APC 523 Problem Set 1

Ming Lyu (吕铭)
Department of Electrical Engineering, Princeton University

March 6, 2019

1 Error in (symmetric) rounding vs. chopping

The binary form of $x : \{b_n = 0, 1\}$ means (with exponent $2^{q-1} < e < 2^{q-1} - 1$):

$$x = \pm 2^e \sum_{n=0}^{\infty} b_n 2^{-n} \tag{1.1}$$

and given $b_0 = 1$, we have:

$$\left|\frac{x - \mathrm{rd}(x)}{x}\right| \leq \frac{|x - \mathrm{rd}(x)|}{2^e b_0} = \begin{cases} \left|\sum_{n=p+1}^{\infty} b_n 2^{-n}\right| & b_{p+1} = 0, \mathrm{rd}(x) = \mathrm{tr}(x) \\ \left|2^{-p+e} - \sum_{n=p+1}^{\infty} b_n 2^{-n}\right| & b_{p+1} = 1, \mathrm{rd}(x) = \mathrm{tr}(x) + 2^{-p+e} \end{cases} \tag{1.2}$$

$$= \left|\sum_{n=p+1}^{\infty} b_n^* 2^{-n}\right| \leq \left|\sum_{n=p+1}^{\infty} 2^{-n}\right| = 2^{-p} \tag{1.3}$$

$$\text{where} \quad b_n^* = \begin{cases} b_n & b_{p+1} = 0 \\ 1 - b_n & b_{p+1} = 1 \end{cases} \tag{1.4}$$

2 An accurate implementation of e^x

See the Python script attached.

(a) Each terms are: 0.10000e1, 0.55000e1, 0.15125e2, 0.27730e2, 0.38129e2, 0.41942e2, 0.38447e2, 0.30208e2, 0.20768e2, 0.12692e2, 0.69805e1, 0.34902e1, 0.15997e1, 0.67679e0, 0.26588e0, 0.97484e-1, 0.33510e-1, 0.10842e-1, 0.33128e-2, 0.95898e-3, 0.26372e-3, 0.69070e-4, 0.17269e-4, 0.41297e-5, 0.94638e-6, 0.20821e-6, 0.44043e-7, 0.89715e-8, 0.17623e-8, 0.33422e-9, 0.61274e-10

(b) Final result is 0.24471e3=244.71 ($k \geq 17$), while double precision $\mathrm{e}^{5.5} \approx 244.69193$, with relative error 7.4e-5

(c) Final result is 0.24470e3=244.70, with relative error 3.3e-5

(d) $\mathrm{e}^{-5.5} \approx 4.0868 \times 10^{-3}$=0.40868e-2

(i) Converge to 0.38363e-2 when $k = 25$, error 0.06

(ii) Converge to 0.40000e-2 when $k = 20$, error 0.02

(iii) Converge to 0 when $k = 18$, error 1.0

(iv) Converge to -0.10000e-1 when $k = 18$, error 3.4

(iii) and (iv) converge quicker but have significant error. This is because the algorithm magnifies subtraction error by having two largest number (sum of all positive terms and sum of all negative terms) subtract each other.

(e)

(i) Pair one positive and one negative term, add these pair first and then sum over all pairs: the result turns out to be similar to (d.i)

(ii) Calculate $1/e^{5.5}$, error 4.2e-5

```python
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from functools import reduce
from math import log10, ceil, factorial, exp
from operator import mul, add
class FiniteDicimal(object):
    def __init__(self, n=5, num=None):
        self.N = n
        self.digits = [0]*n
        self.exp = 0
        self.sign = 1
        if num:
            self.set(num)

    def value(self):
        return self.sign * 10**(self.exp-1) * reduce(
            lambda x, y: x/10.0+y, self.digits)

    def set(self, num):
        if num < 0:
            self.sign = -1
            num = -num
        self.exp = ceil(log10(num) + 1e-10)
        num = round(num/10**(self.exp-self.N))
        for n in range(self.N):
            self.digits[n] = num % 10
            num = num//10

    def __add__(self, num):
        return FiniteDicimal(self.N, self.value() + num.value())

    def __sub__(self, num):
        return FiniteDicimal(self.N, self.value() - num.value())

    def __mul__(self, num):
        return FiniteDicimal(self.N, self.value() * num.value())

    def __truediv__(self, num):
        return FiniteDicimal(self.N, self.value() / num.value())

    def __pow__(self, exp):
```

```
        if exp == 0:
            return FiniteDicimal(num=1)
        return self.__pow__(exp-1)*self

    def __eq__(self, num):
        return self.exp == num.exp and self.digits == num.digits

    def __neg__(self):
        return FiniteDicimal(self.N, -self.value())

    def __repr__(self):
        res = "0." if self.sign > 0 else "-0."
        for n in reversed(self.digits):
            res += str(n)
        return res + "e" + str(self.exp)

def fct(num):
    if num == 0:
        return FiniteDicimal(num=1)
    return FiniteDicimal(num=num)*fct(num-1)

if __name__ == "__main__":
    x = FiniteDicimal(num=5.5)
    terms = [x**n/fct(n) for n in range(31)]
    print("(a) ", terms)

    print("(b)")
    tot = FiniteDicimal()
    for n, s in enumerate(terms):
        tot += s
        print(n, tot, end="\t")
    trueValue = exp(5.5)
    print("Double:", trueValue, "Error", tot.value()/trueValue-1)

    tot = FiniteDicimal()
    for s in reversed(terms):
        tot += s
    trueValue = exp(5.5)
    print("(c)", tot, "Error", tot.value()/trueValue-1)
    e55 = tot

    trueValue = exp(-5.5)
    print("(d) Double:", trueValue)
    terms = [-t if n%2 else t for n, t in enumerate(terms)]
    tot = FiniteDicimal()
    print("(d.i)")
    for n, s in enumerate(terms):
        totnew = tot + s
        if totnew == tot:
            print("\nConverge at k=%d"%(n-1))
```

```
            break
        tot = totnew
        print(n, tot, end="\t")
    print(tot, "Error", abs(tot.value()/trueValue-1))

    print("(d.ii)")
    tot = FiniteDicimal()
    for n in range(1, 31):
        totnew = reduce(add, reversed(terms[:n]))
        if totnew == tot:
            print("\nConverge at k=%d"%(n-1))
            break
        tot = totnew
        print(n, tot, end="\t")
    print(tot, "Error", abs(tot.value()/trueValue-1))

    print("(d.iii)")
    tot = FiniteDicimal()
    for n in range(2, 31):
        totpositive = reduce(add, terms[0:n:2])
        totnegtive = reduce(add, terms[1:n:2])
        totnew = totpositive + totnegtive
        if totnew == tot:
            print("\nConverge at k=%d"%(n-1))
            break
        tot = totnew
        print(n, tot, end="\t")
    print(tot, "Error", abs(tot.value()/trueValue-1))

    print("(d.iv)")
    tot = FiniteDicimal()
    for n in range(2, 31):
        totpositive = reduce(add, reversed(terms[0:n:2]))
        totnegtive = reduce(add, reversed(terms[1:n:2]))
        totnew = totpositive + totnegtive
        if totnew == tot:
            print("\nConverge at k=%d"%(n-1))
            break
        tot = totnew
        print(n, tot, end="\t")
    print(tot, "Error", abs(tot.value()/trueValue-1))

    print("(e.i)")
    tot = FiniteDicimal()
    pairs = [terms[i] + terms[i+1] for i in range(0, 30, 2)]
    for n in range(1, len(pairs)):
        totnew = reduce(add, reversed(pairs[:n]))
        if totnew == tot:
            print("\nConverge at k=%d"%(2*(n-1)))
            break
```

```
        tot = totnew
        print(n*2, tot, end="\t")
    print(tot, "Error", abs(tot.value()/trueValue-1))

    print("(e.ii)")
    tot = FiniteDicimal(num=1)/e55
    print(tot, "Error", abs(tot.value()/trueValue-1))
```

3 Error propagation in exponentiation

Neglect difference between $\varepsilon_{\ln}$, ε_{mul}, $\varepsilon_{\exp}$... and using an ε for all single step machine error.

(a) Assuming x and n are perfect machine number. For repeated multiplication:

$$\begin{aligned}\text{fl}(x^n) &= \text{fl}(x^{n-1}) \cdot x(1+\varepsilon) = \text{fl}(x^{n-2}) \cdot x^2(1+\varepsilon)^2 && (3.1)\\ &= \cdots = x^n(1+\varepsilon)^n = x^n(1+n\varepsilon) && (3.2)\end{aligned}$$

For $\mathrm{e}^{n\ln x}$,

$$\begin{aligned}\text{fl}(\mathrm{e}^{n\ln x}) &= \exp\left[\text{fl}(n\ln x)\right](1+\varepsilon) && (3.3)\\ &= \exp\left[n\,\text{fl}(\ln x)(1+\varepsilon)\right](1+\varepsilon) && (3.4)\\ &= \exp\left[n\ln x(1+2\varepsilon)\right](1+\varepsilon) && (3.5)\\ &= \mathrm{e}^{n\ln x}\exp\left[2\varepsilon n\ln x\right](1+\varepsilon) && (3.6)\\ &= x^n\left[1+(1+|2n\ln x|)\varepsilon\right] && (3.7)\end{aligned}$$

- When $x > \sqrt{\mathrm{e}}$ or $x < 1/\sqrt{\mathrm{e}}$, $1+|2n\ln x| > 1+n > n$, meaning repeated multiplication is better.
- When $1/\sqrt{\mathrm{e}} < x < \sqrt{\mathrm{e}}$, repeated multiplication is better when $n \lessapprox 1/(1-|2\ln x|)$. For $n \in \mathbb{N}$, this is non-trivial only when $x = \sqrt{\mathrm{e}} - \delta$ or $x = 1/\sqrt{e} + \delta$ with $\delta \ll 1$.

In conclusion, repeated multiplication is better in most cases, except for when n is large and $x \lessapprox \sqrt{\mathrm{e}}$ or $x \gtrapprox 1/\sqrt{\mathrm{e}}$.

(b) Apart from the error term in Eq. (3.7), there's

$$\begin{aligned}x^{a(1+\varepsilon_a)} &= x^a x^{a\varepsilon_a} = x^a\left[1+(a\ln x)\varepsilon_a\right] && (3.8)\\ \left[x(1+\varepsilon_x)\right]^a &= x^a(1+\varepsilon_x)^a = x^a(1+a\varepsilon_x) && (3.9)\end{aligned}$$

$a\varepsilon$ error can become an issue when a is large.

4 Conditioning

Assuming $f_A(x) = f(x)(1+\epsilon)$ with machine error ϵ.

(a)

$$\begin{aligned}(\text{cond } f)(x) &\equiv \frac{|\Delta f/f|}{|\Delta x/x|} = \left|\frac{\Delta f}{\Delta x}\frac{x}{f}\right| && (4.1)\\ &= \left|\frac{f'x}{f}\right| = \frac{x}{e^x-1} \leq 1 && (4.2)\end{aligned}$$

where the last inequality holds for all $x \in [0,1]$

(b) $\mathrm{fl}(e^{-x}) = e^{-x}(1+\epsilon)$, $f_A(x) \equiv \mathrm{fl}\left[f(x)\right] = \left[1 - \mathrm{fl}(e^{-x})\right](1+\epsilon)$, so

$$f_A(x) = f(x) + |(1 - \mathrm{e}^{-x})\epsilon| + |\mathrm{e}^{-x}\epsilon| = f(x) + \epsilon \tag{4.3}$$

for $f(x_A) = f_A(x)$, $x_A - x = (f_A - f)/f' = \epsilon \mathrm{e}^x$, therefore

$$(\mathrm{cond}\, A)(x) \equiv \frac{|x_A - x|}{|x|}\frac{1}{\epsilon} = \frac{e^x}{x} \geq \mathrm{e} > 1 \tag{4.4}$$

(c) The poor conditioning for cond A comes from finite $\Delta x = |x_A - x|$ when $x \to 0$.

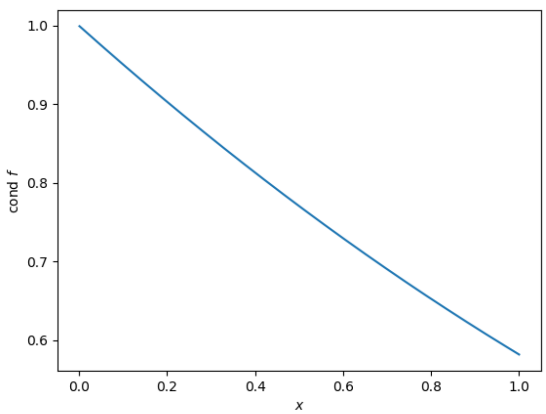

Figure 1: $(\mathrm{cond}\, f)(x)$ on $[0,1]$

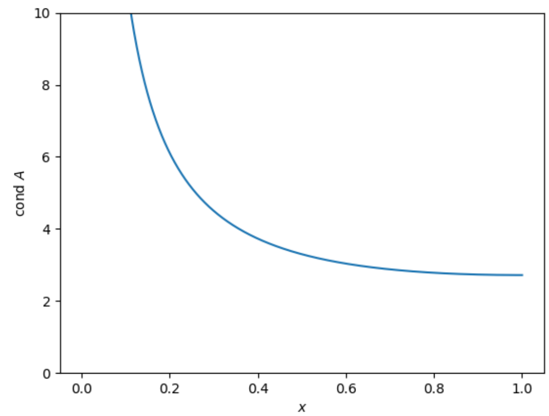

Figure 2: $(\mathrm{cond}\, A)(x)$ on $[0,1]$

(d) n bit lost when $|\Delta y/y| < 2^n \epsilon^*$, where ϵ^* is the floating number rounding error, while

$$\left|\frac{\Delta y}{y}\right| \approx (\mathrm{cond}\, f)(x)\left[\left|\frac{x^* - x}{x}\right| + (\mathrm{cond}\, A)(x^*)\epsilon\right] = \left(\frac{x}{\mathrm{e}^x - 1} + \frac{\epsilon}{\epsilon^*}\frac{1}{1 - \mathrm{e}^{-x}}\right)\epsilon^* \tag{4.5}$$

Assuming exp implementation is ideal so that $\epsilon/\epsilon^* = 1$, $|\Delta y/y| < 2^n \epsilon^*$ gives:

$$\frac{x + \mathrm{e}^x}{\mathrm{e}^x - 1} < 2^n \tag{4.6}$$

- For $n = 1$, $x > 1.146$ meaning for all $x \in [0,1]$ there will be at least 1 bit of significance lost.
- For $n = 2$, $x > 0.378$
- For $n = 3$, $x > 0.152$
- For $n = 4$, $x > 0.069$

(e) (d) is equivalent to requiring forward error to be less than $2^n \epsilon^*$.
(f) for small x, using the series

$$f(x) = \sum_{n=1}^{\infty} \frac{(-x)^n}{n!} \tag{4.7}$$

The algorithmic result:

$$f_A(x) = \mathrm{fl}\left[\sum_{n=1}^{N} \frac{(-x)^n}{n!}\right] \tag{4.8}$$
$$= \sum_{n=1}^{N} \frac{(-x)^n}{n!}(1+n\epsilon) \tag{4.9}$$
$$= \sum_{n=1}^{N} \frac{(-x)^n}{n!} + \epsilon \sum_{n=1}^{N} \frac{|(-x)^n|}{(n-1)!} \tag{4.10}$$
$$= f(x) + x\mathrm{e}^x\epsilon + \mathcal{O}(x^N) \tag{4.11}$$

With large N, $\mathcal{O}(x^N) \sim \epsilon^*$. So for $f(x_A) = f_A(x)$, there is:

$$x_A - x = \frac{x\mathrm{e}^x\epsilon}{f'} = x\mathrm{e}^{2x}\epsilon \tag{4.12}$$
$$(\mathrm{cond}\, A)(x) \equiv \frac{|x_A - x|}{|x|}\frac{1}{\epsilon} = \mathrm{e}^{2x} \tag{4.13}$$

is bounded in $x \in [0,1]$ and performs better in small x regime.

5 Limits in $\mathbb{R}(p,q)$

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
import numpy as np

ns = 10**np.arange(20)
esequence = (1+1/ns)**ns
error = np.abs(np.diff(esequence)/esequence[1:])
nstop = np.argmax(error < 10**(-12))
print("n-stop:", ns[nstop], "value:", esequence[nstop])
print(esequence[:nstop+1])
#   print(error[:nstop+2])
```

The above code gives output:

```
n-stop: 10000000000000 value: 2.716110034086901
[2.          2.59374246 2.70481383 2.71692393 2.71814593 2.71826824
 2.71828047 2.71828169 2.7182818  2.71828205 2.71828205 2.71828205
 2.7185235  2.71611003]
```

(Technically it's not converged because with larger n the value doesn't stop here but goes to 0.)

For IEEE754 double type, $q = 11$, $p = 52$, the rounding error $\epsilon = 2^{-52} \approx 2.22^{-16}$. The difference between e and e_n is:

$$\mathrm{e} - \left(1 + \frac{1}{b}\right)^n = \frac{\mathrm{e}}{2n} - \frac{11\mathrm{e}}{24n^2} + \mathcal{O}(n^-3) \approx \frac{\mathrm{e}}{2n} \tag{5.1}$$

While the rounding error:

$$\mathrm{fl}\left[\left(1+\frac{1}{n}\right)^n\right] - \mathrm{e}_n \approx \left(1 + \frac{1}{n} + \epsilon\right)^n - \left(1+\frac{1}{n}\right)^n \approx n\epsilon \tag{5.2}$$

Total error $\mathrm{e}/2n + n\epsilon$ has minimum at $n \sim \sqrt{e/2\epsilon} \approx 10^9$ which is consistent with the array from code output. However, the converge value 10^{13} happens in ill-behaved regime: in Fig.(3) we can see that the "converged" term $n = 10^{13}$ and 10^{14} locate in the oscillating area, therefore it's a numerical coincidence.

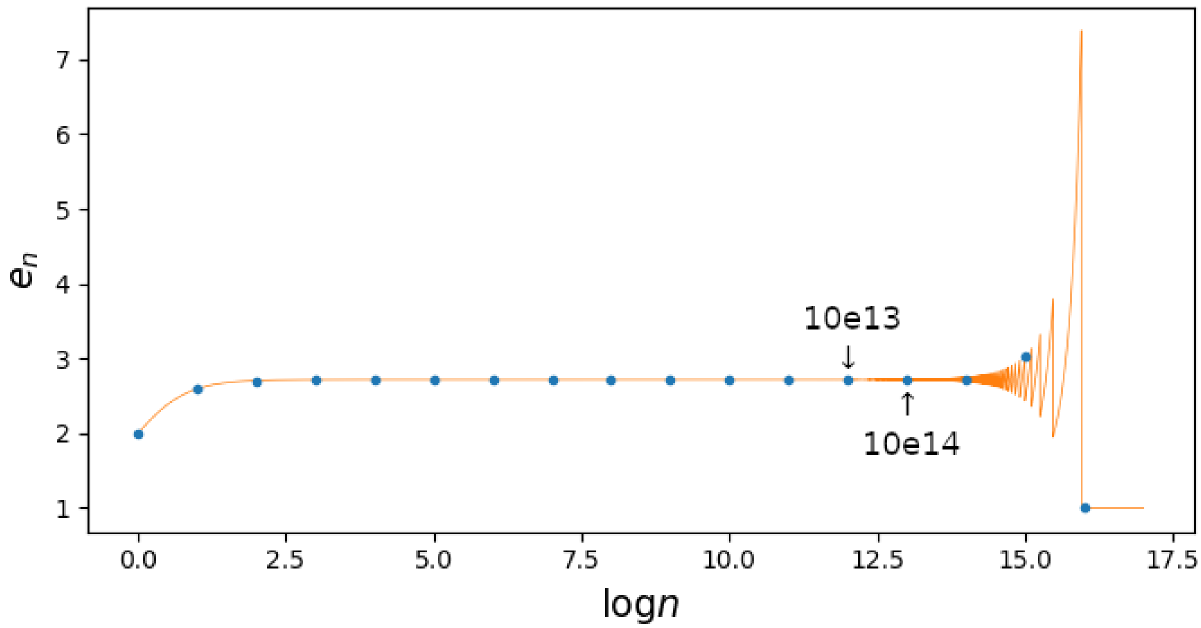

Figure 3: n series of e (log scale), orange line is continuous n, blue dot is $n = 1, 10, 10^2, \cdots$. The oscillation from right to left is $(1 + k\epsilon)^n$ with ϵ the smallest machine number step relative to 1 and $k = 1, 2, \cdots$ ($1 + 1/n$ is rounded to $1 + k\epsilon$).

6 Fun with square roots

Define $y_N \equiv x^{2^{-N}}$ meaning N times square root. For $x \in [0, 1]$, $y_N = 1 - k\epsilon$ where $k = 0, 1, 2, \cdots$ and $\epsilon = 2^{-p-1}$ is the machine float number precision, i.e. rounding error [1].

Because y_N is a step function of x, the final result $y = y_N^{2^N}$ is also a step function of x. $y = x$ only when in real calculation $x^{-2^N} = 1 - k\epsilon$ (Assuming the algorithm is perfect so that the error introduced by it is smaller than rounding error).

Let $M = 2^N$ and $\delta = k2^{-p-1}$, where N and p are close, i.e. $M\delta \sim 1$. The numbers that are intact are:

$$(1 - \delta)^M = \left[(1 - \delta)^{1/\delta}\right]^{M\delta} \approx \mathrm{e}^{-M\delta} \tag{6.1}$$

Plug M and δ in, the intact numbers are $\{\exp[-k2^{N-p-1}] : k = 0, 1, 2, \cdots\}$. Specifically for double precision $p = N = 52$, they are $\{1, \sqrt{\mathrm{e}}, \mathrm{e}^{-1}, \mathrm{e}^{-1.5}, \cdots\}$. This result is validated in Fig. (5)

[1] It's 2^{-p-1} rather than 2^{-p} because $y_N < 1$ has exponential part smaller than that of 1 by 1, which means that for machine number $1 + \delta \neq 1$, smallest $\delta = 2^{-p}$ but for $1 - \delta \neq 1$, $\delta = 2^{-p-1}$.

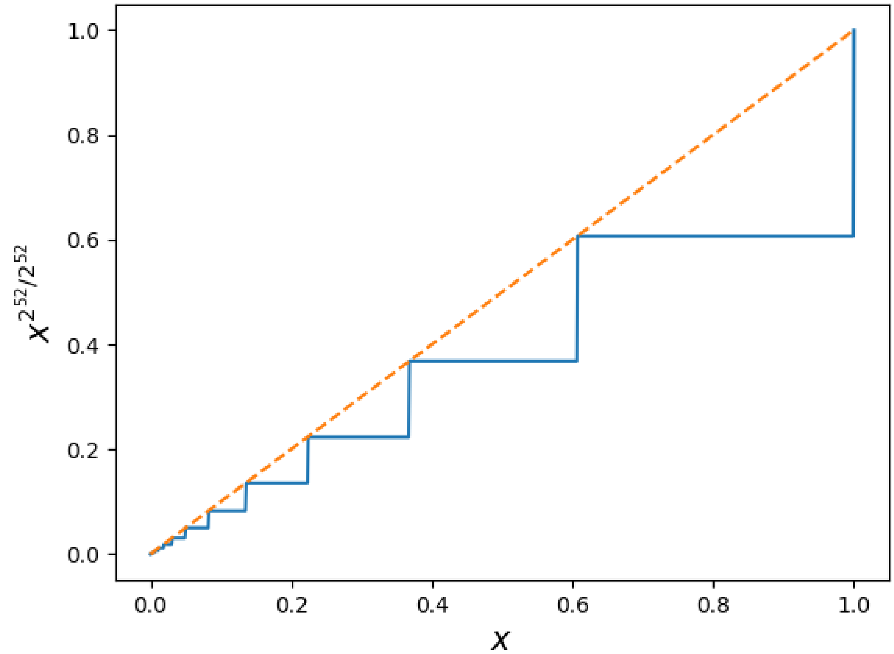

Figure 4: Square root 52 times and square 52 times

7 The issue with polynomial roots

For $N = 20$

$$a_n = (-)^{N-n} \sum_{|\{p\}|=n} \prod_{k\notin\{p\}}^{N} k = (-)^{N-n} \sum_{1\leq q_0<q_1<\cdots<q_{N-n}\leq N} \prod_i q_i \tag{7.1}$$

(a) This is calculated by the following script:

```python
#!/usr/bin/env python
# -*- coding:utf-8 -*-
import numpy as np
from numpy.polynomial.polynomial import Polynomial
N = 20
a = [0]*(N+1)
def prods(N, n=0, prod=1):
    if N == 0:
        a[n] += prod
        return
    prods(N-1, n+1, prod*N)
    prods(N-1, n, prod)

if __name__ == "__main__":
    prods(N)
    coef = [a[n]*(-1)**(N-n) for n in range(len(a))]
    print(np.array(coef))
```

The result is (from a_{20} to a_0):

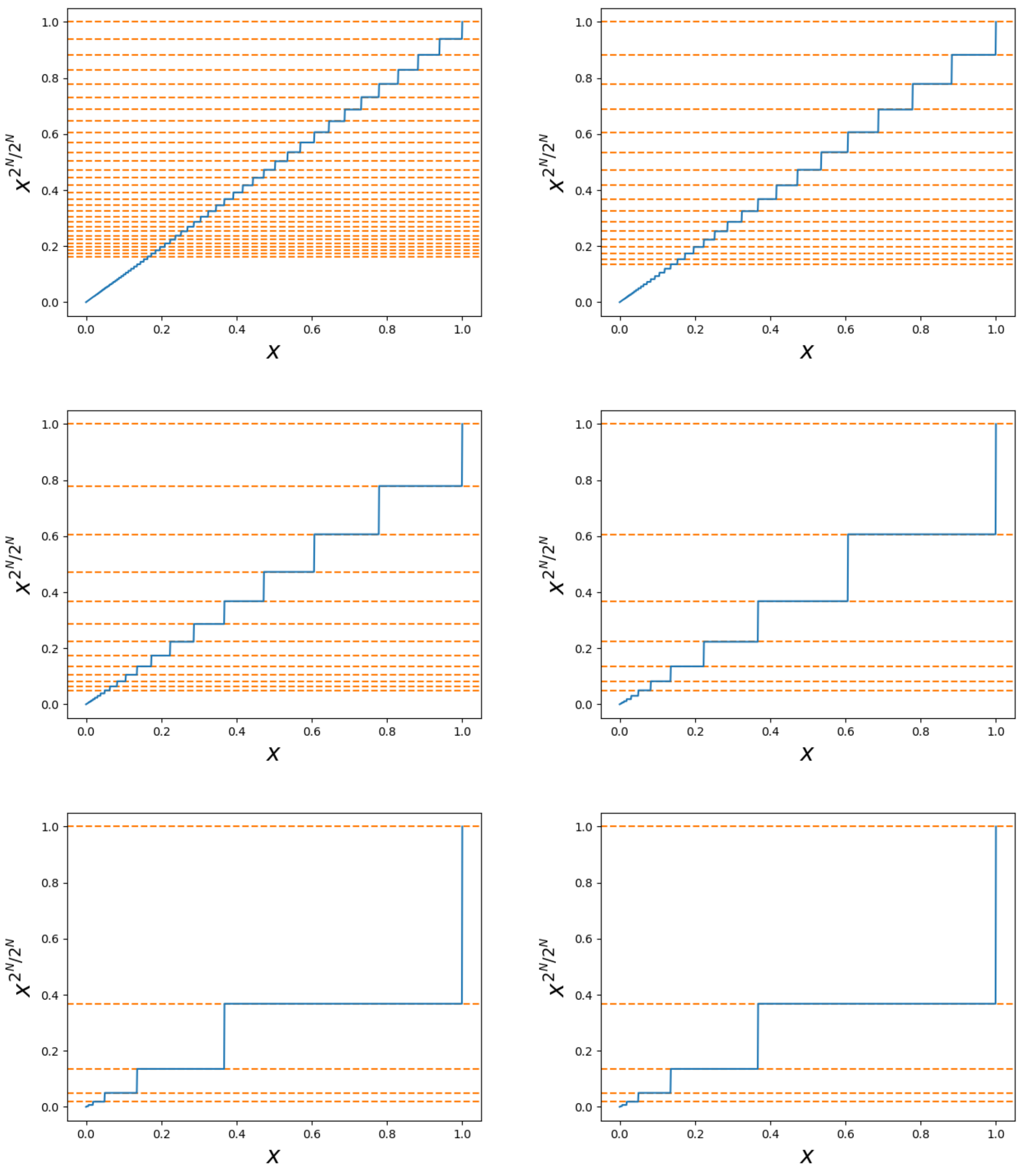

Figure 5: Validation of problem 6 with $N = 49, 50, 51, 52, 53, 54$ respectively. Horizontal line is $\{\exp[-k2^{N-53}] : k \in \mathbb{N}\}$.

```
[1 -210 20615 -1256850 53327946 -1672280820 40171771630 -756111184500
 11310276995381 -135585182899530 1307535010540395 -10142299865511450
 63030812099294896 -311333643161390640 1206647803780373360
 -3599979517947607200 8037811822645051776 -12870931245150988800
 13803759753640704000 -8752948036761600000 2432902008176640000]
```

(b) The polynomial is already bad behaved in evaluation using double precision.

```python
from functools import reduce
print(np.array([reduce(lambda a, b: a*x+b, coef)
                for x in range(1, 21)]))
print(np.array([reduce(lambda a, b: a*x*1.0+b, coef)
                for x in range(1, 21)]))
```

This script shows that for integer (perfect precision in Python) the polynomial behaves as we expect (all 0s) but for float evaluation it deviates.

```python
coef.reverse()
w = Polynomial(np.array(coef, dtype=np.float))
# have to explicitly specify dtype,
# otherwise root finder doesn't work
print([w(x) for x in range(1, 21)])
print(w.roots())
# This uses eigenvalues of the companion matrix for roots
from scipy.optimize import root
print(root(w, 21.0))
# This uses Optimization method root finding
```

The Newton's method `scipy.iptimize.root` finds the root near 21 to be 19.9999717.

(c) for $\delta = 10^{-8}, 10^{-6}, 10^{-4}, 10^{-2}$, the root finder gives root to be 9.58534944, 7.75272109, 5.9693346, 5.46959278 respectively.

```python
for delta in (1e-8, 1e-6, 1e-4, 1e-2):
    coef[20] = 1 + delta
    w = Polynomial(np.array(coef, dtype=np.float))
    print(root(w, 21.0))
```

(d) The root finder converges to the same roots about 8.92 (from function evaluation we can see more digits are not meaningful).

```python
coef[20]=1
coef[19]=-210-2**(-23)
w = Polynomial(np.array(coef, dtype=np.float))
print(root(w, 16.1))
print(root(w, 17.1))
print(w.roots())
```

Companion matrix eigenvalue algorithm shows that there's no real roots between 10 and 20.

(e) For a differentiate $\mathrm{d}\vec{a}$, there is:

$$p(\Omega_k + \mathrm{d}\Omega_k) + \sum_\ell \Omega_k^\ell \,\mathrm{d}a_\ell = 0 \tag{7.2}$$

$$\Rightarrow \mathrm{d}\Omega_k = -\sum_\ell \frac{\Omega_k^\ell}{p'(\Omega_k)} \,\mathrm{d}a_\ell \tag{7.3}$$

$$\Rightarrow \frac{\partial \Omega_k}{\partial a_\ell} = -\frac{\Omega_k^\ell}{p'(\Omega_k)} \tag{7.4}$$

So for condition number:

$$(\mathrm{cond}\,\Omega_k)(\vec{a}) \equiv \sum_\ell \Gamma_{k\ell} \equiv \sum_\ell \left| \frac{\partial \Omega_k}{\partial a_\ell} \frac{a_\ell}{\Omega_k} \right| = \sum_\ell \frac{\Omega_k^{\ell-1} |a_\ell|}{|p'(\Omega_k)|} \tag{7.5}$$

For $\Omega_k = 14, 16, 17, 20$, Eq. (7.5) evaluates to 5.4e13, 3.5e13, 1.8e13 and 1.4e11 respectively. These results are significantly larger than 1, which means that small difference in $\vec{a}$ can change roots unbounded.

No algorithm can help us because this is intrinsic to the problem. Any algorithm that reflects the problem faithfully cannot decrease the conditioning number of the problem itself. This suggests that using coefficients for polynomial with high degree is an ill-behaved problem and should be avoided.

8 Recurrence in reverse

(i) $0 < y_{n+1}$, so $y_n < \mathrm{e}/(n+1)$. Similarly $y_{n+1} < e/(n+2)$, which means $\mathrm{e}/(n+2) < y_n < \mathrm{e}/(n+1)$, so $y_{n+1}/y_n < 1$, and with large n, this ratio goes to 1. Using this bounded approximation for y_n, error from y_{n+1} to y_n is (assuming n as an integer is perfect machine number, neglect error in e):

$$y_n = \frac{\mathrm{e} - y_{n+1}}{n+1} \quad \Rightarrow \quad \frac{\Delta y_n}{y_n} = \frac{1}{n+1} \frac{\Delta y_{n+1}}{y_n} + \epsilon_\div = \frac{1}{n+1} \frac{y_{n+1}}{y_n} \frac{\Delta y_{n+1}}{y_{n+1}} + \epsilon_\div \tag{8.1}$$

$$< \frac{1}{n+1} \frac{\Delta y_{n+1}}{y_{n+1}} + \epsilon_\div \tag{8.2}$$

So for a large N and reversed recurrence result y_k with $k < N$,

$$\frac{\Delta y_k}{y_k} < \frac{k!}{N!} \delta_N + \sum_{i=0}^{N-k} \frac{k!}{(N-i)!} \epsilon_\div \tag{8.3}$$

$$\lesssim \frac{k!}{N!} \delta_N + \epsilon_\div \tag{8.4}$$

where $\delta_N = \Delta y_N / y_N$ is the error in y_N and the last step is estimated for large N and k. Given tolerance ϵ,

- ϵ cannot be smaller than rounding error when do division $\epsilon_\div$
- With larger N, error introduced from y_N will vanish.
- $N - k \sim \log_k[(\epsilon - \epsilon_\div)/\delta_N]$ will be good enough for y_k with tolerance ϵ

(ii) Since δ_N influence goes to 0 with large N, we don't need precise y_N for the algorithm. We can choose $y_N \approx e/(N+1)$ as the initial number for a big N.

(a) From Eq.(8.4), with $\epsilon_{\div} = 0$ the condition number is:

$$(\operatorname{cond} g_k)(y_N) = \left| \frac{\Delta y_k / y_k}{\Delta y_N / y_N} \right| \lesssim \frac{k!}{N!} \tag{8.5}$$

The expression becomes $\operatorname{cond} g_k = k!/N!$ when N is large.
(b) This means $\epsilon \lesssim k!/N!$ or $N > \Gamma^{-1}(k!\epsilon^{-1})$. A very loose approximation is $N > k + \log_k \epsilon$.
(c) For IEEE754 double type, $\epsilon = \text{eps} = 2^{-52}$, for $k = 21$, $N > 31$
(c) Note that for $N = 32$, output for error is 0.

```python
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from scipy.integrate import quad
from numpy import exp, e
k=20
N=31

if __name__ == "__main__":
    yk = quad(lambda x: x**k*exp(x), 0, 1)[0]
    y = [0]*N
    for n in reversed(range(k, N)):
        y[n-1] = (e - y[n])/n
    print("Numerical integratal: ", yk)
    print("Reversed recurrence: ", y[k])
    print("Error: %e"%abs(y[k]/yk-1))

# Output:
#  Numerical integratal:  0.12380383076256998
#  Reversed recurrence:  0.12380383076256918
#  Error: 6.550316e-15
```