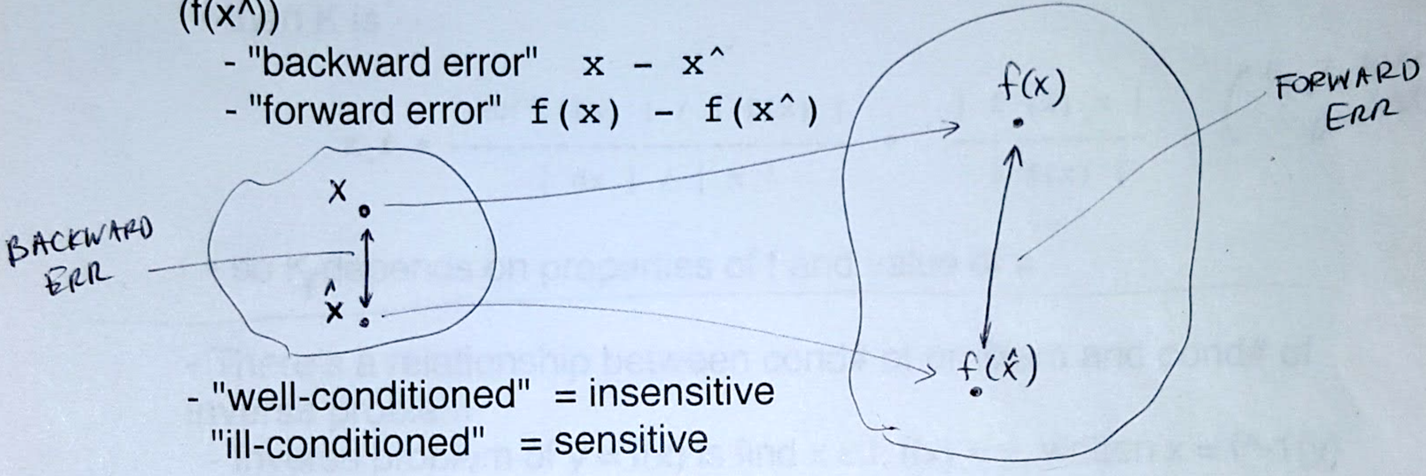
Conditioning and Stability

- Analogous concepts:
  - Conditioning of a *problem* = sensitivity to data errors
  - Stability of an *algorithm* = sensitivity to errors in computation

- Conditioning of a problem
  - problem solution is a map from input x to solution f(x)
  - PICTURE: error/uncertainty in data (x^), and error in solution (f(x^))
    - "backward error" `x - x^`
    - "forward error" `f(x) - f(x^)`



- "well-conditioned" = insensitive
  "ill-conditioned" = sensitive

- How to make this notion *quantitative*?
  - ratio of relative forward error to relative backward error

```
         rel. forward err.            | f(x^) - f(x) | / | f(x) |
    K = ------------------     =     ------------------------------
         rel. backward err.              | x^ - x | / | x |
```

  - rearranging, see that K acts like "amplification factor"

```
                   rel. forward err. =  K * rel.
backward err.
```

  - ill-conditioned ---> large K
  - well-conditioned ---> small K or K close to 1

- Usually what we can derive is an upper bound for K, so that we get bound on rel. forward err.

`rel. forward err. <= K * rel. backward err.`

f is differentiable, $\hat{x} = x + \Delta x$

`f(x + dx) - f(x) ~= dx f'(x)`

- then K is

$$K\_f = \frac{|\ dx\ f'(x)\ |\ /\ |\ f(x)\ |}{|\ dx\ |\ /\ |\ x\ |} = \frac{|\ f'(x)\ x\ |}{|\ f(x)\ |}$$

("relative derivative"

- so K_f depends on properties of f and value of x

- There's a relationship between cond# of problem and cond# of inverse problem
  - Inverse problem of y = f(x) is find x s.t. f(x) = y, written x = f^-1(y) = g(y)
  - so

$$\frac{\text{rel. forward err.}}{\text{rel. backward err.}} = \frac{|\ g(y\hat{})\ -\ g(y)\ |\ /\ |\ g(y)\ |}{|\ y\hat{}\ -\ y\ |\ /\ |\ y\ |} = \frac{|\ x\hat{}\ -\ x\ |\ /\ |\ x\ |}{|\ f(x\hat{})\ -\ f(x)\ |\ /\ |\ f(x)\ |} = \frac{1}{K}$$

Example: - Differentiable f(x), and g(y)
  - `g(f(x)) = x` by def'n
  - using chain rule, `g'(f(x)) f'(x) = 1`, so g' = 1/f'
  - so cond#

$$K\_g = \frac{|\ g'(y)\ y\ |}{|\ g(y)\ |} = \frac{|\ 1/f'(x)\ f(x)\ |}{|\ x\ |} = \frac{1}{K\_f}$$

- Lesson:
  - If K_f near 1, both f and g well-conditioned
  - If K_f big or small, either K_f or K_g ill-conditioned

- Side note: Above is "relative cond#". If seeing x* s.t. f(x*) = 0, use "absolute cond#", defined analogously:

$$K = \frac{\text{abs. forward err.}}{\text{abs. backward err.}} = \frac{|\, f(x\hat{}) - f(x) \,|}{|\, x\hat{} - x \,|}$$

- for differentiable f

$$K\_f\_abs = \frac{|\, dx\, f'(x) \,|}{|\, dx \,|} = |f'(x)|$$

- Example: f(x) = sqrt(x) = x^{1/2}
  f'(x) = 1/2 * x^{-1/2} = 1/(2f(x))

$f(x) = x^{1/2}$

$\boxed{\text{well-conditioned}}$

$$K\_f = \frac{|\, f'(x)\; x \,|}{|\, f(x) \,|} = \frac{|\, x \,|}{|\, 2\, f(x) * f(x) \,|} = \frac{1}{2}$$

- inverse problem: find x s.t. y = sqrt(x), or x = g(y) = y^2

K_g = 2

- Conclusion: both f and g are well-conditioned

- Example: f(x) = tan(x)
  f'(x) = sec^2(x) = 1 + tan^2(x)

$\boxed{\text{ill-conditioned}}$

$$K\_f = \frac{|\, x(1+\tan\hat{}2(x)) \,|}{|\, \tan(x) \,|} = \text{very large near } x = pi/2$$

- at x = 1.57079, K_f = 2.48275 * 10^5 (sensitive!!), so that

tan(1.57079) ~= 1.58058 * 10^5, tan(1.57078) ~= 6.12490 *

10^4

check:
((1.58058 * 10^5 - 6.12490 * 10^4) / (6.12490 * 10^4)) / ((1.57079 - 1.57078)/1.57078) = K_f [illegible] ✓

- g(y) = arctan(y), at y = 1.58058 * 10^5

K_g ~= 4.0278 * 10^{-6} (insensitive!!)

Stability and Accuracy

- An algorithm is *stable* if its results are insensitive to perturbations during computation
  - e.g., truncation, discretization, and rounding errors

- Or, using backward error, algorithm is stable if
  - effect of perturbations during computation is no worse than effect of small amount of data error
  - *however* if problem is ill-conditioned, effect of small data error is really bad!
    - won't get a good (accurate) solution even with a stable algorithm

- So
  - well-conditioned problem + unstable algorithm => inaccurate solution
  - ill-conditioned problem + stable algorithm => inaccurate solution
  - well-conditioned problem + stable algorithm => accurate solution

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!

$$\overset{s}{\begin{bmatrix} \pm \end{bmatrix}} \times \overset{m}{\begin{bmatrix} 0.00 \\ 0.01 \\ 0.10 \\ 0.11 \\ 1.00 \\ 1.01 \\ 1.10 \\ 1.11 \end{bmatrix}} \times \overset{E}{\begin{bmatrix} -1 \\ 0 \\ +1 \end{bmatrix}}$$

In general, number of possibilities

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

$2 \times 8 \times 3 = 48$

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!

$b=10$ $(10.00)_{10} = (10)$

$b=2$ $(10.00)_2 = (2)$

$b=3$ $(10.00)_3 = (3)$