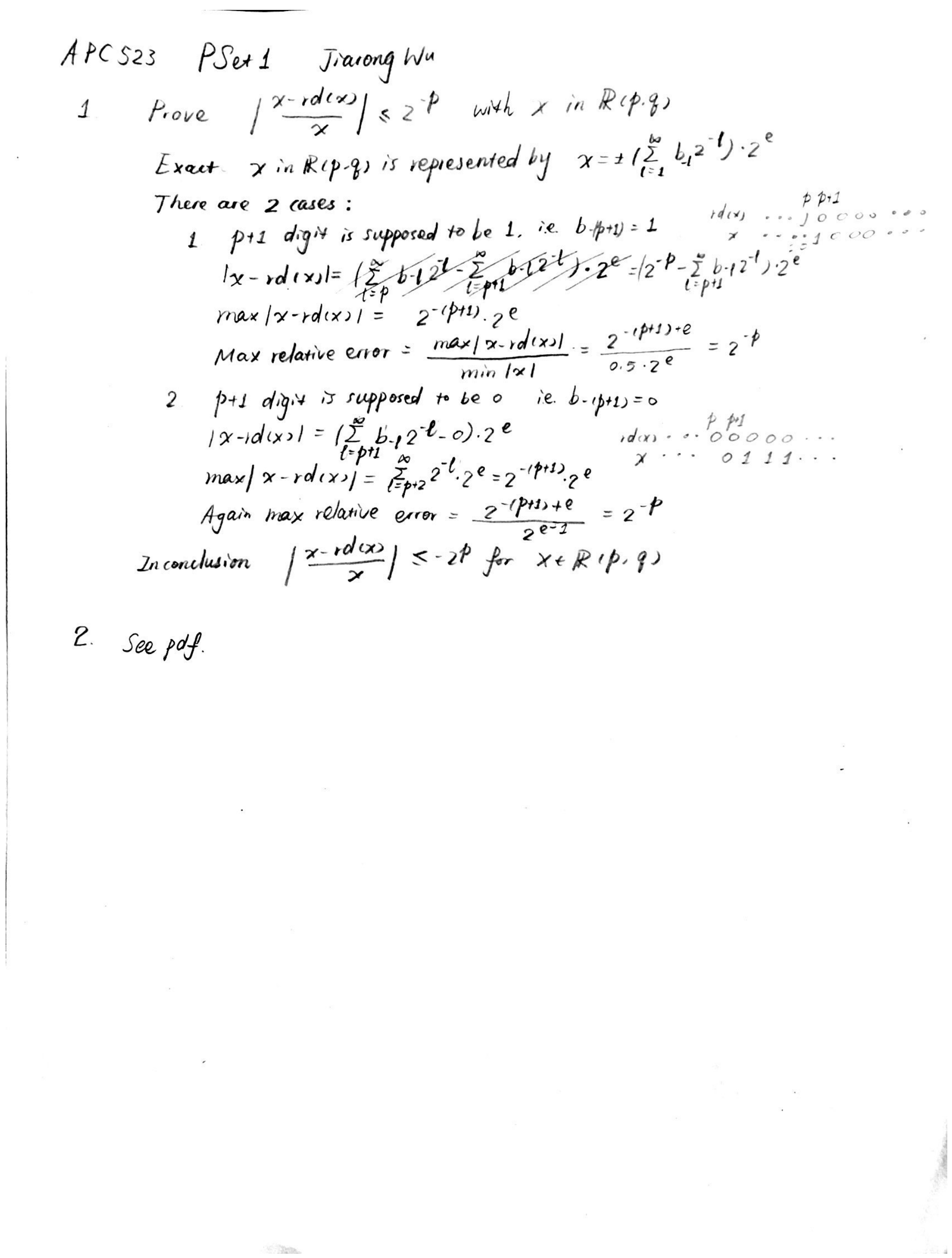
APC523 HW1

Jiarong Wu

March 13, 2019

Problem 1



APC523 PSet 1 Jiarong Wu

1 Prove $\left|\frac{x-rd(x)}{x}\right| \leq 2^{-p}$ with x in $\mathbb{R}(p.q)$

Exact x in $\mathbb{R}(p.q)$ is represented by $x = \pm(\sum_{l=1}^{\infty} b_{-l}2^{-l})\cdot 2^{e}$

There are 2 cases:

1 $p+1$ digit is supposed to be 1, i.e. $b_{-(p+1)} = 1$

$p \quad p+1$
$rd(x) \quad \cdots 1\,0\,0\,0\,0 \cdots$
$x \quad \cdots 1\,0\,0\,0 \cdots$

$|x - rd(x)| = \cancel{(\sum_{l=p}^{\infty} b_{-l}2^{-l} - \sum_{l=p+1}^{\infty} b_{-l}2^{-l})\cdot 2^{e}} = (2^{-p} - \sum_{l=p+1}^{\infty} b_{-l}2^{-l})\cdot 2^{e}$

$\max|x-rd(x)| = 2^{-(p+1)}\cdot 2^{e}$

Max relative error $= \frac{\max|x-rd(x)|}{\min|x|} = \frac{2^{-(p+1)+e}}{0.5\cdot 2^{e}} = 2^{-p}$

2 $p+1$ digit is supposed to be 0 i.e. $b_{-(p+1)} = 0$

$p \quad p+1$
$rd(x) \quad \cdots 0\,0\,0\,0\,0 \cdots$
$x \quad \cdots 0\,1\,1\,1 \cdots$

$|x - rd(x)| = (\sum_{l=p+1}^{\infty} b_{-l}2^{-l} - 0)\cdot 2^{e}$

$\max|x - rd(x)| = \sum_{l=p+2}^{\infty} 2^{-l}\cdot 2^{e} = 2^{-(p+1)}\cdot 2^{e}$

Again max relative error $= \frac{2^{-(p+1)+e}}{2^{e-1}} = 2^{-p}$

In conclusion $\left|\frac{x-rd(x)}{x}\right| \leq -2^{p}$ for $x \in \mathbb{R}(p,q)$

2. See pdf.

```
In [5]: from decimal import Decimal
        import numpy as np
        import matplotlib.pyplot as plt
        import copy
        from math import e
```

Problem 2

Part (a)&(b)

```
In [10]: # rounding sub function
         def rounding(x_origin):
             n = 5
             s = '%.'+str(n)+'g'
             x_float = float(s % x_origin)
             return x_float
         # compute the denominator
         def factorial(n):
             fac = 1.0
             fac = rounding(fac)
             for i in range(1, n+1):
                 fac = fac*i
                 fac = rounding(fac)
             return fac
         # compute the numerator
         def exponential(x, n):
             exp = 1.
             x = rounding(x)
             for i in range(0, n):
                 exp = exp * x
                 exp = rounding(exp)
             return exp
         # compute f(x) = e^x
         func_ex = 0
         x = 5.5
         n = 30
         # compute each term
         term = np.zeros(31, dtype=float)
         for i in range(0, n):
             term[i] = exponential(x, i)/factorial(i)
             term[i] = rounding(term[i])
         print(term)
         # Adding from left to right
         print('Adding from left to right:')
         for i in range(0, n):
             func_ex_laststep = func_ex
```

```
            func_ex = 0
            for j in range(0, i+1):
                func_ex = func_ex + term[j]
                func_ex = rounding(func_ex)
            if func_ex_laststep == func_ex:
                print('Result converges at n =', i)
                break
        print('e^x with 5-digit mantissa =', func_ex)
        print('e^x with built in func =', e**x)
        err = abs(func_ex - e**x)/(e**x)
        print('relative error = %0.6f' % err)
```

```
[1.0000e+00 5.5000e+00 1.5125e+01 2.7730e+01 3.8129e+01 4.1942e+01
 3.8447e+01 3.0208e+01 2.0768e+01 1.2692e+01 6.9805e+00 3.4902e+00
 1.5997e+00 6.7679e-01 2.6588e-01 9.7484e-02 3.3510e-02 1.0842e-02
 3.3128e-03 9.5898e-04 2.6372e-04 6.9070e-05 1.7269e-05 4.1297e-06
 9.4638e-07 2.0821e-07 4.4043e-08 8.9715e-09 1.7623e-09 3.3422e-10
 0.0000e+00]
Adding from left to right:
Result converges at n = 18
e^x with 5-digit mantissa = 244.71
e^x with built in func = 244.69193226422033
relative error = 0.000074
```

Part (c)

```
In [11]: # Adding from right to left
         print('Adding from right to left:')
         func_ex = 0
         x = 5.5
         n = 30
         for i in range(0, n):
             func_ex_laststep = func_ex
             func_ex = 0
             for j in range(i, -1, -1):
                 func_ex = func_ex + term[j]
                 func_ex = rounding(func_ex)
             if func_ex_laststep == func_ex:
                 print('Result converges at n =', i)
                 break
         print('e^x with 5-digit mantissa =', func_ex)
         print('e^x with built in func =', e**x)
         err = abs(func_ex - e**x)/(e**x)
         print('relative error = %0.6f' % err)
```

```
Adding from right to left:
Result converges at n = 17
```

```
e^x with 5-digit mantissa = 244.69
e^x with built in func = 244.69193226422033
relative error = 0.000008
```

Part (d)

```
In [12]: # Flip the sign of the odd-index terms, which is equivalent to set x = -5.5
         term_neg = np.zeros(31, dtype=float)
         for i in range(0, n):
             term_neg[i] = (-1)**i*term[i]
         print(term_neg)
```

```
[ 1.0000e+00 -5.5000e+00  1.5125e+01 -2.7730e+01  3.8129e+01 -4.1942e+01
  3.8447e+01 -3.0208e+01  2.0768e+01 -1.2692e+01  6.9805e+00 -3.4902e+00
  1.5997e+00 -6.7679e-01  2.6588e-01 -9.7484e-02  3.3510e-02 -1.0842e-02
  3.3128e-03 -9.5898e-04  2.6372e-04 -6.9070e-05  1.7269e-05 -4.1297e-06
  9.4638e-07 -2.0821e-07  4.4043e-08 -8.9715e-09  1.7623e-09 -3.3422e-10
  0.0000e+00]
```

```
In [13]: # Always adding from left to right
         print('Adding from left to right:')
         func_ex = 0
         for i in range(0, n):
             func_ex_laststep = func_ex
             func_ex = 0
             for j in range(0, i+1):
                 func_ex = func_ex + term_neg[j]
                 func_ex = rounding(func_ex)
             if func_ex_laststep == func_ex:
                 print('Result converges at n =', i)
                 break
         print('e^x with 5-digit mantissa =', func_ex)
         print('e^x with built in func =', e**(-5.5))
         err = abs(func_ex - e**(-5.5))/e**(-5.5)
         print('relative error = %0.4f' % err)
         # Always adding from right to left
         func_ex = 0
         print('Adding from right to left:')
         for i in range(0, n):
             func_ex_laststep = func_ex
             func_ex = 0
             for j in range(i, -1, -1):
                 func_ex = func_ex + term_neg[j]
                 func_ex = rounding(func_ex)
             if func_ex_laststep == func_ex:
                 print('Result converges at n =', i)
```

```
                break
        print('e^x with 5-digit mantissa =', func_ex)
        print('e^x with built in func =', e**(-5.5))
        err = abs(func_ex - e**(-5.5))/e**(-5.5)
        print('relative error = %0.4f' % err)
```

```
Adding from left to right:
Result converges at n = 26
e^x with 5-digit mantissa = 0.0038363
e^x with built in func = 0.004086771438464068
relative error = 0.0613
Adding from right to left:
Result converges at n = 20
e^x with 5-digit mantissa = 0.004
e^x with built in func = 0.004086771438464068
relative error = 0.0212
```

```
In [14]: # Adding positive from left to right, negative from right to left
         func_ex = 0
         print('Adding positive from left to right, negative from right to left:')
         for i in range(0, n):
             func_ex_laststep = func_ex
             func_ex = 0
             func_ex_1 = 0
             func_ex_2 = 0
             for j in range(0, i+1):
                 if term_neg[j] >= 0:
                     func_ex_1 = func_ex_1 + term_neg[j]
                     func_ex_1 = rounding(func_ex_1)
             for j in range(i, -1, -1):
                 if term_neg[j] <= 0:
                     func_ex_2 = func_ex_2 + term_neg[j]
                     func_ex_2 = rounding(func_ex_2)
             func_ex = func_ex_1 + func_ex_2
             func_ex = rounding(func_ex)
             if func_ex_laststep == func_ex:
                 print('Result converges at n =', i)
                 break
         print('e^x with 5-digit mantissa =', func_ex)
         print('e^x with built in func =', e**(-5.5))
         err = abs(func_ex - e**(-5.5))/e**(-5.5)
         print('relative error = %0.4f' % err)
         # Adding positive from right to left, negative from left to right
         print('Adding positive from right to left, negative from left to right:')
         func_ex = 0
         for i in range(0, n):
             func_ex_laststep = func_ex
```

```
            func_ex = 0
            func_ex_1 = 0
            func_ex_2 = 0
            for j in range(0, i+1):
                if term_neg[j] <= 0:
                    func_ex_1 = func_ex_1 + term_neg[j]
                    func_ex_1 = rounding(func_ex_1)
            for j in range(i, -1, -1):
                if term_neg[j] >= 0:
                    func_ex_2 = func_ex_2 + term_neg[j]
                    func_ex_2 = rounding(func_ex_2)
            func_ex = func_ex_1 + func_ex_2
            func_ex = rounding(func_ex)
            if func_ex_laststep == func_ex:
                print('Result converges at n =', i)
                break
        print('e^x with 5-digit mantissa =', func_ex)
        print('e^x with built in func =', e**(-5.5))
        err = abs(func_ex - e**(-5.5))/e**(-5.5)
        print('relative error = %0.4f' % err)
```

```
Adding positive from left to right, negative from right to left:
Result converges at n = 18
e^x with 5-digit mantissa = 0.0
e^x with built in func = 0.004086771438464068
relative error = 1.0000
Adding positive from right to left, negative from left to right:
Result converges at n = 19
e^x with 5-digit mantissa = 0.01
e^x with built in func = 0.004086771438464068
relative error = 1.4469
```

Comment for (d): Adding positive from left to right, negative from right to left converges the fast. Adding from right to left gives the lowest error. When the exponential is positive, adding from right to left also gives relatively smaller error and faster convergence.

Part (e): We can compute e^{-x} by $\frac{1}{e^x}$ In this case with 5-digit mantissa we have $e^{-5.5} = 1/e^{5.5} = 1/244.69 = 4.0868 \times 10^{-3}$ which gives pretty satisfactory result.

Problem 3

3

(a) Assume x as an exact machine number

The first way :

$fl(x\cdot x) = (x\cdot x)(1+\varepsilon) = x^2(1+\varepsilon_1)$ $\quad \varepsilon_1 = \varepsilon$

$fl(x^2(1+\varepsilon_1)\cdot x) = (x^2\cdot x)(1+\varepsilon_1)(1+\varepsilon) \simeq x^3(1+\varepsilon_1+\varepsilon) = x^3(1+\varepsilon_2)$ $\quad \varepsilon_2 = \varepsilon_1+\varepsilon = 2\varepsilon$

Doing the same operation (n-1)times we get :

$\varepsilon_{n-1} = (n-1)\varepsilon$ $\qquad fl(\underbrace{x\cdot x\cdot x\cdots x}_{n\,x}) = (x\cdot x\cdots x)(1+(n-1)\varepsilon)$

$|\varepsilon| \le eps$

So the error is bounded by $(n-1)eps$

The second way :

$fl(\ln x) = (\ln x)(1+\varepsilon_{ln})$ $\quad \varepsilon_{ln}$ is the error introduced by ln operation

$fl(n\ln x) = n\ln x(1+\varepsilon_{ln})(1+\varepsilon_{multi})$

$= n\ln x(1+\varepsilon_{ln}+\varepsilon_{multi})$

$fl[\exp(n\ln x(1+\varepsilon_{ln}))] = \exp[n\ln x(1+\varepsilon_{ln}+\varepsilon_{multi})(1+\varepsilon_{exp})]$ Taylor expansion

$= e^{n\ln x}(1+n\ln x(\varepsilon_{ln}+\varepsilon_{multi}))(1+\varepsilon_{exp})$

$= e^{n\ln x}(1+n\ln x(\varepsilon_{ln}+\varepsilon_{multi})+\varepsilon_{exp})$

$|\varepsilon_{ln}| \le eps \quad |\varepsilon_{exp}| \le eps \quad |\varepsilon_{multi}| \le eps$

$\therefore$ The error is bounded by $(2n\ln x+1)\,eps$

In general. when $|\ln x| > 1$, repeated multiplication is more accurate than the log-exponential method

(b) In this scenario, we can only use the log-exponential method

for $x^a = e^{a\ln x}$

① x is exact while a comes with ε_a $\quad fl(\ln x) = (\ln x)(1+\varepsilon_{ln})$

$\ln x = \ln x$ (assume no error with $\ln x$ operation)

$e^{\hat{a}\ln x} = e^{a(1+\varepsilon_a)\ln x} = e^{a\ln x + a\varepsilon_a \ln x} \simeq x^a(1+a\varepsilon_a\ln x)$

$\therefore \varepsilon = (a\ln x)\varepsilon_a$ $\quad$ If $a\ln x$ is big, the error of ε is substantial

② a is exact while x comes with ε_x

$\ln\hat{x} = \ln[x(1+\varepsilon_x)] = \ln x + \frac{1}{x}(x\varepsilon_x)$

$e^{a\ln\hat{x}} = e^{(\ln x+\varepsilon_x)a} = x^a(1+a\varepsilon_x)$

$\therefore \varepsilon = a\varepsilon_x$ $\quad$ If a is big the error is substantial

Problem 4

4 a) $f(x) = 1 - e^{-x}$

$T(x) = \dfrac{x\frac{\partial f}{\partial x}}{f} = \dfrac{x e^{-x}}{1-e^{-x}} = \dfrac{x}{e^{x}-1}$

Expand e^{x} on $[0,1]$ with

$e^{x} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \cdots$

$T(x) = \dfrac{x}{x + \frac{x^2}{2} + O(x^3)}$, therefore $T(x) < 1$ on $[0,1]$

b) Suppose x is an exact machine number

$e(1+\varepsilon)$ or $e(1-\varepsilon)$ the sign doesn't matter?

$f_A(x) = f(x_A)$

$fl(e^{-x}) = e^{-x}(1+\varepsilon_e)$

$fl(1-e^{-x}) = [1 - e^{-x}(1+\varepsilon_e)](1+\varepsilon_{sb})]$

$= (1 - e^{-x} - e^{-x}\varepsilon_e)(1+\varepsilon_{sb})$

$= (1-e^{-x})[1+\varepsilon_{sb} - \frac{e^{-x}}{1-e^{-x}}\varepsilon_e + O(\varepsilon^2)]$

$= (1-e^{-x})(1+\varepsilon_{sb} - \frac{1}{e^{x}-1}\varepsilon_e)$

$\therefore \quad f_A(x) = f(x)(1+\varepsilon_{forward}) \qquad \varepsilon_{forward} = |\varepsilon_{sb} - \frac{1}{e^{x}-1}\varepsilon_e|$

Next is to find x_A so that $f(x_A) = f(x)(1+\varepsilon_{forward})$

Take the result from (a)

$f(x + \varepsilon_{backward}) = f(x)(1+\varepsilon_{forward})$ and $\dfrac{\varepsilon_{forward}}{\varepsilon_{backward}} = T(x)$

There are not multiple values of x_A

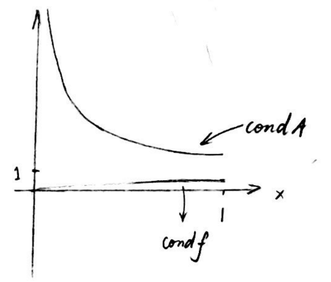
$\dfrac{|x_A - x|}{|x|} = \varepsilon_{backward} = \dfrac{\varepsilon_{forward}}{T} = \dfrac{|\varepsilon_{sb} - \frac{1}{e^{x}-1}\varepsilon_x|}{\frac{x}{e^{x}-1}} = |\frac{e^{x}-1}{x}\varepsilon_{sb} - \frac{1}{x}\varepsilon_x|$

$(\text{cond}A)\ \vec{x} \le \dfrac{|\frac{e^{x}-1}{x}\varepsilon_{sb} - \frac{1}{x}\varepsilon_x|}{eps}$ $\qquad$ eps is the machine error

$|\varepsilon_{sb}| \le \varepsilon \quad |\varepsilon_x| \le \varepsilon$

The upper bound of RHS is $\dfrac{|\frac{e^{x}}{x}eps|}{eps} = \dfrac{e^{x}}{x} \simeq \dfrac{1+x+\frac{x^2}{2}+\frac{x^3}{6}}{x} > 1$

c)



The root of poor conditioning comes from the subtraction between 2 very close number 1 & e^{-x} when $x \to 0$

d) For 2 close number p & q (Here $p=1$, $q=e^{-x}$)

If $2^{-b} \le 1-\frac{q}{p} \le 2^{-a}$

By doing $p-q$ we lose at least a, at most b bits of significance

- To ensure at most 1 bit of significance loss:

  $1-e^{-x} \ge 2^{-1}$ $x \ge -\ln(0.5)$ $x \ge 0.6931$

- at most 2 bits of significance loss:

  $1-e^{-x} \ge 2^{-2}$ $x \ge 0.2877$

- at most 3 bits of significance loss:

  $1-e^{-x} \ge 2^{-3}$ $x \ge 0.1335$

- at most 4 bits of significance loss:

  $1-e^{-x} \ge 2^{-4}$ $x \ge 0.0645$

(e) $\frac{|y_A^*-y|}{|y|} \le (\mathrm{cond} f)(x)\{\varepsilon + (\mathrm{cond} A)\,x \cdot eps\}$

ε is the rounding error of x $\varepsilon = \frac{|x^*-x|}{|x|}$

Since x is a machine number, $\varepsilon = 0$

$\frac{|y_A^*-y|}{|y|} \le \frac{x}{e^x-1} \cdot \frac{e^x}{x} \cdot eps = \frac{e^x}{e^x-1}\, eps$

1 bit of significance loss	$\rightarrow e^{-x} \le \frac{1}{2}$	$\frac{|y_A^*-y|}{|y|} \le 2\,eps$
2 ~	$\rightarrow e^{-x} \le \frac{3}{4}$	$\le 4\,eps$
3 ~	$\rightarrow e^{-x} \le \frac{7}{8}$	$\le 8\,eps$
4 ~	$\rightarrow e^{-x} \le \frac{15}{16}$	$\le 16\,eps$

f) an alternative way is to compute $1-e^{-x}$ for $x \in [0,1]$

e^{-x} is of the order of x.

So instead of calculating $1-e^{-x}$, calculate (for $x \ne 0$)

$\frac{1}{x}$ and $\frac{e^{-x}}{x}$ separately. then subtract

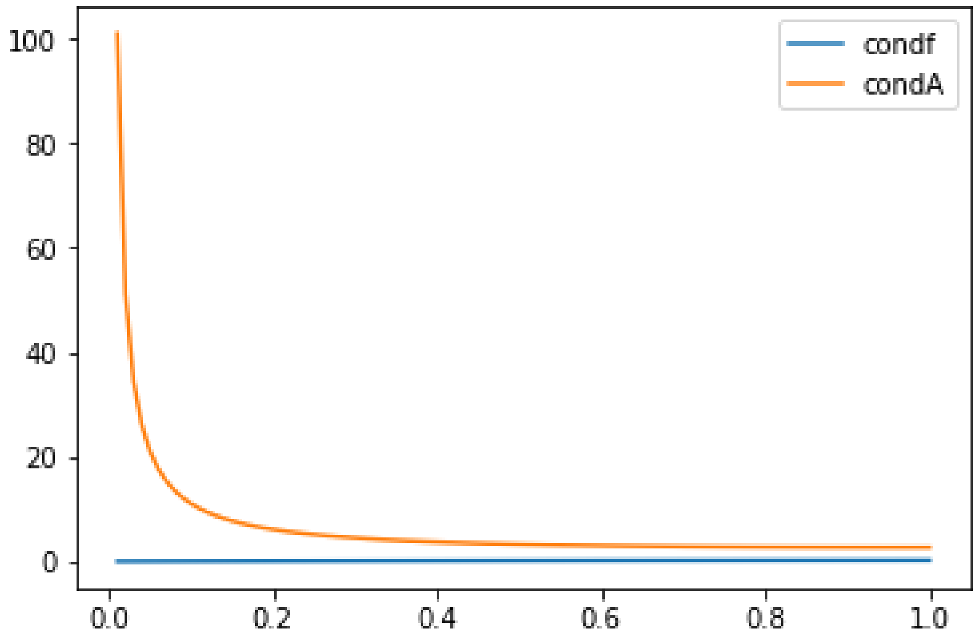
$$f(x) = \begin{cases} 1-e^{-x} = \left(\frac{1}{x} - \frac{e^{-x}}{x}\right)x & (x \ne 0) \\ 0, & (x=0) \end{cases}$$

Problem 4 - Plot

```
In [15]: # rr = np.arange(0.001, 1, 0.001)
         rr = np.linspace(0.01, 1, num=101)
         def y1(x):
             return x / (np.exp(x) + 1.)
```

```
def y2(x):
    return (np.exp(x)) / x

plt.plot(rr, y1(rr))
plt.plot(rr, y2(rr))
plt.legend(['condf', 'condA'])
plt.show()
```



Problem 5

```
In [21]: elim = list()
         elim.append(2) # n =1
         n_stop = 1
         for i in range(1, 100):
             n = 10**i
             e = (1+1/n)**n
             elim.append(e)
             diff = elim[i] - elim[i-1]
             if abs(diff) < 10**(-13):
                 n_stop = n
                 break
         print(diff)
         print(elim)
         print('n = ''%.2E' % Decimal(n))
```

```
0.0
[2, 2.5937424601000023, 2.7048138294215285, 2.7169239322355936, 2.7181459268249255, 2.7182682371
```

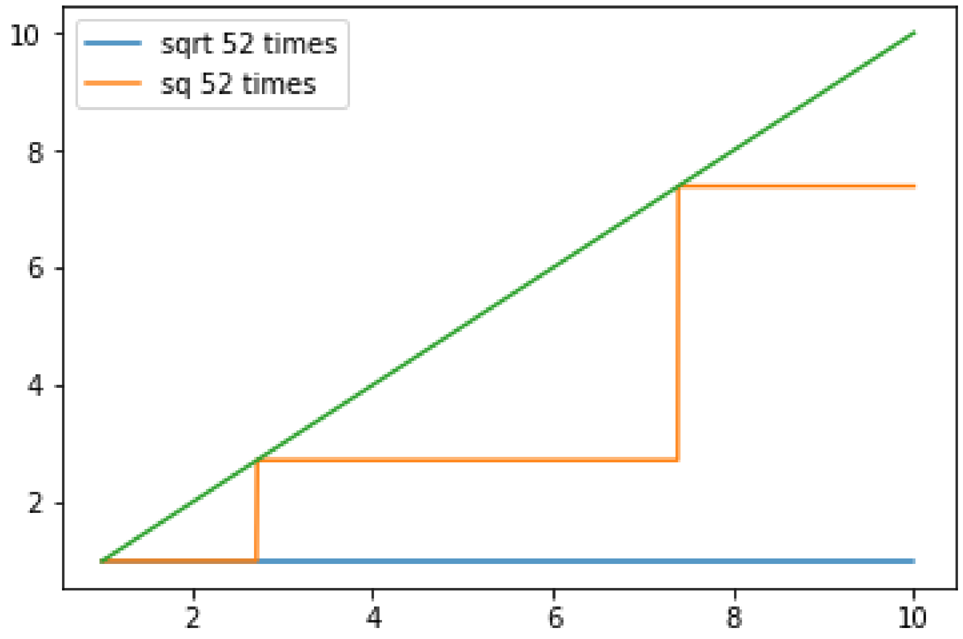
```
n = 1.00E+17
```

Analysis for Problem 5:

An explanation that accounts for why the value converges to the value it did: Because double precision only has 52 digits for mantissa which is approximately 15 decimal figures. When n increases to 16 $1/10^n$ is actually stored as 0, therefore $(1 + 1/10^n)^n$ becomes 1. In other word, there is a limit on how accurate $(1 + 1/10^n)^n$ can be as an approximate of e. With double precision 13-sigfig cannot be reached.

Problem 6

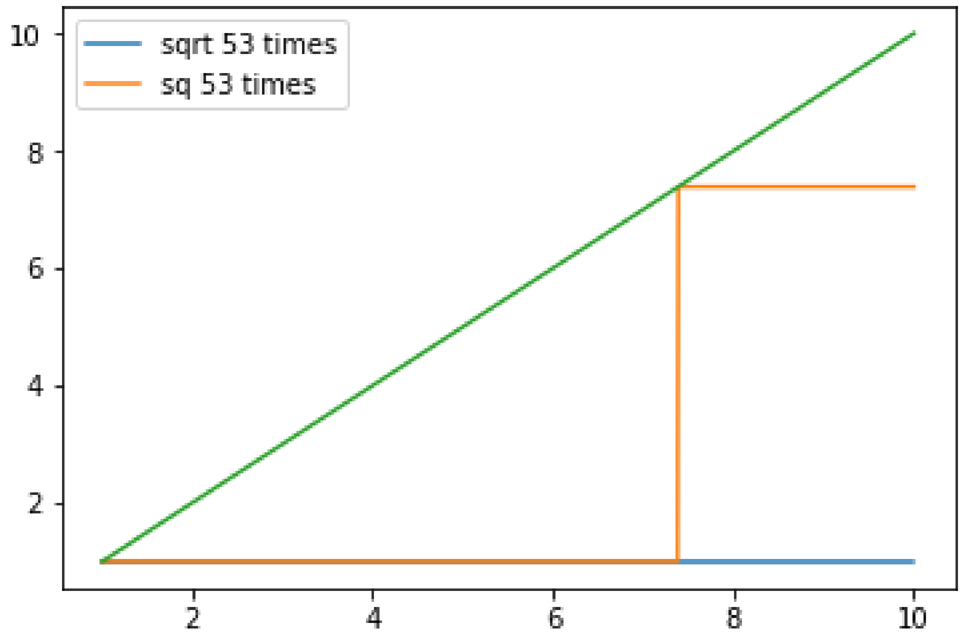
```
In [17]: x = np.linspace(1, 10, 1001, dtype='float64')
         x1 = np.linspace(1, 10, 1001, dtype='float64')
         for i in range(0,52):
             x1 = x1**0.5
             i = i+1
         x2 = copy.deepcopy(x1)
         for i in range(0,52):
             x2 = x2**2
             i = i+1
         plt.plot(x, x1)
         plt.plot(x, x2)
         plt.plot(x, x, '-')
         plt.legend(['sqrt 52 times', 'sq 52 times'])
         plt.show()
```



```
In [18]: x_exact = list()
         for i in range(0, 1001):
             if abs(x[i] - x2[i])/(x[i]+0.0000000001) < 10**(-3):
                 x_exact.append(x[i])
         print(x_exact)
```

```
[1.0, 2.719, 7.39]
```

```
In [19]: x = np.linspace(1, 10, 1001, dtype='float64')
         x1 = np.linspace(1, 10, 1001, dtype='float64')
         for i in range(0,53):
             x1 = x1**0.5
             i = i+1
         x2 = copy.deepcopy(x1)
         for i in range(0,53):
             x2 = x2**2
             i = i+1
         plt.plot(x, x1)
         plt.plot(x, x2)
         plt.plot(x, x, '-')
         plt.legend(['sqrt 53 times', 'sq 53 times'])
         plt.show()
```



```
In [20]: x_exact = list()
         for i in range(0, 1001):
             if abs(x[i] - x2[i])/(x[i]+0.0000000001) < 10**(-3):
                 x_exact.append(x[i])
         print(x_exact)

[1.0, 7.39]
```

Ananlysis of Problem 6:

When doing the operation 52 times, the "close points" are: [1.0, 2.719, 7.39] When doing 53 times: [1.0, 7.39]

We find some interesting numbers here: 2.719 $\simeq e$, 7.39 $\simeq e^2$ ... apparently the calculation is done with e as the base number. Note that the number of sqaure root operation 52 is the significant digits in double precision binary representation. Therefore we can see what happened is that there are a few steps in the square root implementation: 1. A number x is handed in and ln(x) is computed 2. ln(x) is transformed into its double-precision binary representation 3. ln(x) is devided by 2, which is equal to moving the mantissa $\sum_{l=1}^{52} b_l \times 2^{-l}$one digit to the right and rounding up 4. transform ln(x)/2 back by e^(ln(x)/2) = x^(1/2)

After 52 operations, ln(x) almost loses all its significant figures no matter where it started with. Only 2^0 and 2^1 survive which corresponds to e^1 and e^2. If we push even further, only 2^1 survives. This accounts for the steps we are seeing in the plots respectively.

Problem 7

Part (a) & (b)

```
In [3]: from sympy import *
        from sympy import poly
        x = symbols('x')
        # expand the multiplication to get the coefficients
        w_int = poly(expand((x-1)*(x-2)*(x-3)*(x-4)*(x-5)*(x-6)*(x-7)*(x-8)
               *(x-9)*(x-10)*(x-11)*(x-12)*(x-13)*(x-14)
               *(x-15)*(x-16)*(x-17)*(x-18)*(x-19)*(x-20)))
        # store the coeficients as floats
        w_float = [float(i) for i in w_int.coeffs()]
        print('Integer coefficients:')
        print(w_int.coeffs())
        print('Integer coefficients:')
        print(w_float)

Integer coefficients:
[1, -210, 20615, -1256850, 53327946, -1672280820, 40171771630, -756111184500, 11310276995381, -1
Integer coefficients:
[1.0, -210.0, 20615.0, -1256850.0, 53327946.0, -1672280820.0, 40171771630.0, -756111184500.0, 11
```

```
In [9]: from scipy.optimize import newton
        # func_w_int = np.polynomial.Polynomial(w_int.coeffs())
        # Why the above isn't working?
        # examing func_w_int(1) will give 160 instead of 0
        func_w_float = np.poly1d(w_float)
        res = newton(func_w_float, 21, maxiter=50000)
        print("Root searching with Newton's method:")
        print(res)
        # find roots using np.roots()
        print("Root searching with np.root:")
        print(np.roots(w_float))
```

```
Root searching with Newton's method:
9.585389646516598
Root searching with np.root:
[20.64758289+1.18692619j 20.64758289-1.18692619j 18.17159972+2.76910111j
 18.17159972-2.76910111j 15.20906456+2.88014671j 15.20906456-2.88014671j
 12.75239289+2.12817817j 12.75239289-2.12817817j 10.87966961+1.10932294j
 10.87966961-1.10932294j  9.58610174+0.j          9.09787506+0.j
  7.99519837+0.j          7.00021182+0.j          5.9999911 +0.j
  5.00000046+0.j          3.99999999+0.j          3.        +0.j
  2.        +0.j          1.        +0.j        ]
```

Part (c) & (d)

```
In [10]: for delta in (10**(-8), 10**(-6), 10**(-4), 10**(-2)):
             print('delta =', delta)
             w_float[0] = 1 + delta
             func_w_float = np.poly1d(w_float)
             res = newton(func_w_float, 21, maxiter=50000)
             print("Newton's method:", res)
             print("Root with largest real part using np.root:", np.roots(w_float)[0])
```

```
delta = 1e-08
Newton's method: 9.585389646516598
Root with largest real part using np.root: (20.647582887998496+1.1869261883090942j)
delta = 1e-06
Newton's method: 7.752713003402644
Root with largest real part using np.root: (23.149016041505767+2.740984630381872j)
delta = 0.0001
Newton's method: 5.969334849605957
Root with largest real part using np.root: (28.40021241591655+6.5104342165628175j)
delta = 0.01
Newton's method: 5.469592915093453
Root with largest real part using np.root: (38.478183617151515+20.83432358712749j)
```

```
In [11]: w_float[0] = 1.
         w_float[1] = - 210 - 2**(-23)
```

```
    func_w_float = np.poly1d(w_float)
    print(np.roots(w_float))
```

```
[20.84691022+0.j          19.50244237+1.94033198j 19.50244237-1.94033198j
 16.73074488+2.8126249j  16.73074488-2.8126249j  13.99240668+2.51882443j
 13.99240668-2.51882443j 11.79389059+1.65247714j 11.79389059-1.65247714j
 10.09545563+0.64493282j 10.09545563-0.64493282j  8.91581637+0.j
  8.00777203+0.j          6.99960207+0.j          6.00002048+0.j
  4.99999857+0.j          4.00000009+0.j          3.        +0.j
  2.        +0.j          1.        +0.j        ]
```

Comment: In both cases, the roots are changed and become complex.

7

(e) Ω is a map from space of n free coefficients to the space of its n roots

$\Omega = \Omega(a_0, a_1, \cdots a_{n-1})$

Ω_k = the k th roots of $p(x)$

$p(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_{n-1} x^{n-1} + x^n = 0$

(i) $(\text{Cond}\,\Omega_k)(\vec{a}) = \sum_{l=0}^{n-1} (T_{kl})(\vec{a})$

Δp denote the small change in p. $\Delta\Omega_k$ denote small change in x_k.
Δa_l denote small change in a_l Think of p as $p(a_l, x)$

$\frac{\Delta p}{\Delta a_l} = \frac{\partial p}{\partial a_l}(x=\Omega_k) = \Omega_k^l$

$\frac{\Delta p}{\Delta \Omega_k} = \frac{\partial p}{\partial x}(x=\Omega_k) = p'(\Omega_k) = \prod_{i \neq k} (\Omega_k - \Omega_i)$

$\therefore \quad \Delta a_l \cdot \Omega_k^l = \Delta\Omega_k \cdot p'(\Omega_k)$

$\varepsilon_{\Omega_k} = \frac{\Delta\Omega_k}{\Omega_k} = \frac{\Delta a_l \, \Omega_k^{l-1}}{p'(\Omega_k)} = \frac{a_l \Omega_k^{l-1}}{p'(\Omega_k)} \frac{\Delta a_l}{a_l} = \frac{a_l \Omega_k^{l-1}}{p'(\Omega_k)} \cdot \varepsilon_{a_l}$

$\therefore \quad T_{kl} = \frac{\varepsilon_{\Omega_k}}{\varepsilon_{a_l}} = \frac{a_l \Omega_k^{l-1}}{p'(\Omega_k)}$

$(\text{cond}\,\Omega_k)\,\vec{a} = \sum_{l=0}^{n-1} (T_{kl})(\vec{a}) = \frac{1}{p'(\Omega_k)} \sum_{l=0}^{n-1} a_l \Omega_k^{l-1} \quad \left(p'(\Omega_k) = \prod_{i \neq k} (\Omega_k - \Omega_i) \right)$

(ii) See pdf

```
In [2]: import numpy as np
        import math
```

Part (e)

```
In [12]: # r = 14
         for r in [14, 16, 17, 20]:
             p = (x-1)*(x-2)*(x-3)*(x-4)*(x-5)*(x-6)*(x-7)*(x-8)*(x-9)*(x-10)*(x-11)*(x-12)*(x-1
             p_prime = p/(x-r)
             denominator = p_prime.subs(x, r)
             sum = 0
             for i in range(0, 20):
                 sum = sum + w_int.coeffs()[20 - i]*(r**(i-1)) # Here using the int coefficients
             cond = sum/denominator
             print("cond%d = %10.2f" %(r, cond))
```

```
cond14 = -1332968434.15
cond16 = -2407513979.74
cond17 = 1904402145.26
cond20 = -43099804.12
```

A clever result cannot help us out. Actually when evaluating the $cond\Omega_k$ quantity, we assume the numbers are machine numbers and there is no rounding error. The large condition number does not come from the algorithm we use, rather, it is intrinsic. The problem itself is ill-conditioned.

Problem 8

8 (a)

$y_n = \frac{e - y_{n+1}}{n+1}$

Starting from y_N, which comes with ε_N

$y_{N-1} = \frac{e - y_N}{N} = \frac{e - (y_N + \varepsilon_N)}{N} = \frac{e - y_N}{N}(1 + \frac{\varepsilon_N}{e - y_N}) = y_{N-1}(1 + \varepsilon_{N-1})$

$y_{N-2} = \frac{e - y_{N-1}}{N-1} = \frac{e - (\frac{e - y_N}{N})}{N-1} = \frac{e}{N} + \frac{y_N}{N(N-1)}$

...

$\therefore\ y_K = \frac{K!}{N!} y_N$ + some function of N

$\therefore\ \varepsilon_K = \frac{K!}{N!} \varepsilon_N$

(b)

$\varepsilon = 1,\ y_N = 0$

$\varepsilon_K = \frac{K!}{N!} \varepsilon_N = \frac{K!}{N!}$

For $\varepsilon_K < \varepsilon_{K\text{-tolerance}}$ $\quad \frac{K!}{N!} < \varepsilon_{K\text{-tolarance}}$

$N > \sum_{i=0}^{K} i$

(c)

$\varepsilon_{K\text{-tolerance}} = \varepsilon \qquad N! > \frac{K!}{\varepsilon}$

$\hookrightarrow \varepsilon = 2^{-52} = 2.22 \times 10^{-16} \qquad N \geqslant 32$

(d) The first 16 digits match between 2 results (see pdf for detail)
Using backward recurrence the result already converges

```
In [13]: y1 = 0.
         n = 20
         for i in range(32, n, -1):
             y1 = (math.e - y1)/i
         print('Compute with recurrence relation:', y1)
         from scipy.integrate import quad
         def integrand(x):
              return x**n * math.e**x
         y2 = quad(integrand, 0, 1)
```

```
    print('Compute with built in function:', y2[0])
```

```
Compute with recurrence relation: 0.12380383076256993
Compute with built in function: 0.12380383076256996
```