LangGraph Mathematical Agent - Implementation Report

Overview

This project implements an intelligent agent using LangGraph that can handle both general conversational queries and mathematical operations. The agent seamlessly switches between using an LLM for general questions and custom mathematical functions for arithmetic operations.

Architecture & Components

1. State Management

```
class State(TypedDict):
    messages: Annotated[List, add_messages]
```

- Uses TypedDict to define the state schema
- Maintains conversation history through a list of messages
- Leverages LangGraph's `add_messages` reducer for state updates

2. Large Language Model Integration

- **LLM Provider**: Groq API with Llama-3.3-70b-versatile model
- **Configuration**: Temperature set to 0 for consistent responses
- **Tool Binding**: LLM is bound with mathematical tools for function calling

3. Custom Mathematical Tools

Four mathematical functions implemented as LangChain tools:

Addition Function

```
@tool
def plus(a: float, b: float) -> float:
    """Add two numbers together."""
    return a + b
```

Subtraction Function

```
@tool
def subtract(a: float, b: float) -> float:
    """Subtract the second number from the first number."""
    return a - b
```

Multiplication Function

```
@tool
def multiply(a: float, b: float) -> float:
    """Multiply two numbers together."""
    return a * b
```

Division Function

```
@tool
def divide(a: float, b: float) -> float:
    """Divide the first number by the second number."""
    if b == 0:
        raise ValueError("Cannot divide by zero!")
    return a / b
```

Key Features:

- All functions accept float parameters for decimal support
- Division includes error handling for division by zero
- Each function has descriptive docstrings for the LLM to understand their purpose

Graph Structure & Flow

1. Node Architecture

The graph consists of two main nodes:

Agent Node (`chatbot`)

- Processes incoming messages
- Decides whether to use tools or provide direct responses
- Returns updated state with LLM response

Tools Node (`tool_node`)

- Handles execution of mathematical functions
- Automatically invoked when the LLM makes tool calls
- Returns results back to the conversation flow

2. Conditional Logic

```
def should_continue(state: State):
    """Determine whether to continue to tools or end the conversation."""
    messages = state['messages']
    last_message = messages[-1]

    if last_message.tool_calls:
        return "tools"
    return END
```

Decision Flow:

- If the LLM response contains tool calls → Route to tools node
- If no tool calls → End the conversation turn
- Tools always return to the agent node for final response

3. Graph Compilation

```
workflow = StateGraph(State)
workflow.add_node("agent", chatbot)
workflow.add_node("tools", tool_node)
workflow.set_entry_point("agent")
workflow.add_conditional_edges("agent", should_continue, {"tools": "tools", END: END})
workflow.add_edge("tools", "agent")
app = workflow.compile()
```

Intelligent Query Classification

Mathematical Query Detection

The system includes a sophisticated query classifier:

```
def is_math_query(query: str) -> bool:
    math_keywords = ['add', 'plus', 'sum', 'addition', ...]
    math_symbols = ['+', '-', '*', '/', '×', '÷']
    # Logic to detect mathematical intent
```

Detection Methods:

1. **Keyword Matching**: Searches for mathematical operation terms
2. **Symbol Recognition**: Identifies mathematical symbols
3. **Pattern Matching**: Uses regex to find number patterns
4. **Context Analysis**: Combines multiple indicators for accuracy

Query Enhancement

When a mathematical query is detected, the system enhances the prompt:

enhanced_prompt = f"""The user is asking a mathematical question: "{user_input}"

Please identify the mathematical operation needed and use the appropriate tool:
- For addition: use the plus(a, b) tool
- For subtraction: use the subtract(a, b) tool
- For multiplication: use the multiply(a, b) tool
- For division: use the divide(a, b) tool

Extract the numbers from the question and call the appropriate tool."""

This ensures the LLM understands the mathematical context and uses appropriate tools.

Program Flow

1. Initialization Phase

Setup Groq API → Define Tools → Create LLM with Tools → Build Graph → Compile Agent

2. Query Processing Phase

User Input → Query Classification → Prompt Enhancement → LLM Processing → Tool Decision

3. Execution Phase

If Math Query: LLM → Tool Call → Mathematical Function → Result → Response
If General Query: LLM → Direct Response

4. Response Phase

Format Response → Display to User → Wait for Next Input

Key Features & Capabilities

1. Dual-Mode Operation

- **General Mode**: Handles conversational queries using LLM knowledge
- **Mathematical Mode**: Executes precise calculations using custom functions

2. Error Handling

- Division by zero protection
- Input validation for mathematical operations
- Graceful handling of API errors and exceptions

3. Flexible Input Processing

- Supports natural language mathematical queries
- Handles various question formats ("What is 5 plus 3?", "Calculate 10 + 5")
- Processes decimal numbers and integers

4. Interactive Interface

- Real-time conversation capability
- Clear command structure (quit/exit to end)
- Informative startup instructions

Testing & Validation

The implementation includes comprehensive testing:

Test Cases

1. **Basic Arithmetic**: Addition, subtraction, multiplication, division
2. **Error Scenarios**: Division by zero handling
3. **Decimal Support**: Operations with floating-point numbers
4. **General Queries**: Non-mathematical questions
5. **Edge Cases**: Complex mathematical expressions

Sample Test Queries

- "What is 5 plus 3?" → Mathematical processing

- "What is the capital of France?" → General LLM response
- "Divide 20 by 0" → Error handling demonstration
- "Add 2.5 and 3.7" → Decimal number support

Technical Advantages

1. Modular Design

- Separated concerns between general AI and mathematical processing
- Easy to extend with additional mathematical functions
- Clean separation of LLM and tool logic

2. Robust State Management

- Maintains conversation context
- Proper message flow through the graph
- Automatic state updates with LangGraph reducers

3. Scalable Architecture

- Easy to add new tools and capabilities
- Flexible conditional routing
- Can handle complex multi-step operations

4. Production-Ready Features

- Comprehensive error handling
- API key management
- Configurable LLM parameters
- User-friendly interface

Usage Instructions

Installation Requirements

pip install langchain-groq langgraph langchain-core

Running the Agent

1. **Interactive Mode**: Run `run_agent()` for live conversation
2. **Test Mode**: Run `test_agent()` for automated testing
3. **Custom Integration**: Use `create_agent_graph()` to integrate into other applications

Conclusion

This LangGraph implementation successfully creates an intelligent agent that seamlessly handles both general conversational queries and mathematical operations. The architecture demonstrates best practices for:

- Tool integration with LLMs
- State management in conversational AI
- Conditional routing based on query type
- Error handling and user experience design

The agent provides a solid foundation for building more complex multi-modal AI systems that can switch between different processing modes based on user intent.