isinstance(x, ListNode) or \
       isinstance(x, tuple) or isinstance(x, TupleNode):
        return sum([to_scalar(item) for item in x])
    return np.sum(np.real(np.sin(x)))

def quick_grad_check(fun, arg0, extra_args=(), kwargs={}, verbose=True,
                                    eps=EPS, rtol=RTOL, atol=ATOL, rs=None):
    """Checks the gradient of a function (w.r.t. to its first arg) in a random direction"""

    if verbose:
        print("Checking gradient of {0} at {1}".format(fun, arg0))

    if rs is None:
        rs = np.random.RandomState()

    random_dir = rs.standard_normal(np.shape(arg0))
    random_dir = random_dir / np.sqrt(np.sum(random_dir * random_dir))
    unary_fun = lambda x : fun(arg0 + x * random_dir, *extra_args, **kwargs)
    numeric_grad = unary_nd(unary_fun, 0.0, eps=eps)

    analytic_grad = np.sum(grad(fun)(arg0, *extra_args, **kwargs) * random_dir)

    assert np.allclose(numeric_grad, analytic_grad, rtol=rtol, atol=atol), \
        "Check failed! nd={0}, ad={1}".format(numeric_grad, analytic_grad)

    if verbose:
        print("Gradient projection OK (numeric grad: {0}, analytic grad: {1})".format(
            numeric_grad, analytic_grad))

equivalence_class = {}
for float_type in [np.float64, np.float32, np.float16]:
    equivalence_class[float_type] = float
for complex_type in [np.complex64, np.complex128]:
    equivalence_class[complex_type] = complex
```

```
def base_class(t):
    if t in equivalence_class:
        return equivalence_class[t]
    else:
        return t
```