

Task 3 (40% of total marks)

Part A (20% of total marks)

Apply root-solving methods to calculate n_5 (Oxygen Balance) with Gibbs free energy. You are given a basic code script and a function

1. "Task_3.m" this is the basic code script that creates the variables and calls the function.
2. "mydissfunii.m" this is the function to calculate n_5 .

The code script calls a function that calculates n_5 , this is fundamental in developing a two-zone model. You should improve the function by applying root-solving methods. The following is to be submitted for part A

- PDF of results table for each applied method. This table should contain the number of iterations, computation time and, comparison of zero tolerance (you will need to add an iteration counter to the original function and "tic" "toc" to the Task_3.m file for calculating these comparisons).
- Your code files. Any functions or code you made to complete Part A

Newton and Bi-section functions have been used .

Newton method to calculate N_5

tic toc and number of iterations have been worked out for each method used for comparison

```
clc;
clear;

% Define constants
c = 8;
h = 18;
o = 0;
airs = 12.5;

% Define input ranges (same as original)
temp = (100:25:3000)'; % 49 temperature values
press = (20:5:120)'; % 21 pressure values
feq = (0.8:0.02:1.3)'; % 26 equivalence ratio values

% Initialize output array for  $n_{5nt}$  values
n5nt = zeros(49, 21, 26);

% Start timing
tic;

% Loop over all values (same as original structure)
for i = 1:49
    for j = 1:21
        for b = 1:26
```



```
0.03962
0.03962
0.03962
0.03962
0.03962
0.03962
0.03962
0.03962
0.03962
```

Bi-section method to calculate N5nt

```
clc;
clear;

% Given constants from function
c = 8;
h = 18;
o = 0;
airs = 12.5;
temp = (100:25:3000)'; % Temperature range
press = (20:5:120)'; % Pressure range
feq = (0.8:0.02:1.3)'; % Fuel equivalence ratio

% Imake results matrix
n5nt = zeros(49, 21, 26);
error_matrix = zeros(49, 21, 26);
tic
% Compute using bisection method
for i = 1:49 % Temperature index
    for j = 1:21 % Pressure index
        for b = 1:26 % Fuel equivalence ratio index
            [n5nt(i, j, b), error_matrix(i, j, b)] = mydissfun_bisection(c, h, o,
temp(i), feq(b), airs, press(j)); % the function
        end
    end
end
time_elapsed = toc;
% Display the first few values
fprintf('Total Computation Time: %10.6f seconds\n', time_elapsed);
```

```
Total Computation Time: 1.279907 seconds
```

```
disp(['Number of iterations: ', num2str(j)]);
```

```
Number of iterations: 21
```

```
disp('Computed n5nt values using Bisection Method:');
```

```
fprintf( '%.5f\n', n5nt(:,1,1));
```

4

Method	Iterations	Time Taken	First value of N5nt
Bi-section	21	0.002825	0.03962
Newton	21	1.2799	0.03962

The table presents a comparison between the **Bisection Method** and **Newton's Method** for computing the first value of $N5NTN_{\{5NT\}N5NT}$. Both methods required **21 iterations**, indicating similar convergence behavior in this particular case. However, a significant difference is observed in the **time taken** for computation. The Bisection Method completed in **0.002825 seconds**, whereas Newton's Method took **1.2799 seconds**, making it much slower in this instance. Despite the difference in execution time, both methods resulted in the same first value of $N5NTN_{\{5NT\}N5NT}$ (**0.03962**). This suggests the Newton function is more suitable for this type of problem.

Part B (20% of total marks)

Research MATLAB functions and mathematical techniques to calculate n_5 , there are many chemistry linear algebra solvers, symbolic maths solvers and engineering root solvers. You should create a table of methods ranking them based on criteria such as ease of use, efficiency, and accuracy. This table should be referenced. Then you are to attempt to calculate n_5 with one of the methods researched. The following is to be submitted for Part B

- PDF table with references for each researched method
- Your code files for the application of your chosen researched solution.

Intro

Comparison of Root-Finding Methods for Calculating n_{5n_5}

In this study, several MATLAB solvers were analyzed to determine their effectiveness in calculating n_{5n_5} . The solvers used include **Newton's Method (fzero)**, **fsolve**, **vpasolve**, **lsqnonlin**, **fminunc**, and **Bisection Method**. These methods were evaluated based on three key performance metrics: **number of iterations**, **execution time**, and **residual error**.

The **number of iterations** is a crucial factor in assessing efficiency, as fewer iterations generally indicate faster convergence. The **execution time** was measured to compare computational efficiency, highlighting which methods are more practical for large-scale computations. Finally, the **residual error** was used to assess accuracy, ensuring the computed root is sufficiently close to the true solution.

A **weighted scoring system** was applied, where execution time and iterations were given higher importance due to their impact on computational efficiency, while residual error contributed to overall accuracy. Based on this analysis, the most effective solver was selected for final implementation. The results of this comparison, including the numerical values for each criterion, are presented in the accompanying table.

Newtons

Newton's Method is a way to estimate the roots of a function using an iterative process. It starts with an initial guess and improves it by using the function's derivative. When conditions are favorable, the method converges quickly, making it highly efficient. However, its success depends on the choice of the starting point and the behavior of the function. If the derivative is zero at the initial guess or the function is irregular, the method might not work properly or fail to converge (Liu, Y. n.d.).

```
clear,clc, close all
%% Newton's Method
%% Newton's Method
% Counter system, time for completion (tic toc)
a = 1;
b = 4;
tol = 1e-6;
imax = 50;
syms x
Derv1_sym = diff(exp(x) - (1 + x + ((x^2)/2)));
x_guess = 5;

% Timer
tic; % Start time

% Call Newton's method once, storing only the final correct root
[x_next, counter] = Newton(@Function1, @Derv1, imax, tol, x_guess);
```

```
ITERATION 1
First Guess = 5.000000
Next guess = 4.087773
```

```
ITERATION 2
First Guess = 4.087773
Next guess = 3.241020
```

```
ITERATION 3
First Guess = 3.241020
Next guess = 2.487381
```

```
ITERATION 4
First Guess = 2.487381
Next guess = 1.849522
```

```
ITERATION 5
First Guess = 1.849522
Next guess = 1.337187
```

```
ITERATION 6
First Guess = 1.337187
Next guess = 0.944907
```

```
ITERATION 7
First Guess = 0.944907
Next guess = 0.656151
```

```
ITERATION 8
First Guess = 0.656151
Next guess = 0.449883
```

```
ITERATION 9
```

First Guess = 0.449883
Next guess = 0.305706

ITERATION 10
First Guess = 0.305706
Next guess = 0.206451

ITERATION 11
First Guess = 0.206451
Next guess = 0.138834

ITERATION 12
First Guess = 0.138834
Next guess = 0.093096

ITERATION 13
First Guess = 0.093096
Next guess = 0.062307

ITERATION 14
First Guess = 0.062307
Next guess = 0.041646

ITERATION 15
First Guess = 0.041646
Next guess = 0.027812

ITERATION 16
First Guess = 0.027812
Next guess = 0.018563

ITERATION 17
First Guess = 0.018563
Next guess = 0.012385

ITERATION 18
First Guess = 0.012385
Next guess = 0.008261

ITERATION 19
First Guess = 0.008261
Next guess = 0.005509

ITERATION 20
First Guess = 0.005509
Next guess = 0.003674

ITERATION 21
First Guess = 0.003674
Next guess = 0.002449

ITERATION 22
First Guess = 0.002449
Next guess = 0.001633

ITERATION 23
First Guess = 0.001633
Next guess = 0.001089

ITERATION 24
First Guess = 0.001089
Next guess = 0.000726

ITERATION 25

```
First Guess = 0.000726
Next guess = 0.000484
```

```
ITERATION 26
First Guess = 0.000484
Next guess = 0.000323
```

```
ITERATION 27
First Guess = 0.000323
Next guess = 0.000215
```

```
ITERATION 28
First Guess = 0.000215
Next guess = 0.000143
```

```
ITERATION 29
First Guess = 0.000143
Next guess = 0.000096
```

```
ITERATION 30
First Guess = 0.000096
Next guess = 0.000064
```

```
ITERATION 31
First Guess = 0.000064
Next guess = 0.000043
```

```
ITERATION 32
First Guess = 0.000043
Next guess = 0.000028
```

```
ITERATION 33
First Guess = 0.000028
Next guess = 0.000019
```

```
ITERATION 34
First Guess = 0.000019
Next guess = 0.000011
```

```
ITERATION 35
First Guess = 0.000011
Next guess = 0.000008
```

```
ITERATION 36
First Guess = 0.000008
Next guess = 0.000001
```

```
ITERATION 37
First Guess = 0.000001
Next guess = 0.000001
```

```
execution_time = toc; % Get execution time in seconds

% Compute the final residual error (only for the last iteration)
E_residual_last = abs(Function1(x_next));

% Display results
disp(['Root found by Newton method: ', num2str(x_next)]);
```

Root found by Newton method: 8.6219e-07

```
disp(['Number of iterations: ', num2str(counter)]);
```

Number of iterations: 37

```
disp(['Execution Time: ', num2str(execution_time), ' seconds']);
```

Execution Time: 0.10464 seconds

```
disp(['Final Residual Error: ', num2str(E_residual_last)]); % Only display last residual error
```

Final Residual Error: 0

Fzero

The `fzero` function in MATLAB is used to find the root of a single-variable function. It combines bisection, secant, and inverse quadratic interpolation methods to determine a zero. Its efficiency depends on the function's characteristics and the chosen initial guess or interval (MathWorks ,n.d.)

```
%% fzero method
% Define the function
Function1 = @(x) exp(x) - (1 + x + (x.^2)/2); % Ensure element-wise operations using .^

% Choose an initial guess
x_guess = 5;

% Set tolerance and display iterations
tol = 1e-6;
options = optimset('TolX', tol, 'Display', 'iter');
tic

% Use fzero to find the root with specified tolerance and tracking iterations
[rootfzero, fval, exitflag, iterations_fzero] = fzero(Function1, x_guess, options);
```

Search for an interval around 5 containing a sign change:

Func-count	a	f(a)	b	f(b)	Procedure
1	5	129.913	5	129.913	initial interval
3	4.85858	111.179	5.14142	151.6	search
5	4.8	104.19	5.2	161.552	search
7	4.71716	95.0069	5.28284	176.692	search
9	4.6	83.3043	5.4	200.426	search
11	4.43431	69.0284	5.56569	239.25	search
13	4.2	52.6663	5.8	306.68	search
15	3.86863	35.5249	6.13137	434.138	search
17	3.4	19.7841	6.6	705.715	search
19	2.73726	7.96103	7.26274	1391.52	search
21	1.8	1.62965	8.2	3598.13	search
23	0.474517	0.0201375	9.52548	13648.7	search
24	-1.4	-0.333403	9.52548	13648.7	search

Search for a zero in the interval [-1.4, 9.52548]:

Func-count	x	f(x)	Procedure
------------	---	------	-----------

24	-1.4	-0.333403	initial
25	-1.39973	-0.333231	interpolation
26	-0.884305	-0.0936916	interpolation
27	-0.884305	-0.0936916	bisection
28	-0.876268	-0.0913209	interpolation
29	-0.56712	-0.0265361	interpolation
30	-0.56712	-0.0265361	bisection
31	-0.53336	-0.0222458	interpolation
32	-0.360336	-0.00714307	interpolation
33	-0.360336	-0.00714307	bisection
34	-0.277057	-0.003312	interpolation
35	-0.20765	-0.00141791	interpolation
36	-0.20765	-0.00141791	bisection
37	-0.0746258	-6.79921e-05	interpolation
38	-0.0680456	-5.16296e-05	interpolation
39	-0.0475706	-1.77304e-05	interpolation
40	-0.0475706	-1.77304e-05	bisection
41	-0.0370038	-8.36723e-06	interpolation
42	-0.0278631	-3.5803e-06	interpolation
43	-0.0278631	-3.5803e-06	bisection
44	-0.0129137	-3.57763e-07	interpolation
45	-0.0112972	-2.39625e-07	interpolation
46	-0.00807886	-8.77047e-08	interpolation
47	-0.00807886	-8.77047e-08	bisection
48	-0.00578966	-3.22983e-08	interpolation
49	-0.0045019	-1.51897e-08	interpolation
50	-0.00338571	-6.46293e-09	interpolation
51	-0.00338571	-6.46293e-09	bisection
52	-0.00234337	-2.14347e-09	interpolation
53	-0.00184451	-1.04543e-09	interpolation
54	-0.00138087	-4.38688e-10	interpolation
55	-0.00138087	-4.38688e-10	bisection
56	-0.00094226	-1.39399e-10	interpolation
57	-0.000745255	-6.89733e-11	interpolation
58	-0.000556902	-2.87822e-11	interpolation
59	-0.000556902	-2.87822e-11	bisection
60	-0.000378333	-9.02456e-12	interpolation
61	-0.000299679	-4.48519e-12	interpolation
62	-0.000223811	-1.86839e-12	interpolation
63	-0.000223811	-1.86839e-12	bisection
64	-0.000151868	-5.83866e-13	interpolation
65	-0.000120334	-2.90434e-13	interpolation
66	-8.98642e-05	-1.20903e-13	interpolation
67	-8.98642e-05	-1.20903e-13	bisection
68	-6.09724e-05	-3.77476e-14	interpolation
69	-4.83249e-05	-1.88738e-14	interpolation
70	-3.59788e-05	-7.77156e-15	interpolation
71	-3.59788e-05	-7.77156e-15	bisection
72	-2.44735e-05	-2.44249e-15	interpolation
73	-1.9388e-05	-1.22125e-15	interpolation
74	-1.44236e-05	-4.44089e-16	interpolation
75	-1.44236e-05	-4.44089e-16	bisection
76	-1.00281e-05	-2.22045e-16	interpolation
77	-5.86401e-06	0	interpolation

Zero found in the interval [-1.4, 9.52548]

```
execution_time_fzero_method = toc;
```

```
E_residual_fzero = abs(Function1(rootfzero)); % Residual error |f(x_n)|
```

```
% Display results
disp(['Root found fzero: ', num2str(rootfzero)]);
```

Root found fzero: -5.864e-06

```
disp(['Number of iterations fzero: ', num2str(iterations_fzero.iterations)]);
```

Number of iterations fzero: 53

```
disp(['Execution time fzero:', num2str(execution_time_fzero_method)]);
```

Execution time fzero: 0.14598

```
disp(['E_residual_fzero: ', num2str(E_residual_fzero)]);
```

E_residual_fzero: 0

```
% Define x range for plotting
x1 = linspace(-0.5, 0.5, 100);
function1 = arrayfun(Function1, x1); % Apply Function1 to each element in x1
```

Fsolve

The `fsolve` function in MATLAB is used to solve nonlinear equation systems. It relies on algorithms like trust-region-dogleg and Levenberg-Marquardt, which are effective for many types of problems. However, to achieve reliable results, it often needs good initial guesses. Additionally, for large systems, `fsolve` can be computationally demanding (Meade, 2004).

```
%% Fsolve
% Define the function
Function1 = @(x) exp(x) - (1 + x + (x.^2)/2); % Ensure element-wise operations
using .^

% Choose an initial guess close to the actual root
x_guess = 5; % Set a very small initial guess to help `fsolve` converge correctly

% Set tolerance for fsolve
tol = 1e-6;
options = optimset('TolX', tol, 'Display', 'iter'); % Use optimset for `fsolve`

% Start timer
tic;
% Use fsolve to find the root
[rootfsolve, fval, exitflag, iterations_fsolve] = fsolve(Function1, x_guess,
options);
```

Iteration	Func-count	$ f(x) ^2$	Norm of step	First-order optimality	Trust-region radius
0	2	16877.4		1.85e+04	1
1	4	2131.14	0.912227	2.52e+03	1
2	6	258.138	0.846752	343	1
3	8	29.6896	0.753639	46.5	1
4	10	3.22882	0.637859	6.3	1

5	12	0.333037	0.512335	0.849	1
6	14	0.0328488	0.39228	0.114	1
7	16	0.00312946	0.288756	0.0152	1
8	18	0.000290648	0.206268	0.00202	1
9	20	2.65126e-05	0.144177	0.000267	1
10	22	2.38857e-06	0.0992545	3.53e-05	1
11	24	2.13374e-07	0.0676172	4.67e-06	1
12	26	1.8952e-08	0.0457377	6.16e-07	1

Equation solved.

fsolve completed because the vector of function values is near zero as measured by the value of the function tolerance, and the problem appears regular as measured by the gradient.

<stopping criteria details>

```

execution_time_Fsolve_method = toc;

% Compute the final residual error
E_residual_fsolve = abs(Function1(rootfsolve));

% Display results
disp(['Root found: ', num2str(rootfsolve)]);

```

Root found: 0.093097

```
disp(['Function value at root (should be near zero): ', num2str(fval)]);
```

Function value at root (should be near zero): 0.00013767

```
disp(['Number of iterations: ', num2str(iterations_fsolve.iterations)]);
```

Number of iterations: 12

```
disp(['Execution Time: ', num2str(execution_time_Fsolve_method), ' seconds']);
```

Execution Time: 0.051649 seconds

```
disp(['Final Residual Error: ', num2str(E_residual_fsolve)]); % Display the last residual error
```

Final Residual Error: 0.00013767

```

% Define x range for plotting
x1 = linspace(-0.5, 0.5, 100);
function1 = arrayfun(Function1, x1); % Apply Function1 to each element in x1

```

Vpasolve

The `vpasolve` function in MATLAB uses variable-precision arithmetic to solve equations symbolically. It is especially useful for problems that need high precision or exact symbolic solutions. However, as precision increases, the computational cost also rises, making `vpasolve` slower than numerical solvers (The MathWorks, Inc, n.d.).

```

%% vpsolve
% Define the symbolic variable
syms x;

% Define the function
Function1 = exp(x) - (1 + x + (x^2)/2); % No need for element-wise operators in
symbolic form

% Set numerical tolerance
tol = 1e-6;
tic
% Solve the equation using vpsolve with a search interval
root_vpsolve = vpsolve(Function1 == 0, x, [5]); % Restrict search range for
stability
execution_time_vpsolve_method = toc;

% Convert symbolic root to numeric
root_vpsolve_numeric = double(root_vpsolve);

% Compute the final residual error
E_residual_vpsolve = abs(double(subs(Function1, root_vpsolve_numeric)));

% Display the result
disp(['Root found: ', char(root_vpsolve)]);

```

```
Root found: 0.0000000000000061615614051520440059753194542351
```

```
disp(['Execution Time: ', num2str(execution_time_vpsolve_method), ' seconds']);
```

```
Execution Time: 0.13761 seconds
```

```
disp(['Final Residual Error: ', num2str(E_residual_vpsolve)]); % Display final
residual error
```

```
Final Residual Error: 3.8987e-44
```

```

% Define x range for plotting
x1 = linspace(-0.5, 0.5, 100);
function1 = arrayfun(@(x) double(subs(Function1, x)), x1); % Convert symbolic
function to numerical for plotting

```

lsqnonlin function

The `lsqnonlin` function is used to solve nonlinear least-squares problems by minimizing the sum of squares of nonlinear functions. It is a powerful tool but can converge to local minima. The quality of the solution depends on the initial guess and the behavior of the objective function (The MathWorks, Inc. n.d.).

```

%% lsqnonlin
% Define the function
Function1 = @(x) exp(x) - (1 + x + (x.^2)/2); % Ensure element-wise operations
using .^

% Choose an initial guess
x_guess = 5; % Start slightly away from zero

% Set solver options for better convergence control
tol = 1e-6;
options = optimset('TolX', tol, 'TolFun', tol, 'Display', 'iter'); % Set tolerances

tic;
% Use lsqnonlin to minimize the function squared (equivalent to finding the root)
[root_isqnonlin, resnorm, residual, exitflag, iterations_isqnonlin] =
lsqnonlin(Function1, x_guess, [], [], options);

```

Iteration	Func-count	Resnorm	Norm of step	First-order optimality
0	2	16877.4		1.85e+04
1	4	2131.14	0.912227	2.52e+03
2	6	258.138	0.846752	343
3	8	29.6896	0.753639	46.5
4	10	3.22882	0.637859	6.3
5	12	0.333037	0.512335	0.849
6	14	0.0328488	0.39228	0.114
7	16	0.00312946	0.288756	0.0152
8	18	0.000290648	0.206268	0.00202
9	20	2.65126e-05	0.144177	0.000267
10	22	2.38857e-06	0.0992545	3.53e-05
11	24	2.13374e-07	0.0676172	4.67e-06
12	26	1.8952e-08	0.0457377	6.16e-07

Local minimum found.

Optimization completed because the size of the gradient is less than the value of the optimality tolerance.

<stopping criteria details>

```

execution_time_Isqnonlin_method = toc;

% Compute the final residual error
E_residual_lsqnonlin = abs(Function1(root_isqnonlin));

% Display results
disp(['Root found: ', num2str(root_isqnonlin)]);

```

Root found: 0.093097

```
disp(['Number of iterations: ', num2str(iterations_isqnonlin.iterations)]);
```

Number of iterations: 12

```
disp(['Exit flag: ', num2str(exitflag)]); % Check if solver converged successfully
```

Exit flag: 1

```
disp(['Execution Time: ', num2str(execution_time_lsqnonlin_method), ' seconds']);
```

Execution Time: 0.024397 seconds

```
disp(['Final Residual Error: ', num2str(E_residual_lsqnonlin)]); % Display the last residual error
```

Final Residual Error: 0.00013767

```
% Define x range for plotting
x1 = linspace(-0.5, 0.5, 100);
function1 = arrayfun(Function1, x1); % Apply Function1 to each element in x1
```

fminunc function

The `fminunc` function is used to find the minimum of an unconstrained multivariable function. By minimizing the square of the function, it can also be applied to root-finding problems. However, similar to `lsqnonlin`, it can get stuck in local minima, and its effectiveness depends on the choice of the initial guess (The MathWorks, Inc. n.d.).

```
%% fminunc (Optimization-Based Root Finding)
% Define the function
Function1 = @(x) exp(x) - (1 + x + (x.^2)/2); % Ensure element-wise operations
using .^

% Define an objective function to minimize (squared function value)
ObjectiveFunction = @(x) Function1(x).^2; % Squaring ensures minimization

% Choose an initial guess
x_guess = 0.1; % A reasonable starting point

% Set optimization options
tol = 1e-6;
options = optimoptions('fminunc', 'TolX', tol, 'TolFun', tol, 'Display', 'iter');

% Start timer
tic;
% Use fminunc to minimize the function squared (effectively finding the root)
[root_fminunc, fval, exitflag, iterations_fminunc] = fminunc(ObjectiveFunction,
x_guess, options);
```

Iteration	Func-count	f(x)	Step-size	First-order optimality
0	2	2.9213e-08		1.77e-06
1	12	1.25435e-08	7381	8.72e-07

Local minimum found.

Optimization completed because the size of the gradient is less than the value of the optimality tolerance.

<stopping criteria details>

```
execution_time_fminunc_method = toc;  
  
% Compute the final residual error  
E_residual_fminunc = abs(Function1(root_fminunc));  
  
% Display results  
disp(['Root found: ', num2str(root_fminunc)]);
```

Root found: 0.086953

```
disp(['Function value at root (should be near zero): ', num2str(fval)]);
```

Function value at root (should be near zero): 1.2544e-08

```
disp(['Number of iterations: ', num2str(iterations_fminunc.iterations)]);
```

Number of iterations: 1

```
disp(['Exit flag: ', num2str(exitflag)]); % Check if solver converged successfully
```

Exit flag: 1

```
disp(['Execution Time: ', num2str(execution_time_fminunc_method), ' seconds']);
```

Execution Time: 0.016296 seconds

```
disp(['Final Residual Error: ', num2str(E_residual_fminunc)]); % Display the last residual error
```

Final Residual Error: 0.000112

```
% Define x range for plotting  
x1 = linspace(-0.5, 0.5, 100);  
function1 = arrayfun(Function1, x1); % Apply Function1 to each element in x1
```

Bisection Method

The Bisection Method is a simple bracketing technique that repeatedly reduces an interval containing a root. It guarantees convergence for continuous functions where a sign change occurs within the interval. However, its convergence rate is linear, making it slower than methods like Newton's. Despite this, the method is reliable and does not require calculating derivatives, making it a robust choice for root-finding (Volkov, 2017).

```
%% Bi-section Method  
Function1 = @(x) exp(x) - (1 + x + (x.^2)/2); % Ensure element-wise operations using .^
```

```

tol = 1e-6;
imax = 50;
a = 0;
b = 4;

% Start timer
tic;
[mid_point, counter_bisection] = bisect2(Function1, a, b, imax, tol);

```

```

ITERATION 0
Lower guess = 0.000000
Upper guess = 4.000000
Mid-point = 2.000000

```

```

ITERATION 1
Lower guess = 0.000000
Upper guess = 2.000000
Mid-point = 1.000000

```

```

ITERATION 2
Lower guess = 0.000000
Upper guess = 1.000000
Mid-point = 0.500000

```

```

ITERATION 3
Lower guess = 0.000000
Upper guess = 0.500000
Mid-point = 0.250000

```

```

ITERATION 4
Lower guess = 0.000000
Upper guess = 0.250000
Mid-point = 0.125000

```

```

ITERATION 5
Lower guess = 0.000000
Upper guess = 0.125000
Mid-point = 0.062500

```

```

ITERATION 6
Lower guess = 0.000000
Upper guess = 0.062500
Mid-point = 0.031250

```

```

ITERATION 7
Lower guess = 0.000000
Upper guess = 0.031250
Mid-point = 0.015625

```

```

ITERATION 8
Lower guess = 0.000000
Upper guess = 0.015625
Mid-point = 0.007812

```

```

ITERATION 9
Lower guess = 0.000000
Upper guess = 0.007812
Mid-point = 0.003906

```

```

ITERATION 10
Lower guess = 0.000000
Upper guess = 0.003906

```

```

Mid-point = 0.001953

ITERATION 11
Lower guess = 0.000000
Upper guess = 0.001953
Mid-point = 0.000977

ITERATION 12
Lower guess = 0.000000
Upper guess = 0.000977
Mid-point = 0.000488

ITERATION 13
Lower guess = 0.000000
Upper guess = 0.000488
Mid-point = 0.000244

ITERATION 14
Lower guess = 0.000000
Upper guess = 0.000244
Mid-point = 0.000122

ITERATION 15
Lower guess = 0.000000
Upper guess = 0.000122
Mid-point = 0.000061

ITERATION 16
Lower guess = 0.000000
Upper guess = 0.000061
Mid-point = 0.000031

ITERATION 17
Lower guess = 0.000000
Upper guess = 0.000031
Mid-point = 0.000015

ITERATION 18
Lower guess = 0.000000
Upper guess = 0.000015
Mid-point = 0.000008

ITERATION 19
Lower guess = 0.000008
Upper guess = 0.000015
Mid-point = 0.000011

ITERATION 20
Lower guess = 0.000008
Upper guess = 0.000011
Mid-point = 0.000010

ITERATION 21
Lower guess = 0.000008
Upper guess = 0.000010
Mid-point = 0.000009

```

```

execution_time_bi_method = toc;

% Compute the final residual error
E_residual_bisection = abs(Function1(mid_point));

```

```
% Display results
disp(['Root found using Bi-section: ', num2str(mid_point)]);
```

Root found using Bi-section: 8.5831e-06

```
disp(['Number of iterations: ', num2str(counter_bisection)]);
```

Number of iterations: 22

```
disp(['Execution Time: ', num2str(execution_time_bi_method), ' seconds']);
```

Execution Time: 0.093504 seconds

```
disp(['Final Residual Error: ', num2str(E_residual_bisection)]); % Display the last residual error
```

Final Residual Error: 0

Comparison Table of Functions

Comparison table is to show the difference between functions based on root found, iterations, execution time and E residuals.

```
%% Comparison Table Plot
```

```
methods = {'Newton', 'fzero', 'fsolve', 'vpasolve', 'lsqnonlin', 'fminunc',
'Bi-section'}; % Added Bisection
roots = [x_next, rootfzero, rootfsolve, double(root_vpasolve), root_isqnonlin,
root_fminunc, mid_point]; % Added mid_point for Bisection
iterations = [counter, iterations_fzero.iterations, iterations_fsolve.iterations,
NaN, iterations_isqnonlin.iterations, iterations_fminunc.iterations,
counter_bisection]; % Added counter_bisection
execution_times = [execution_time, execution_time_fzero_method,
execution_time_Fsolve_method, ...
execution_time_vpasolve_method, execution_time_Isqnonlin_method,
execution_time_fminunc_method, execution_time_bi_method];
```

```
% Ensure residual error array has the same length as methods
```

```
E_Residual = [E_residual_last, E_residual_fzero, E_residual_fsolve,
E_residual_vpasolve, ...
E_residual_lsqnonlin, E_residual_fminunc, E_residual_bisection]; %
Added E_residual_bisection
```

```
% Create a table
```

```
T = table(methods', roots', iterations', execution_times', E_Residual', ...
'VariableNames', {'Method', 'Root_Found', 'Iterations', 'Execution_Time_s',
'E_Residual'});
```

```
% Display the table in MATLAB
```

```
disp(T);
```

Method	Root_Found	Iterations	Execution_Time_s	E_Residual
{'Newton' }	8.62187885310581e-07	37	0.1046419	0
{'fzero' }	-5.8640127856478e-06	53	0.1459785	0
{'fsolve' }	0.0930965350863694	12	0.0516493	0.000137666436370143
{'vpasolve' }	6.16156140515204e-15	NaN	0.1376095	3.89871144039669e-44
{'lsqnonlin' }	0.0930965350863694	12	0.0243972	0.000137666436369921
{'fminunc' }	0.086953308221592	1	0.0162959	0.000111997874436165
{'Bisection' }	8.58306884765625e-06	22	0.0935035	0

Score table

score table is done to determine the best function.

```
% Normalize each metric (Higher score means better)
normalize = @(x) (max(x) - x) / (max(x) - min(x));

% Handle NaN values in iterations (Replace NaN with max value)
iterations(isnan(iterations)) = max(iterations(~isnan(iterations)));

% Compute Normalized Scores
normalized_residual = normalize(E_Residual);
normalized_iterations = normalize(iterations);
normalized_execution_times = normalize(execution_times);

% Define Weights
w_residual = 0.5;
w_iterations = 0.1;
w_execution_time = 0.2;

% Compute Final Scores
final_scores = (w_residual * normalized_residual) + ...
               (w_iterations * normalized_iterations) + ...
               (w_execution_time * normalized_execution_times);

% Add Scores to Table
T.Final_Score = final_scores';

% Display Updated Table
disp(T);
```

Method	Root_Found	Iterations	Execution_Time_s	E_Residual	Final_Score
{'Newton' }	8.62187885310581e-07	37	0.1046419	0	0.6263948582
{'fzero' }	-5.8640127856478e-06	53	0.1459785	0	
{'fsolve' }	0.0930965350863694	12	0.0516493	0.000137666436370143	0.297061704
{'vpasolve' }	6.16156140515204e-15	NaN	0.1376095	3.89871144039669e-44	0.5193603459
{'lsqnonlin' }	0.0930965350863694	12	0.0243972	0.000137666436369921	0.3601050891
{'fminunc' }	0.086953308221592	1	0.0162959	0.000111997874436165	0.4932273784
{'Bisection' }	8.58306884765625e-06	22	0.0935035	0	0.681007923

```
% Identify the Best Method
```

```
[~, best_index] = max(final_scores);
disp(['Best Method: ', methods{best_index}, ' with a final score of ',
num2str(final_scores(best_index))]);
```

Best Method: Bisection with a final score of 0.68101

Conclusion on chosen method

ewton's Method and the Bisection Method have already been applied to find N5NT. Newton's Method converged in 37 iterations with an execution time of 0.1046s, while the Bisection Method required 22 iterations and took 0.0935s, both achieving zero residual error. Since these methods have already been used, the next best option for the task is the fzero function. fzero found a root of -5.86×10^{-6} in 53 iterations, with an execution time of 0.146s and zero residual error. It is a hybrid method that combines bisection, secant, and inverse quadratic interpolation, making it a robust choice when other methods have already been explored. Given that both Bisection and Newton's Method have been applied, fzero is the logical next choice to refine the root-finding process. It must be stated that execution time changes on factors like

- Amount of time script has been run
- Load on the laptop

Applying Fzero to find N5nt

```
clc;
clear;

% Given constants
c = 8;
h = 18;
o = 0;
airs = 12.5;
temp = (100:25:3000)'; % Temperature range
press = (20:5:120)'; % Pressure range
feq = (0.8:0.02:1.3)'; % Fuel equivalence ratio

% Initialize results matrix
n5nt = zeros(49, 21, 26);
error_matrix = zeros(49, 21, 26);

% Compute using fzero method
for i = 1:49 % Temperature index
    for j = 1:21 % Pressure index
        for b = 1:26 % Fuel equivalence ratio index
            [n5nt(i, j, b), error_matrix(i, j, b)] = mydissfun_fzero(c, h, o,
temp(i), feq(b), airs, press(j)); % Using fzero function
        end
    end
end
```

```
% Display the first few values
disp('Computed n5nt values using fzero Method:');
```

```
fprintf('%0.5f\n', n5nt(:,1,1)); % Display n5nt for first equivalence ratio
```

N5nt is a 49by21by26

References

Liu, Y. (n.d.). *First Semester in Numerical Analysis with Python*. Retrieved from <https://yaningliucudenver.github.io/Numerical-Analysis-I/bookchapter2-3.html>

MathWorks. (n.d.). *fzero*. Retrieved from <https://www.mathworks.com/help/matlab/ref/fzero.html>

Meade, D. B. (2004). *Newton's Method and fsolve*. University of South Carolina. Retrieved from <https://people.math.sc.edu/meade/141L-F04/labs/NewtonMethod.pdf>

The MathWorks, Inc. (n.d.). *vpasolve*. Retrieved from <https://www.mathworks.com/help/symbolic/vpasolve.html>

The MathWorks, Inc. (n.d.). *lsqnonlin*. Retrieved from <https://www.mathworks.com/help/optim/ug/lsqnonlin.html>

The MathWorks, Inc. (n.d.). *fminunc*. Retrieved from <https://www.mathworks.com/help/optim/ug/fminunc.html>

Volkov, A. (2017). *Numerical Analysis I*. University of Alabama. Retrieved from <https://volkov.eng.ua.edu/ME349/2017-Fall-ME349-03-NumAnalysis1.pdf>

Function are below and in a supporting file

For Newton

```
function [n5ntvalue, counter, time_elapsed] = mydissfuniii_newton(carbon, hydrogen,
    oxygen, temperature, fuelequiv, molar_air_stoich, pressure)
    % Newton's Method for n5nt

    tic; % Start timing
    counter = 0; % Iteration counter
    tol = 1e-12; % Lower tolerance for better accuracy
    imax = 500; % Higher max iterations to ensure proper convergence

    %
    n5ntvalue = 0.1;
    abs_diff = inf;

    %
    gibbseng = [
        0 -393.151 -238.921 -113.805;
        100 -393.683 -236.584 -120.239;
        200 -394.085 -232.766 -128.526;
        298.15 -394.389 -228.582 -137.163;
        300 -394.394 -228.5 -137.328;
        400 -394.675 -223.901 -146.338;
        500 -394.939 -219.051 -155.414;
        600 -395.182 -214.007 -164.486;
        700 -395.398 -208.812 -173.518;
        800 -395.586 -203.496 -182.497;
        900 -395.748 -198.083 -191.416;
        1000 -395.886 -192.59 -200.275;
        1100 -396.001 -187.033 -209.075;
        1200 -396.098 -181.425 -217.819;
        1300 -396.177 -175.774 -226.509;
```

```

1400 -396.24 -170.089 -235.149;
1500 -396.288 -164.376 -243.74;
1600 -396.323 -158.639 -252.284;
1700 -396.344 -152.883 -260.784;
1800 -396.353 -147.111 -269.242;
1900 -396.349 -141.325 -277.658;
2000 -396.333 -135.528 -286.034;
2100 -396.304 -129.721 -294.372;
2200 -396.262 -123.905 -302.672;
2300 -396.209 -118.082 -310.936;
2400 -396.142 -112.252 -319.165;
2500 -396.062 -106.416 -327.358;
2600 -395.969 -100.575 -335.517;
2700 -395.862 -94.729 -343.643;
2800 -395.742 -88.878 -351.736;
2900 -395.609 -83.023 -359.797;
3000 -395.461 -77.163 -367.826
];

%
gco2 = interp1(gibbseng(:,1), gibbseng(:,2), temperature) * 1e3;
gh2o = interp1(gibbseng(:,1), gibbseng(:,3), temperature) * 1e3;
gco = interp1(gibbseng(:,1), gibbseng(:,4), temperature) * 1e3;

% Compute equilibrium constants
kco2 = exp(-(gco2 - gco) * 1e3 / (8314.46 * temperature));
kh2o = exp(-gh2o * 1e3 / (8314.46 * temperature));

%
airs = molar_air_stoich; % Ensure the same variable is used

%
while abs_diff > tol && counter < imax
    counter = counter + 1;

    % Compute function value f(n5nt)
    a = sqrt(n5ntvalue * pressure);
    n3 = (carbon * fuelequiv) / (kco2 * a + 1);
    n4 = ((hydrogen / 2) * fuelequiv) / (kh2o * a + 1);
    n1 = carbon * fuelequiv - n3;
    n2 = (hydrogen / 2) * fuelequiv - n4;
    n5 = (airs * 2 + oxygen * fuelequiv - 2 * n1 - n2 - n3) / 2;
    n6 = airs * 3.76;
    ntsum = n1 + n2 + n3 + n4 + n5 + n6;
    f_val = n5ntvalue - n5 / ntsum;

    % Compute derivative df/d(n5nt) numerically
    h_step = 1e-8;
    a_h = sqrt((n5ntvalue + h_step) * pressure);
    n3_h = (carbon * fuelequiv) / (kco2 * a_h + 1);

```

```

n4_h = ((hydrogen / 2) * fuelequiv) / (kh2o * a_h + 1);
n1_h = carbon * fuelequiv - n3_h;
n2_h = (hydrogen / 2) * fuelequiv - n4_h;
n5_h = (airs * 2 + oxygen * fuelequiv - 2 * n1_h - n2_h - n3_h) / 2;
ntsum_h = n1_h + n2_h + n3_h + n4_h + n5_h + n6;
f_val_h = (n5ntvalue + h_step) - n5_h / ntsum_h;
df_val = (f_val_h - f_val) / h_step;

%
newton_step = f_val / df_val;
n5nt_next = n5ntvalue - newton_step;

% step
if abs(n5nt_next - n5ntvalue) > 1e-3
    n5nt_next = n5ntvalue - sign(f_val) * 1e-3;
end

% Check tolerance
abs_diff = abs(n5nt_next - n5ntvalue);

% Update value
n5ntvalue = n5nt_next;
end

% Stop timing
time_elapsed = toc;
end

```

For Bi-Section

```

function [n5nt, error] = mydissfun_bisection(carbon, hydrogen, oxygen, temperature,
fuelequiv, molar_air_stoich, pressure)
% Initialize bounds for bisection method
lower_bound = 0;
upper_bound = 0.1;
tolerance = 1e-9; % Convergence tolerance
max_iter = 1000; % Maximum number of iterations
iter = 0;

% Gibbs free energy data(same data from og diss)
gibbseng = [
    0, -393.151, -238.921, -113.805;
    100, -393.683, -236.584, -120.239;
    200, -394.085, -232.766, -128.526;
    298.15, -394.389, -228.582, -137.163;
    300, -394.394, -228.5, -137.328;
    400, -394.675, -223.901, -146.338;
    500, -394.939, -219.051, -155.414;
    600, -395.182, -214.007, -164.486;
    700, -395.398, -208.812, -173.518;

```

```

800, -395.586, -203.496, -182.497;
900, -395.748, -198.083, -191.416;
1000, -395.886, -192.59, -200.275;
1100, -396.001, -187.033, -209.075;
1200, -396.098, -181.425, -217.819;
1300, -396.177, -175.774, -226.509;
1400, -396.24, -170.089, -235.149;
1500, -396.288, -164.376, -243.74;
1600, -396.323, -158.639, -252.284;
1700, -396.344, -152.883, -260.784;
1800, -396.353, -147.111, -269.242;
1900, -396.349, -141.325, -277.658;
2000, -396.333, -135.528, -286.034;
2100, -396.304, -129.721, -294.372;
2200, -396.262, -123.905, -302.672;
2300, -396.209, -118.082, -310.936;
2400, -396.142, -112.252, -319.165;
2500, -396.062, -106.416, -327.358;
2600, -395.969, -100.575, -335.517;
2700, -395.862, -94.729, -343.643;
2800, -395.742, -88.878, -351.736;
2900, -395.609, -83.023, -359.797;
3000, -395.461, -77.163, -367.826;
];

% Interpolating Gibbs Free Energy ( from og diss)
gco2 = interp1(gibbseng(:,1), gibbseng(:,2), temperature) * 1e3;
gh2o = interp1(gibbseng(:,1), gibbseng(:,3), temperature) * 1e3;
gco = interp1(gibbseng(:,1), gibbseng(:,4), temperature) * 1e3;

% Equilibrium constants same a og function :)
kco2 = exp((-gco2 - gco) * 1e3 / (8314.46 * temperature));
kh2o = exp((-gh2o) * 1e3 / (8314.46 * temperature));

% Bisection Method to Find n5nt
while (upper_bound - lower_bound) > tolerance && iter < max_iter
    iter = iter + 1;
    n5ntvalue = (upper_bound + lower_bound) / 2;
    a = sqrt(n5ntvalue * pressure);

    n3 = (carbon * fuelequiv) / (kco2 * a + 1);
    n4 = ((hydrogen / 2) * fuelequiv) / (kh2o * a + 1);
    n1 = carbon * fuelequiv - n3;
    n2 = (hydrogen / 2) * fuelequiv - n4;
    n5 = (molar_air_stoich * 2 + oxygen * fuelequiv - 2 * n1 - n2 - n3) / 2;
    n6 = molar_air_stoich * 3.76;
    ntsum = n1 + n2 + n3 + n4 + n5 + n6;

    error = n5ntvalue - n5 / ntsum;

```

```

    if error > 0
        upper_bound = n5ntvalue;
    else
        lower_bound = n5ntvalue;
    end
end

n5nt = (upper_bound + lower_bound) / 2; % Final result
end

```

For Fzero

```

function [n5nt, error] = mydissfun_fzero(carbon, hydrogen, oxygen, temperature,
fuelequiv, molar_air_stoich, pressure)
    % Gibbs free energy data
    gibbseng = [
        0, -393.151, -238.921, -113.805;
        100, -393.683, -236.584, -120.239;
        200, -394.085, -232.766, -128.526;
        298.15, -394.389, -228.582, -137.163;
        300, -394.394, -228.5, -137.328;
        400, -394.675, -223.901, -146.338;
        500, -394.939, -219.051, -155.414;
        600, -395.182, -214.007, -164.486;
        700, -395.398, -208.812, -173.518;
        800, -395.586, -203.496, -182.497;
        900, -395.748, -198.083, -191.416;
        1000, -395.886, -192.59, -200.275;
        1100, -396.001, -187.033, -209.075;
        1200, -396.098, -181.425, -217.819;
        1300, -396.177, -175.774, -226.509;
        1400, -396.24, -170.089, -235.149;
        1500, -396.288, -164.376, -243.74;
        1600, -396.323, -158.639, -252.284;
        1700, -396.344, -152.883, -260.784;
        1800, -396.353, -147.111, -269.242;
        1900, -396.349, -141.325, -277.658;
        2000, -396.333, -135.528, -286.034;
        2100, -396.304, -129.721, -294.372;
        2200, -396.262, -123.905, -302.672;
        2300, -396.209, -118.082, -310.936;
        2400, -396.142, -112.252, -319.165;
        2500, -396.062, -106.416, -327.358;
        2600, -395.969, -100.575, -335.517;
        2700, -395.862, -94.729, -343.643;
        2800, -395.742, -88.878, -351.736;
        2900, -395.609, -83.023, -359.797;
        3000, -395.461, -77.163, -367.826;
    ];

```

```

% Interpolating Gibbs Free Energy
gco2 = interp1(gibbseng(:,1), gibbseng(:,2), temperature) * 1e3;
gh2o = interp1(gibbseng(:,1), gibbseng(:,3), temperature) * 1e3;
gco = interp1(gibbseng(:,1), gibbseng(:,4), temperature) * 1e3;

% Equilibrium constants
kco2 = exp(-(gco2 - gco)) * 1e3 / (8314.46 * temperature));
kh2o = exp(-(gh2o) * 1e3 / (8314.46 * temperature));

% Define the function to be solved using `fzero`
function eq_error = equilibrium_eq(n5ntvalue)
    a = sqrt(n5ntvalue * pressure);
    n3 = (carbon * fuelequiv) / (kco2 * a + 1);
    n4 = ((hydrogen / 2) * fuelequiv) / (kh2o * a + 1);
    n1 = carbon * fuelequiv - n3;
    n2 = (hydrogen / 2) * fuelequiv - n4;
    n5 = (molar_air_stoich * 2 + oxygen * fuelequiv - 2 * n1 - n2 - n3) / 2;
    n6 = molar_air_stoich * 3.76;
    ntsum = n1 + n2 + n3 + n4 + n5 + n6;
    eq_error = n5ntvalue - n5 / ntsum;
end

% Solve using `fzero`
n5nt = fzero(@equilibrium_eq, [0, 0.1]); % Initial guess range

% Compute final error
error = equilibrium_eq(n5nt);
end

```