Overloading and Extending

1. Computing with Differential Numbers
   - algorithmic differentiation
   - the class `DifferentialNumber`
   - operator overloading
2. Extending Python
   - importing C code into a Python session
   - a shared object file defines a Python module
3. Computing with Double Doubles
   - the class DoubleDouble

MCS 507 Lecture 8
Mathematical, Statistical and Scientific Software
Jan Verschelde, 8 September 2023

overloading and extending

1. Computing with Differential Numbers
   - algorithmic differentiation
   - the class `DifferentialNumber`
   - operator overloading
2. Extending Python
   - importing C code into a Python session
   - a shared object file defines a Python module
3. Computing with Double Doubles
   - the class DoubleDouble

computing with differential numbers

A differential number is a tuple $df = (f, f')$ of two doubles where f' is the evaluated derivative of f.

The operator overloading encodes the elementary derivative rules for addition, subtraction, multiplication, and division of two real numbers.

As we evaluate any expression written in these four elementary operations, we also compute its derivative.

With a Python script we illustrate *the forward mode* of algorithmic differentiation.

Reference: section (c) on *computing with differential numbers* of **Introduction to Numerical Analysis**, pages 7 to 10, by Arnold Neumaier, Cambridge 2001.

Gradients play a central role in many aspects of Machine Learning.

encoding differentiation rules

Let $df = (f, f')$ and $dg = (g, g')$,
where f and g are dependent on one single variable x.

The four basic elementary operations are $+$, $-$, $\star$ and $/$,
defined as follows:

- $df + dg = (f, f') + (g, g') = (f + g, f' + g')$
- $df - dg = (f, f') - (g, g') = (f - g, f' - g')$
- $df \star dg = (f, f') \star (g, g') = (f \star g, f' \star g + f \star g')$
- $df/dg = (f, f')/(g, g') = (f/g, (f' - (f/g) * g')/g)$

Constants and the independent variable as differential numbers:

- Any constant c (independent of x) is represented as $(c, 0)$.
- The independent variable x is represented as $(x, 1)$.

evaluating derivatives without explicit formulas

Consider $f(x) = \dfrac{(x-1)(x+3)}{x+2}$ at $x_0 = 3$. Recall the rules:

- $(f, f') \pm (g, g') = (f \pm g, f' \pm g')$,
- $(f, f') \star (g, g') = (f \star g, f' \star g + f \star g')$,
- $(f, f')/(g, g') = (f/g, (f' - (f/g) * g')/g)$.

We substitute the differential number $(3, 1)$ in f: $(f(3), f'(3)) =$

$$
\begin{aligned}
f((3,1)) &= \frac{((3,1)-1) \star ((3,1)+3)}{(3,1)+2} \\
&= \frac{(2,1) * (6,1)}{(5,1)} \\
&= \frac{(12,8)}{(5,1)} \\
&= (2.4, (8 - 2.4 \star 1)/5) \;=\; (2.4, 1.12)
\end{aligned}
$$

overloading and extending

1. Computing with Differential Numbers
   - algorithmic differentiation
   - the class `DifferentialNumber`
   - operator overloading
2. Extending Python
   - importing C code into a Python session
   - a shared object file defines a Python module
3. Computing with Double Doubles
   - the class DoubleDouble

the data attribute of the class

```
class DifferentialNumber(object):
    """
    A differential number is a tuple df = (f, f')
    of two doubles
    where f' is the evaluated derivative of f.
    """
    def __init__(self, xv, xp=1):
        """
        Initializes a differential number
        to the tuple (xv, xp).
        """
        self.val = float(xv)
        self.der = float(xp)
```

By default, `DifferentialNumber`(x) for any x is $(x, 1)$.
Note the explicit conversion to type `float`.

the string representation

A differential number (f, f') is represented as a tuple.

```
def __str__(self):
    """
    Returns the tuple representation
    of the differential number.
    """
    return str((self.val, self.der))
```

overloading and extending

1. Computing with Differential Numbers
   - algorithmic differentiation
   - the class `DifferentialNumber`
   - operator overloading
2. Extending Python
   - importing C code into a Python session
   - a shared object file defines a Python module
3. Computing with Double Doubles
   - the class DoubleDouble

adding differential numbers

The rule $df + dg = (f, f') + (g, g') = (f + g, f' + g')$ is implemented by overriding the builtin operator $+$ defined by `__add__`.

The `self` refers to the first operand,
while the second operand is given by the `other` argument.

```
def __add__(self, other):
    """
    Defines the addition of two differential numbers.
    """
    return DifferentialNumber(self.val + other.val, \
                              self.der + other.der)
```

dealing with cases like $x + 2.0$

```
def __add__(self, other):
    """
    Defines the addition of two differential numbers.
    """
    if isinstance(other, float):
        return DifferentialNumber \
                    (self.val + other, self.der)
    else:
        return DifferentialNumber \
                    (self.val + other.val, \
                     self.der + other.der)
```

For $2.0 + x$ we can do `DifferentialNumber(2.0, 0.0) + x`.

reflected addition

The reflected (or swapped) addition happens when the first operand in `+` is not a `DifferentialNumber` but an ordinary number.

```
def __radd__(self, other):
    """
    Addition when operand is not a
    DifferentialNumber as in 2.0 + x.
    """
    result = self + other
    return result
```

When `x` is a `DifferentialNumber`, then `x+y` is executed as `x.__add__(y)`, where `x` is `self` and `y` is other.

For `x + y` when `x` is not a `DifferentialNumber`,
but `y` is a `DifferentialNumber`,
then `y.__radd__(x)` is executed.

subtraction of two differential numbers

The operator $-$ is defined by the `__sub__()` method:

```
def __sub__(self, other):
    """
    Defines the subtraction
    of two differential numbers.
    """
    if isinstance(other, float):
        return DifferentialNumber \
                   (self.val - other, self.der)
    else:
        return DifferentialNumber \
                   (self.val - other.val, \
                    self.der - other.der)
```

This implements the rule $df - dg = (f, f') - (g, g') = (f - g, f' - g')$ and also deals with the case $g = (c, 0)$.

multiplying two differential numbers

The rule $(f, f') \star (g, g') = (f \star g, f' \star g + f \star g')$
is implemented by overriding the method `__mul__()`:

```
def __mul__(self, other):
    """
    Defines the product of two differential numbers.
    """
    return DifferentialNumber \
                (self.val*other.val, \
                 self.der*other.val + \
                 self.val*other.der)
```

multiplication with a constant

For a constant c: $(f, f') \star (c, 0) = (f \star c, f' \star c)$.

```
def __mul__(self, other):
    """
    Defines the product of two differential numbers.
    """
    if isinstance(other, float):
        return DifferentialNumber \
                   (self.val*other, self.der*other)
    else:
        return DifferentialNumber
                   (self.val*other.val, \
                    self.der*other.val + \
                    self.val*other.der)
```

dividing two differential numbers

We override the `__truediv__()` method to define division.

Coding the rule $(f, f')/(g, g') = (f/g, (f' - (f/g) * g')/g)$:

```
def __truediv__(self, other):
    """
    Defines the division of two differential numbers.
    """
    val = self.val/other.val
    return DifferentialNumber \
               (val, (self.der \
                      - val*other.der)/other.val)
```

dividing by a constant

For a constant c: $(f, f')/(c, 0) = (f/c, f'/c)$.

```
def __truediv__(self, other):
    """
    Defines the division of two differential numbers.
    """
    if isinstance(other, float):
        return DifferentialNumber \
                    (self.val/other, self.der/other)
    else:
        val = self.val/other.val
        return DifferentialNumber \
                    (val, (self.der \
                           - val*other.der)/other.val)
```

the main test program

```
def main():
    """
    Test on the expression ((x-1)*(x+3))/(x+2).
    """
    dfx = DifferentialNumber(3, 1)
    fun = lambda x: ((x-1.0)*(x+3.0))/(x+2.0)
    print('((x-1)*(x+3))/(x+2) at', dfx, ':')
    dfy = fun(dfx)
    print(dfy)

if __name__ == "__main__":
    main()
```

running the test

```
$ python differential_numbers.py
((x-1)*(x+3))/(x+2) at (3.0, 1.0) :
(2.4, 1.1199999999999999)
$
```

extensions of differential numbers

Exercise 1:
Extend the class `DifferentialNumber` so it works for expressions in two variables and a differential number df is (f, f_x, f_y), where f_x is the derivative with respect to the first variable and f_y is the derivative with respect to the second variable.

Exercise 2:
Extend the class `DifferentialNumber` to compute also the second derivative and a differential number df is (f, f', f'').

numerical floats

Exercise 3:

We estimate the roundoff error of an arithmetical operation $\square$, for $\square \in \{+, -, \star, /\}$ and floating-point numbers x and y as $x\widetilde{\square}y = (x\square y)(1+\epsilon)$, where $\widetilde{\square}$ is the floating-point version of $\square$ and $0 \leq \epsilon < \epsilon_{\text{mach}}$, for the machine precision ϵ_{mach}. The roundoff error is bounded by $|x\square y|\epsilon_{\text{mach}}$. For accumulated roundoff:

$$\underbrace{x_0\widetilde{\square}x_1\widetilde{\square}\cdots\widetilde{\square}x_n}_{n \text{ operations}} \approx (x_0\square x_1\square\cdots\square x_n)(1+\epsilon_1+\epsilon_2+\cdots+\epsilon_n+\cdots),$$

Truncating higher-order terms, the accumulated roundoff on n operations is estimated by $|x_0\square x_1\square\cdots\square x_n|n\epsilon_{\text{mach}}$.

Define a class `NumericalFloat`, overriding $+$, $-$, $\star$, and $/$, to also compute the upper bound on the roundoff for each operation. The accumulated roundoff is stored as an extra data attribute with each float.

overloading and extending

1. Computing with Differential Numbers
   - algorithmic differentiation
   - the class `DifferentialNumber`
   - operator overloading

2. Extending Python
   - importing C code into a Python session
   - a shared object file defines a Python module

3. Computing with Double Doubles
   - the class DoubleDouble

extending Python

Making explicit system calls to an executable program requires extra overhead of the communication through files.

C code can be wrapped for use in Python.
This requires the making of a shared object file,
which is a `.so` file on Linux, a `.dll` file on Windows,
or a `.dylib` file on Mac OS X.

Three stages:

1. Adjust the C code so that the main program becomes an interactive test function.
2. Write some "boilerplate" code.
3. Write `setup.py` and do `python setup.py build`.

Our example: take a Python string, revert its characters in a C program and pass the string back to Python.

prototypes of revertString.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int passString ( int n, char *s );
/*
 * Takes the n characters in the string s
 * and reverts the characters in the string. */

int main ( int argc, char *argv[] );
/*
 * Prompts for the length of the string
 * and a string of characters.
 * Prints the input and output of
 * the function passString. */
```

reverting characters in a string

```
int passString ( int n, char *s )
{
   int i;
   char buffer;

   for(i=0; i<n/2; i++)
   {
      buffer = s[i];
      s[i] = s[n-1-i];
      s[n-1-i] = buffer;
   }
   return 0;
}
```

the main program

```
int main ( int argc, char* argv[] )
{
   int n;
   char *s;

   printf("give length n : "); scanf("%d",&n);
   s = (char*)calloc(n,sizeof(char));
   printf("give a string : "); scanf("%s",s);
   n = strlen(s);

   printf(" input string : %s\n",s);
   passString(n,s);
   printf("output string : %s\n",s);

   return 0;
}
```

compiling and testing

To compile we use the `gcc` compiler:

```
$ gcc -o revertString revertString.c
```

At the command prompt `$` we run the program:

```
$ ./revertString
give length n : 5
give a string : hello
 input string : hello
output string : olleh
$
```

overloading and extending

1. Computing with Differential Numbers
   - algorithmic differentiation
   - the class `DifferentialNumber`
   - operator overloading
2. Extending Python
   - importing C code into a Python session
   - a shared object file defines a Python module
3. Computing with Double Doubles
   - the class DoubleDouble

the Python module `revertStrings`

```
>>> from revertStrings import passString, test
>>> test()
give length n : 5
give a string : hello
 input string : hello
output string : olleh
>>> s = 'hello there'
>>> t = passString(len(s),s)
>>> t
'ereht olleh'
>>>
```

In the first `test()` the C program is in control.

Using `passString`, the Python user is in control.

Python.h and test()

For the extension module, the file `Python.h` needs to included during the compilation stage. To locate `Python.h`, do

```
$ python3
>>> import sys
>>> sys.prefix
```

Then the test program is wrapped as

```
static PyObject *revertStrings_test
  ( PyObject *self, PyObject *args )
{
   test();
   return (PyObject*)Py_BuildValue("");
}
```

wrapping the `passString` function

```
static PyObject *revertStrings_revert
  ( PyObject *self, PyObject *args )
{
   PyObject *result;
   int n;               /* length of string */
   char *s;             /* input string */

   if(!PyArg_ParseTuple(args,"is",&n,&s)) return NULL;

   passString(n,s);

   result = (PyObject*)Py_BuildValue("s",s);

   return result;
}
```

registration table and initialization

```
static PyMethodDef revertStringsMethods[] =
{
   { "passString" , revertStrings_revert , METH_VARARGS,
     "reverting the characters in a string" } ,
   { "test" , revertStrings_test , METH_VARARGS,
     "interactive test on reverting a string" } ,
   { NULL , NULL, 0, NULL }
};

static struct PyModuleDef revertStringsModule = {
   PyModuleDef_HEAD_INIT,
   "revertStrings",
   NULL, /* no module documentation */
   -1,
   revertStringsMethods
};

PyMODINIT_FUNC
PyInit_revertStrings(void)
{
   return PyModule_Create(&revertStringsModule);
}
```

defining `setup.py`

The content of the file `setup.py`:

```
from distutils.core import setup, Extension

MOD = 'revertStrings'
setup(name=MOD,\
    ext_modules=[Extension(MOD,\
                  sources=['revertStrings.c'])])
```

Then, at the command prompt, type

```
python setup.py build
```

to make the shared object file.

To use the module, make sure the shared object is in the module path.

the module path of Python

Where does Python find its modules?

```
>>> from sys import path
>>> path
```

The `path` is a list which can be modified.

For example, to insert `/tmp` in front of the path:

```
>>> path.insert(0, '/tmp')
```

This modification is not permanent and only for the current session.

two more exercises

Exercise 4:
Make the extension module `revertStrings` on your computer.
Report the changes you had to make (if any), to make it work.

Exercise 5:
Make an extension module to compute the average
of a sequence of doubles.
The sequence of doubles is passed from a Python session
to a C function which returns the average of the sequence.

overloading and extending

1. Computing with Differential Numbers
   - algorithmic differentiation
   - the class `DifferentialNumber`
   - operator overloading
2. Extending Python
   - importing C code into a Python session
   - a shared object file defines a Python module
3. Computing with Double Doubles
   - the class DoubleDouble

double double arithmetic in Python

Recall the QDlib and the CAMPARY software of L-7.

1. Consider a C program to use the double double arithmetic of QDlib to apply Newton's method to approximate $\sqrt{2}$.
2. Define a module `doubleDouble` that exports the basic functionality of the C interface of QDlib.
3. Define the arithmetical operators, for use in Python.
   To add two double doubles `x` and `y`, instead of
   `z = doubleDouble.add(x[0],x[1],y[0],y[1])`
   we want to write `z = x + y`.

an example of the C interface to QDlib

```
int my_sqrt ( void )
{
   double n[2],x[2],y[2],z[2],e[2],a[2];
   const int max_steps = 7;
   char sqrt2[34] = "1.41421356237309504880168872420 97";
   char outsqrt2[34];
   int i;

   n[0] = 2.0; n[1] = 0.0; c_dd_copy(n,x);
   puts("\nrunning Newton's method for sqrt(2) ...");
   c_dd_read(sqrt2,y);
   printf("y hi = %22.16e  y lo = %22.16e\n",y[0],y[1]);
   c_dd_swrite(y,34,outsqrt2,34);
   printf("%s\n",outsqrt2); printf("%s\n",sqrt2);
   printf("step 0: "); c_dd_write(x); printf("\n");
   for(i=1; i <= max_steps; i++)
   {
      c_dd_mul(x,x,z);          /* z = x*x */
      c_dd_add(z,n,z);          /* z += n */
      c_dd_div(z,x,z);          /* z /= x */
      c_dd_mul_dd_d(z,0.5,z);   /* z *= 0.5 */
      printf("step %d: ",i); c_dd_write(z);
      c_dd_copy(z,x);
      c_dd_sub(x,y,e);
      c_dd_abs(e,a);
      printf("  error : "); c_dd_write(a); printf("\n");
   }
   return 0;
}
```

approximating $\sqrt{2}$ with Newton's method

```
$ /tmp/dd_test

running Newton's method for sqrt(2) ...
y hi = 1.4142135623730951e+00  y lo = -9.6672933134529147e-17
1.4142135623730950488016887242096
1.4142135623730950488016887242097
step 0: 2.0000000000000000000000000000000e+00

step 1: 1.5000000000000000000000000000000e+00
  error : 8.5786437626904951198311275790318e-02

step 2: 1.4166666666666666666666666666667e+00
  error : 2.4531042935716178649779424569892e-03

step 3: 1.4142156862745098039215686274510e+00
  error : 2.1239014147551198799032412968447e-06

step 4: 1.4142135623746899106262955788901e+00
  error : 1.5948618246068546804370127820519e-12

step 5: 1.4142135623730950488016896235025e+00
  error : 8.9929284076365753558767725231597e-25

step 6: 1.4142135623730950488016887242097e+00
  error : 1.2325951644078309459558258832544e-32

step 7: 1.4142135623730950488016887242097e+00
  error : 1.2325951644078309459558258832544e-32
```

two programming exercises

Exercise 6:
Wrap the basic arithmetical operations on double doubles
so the running of Newton's method for $\sqrt{2}$ runs in Python.

Exercise 7:
The CAMPARY library exports code that generates the arithmetical operations on any multiple double.

Design Python code that would build and wrap a library for the basic arithmetical operations to work on multiple doubles of size m, where the code treats m is a parameter.

Note that this is a *design* problem,
requiring the outline of an implementation plan, not the realization.