Introduction to IDL

2 - Variables

Paulo Penteado
pp.penteado@gmail.com
http://www.ppenteado.net

Variable types

What is a variable?

In the old days, just a name for a position in memory:

Instead of saying

> *Store integer **2** on position **47** (of the memory)*
> *Add integer **1** to the contents of position **47***
> *Print the contents of position **47***

One could say (in pseudocode*)

```
int number_of_days
number_of_days=2
number_of_days=number_of_days+1
print number_of_days
```

Much better to use *number_of_days* than memory positions:

- The name gives a cue to the meaning of the number
- More readable and portable

*No specific language

Types

A variable is much more elaborate than just a position in memory

If I put π in memory position 435, what is in there?

Types

A variable is much more elaborate than just a position in memory

If I put π in memory position 435, what is in there?

?

Types

A variable is much more elaborate than just a position in memory

If I put π in memory position 435, what is in there?

?

Position 435

No. More like:

...10011100100100101001000000010010010000111111011011000101001...

Types

A variable is much more elaborate than just a position in memory

- A variable is a representation of a **type** in one's program.
- A **type** is an abstract concept
  - Exs: integers, reals, strings (text), complex numbers, etc.
- **Internally, these concepts do not exist:** There are no reals in memory. There are (binary) representations of them, through rules defined by the type **real**.
- **This concept includes rules about their use.** Exs:
  - Adding 2 integers is not the same as adding 2 reals. The processor uses different algorithms.
  - **3/2** is **1**, while **3.0/2.0** é **1.5**
  - **acos(2.0)** does not exist for reals, but it does for complex numbers.
  - Strings can be coded in many different ways.
  - Rules for ordering string may differ (does capitalization matter? where do numbers go in the order?)

Types

A variable can be much more complicated than a number or a string

- **Container:** stores several values, by orders, name or hierarchy
  - ➔ Exs: vector, matrix, array, list, map, dictionary, tree, etc.
- **Several values of different types, organized** - Structure
- **Reference to another variable** - pointer
- **A representation of any complicated concept - object**
  - ➔ Data, resources and ways to operate on them
  - ➔ It is an active variable ("smart"): not just a static data store, it can do operations.

Common types

Some commonly used types (*standard, built-in, primitive*):

Python: int, long, float, complex, str

IDL: int, string, float, double, byte, complex, ptr, obj

Fortran: integer, character, logical, real, double precision, complex, pointer

C, C++, Java: int, char, float, double

SQL: int, small int, bool, float, double

These types are common in all languages, but are not necessarily the same. Exs:

- Python' s ***Float*** usually corresponds to a ***double*** in other languages (double precision)
- IDL's ***int*** has 16 bits, Fortran's ***integer*** usually has 32 bits
- Fortran's ***real*** might have 32 bits or 64 bits

All of IDL's types: www.exelisvis.com/docs/idl_data_types.html

Common types – differences for similar ideas

Types are not different only in the "nature" of the idea. Exs:

- Differently sized integers
  - **Byte**: 8 bits, stores integers from **0** to **255** (2^8-1)
  - **int**: 16 bits, stores integers from -**32768** (-(2^{15})) to **32767** (2^{15}-1)
- Precision for reals: **single** and **double**:
  - `1.0+1e-8 is 1.0` (32bits, 6 or 7 significant digits)
  - `1d0+1d-8 is 1.00000001` (64 bits, ~14 significant digits)

The same function/operator is usually different with different types:

- `3/2` is `1`, while `3.0/2.0` is `1.5`
- `sqrt(-1.0)` is `-NaN`, while `sqrt(complex(-1.0))` is `(0.0,1.0)`

Types – empty type

Just as zero did not exist until modern number systems, modern languages do have empty types, with important uses:

- **To indicate something is missing**
  - Ex: A list where each element tells which observations were taken of the corresponding target. Some targets may have no observations.

- **To indicate no results**
  - Ex: functions that query some data source, to indicate that nothing was found.

- **Undefined variables / elements**
  - Indicates an input argument must be replaced by defaults
  - Indicates some output argument is not required

Examples:

- **None** (Python)
- **!null** (IDL ≥8)
- **NULL** (R, C++, Perl)
- **null** (Java)

Types – empty type

Just as zero did not exist until modern number systems, modern languages do have empty types, with important uses:

- **To indicate something is missing**
  - Ex: A list where each element tells which observations were taken of the corresponding target. Some targets may have no observations.

- **To indicate no results**
  - Ex: functions that query some data source, to indicate that nothing was found.

- **Undefined variables / elements**
  - Indicates an input argument must be replaced by defaults
  - Indicates some output argument is not required

```
IDL> x=[1,2,3,4,5]
IDL> print,where(x gt 2)
           2           3           4
IDL> print,where(x gt 7)
?
```

Types – empty type

Just as zero did not exist until modern number systems, modern languages do have empty types, with important uses:

- **To indicate something is missing**
  - Ex: A list where each element tells which observations were taken of the corresponding target. Some targets may have no observations.

- **To indicate no results**
  - Ex: functions that query some data source, to indicate that nothing was found.

- **Undefined variables / elements**
  - Indicates an input argument must be replaced by defaults
  - Indicates some output argument is not required

```
IDL> x=[1,2,3,4,5]
IDL> print,where(x gt 2,/null)
           2           3           4
IDL> print,where(x gt 7)
          -1
IDL> print,where(x gt 7,/null)
!NULL
```

Number representations and their consequences

Numbers in variables are not the same as the mathematical concept.

A variable has limited memory. Therefore, a number's digits are limited:

- The amount of different numbers that can be stored is finite.
- The numbers that are representable are predefined by the type being used.
- The precision and range numbers can have are limited.

Basic number types (integers, reals) are usually the same as the native processor number types.

They usually have fixed memory size. Exs: 8, 16, 32, 64 bits (1, 2, 4, 8 bytes).

Each bit (*binary digit*) is a memory position, which can only hold either **0** or **1**.

A type with **n** bits can only hold 2^n different values.

The most common types have **256**, **65536**, **~4.3x10^9** (4 giga, in binary sense), or **~1.8x10^{19}** (16 exa, in binary sense) different values.

Number representation - integers

There are types for positive (*unsigned*) and types for negative/positive.

Positive integers are simply the number in binary.

Ex: with 8 bits, there is room for only 0 to 255:

```
Decimal     memory representation
0           00000000
1           00000001
2           00000010
255         11111111
```

Types that can take negatives are (usually) the same, with ~half the numbers being positive, **at the beginning**, then the negatives:

Ex: with 8 bits, there is only room for -128 to +127:

```
Decimal     memory representation
0           00000000
1           00000001
127         01111111
-128        10000000
-127        10000001
-126        10000010
-2          11111110
-1          11111111
```

Integer representations – common names and sizes*

8 bits:

- byte (IDL, only positives)
- byte (Java)
- char (C, C++)
- tinyint (MySQL)

16 bits:

- int (IDL)
- short int (C++)
- short (Java)
- smallint (MySQL)

32 bits:

- integer (Fortran, R)
- long (IDL)
- int (C, C++, Java, Python,MySQL)
- long int (C, C++)

64 bits:

- long (Python)
- long64 (IDL)
- bigint (MySQL)

*In some languages, the standard does not specify which type is which size; each implementation may make different choices. The values above are the most common.

Integer representations – common names and sizes*

Literals* usually have a default type, and can be changed with modifiers:

- 9 9 of the default integer type
- 8L 8 of type *long* (32 bits)
- 25B 25 of type *byte* (8 bits)
- 12UL 12 of type *unsigned long* (64 bits)

*constants that appear *literally* inside the code

Number representation – consequences (integers)

What happens if you try to put in a variable a number that does not fit in it?

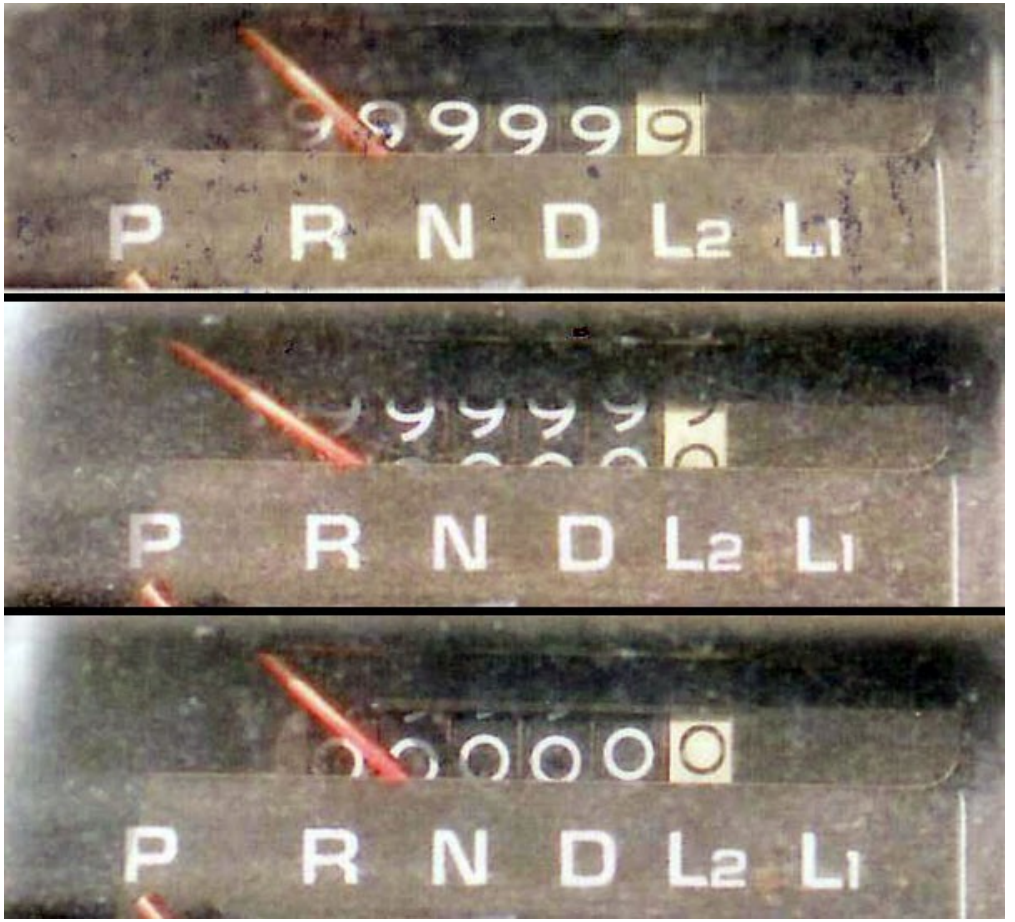
- In a *byte* type, which only holds **0** to **255**, how much is **255B+1B**? What about **0B-1B**?
- In a *short* type, which only holds **-32768** to **+32767**, how much is **-32767S-2S**?

Number representation – consequences (integers)

What happens if you try to put in a variable a number that does not fit in it?

- In a *byte* type, which only holds **0** to **255**, how much is **255B+1B**? What about **0B-1B**?
- In a *short* type, which only holds **-32768** to **+32767**, how much is **-32767S-2S**?

An overflow (or *rollover*). Like a car's odometer:



Number representation – consequences (integers)

What happens if you try to put in a variable a number that does not fit in it?

- In a *byte* type, which only holds **0** to **255**, how much is **255B+1B**? What about **0B-1B**?
- In a *short* type, which only holds **-32768** to **+32767**, how much is **-32767S-2S**?

An overflow (or *rollover*). Like a car's odometer:

```
IDL> a=255B
IDL> help,a
A               BYTE      =  255
IDL> a=a+1B
IDL> help,a
A               BYTE      =    0
IDL> print,0B-1B
 255
IDL> print,-32768S-1S
   32767
```

Internally (binary):

```
 11111111
+ 00000001
 100000000
=  00000000

 1000000000000000
-  0000000000000001
=  0111111111111111
```

Number representation – consequences (integers)

Not considering integer size is a common error:

Ex: In IDL, where default integers are type int (16 bits):

```
IDL> print,10^4
?

IDL> print,10^5
?
```

Number representation – consequences (integers)

Not considering integer size is a common error:

Ex: In IDL, where default integers are type int (16 bits):

```
IDL> print,10^4
   10000

IDL> print,10^5
  -31072
```

It is not just IDL that does this. Ex (Python):

```
In [28]: import numpy

In [29]: b=numpy.array((10,10),dtype='int16')

In [30]: print b
[10 10]

In [31]: b[0]=b[0]**4

In [32]: b[1]=b[0]*10

In [33]: print b
[ 10000 -31072]
```

Number representation – consequences (integers)

Not considering integer size is a common error:

Ex: In IDL, where default integers are type int (16 bits):

```
IDL> print,10^4
   10000

IDL> print,10^5
  -31072
```

The result for 10^5 is not wrong:

- **10^5** is larger than the largest integer that can fit in 16 bits (**32767**)
- After **32767** comes **-32768**, then **-32767**, etc.

With a larger type, there is no overflow for this number:

```
IDL> print,10L^5
   100000
```

Number representation – consequences (integers)

YouTube

Shared publicly - Dec 1, 2014

We never thought a video would be watched in numbers greater than a 32-bit integer (=2,147,483,647 views), but that was before we met PSY. "Gangnam Style" has been viewed so many times we had to upgrade to a 64-bit integer (9,223,372,036,854,775,808)!

Hover over the counter in PSY's video to see a little math magic and stay tuned for bigger and bigger numbers on YouTube.

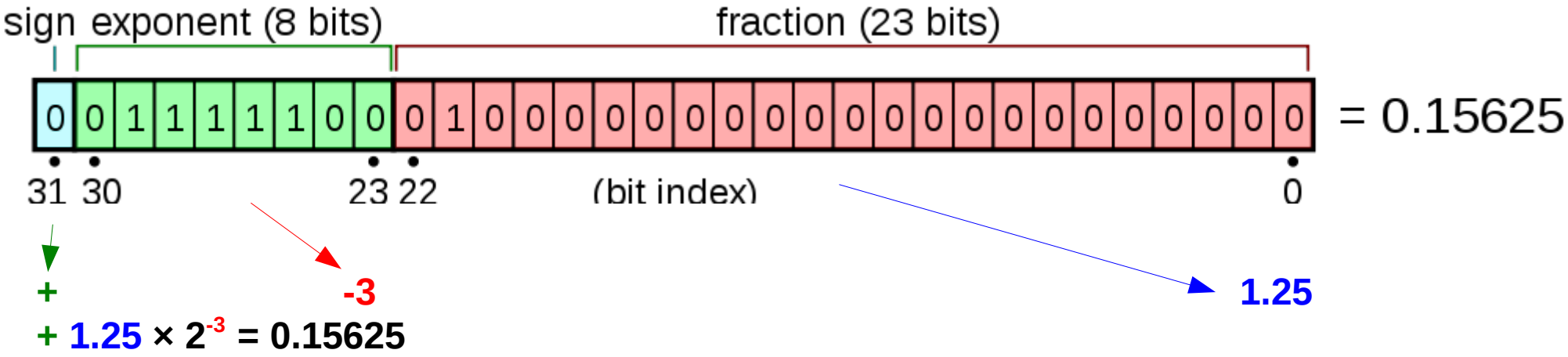
https://plus.google.com/+youtube/posts/BUXfdWqu86Q

Number representation - reals

Usual types come from the IEEE 754 standard for floating point number representation and manipulation:

- ***single precision / float / real*** (32 bits)
- ***double precision / double*** (64 bits)

Numbers are represented by a fraction signficand, and exponent and a sign, similarly to scientific notation (ex: **0.31416E1**), but with binary digits:



	Sign	Exponent	Fraction	Range	Significant digits (decimal)
Float	1 bit	8 bits	23 bits	~±10^{38}	6-9
Double	1 bit	11 bits	52 bits	~±10^{308}	15-17
Quad*	1 bit	15 bits	112 bits	~±10^{4932}	33-36

*Rarely implemented

Number representation - reals

Single precision floats **are common, but insufficient for scientific computing.**

Literals / strings such as 1.0 and 1e5 might be interpreted as *floats*.

Doubles **might be written as 1.0d0 e 1d5. But a "d" in a string usually does not change its interpretation.**

Attention to the type used in literals:

```
IDL> print,1/3
       0

IDL> print,1.0/3.0
     0.333333

IDL> print,1d0/3d0
      0.33333333

IDL> print,16d0^(1/2)
       ?
```

Number representation - reals

Single precision floats **are common, but insufficient for scientific computing.**

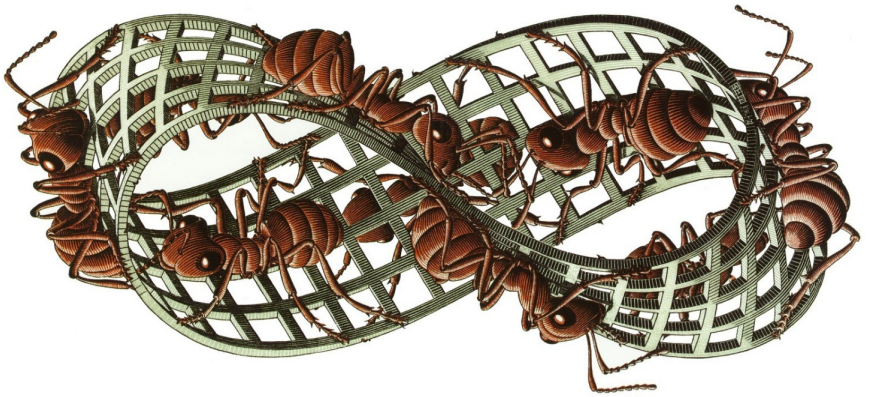
Literals / strings such as 1.0 and 1e5 might be interpreted as *floats*.

Doubles **might be written as 1.0d0 e 1d5. But a "d" in a string usually does not change its interpretation.**

Attention to the type used in literals:

```
IDL> print,1/3
       0

IDL> print,1.0/3.0
     0.333333

IDL> print,1d0/3d0
      0.33333333

IDL> print,16d0^(1/2)
       1.0000000

IDL> print,16d0^(1d0/2d0)
       4.0000000
```

Number representation: +*Infinity* and -*Infinity*

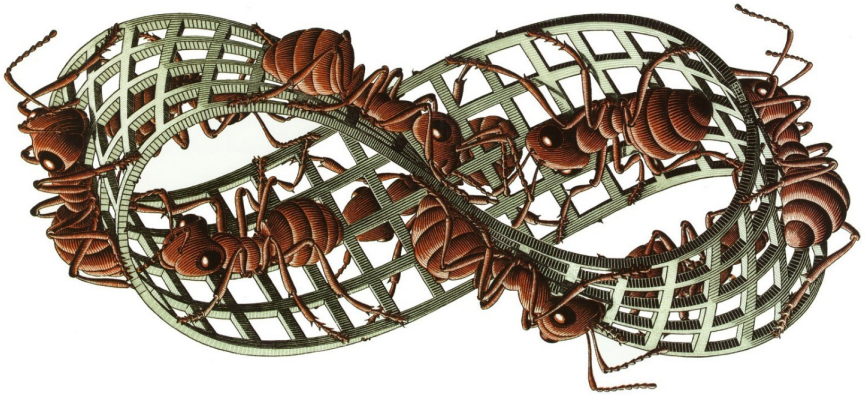


Number representation: +*Infinity* e *-Infinity*

Produced by several functions/expressions and overflows.

```
IDL> help,1.0/0.0
<Expression>    FLOAT     =           Inf
% Program caused arithmetic error: Floating divide by 0

IDL> print,exp(-!values.f_infinity)
      0.00000

IDL> help,atan(1d0/0d0)/!dpi
<Expression>    DOUBLE    =       0.50000000
% Program caused arithmetic error: Floating divide by 0

IDL> print,!values.d_infinity gt 0. ;Infinity is larger than 0
   1

IDL> print,!values.d_infinity eq !values.d_infinity ;Inf is equal to itself
   1
IDL> print,exp(89.0)
          Inf
% Program caused arithmetic error: Floating overflow
```

Just warnings, not errors.



Number representation: *+NaN* and *-NaN*

Number representation: *+NaN* and *-NaN*

Number representation: *+NaN* and *-NaN*

Not a Number

Invalid results.

```
IDL> help,0./0.
<Expression>    FLOAT     =           -NaN
% Program caused arithmetic error: Floating illegal operand

IDL> help,!values.d_infinity/!values.d_infinity
<Expression>    DOUBLE    =             -NaN
% Program caused arithmetic error: Floating illegal operand

IDL> print,sqrt(-1d0)
             NaN
% Program caused arithmetic error: Floating illegal operand

IDL> print,sqrt(complex(-1d0))
(      0.00000,      1.00000)

IDL> print,!values.d_nan gt 0d0 ;NaN is not larger than anything
   0

IDL> print,!values.d_nan le 0d0 ;NaN is not smaller than anything
   0

IDL> print,!values.f_nan eq !values.f_nan ;NaN is not equal to NaN
   0
```

Just warnings, not errors

Number representation: *+NaN* and *-NaN*

Commonly used to indicate missing or nonsense data. Ex:

- Bad pixels

- Data not taken:
  - Sky area not observed
  - Magnitude not known for the object
  - Region not included in the model

Better than the common practice of picking some value like 99, -99, -1 or 0:
It is a “special” value, depending on prior knowledge.
What if not value can be special (no number makes no sense)?

Lots of software know to ignore **NaNs *in input:***

- Leave a hole in a plot.

- Ignore them when querying for maximum, minimum, mean, etc.

On most operations with NaN the result is (properly) NaN:
- Adding a number / multiplying a number to an image should not magically turn bad pixels (NaNs) into some number.
- **NaN is not 0, 1, or any other neutral element.**

Number representation: zeros (reals)

There are two zeros (+0 and -0):

Equal in comparisons, but show the difference in limits. Exs. (IDL Python):

```
IDL> print,1d0/0d0
         Infinity
% Program caused arithmetic error: Floating divide by 0


IDL> print,1d0/(-0d0)
        -Infinity
% Program caused arithmetic error: Floating divide by 0


IDL> print,0d0 eq -0d0
   1
```

Just warnings, not errors.

Number representations – reals - consequences

Just like for integers, need to consider their **range**. Also their **precision limit**.

```
IDL> print,exp(-103.)
  1.40130e-45
% Program caused arithmetic error: Floating underflow

IDL> print,exp(-104.)
% Program caused arithmetic error: Floating underflow
      0.00000
```

Just warnings, not errors.

Number representations – reals - consequences

The digits shown when a number is printed out do not necessarily correspond to its precision.

They may show more, or less than the precision, depending on how the number was printed.

```
IDL> print,1.0d0+1d-8
       1.0000000

IDL> print,1.0d0+1d-8,format='(E22.15)'
 1.000000010000000E+00
```

Since the representation is binary, **only numbers that are rational in binary** (sums of powers of 2) can be represented exactly.

```
IDL> print,1.0,0.1,0.7
      1.00000      0.100000      0.700000

IDL> print,1.0,0.1,0.7,format='(3E19.9)'
        ?                ?                ?
```

Number representations – reals - consequences

The digits shown when a number is printed out do not necessarily correspond to its precision.

They may show more, or less than the precision, depending on how the number was printed.

```
IDL> print,1.0d0+1d-8
       1.0000000

IDL> print,1.0d0+1d-8,format='(E22.15)'
 1.000000010000000E+00
```

Since the representation is binary, **only numbers that are rational in binary** (sums of powers of 2) can be represented exactly.

```
IDL> print,1.0,0.1,0.7
      1.00000      0.100000      0.700000

IDL> print,1.0,0.1,0.7,format='(3E19.9)'
    1.000000000E+00    1.000000015E-01    6.999999881E-01
```

Number representations – reals - consequences

In computational science, single precision is usually not enough:

- **Inverting a matrix usually does not work** (it seems singular, when it is not).

- Even if the data do not have 6 digits of precision, it may take double precision, since **consecutive operations may accumulate large errors.**

- We frequently get numbers with powers beyond ±38:
  - $h=6.62\times10^{-34}$ J×s
  - $M_{\odot}=1.99\times10^{33}$g
  - $M_{\oplus}=5.97x10^{27}$g

- Julian dates take many digits (1 s is $1.16x10^{-5}$ d).
  - Ex: **2455563.024502**
  - 7 digits just to get to 1 day
  - + 5 digits to get to ~1s

- Spherical coordinates are at the limit of single precision (1” takes 7 digits in decimal degrees).

Number representations – reals - consequences

Integer types have more significant digits (but smaller ranges) than reals:

- A 32 bit unsigned integer holds exactly all numbers between **0** and **4294967295** (4 giga -1 , in binary).
- A 32 bit real can hold numbers up to ~10^{38}, but **4294967295** does not exist (it would take 10 decimal digits).

```
IDL> a=4294967295UL

IDL> print,a,format='(I0)' & print,float(a),format='(F0)'
4294967295
4294967296.000000

IDL> print,a-25, format='(I0)' & print,float(a-25),format='(F0)'
4294967270
4294967296.000000
```

Comparing the 64 bit types:

```
IDL> a=123456789012345678 90ULL

IDL> print,a,format='(I0)' & print,double(a),format='(F0)'
12345678901234567890
12345678901234567168.000000
```

Number representations – reals - consequences

Attention to comparison of real values. Exs (IDL):

```
IDL> a=dindgen(3)*!dpi
IDL> print,a
      0.0000000       3.1415927       6.2831853
IDL> print,where(a eq 3.1415927,/null)
!NULL
IDL> print,where(a eq !dpi,/null)
           1
```

Generates an array with elements 0π, 1π, 2π

No element is equal to **3.1415927**

Element **1** is equal to **!dpi**

Number representations – reals - consequences

Attention to comparison of real values.

```
IDL> a=dblarr(100000)+!dpi

IDL> b=(total(a)/n_elements(a))

IDL> print,b
       3.1415927

IDL> print,b eq !dpi
?

IDL> print,b - !dpi
?
```

a is an array with 100000 elements equal to **! dpi**

b is the sum of the elements of **a**, divided by the number of elements in **a**

Is **b** equal to **!dpi** (?)

Number representations – reals - consequences

Attention to comparison of real values.

```
IDL> a=dblarr(100000)+!dpi

IDL> b=(total(a)/n_elements(a))

IDL> print,b
       3.1415927

IDL> print,b eq !dpi
   0

IDL> print,b - !dpi
  -2.6694202e-12
```

a is an array with 100000 elements equal to **!dpi**

b is the sum of the elements of **a**, divided by the number of elements in **a**

Is **b** equal to **!dpi** (?)

No!

Usually, one can only expect reals to be equal if **one is a copy of the other, and no processing was applied to them**.

Even associativity might not be true: A+(B+C) might be different from (A+B)+C.

Results may not be identical, even with the same data, with:

- Different implementations of the same algorithm.
- Different runs of the same parallel code.

Other variable types

Can Integers / Reals / Strings do everything?

No!

What if I need to carry around a lot of information?

Ex: When processing observations, the program needs to know, for each image:

- File name
- Number of objects detected in the image
- Coordinates of each object in the image
- Image quality measurements
- Object measurements (shape, size, flux)
- Observation date/time
- Instrument
- Exposure time
- ....

Other variable types

Then carrying around variables is cumbersome.

```
for i=0,20 do begin:
   do_fancy_processing(file=file[i],nsources=nosources[i],lats=lats[i],
lons=lons[i],npoints=npoints[i],obsdate=obsdate[i],filter=filter[i],
resolution=resolution[i],temperature=temparature[i],....)
endfor
```

Filtering the data is even worse:

```
w=where(lats gt 0.)
file=file[w]
nsources=nsources[w]
lats=lats[w]
lons=lons[w]
npoints=npoints[w]
obsdate=obsdate[w]
filter=filter[w]
resolution=resolution[w]
temperature=temperature[w]
...
```

And don't you dare forget to do this to one of the 49 variables!

There must be a better way...

Other variable types - structures

A **structure*** is a compound type.

- Contains several fields
- Each field is a variable, of any type (even structure)
- Each field is identified by a name

Ex:

*Not to be confused with *data structure*, which means *a way to organize data* (i.e., arrays, lists, dictionaries, trees, etc.)

Other variable types - structures

Structure creation and use

```
IDL>observation={file:'something.fits',nsources:1701,lons:dblarr(1701),lats:d
blarr(1701),npoint:1208,nmoving:7,fwhm:0.58d0,mags:dblarr(1701),obsdate:'2014
-01-17-17:43:26.34'}
IDL> observations=replicate(observation,172)
IDL> help,observations
OBSERVATIONS    STRUCT    = -> <Anonymous> Array[172]
IDL> help,observations[0]
** Structure <de2578>, 9 tags, length=40880, data length=40870, refs=3:
   FILE            STRING    'something.fits'
   NSOURCES        INT             1701
   LONS             DOUBLE    Array[1701]
   LAS             DOUBLE    Array[1701]
   NPOINT          INT             1208
   NMOVING         INT                7
   FWHM            DOUBLE          0.58000000
   MAGS            DOUBLE    Array[1701]
   OBSDATE         STRING    '2014-01-17-17:43:26.34'

IDL> print,observations[0].nsources
    1701

IDL> help,observations.nsources
<Expression>    INT       = Array[172]

IDL> foreach observation,observations do do_fancy_processing(observation)

IDL> observations=observations[where(observations.nsources gt 0)]
```

Other variable types - structures

Common uses for structures (and arrays of structures):

- Group together a lot of variables that are related:
  - Information on observations, files, models, objects (previous example)
  - All the many inputs and outputs of a complicated program.
  - Represent tables of data from files / databases (each row is a structure)
  - Contain the data from files with multiple variables (including hierarchical files, like HDF).

Other variable types

References / Pointers

Most languages have variables that are just references to other variables.

The meaning, occurrences and uses of references vary a lot between languages.

A **reference/pointer** is only a **link**, which points to some **target**.

A target may have several references pointing to it.

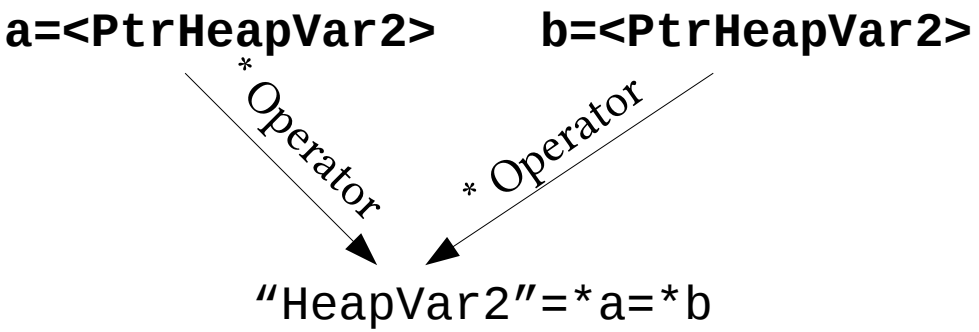
Pointers - use

A pointer's target (in IDL, a *heap variable*) can be pointed to by any number of pointers (including 0).

```
IDL> a=ptr_new(2.0) ;creates a pointer (a) and a target (float 2.0)

IDL> print,*a    ; * shows the pointer's target
      2.00000
IDL> b=ptr_new() ;creates a pointer to nothing (null)

IDL> print,b
<NullPointer>

IDL> print,ptr_valid(b) ;verifies whether b points to something
   0

IDL> b=a ;makes b point to the same target as a

IDL> print,*b ;*b=*a:
      2.00000
```

Pointers - use

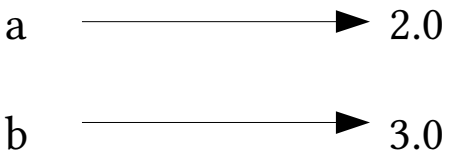
```
IDL> *a=3.0 ;changes the value of a's target, which is also b's
target

IDL> print,*b
      3.00000

IDL> *b=5.0 ;changes the value of b's target, which is also a's
target

IDL> print,*a
      5.00000

IDL> print,a,b
<PtrHeapVar2><PtrHeapVar2>
```



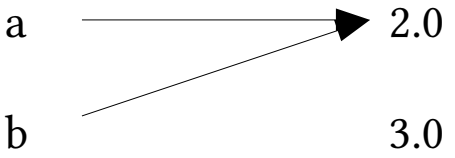
Pointers - use

A pointer that already has a target can be retargeted, by assigning to it another pointer – even retargeted to nothing, by assigning to it the null pointer.

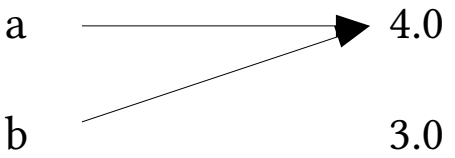
```
IDL> a=ptr_new(2.0) ;creates a pointer and its target (float 2.0)

IDL> b=ptr_new(3.0) ;creates a pointer and its target (float 3.0)

IDL> print,a,b
<PtrHeapVar1><PtrHeapVar2>
```




```
IDL> print,*a,*b
      2.00000      3.00000

IDL> b=a ;retargets b to a's target

IDL> print,a,b
<PtrHeapVar1><PtrHeapVar1>
```




```
IDL> print,*a,*b
      2.00000      2.00000

IDL> *a=*a+2.0

IDL> print,*a,*b
      4.00000      4.00000
```



Now no one points to **“HeapVar2” (3.0)**. Before IDL 8, it still exists and uses memory.

Pointers - uses

Main uses of pointers in IDL:

- To allow changing the type/dimensions of a structure field:

```
IDL>
a={temperature:19.5,wavelenghts:ptr_new(),fluxes:ptr_new()}
IDL> a.wavelenghts=ptr_new([1.3,4.9,5.8])
IDL> a.wavelenghts=ptr_new([1.3,4.9,5.8,18.2])
```

- To have an array of structures, where each element has a field with a different size. Ex: reading multiple files, where files can be of different sizes:

```
IDL> a=replicate({filename:'',nobjects:0,object_sizes:ptr_new()},10)
IDL> a[0].nobjects=7 & a[0].object_sizes=ptr_new(dblarr(7))
IDL> a[1].nobjects=9 & a[1].object_sizes=ptr_new(dblarr(9))
IDL> help,*(a[0].object_sizes)
<PtrHeapVar3>   DOUBLE    = Array[7]
IDL> help,*(a[1].object_sizes)
<PtrHeapVar4>   DOUBLE    = Array[9]
```

Pointers - uses

Main uses of pointers in IDL:

In a future class:

- Making “irregular arrays”: tables where each line has a different number or rows.
- Making arrays where each element can be of different type or dimensions.

Other variable types - objects

Objects are the next step in complexity for types:

- **Integers**
  - One value, an integer with a simple binary coding.
- **Reals**
  - One value, represented by a complicated standard (sign, exponent, fraction, special values).
- **Structures**
  - Several values (fields) in a group, of varied types, identified by names.
  - **Code must know specifically what to do with each field. If they receive a structure of a different type, or with inconsistent data, they may end up doing the wrong thing.**
- **Objects**
  - Structures (where the data is stored) + code (which operates on the object's data)
  - Data is kept inside the object. Only the object's routines have access to the data.
  - The previous types are just static data stores. They do nothing. Objects are “variables that do stuff”.

Other variable types - objects

What are objects for? Why would I want one?

- Procedural (non-object) programming:
  - There are a lot of variables around, of many different types.
  - The programmer must know what each variable means, and what to do with them.
  - The programmer must carry all associated variables around, and keep them valid. Ex:

```
IDL> help,observations[0]
** Structure <de2578>, 9 tags, length=40880, data length=40870, refs=3:
   FILE            STRING    'something.fits'
   NSOURCES        INT             1701
   LONS             DOUBLE    Array[1701]
   LATS            DOUBLE    Array[1701]
   NPOINT          INT             1208
   NMOVING         INT                7
   FWHM            DOUBLE          0.58000000
   FLUXES            DOUBLE    Array[1701]
   OBSDATE         STRING    '2014-01-17-17:43:26.34'
```

Must be kept consistent. It is up to the programmer to make sure `nsources, lats, lons and fluxes` match.

  - **The programmer calls routines, giving variables to them. These routines must know what to do with whatever variables they are given. Ex:**

```
a=mean(b)
```

What is the type of **b**? (array? list? dictionary?) Does the function (**mean**) know what to do with it?

What happens if I make up a new type? Do I have to change the function (**mean**) so that it can handle the new type?

Other variable types - objects

What are objects for? Why would I want one?

- Object-oriented programming (OOP)
  - ➔ There are few variables visible, of different types.
  - The objects contain a lot of variables inside them, but these are not visible.
  - The programmer asks the variables to do things.
  - The code that does these things lives inside the variable's type definition, so it knows how data is organized, and what to do with it.

Ex:

- Procedural programming:

```
a=mean(b)
```

This is a call to a function called **`mean`**, **which is global** will have to figure out what to do with the variable (`b`).

- Object-oriented programming:

```
a=b.mean()
```

This is a call to the function called **`mean`**, which belongs to the type of the variable (`b`), whatever that type is. If `b` is an array, array's **`mean`** will be called. If `b` is a list, list's **`mean`** will be called. There is no risk the function will get the wrong type of variable.

Objects x structures

A passive variable (structure) does nothing. The programmer must know the variables,

know what to do with them, and do it.

Objects x structures

A passive variable (structure) does nothing. The programmer must know the variables,

know what to do with them, and do it.

An active variable (object), however, contains all the necessary variables, and knows what to do with them. The programmer just has to turn it on (call the object's functions):

Objects x structures

A passive variable (structure) does nothing. The programmer must know the variables,

know what to do with them, and do it.

An active variable (object), however, contains all the necessary variables, and knows what to do with them. The programmer just has to turn it on (call the object's functions):

Objects - nomenclature

- *Objects* are variables, and are ***instances*** of ***classes***.
- The **class** is an object's **type**.
- An instance is an example of a type::
  - **2** is an **instance** of the type **integer**.
  - **1.0** is an **instance** of the type **real**
- Objects are structures, with added routines (***methods***) which operate on them.
- Classes have ***inheritance***:
  - A new class (**ClassB**) can be made by inheriting from another class (**ClassA**).
  - **ClassA** is a ***superclass*** of **ClassB**.
  - Every object of **ClassB** is also an object of **ClassA**, and inherits all of **ClassA**'s characteristics. It may add characteristics (data, methods), or change those it inherited.

Objects – why bother?

1 – Routines are subordinated to the types

When a method is called, you can be sure that only a routine that belongs to that class (or its superclasses) gets called. There is no mixing of functions made for different types.

```
IDL> l=list(5,2)
IDL> h=hash([5,2])
IDL> l.remove,1
IDL> print,l
       5
IDL> h.remove,2
IDL> print,h
5: !NULL
```

`remove` method from the `list` class: `list::remove`

`remove` method from the `hash` class: `hash::remove`

Each `remove` was **made just for one class:** `list::remove` only knows about lists.

No need for a global `remove` procedure, which must know how to handle anything that might need a **remove** (lists, hashes, sets, etc).

Objects – why bother?

2 – No need to carry many variables around over many routine calls. **The objects keep all the data inside them, and provide the data on demand.**

```
;Create the object, reading the cube file
a=pp_readcube('CM_1553510065_1_ir.cub')
;Get the core and its wavelengths
a.getproperty,core=core,wavelengths=wavs

(many code lines)

;Find out the names of the backplanes
print,a.getproperty(/backnames)
;Make a contour plot of the latitudes
c=contour(a.getsuffixbyname('LATITUDE'))

(many code lines)

;Get the band with wavelength nearest to 2.1 (in the units used in the cube)
selband=a.getbandbywavelength(2.1,wavelengths=selwavs)

(many code lines)

;Get the start time of the cube
print,a.getfromheader('START_TIME')
;"2007-084T10:00:57.286Z"
```

Much better than at the beginning having to do:

```
pp_readcube,'CM_1553510065_1_ir.cub',core=core,wavelengths=wavs,backnames=bna
mes,latitude=latitude,start_time=start_time,lines=lines,samples=samples,...
```

Objects – why bother?

3 – Operator/method *overloading:*

```
IDL> l1=list(4,9,16,25)

IDL> l2=list(36,49)

IDL> l3=l1+l2

IDL> help,l3
L3              LIST  <ID=110  NELEMENTS=6>

IDL> print,l1 eq l2
   0   0

IDL> l=list()

IDL> if (l) then print,'list is true' else print,'list is false'
list is false

IDL> l.add,9

IDL> if (l) then print,'list is true' else print,'list is false'
list is true
```

Objects – why bother?

4 – Data are kept valid.

In structures, the user can change the values in any field, to anything.

Often, a structure's fields should be kept consistent among them. There is no way to enforce this with structures.

Ex: a structure containing a data cube, with:

```
IDL> cube={nlines:100, ncolumns:200, nbands:300, data:ptr_new()}
IDL> cube.data=ptr_new(dblarr(cube.ncolumns,cube.nlines,cube.nbands))
```

So far, so good. However,

```
IDL> cube.nlines=-5
```

Does not make sense.

```
IDL> cube.data=ptr_new(dblarr(12,78,47))

IDL> help,cube
** Structure <b9591678>, 4 tags, length=12, data length=10, refs=1:
   NLINES          INT              -5
   NCOLUMNS        INT             200
   NBANDS          INT             300
   DATA            POINTER   <PtrHeapVar2>

IDL> help,*cube.data
<PtrHeapVar2>   DOUBLE    = Array[12, 78, 47]
```

Not consistent

Objects – why bother?

Objects can enforce that only valid data are used, and that **all internal data are kept valid and consistent**.

```
IDL> cube=pp_editablecube(file='CM_1503394149_1_ir_eg.cub')

IDL> print,cube.lines,cube.samples,cube.bands
          64          64         256

IDL> help,cube.core
<Expression>    FLOAT     = Array[64, 64, 256]

IDL> cube.core=dblarr(100,100,256)
```

This was not an assignment: it is a method call, equivalent to:

```
cube.setproperty,core=dblarr(100,100,256)
```

The result:

```
IDL> help,cube.core
<Expression>    DOUBLE    = Array[100, 100, 256]

IDL> print,cube.lines,cube.samples,cube.bands
         100         100         256
```

Objects in IDL 8.4

In IDL 8.4, all variable types (except object and structure) were promoted to object. They come with many useful methods and attributes:

```
IDL> x=[1,9,1,3,9,9,5,3,12]
IDL> x.uniq()
       1       3       5       9      12
IDL> x.max()
      12
IDL> x.length
           9
IDL> x.TYPENAME
INT
IDL> a='some string'
IDL> a.capwords()
Some String
IDL> a.replace('string','bananas')
some bananas
IDL> a.contains('some')
   1
```

Number representations - consequences

What is wrong with this?

```
function stefanboltzmann,j
  sigma=5.670400e-8 ;Js^-1m^-2K^-4
  return, (j/sigma)^(1/4)
end

print, stefanboltzmann(6.3200984e7)
end
?
```

Number representations - consequences

What is wrong with this?

```
function stefanboltzmann,j
  sigma=5.670400e-8 ;Js^-1m^-2K^-4
  return, (j/sigma)^(1/4)
end

print, stefanboltzmann(6.3200984e7)
end
      1.00000
```

Number representations - consequences

What is wrong with this?

```
function comparecolors,color1,color2
  return,max(abs(color1-color2))
end

print,comparecolors([200B,190B,0B],[198B,190B,0B])
print,comparecolors([200B,190B,0B],[201B,190B,0B])
end
?
?
```

Number representations - consequences

What is wrong with this?

```
function comparecolors,color1,color2
  return,max(abs(color1-color2))
end

print,comparecolors([200B,190B,0B],[198B,190B,0B])
print,comparecolors([200B,190B,0B],[201B,190B,0B])
end
   2
 255
```

Real questions, from the IDL newsgroup

1)
This may be a stupid question, but I really want to know why.
Please, see below and explain. Thanks.

```
IDL> print, 132*30
    3960
IDL> print, 132*30*10
  -25936
```

2)
There's something I can not explain to myself, so maybe someone can enlighten me?

```
IDL> print, fix(4.70*100)
        469
```

To try and find where the problem is, we tried the following lines:

```
  IDL> a = DOUBLE(42766.080001)
  IDL> print,a,FORMAT='(F24.17)'
      42766.07812500000000000
```

As you see, the number we get out isn't the same as the number we entered.

3)
I have a problem related to float-point accuracy
If I type in: 50d - 1d-9, I get 50.000000
And here lies my problem, I'm doing a numerical simulation where such an arithmetic is common place, and as a result i get a lot or errors. I know for example, that if i simply type ***print, 50d - 1d-9, format = '(f.20.10)' , i'll get:***
49.9999999990
But how can I convince IDL to do it on its own during computations?

Real questions, from the IDL newsgroup

4)
Hi guys,
*IDL> **print,((10^5)/(exp(10)*factorial(5)))***
The actual result of the above line is 0.0378332748
But when we run it in IDL we get the result as -0.011755556

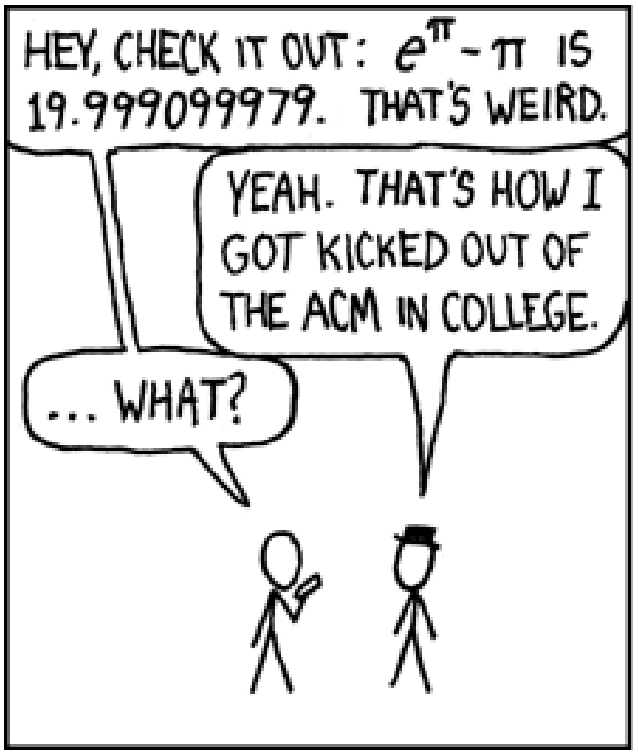
5)
*I ran into a number transformation error yesterday that is still confusing me this morning. The problem is that the number **443496.984** is being turned into the number **443496.969** from basic assignments using Float() or Double(), despite the fact that even floats should easily be able to handle a number this large (floats can handle "±10^38, with approximately six or seven decimal places of significance").*

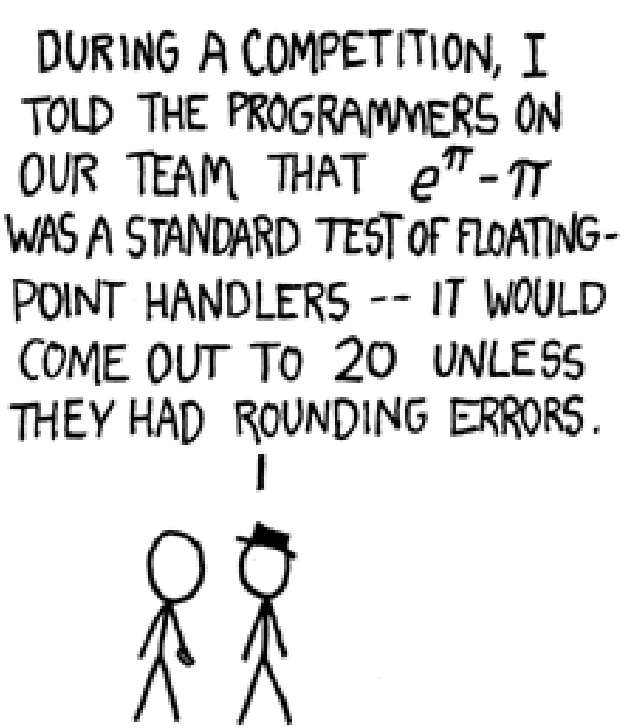
Some References

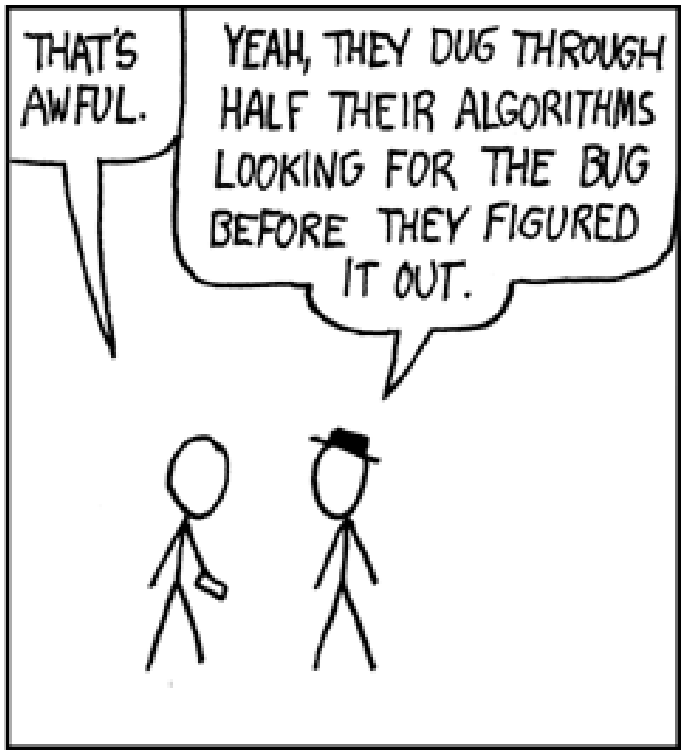
Help! The sky is falling!
http://www.dfanning.com/math_tips/sky_is_falling.html

What every programmer should know about floating-point arithmetic
or
Why don't my numbers add up?
http://floating-point-gui.de/

What every computer scientist should know about floating-point arithmetic
http://docs.sun.com/source/806-3568/ncg_goldberg.html







http://www.xkcd.org/217