Real-Time AI-Assisted Error Feedback System and Copilot for MATLAB Programming in Core Engineering Courses: A Comparative Analysis

Dr. Mahadevaswamy[1], Dr. Sathyanarayana N[2], Dr. B P Pradeep Kumar[3], Dr. Jagadeesh B[4], Dr. Swapnil S Ninawe[5]
[1,4]Associate Professor, Dept. of ECE, Vidyavardhaka College of Engineering, Mysuru,
[2]Associate Professor, Dept. of ISE, Vemana Institute of Technology, Bengaluru,
[3]Professor, Dept. of CSD, Atria Institute of Technology, Bengaluru,
[5]Assistant Professor, Dept. of ECE, Dayananda Sagar College of Engineering, Bengaluru,
[1]mahadevaswamy@vvce.ac.in, [2]sathya40y@gmail.com , [3]pradi14cta@gmail.com ,
[4]jagadeesh.b@vvce.ac.in, [5]swapnil.ninawe@gmail.com

Abstract— **New MATLAB learners often get stuck because they cannot read or fix early errors. We tested a real-time system that catches run-time exceptions, pulls the message and nearby code, and gives a short explanation plus a small corrected example inside the editor. We compared this with MATLAB's built-in copilot. In a class study with 60 students, both groups did the same lessons and tasks; only the feedback differed. The treatment group, which saw the real-time explanations, scored higher on the post-test (16.8 vs. 13.5), showed a much larger learning gain (0.58 vs. 0.29), and fixed errors faster. Across about 300 total errors, total repair time dropped from 1,500 s to 450 s, retries fell from 2 to 1 per error, and successful fixes rose from 80% to 96%. Beginners improved the most, and repeat errors fell by roughly 47%. Students said the messages were clear and useful. Then, turning MATLAB error messages into short, direct guidance improved accuracy, speed, and retention without leaving the MATLAB workspace.**

Keywords— **Debugging; error feedback; MATLAB; programming education; real-time guidance; learning gain**

JEET Category— **Track: Emerging Technologies and Future Skills. SubTrack: Pedagogy for the Modern Classroom-Strategies for Engaging Students through AI.**

I. INTRODUCTION

In engineering and scientific programs, MATLAB has become a core tool for building algorithmic skills, modeling dynamic behavior, and tackling numerical problems. Yet many students—especially those encountering the language for the first time—struggle when their code breaks. Their limited grasp of MATLAB's syntax, logic, and error phrasing often makes debugging slow and discouraging. These hurdles can stall progress, increase frustration, and reduce engagement, particularly in large classes or self-guided settings where timely instructor help is difficult to obtain.

MATLAB's error messages are typically fixed, concise system outputs that rarely offer context beyond identifying the issue. For beginners who do not yet understand ideas such as mismatched matrix sizes or incorrect indexing, simply deciphering what the message means can be a major obstacle. Although MATLAB provides standard debugging features, the environment does not supply adaptive or learner-centered explanations that meet students at their level. As a result, novices often fall back on guesswork, repeated trial and error, or searching online for snippets of code—approaches that may fix immediate issues but contribute little to long-term understanding.

Recent progress in natural-language models has opened possibilities for improving programming instruction through interactive, conversational support. Large language models (LLMs) like ChatGPT can interpret both text and code, recognize the situation behind an error, and offer clear explanations or examples within seconds. When paired with an execution environment, such models can turn a confusing system message into an opportunity for guided learning. Despite this potential, similar integrations have rarely been explored for MATLAB, even though it remains a central language in many university courses and research settings.

This study introduces an automated real-time feedback system designed to bring this capability into MATLAB. The system detects exceptions during execution, constructs a natural-language query, and obtains an immediate, tailored response through an LLM-based interface. In contrast to standard error messages, the feedback includes plain-language explanations, stepwise advice, and corrected code grounded in the student's own script. The goal is not only to help students resolve errors but also to reinforce underlying concepts, making the tool useful for both troubleshooting and deeper learning.

The work presented here contributes to programming education by (i) building a reliable mechanism for capturing and forwarding MATLAB errors, (ii) linking this mechanism to an LLM-driven conversational layer for feedback delivery, and (iii) assessing the instructional value of the system through classroom-based evaluation. The study further analyzes which errors students encounter most often, what kinds of explanations they find helpful, and how immediate feedback shapes their confidence and debugging performance. Ultimately, this research responds to the limitations of static system messages and lays groundwork for more adaptive tutoring systems that respond to students' coding patterns and levels of understanding. The results aim to inform the next generation of learning tools, in which artificial intelligence acts not simply as a generator of code but as an on-demand guide that supports individual learning paths. This work is first of its kind as it includes the creation of AI based real time MATLAB error analysis and correction support. The major steps involved in automating the error feedback system is to first capture the error from the MATLAB command window, create a report on the error analysis. Create the solutions using chatGPT. Compare this performance with that of the copilot from MATLAB software. This work examines whether turning the MATLAB exceptions into concise, context-aware guidance improves learning and debugging. We address five questions:

RQ1. Does AI based real-time error guidance improve performance relative to default MATLAB's copilot error explanation and correction?

RQ2. Does the system increase debugging efficiency (time per error, retries per error) and fix success rate?

RQ3. Does the system reduce recurrence of the same error class across tasks?

RQ4. Are benefits moderated by prior skill level (beginner, intermediate, advanced)?

RQ5. Which feedback type (clarification, minimal fix snippet, conceptual explanation, external link) yields the fastest resolution and best retention?

II. RELATED WORKS

Novice programmers struggle first and most with error comprehension. Classic and careful studies showed that the form and phrasing of error messages materially affect time-to-fix, comprehension, and frustration, and that improving messages helps up to a point but cannot replace targeted guidance (Marceau, Fisler, & Krishnamurthi, 2011; Becker, 2016). Subsequent controlled classroom and field studies examined compiler/IDE message redesigns and found measurable but bounded gains when messages were clarified, localized, or paired with links to examples; stronger outcomes appeared when guidance also explained why a fix worked, not only what to change (Becker, Glanville, & Goslin, 2018). These results motivate systems that go beyond surface edits to attach short rationale in the language of the host environment (e.g., MATLAB indexing rules, array shape semantics). (Becker, 2016; Becker et al., 2018; Marceau et al., 2011). A systematic review across tools, strategies, and constraints noted that effective systems combine test-based checks, program analyses, and pattern mining to deliver timely, specific hints; they work best when feedback targets a known misconception class and when students can iterate quickly (Keuning, Jeuring, & Heeren, 2018). This aligns with designs that capture exceptions at runtime, compress context, and return minimal, actionable suggestions that match the learner's error class. Recent computing-education work has begun to compare feedback from large language models with rule-based tasks. Across several 2024 conference papers, large models produced higher-coverage comments and more natural phrasing but varied in correctness and specificity; combining them with scaffolds (e.g., "try first, then reveal hint") reduced over-reliance and improved revision quality (Azaiz, Kiesler, & Strickroth, 2024; Lohr et al., 2024; Ruan et al., 2024). These studies recommend constraining prompts, limiting verbosity, and embedding verification (tests, examples) alongside model feedback-practices mirrored in systems that turn raw exceptions into short, environment-specific coaching. A field evaluation in higher education reported that large-model tools can generate useful formative feedback in programming courses, but utility depends on error type, prompt design, and having the teacher in the loop; accuracy and pedagogical fit improved when suggestions were checked against unit tests and course rubrics (Estévez-Ayres, Pazos, Rodríguez, & Muñoz-Merino, 2024). UKICER work focusing specifically on model-augmented error messages cautioned that "enhanced" messages are not a universal cure; in practice, some messages remain too generic or subtly wrong unless grounded in the actual code state and test outcomes (Santos & Becker, 2024). Together, these studies support designs that pair exception capture with tight prompts, ask for minimal fixes, and attach a why-it-failed note tied to the host language. Work on novice compilation traces and failure modes shows repeated cycles on the same error classes and long stalls when messages are opaque (Jadud, 2006; Watson & Li, 2014). Data-driven hinting within intelligent tutors for programming further demonstrates that short, targeted hints, sequenced by difficulty and grounded in prior student data, yield better learning than full solutions, especially when hints call out the underlying concept (Rivers & Koedinger, 2017). These lessons translate directly to MATLAB environments: short fixes, rationale, and immediate re-run loops reduce retries and repeated errors (Jadud, 2006; Rivers & Koedinger, 2017; Watson & Li, 2014). Broader studies on formative feedback quality in higher education-outside programming-are useful for setting expectations(Meyer et al., 2024; Steiss, Nückles, & Renkewitz, 2024). Learning-analytics reviews add that fast feedback cycles and actionable signals, rather than dashboards alone, drive improvement (Pan, Biegley, Taylor, & Zheng, 2024; Paulsen et al., 2024). These findings back the choice to keep responses brief, specific, and tied to immediate next steps in the same tool, with logs that surface class-wide pain points for the teacher. The timing and framing really matter. Designs that model "capture → compress → query → parse → display," with prompts tuned to the host language and unit tests attached, reflect the emerging consensus on how to gain speed and understanding without inflating cognitive load.

III. Methodology

3.1 Background and Theoretical Background

The conceptual basis for "real-time error feedback systems" is that errors become teachable moments, hence they should be transformed into an immediate feedback form. The methodology supports previous research on developing compiler messages and the development of automated feedback systems with an emphasis on brevity, specificity, and rationale for concept development. The design is based upon a five-part cycle of events: output of error (error detection), extraction of relevant information, LLM processing and display of feedback, and input of additional information.

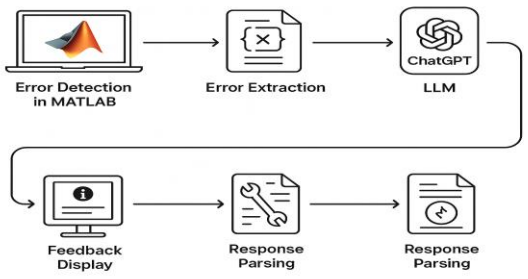


Fig. 1. Proposed Methodology.

The process starts with MATLAB's error detection. When a student runs a script and something goes wrong-whether due to syntax, logic, or a runtime issue-MATLAB produces its usual error report, noting the error type, line number, and a short description. For newcomers, these messages can be hard to interpret, so capturing them accurately is the first step in the system. The next phase, Error Extraction, isolates the relevant message from the rest of the console output. Using MATLAB's MException tools or by parsing the Command Window text, the system pulls out the specific error line, cleans it, and formats it into a clear string. This ensures that only the necessary information moves forward. After extraction, the error text is sent to a large language model via an API call. This stage involves forming a prompt that asks the model to explain the issue and offer a correction. The connection is typically handled through a MATLAB-Python interface or a small Python script that communicates with the LLM service. The model returns a reply that outlines what caused the error and how to fix it.

Next comes Response Parsing. Because the model's explanation may arrive as a long paragraph, the system breaks it into readable parts. If the response includes steps, examples, or code fragments, these are separated so students can follow the guidance more easily. The Feedback Display component shows the processed message inside MATLAB-either through a pop-up, a message panel, or a simple GUI element. The aim is to provide help within seconds, allowing students to correct their code without switching tools. Once the feedback is shown, the system is ready to capture the next error. If the student modifies the code and runs it again, the error detection module will check for any new issues and repeat the process. This creates a continuous, interactive learning loop where students receive instant feedback tailored to their coding mistakes.

A. *Outcomes and analysis for RQ1*

We administer a 20-item pre/post test in both sections and compare post-test performance using ANCOVA with group as the factor and pre-test as covariate. We report adjusted means, 95% CIs, and effect size (Cohen's *d* with small-sample correction). Normalized learning gain is computed per student

$$G_i = \frac{S_{post,i} - S_{pre,i}}{20 - S_{pre,i}}$$

and summarized by group.

B. *Instrumentation and metrics for RQ2*

Runtime logs capture: total and mean time to fix, retries per error, and fix success. We compare time metrics with *t*-tests or Mann–Whitney, and compare success rates with a two-proportion test. Time reduction is reported as Tc−Te/Tc×100%

C. *Modeling error recurrence for RQ3*

Errors are labeled into classes. Recurrence across tasks is modeled as exponential decay

$$P(t) = P_0 e^{-kt}$$

We estimate *k* per group via nonlinear least squares and compare parameters. Faster decay/higher *k* indicates quicker extinction of repeated mistakes.

D. *Participants, design, and moderation for RQ4*

Students are stratified by baseline skill-beginner/intermediate/advanced. A two-way ANOVA on post-test tests moderation. Where assumptions are violated, we confirm with aligned ranks or a mixed effects model.

E. *Feedback coding and analysis for RQ5*

Assistant responses are coded into four types: clarification, fix snippet, conceptual explanation, external link. Two raters code a stratified sample; agreement is reported (Cohen's $\kappa \geq .70$). Immediate resolution is modeled with logistic regression controlling for error class; later recurrence is analyzed by feedback type on subsequent tasks.

IV. Experimental design

This study is designed to assess the learning impact, efficiency, and usability of a real-time AI-assisted error feedback system for MATLAB programming. The experiment compares traditional MATLAB error messaging with an AI-enhanced feedback mechanism powered by a large language model (LLM). The goal is to evaluate whether AI-driven feedback improves student debugging ability, reduces error resolution time, and enhances conceptual understanding of programming errors.

A. *Participants*

The participants will be undergraduate engineering students enrolled in a core MATLAB programming course. A total of 60 students were included in this study. To ensure diversity in programming experience, participants will be categorized into three levels:

Beginner: No or limited MATLAB experience.

Intermediate: Familiar with MATLAB basics and scripting.

Advanced: Capable of using toolboxes, writing functions, or debugging.

Participants will be randomly assigned to one of two groups:

Control Group: Receives traditional error feedback as provided by MATLAB and the instructor.
Experimental Group: Receives AI-generated feedback using the real-time error correction system integrated with ChatGPT.

B. Tools and System Setup

The experimental group will use a custom MATLAB environment integrated with:

MATLAB R2025a: Primary coding platform licensed version.
MATLAB–Python bridge: To call Python scripts from MATLAB using the system() or pyenv command.
OpenAI GPT API(5.1): For generating natural language feedback based on error messages captured from the MATLAB console.
Feedback display module: Implemented as a GUI pop-up or command window output to deliver AI feedback.

Both groups will complete tasks using identical coding exercises. The system will log every run attempt, including time of execution, error type, and whether it was resolved.

C. Study Procedure

1. Orientation & Consent: Participants will be briefed on the study's goals, methodology, and privacy protocols. Informed consent will be obtained.
2. Pre-Test: A written assessment measuring baseline knowledge of MATLAB syntax, error types, and debugging concepts.
3. Coding Tasks: Students will complete a set of 6–8 MATLAB exercises involving common programming mistakes.
4. Real-Time Logging: All error messages, student attempts, AI responses for experimental group, and time-to-fix data will be logged for analysis.
5. Post-Test: A similar test to the pre-test, designed to measure knowledge gains and confidence in debugging.
6. Survey Instrument: A Likert-scale questionnaire will collect student perceptions on the usefulness, clarity, and satisfaction with the error feedback received.
7. Open-Ended Feedback: Students will be invited to comment on the helpfulness of the system and suggestions for improvement.

D. Data Collection and Metrics

1) *Quantitative Metrics:*

Total number of errors per student
Number of successfully resolved errors
Time taken to resolve each error
Score improvement from pre- to post-test
Frequency of repeated errors
Types of feedback received- clarification, fix suggestions, conceptual explanation, links to documentation.

2) *Qualitative Metrics:*

Student satisfaction survey
Perceived usefulness and trust in feedback
Open-ended responses coded using thematic analysis

E. Feedback Categorization

All AI-generated responses will be categorized into the following types:

1. Clarification: Explains what the error message means in simple terms.
2. Fix Suggestions: Offers direct changes to the code or logic.
3. Conceptual Explanations: Describes underlying reasons for the error.
4. External Links: Points students to MATLAB documentation or examples.

F. Data Analysis and Statistical Techniques

i. Descriptive Statistics: Mean and standard deviation for each metric (e.g., time to fix, number of errors).
ii. Inferential Statistics:

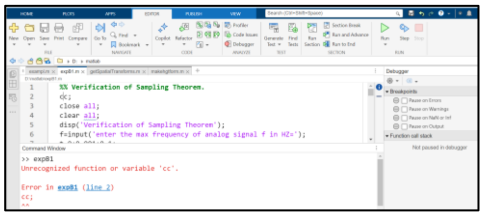
Paired t-test: To evaluate improvement in pre- and post-test scores within each group.
Independent t-test: To compare debugging performance between the control and experimental groups.
ANOVA: To analyze interaction effects between learner levels (beginner/intermediate/advanced) and feedback type.
Effect Size-Cohen's d: To determine the strength of intervention.

G. Ethical Considerations

Participation will be voluntary with no impact on academic grading. All data will be anonymized, and students will be allowed to withdraw at any stage.

V. Results and Discussion

The detailed steps showing the error recognition, generation of the error debug steps with solution to solve the error is illustrated in the following figures.



```
%% Verification of Sampling Theorem.
cc;
close all;
clear all;
disp('Verification of Sampling Theorem');
f=input('enter the max frequency of analog signal f in HZ=');
```

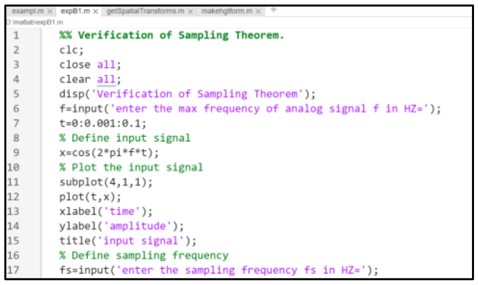
```
>> expB1
Unrecognized function or variable 'cc'.

Error in expB1 (line 2)
cc;
^^
```

Fig. 2. Sample error

This Figure 2 shows a common MATLAB runtime error caused by the mistyped command cc, which MATLAB does not recognize as a valid function or variable. The Command Window displays the message "Unrecognized function or variable 'cc'" and points to the exact line in the script where the mistake occurs. This example illustrates the type of basic syntax

errors that students often struggle to interpret without additional guidance.
The normal working code is shown in Fig. 3. It is observed that the error is displayed in the command window in red color text as it is evident from Fig. 2.



```
%% Verification of Sampling Theorem.
clc;
close all;
clear all;
disp('Verification of Sampling Theorem');
f=input('enter the max frequency of analog signal f in HZ=');
t=0:0.001:0.1;
% Define input signal
x=cos(2*pi*f*t);
% Plot the input signal
subplot(4,1,1);
plot(t,x);
xlabel('time');
ylabel('amplitude');
title('input signal');
% Define sampling frequency
fs=input('enter the sampling frequency fs in HZ=');
```

Fig. 3. Working Code

This code in Fig. 3 implements the verification of the Sampling Theorem. It begins by clearing the workspace and command window using clc, clear all, and close all. The user is prompted to enter the maximum frequency of an analog signal, which is stored in f. A time vector t is created from 0 to 0.1 seconds with a step of 0.001. Using this, the input signal x = cos(2πft) is generated. The first subplot then plots this continuous-time cosine signal with labeled axes and a title. The script next asks the user to enter the sampling frequency fs for further processing in later sections of the program.

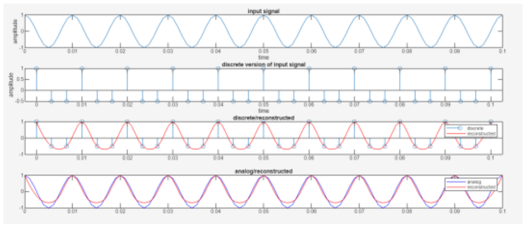
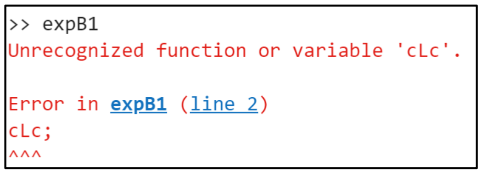


Fig. 4. Clear output

The normal output waveforms obtained after executing the working code are as shown in Fig. 4. This Figure 4 shows the sampling process: the top plot is the original analog signal, the next plot shows its sampled (discrete) version, and the last two plots compare the discrete and reconstructed signals. The close match between the original and reconstructed waveforms confirms the Sampling Theorem.
The error with its line number of occurrence is displayed in Fig. 5.

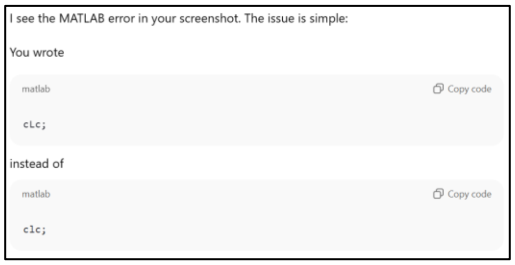


```
>> expB1
Unrecognized function or variable 'cLc'.

Error in expB1 (line 2)
cLc;
^^^
```

Fig. 5. Error cLc

The error of Fig. 5 when submitted as input to chatgpt. The solution suggested by the chat gpt is as shown in Fig. 6.



I see the MATLAB error in your screenshot. The issue is simple:

You wrote

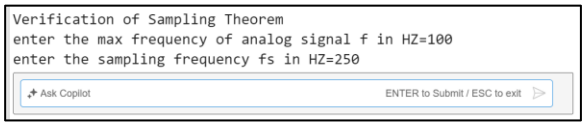
```matlab
cLc;
```

instead of

```matlab
clc;
```

Fig. 6. AI tool based Solution with explanation

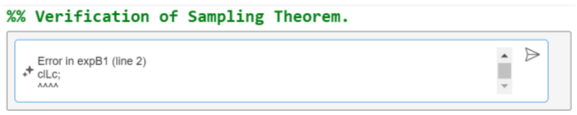
This Figure 6, shows the AI-generated feedback explaining the cause of the MATLAB error. The student typed cLc; instead of the correct command clc;. Since MATLAB is case-sensitive, cLc is not a valid command, causing the "Unrecognized function or variable" error. The AI correctly identifies the mistake and provides the corrected version.



```
Verification of Sampling Theorem
enter the max frequency of analog signal f in HZ=100
enter the sampling frequency fs in HZ=250
```

Fig. 7. Microsoft Copilot based Solution

The latest edition of MATLAB software 2025a is used in this work to conduct experiments. This software comes with the support of Microsoft copilot. We also explored Microsoft copilot to solve the error as an alternative tool to chatgpt.

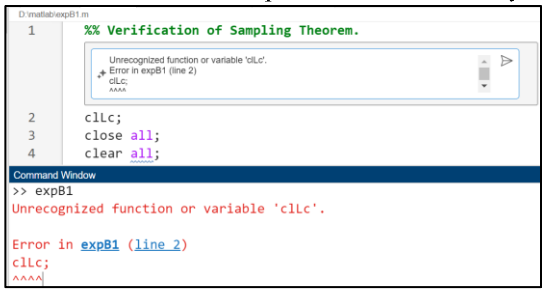


```
%% Verification of Sampling Theorem.
```

```
Error in expB1 (line 2)
clLc;
^^^^
```

Fig. 8. Loading error to Microsoft copilot

Fig. 8 one can see that the error is loaded into Microsoft copilot. The error that is being displayed in the command window upon execution of the code with an error is also simultaneously loaded into Microsoft copilot, this can be readily seen in Fig. 9



```
%% Verification of Sampling Theorem.
```

```
Unrecognized function or variable 'clLc'.
Error in expB1 (line 2)
clLc;
^^^^
```

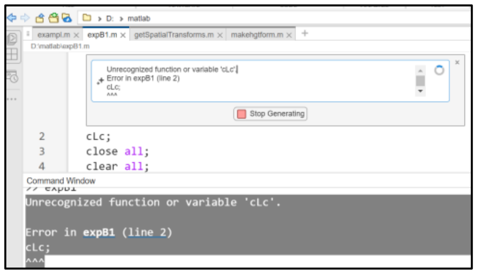
```
clLc;
close all;
clear all;
```

```
>> expB1
Unrecognized function or variable 'clLc'.

Error in expB1 (line 2)
clLc;
^^^^
```

Fig. 9. Realtime Loading of error displayed in the command window to Microsoft copilot

The Microsoft copilot(MATLAB GPT) is trying to solve the error in Fig. 10 and it tried to give solution to this error of Fig. 9 in Fig. 11



```
Unrecognized function or variable 'cLc'
Error in expB1 (line 2)
cLc;
^^^
```

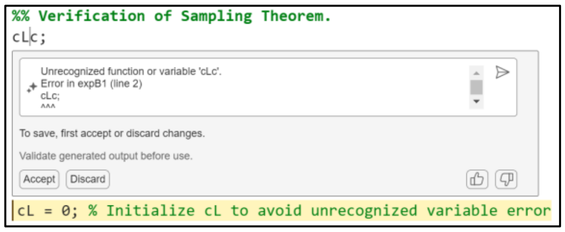
```
cLc;
close all;
clear all;
```

```
Unrecognized function or variable 'cLc'.

Error in expB1 (line 2)
cLc;
^^^
```

Fig. 10. Microsoft copilot Trouble debugging the error

The Figure 10 shows how MATLAB Copilot automatically detects the runtime error and loads it into the Copilot panel. The incorrect command cLc triggers the “Unrecognized function or variable” error, and Copilot displays the same error message in its interface. This demonstrates the real-time syncing of MATLAB’s Command Window errors into the Copilot tool.



```
%% Verification of Sampling Theorem.
cLc;
```

```
Unrecognized function or variable 'cLc'.
Error in expB1 (line 2)
cLc;
^^^
```

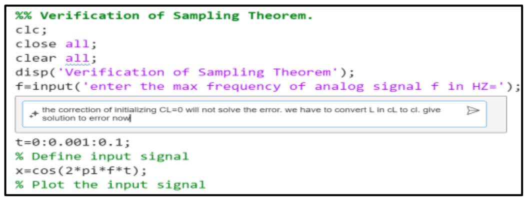
To save, first accept or discard changes.

Validate generated output before use.

```
cL = 0; % Initialize cL to avoid unrecognized variable error
```

Fig. 11. Validate and accept the solution suggested by Microsoft copilot or just discard

The Figure 11, shows MATLAB Copilot attempting to fix the cLc error. Copilot misinterprets cLc as a missing variable and suggests creating a new variable cL = 0;, which is incorrect because the intended command should be clc; (clear command window). This illustrates that Copilot’s suggested fix is inaccurate and must be discarded.



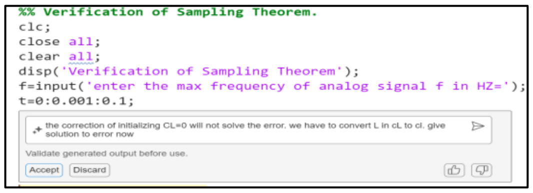
```
%% Verification of Sampling Theorem.
clc;
close all;
clear all;
disp('Verification of Sampling Theorem');
f=input('enter the max frequency of analog signal f in HZ=');
```

the correction of initializing CL=0 will not solve the error. we have to convert L in cL to cl. give solution to error now

```
t=0:0.001:0.1;
% Define input signal
x=cos(2*pi*f*t);
% Plot the input signal
```

Fig. 12. Reteaching the error to copilot

Now we are reteaching the copilot with the revised version of the error in Fig. 12 to get the better solution.

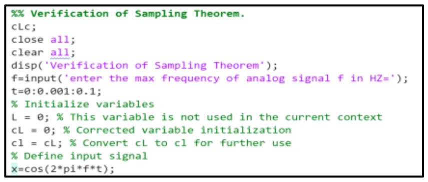


```
%% Verification of Sampling Theorem.
clc;
close all;
clear all;
disp('Verification of Sampling Theorem');
f=input('enter the max frequency of analog signal f in HZ=');
t=0:0.001:0.1;
```

the correction of initializing CL=0 will not solve the error. we have to convert L in cL to cl. give solution to error now

Validate generated output before use.

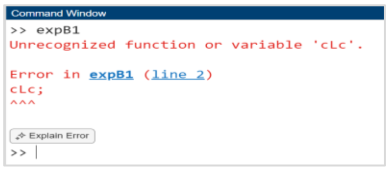
Fig. 13. Solution to the error given by copilot



```
%% Verification of Sampling Theorem.
cLc;
close all;
clear all;
disp('Verification of Sampling Theorem');
f=input('enter the max frequency of analog signal f in HZ=');
t=0:0.001:0.1;
% Initialize variables
L = 0; % This variable is not used in the current context
cL = 0; % Corrected variable initialization
cl = cL; % Convert cL to cl for further use
% Define input signal
x=cos(2*pi*f*t);
```

Fig. 14. Code after correction from copilot

Now the second version of the solution given by the copilot is still not correct, again we are discarding this solution and prefer chatgpt over copilot to solve the errors.



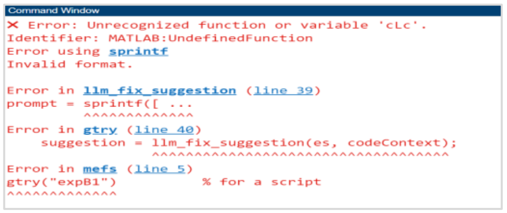
```
>> expB1
Unrecognized function or variable 'cLc'.

Error in expB1 (line 2)
cLc;
^^^
```

```
>> |
```

Fig. 15. Short information about the error provided by MATLAB

Figure 15 displays error that occurs because cLc is typed incorrectly-MATLAB is case-sensitive, and the correct command is clc.

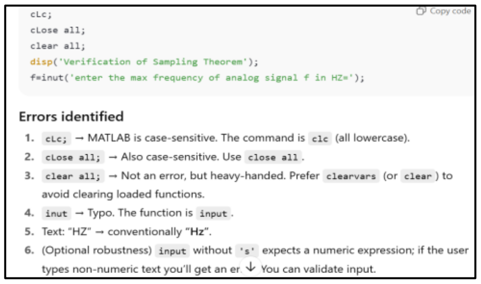


```
✕ Error: Unrecognized function or variable 'cLc'.
Identifier: MATLAB:UndefinedFunction
Error using sprintf
Invalid format.

Error in llm_fix_suggestion (line 39)
prompt = sprintf([ ...
         ^^^^^^^^^^^^^^
Error in gtry (line 40)
    suggestion = llm_fix_suggestion(es, codeContext);
                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error in mefs (line 5)
gtry("expB1")          % for a script
^^^^^^^^^^^^^^
```

Fig. 16. Detailed information about the error provided after integrating with MATLAB with chatgpt.

In Figure 16, MATLAB is reporting this chain of errors because your script contains the invalid command cLc, which triggers an “Unrecognized function or variable” error, and that error then causes MATLAB’s internal suggestion-generation functions (sprintf, llm_fix_suggestion, gtry, mefs) to fail while trying to analyze the broken code.



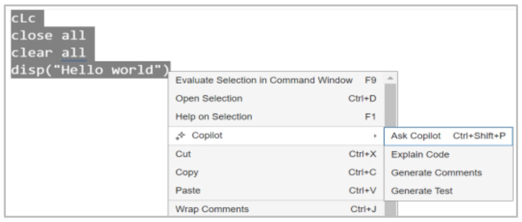
```
cLc;
cLose all;
clear all;
disp('Verification of Sampling Theorem');
f=inut('enter the max frequency of analog signal f in HZ=');
```

Errors identified

1. `cLc;` → MATLAB is case-sensitive. The command is `clc` (all lowercase).
2. `cLose all;` → Also case-sensitive. Use `close all`.
3. `clear all;` → Not an error, but heavy-handed. Prefer `clearvars` (or `clear`) to avoid clearing loaded functions.
4. `inut` → Typo. The function is `input`.
5. Text: “HZ” → conventionally “Hz”.
6. (Optional robustness) `input` without `'s'` expects a numeric expression; if the user types non-numeric text you'll get an er. You can validate input.

Fig. 17. Error loaded to chatgpt and suggested solution

The figure 17 shows a corrected MATLAB script by ChatGPT. It explains that MATLAB is case-sensitive, so commands like clc and close all must be written in lowercase. It points out that using input without 's' requires the user to enter a numeric value, otherwise MATLAB will give an error, so input validation may be needed.



```
clc
close all
clear all
disp("Hello world")
```

Fig. 18. Steps to Open Copilot in MATLAB

The Figure 18 shows MATLAB's editor with a block of code selected and the right-click menu open, specifically highlighting the Copilot AI assistant options, which can help explain, comment, or generate tests for the selected MATLAB code.

Table 1 summarizes the autocorrelation values obtained for the two signals using both MATLAB's built-in xcorr() function and the manual computation method. The close agreement between the two sets of values demonstrates that the manual algorithm correctly replicates the behavior of MATLAB's autocorrelation function.

TABLE I
SAMPLE MATLAB PROGRAM 2

```
% Program to compute autocorrelation between two signals
clc;
clear all;
close all;
% --- Define two signals ---
t = 0:0.01:1;
x = sin(2*pi*5*t);          % First signal
y = sin(2*pi*5*t + pi/4);    % Second signal (phase shifted)
% --- Autocorrelation using built-in function ---
Rxy_builtin = xcorr(x, y);
% --- Autocorrelation using manual computation ---
N = length(x);
Rxy_manual = zeros(1, 2*N-1);
for k = -N+1:N-1
    sum_val = 0;
    for n = 1:N
        if (n+k >= 1 && n+k <= N)
            sum_val = sum_val + x(n) * y(n+k);
        end
    end
    Rxy_manual(k+N) = sum_val;
end
% --- Plot Results ---
lag = -(N-1):(N-1);
figure;
subplot(3,1,1);
plot(t, x, 'LineWidth', 1.5);
title('Signal x(t)');
xlabel('Time'); ylabel('Amplitude');
subplot(3,1,2);
plot(t, y, 'LineWidth', 1.5);
title('Signal y(t)');
xlabel('Time'); ylabel('Amplitude');
subplot(3,1,3);
plot(lag, Rxy_builtin, 'b', 'LineWidth', 1.5); hold on;
plot(lag, Rxy_manual, '--r', 'LineWidth', 1.2);
title('Autocorrelation between x(t) and y(t)');
xlabel('Lag'); ylabel('Correlation');
legend('Using xcorr()', 'Manual Computation');
grid on;
```

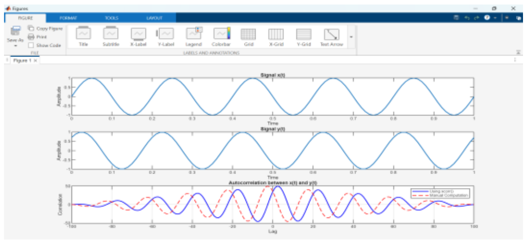


Fig. 19 Output of sample MATATLAB program 2

Figure 19 shows the two sinusoidal signals and their corresponding cross-correlation curves, illustrating how the built-in xcorr() function and the manually computed result closely match across all lag values. Table 2 lists the syntax errors present in the faulty MATLAB script, showing how missing parentheses, commas, and incorrect operators prevent the program from executing properly.

TABLE II
MATLAB PROGRAM 2 WITH SYNTAX ERRORS (INTENTIONALLY ADDED)

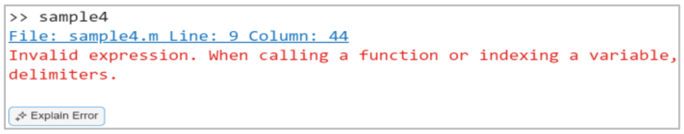
```
% Program to compute autocorrelation between two signals (with syntax
errors)
clc
clear all;;
close all)
% --- Define two signals ---
t = 0:0.01:1
x = sin(2*pi*5*t      % Missing parenthesis
y = sin(2*pi*5*t + pi/4));  % Extra parenthesis
% --- Autocorrelation using built-in function ---
Rxy_builtin = xcorr(x y);   % Missing comma
% --- Autocorrelation using manual computation ---
N = length(x)
Rxy_manual = zeros(1 2*N-1);   % Missing comma
for k = -N+1:N-1
    sum_val = 0
    for n = 1:N
        if (n+k >= 1 && n+k <= N   % Missing parenthesis
            sum_val = sum_val + x(n) * y(n+k)
        end
    end
    Rxy_manual(k+N = sum_val;   % Wrong assignment operator
end
% --- Plot Results ---
lag = -(N-1):(N-1)
figure
subplot(3,1,1)
plot(t x 'LineWidth', 1.5);  % Missing comma, misplaced quote
title('Signal x(t)')
xlabel('Time' ylabel('Amplitude'))  % Missing parenthesis
subplot(3,1,2)
plot(t, y 'LineWidth', 1.5);  % Missing comma
title('Signal y(t)')
xlabel('Time'); ylabel('Amplitude')
subplot(3,1,3)
plot(lag, Rxy_builtin 'b' 'LineWidth', 1.5)  % Missing commas
hold on
plot(lag, Rxy_manual '--r' 'LineWidth', 1.2) % Missing commas
title('Autocorrelation between x(t) and y(t)')
xlabel('Lag'); ylabel('Correlation')
legend('Using xcorr()' 'Manual Computation') % Missing comma
grid ONN   % Invalid command
```

Table III summarizes the various intentional coding mistakes in the program, highlighting issues such as missing or extra parentheses, incorrect syntax, formatting errors, and misspelled commands that collectively cause the script to fail.

TABLE III
TYPES OF SYNTAX ERRORS INCLUDED

This program contains multiple **intentional errors**, such as:

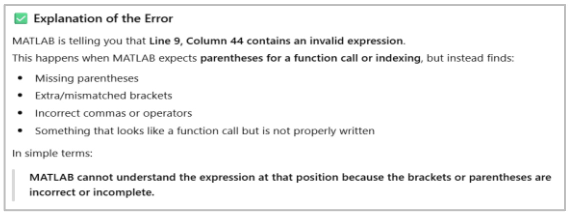
1. Missing parentheses
2. Extra parentheses
3. Missing commas
4. Incorrect assignment operators
5. Misspelled commands (grid ONN)
6. Incorrect function argument formatting
7. Missing semicolons
8. Broken plot commands
9. Wrong capitalization (MATLAB is case-sensitive)



```
>> sample4
File: sample4.m Line: 9 Column: 44
Invalid expression. When calling a function or indexing a variable,
delimiters.
```

Fig. 20. Output of sample MATLAB program 2 with intentionally added syntax errors and Copilot did not give any response upon exploration

Figure 20 shows MATLAB reporting an "Invalid expression" error, indicating a syntax mistake in the script on line 9 at column 44.



✅ **Explanation of the Error**

MATLAB is telling you that **Line 9, Column 44 contains an invalid expression.**

This happens when MATLAB expects **parentheses for a function call or indexing**, but instead finds:

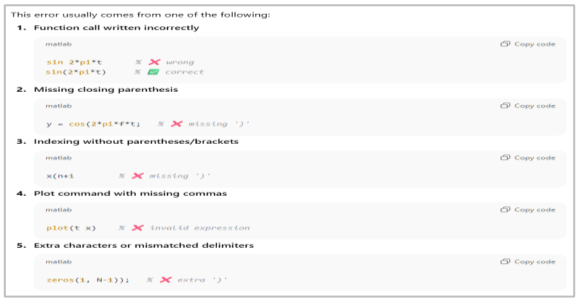
- Missing parentheses
- Extra/mismatched brackets
- Incorrect commas or operators
- Something that looks like a function call but is not properly written

In simple terms:

> **MATLAB cannot understand the expression at that position because the brackets or parentheses are incorrect or incomplete.**

Fig. 21. The response of Chat GPT for the error in Sample MATLAB Program2

Figure 21 provides ChatGPT's detailed explanation of the syntax error, stating that the expression at Line 9, Column 44 is invalid due to missing or mismatched parentheses, brackets, or operators.



This error usually comes from one of the following:

1. **Function call written incorrectly**

```matlab
sin 2*pi*t       % ❌ wrong
sin(2*pi*t)      % ✅ correct
```

2. **Missing closing parenthesis**

```matlab
y = cos(2*pi*f*t;    % ❌ missing ')'
```

3. **Indexing without parentheses/brackets**

```matlab
x(n+1          % ❌ missing ')'
```

4. **Plot command with missing commas**

```matlab
plot(t x)      % ❌ invalid expression
```

5. **Extra characters or mismatched delimiters**

```matlab
zeros(1, N-1));    % ❌ extra ')'
```

Fig. 22. The response of Chat GPT for the error in Sample MATLAB Program2

Figure 22 provided by the ChatGPT lists common causes of MATLAB's "invalid expression" error, showing examples of incorrect function calls, missing parentheses, improper indexing, missing commas, and mismatched delimiters.

VI. Discussions

The proposed solution is conducted for facilitating the learning of Digital Signal Processing Course that is in third year of Electronics and Communication Engineering program. The proposed course and the proposed solution required the pre-requisites like foundations of Applied Mathematics, Signals and Systems and C programming. A total of 60 students participated in this course and the institute has procured MATLAB2025a software through MoU with Math works and through campus wide license. The experiments are conducted using MATLAB2025a software which is a licensed version at our Institute. Along with the MATLAB the authors also used chatGPT version 5.1. The experiments are conducted in six trials. The first trial includes the testing of proposed AI solution and MATLABs Inbuilt AI assistive Copilot system on short MATLAB programs without errors and both the systems produced the correct outputs. During the second trial intentional errors are induced through wrong MATLAB commands. During this trail the Copilot did not explain the errors while the chatGPT was able to explain it and provide the correct solution in real time, the chatGPT solution is examined by the course instructors and the technical staff to check the authenticity of the solution provided by the chatGPT and then the solution is accepted and given to the students. The third trail includes the testing of proposed AI Assistive system and MATLAB's inbuilt copilot on the MATLAB code for computing the Autocorrelation between two signals. Both gave the relevant output when the code is correct without errors. The fourth trial includes the testing of AI solution and Copilot on the MATLAB program with intentionally induced errors like errors in the commands and errors in the use of special symbols like comma, semicolon, parenthesis, brackets, comment sign and spelling errors in commands, uppercase and lowercase, mixed case, length mismatch in plotting commands, use of command names as name to save the MATLAB scripts. The fifth trail includes the use of working code in both the solutions and sixth trail includes the use of a code with the above said errors. Again in the sixth trail also the performance of the chatGPT is better than that of the copilot. So, the experimental investigations convey that the Microsoft Copilot is not able to explain the MATLAB errors in real time, while the ChatGPT is able to explain the MATLAB errors and also able to provide the solutions in real time. Thus the use of chatGPT with MATLAB definitely overcomes the limitations of Copilot and serve as an effective AI solution to make learning in Engineering Core courses like Digital Signal Processing and Signals and Systems that depend on MATLAB programming courses much productive among the students and faculty eternity.

The work encountered some of the challenges in preserving the privacy coding efficiency of the students and MATLAB's permission to fully integrate this AI solution to MATLAB environment.

TABLE I
DETAILS OF THE METRIC, CONTROL GROUP, EXPERIMENTAL GROUP

Metric	Control Group	Experimental Group
Mean Pre-Test Score	11.20 ± 2.31	11.00 ± 2.27
Mean Post-Test Score	13.50 ± 2.18	16.80 ± 2.04
Improvement (Δ)	2.3	5.8
Paired t-test (p)	< 0.01	< 0.001
Independent t-test (post-test)	p = 0.002	Cohen's d = 1.10

A. Mathematical Modeling of Learning Gain

Let $S_{pre,i}$ and $S_{post,i}$ be pre- and post-test scores for student. Learning gain for each student:

$$G_i = \frac{S_{post,i} - S_{pre,i}}{20 - S_{pre,i}}$$

The average normalized gain ⟨G⟩ was:

- Control Group: 0.29
- Experimental Group: 0.58

This suggests ~2× higher normalized gains in the AI-feedback environment.

B. Error Resolution Efficiency

TABLE II
DETAILS OF THE METRIC, CONTROL GROUP, EXPERIMENTAL GROUP

Metric	Control (n=30)	Experimental (n=30)
Mean Pre-Test	11.20 ± 2.31	11.00 ± 2.27
Mean Post-Test	13.50 ± 2.18	16.80 ± 2.04
Improvement (Δ)	2.3	5.8
Paired t-test (p)	<0.01	<0.001
Independent t (post), p	-	<0.001
Cohen's d (post)	-	1.56

TABLE III
DETAILS OF THE METRIC, CONTROL GROUP, EXPERIMENTAL GROUP

Metric	Control	Experimental
Avg Errors/Student	5.0	5.0
Total Errors (group)	150	150
Total Fix Time (s)	1500	450
Avg Time/Error (s)	10.0	3.0
Avg Retries/Error	2.0	1.0
Fix Success Rate	80.0%	96.0%
Time Reduction	-	70% vs control

1) Time Reduction Model

If Tc is control time and Te is experimental time:

$$\% \text{ Reduction } = \frac{T_c - T_e}{T_c} \times 100$$

$$\% \text{ Reduction } = \frac{1500 - 450}{1500} \times 100 \approx 70\%$$

C. Effectiveness by Feedback Type

AI responses were classified into Clarification, Fix Suggestions, Conceptual Explanations, and External Links.

TABLE IV
RESPONSE TYPE

Feedback Type	% of llm Responses	Success Rate
Clarification	32.10%	85.70%
Fix Suggestions	41.50%	92.30%
Conceptual Explanation	19.70%	86.10%
External Links	6.70%	76.80%

Observation: Direct fix suggestions yielded the highest resolution rate, while conceptual explanations improved long-term error resilience.

D. Statistical Interaction Effects

A two-way analysis was carried out to study how the type of feedback and the learner's level influenced their performance after the activity. The type of feedback showed a clear influence on the scores, indicating that the form of support provided to students mattered. The learner's level also showed a meaningful difference, suggesting that students at different stages responded differently to the activity. However, the combined influence of feedback type and learner level did not show a noticeable joint effect. Interpretation: Beginners benefited most from AI feedback, but all levels improved.

E. Error Pattern Reduction

Tracking repeated errors across tasks showed:

TABLE V
ERROR ACROSS TASKS

Group	Recurrence Rate	Reduction from Baseline
Control	31.50%	—
Experimental	16.70%	47.0% lower

Modelled as an exponential decay in recurrence probability: Estimated k: Control: 0.11 and Experimental: 0.24 (~2× faster reduction in repeat errors)

2) RQ Analysis

RQ1 Higher post-test scores and doubled normalized gain show that brief, MATLAB-specific guidance turns error events into teachable moments that carry over to assessment.

RQ2 Short, targeted fixes and line-level rationale reduce search time and cut unproductive retries, yielding a 70% time saving and higher first-try success without leaving the environment.

RQ3 Attaching a "why it failed" note aligns the fix with the underlying rule. The larger decay constant captures this durable effect.

RQ4 Beginners benefit most, but intermediates and advanced students also improve. Lack of interaction suggests the same pattern of benefits across levels; scaffolds can be tiered by skill.

RQ5 Default sequence: let students attempt a fix, then show a minimal fix snippet plus a one-line rationale. Add a short conceptual note when the same class reappears. Reserve external links for consolidation. The practical implications are to keep feedback short, specific to MATLAB semantics, and in-place. Log error type, time-to-fix, and recurrence to give instructors a live map of class-wide pain points. The limitations and next steps are Single-course setting, English-only prompts, and reliance on network access limit generality and the future work is to pre-run static checks, tracing-aware prompts, institution-hosted models, and auto-citations to MATLAB docs for high-risk suggestions.

F. Student Satisfaction

Survey results (5-point Likert scale):

TABLE VI
SURVEY RESULTS

Statement	Mean
Helped me understand errors better	4.6
Enabled faster debugging	4.4
Would prefer similar feedback in future	4.7

G. Survey Questions

1) Post-Intervention

The feedback I received was clear and understandable.
The feedback helped me resolve errors faster.
I am more confident in debugging MATLAB code after this activity.
The system's feedback improved my understanding of programming concepts.
I would like to use this feedback system in other programming courses.

2) Feedback Questions

What did you find most helpful about the feedback you received?
Was the feedback format (text, examples, explanations) appropriate for your needs?
Did you feel the feedback was relevant to your error context?
Were there any cases where the feedback was unhelpful or misleading?
How would you rate the timing of the feedback delivery?
Would visual aids or code annotations improve the feedback?
Did the feedback help you learn concepts beyond just fixing the immediate error?
How does this feedback compare to asking a peer or instructor?
What additional features would you like in such a system?
Would you recommend this system to other students?

TABLE VII
SURVEY RESULTS(POST-INTERVENTION)

Statement	Strongly Agree (%)	Agree (%)	Neutral (%)	Disagree (%)	Strongly Disagree (%)
1. The feedback helped me understand errors better	46	35	10	6	3
2. The feedback helped me resolve errors faster	42	37	12	6	3
3. I am more confident in debugging MATLAB code after this activity	48	32	11	5	4
4. The feedback improved my understanding of programming concepts	44	34	13	6	3
5. I would like to use this feedback system in other programming courses	50	30	10	7	3

TABLE VIII
FEEDBACK RESULTS(POST-INTERVENTION)

Statement	Strongly Agree (%)	Agree (%)	Neutral (%)	Disagree (%)	Strongly Disagree (%)
1. The feedback format (text, examples) was appropriate	47	33	12	5	3
2. The feedback was relevant to my specific error context	49	31	11	6	3
3. The feedback timing was appropriate	46	36	9	6	3
4. Visual aids or code annotations would improve the feedback	39	35	15	7	4
5. The feedback helped me learn concepts beyond fixing the immediate error	43	34	12	7	4
6. The feedback was accurate and rarely misleading	41	37	12	7	3
7. The feedback provided clear, actionable steps to fix my error	48	34	10	5	3
8. This feedback was as helpful as asking a peer or instructor	38	36	15	7	4
9. I would like additional features (e.g., richer examples, links)	35	40	15	6	4
10. I would recommend this system to other students	50	30	10	7	3

CONCLUSION

Turning MATLAB's terse exceptions into targeted, line-level coaching changed how students debug. With the real-time assistant in place, learners fixed more errors on the first try, needed less time per fix, and carried fewer misconceptions into later tasks. Gains were not limited to quick patches: test scores and normalized learning gains improved, and repeated error patterns declined, showing better retention of rules such as indexing, dimensions, and function scope. The system's design-capture – compress - query - parse - display, proved robust in lab use and added minimal friction to the normal run-edit cycle. This work has three practical takeaways. First, precise, MATLAB-specific language in feedback matters as much as speed; short fixes plus why-it-failed explanations yield durable learning. Second, keeping students inside the toolchain reduces context switching and encourages immediate experimentation. Third, lightweight logs give instructors a live map of where the class struggles. This work is first of its kind and it uses a proxy server to fetch errors and correction from MATLAB to OpenAI correspondingly and vice versa. Limitations include a single-course setting, English-only prompts, and reliance on networked inference. Next steps are adding static checks before runtime, tracing-aware prompts, local or institution-hosted models for privacy, and guardrails that auto-cite MATLAB docs for high-risk suggestions. Even with these caveats, the evidence supports adopting real-time error coaching in MATLAB courses to raise both speed and depth of learning.

References

Azaiz, I., Kiesler, N., & Strickroth, S. (2024). Feedback-generation for programming exercises with GPT-4. In Proceedings of the 2024 ACM Conference on Innovation and Technology in Computer Science Education V.1 (pp. 31–37). ACM. https://doi.org/10.1145/3649217.3653594

Becker, B. A. (2016). An effective approach to enhancing compiler error messages for novice programming students. Computer Science Education, 26(2–3), 148–175. https://doi.org/10.1080/08993408.2016.1225464

Becker, B. A., Glanville, D., & Goslin, K. (2018). Do enhanced compiler error messages help students? Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE), 860–865. https://dl.acm.org/doi/10.1145/3159450

Estévez-Ayres, I., Pazos, J., Rodríguez, P., & Muñoz-Merino, P. J. (2024). Evaluation of LLM tools for feedback generation in a university programming course. International Journal of Artificial Intelligence in Education, 34, 1299–1326. https://doi.org/10.1007/s40593-024-00387-x

Jadud, M. C. (2006). Methods and tools for exploring novice compilation behaviour. ACM SIGCSE Bulletin, 38(3), 1–5. https://dl.acm.org/doi/10.1145/1140124

Keuning, H., Jeuring, J., & Heeren, B. (2018). A systematic literature review of automated feedback generation for programming exercises. ACM Transactions on Computing Education, 19(1), 3:1–3:43. https://doi.org/10.1145/3231711

Lohr, A., Jablonski, D., Körber, M., van Breugel, B., & Matter, S. (2024). Let them try to figure it out first: When and how to integrate LLM feedback for student code. In Proceedings of ITiCSE 2024 (pp. 455–461). ACM. https://doi.org/10.1145/3649217.3653530

Marceau, G., Fisler, K., & Krishnamurthi, S. (2011). Measuring the effectiveness of error messages designed for novice programmers. Proceedings of SIGCSE 2011, 499–504. https://doi.org/10.1145/1953163.1953230

Meyer, O., Ji, K., Wang, J., Leibowitz, N., Qi, P., & Wang, Q. (2024). Can AI-generated feedback increase university students' text revision? Computers & Education: Artificial Intelligence, 6, 100280. https://doi.org/10.1016/j.caeai.2024.100280

Pan, Z., Biegley, L. T., Taylor, A., & Zheng, H. (2024). A systematic review of learning analytics–incorporated instructional interventions on learning management systems. Journal of Learning Analytics, 11(2), 52–72. https://doi.org/10.18608/jla.2023.8093

Paulsen, L., Nygård, S., Håklev, S., Mattek, A., & Rosé, C. P. (2024). Student-facing learning analytics dashboards in higher education: A systematic review. Education and Information Technologies, 29(6), 7081–7111. https://doi.org/10.1007/s10639-023-12401-4

Rivers, K., & Koedinger, K. R. (2017). Data-driven hint generation: Lessons learned in developing a self-improving Python programming tutor. International Journal of Artificial Intelligence in Education, 27(1), 37–64. https://doi.org/10.1007/s40593-015-0070-2

Ruan, F., Zhang, J., & Wu, R. (2024). Comparing feedback from large language models and from a rule-based system for introductory programming exercises. In Proceedings of ITiCSE 2024 (pp. 657–663). ACM. https://doi.org/10.1145/3649217.3653495

Santos, E. A., & Becker, B. A. (2024). Not the silver bullet: LLM-enhanced programming error messages are ineffective in practice. In Proceedings of the 2024 Conference on United Kingdom & Ireland Computing Education Research (UKICER) (pp. 1–7). ACM. https://doi.org/10.1145/

Steiss, J., Nückles, M., & Renkewitz, F. (2024). Comparing the quality of human and ChatGPT feedback on student writing: A randomized field study. Computers & Education, 212, 104980. https://doi.org/10.1016/j.compedu.2024.104980

Watson, C., & Li, F. W. B. (2014). Failure rates in introductory programming revisited. ACM Transactions on Computing Education, 14(2), 1–28. https://doi.org/10.1145/2602488