01.03 rootfinding : error

1 forward and backward

example 07

find root of $f(x) = x^3 - 2x^2 + \frac{4}{3}x - \frac{8}{27}$ to within six significant digits using bisection.

$f(0) \cdot f(1) = (-\frac{8}{27}) \cdot (\frac{1}{27}) < 0$ so IVT guarantees a solution in $[0, 1]$ and example 02 calculates 20 steps as sufficient for six significant digits. it is also easy to eyeball that

$$f(\tfrac{2}{3}) = \frac{8}{27} - 2\left(\frac{4}{9}\right) + \left(\frac{4}{3}\right)\left(\frac{2}{3}\right) - \left(\frac{8}{27}\right) = 0. \ \checkmark$$

however...

code, example 07

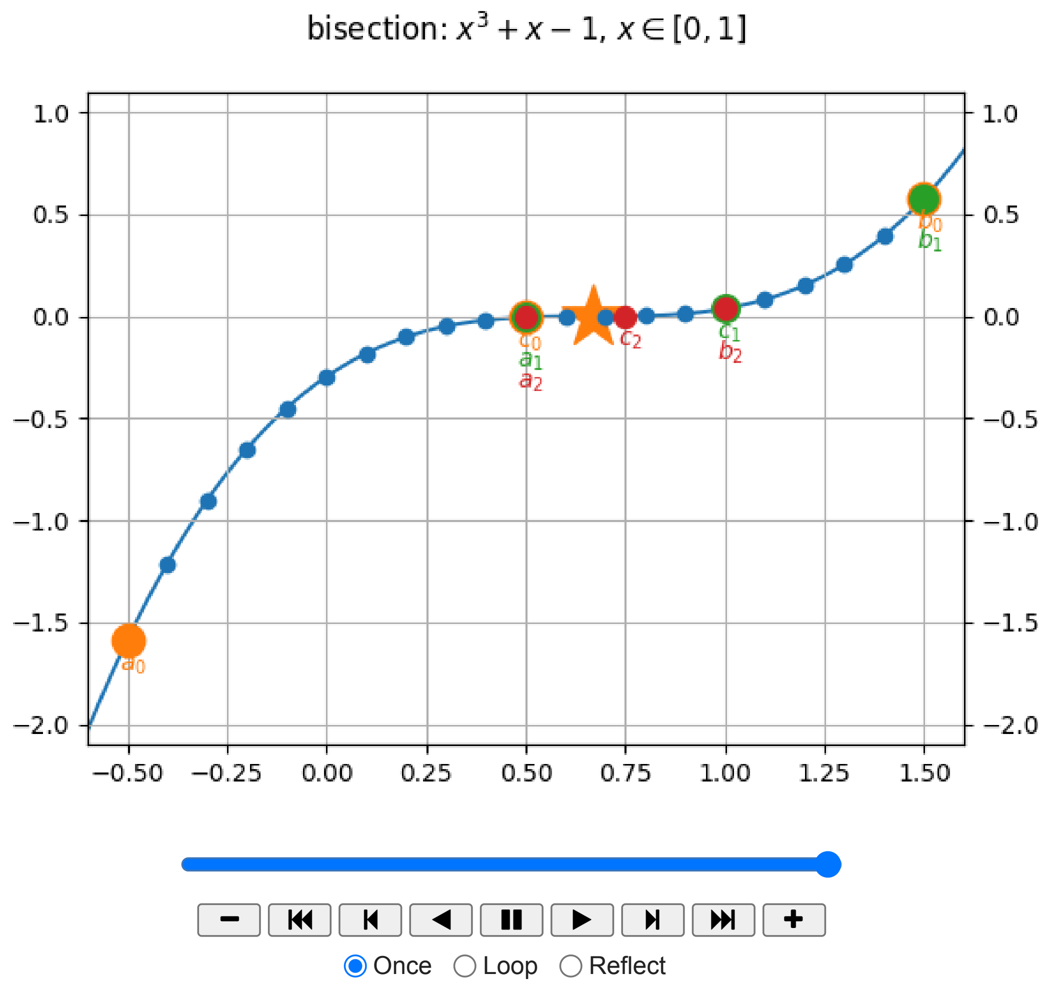
```
# algorithm, expanded for lecture 01.01

def bisect_expanded(f,ab,tol,all=False,workspace=False):

```

```
# example 07 updates first code of lecture 01.01

if __name__ == "__main__":

```

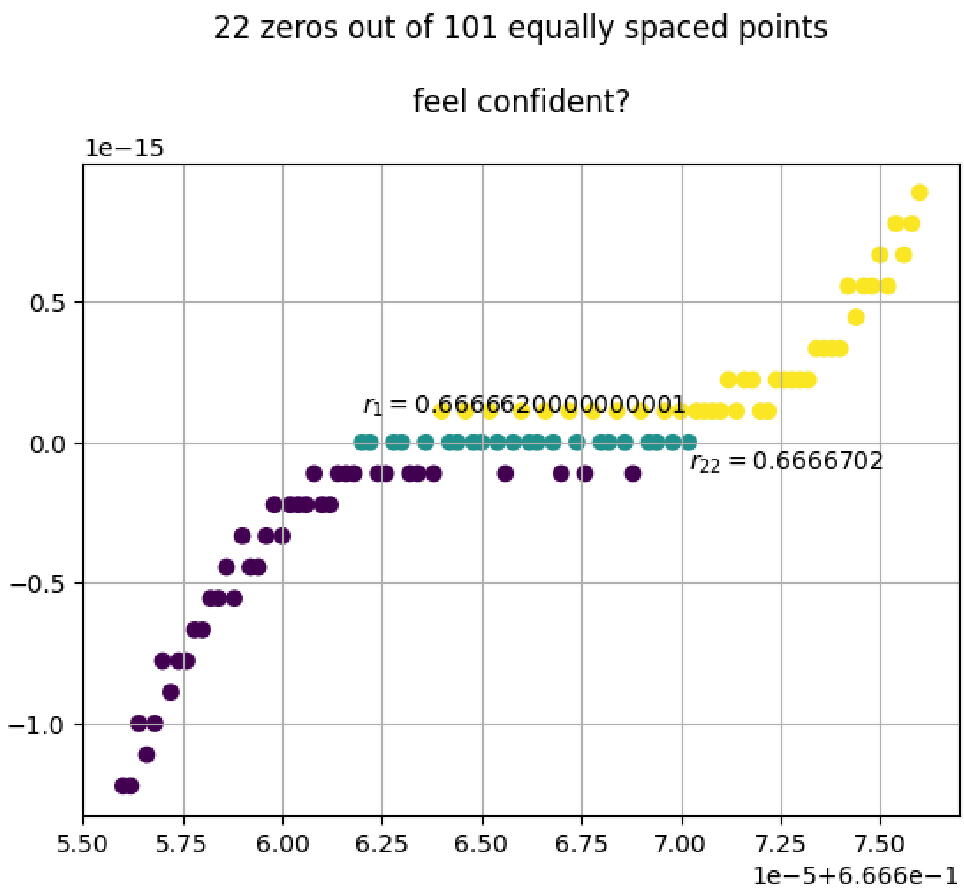
```
|   i |             a |          f(a) |            b |         f(b) |            c |          f(c) |  ±
|-----|---------------|---------------|--------------|--------------|--------------|---------------|----
| 000 |   -0.50000000 |   -1.58796296 |   1.50000000 |   0.57870370 |   0.50000000 |   -0.00462963 |  -01
| 001 |    0.50000000 |   -0.00462963 |   1.50000000 |   0.57870370 |   1.00000000 |    0.03703704 |  001
| 002 |    0.50000000 |   -0.00462963 |   1.00000000 |   0.03703704 |   0.75000000 |    0.00057870 |  001
| 003 |    0.50000000 |   -0.00462963 |   0.75000000 |   0.00057870 |   0.62500000 |   -0.00007234 |  -01
| 004 |    0.62500000 |   -0.00007234 |   0.75000000 |   0.00057870 |   0.68750000 |    0.00000904 |  001
| 005 |    0.62500000 |   -0.00007234 |   0.68750000 |   0.00000904 |   0.65625000 |   -0.00000113 |  -01
| 006 |    0.65625000 |   -0.00000113 |   0.68750000 |   0.00000904 |   0.67187500 |    0.00000014 |  001
| 007 |    0.65625000 |   -0.00000113 |   0.67187500 |   0.00000014 |   0.66406250 |   -0.00000002 |  -01
| 008 |    0.66406250 |   -0.00000002 |   0.67187500 |   0.00000014 |   0.66796875 |    0.00000000 |  001
| 009 |    0.66406250 |   -0.00000002 |   0.66796875 |   0.00000000 |   0.66601562 |   -0.00000000 |  -01
| 010 |    0.66601562 |   -0.00000000 |   0.66796875 |   0.00000000 |   0.66699219 |    0.00000000 |  001
| 011 |    0.66601562 |   -0.00000000 |   0.66699219 |   0.00000000 |   0.66650391 |   -0.00000000 |  -01
| 012 |    0.66650391 |   -0.00000000 |   0.66699219 |   0.00000000 |   0.66674805 |    0.00000000 |  001
| 013 |    0.66650391 |   -0.00000000 |   0.66674805 |   0.00000000 |   0.66662598 |   -0.00000000 |  -01
| 014 |    0.66662598 |   -0.00000000 |   0.66674805 |   0.00000000 |   0.66668701 |    0.00000000 |  001
| 015 |    0.66662598 |   -0.00000000 |   0.66668701 |   0.00000000 |   0.66665649 |   -0.00000000 |  -01
| 016 |    0.66665649 |   -0.00000000 |   0.66668701 |   0.00000000 |   0.66667175 |    0.00000000 |  001
| 017 |    0.66665649 |   -0.00000000 |   0.66667175 |   0.00000000 |   0.66666412 |    0.00000000 |  000
```

so the algorithm stops at 16 iterations and at less than six significant digits bc it thinks that last $c_{16} = 0.66666412$ as $f(r) = 0$.

```
ani # why not; its already writ
```




```
# example 07, multiple roots in the wrong way

if __name__ == "__main__":…
```



its kinda ok that theres a range of zeros but there are positive and negative almost-zeros among them. 👀

definition 08

assume function f has root r such that $f(r) = 0$. assume x_a is an approximation to r. for root-finding, **backward error** is $|f(x_a)|$ and **forward error** is $|r - x_a|$.

ie, wrt solution for a problem.

problem (type)	input	method	output
evaluation	x	f(x)	y=f(x)=?
root-finding	f(x)=0	solver	r=x=?

example 07, continued

its forward error is approximately 10^{-5}; however, its backward error is near $\epsilon_{\text{mach}} \approx 2.2 \times 10^{-16}$. ie, ϵ_{mach} limits backward error which limits forward error.

also,

$$f(x) = x^3 - 2x^2 + \tfrac{4}{3}x - \tfrac{8}{27} = \left(x - \tfrac{2}{3}\right)^3.$$

definition 09

r is a root of differentiable function f. ie, $f(r) = 0$. then if $0 = f'(r) = f''(r) = \cdots = f^{(m-1)}(r)$ but $f^{(m)}(r) \neq 0$ then f has **root** of **multiplicity** m at r. if $m > 1$, then r is a **multiple root**; if $m = 1$, then r is a **simple root**.

eg, $f(x) = x^2$ has $r = 0$ and $m = 2$ bc $f(0) = 0$, $f'(0) = 2(0) = 0$ and $f''(0) = 2 \neq 0$. likewise, $f(x) = x^3$ has triple root at $r = 0$ and $f(x) = x^m$ has multiplicity m root $r = 0$.

example 07, continued

example 07 has triple root at $r = \frac{2}{3}$. bc example 07 is flat near its triple root, there exists disparity between its backward and forward errors for nearby approximate solutions.

example 08

function $f(x) = sin\, x - x$ has triple root at $r = 0$. calculate forward and backward error at approximate root $x_c = 0.001$.

$$\begin{aligned} f(0) &= sin\, 0 - 0 = 0 \\ f'(0) &= cos\, 0 - 1 = 0 \\ f''(0) &= -sin\, 0 - 0 = 0 \\ f'''(0) &= -cos\, 0 = -1 \neq 0. \end{aligned}$$

$$\Rightarrow \quad r = 0 \text{ is a triple root. } \checkmark$$

- forward error: $|r - x_c| = |0 - 0.001| = 0.001$;
- backward error: $|f(x_c)| = |sin(0.001) - 0.001| \approx 1.6667 \times 10^{-10}$.

usw

forward and backward error are important to stopping criteria for equation solvers. which one is more appropriate? it depends. if using bisection to solve for a root, both errors are observable; if using FPI, only backward error is available bc the true root is typically unknown. also functions are flat near a multiple root. usw.

2 wilkinson polynomial

wilkinson polynomial.

$$W(x) = (x-1)(x-2)\dots(x-20)$$

$$\begin{aligned} &= x^{20} - 210x^{19} + 20615x^{18} - 1256850x^{17} + 53327946x^{16} - 1672280820x^{15} \\ &\quad + 40171771630x^{14} - 756111184500x^{13} + 11310276995381x^{12} \\ &\quad - 135585182899530x^{11} + 1307535010540395x^{10} - 10142299865511450x^{9} \\ &\quad + 63030812099294896x^{8} - 311333643161390640x^{7} \\ &\quad + 1206647803780373360x^{6} - 3599979517947607200x^{5} \\ &\quad + 8037811822645051776x^{4} - 12870931245150988800x^{3} \\ &\quad + 13803759753640704000x^{2} - 8752948036761600000x \\ &\quad + 2432902008176640000. \end{aligned}$$

code, wilkinson, expanded

```
# algorithm, basic # from lecture 01.02

def fpi(g,x,tol=1e-8,max_iter=100):
  count = 0
```

```
# tried some stuffs

if __name__ == "__main__":

  import numpy as np
  import scipy as sp

  w = lambda x: pow(x,20) - 210*pow(x,19) + 20615*pow(x,18) \
      - 1256850*pow(x,17) + 53327946*pow(x,16) - 1672280820*pow(x,15) \
      + 40171771630*pow(x,14) - 756111184500*pow(x,13) + 11310276995381*pow(x,12) \
      - 135585182899530*pow(x,11) + 1307535010540395*pow(x,10) - 10142299865511450*pow(x,9) \
      + 63030812099294896*pow(x,8) - 311333643161390640*pow(x,7) \
      + 1206647803780373360*pow(x,6) - 3599979517947607200*pow(x,5) \
      + 8037811822645051776*pow(x,4) - 12870931245150988800*pow(x,3) \
      + 13803759753640704000*pow(x,2) - 8752948036761600000*x \
      + 2432902008176640000

  print(f"root: whatever\n\n{sp.optimize.root(w,16)}\n")
  print(f"root_scalar: bisection\n\n{sp.optimize.root_scalar(w,bracket=(15.9,16.1),method='bisect')}\n")
  print(f"root_scalar: newton ~ FPI\n\n{sp.optimize.root_scalar(w,x0=16,method='newton')}\n")

  gw = lambda x: (pow(x,20) - 210*pow(x,19) + 20615*pow(x,18) \
      - 1256850*pow(x,17) + 53327946*pow(x,16) - 1672280820*pow(x,15) \
      + 40171771630*pow(x,14) - 756111184500*pow(x,13) + 11310276995381*pow(x,12) \
      - 135585182899530*pow(x,11) + 1307535010540395*pow(x,10) - 10142299865511450*pow(x,9) \
      + 63030812099294896*pow(x,8) - 311333643161390640*pow(x,7) \
      + 1206647803780373360*pow(x,6) - 3599979517947607200*pow(x,5) \
      + 8037811822645051776*pow(x,4) - 12870931245150988800*pow(x,3) \
      + 13803759753640704000*pow(x,2) + 2432902008176640000) / 8752948036761600000

  for x0 in np.arange(15.1,16.,0.1): # lolwut
    root = fpi(gw,x0)
    print(f"FPI({x0}): {root}") # lol

```

```
root: whatever

 message: The solution converged.
 success: True
  status: 1
     fun: [-6.029e+09]
       x: [ 1.600e+01]
    nfev: 3
    fjac: [[-1.000e+00]]
       r: [ 2.474e+17]
     qtf: [ 6.029e+09]

root_scalar: bisection

      converged: True
           flag: converged
 function_calls: 39
     iterations: 37
           root: 16.003582954652668

root_scalar: newton ~ FPI

      converged: False
           flag: convergence error
 function_calls: 100
     iterations: 50
           root: 16.000000171386752

FPI(15.1): 15.099987116761817
FPI(15.2): 15.199972574526747
FPI(15.299999999999999): 15.29995870626266
FPI(15.399999999999999): 15.399944957194926
FPI(15.499999999999998): 15.499936025285603
FPI(15.599999999999998): 15.599931663603696
FPI(15.699999999999998): 15.699935685635873
FPI(15.799999999999997): 15.799947960421843
FPI(15.899999999999997): 15.89996854919591
FPI(15.999999999999996): 16.00000008250424
```

code, bonus

```
# just bc

import numpy as np
import numpy.polynomial.polynomial as npp

p = npp.Polynomial.fromroots(range(1,21)) # [1,21) = [1,20]
print(f"roots, reconstitute:\n\n{p.roots()}\n")
print(f"coeffs, calculated:\n\n{p.coef}\n")
print(f"wilkinson, expanded:\n\n{p}\n")
print(f"w(16): {p(16)} = LOLS!!")

```

```
roots, reconstitute:

[ 1.          2.          3.          4.00000002  4.9999996   6.00000521
  6.99995561  8.00026686  8.99881078 10.00409792 10.98921356 12.02307993
 12.96334362 14.04714444 14.95450431 16.03179803 16.98312518 18.00576725
 18.99876967 20.00011801]

coeffs, calculated:
```

```
[ 2.43290201e+18 -8.75294804e+18  1.38037598e+19 -1.28709312e+19
  8.03781182e+18 -3.59997952e+18  1.20664780e+18 -3.11333643e+17
  6.30308121e+16 -1.01422999e+16  1.30753501e+15 -1.35585183e+14
  1.13102770e+13 -7.56111184e+11  4.01717716e+10 -1.67228082e+09
  5.33279460e+07 -1.25685000e+06  2.06150000e+04 -2.10000000e+02
  1.00000000e+00]

wilkinson, expanded:

2.43290201e+18 - (8.75294804e+18)·x + (1.38037598e+19)·x² -
(1.28709312e+19)·x³ + (8.03781182e+18)·x⁴ - (3.59997952e+18)·x⁵ +
(1.2066478e+18)·x⁶ - (3.11333643e+17)·x⁷ + (6.30308121e+16)·x⁸ -
(1.01422999e+16)·x⁹ + (1.30753501e+15)·x¹⁰ - (1.35585183e+14)·x¹¹ +
(1.1310277e+13)·x¹² - (7.56111184e+11)·x¹³ + (4.01717716e+10)·x¹⁴ -
(1.67228082e+09)·x¹⁵ + 53327946.0·x¹⁶ - 1256850.0·x¹⁷ + 20615.0·x¹⁸ -
210.0·x¹⁹ + 1.0·x²⁰

w(16): -5433720832.0 = LOLS!!
```

wilkinson, factored

well. no problems? mostly?

- wilkinson comparison sympy-style @ cmu
- sympy

3 sensitivity

a problem is **sensitive** if small errors in input lead to large errors in output.

this error magnification wrt rootfinding, consider a small change in the problem -- ie, the equation for which to find the root.

assume problem is to find root r to $f(x) = 0$ and small change $\epsilon\, g(x)$ made to input such that

$$f(r + \Delta r) + \epsilon\, g(r + \Delta r) = 0$$

where ϵ is small and Δr is change in root. expand f, g in degree-one taylor polynomials,

$$f(r) + (\Delta r) f'(r) + \epsilon\, g(r) + \epsilon\, (\Delta r) g'(r) + \underbrace{\cancel{\mathcal{O}((\Delta r)^2)}}_{\text{meh}} = 0$$

$$(\Delta r)(f'(r) + \epsilon\, g'(r)) \approx \cancelto{0}{-f(r)} - \epsilon\, g(r)$$

$$\Rightarrow \quad \Delta r \approx \frac{-\epsilon\, g(r)}{f'(r) + \epsilon\, g'(r)}, \quad \epsilon \ll f'(r) \text{ and } f'(r) \neq 0$$

$$\approx \epsilon\, \frac{g(r)}{f'(r)}.$$

sensitivity formula for roots

assume r is root of $f(x)$ and $r + \Delta r$ is a root of $f(x) + \epsilon\, g(x)$. then

$$\Delta r \approx -\frac{\epsilon\, g(r)}{f'(r)}, \quad \epsilon \ll f'(r).$$

example 09

estimate largest root of $P(x) = (x-1)(x-2)(x-3)(x-4)(x-5)(x-6) - 10^{-6} x^7$.

$$\begin{aligned} \Rightarrow f(x) &= (x-1)(x-2)(x-3)(x-4)(x-5)(x-6) \\ \epsilon &= -10^{-6} \\ g(x) &= x^7. \end{aligned}$$

with only $f(x)$, largest root is $r = 6$. with $\epsilon\, g(x)$,

$$\Delta r \approx -\frac{\epsilon\, 6^7}{5!} = -2332.8\, \epsilon.$$

ie, input errors of relative size ϵ in $f(x)$ are magnified over $2000\times$ into output root. so largest root of $P(x) \Rightarrow r + \Delta r = 6 - 2332.8\, \epsilon = 6.0023328$.

```python
# example 09

if __name__ == "__main__":

  import scipy as sp

  f = lambda x: (x-1)*(x-2)*(x-3)*(x-4)*(x-5)*(x-6) - 1e-6*pow(x,7)

  soln = sp.optimize.root(f,x0=6)
  print(soln)
  print(f"\nroot: {soln.x[0]}")

```

```
 message: The solution converged.
 success: True
  status: 1
     fun: [-3.236e-14]
       x: [ 6.002e+00]
    nfev: 6
    fjac: [[-1.000e+00]]
       r: [-1.210e+02]
     qtf: [ 2.255e-10]

root: 6.00232675474645
```

ie, an error in the sixth digit of the problem data caused an error in the third digit of the answer. ie, three decimal digits were lost due to that factor 2332.8.

for a general algorithm that produces an approximation x_c,

$$\textbf{error magnification factor} = \frac{\text{relative forward error}}{\text{relative backward error}}.$$

example 09, continued

$$\text{error magnification factor} = \left|\frac{\Delta r/r}{\epsilon\, g(r)/g(r)}\right| = \left|\frac{-\epsilon\, g(r)}{f'(r)}\cdot\frac{1}{r}\cdot\frac{1}{\epsilon}\right| = \frac{|g(r)|}{|r f'(r)|} = \frac{6^7}{6\cdot 5!} = 388.8.$$

example 10

use the sensitivity formula for roots to investigate the effect of changes in the x^{15} term of the wilkinson polynomial on the root $r = 16$ find error magnification factor for this problem.

define perturbed function $W_\epsilon(x) = W(x) + \epsilon\, g(x)$, where $g(x) = -1672280820\, x^{15}$. $W'(16) = 15!4!$ and

$$\Delta r \approx \frac{\epsilon \cdot 1672280820 \cdot 16^{15}}{15!4!} \approx 6.1432 \times 10^{13}\, \epsilon \approx 6.1432 \times 10^{13}\, \epsilon_{\text{mach}}$$

$$\approx (6.1432 \times 10^{13})(\pm 2.22 \times 10^{-16}) \approx \pm 0.0136$$

$$\text{error magnification:} \quad \frac{g(r)|}{|rf'(r)|} = \frac{16^{15} \cdot 1672280802}{15!4!16} \approx 3.8 \times 10^{12}.$$

ie, an error magnification factor of 10^{12} results in loss of 12 of 16 bits of operating percision from input to output. ie, instead of $r = 16$, it is $r + \Delta r = \underline{16.01}4....$

usw

the **condition number** *of a problem* is defined to be the maximum error magnification over all input changes. a problem with high condition number is **ill-conditioned** and a problem with a condition number near one is **well-conditioned**.