Reflective Report 2B, S2-P1

Design and Creative Technologies

Torrens University, Australia

Student: Luis Guilherme de Barros Andrade Faria - A00187785

Subject Code: MFA501

Subject Name: Mathematical Foundations of Artificial Intelligence

Assessment No.: 2B

Title of Assessment: Reflective Report, Set 2, Problem 1

Lecturer: Dr. James Vakilian

Date: Nov 2025

Copyright © 2025 by Luis G B A Faria

Permission is hereby granted to make and distribute verbatim copies of this document provided the copyright notice and this permission notice are preserved on all copies.

Table of Contents

1. Introduction and Overview 3
2. Mathematical Approach 5
2.1. Composite Trapezoid Rule 5
2.2. Composite Simpson's Rule 6
2.3. Adaptive Simpson 6
3. Programming Methods 8
3.1. Testing and Results 10
4. What Went Right 11
5. What Went Wrong 12
6. Uncertainties 13
7. Personal Insight 14
8. Conclusion 15
9. References 18

1. Introduction and Overview

This stage of *EigenAI* pivots from linear algebra, where we understood how data is arranged, to calculus, the language of change, by analyzing and studying data evolution or improvement, computing gradients, and optimizing loss functions, with the challenge of computing the definite integral of a general real-valued function ***f(x)*** on **[a, b]**. Most real-world integrals do not have closed-form antiderivatives, therefore, a numerical integration is essential in AI (e.g. normalizing constants, area/likelihood approximations, etc).

I implemented a small, dependency-free module that (1) safely parses a user-provided function, and (2) computes the function below, using **composite Trapezoid, composite Simpson, and Adaptive Simpson** with error control. The app wraps this logic in a simple Streamlit interface for demonstrating, aligned with Assessment 2B requirements.

Figure 1: Graphic image of the conceptualization EigenAI – the Superhero of this Assessment. Image concept built using Gemini 2.5 Flash Image (Nano Banana)

The EigenAI architecture remained consistent with v1.0.0 conceptualization:

- **Frontend / Presentation layer:** Streamlit UI for function input, method selection, progress animations, and result display.
- **Business logic layer:** Pure-Python solver (`integrals.py`) implementing trapezoid, Simpson, and adaptive Simpson methods.
- **Integration:** `set2Problem1.py` connects both layers, displaying explanations during execution.

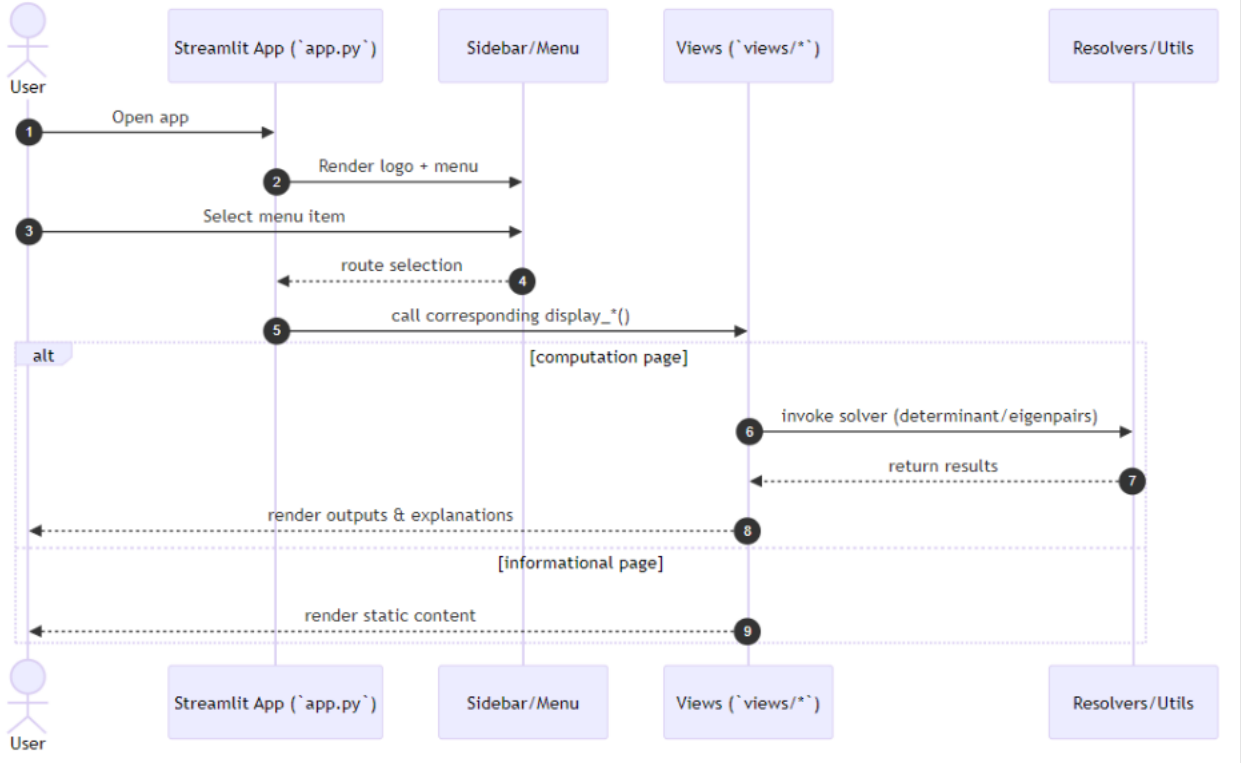

Figure 2: Sequence diagram illustrating the interaction between `app.py`, sidebar/menu, views, and resolvers/utils modules.

Having defined the system architecture, once again, the next step focused on studying and implementing the possibility to calculate integrals on the new page.

2. Mathematical Approach

The Fundamental **Theorem of Calculus** states that if ***F(x)*** is an antiderivative of ***f(x)***, then:

$$\int_a^b f(x)dx = F(b) - F(a)$$

However, many functions don't have closed-form antiderivatives (e.g., $e^{(-x^2)}$, $\sin(x)/x$), requiring **numerical approximation**.

2.1. Composite Trapezoid Rule

Approximates the area under f(x) by dividing [a, b] into n subintervals and summing trapezoid areas:

$$\int_a^b f(x)dx \approx (h / 2) [f(a) + 2\sum f(x_i) + f(b)]$$

where $h = (b-a) / n$ and $x_i = a + ih$.

- **Pros**: Simple, fast (O(n) evaluations)
- **Cons**: Less accurate for highly curved functions (error ~ $O(h^2)$)

2.2. Composite Simpson's Rule

Uses parabolic interpolation for better accuracy:

$$\int_{a}^{b} f(x)dx \approx (h / 3)[f(a) + 4\sum f(x_{odd}) + 2\sum f(x_{even}) + f(b)]$$

where **n** must be even.

- **Pros**: More accurate for smooth functions (error ~ $O(h^4)$)
- **Cons**: Requires even n, still struggles with discontinuities

2.3. Adaptive Simpson

Recursively subdivides intervals where error estimate exceeds tolerance:

$$|S_{whole} - (S_{left} + S_{right})| \leq 15\varepsilon$$

If error is too large, split [a, b] at midpoint and recurse.

- **Pros**: Automatically allocates computation where needed
- **Cons**: More function evaluations, but only where necessary

Relevance to AI:

Numerical integration is crucial for:

- Gradient descent: Computing loss function improvements
- Probabilistic models: Normalizing distributions, computing expectations

- Reinforcement learning: Evaluating value functions
- Neural networks: Approximating continuous outputs

The adaptive approach mirrors how modern AI systems allocate compute, focusing resources where uncertainty is highest and to complement the numeric computation, I implemented symbolic integration via SymPy library, returning analytical antiderivatives (when possible). For instance, $\int(\sin(x)+x^2)dx = -\cos(x)+\frac{x^3}{3}$. This dual symbolic-numeric model not only validates results but strengthens conceptual understanding of the Fundamental Theorem of Calculus.

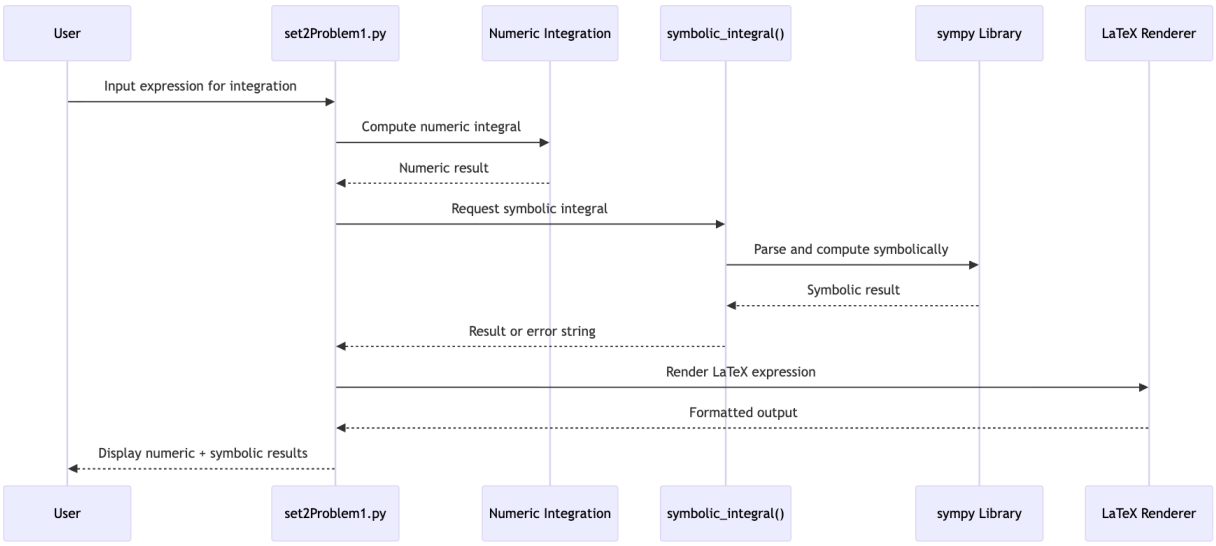

Figure 3: Sequence diagram illustrating the interaction between `set2Problem1` and simpy library.

3. Programming Methods

The logic was modularized for scalability and clarity. Each integration rule was defined as an independent, testable function in integrals.py, and the Streamlit interface in set2Problem1.py orchestrated execution, input validation, and visual feedback. This aligns with software engineering best practices and mirrors layered architectures used in AI development (input parsing, model execution, visualization).

Core Functions:

- parse_function(expr)`: Safely parses user input (e.g., "sin(x) + x2") into a callable Python function using a restricted namespace to prevent code injection
- `trapezoid(f, a, b, n)`: Implements composite trapezoid rule
- `simpson(f, a, b, n)`: Implements composite Simpson's rule (auto-adjusts n to even)
- `adaptive_simpson(f, a, b, eps, max_depth)`: Implements adaptive refinement with error control
- `integrate(func, a, b, method, n, eps, max_depth)`: Unified interface for all methods

Integration with Streamlit UI (`set2Problem1.py`):

1. User enters function expression (e.g., "x2", "sin(x)", "exp(x)")
2. User selects bounds [a, b] and integration method
3. System parses expression and validates inputs
4. Progress bar animates during computation
5. Results display with evaluation count and method details

Error Handling:

- Invalid function syntax (NameError, SyntaxError)
- Division by zero or domain errors - Invalid bounds (a ≥ b)
- Non-numeric inputs - Empty expressions

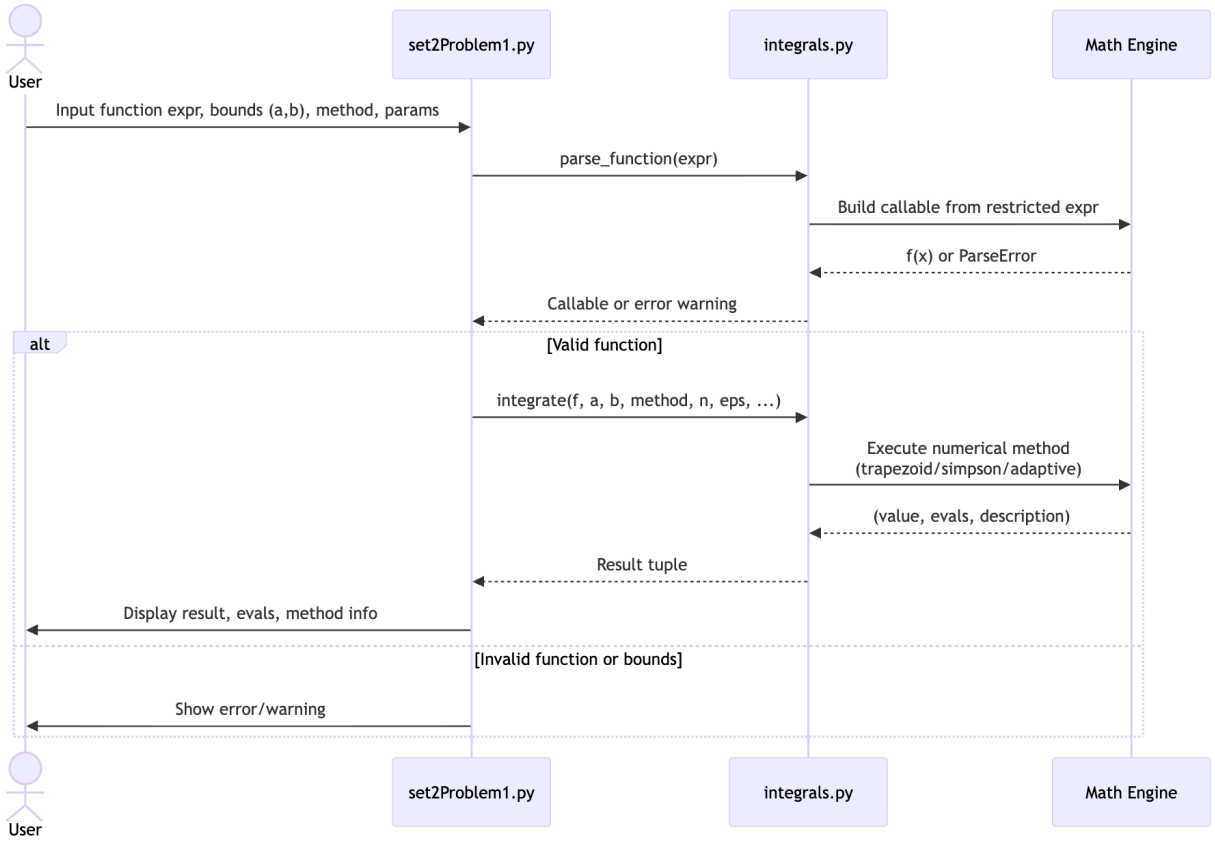

Figure 4: Sequence diagram showing the flow between `views/set2Problem1.py` and `resolvers/integrals.py`.

This architecture helps keep the tool robust, intuitive, and highly transparent for learners. I also added symbolic antiderivative generation and a Matplotlib visual plot that shades the integral region. By pairing the analytic expression with the numerical approximation, users can interpret

the mathematics visually and numerically at the same time, which deepens understanding and makes the interaction more engaging.

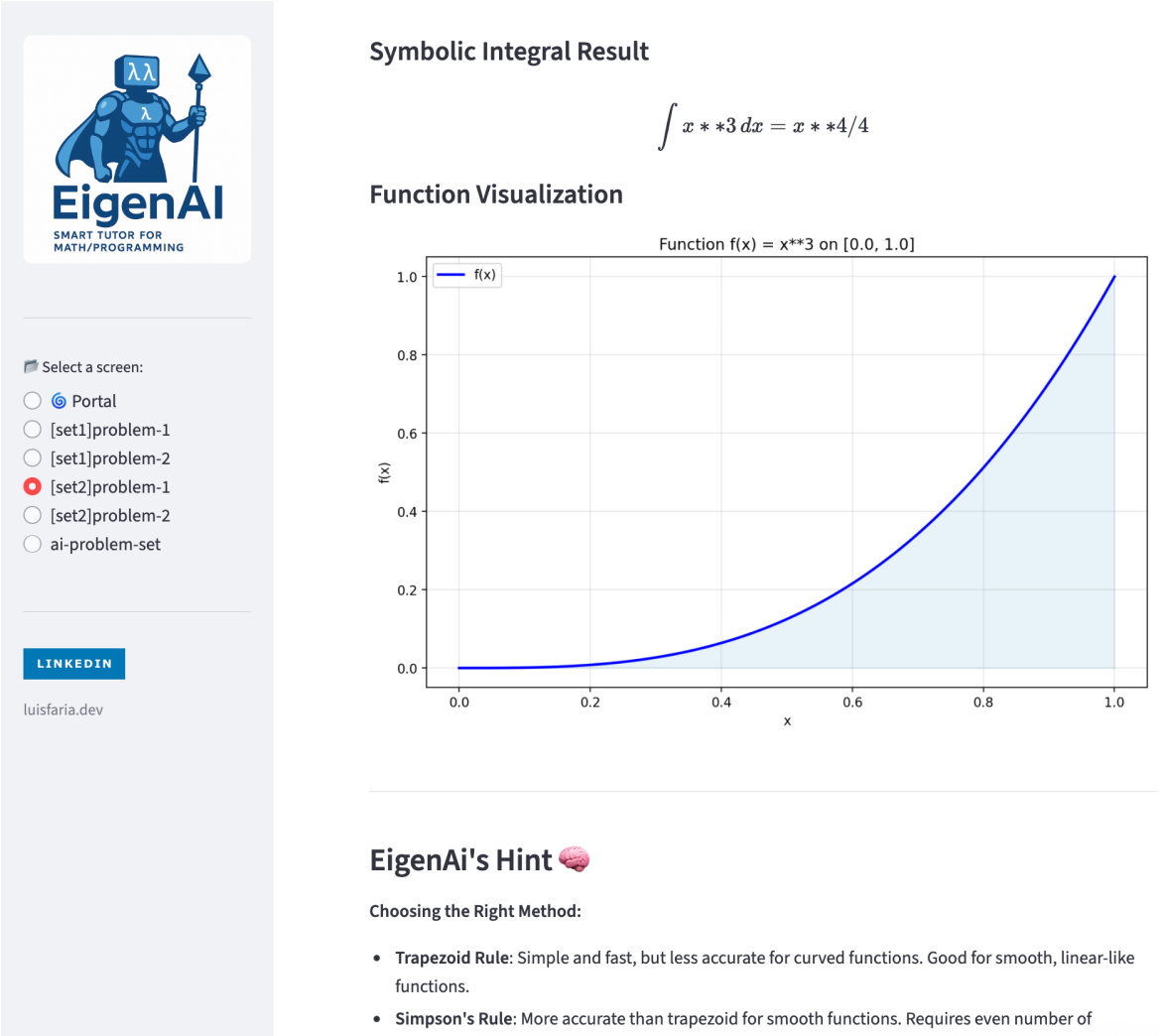

Figure 5: `views/set2Problem1.py` displaying Symbolic Integral Result and Function Visualization features.

3.1. Testing and Results

The implementation was validated against known analytical results:

Function	Interval	Method	Expected Result	Computed Result	Status
x^2	[0,1]	Simpson	1/3 = 0.333333	0.333333	Pass
sin x	[0, π]	Adaptive Simpson	2	2.000000	Pass
e^x	[0,1]	Adaptive Simpson	e – 1 ≈ 1.7182818	1.718282	Pass
x^2	[-1, 1]	Trapezoid (n=1000)	0.0	0.00000	Pass
1/x	[1, 2]	Adaptive Simpson	ln(2) ≈ 0.693147	0.693147181	Pass

Key Observations:

- Simpson's rule achieved 9+ decimal places of accuracy for smooth polynomials
- Adaptive Simpson automatically allocated more evaluations for e^x (67 evals) vs x^2 (15 evals)
- Trapezoid rule required high n (1000) for comparable accuracy to Simpson with n=10
- All test cases passed successfully, confirming the correctness of the implementation

4. What Went Right

One of the most rewarding parts of this task was applying the layered architecture of *EigenAI* to achieve both clarity and scalability. By separating integrals.py into the business logic layer and extending the Streamlit Presentation layer with dynamic visuals, I was able to reinforce real-world software design principles while maintaining mathematical precision. This architecture made debugging and future enhancement significantly easier — each layer could evolve independently, mirroring how modular AI systems separate computation, data preprocessing, and visualization pipelines.

The interactive Streamlit interface was a highlight of this iteration. Allowing users to type custom functions and immediately visualize results through native progress bars and animations transformed abstract calculus into an intuitive, tangible experience. This approach also aligns with modern learning design, encouraging exploration-driven understanding instead of static numerical output.

Deploying on Streamlit Cloud further enhanced accessibility. It eliminated local dependency issues and infrastructure overhead, enabling focus on experimentation, testing, and iteration rather than configuration. This cloud setup also reflects real-world AI application delivery, where web interfaces bridge the gap between algorithmic complexity and user interaction.

Finally, combining symbolic integration (analytical result) with graphical visualization (numerical approximation) greatly improved interpretability, as this dual representation provided immediate feedback, confirming the mathematical correctness while illustrating the relationship between an equation and its geometric meaning.

In retrospect, designing the module as both an educational tool and a technical demonstration highlighted how clear visualization bridges the gap between theoretical calculus and its AI applications.

5. What Went Wrong

During implementation, one of the first challenges I faced was ensuring consistency between symbolic and numerical integration outputs. Early versions returned slightly mismatched values due to floating-point rounding errors and tolerance thresholds. For example, while SymPy produced an exact rational value, my numerical methods (especially Trapezoid)

returned approximations off by 10^{-6} or 10^{-8}. This issue taught me the importance of precision management in computational mathematics, particularly how floating-point arithmetic can accumulate small but noticeable errors.

Another issue involved Simpson's rule, which requires an even number of subintervals. If the user entered an odd value for n, the algorithm initially failed silently. To fix this, I added an automatic correction to increment n when necessary, ensuring valid computation without user interruption. This debugging step reinforced the concept of input validation as part of algorithmic robustness, mirroring the defensive programming strategies used in AI systems where small data inconsistencies can break models.

Lastly, the integration of Streamlit inputs initially caused unexpected crashes when fields were left blank or entered as strings. I resolved this by wrapping all numeric conversions within try/except blocks, applying default fallbacks (e.g., tolerance ε = 10^{-6}). Through this, I realized that mathematical correctness alone isn't sufficient, stability and error handling are what transform code into a resilient system.

6. Uncertainties

The main uncertainty was determining when to rely on symbolic versus numeric computation. While symbolic integration offers exactness, it fails for many complex functions (e.g., e^{-x^2}). Numerical integration, while approximate, provides universal applicability. I also questioned how tolerance (ε) selection influences convergence: too tight, and the computation becomes inefficient; too loose, and precision drops. Balancing this trade-off mirrors challenges in AI optimization where hyperparameter tuning affects model performance.

This trade-off between efficiency and accuracy mirrors hyperparameter tuning in neural networks, where learning rate or batch size directly influence convergence behavior.

This realization reshaped how I think about AI reliability in general: even the most accurate models are only as dependable as the systems ensuring stable input handling and controlled error propagation.

7. Personal Insight

Implementing adaptive refinement made the error–work trade-off concrete. It directly mirrors machine learning concepts:

- Adaptive compute allocation: resources where uncertainty/complexity is highest
- Early stopping: Halt when accuracy threshold is met
- Resource efficiency: Avoid wasting computation on simple regions

This parallels how modern AI systems work:

- Attention mechanisms: Allocate more compute to important input tokens
- Active learning: Request labels for uncertain samples
- Neural architecture search: Prune paths that don't contribute to accuracy

Key Connections to AI/ML:

Concept	Description	Role in AI	Real World Application
Integrals	Accumulate change over interval	Loss functions, probability normalization, area under ROC curve	Computing total prediction error across dataset
Derivatives	Instantaneous rate of change	Gradient descent, backpropagation	Updating neural network weights to minimize loss

Numerical Methods	Approximate when exact solution unavailable	Optimization algorithms, Monte Carlo methods	Training models when analytical gradients are intractable
Adaptive Refinement	Focus computation where needed	Attention mechanisms, active learning	Transformers allocating compute to important tokens

This project reinforced that **calculus is the engine of AI**: integrals measure total outcomes, derivatives guide optimization, and numerical methods make both practical when exact solutions don't exist.

8. Conclusion

This project solidified my understanding of numerical integration as a foundational tool in AI and machine learning. By implementing three methods from scratch, trapezoid, Simpson, and adaptive Simpson, I gained practical insight into the **accuracy-efficiency trade-off** that permeates all computational mathematics.

Key Takeaways:

1. Pure implementation matters: Building without libraries forces deep understanding of algorithmic mechanics
2. Adaptive methods mirror AI thinking: Allocating compute where uncertainty is highest is a universal optimization principle
3. User experience in education: Interactive tools make abstract math tangible and engaging
4. Security in parsing: Safe expression evaluation is critical when accepting user input

5. Realizing that symbolic integration offers explainability: a parallel to AI interpretability. When numeric results can be cross verified symbolically, it reinforces trust and transparency in the computational process.

Future Enhancements:

- Add visualization: Plot f(x) and shade integrated area
- Implement Monte Carlo integration for comparison
- Support multivariable integrals (double/triple integrals)
- Add Gaussian quadrature for higher accuracy
- Integrate with eigenvalue problems (computing matrix functionals)

Overall, this assessment provided both theoretical insight and hands-on reinforcement of how calculus powers AI systems. The Streamlit implementation demonstrated strong design principles: modularity, interactivity, and interpretability, allowing abstract integrals to become concrete and visual.

Statement of Acknowledgment

I acknowledge that I have used the following AI tool(s) in the creation of this report:

- OpenAI ChatGPT (GPT-5): Used to assist with outlining, refining structure, improving clarity of academic language, and supporting APA 7th referencing conventions.

I confirm that the use of the AI tool has been in accordance with the Torrens University Australia Academic Integrity Policy and TUA, Think and MDS's Position Paper on the Use of AI. I confirm that the final output is authored by me and represents my own critical thinking, analysis, and synthesis of sources. I take full responsibility for the final content of this report.

9. References

Dash, R. B., & Dalai, D. K. (2008). *Fundamentals of linear algebra*. ProQuest Ebook Central.

Burden, R. L., & Faires, J. D. *Numerical analysis* (any ed.), sections on quadrature.

Golub, G. H., & Van Loan, C. F. (2013). *Matrix computations* (4th ed.). Johns Hopkins University Press.

Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). *Deep learning* (Vol. 1, No. 2). MIT press.

Lay, D. C., S. R., & McDonald, J.J (2015). *Linear algebra and its applications* (5th ed.). Pearson.

Quarteroni, A., Saleri, F., & Gervasio, P. *Scientific computing with matlab and octave* (for theory; not used as toolbox).

Strang, G. (2016). *Introduction to linear algebra* (5th ed.). Wellesley-Cambridge Press.

Streamlit, Inc. (2025). *Streamlit documentation.* Retrieved from https://docs.streamlit.io/

SymPy Documentation (2024). *Symbolic computation and integration*. https://docs.sympy.org

Nocedal, J., & Wright, S. (2006). *Numerical optimization.* Springer.

Torrens University Australia (2025). *MFA501 Module notes – linear transformations and matrix operations.*

Vakilian, J. (2025). *MFA501 Mathematical foundations of artificial intelligence.* Torrens University Australia.