Floating Point

- Generally use floating point, which is a *finite precision* system
  - introduced *rounding* errors

- standard is IEEE 754 (1985)
  - adherence made numerical code more portable and reliable

- as opposed to fixed point : point is always after the 10^0 place

```
  1234.567
     1.3
     0.001
```

- floating point : point can "float"

```
  1.234567 * 10^3
  1.3 * 10^0
  1.0 * 10^-3
```

- General floating point system

```
      b    base
      p    number of digits of precision
  [U,L]    exponent range
```

		b	p	L	U	field width
IEEE	SP	2	23(+1)=24	-126	127	(1+8+23 = 32)
IEEE	DP	2	52(+1)=53	-1022	1023	(1+11+52 = 64)

- Floating point number x

```
             (   d0 +  d1  +  d2  +   ...  + d(p-1)   )
    x  =  +- (         --     --             ------   ) * b^E
             (         b      b^2            b^(p-1)  )

      0 <=  di  <=  b-1,   i = 0,  ... , p-1  (p digits)

      L <=  E   <=  U
```

mantissa: d0d1...d(p-1)
exponent: E

```
Example 1 (1):
--------------
      b =  2
      p =  3
      L = -1
      U =  1
```

start enumerating possibilities:

```
     +-  m     E
     +- 0.00  -1  -> 0
     +- 0.00   0  -> 0
     +- 0.00  +1  -> 0
     +- 0.01  -1  -> 0.001
     +- 0.01   0  -> 0.01
     +- 0.01  +1  -> 0.1
     +- 0.10  -1  -> 0.01
     +- 0.10   0  -> 0.1
     +- 0.10  +1  -> 1.0
```

duplicates!

In general, number of possibilities

2 * b^p * (U - L + 1)

but
- lots of duplicates
- non-unique representation

Normalization
- require the leading digit to be non-zero
- so mantissa, m
  1 <= m < b
- nice because:
  - representation is now *unique*
  - don't waste digits on any leading 0's
  - for binary base, leading digit must be 1
    - so don't need to store it, just assume number is 1.d1d2..dp
      - gain an extra bit of precision!

Properties
- finite and discrete system
- finite: how many (normalized) numbers can be represented?

count them:

```
2 * (b - 1) * b^(p-1) * (U - L + 1) + 1
```

- what's the smallest (positive) normalized number? or "underflow level (UFL)"

```
1.0 ... 0 * b^L = b^L
```

- what's the biggest normalized number? or "overflow level (OFL)"

```
(b-1).(b-1) ... (b-1) * b^U
    = ( b - b^(-(p-1)) ) * b^U
    = ( 1 - b^(-p) ) * b^(U+1)
```

```
Example 1 (2):
--------------
      b =  2
      p =  3
      L = -1
      U =  1
```

- number of normalized

```
2 * (b - 1) * b^(p-1) * (U - L + 1) + 1
    = 2 * (2 - 1) * 2^(3-1) * (1 - -1 + 1) + 1
    = 2 * 1 * 4 * 3 + 1
    = 25
```

- UFL

```
b^L
    = 2^-1
    = .5
```

- OFL

(1 - b^(-p)) * b^(U+1)
  = (1 - 2^(-3)) * 2^2
  = 3.5

PICTURE of representable numbers

- note evenly spaced only for a given exponent

```
     |    |    |    |  |  |  |  |||||   |   |||||  |  |  |  |    |    |    |
-4      -3        -2          -1        0       1           2         3         4
```

Subnormals

- normalized numbers: gap between 0 and b^L
- fill in by allowing denormalized or subnormal numbers
- can make use of capacity for non-normalized numbers by allowing leading 0's
- though precision won't be full precision, since have leading 0's

```
Example 1(3):
-------------
     |    |    |    |  |  |  |  |||||||||||||||||  |  |  |  |    |    |    |
-4      -3        -2          -1        0       1           2         3         4
```

- allows 6 new numbers around 0
- new smallest number is (0.01)_2 ^ 2^-1 = (0.125)_10

- called "gradual underflow" because we gradually lose precision

- implementation: reserved value of exponent field
  - leading bit not stored

Exceptional values

- `Inf`
  - dividing finite number by 0
  - exceeding OFL
- `NaN`
  - undefined operation 0/0, Inf/Inf, 0*Inf
- implemented through reserved values of exponent field