

## Problem 1.

In approximation theory, there is an well-known result called Weierstrass theorem (1885). It says that: *given a continuous function  $f$  defined on an interval  $[a, b]$  and a prescribed error  $\epsilon$ , one can always approximate  $f$  by a polynomial on  $[a, b]$  such that the error is under  $\epsilon$ .* In this problem, we will find explicitly such a polynomial using Taylor polynomial (without invoking Weierstrass theorem).

1. Find a polynomial  $P$  such that

$$\max_{x \in [2, 4]} |\cos(x^2) - P(x)| < 10^{-3}.$$

Hint: use the fact that  $\cos(t) = 1 - \frac{t^2}{2!} + \frac{t^4}{4!} - \frac{t^6}{6!} + \dots$

2. Plot function  $f(x) = \cos(x^2)$  and function  $P(x)$  which you found in Part (a) on the interval  $[2, 4]$  on the same plot.

Note: the graphs might be too close to each other to distinguish.

## Solution

$$\cos(t) = \sum_{k=0}^{\infty} (-1)^k \frac{t^{2k}}{(2k)!} \xrightarrow{t=x^2} \cos(x^2) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{4k}}{(2k)!}$$

We can extract  $p_n$  by truncating the infinite series into a finite series. Our strategy is to find  $p_n$  that satisfies the error bound, then label this polynomial as  $P$ .

$$p_{4n}(x) = \sum_{k=0}^n (-1)^k \frac{x^{4k}}{(2k)!}.$$

The error of the  $(4n)$ 'th Taylor approximation of function  $f(x)$  is  $R_{4n}(x) = f(x) - p_{4n}(x)$ . It is difficult to estimate  $R_{4n}$  using Lagrange theorem for  $f$  because the higher derivatives of  $f$  are messy. Instead, we will go through function  $g(t) = \cos t$ . We know that

$$g(t) = \cos(t) = \sum_{k=0}^{\infty} (-1)^k \frac{t^{2k}}{(2k)!} = q_{2n}(t) + r_{2n}(t)$$

where  $q_{2n}(t) = \sum_{k=0}^n (-1)^k \frac{t^{2k}}{(2k)!}$ . Replace  $t$  by  $x^2$ ,

$$f(x) = g(x^2) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{4k}}{(2k)!} = q_{2n}(x^2) + r_{2n}(x^2).$$

Note that  $q_{2n}(x^2) = p_{4n}(x)$ . Thus, the remainder term of  $f$  and the remainder term of  $g$  are related to each other by  $R_{4n}(x) = r_{2n}(x^2)$ . We can use Lagrange theorem to estimate  $r_{2n}(t)$ .

$$r_{2n}(t) = \frac{g^{(2n+1)}(c)}{(2n+1)!} t^{2n+1}$$

for some  $c$  in between 0 and  $t$ . Then

$$R_{4n}(x) = r_{2n}(x^2) = \frac{g^{(2n+1)}(c)}{(2n+1)!} x^{2(2n+1)}$$

for some  $c$  in between 0 and  $x^2$ . We know that the derivatives of  $g$  can only be  $\cos, \sin, -\sin, -\cos$ , which are always in between  $-1$  and  $1$ . Thus,  $|g^{(2n+1)}(c)| \leq 1$ . We get

$$|R_{4n}(x)| \leq \frac{x^{4n+2}}{(2n+1)!}$$

We want to find  $n$  such that  $\frac{x^{4n+2}}{(2n+1)!} < 10^{-3}$  for all  $x \in [2, 4]$ . For this, we only need to find  $n$  such that

$$\frac{4^{4n+2}}{(2n+2)!} < 10^{-3}$$

With a calculator we can see that  $n \geq 24$  will do it. As we are looking for the associated polynomial degree, we require the 96'nd degree polynomial. Thus

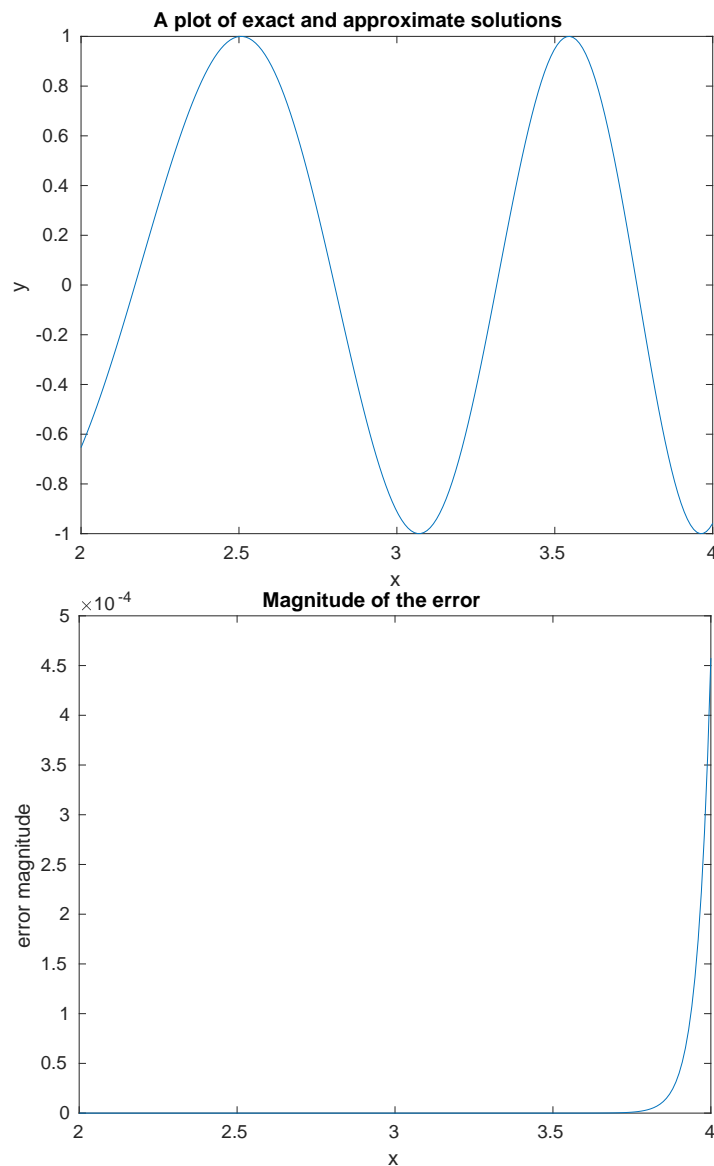
$$P(x) = \sum_{k=0}^{24} (-1)^k \frac{x^{4k}}{(2k)!}$$

We plot the value (required) and we cannot see both lines. To see just how good our approximation is, we can plot the absolute error.

```

1 x_vals = linspace(2,4,200); % Make some data
2 cos_exact = cos(x_vals.^2); % Exact values
3 cos_approx = P_sum(x_vals); % Series approximation
4
5 % Now let's make a figure of our solutions
6 f1 = figure();
7 plot(x_vals, cos_exact)
8 plot(x_vals, cos_approx)
9 xlabel('x')
10 ylabel('y')
11 title('A plot of exact and approximate solutions')
12 saveas(gcf, 'MTH_351_HW3_1A', 'eps')
13
14 % Let's look at the error
15 f2 = figure();
16 plot(x_vals, abs(cos_exact - cos_approx))
17 xlabel('x')
18 ylabel('error magnitude')
19 title('Magnitude of the error')
20 saveas(gcf, 'MTH_351_HW3_1B', 'eps')
21
22
23 function value = P_sum(x)
24     value = 0;
25     for degree = 0:23
26         value = value + (-1)^(degree) * x.^(4*degree)/(factorial(2*degree))
27         );
28     end
end

```



## Problem 2.

Consider the toy model of the IEEE double precision floating-point format as described in Homework 2. Perform the following operations on floating-point numbers. *Write your final answers in both floating-point format and decimal format.*

1.  $(1.001)_2 \times 2^2 + (1.100)_2 \times 2^4$
2.  $(0.010)_2 \times 2^{-6} + (1.001)_2 \times 2^2$
3.  $(1.101)_2 \times 2^7 + (1.000)_2 \times 2^7$
4.  $(0.001)_2 \times 2^{-3} \times (1.110)_2 \times 2^{-4}$

What do you notice when adding two numbers of quite different sizes?

### Solution

1.  $(1.001)_2 \times 2^2 + (1.100)_2 \times 2^4 = 100.1_2 + 110000_2 = 111001_2 = 1.11001_2 \times 2^4 \approx 1.110_2 \times 2^4 = 28$  (the exact value is 28.5)
2.  $(0.010)_2 \times 2^{-6} + (1.001)_2 \times 2^2 = 0.000000010_2 + 100.1_2 = 100.10000001_2 \approx 100.1_2 = 1.001_2 \times 2^2 = 4.5$  (exact value is 4.50390625).
3.  $(1.101)_2 \times 2^7 + (1.000)_2 \times 2^7 = 11010000_2 + 10000000_2 = 101010000_2 \approx 1.011 \times 2^8 = 352$ . This number is too big for our floating point format ( $8 > 7$ ) so the result is  $\infty$  in floating point format (the exact answer is 336 in decimal)
4.  $(0.001)_2 \times 2^{-3} \times (1.110)_2 \times 2^{-4} = 0.000001_2 \times 0.0001110_2 = 1.110 \times 2^{-10} = 0.001708984375$ . This value is too small to be stored in the floating point format ( $-10 < -7$ ) so the number stored in the floating point format is 0.

If two numbers are of different enough sizes, we will lose many (or all!) of the values from the smaller number when working in floating point format. We see this in 2 above, where  $4.5 + 0.00390625 = 4.5$  in floating point.

### Problem 3.

On an attempt to have Matlab compute the sum  $S = 0.1 + 0.2 + \dots + 0.9$ , someone writes the following code:

```
s = 0
x = 0
while x~=1.0
    s = s + x
    x = x + 0.1
end
S = s
```

1. Test this code on Matlab. Why does the program keep running indefinitely?  
*Note:* to terminate the procedure, place the cursor in the command window and press Ctrl + C.
2. What should be changed in the code to make it stop?

### Solution

This program runs indefinitely as the while loop never exists. This occurs when the logical test ( $x \sim= 1.0$ ) never returns true. The value of  $x$  is stored in a floating point format and is subject to truncation error. As a result, 0.1 is not *exactly* stored in the floating point format. Analytically, we require an infinity of values to store 0.1 ( $0.1 \approx 0.00011001100110011001101\dots_2$ ) even though we can store 1 with exactly one bit.

If we want the code to stop (which we generally do) we have several options.

1. Change  $x \sim= 1$  to  $x < 1$ .
2. perform the calculation with integer data types (integers can be compared exactly) and then divide to obtain a floating point result.
3. Use a for loop instead of a while-loop and precompute how many iterations you need to do (anything you can do with a 'for' loop you can do with a 'while' loop).

## Problem 4.

On an attempt to have Matlab compute the sum  $S = 1 + 2 + \dots + 9$ , a person writes the following code:

```
s = 0
x = 0
while x~=10
    s = s + x
    x = x + 1
end
S = s
```

1. Test this code on Matlab. Does the program keep running indefinitely?
2. What causes the difference compared to Problem 3?

## Solution

This program terminates (in finite time!) and does not run indefinitely.

We can represent sufficiently small integers exactly in a floating point format. Unlike above, we do not need to truncate  $x$  or  $s$  to fit into memory.

## Problem 5.

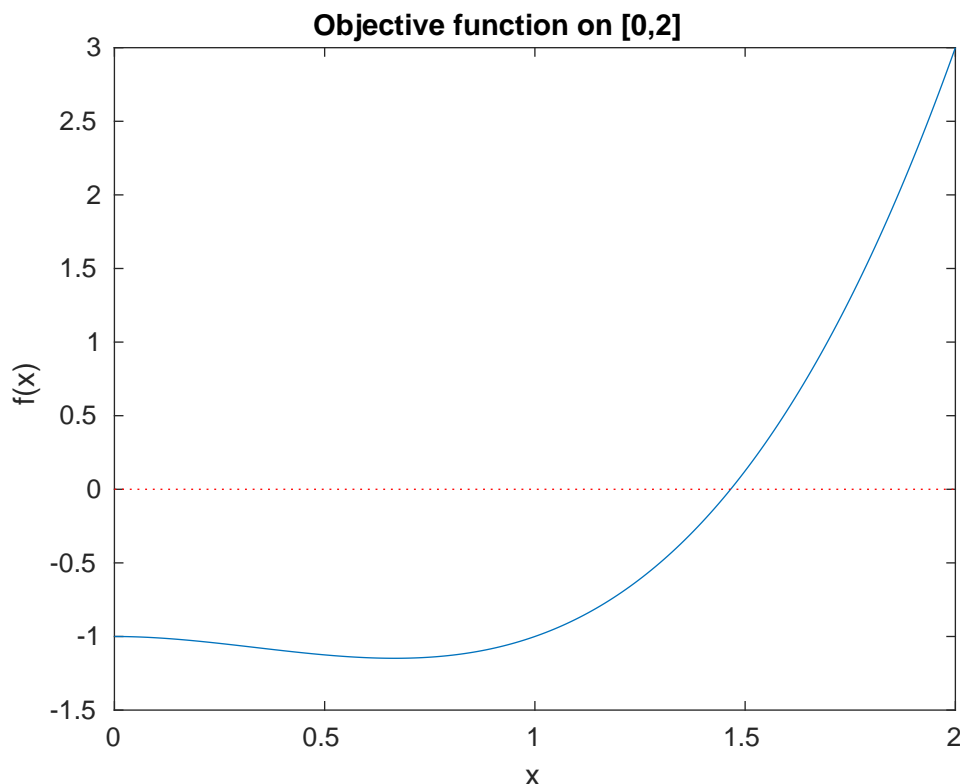
In this problem, we will compute approximately a real root of the equation  $x^3 - x^2 - 1 = 0$ .

1. Graph the function  $f(x) = x^3 - x^2 - 1$  on the interval  $[a_0, b_0] = [0, 2]$ .
2. Use the bisection method to find the interval  $[a_4, b_4]$ .
3. Approximate the root of  $f(x) = 0$  with error not exceeding  $10^{-2}$ .

## Solution

We can plot this in Matlab.

```
1 x_vals = linspace(0,2,200);
2 y_vals = f(x_vals); % Defined below
3 zero_vec = zeros(1,200);
4 f3 = figure();
5 plot(x_vals, y_vals) % The function
6 hold on
7 plot(x_vals, zero_vec, ':r') % Let's add the x-axis
8 xlabel('x')
9 ylabel('f(x)')
10 title('Objective function on [0,2]')
11 hold off
12 saveas(gcf, 'MTH_351_HW3_3A', 'eps')
13
14 function out = f(in)
15     out = in.^3 - in.^2 - 1;
16 end
```



We can see by inspection of the plot that the root should be near 1.5.

### Bisection method

We can collect our results into a table.

$n$	$a_n$	$b_n$	$\text{sgn}\left(f\left(\frac{a_n+b_n}{2}\right)\right)$
0	0	2	-
1	1	2	+
2	1	$\frac{3}{2}$	-
3	$\frac{5}{4}$	$\frac{3}{2}$	-
4	$\frac{11}{8}$	$\frac{3}{2}$	-

And we find  $[a_4, b_4] = \left[\frac{11}{8}, \frac{3}{2}\right]$ .

To determine how many iterations we must perform, we need to bound the error, so we need to find the smallest  $n$  such that

$$10^{-2} < \frac{1}{2^n} \cdot (2 - 0) = \frac{1}{2^{n-1}}$$

We can check this quickly in a calculator and find that  $n = 8$  is the smallest integer  $n$  such that we can guarantee that the root is within the interval. Thus  $c_8$  is a sufficient approximation. (You can s)

$n$	$a_n$	$b_n$	$\operatorname{sgn}\left(f\left(\frac{a_n+b_n}{2}\right)\right)$
4	$\frac{11}{8}$	$\frac{3}{2}$	-
5	$\frac{23}{16}$	$\frac{3}{2}$	+
6	$\frac{23}{16}$	$\frac{47}{32}$	-
7	$\frac{93}{64}$	$\frac{47}{32}$	-
8	$\frac{187}{128}$	$\frac{47}{32}$	-

(You may use a calculator to find the midpoints rather than working with fractions.) Our estimate is  $\frac{187}{128}$ . Computing the root  $x^* : f(x^*) = 0$  with a computer algebra system, we see that  $\left|\frac{187}{128} - x^*\right| = 0.0046337 \dots < 10^{-2}$ . as desired.