Numerical Programming in Python

Part II: Arithmetic and Exception Handling

Nick Maclaren

Computing Service

nmm1@cam.ac.uk, ext. 34761

February 2006

Computer Arithmetic (1)

Include some material from another course
"How Computers Handle Numbers"

Integers ($\mathbb{Z}$), reals ($\mathbb{R}$) and complex ($\mathbb{C}$)
Hardware has limited approximations

- Python's integers already covered

- Principles apply to all languages

You won't have to relearn for another one

Computer Arithmetic (2)

Most (not all) details apply to any language
Fortran, C++, Matlab, Excel etc.

To summarise the problem:
Mismatch between mathematics and computing
Not just floating–point, nor even just hardware

A lot more that will not be covered

- Just what programmers need to know

Basics of Floating-Point

Also called (leading zero) scientific notation

sign $\times$ mantissa $\times$ base$^{\text{exponent}}$

E.g. $+0.12345 \times 10^2$ = 12.345

Like fixed–point –1.0 < sign+mantissa < +1.0

Scaled by base$^{\text{exponent}}$ (10^2 in above)

Precision And Range

$1 > \text{mantissa} \geq 1/\text{base}$ (“normalised”)

P sig. digits $\Rightarrow$ relative acc. $\times\ (1 \pm \text{base}^{1-P})$

base^{1-P} is called machine epsilon

Smallest value such that $1.0 + \text{base}^{1-P} > 1.0$

Also (roughly) $-\text{maxexp} < \text{exponent} < \text{maxexp}$

$-\text{base}^{\text{maxexp}}$ to $+\text{base}^{\text{maxexp}}$ called range

Floating-Point versus Reals (1)

Floating–point effectively not deterministic
Predictable only to representation accuracy

Differences are either trivial – $\times\ (1 \pm \text{base}^{1-P})$
Or only for infinitesimally small numbers

- Regard floating–point results as ‘‘noisy’’

Not worth trying to predict exact result

Floating-Point versus Reals (2)

Fixed–point breaks many rules of real arithmetic
Floating–point breaks even more
Wrong assumptions cause wrong answers

- Key is to think floating–point, not real

Practice makes this semi–automatic
50 years of Fortran can't be wrong . . .

Seriously, that IS all you need to do

Python's Floating-Point (2)

Almost always IEEE 754 double precision
http://754r.ucbtest.org/standards/754.pdf
Binary, signed magnitude – details are messy

Double precision is 64–bit = 8 byte

- Accuracy is 2.2×10^{-16} (52/53 bits)
- Range is 2.2×10^{-308} to 1.8×10^{308}

Not quite as simple or the same on all systems

- You can ignore most of the differences

Things That Just Work

Mathematicians will recognise this . . .
It describes what you can assume in your code

A+B = B+A, A∗B = B∗A

A+0.0 = A, A∗0.0 = 0.0, A∗1.0 = A

Each A has a B = –A, such that A+B = 0.0

A $\geq$ B and B $\geq$ C means that A $\geq$ C

A $\geq$ B is equivalent to NOT B > A

Things To Watch Out For (1)

(A+B)+C may not be A+(B+C)   (ditto for '*')

(A+B)–B may not be A   (ditto for '*' and '/')

Try:

```
x = 0.001
y = (1.0+x)–1.0
print x, y, x == y

print "%.16f %.16f" % (x,y)
```

Python Output

0.001 0.001 False

0.0010000000000000 0.0009999999999999

Things To Watch Out For (2)

A+A+A may not be exactly 3.0*A

Try:

```
x = 1.0/6.0
y = x+x+x+x+x+x
print y, y == 1.0

print "%.18f %.18f" % (x, y)
```

Python Output

1.0 False

0.16666666666666657 0.99999999999999989

Things To Watch Out For (3)

Not all A have a B = 1.0/A, such that A∗B = 1.0

Try:

```
from math import e
x = e/11.0
y = 1.0/x
z = 1.0/y
print x == z

print "%.18f %.18f %.18f" % (x,y,z)
```

Python Output

False

```
0.24711652985991319 8 4.04667385288586523 0
    0.24711652985991322 5
```

Things To Watch Out For (4)

B > 0.0 may not mean A+B > A
A > 0.0 may not mean 0.5*A > 0.0

Try:

```
x = 1.0e-20
y = 5.0e-324
print 1.0+x == 1.0, y/2.0

print "%.6e %.6e" % (x,y)
```

Python Output

True 0.0

1.000000e–20 4.940656e–324

Things To Watch Out For (5)

A > B and C > D may not mean A+C > B+D

Try:

```
a = 0.75+1.0e–16
b = 0.75
c = 0.5
d = 0.5–1.0e–16
print a > b, c > d, a+c > b+d

print "%.16f %.16f %.16f %.16f" % (a,b,c,d)
print "%.16f %.16f" % (a+c,b+d)
```

Python Output

True True False

0.75000...000111 0.75000...000
0.5000...000 0.4999...999889
1.25000...000 1.25000...000

Reminder

Above are either trivially small differences
Or only for infinitesimally small numbers

- They can build up – not covered here

Remaining problem is errors and exceptions
Messiest part of IEEE 754 arithmetic

Exceptional Values (1)

$\pm$infinity represents value that overflowed
Not necessarily huge – e.g. log(exp(1000.0))

NaN (Not–a–Number) represents result of error
Typically mathematically invalid calculation

In theory, both propagate appropriately
In practice, the error state is not not reliable
Python avoids most IEEE 754 ‘‘gotchas’’

Exceptional Values (2)

Python raises exceptions to avoid ‘‘gotchas’’
Always delivers exceptional value if not

Try:

```
print 1.0/1.0e–320
print 1.0/0.0
```

But invariants may break near limits:

```
x = 5.0e–324
print 1.0/x == 2.0/x, x > 0.8*x
print x, 1.0/x, 2.0/x, 0.8*x
```

Python Output

```
inf
Traceback (most recent call last):
  File "Demos/demo_15a.py", line 2, in
                <module>
    print 1.0/0.0
ZeroDivisionError: float division

True False
4.94065645841e-324 inf inf
     4.94065645841e-324
```

Exceptional Values (3)

Be a little cautious, especially of math:
Two main trap areas that I know of:

```
from math import fmod, modf
x = float("inf")        # or 1e400
print x/1.0, x//1.0, x%1.0, modf(x), fmod(x,1.0)
```

Neither approach is actually wrong

```
print pow(0.0,x)
```

But 0.0^{∞} is mathematically invalid!

Python Output

inf nan nan (0.0, inf) nan

0.0

Conversions

Left to C – which is not good news

float("inf") etc. will usually work
Expect "±infinity", "±inf" and "nan"

Have copied an error from Java and C99:

```
x = 0.0*1.0e400
n = int(x)
print x, n
```

Python Output

nan 0

NaN Comparison

Main IEEE 754 “gotcha” in Python
NaN comparison is numerical nonsense
Everything is False except for ‘!=’

```
x = 1.0/1.0e–320
y = x/x
print y > y, y <= y, y < y, y >= y
print y == y+0.0, y == y
print y != y+0.0, y != y
```

Python Output

False False False False
False False
True True

Sanity Checking and NaNs (1)

if x != x then we have a NaN

- But it may not always detect NaNs

Don't make all tests positive checks
For example, NaN–safe code is like:

```
if speed > 0.0 and speed < 3.0e8 :
    Do the real work
else :
    panic("Speed error")
```

Sanity Checking and NaNs (2)

Following is almost as reliable (in Python):

```
if not (speed > 0.0 and speed < 3.0e8) :
    panic("Speed error")
```

- Put quite a lot of such tests in your code

Helps to pick up problems close to failure

- Check all args on input to major functions
- Consider checking results before return

Exception Handling (1)

Not strictly numeric, so will gloss over
Will briefly describe how to handle them

- Don't need to do anything in Python

If you don't handle them, will get diagnostic
Unlike most C and Fortran compilers

Or can check data is valid before operation

Exception Handling (2)

This is what happens by default:

```
array = [1,2,3,4,0,5,6,7,0,8,9]
sum = 0
for x in array :
    sum = sum+100/x

print sum
```

Python Output

```
Traceback (most recent call last):
  File "Demos/demo_19.py", line 4, in
                  <module>
    sum = sum+100/x
ZeroDivisionError: integer division or
     modulo by zero
```

Exception Handling (3)

```
array = [1,2,3,4,0,5,6,7,0,8,9]
sum = 0
errors = 0
for x in array :
    try :
        sum = sum+100/x
    except (ZeroDivisionError) :
        errors = errors+1

print sum, errors
```

Python Output

281 2

Exception Handling (4)

```
array = [1,2,3,4,0,5,6,7,0,8,9]
sum = 0
errors = 0
for x in array :
    if x != 0 :
        sum = sum+100/x
    else :
        errors = errors+1

print sum, errors
```

Python Output

281 2

Exception Practical

Use previous method to add NaN checking
Change:

```
array = [1,2,1.0e400,float("NaN"),1.0e400,  \
      3,4,0,5,float("NaN"),1.0e400,6,7,0,8,9]
```

Test that your code gets the result right
Remember that $100/\infty$ is zero

Exception Answer

```
array = [1,2,1.0e400,float("NaN"),1.0e400,  \
        3,4,0,5,float("NaN"),1.0e400,6,7,0,8,9]
sum = 0
errors = 0
for x in array :
        if x == x and x != 0 :
                sum = sum+100/x
        else :
                errors = errors+1

print sum, errors
```

Complex Exceptions

Numbers apply to IEEE double precision
You will be fairly safe if following is true:

- No infinities or NaNs in float $\Rightarrow$ complex
- abs of all args/results $\leq 10^{150}$ and $\geq 10^{-150}$
- Arc functions stay well clear of branch cuts
- Don't push pow/'$\ast\ast$' or cmath too far

- Numbers with abs $\leq 10^{-150}$ are OK IF
  your code still works if they become zero

Branch Cuts

- Arcane aspect of complex arithmetic

Most fields that use them have conventions
- Must check Python does them ‘‘right’’

May need to wrap functions to fix them up

Other fields don’t need them, or make no sense
Have lost out politically, at least for now
- Treat as errors, and check for yourself

Check Complex Values

Can assume that abs is reliable

```
if not abs(current) < 1.0e150 :
    panic("Speed error")

if not abs(value) > 1.0e–150 :
    panic("Value error")
else :
    return exp(sqrt(log(1/value)))
```

The Sordid Reasons (1)

Some implementations may ‘lose’ NaN state
C99 specifies such behaviour, too often
Python follows C in many places

You can expect system differences
You can expect changes with Python versions
You can expect errors to escape unnoticed

- This is why NaNs are not reliable

Complex exception handling isn’t, either

Complex Exceptions Summary

This is an intrinsically foul problem
IEEE 754 makes a bad situation much worse

- NO language gets this even half–right

Not even Fortran, the numeric leader

Can get spurious zeroes, infinities, NaNs
Failures often occur without an exception

- Only safe rule is to stay clear of limits

Don't rely on any language to protect you

The Sordid Reasons (2)

Why is this?

Operations like complex division are evil
http://www-uxsup.csx.cam.ac.uk/courses/...
...../Arithmetic/foils_extra.pdf
[Python complex divide is actually pretty good]

Also relies largely on C's primitives
C99 has complex as (real,imaginary) tuple
Its exception handling is completely broken

The Sordid Reasons (3)

Python CURRENTLY mostly fails safe
Some oddities, spurious NaNs and exceptions
Here are some examples of many:

```
from cmath import sqrt, atan
x = 1.0e400+0.0j
print x, x+0.0, x*1.0
print pow(x,-x), atan(x)
print sqrt(x)
```

Python Output

```
(inf+0j) (inf+0j) (inf+nanj)
(nan+nanj) (nan+nanj)
Traceback (most recent call last):
  File "Demos/demo_22.py", line 5, in
                <module>
    print sqrt(x)
OverflowError: math range error
```