LangGraph Mathematical Agent

Implementation Report

Overview

This project implements an advanced AI agent using LangGraph that seamlessly handles both general conversational queries and mathematical operations. The agent intelligently routes queries to either an LLM for general questions or custom mathematical functions for arithmetic operations.

Architecture

Core Components

1. **LangGraph State Management**: Uses AgentState to maintain conversation context and routing information
2. **LLM Integration**: Groq API with Llama3-8b model for general reasoning
3. **Custom Mathematical Tools**: Four specialized functions for arithmetic operations
4. **Intelligent Routing**: Pattern-based query classification system
5. **Graph Workflow**: Conditional node routing based on query type

System Flow

User Query → Query Analysis → Route Decision → Node Execution → Response

↓

[Math Pattern?]

↙ ↘

Math Node Chatbot Node

(Custom (LLM Response)

Functions)

↘ ↙

Final Response

Implementation Details

1. State Management

The AgentState TypedDict defines the conversation state:

```
class AgentState(TypedDict):
    messages: List[Any]     # Conversation history
    query_type: str         # "general" or "mathematical"
    result: str             # Final response
```

2. Mathematical Tools

Four custom tools implemented using LangChain's @tool decorator:

- **plus(a, b)**: Addition with float precision
- **subtract(a, b)**: Subtraction with proper argument handling
- **multiply(a, b)**: Multiplication operation
- **divide(a, b)**: Division with zero-division error handling

Each tool includes proper type hints and docstrings for LangGraph integration.

3. Query Classification System

The agent uses sophisticated pattern matching to identify mathematical queries:

Pattern Categories:

- **Addition**: 15 + 27, add 15 and 27, plus 15 and 27, sum of 15 and 27
- **Subtraction**: 50 - 15, subtract 15 from 50, minus 15 from 50
- **Multiplication**: 7 * 8, multiply 7 by 8, 7 times 8
- **Division**: 100 / 4, divide 100 by 4, 100 divided by 4

Implementation:

```
def _detect_math_query(self, query: str) -> tuple:
    """Detect mathematical operations using regex patterns"""
    # Returns (operation_type, num1, num2) or (None, None, None)
```

4. Graph Workflow

The LangGraph workflow consists of:

Nodes:

- **chatbot_node**: Handles general queries using LLM
- **math_node**: Processes mathematical operations

Routing Logic:

- **_route_query()**: Analyzes query and determines node destination
- **Conditional Edges**: Route based on query type detection
- **Linear Flow**: Each node connects directly to END after processing

Graph Structure:

START → [Conditional Router] → {chatbot_node, math_node} → END

5. LLM Integration

Uses Groq API with optimized parameters:

- **Model**: llama3-8b-8192 for balanced performance
- **Temperature**: 0.1 for consistent, focused responses
- **Max Tokens**: 1000 for comprehensive answers
- **System Prompt**: Configured for general assistance

Key Features

1. Seamless Query Handling

- Automatic detection of mathematical vs. general queries
- No explicit user commands needed
- Natural language processing for math operations

2. Robust Error Handling

- Division by zero protection
- Invalid input validation
- Graceful failure recovery

3. Flexible Pattern Matching

- Multiple expression formats supported
- Case-insensitive operation detection
- Natural language mathematical queries

4. Interactive Interface

- Command-line chat interface
- Real-time query processing
- Easy exit commands

Code Structure

Class Organization:

```
class MathematicalAgent:
    def __init__()          # Initialize LLM and tools
    def _detect_math_query()   # Pattern matching logic
    def _chatbot_node()       # LLM processing
    def _math_node()          # Mathematical operations
    def _route_query()        # Routing decision
    def _build_graph()        # Graph construction
    def query()               # Single query processing
    def run_interactive_session()  # Interactive mode
```

Dependencies:

- langchain_groq: LLM integration
- langgraph: Graph workflow management
- langchain_core: Tools and message handling
- re: Pattern matching
- typing: Type annotations

Testing Strategy

The implementation includes comprehensive testing:

Test Cases:

1. **General Knowledge**: "What is the capital of France?"
2. **Basic Arithmetic**: "What is 15 + 27?"
3. **Complex Topics**: "Tell me about machine learning"
4. **Division**: "Divide 100 by 4"
5. **Multiplication**: "What is 7 times 8?"
6. **Subtraction**: "Subtract 15 from 50"
7. **Mixed Queries**: General questions interspersed with math
8. **Error Cases**: Division by zero handling

Test Results Expected:

- General queries routed to LLM
- Mathematical queries processed by custom functions
- Proper error handling for edge cases
- Consistent response formatting

Usage Instructions

Setup:

1. Install required packages:
2. pip install langchain-groq langgraph langchain-core
3. Set up API key:
4. os.environ["GROQ_API_KEY"] = "your_actual_api_key"

Running the Agent:

1. **Test Mode**: Run test_agent() for automated testing
2. **Interactive Mode**: Run agent.run_interactive_session() for chat interface

Example Interactions:

You: What is the weather like today?

Agent: I don't have access to real-time weather data...

You: What is 25 + 17?

Agent: The sum of 25.0 and 17.0 is 42.0

You: Tell me about Python programming

Agent: Python is a high-level programming language...

Performance Considerations

Efficiency:

- **Pattern Matching**: O(1) regex operations for query classification
- **State Management**: Minimal memory footprint with TypedDict
- **LLM Calls**: Only when necessary (non-mathematical queries)

- **Tool Execution**: Direct function calls for mathematical operations

Scalability:

- **Stateless Design**: Each query processed independently
- **Modular Architecture**: Easy to add new mathematical operations
- **Configurable LLM**: Simple model swapping capability

Future Enhancements

Potential Improvements:

1. **Advanced Mathematics**: Trigonometry, logarithms, complex numbers
2. **Multi-step Calculations**: Chain multiple operations
3. **Unit Conversions**: Distance, weight, temperature conversions
4. **Graphing Capabilities**: Plot mathematical functions
5. **Memory Persistence**: Remember previous calculations
6. **Voice Interface**: Speech-to-text integration
7. **Web Interface**: Browser-based interaction

Technical Enhancements:

1. **Caching**: Store frequent calculations
2. **Parallel Processing**: Multiple concurrent queries
3. **Model Fine-tuning**: Specialized mathematical reasoning
4. **API Rate Limiting**: Robust production deployment

Conclusion

This implementation successfully demonstrates the power of LangGraph for building intelligent agent workflows. The combination of pattern-based routing, custom tool integration, and LLM reasoning creates a versatile assistant capable of handling diverse query types.

The modular architecture ensures maintainability and extensibility, while the comprehensive error handling provides robust operation. The agent effectively bridges the gap between general AI capabilities and specialized mathematical computation, showcasing the potential for hybrid AI systems.

Technical Specifications

- **Python Version**: 3.8+

- **Primary Framework**: LangGraph
- **LLM Provider**: Groq (Llama3-8b-8192)
- **Tool Framework**: LangChain Tools
- **State Management**: TypedDict
- **Pattern Matching**: Python re module
- **Error Handling**: Try-catch with graceful degradation

This implementation serves as a solid foundation for more complex agent architectures and demonstrates best practices for LangGraph-based AI systems.