HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

----------***----------

MAJOR ASSIGNMENT REPORT

MODULE: CALCULUS 1

CODE: MT1003

Students: TRẦN NGUYỄN GIA PHÁT VƯƠNG GIA HÂN TẤT PHỤNG NHI TRƯƠNG ANH QUÂN PHAN LÊ TIẾN THUẬN VŨ LÊ BÌNH **Cohort: CC07** **Semester: 212** **Academic year: 2021-2022** **Lecturer: ĐẬU THẾ PHIỆT** **Submission date: 12th May 2022**	FOR EXAMINERS ONLY **Grade (in number):** ……………….. **Grade (in words):** ……………….. **Examiner 1** (Signature & Full name) ……………….. **Examiner 2** (Signature & Full name) ………………..

Ho Chi Minh City, May 2022

STATEMENT OF AUTHORSHIP

Except where reference is made in the text of the report, this assignment contains no material published elsewhere or extracted in a whole or in part from an assignment which we have submitted or qualified for or been awarded another degree or diploma.

No other person's work has been used without the acknowledgments in the report.

This report has not been submitted for the evaluation of any other models or the award of any degree or diploma in other tertiary institutions.

Ho Chi Minh City, May 2022

ZR

Tran Nguyen Gia Phat

[illegible]

Vuong Gia Han

[illegible]

Tat Phung Nhi

Thuận

Phan Le Tien Thuan

[illegible]

Truong Anh Quan

[illegible]

Vu Le Binh

ACKNOWLEDGEMENT

We would like to express our gratitude towards Mr Dau The Phiet, for helping us to coordinate in writing this report. We have explored meaningful findings during this work.

DISCLAIMER

Our goal is to create user-friendly, multi-purpose and interactive programs for convenience in providing versatile inputs and returning desired results. Therefore, robust codes and advanced commands/syntaxes used in the process which are out of this subject's scope are not explained in the central part of the report. For concise representation, only the snippets that contain necessary calculations are shown, not the complete code to handle edge cases, and may differ from the source codes in terms of arguments due to interactive purposes. Comments are minimized for readability.

Please only run the codes provided in the *Appendices* for demostration.

Readers may look up essential installation, required libraries and guidelines in the *Appendices*.

ABOUT THE AUTHORS

- Trần Nguyễn Gia Phát **2153681**
- Vương Gia Hân **2153339**
- Tất Phụng Nhi **2153660**
- Trương Anh Quân **2153751**
- Phan Lê Tiến Thuận **2153013**
- Vũ Lê Bình **2152440**

INTRODUCTION

Calculus is the broad area of mathematics dealing with such topics as instantaneous rates of change, areas under curves, sequences and series. Underlying all of these topics is the concept of limit, which consists of analyzing the behaviour of a function at points ever closer to a particular point, but without ever actually reaching that point. Calculus has two primary applications: differential calculus and integral calculus.

In the report, the writers will give explanations and solutions for problems involving (a) parametric equations, (b) arclength and trapezoidal rule and (c) linear differential equation and Euler's method for approximation.

TABLE OF CONTENT

I. Problem 1

1.1. Theories 5

1.2. Algorithms 6

1.3. Results and conclusion 8

II. Problem 2

2.1. Theories 10

2.2. Algorithms 13

2.3. Results and conclusion 16

III. Problem 3

3.1. Theories 18

3.2. Algorithms 20

3.3. Results and conclusion 22

IV. Conclusion 25

V. References 26

VI. Appendices 28

This report also provides coded calculators for solving arbitrary equations and approximate integral problems using numerical approximation theories. Numerical analysis is a branch of mathematics that solves continuous problems using numeric approximation. It involves designing methods that give approximate but accurate numeric solutions, which is helpful in cases where the exact solution is impossible or prohibitively expensive to calculate.

I. PROBLEM 1

Given the parametric equations

$$\begin{cases} x(t) = t^3 - 4t - 2 \\ y(t) = -2t^2 + 1 \end{cases}$$

Find a intersection of the curve and the tangents at that point. Draw the figure.

1.1. Theories

1.1.1. Intersection of a single Parametric Curve

For a graph of a parametric curve to intersect itself, there must be two distinct values of t, a and b such that when plugged in, the parametric equations will return the same result. This is because these two t values create two ordered pairs of points that overlap each other.

Moreover, there will be two distinct tangent lines at the intersection point on the graph.

1.1.2. Tangent lines for Parametric Curves

The slope of the tangent line for a curve $y = f(x)$ (in Cartesian coordinates) is:

$$\frac{dy}{dx} \text{ or } f'(x)$$

If the curve is given by parametric equations $x = f(t), y = g(t)$, we apply the chain rule:

$$\frac{dy}{dt} = \frac{dy}{dx} \cdot \frac{dx}{dt}$$

then the slope of its tangent line is: $\frac{dy}{dx} = \frac{\frac{dy}{dt}}{\frac{dx}{dt}}$ or $\frac{dy}{dx} = \frac{g'(t)}{f'(t)}$

The parametric curve's tangent line on the graph at a certain point (x_0,y_0) is given by the formula:

$$y = k(x - x_0) + y_0$$

where k is the slope of the line at that point ($k = \frac{g'(t_0)}{f'(t_0)}$ $where\ f(t_0) = x_0$)

Explicitly:

$$y = \frac{g'(t_0)}{f'(t_0)}.x - \frac{g'(t_0)}{f'(t_0)}.f(t_0) + g(t_0)$$

1.2. Algorithms

1.2.1. Find the self-intersection point of a parametric curve and the slopes of its tangent lines

Step 1: Let $x(a) = x(b)$, solve the equation for a in terms of b, only keep $real$ solutions and ignore the term $a = b$.

Step 2: Let $y(a) = y(b)$, substitute the solutions from step 1 to explicitly solve for b, ignore values that lead to $a = b$ and only keep $real$ solutions.

Step 3: Substitute remaining values from step 2 to $x(t)$ and $y(t)$ to find the intersection point. It is guaranteed that these values produce the same x and y values.

Step 4: For each value from step 2, calculate $\frac{dy}{dx} = \frac{y\prime(t)}{x\prime(t)}$ to get the slopes of the tangent lines at the intersection point. From that, write the corresponding linear equations.

Python’s implementation:

```python
def get_intersect(x,y):
    # return list of a in terms of b and store in 'b_implicit'
    b_implicit=[re.search('(?<=a\:\s)(.*)(?=})',str(i)).group(0) for i in
solve(f'({x.replace("t","a")})-({x.replace("t","b")})') if str(i)!='{a: b}']
    # substitute each solution to the other equation found to find b
values
    for sol in b_implicit:
      a_explicit = [i for i in solve(f'({y.replace("t",f"({sol})")})-
({y.replace("t","b")})') if type(i)!=ComplexRootOf]
      b = symbols('b')
      func = sympify(sol)
      # for each value of b, check and skip values that lead to a=b
      for j in a_explicit:
        if func.subs(b,j)!=j:
          t_set.append(j)
    # calculate intersection point's x and y coordinates
    point_x = Counter([x_t(t) for t in t_set]).most_common(1)[0][0]
    point_y = Counter([y_t(t) for t in t_set]).most_common(1)[0][0]
    # calculate the slopes at the intersection point
    dydx = [diff(y, t).subs(t, i)/diff(x, t).subs(t, i) for i in t_set if
x_t(i)==point_x]
    if 'I' in str(point_x) or 'I' in str(point_y):
      raise Exception("Invalid Imaginary Value")
    # return the point's coordinates as string and list of dy/dx values
    return f'({point_x},{point_y})',dydx
```

1,2.2. Find the tangent line, given the slope and the point

Python’s implementation:

```python
def get_tangent(dydx,x0,y0):
  return f"{dydx}*x-{dydx}*{x0}+{y0}"
```

1.2.3. Draw the figure

Python's code snippet:

```
#NOTE: Variables that are not mentioned below have been pre-computed
fig, ax = plt.subplots(constrained_layout=True, figsize=(9.5,6))
x = lambda t: eval(get_func(x_expr.value))
y = lambda t: eval(get_func(y_expr.value))
line_1 = lambda x: eval(res_tangent1.value)
line_2 = lambda x: eval(res_tangent2.value)
t_range = [t for t in np.linspace(-5,5, 1000)]
iteration = [i for i in np.linspace(eval(res_point.value)[0]-
10,eval(res_point.value)[0]+ 10, 1000)]
x_coor = [x(t) for t in t_range]
y_coor = [y(t) for t in t_range]
#plot parametric curve
ax.plot(x_coor,y_coor)
#plot intersect point
ax.plot(eval(res_point.value)[0], eval(res_point.value)[1])
#plot tangent lines
ax.plot([i for i in iteration],[line_1(i) for i in iteration])
ax.plot([i for i in iteration],[line_2(i) for i in iteration])
```

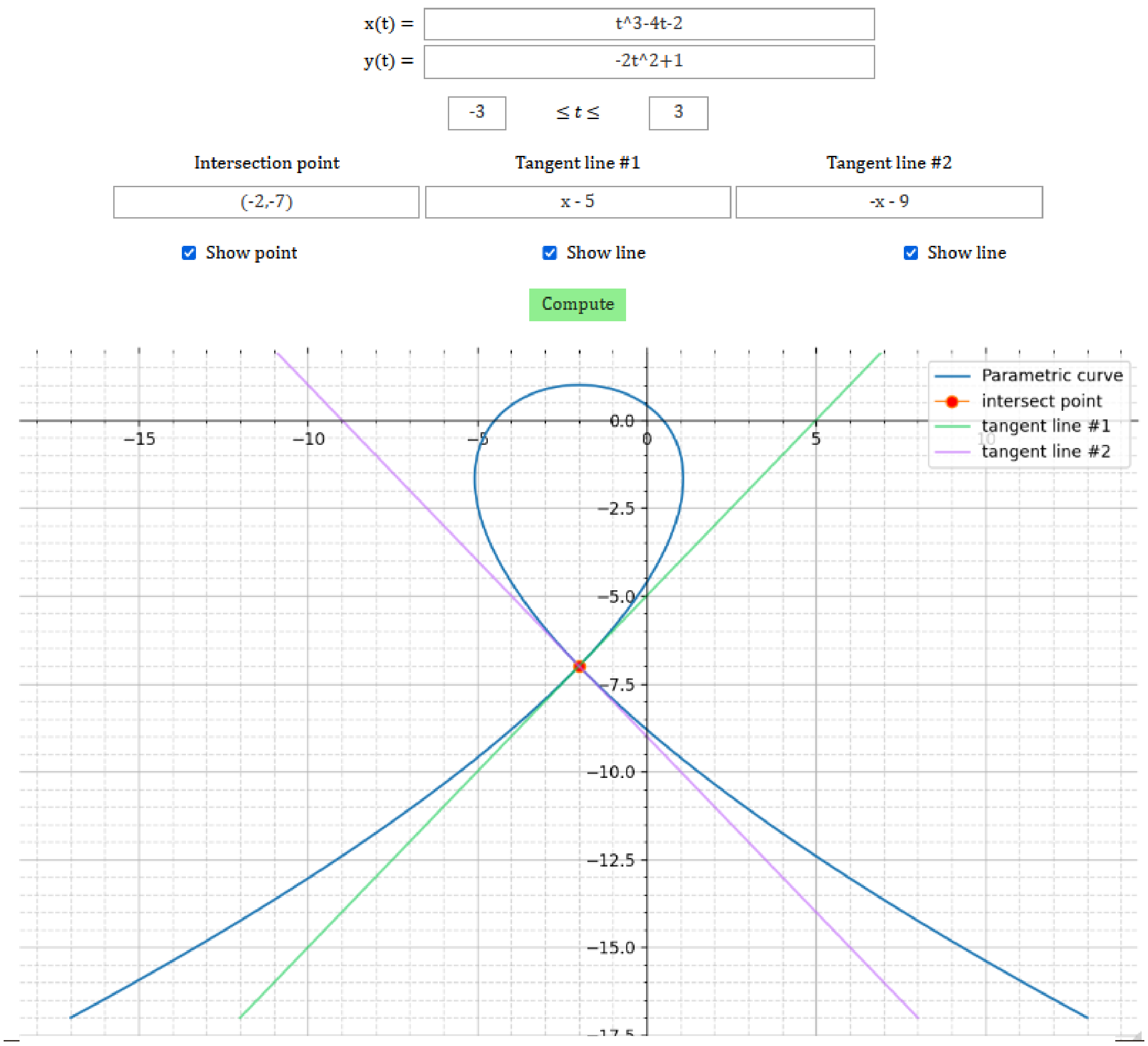
1.3. Results and conclusion

Follow the steps in *2.2.1.*, let $y(a) = y(b)$, we have $a = -b$.

Next, calculate $x(-b) = x(b)$ => $b \in \{-2,2\}$. Substitute in $x(t)$ & $y(t)$ we get the intersection point $(-2, -7)$.

The tangent lines at this point are $y = x - 5$ and $y = -x - 9$.

The coding result matches precisely with manual calculation and is the final result for Problem 1. See below figure:


x(t) = t^3-4t-2

y(t) = -2t^2+1

-3 ≤ t ≤ 3

Intersection point: (-2,-7)

Tangent line #1: x - 5

Tangent line #2: -x - 9

✓ Show point ✓ Show line ✓ Show line

Figure 1.1: Final result for Problem 1

This version of the program works best if the degree of both equations is lower than four and can throw unexpected errors with higher ones. Users may try different test cases to have a closer look. (e.g $x = t^2 - t,\ y = t^3 - 3t - 1\ /\ x = -2t^2 + 1,\ y = t^3 - 4t - 2$)

Features of the program:

- Accept casual writing style of expression (see above figure).
- Able to zoom in/out and move around the graph with the help of the toolbar.
- Able to adjust the range of t for better visualization.
- Toggle on/off the visibility of the point and tangent lines.

II. PROBLEM 2

A car moves along a path of street with the shape is the graph of the function $y = xe^x$. At each position, the light ray from the car is considered to be tangent to the street. There is a statue at the coordinates (0.35, 0.37) (km). Assume that the car moves at a constant velocity 60 km/h. How long does the car take in moving from *A*, that has *x*-coordinate = - 0.6 to *B*, where the light ray spot to the statue. The trapezoidal formula is a using to approximate the integral of a function. Study the formula and use it to approximate the path *AB*.

2.1. Theories

2.1.1. Arclength of a graph

In general, it is possible to approximate the length of a curve $y = f(x)$ between $x = a$ and $x = b$ by dividing it up into *n* small pieces and approximate each curved piece with a line segment.

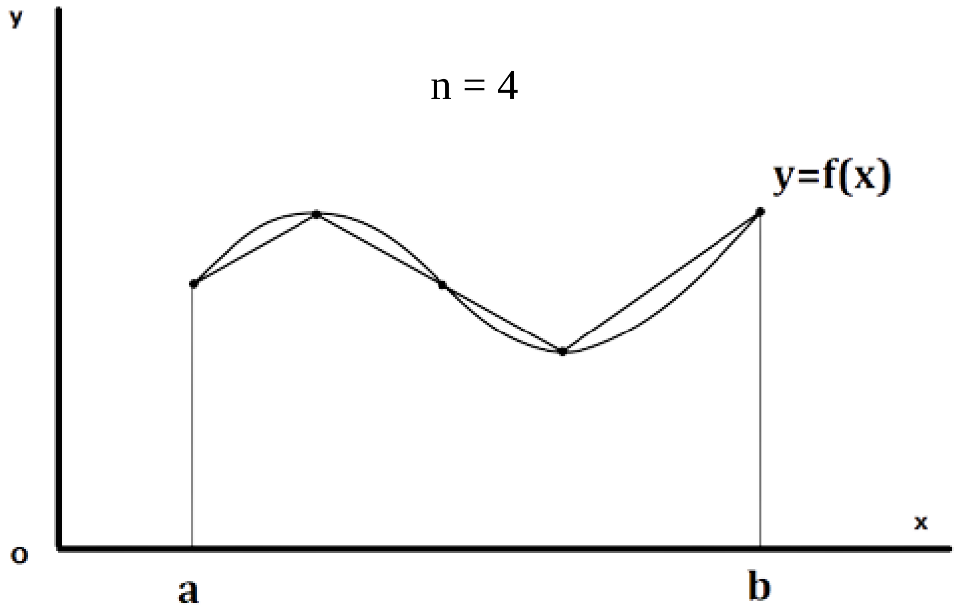


Figure 2.1: Approximation to a curve by 4 linear segments

Label the aribtrary point $P_i(x_i, f(x_i))$, then length of ith segment is:

$$\sqrt{(x_i - x_{i-1})^2 + (f(x_i) - f(x_{i-1}))^2}$$

Approximately,

$$\text{Length of curve} \approx \sum_{i=1}^{n} \sqrt{(x_i - x_{i-1})^2 + (f(x_i) - f(x_{i-1}))^2}$$

$$\approx \sum_{i=1}^{n} \sqrt{(x_i - x_{i-1})^2 + (f(x_i) - f(x_{i-1}))^2}.\frac{(x_i-x_{i-1})}{(x_i-x_{i-1})}$$

$$\approx \sum_{i=1}^{n} \sqrt{\frac{(x_i-x_{i-1})^2+(f(x_i)-f(x_{i-1}))^2}{(x_i-x_{i-1})^2}}.(x_i - x_{i-1})$$

$$\approx \sum_{i=1}^{n} \sqrt{1 + (f'(x_i^*))^2}.\Delta x \text{ (Riemann sum)}$$

Therefore, we derive a formula for arclength:

$$\text{Arclength } L = \lim_{n\to\infty} \sum_{i=1}^{n} \sqrt{1 + (f'(x_i^*))^2}.\Delta x = \int_a^b \sqrt{1 + (f'(x))^2}dx$$

2.1.2. Trapezoidal rule

In mathematics, especially in numerical analysis, the trapezoidal rule is a technique used for approximating the *definite integral* $\int_a^b f(x)$.

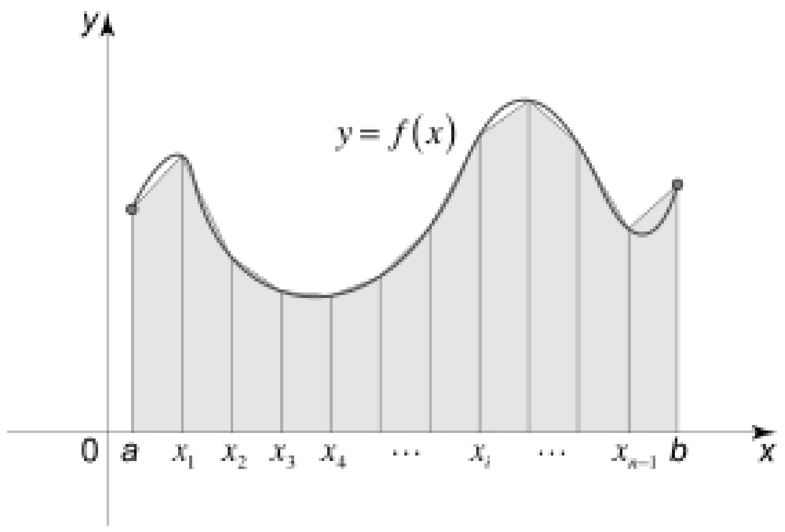


Figure 2.2: The concept of trapezoidal rule

The rule works by approximating the region under the graph of the function $f(x)$ as trapezoid and calculating its area. Let $\{x_k\}$ be a partition of $[a, b]$ such that $a = x_0 < x_1 < \cdots < x_{N-1} < x_N = b$ and $\Delta x_k = x_k - x_{k-1}$ be the length of the kth sub-interval, then:

$$\int_a^b \boldsymbol{f(x)}\,\mathbf{dx} \approx \sum_{k=1}^{N} \frac{\boldsymbol{f(x_k) + f(x_{k-1})}}{\mathbf{2}} . \boldsymbol{\Delta x_k}$$

For a domain divided into N equally spaced parts, let $\Delta x = \frac{b-a}{N}$ and $x_k = a + k\Delta x$, the approximation becomes:

$$\int_a^b f(x)\,\mathrm{dx} \approx \frac{\Delta x}{2}\sum_{k=1}^{N}\big(f(x_{k-1}) + f(x_k)\big) \approx \Delta x\left(\sum\nolimits_{k=1}^{N-1} f(x_k) + \frac{f(x_N) + f(x_0)}{2}\right)$$

<u>*2.1.3. Solving arbitrary equation with Secant method*</u> *(For later implementation)*

Newton's method: Let x_n is an approximate guess for a root of $f(x) = 0$, then the better approximation of the root is:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

The process is repeated until a sufficiently precise value is reached. However, when finding the derivative $f'(x_n)$ is problematic or time-consuming, we may adjust the method slightly, by using secant lines instead of tangents.

The slope of a secant line passing two points is: $\frac{\mathrm{f(x_n) - f(x_n - 1)}}{\mathrm{x_n - x_{n-1}}}$. Insert this expression for $f'(x_n)$ in Newton's method gives us the **Secant method**:

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

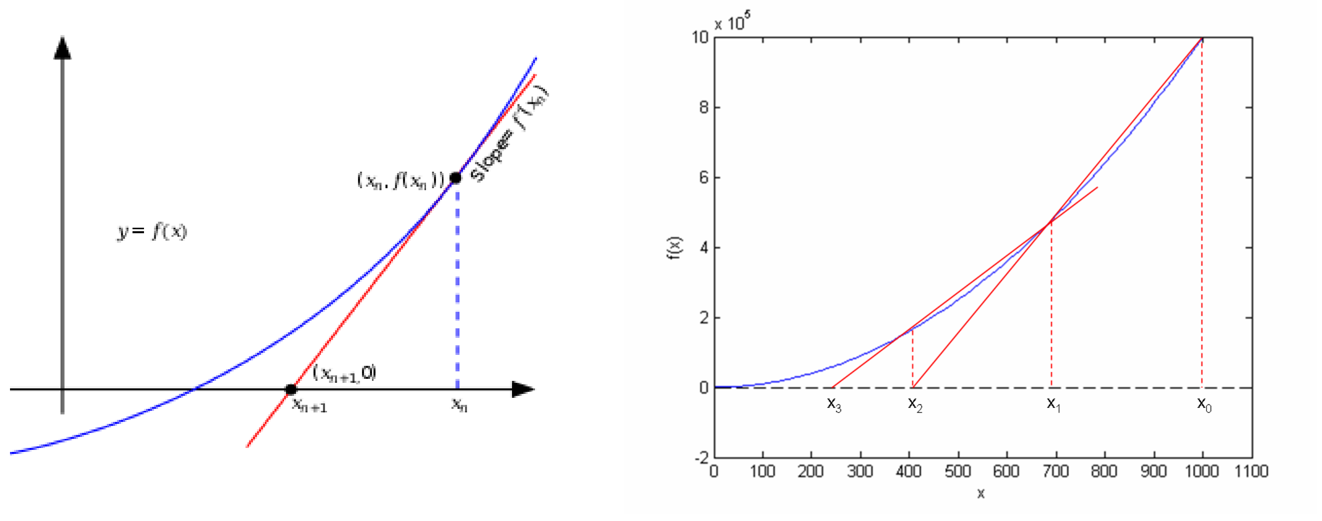


Figure 2.3: Newton's method with tangent lines vs. Secant method with secant lines

2.2. Algorithms

<u>*2.2.1. Find the satisfied stop position*</u>

The car x-position where its light can reach the statue is the root of below system of equations:

$$\begin{cases} f(x) = k(x - x_0) + y_0 \\ k = f'(x_0) \\ x_A < x < x_0 \end{cases}$$

where $f(x)$: the path expression, (x_0, y_0) : coordinates of the statue, x_A: Start A.

Python's implementation:

```python
#return the expression f(x) = f'(x)(x-x0) + y0
def location_equation(y,x0,y0):
    return f"{y}-({str(diff(y))})*x+({str(diff(y))})*{x0}-{y0}"
#solve the equation
def secant(y, a, b, eps=10e-7):
    try: #if the arbitrary equation cannot be solved by built-in solve(), 
then apply secant's method to find roots
        from sympy.solvers import solve
        return solve(y)
    except:
        f = lambda x: eval(y)
        f_a = f(a)
        f_b = f(b)
        iteration_counter = 0
        while abs(f_b) > eps and iteration_counter < 1000000:
            try:
                denominator = float(f_b - f_a)/(b - a)
                x = b - float(f_b)/denominator
            except ZeroDivisionError:
                return ""
            a = b
            b = x
            f_a = f_b
            f_b = f(b)
            iteration_counter += 1
        if x:
            return round(x, 4)
```

2.2.2. Approximate the arclengh of a function using trapezoidal rule

This function returns $\int_a^b f(x)\,\mathrm{dx} = \Delta x\left(\sum_{k=1}^{N-1} f(x_k) + \frac{f(x_N)+f(x_0)}{2}\right)$ where $f(x) = \sqrt{1+(y')^2}$ up to 3 decimal places precision, hence comments are redundant.

Python’s implementation:

```
def arclen_approx(y,a,b,N=10000):
    dx = str(diff(y))
    delta_x = (b-a)/N
    f= lambda x: eval(f"sqrt(1+({dx})*({dx}))")
    sigma = sum(f(a + k*delta_x) for k in range(1,N))
    return round(delta_x*(sigma +(f(a)+f(b))/2),3)
```

2.2.3. Draw the figure

Python’s code snippet:

```
#NOTE: Variables that are not mentioned below have been pre-computed
iteration = [t for t in np.linspace(eval(x1.value),eval(x2.value), 100)]
```

```
fx = get_func(path_expr.value)
y = lambda x: eval(fx)
f = location_equation(fx,eval(statue_xcoor.value),eval(statue_ycoor.value))
#Get the stop point
if type(secant(f,0,0))!=list:
      res = list(set(filter(None,(secant(f,i,i+1) for i in range(-5,5)))))
      res[:] =[i for i in res if i<eval(statue_xcoor.value)]
else:
      res = secant(f,-5,5)
      res[:]=[i for i in res if i<eval(statue_xcoor.value)]
b_xcoor = round(float(res[0]),4)
res_car.value = f'({b_xcoor},{round(y(b_xcoor),4)})'
#Get the path length
path = arclen_approx(fx,eval(start_x.value),b_xcoor)*1000
res_len.value = f"{path} m"
#Get the time to move from A to B if speed input is given
if speed.value:
      res_time.value = f"{round(path*3.6/eval(speed.value),2)} s"
fig,ax = plt.subplots(constrained_layout=True, figsize=(9.5,6))
#Plot drive path
ax.plot(iteration,[y(i) for i in iteration],label=f"Drive path")

#Visualize AB arclength
ax.plot([i for i in np.linspace(eval(start_x.value),b_xcoor)],
[y(i) for i in np.linspace(eval(start_x.value),b_xcoor)],label="AB")
#Plot car light
ax.plot([b_xcoor,eval(statue_xcoor.value)+0.1],
[y(b_xcoor),eval(statue_ycoor.value)+0.1],label="Car light")
#Plot start point A and stop point B
ax.plot(eval(start_x.value),y(eval(start_x.value)),label="Start A")
ax.plot(b_xcoor,y(b_xcoor),label="Stop B")
#Plot statue position
plot(eval(statue_xcoor.value),eval(statue_ycoor.value),label="Statue")
```

2.3. Results and conclusion

We have x-coordinate of B is the solution of the system of equations:

$$\begin{cases} xe^x = k(x-0.35)+0.37 \\ k = (xe^x)' = xe^x + e^x \\ x_A < x < 0.35 \end{cases}$$

From there we get B approximately close to the point (0.0292,0.0301).

$$AB = \int_{-0.6}^{0.0292} \sqrt{1+(xe^x+e^x)^2}\,dx \approx 0.736$$

⇨ AB $\approx 736m$. Convert $60km/h = \frac{50}{3} m/s$.

⇨ Time to go from A to B: $\frac{736}{50/3} \approx 44.16s$.

The coding result matches precisely with manual calculation and is the final result for Problem 2. See below figure:

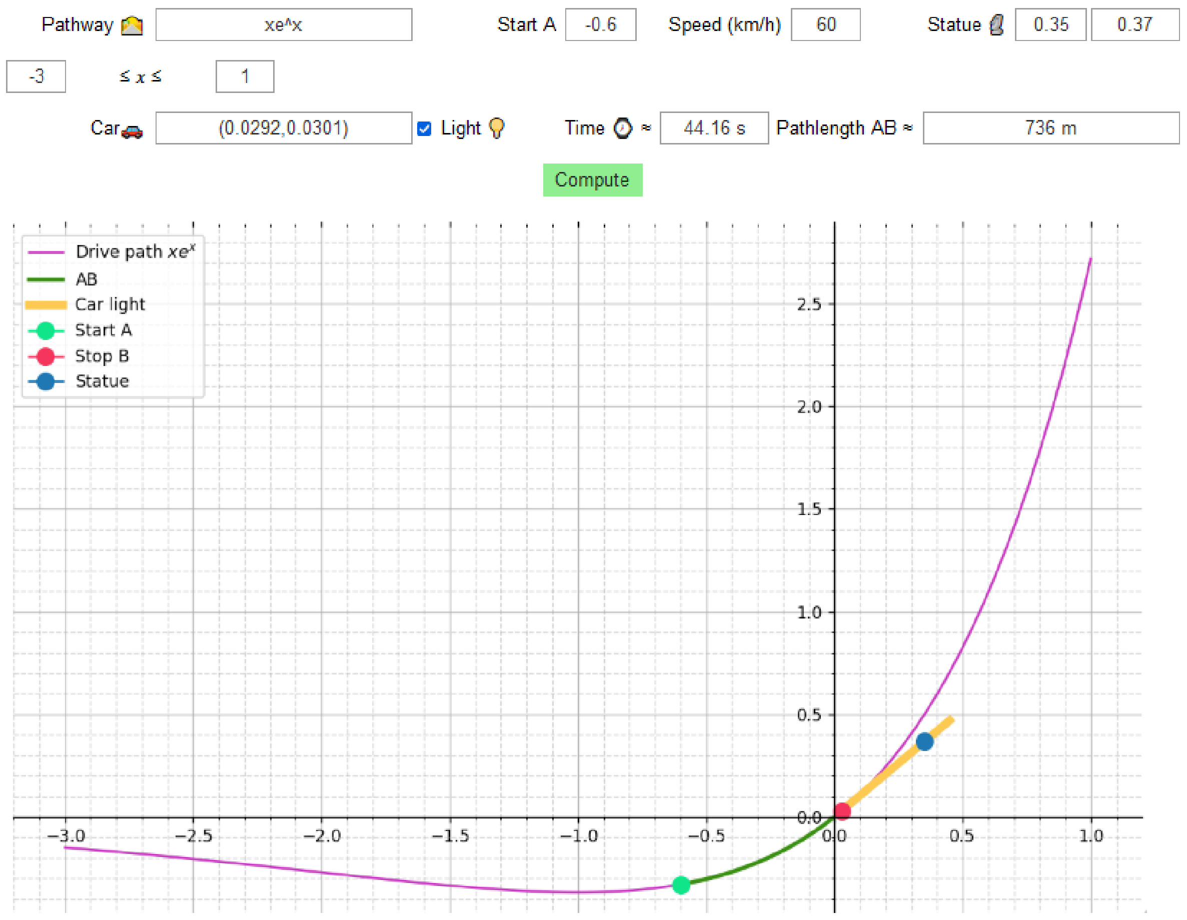


Figure 2.4: Final result for Problem 2

This program version works best if the input guarantees a valid outcome. The disadvantage is the slow run time to produce and plot output, up to 7-10s if the secant's method is used to approximate the equation's root. Users may try different test cases to have a closer look. (e.g $y = x^2 - 3$, statue's coordinates $(3,3)$)

Features of the program:

- Accept casual writing style of expression (see above figure).
- Able to zoom in/out and move around the graph with the help of the toolbar.
- Skip the move time if speed is not given.
- Able to adjust the range of x for better visualization.
- Toggle the car's light on/off.

Source codes for all problems are available at:

https://github.com/Zaphat/Calculus1-Project

III. PROBLEM 3

Study the Euler method to approximate the solution of first order differential equations. Program a calculator or computer to use Euler's method to compute y(1), where $y(x)$ is the solution of the initial-value problem

$$\frac{dy}{dx} + 3x^2y = 6x^2, y(0) = 3$$

Verify that $y = 2 + e^{-x^3}$ is the exact solution of the differential equation. Find the errors in using Euler's method to compute y(1) with the given step sizes. Draw the figure in describing the exact solution and approximated solution.

3.1. Theories

3.1.1. Exact solution for First Order Linear Differential Equation

In general, the linear differential equation is given by the standard form:

$$y' + p(x)y = q(x)$$

Let $P(x)$ be an inverse derivative of $p(x)$. Multiply both sides by $e^{P(x)}$, we have:

$$\Leftrightarrow y' e^{P(x)} + e^{P(x)}p(x)y = q(x)e^{P(x)}$$

$$\Leftrightarrow \left(\mathrm{y}e^{P(x)}\right)' = q(x)e^{P(x)}$$

$$\Leftrightarrow \mathrm{y}e^{P(x)} = \int q(x)e^{P(x)}dx$$

$$\Leftrightarrow \mathrm{y} = e^{-P(x)} \int q(x)e^{P(x)}dx$$

3.1.2.(Forward) Euler's method

Assume that $y = f(x)$ is written in the form of Taylor's Series:

$$y(x+h) \approx y(x) + hy'(x) + \frac{h^2y''(x)}{2!} + \cdots + \frac{h^ny^{(n)}(x)}{n!}$$

This gives us a reasonably good approximation if we expand plenty of terms and if the value h is reasonably small.

Let $\frac{dy}{dx}\Big|_{x=x_0} = f(x_0, y_0)$. For Euler's method, we only take the first 2 terms:

$$y(x+h) \approx y(x) + hf(x_0, y_0)$$

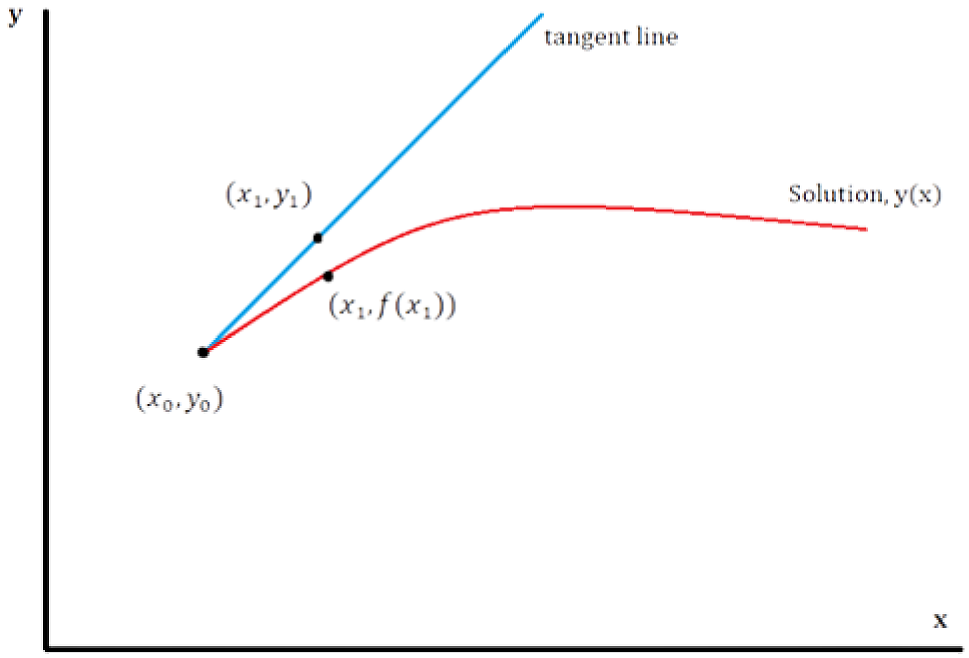


Figure 3.1: The concept of Euler's method

If x_1 is close enough the the initial value x_0, then the point y_1 is fairly equal to the actual value of the solution at x_1 or $f(x_1)$. To find y_1, we plug in x_1 to the tangent line equation:

$$y_1 = y_0 + f(x_0, y_0)(x_1 - x_0)$$

Often, we assume that the step size h between arbitrary points x_0, x_1,... x_n is uniformly-spaced $h = x_n - x_{n-1}$. Continue in this fashion, we derive the general fomula:

$$y_{n+1} = y_n + hf(x_n, y_n)$$

As $\lim\limits_{x\to 0} h$, the approximation gives $y_n \approx f(x_n)$. Therefore, as stated earlier:

$$f(x_n) \approx y_{n-1} + hf(x_n - h, y_n - h)$$

3.1.3.(Backward) Euler's method

The backward Euler method computes the approximations using:

$$f(x_n) \approx y_{n-1} + hf(x_n, y_n)$$

The difference from this and the forward Euler's method is that it uses $f(x_n, y_n)$ in place of $f(x_n - h, y_n - h)$.

3.1.4. Error analysis

The error of Euler's method is the absolute difference between exact value of the function and the approximate result, or:

$$Error\ \% = \frac{|exact - approximate|}{|exact|}.100$$

3.2. Algorithms

3.2.1. Solve for implicit/exact solution of linear differentiate equation

Step 1: Solve for implicit solution $\mathrm{y} = e^{-P(x)} \int q(x)e^{P(x)}dx$.

Step 2: If initial value $y(x_0)$ is given, then substitute to the implicit solution and calculate constant term C.

Python's implementation:

```
#Return the implicit solution for linear ode, and exact solution if
exact=True
def ode_linear(p_x,q_x,exact=False,x0=None,y0=None):
    x,y,C = symbols('x y C')
    P_x = integrate(p_x,x)
    implicit = f"exp(-({P_x}))*({integrate(f'exp({P_x})*({q_x})')}+C)"
    if exact:
        f = sympify(f'{implicit}-y')
        explicit =
implicit.replace('C',f'{solve(f.subs(x,x0).subs(y,y0),C)[0]}')
```

```
        return str(simplify(expand(explicit)))
    else:
        return str(simplify(expand(implicit)))
```

3.2.2: Approximation using Euler's method

If the initial $x_0 < x$, the function applies forward Euler's method, otherwise it uses the backward method (negative h).

While $x_0 \neq x$, the function repeatedly updates $y_0 = y_0 \mp hf(x_0, y_0)$ and $x_0 = x_0 \mp h$. When $x_0 = x$, then y_0 is the desired approximate result. Return y_0.

Python's implementation:

```

#if record=True, the function return 2 additional lists to track x and y
values
def ode_euler(f,x0,y0,x,h=0.01,record=False):
    dx = lambda x,y: eval(f)
if record:
        list_x,list_y=[],[]
        if x<x0:
            while(x0>x):
                list_x.append(x0)
                list_y.append(y0)
                y0 -= h*dx(x0,y0)
                x0 -= h

        else:
            while(x0<x):
                list_x.append(x0)
                list_y.append(y0)
                y0 += h*dx(x0,y0)
                x0 += h
        return y0,list_x,list_y
```

3.2.3: Draw the figure

Python's code snippet:

```
#NOTE: Variables that are not mentioned below have been pre-computed

fig,ax = plt.subplots(constrained_layout=True, figsize=(9.5,6))

solution_y =
ode_linear(p_x,q_x,exact=True,x0=eval(init_x0.value),y0=eval(init_y0.value))
 if init_x0.value and init_y0.value else  ode_linear(p_x,q_x)
f = lambda x: eval(solution_y)
val,list_x,list_y = ode_euler(f'({q_x})-
({p_x})*y',x0,y0,x,h=stepsize,record=True)
#Plot the exact solution
ax.plot([i for i in np.linspace(x0-0.5,x+0.5,100)],[f(i) for i in
np.linspace(x0-0.5,x+0.5,100)], label='Exact y')
#Plot the points of approximation
ax.scatter(list_x,list_y,color='#f7485f',s=7,label="Approximation points")
#Plot the approximation curve in comparison with exact curve
ax.plot(list_x,list_y,color='#4cf578',label='Approximation line')
```

3.3. Results and conclusion

Solution verification:

$$y' + 3x^2y = 6x^2$$

We have: $p(x) = 3x^2, q(x) = 6x^2, P(x) = x^3,$

$$y = e^{-P(x)}\int q(x)e^{P(x)}dx = e^{-x^3}\int e^{x^3}6xdx = 2 + Ce^{-x^3}$$

Substitute $y(0) = 3$:

$$y = 2 + e^{-x^3}$$

Finding errors with different step sizes:

$$y(1) = 2 + e^{-1} \;\approx\; 2.367879$$

- *Step size h=0.1:*

$$y(1) \approx 2.274956 \quad \text{Error \%: 3.9243 \%}$$

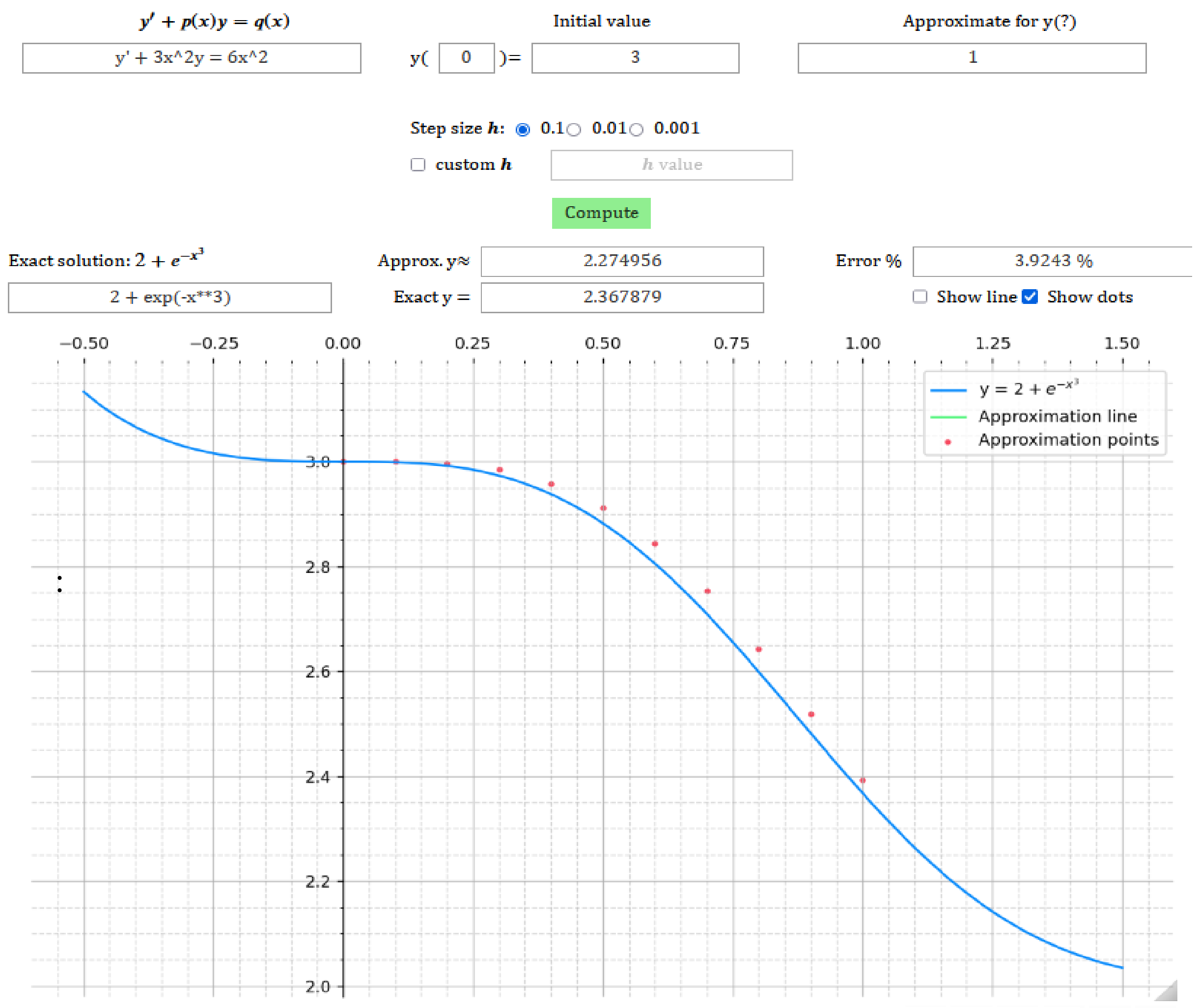
- *Step size h=0.01:*

$$y(1) \approx 2.370111 \quad \text{Error \%: 0.0943 \%}$$

- *Step size h=0.001:*

$$y(1) \approx 2.3681 \quad \text{Error \%: 0.0093 \%}$$

The coding result matches precisely with manual calculation and is the final result for Problem 3. See below figure:



$y' + p(x)y = q(x)$: y' + 3x^2y = 6x^2

Initial value: y(0)= 3

Approximate for y(?): 1

Step size h: [✓ 0.1] 0.01 0.001

☐ custom h: h value

Compute

Exact solution: $2 + e^{-x^3}$: 2 + exp(-x**3)

Approx. y≈ 2.274956

Exact y = 2.367879

Error % 3.9243 %

☐ Show line ☑ Show dots

Figure 3.2: Final result for Problem 3

This version of the program works best if the equation given is in the correct form. It is noticeable that the approximation points are closer to the exact y solution as h goes smaller. Users may try different test cases to have a closer look. (e.g $y' - y = 3x$, $y(0) = -1$) .

Features of the program:

- Accepts casual writing style of expression (see above figure).
- Able to zoom in/out and move around the graph with the help of the toolbar.
- Returns implicit solution if the initial value is not given.
- Allows customized h value.
- Able to calculate both Euler’s forward and backward method.
- Toggle on/off line connection between points of approximation (unavailable when $h < 0.001$).
- Toggle on/off points of approximation.

Source codes for all problems are available at:

https://github.com/Zaphat/Calculus1-Project

IV. Conclusion

In the real world, various problems involve solving differential equations. Nonetheless, even if we can solve some of them algebraically, it is nearly impossible to find the exact solution in most cases. As a result, the numerical approach is the best suit in such cases to give us a good approximation within an acceptable tolerance. With the help of computer programs, numerical methods have surpassed the analytical ones to become the most powerful tool for solving any generic problem.

While the calculator programs written for this report might be helpful in finding solutions on most occasions, there are still issues that need improvement for better quality and performance.

Advantages of the calculators:

- Using advanced regular expression structures to parse informal-written expressions for solving.
- Interactive plots and versatile input options.
- Robust and optimized code for edge cases to minimize errors.
- Time-saving and convenient UI/widgets for multiple inputs at a time.

Limitations of the calculators:

- Occasionally throw unexpected errors that need restarting kernel.
- Slow run time if the expression is complicated (see Problem 2).
- Figures occasionally lose visibility when zoomed in and need re-running the cell.

V. References

1. Beerli, P. (n.d.). *Matplotlib - 2D and 3D plotting in python* . Retrieved May 3, 2022, from http://www.peterbeerli.com/classes/images/2/26/Isc4304matplotlib6.pdf

2. Bourne, M. (n.d.). *11. euler's method - a numerical solution for differential equations*. intmathcom RSS. Retrieved May 3, 2022, from https://www.intmath.com/differential-equations/11-eulers-method-des.php

3. Brownlee, J. (2018, March 14). *Analytical vs numerical solutions in machine learning*. Machine Learning Mastery. Retrieved May 3, 2022, from https://machinelearningmastery.com/analytical-vs-numerical-solutions-in-machine-learning/
4. Chow, S. (2018, October 18). *Euler's method, Intro & example*. Retrieved May 3, 2022, from https://www.youtube.com/watch?v=Pm_JWX6DI1I

5. *Euler's Method*. Differential equations - euler's method. (n.d.). Retrieved May 3, 2022, from https://tutorial.math.lamar.edu/classes/de/eulersmethod.aspx

6. F., F. J. E. (2009). *Mastering regular expressions*. O'Reilly.

7. *Finding where a parametric curve intersects itself*. Mathematics Stack Exchange. (n.d.). Retrieved May 3, 2022, from https://math.stackexchange.com/questions/246442/finding-where-a-parametric-curve-intersects-itself

8. Green, D. L. (n.d.). *Calculus 2 - full college course*. Retrieved May 3, 2022, from https://www.youtube.com/watch?v=7gigNsz4Oe8

9. *How to calculate indefinite integral in Python*. (2019, April 1). Retrieved May 3, 2022, from http://how.okpedia.org/en/python/how-to-calculate-an-indefinite-integral-in-python

10. *Ipywidgets with matplotlib*. Kapernikov. (2021, July 28). Retrieved May 3, 2022, from https://kapernikov.com/ipywidgets-with-matplotlib/

11. Libretexts. (2022, May 2). *Introduction to calculus*. Mathematics LibreTexts. Retrieved May 3, 2022, from https://math.libretexts.org/Bookshelves/Precalculus/Precalculus_(OpenStax)/12%3A_Introduction_to_Calculus

12. *Newton's Method*. (n.d.). Retrieved May 3, 2022, from https://tutorial.math.lamar.edu/classes/calci/newtonsmethod.aspx

13. Quora. (n.d.). *Why do we do a numerical approximation?* Retrieved May 3, 2022, from https://www.quora.com/Why-do-we-do-a-numerical-approximation

14. *Solving nonlinear algebraic equations*. (n.d.). Retrieved May 3, 2022, from http://hplgit.github.io/prog4comp/doc/pub/p4c-sphinx-Python/._pylight007.html

15. Swearingen, R. (n.d.). *Finding the equations of tangent lines where a parametrically defined* ... Retrieved May 3, 2022, from https://www.youtube.com/watch?v=IASowcfZi7Q

16. Wikimedia Foundation. (2022, April 2). *Newton's method*. Wikipedia. Retrieved May 3, 2022, from https://en.wikipedia.org/wiki/Newton%27s_method

17. Wikimedia Foundation. (2022, February 26). *Secant method*. Wikipedia. Retrieved May 3, 2022, from https://en.wikipedia.org/wiki/Secant_method

VI. Appendices

Download the source codes for Problem 1, 2 and 3:

1. Go to: https://github.com/Zaphat/Calculus1-Project

2. Select Code, click on Download ZIP and unzip the downloaded file.

Install Jupyter Notebook using Anaconda:

1. Go to: https://www.anaconda.com/ and download the latest version.

2. Install and open Anaconda, then launch Jupiter Notebook.

Install Python:

1. Go to https://www.python.org/downloads/ and download the latest version.

2. Install and restart the computer.

3. Add Python to system PATH: https://www.javatpoint.com/how-to-set-python-path

Install Python libraries:

1. Run Command Prompt as administrator.

2. Type each line below to the Command Prompt and hit Enter:

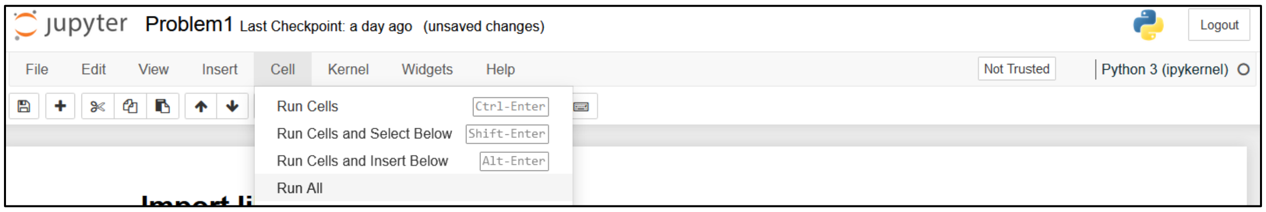
```
pip install sympy
pip install ipywidgets
pip install numpy
pip install matplotlib
pip install ipython
```

Lauch the code:

1. From Start Menu browse for Anaconda folder and click on Jupyter Notebook, a web page will be opened in the default browser.

2. Navigate to where the code files are located, click on the file name to open.

3. From Cells select Run All, after that the program is ready to use.


jupyter
Problem1 Last Checkpoint: a day ago (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Not Trusted
Python 3 (ipykernel)
Run Cells Ctrl-Enter
Run Cells and Select Below Shift-Enter
Run Cells and Insert Below Alt-Enter
Run All

Video tutorial: https://youtu.be/lRqOPPG0d0w

Frequently used methods/syntaxes that are excluded in the main body:

1. get_func(s): Returns parsed string from informal-written expression to valid symbolic format.

```
#Parse input string to the right format
def get_func(s):
    import re
    # step1: trim all whitespaces
    s = re.sub(r'\s+',"",s).replace('e**','e^')
    #step2: convert e^() to exp(), e^__ to exp(x), ln__ to ln()
    s = re.sub(r"[e]\^\((\w*)",r"exp(\1",s)
    s = re.sub('e\^([0-9a-z.]*)','exp(\g<1>)',s)
    s = re.sub(r"ln([0-9a-z.()]*)",r'ln(\g<1>)',s)
    #step3.1: add * between 2 consecutive alphabetical characters | a digit
and an alphabetical character
    s = re.sub('(?i)(?<=[a-z0-9])(?=[a-z])',r'*',s)
    #step3.2: add * between an alphabetical character and a digit
    s = re.sub('(?i)(?<=[a-z])(?=[0-9])',r'*',s)
    #step3.3 add * between number and open bracket
    s = re.sub('(?<=[a-z0-9])(?=[([{])',r'*',s)
    #step3.4 add * between closing bracket and number or open bracket, fix
error from step3
    s = re.sub('(?:(?<=[)])|(?<=[}])|(?<=[]]))(?:(?=([a-z0-
9]))|(?=[({[]))',r'*',s)
    s =
s.replace("l*n*","ln").replace("l*o*g*","log").replace("t*a*n*","tan").replac
e("c*o*s*","cos").replace("s*i*n*","sin").replace("e*x*p*","exp").replace("^"
,"**").replace('-','-')
    return s
```

2. eval(): parses the expression passed to this method and runs python expression (code) within the program.

3. sympy. sympify(): converts an arbitrary expression to a type that can be used inside SymPy.

4. sympy.simplify(): simplifies any mathematical expression.

5. sympy.expand(): expands brackets of any mathematical expression.

6. lambda : an anonymous function is a function that is defined without a name.

For example:

```
f = lambda x: eval('x**2')
```

is equal to

```
def f(x):
    return x*x
```

7. List comprihension: an elegant way to define and create lists based on existing lists.

For example:

```
a = [i for i in range(0,5)]
will store a list ‘a’ of [1,2,3,4]
```

8.Ternary operator: an elegant way to avoid writing lengthy branches conditions. The syntax expression is:

```
a if condition else b
```

Further documentation for *ipywidgets*: https://ipywidgets.readthedocs.io/en/stable/

Further documentation for *matplotlib*: https://matplotlib.org/stable/index.html