# マルチモーダルフィードバック統合型知識検索強化ロボット制御システム（特願2025-067322、出願人：New York General Group, Inc.、発明者：村上 由宇）

**New York General Group**
**2025**

## 1. 発明の名称

マルチモーダルフィードバック統合型知識検索強化ロボット制御システム

## 2. 技術分野

本発明は、人工知能とロボット工学の分野に関し、特に大規模言語モデル（LLM）と検索強化生成（RAG）技術を用いて、視覚および力覚フィードバックを統合したロボット制御システムに関するものである。より具体的には、不確実かつ予測不可能な環境下において、人間の自然言語による高レベルの指示を理解し、複雑な長期的タスクを実行可能なロボットシステムの実現に関するものである。本発明は、人間の認知プロセスにおける「身体化された認知」の概念に着想を得て、言語処理能力と感覚運動フィードバックを統合することにより、ロボットの適応能力と知能を向上させるものである。

## 3. 背景技術

近年、人工知能とロボット工学の分野では、ロボットが複雑なタスクを実行する能力の向上が求められている。従来のロボットシステムは、事前にプログラムされた応答に依存し、環境の変化や不確実性に対する適応能力が限られていた。特に、家庭や医療現場などの予測不可能な環境下での複雑なタスク実行は困難であった。例えば、「コーヒーを作る」という一見単純なタスクでも、マグカップの位置の特定、引き出しの開閉、コーヒー豆のすくい取り、お湯の注入量の調整など、多くのサブタスクと環境の不確実性に対処する必要がある。

人間の知能は「身体化された認知」として理解されることが多く、注意、言語、学習、記憶、知覚などの認知プロセスは、身体が周囲の環境とどのように相互作用するかと本質的に結びついている。Lakoff & Johnson (1999)の研究によれば、人間の概念システムは身体的経験に根ざしており、抽象的な思考でさえも身体的メタファーを通じて構築されている。また、Wilson (2002)は、認知は環境との相互作用を通じて形成されるものであり、脳内の抽象的な情報処理に限定されるものではないと主張している。さらに、Shapiro (2011)は、身体の形態や運動能力が認知プロセスに影響を与えることを示している。これらの研究は、人間の知能が感覚運動プロセスに基づいているという証拠を提供しており、これは機械知能の設計に重要な示唆を与える。

従来技術では、ロボットの感覚運動能力と人工知能の発展は並行して進んできたが、これらを効果的に統合する試みは限られていた。一方では、ロボットの感覚運動能力の向上に焦点を当てた研究が進められ、視覚フィードバックを用いた物体認識や把持（Levine et al., 2018）、力覚フィードバックを用いた精密な操作（Hogan, 1985; Siciliano & Khatib, 2016）などの技術が開発されてきた。他方では、人工知能の分野では、大規模言語モデル（LLM）の発展により、自然言語理解や生成、推論能力の向上が実現されてきた（Brown et

al., 2020; Devlin et al., 2019）。しかし、これらの技術を統合し、言語理解能力と感覚運動フィードバックを組み合わせたロボットシステムの開発は、技術的課題が多く、十分に進展していなかった。

強化学習や模倣学習などのアプローチは複雑なタスクの実行に有効であることが示されている。例えば、強化学習を用いたロボットの運動制御（Schulman et al., 2017; Haarnoja et al., 2018）や、模倣学習を用いたスキル獲得（Finn et al., 2016; Ho & Ermon, 2016）などの研究が進められてきた。しかし、これらのアプローチは新しいタスクや多様なシナリオへの適応に課題がある。強化学習は大量のデータと試行錯誤を必要とし、実世界での学習には時間と資源がかかる。また、模倣学習は人間のデモンストレーションに依存するため、新しい状況への一般化が難しい場合がある。

近年、大規模言語モデル（LLM）を用いたロボット制御の研究が進んでいる。例えば、VoxPoser（Huang et al., 2023）はLLMを活用して日常的な操作タスクを実行し、Robotics Transformer（RT-2）（Brohan et al., 2023）は大規模なウェブデータとロボット学習データを活用して、訓練シナリオを超えたタスクを実行する適応能力を示している。また、Hierarchical diffusion policy（Chi et al., 2023）は、高レベルのLLM決定入力からタスク固有のモーションを強化するコンテキスト対応のモーショントラジェクトリを生成するモデル構造を導入している。

しかし、これらのアプローチには複雑なプロンプト要件、リアルタイムフィードバックの欠如、力覚フィードバックの活用不足、タスク実行を妨げる非効率なパイプラインなどの課題がある。特に、LLMを用いたロボット制御では、プロンプトの設計が複雑であり、適切な指示を与えないと望ましい結果が得られないことが多い。また、リアルタイムでの環境変化に対応するフィードバック機構が不足しており、予測不可能な状況での適応能力が限られている。さらに、多くのアプローチでは視覚フィードバックに重点が置かれ、力覚フィードバックの活用が不十分であった。これにより、視覚情報だけでは捉えられない物理的相互作用（例：液体の注入量の調整、ドアの開閉に必要な力の調整）に対する適応能力が制限されていた。

また、検索強化生成（RAG）技術のロボット工学への応用は、その潜在的可能性にもかかわらず、十分に探求されていない。RAGは、言語モデルの出力を改善するために、外部知識ベースから関連情報を検索し、生成プロセスに組み込む技術である（Lewis et al., 2020）。この技術は、言語モデルの幻覚（事実と異なる情報の生成）を減少させ、より正確で信頼性の高い出力を生成することが示されている。しかし、RAGのロボット制御への応用、特に複雑なタスク実行のための動作生成への適用は限られていた。

これらの背景から、言語理解能力と感覚運動フィードバックを効果的に統合し、予測不可能な環境下での複雑なタスク実行を可能にするロボットシステムの開発が求められている。本発明は、これらの課題に対応するために、大規模言語モデル（LLM）、検索強化生成（RAG）技術、視覚および力覚フィードバックを統合したロボット制御システムを提供するものである。

## 4. 発明が解決しようとする課題

本発明は、以下の課題を解決することを目的とする：

1. 予測不可能な環境下での複雑なタスク実行能力の向上：
　従来のロボットシステムは、事前にプログラムされた応答に依存し、環境の変化や不確実性に対する適応能力が限られていた。特に、家庭や医療現場などの予測不可能な環境下での複雑なタスク実行は困難であった。本発明は、環境の変化や不確実性に対応しながら、複雑なタスクを実行する能力を向上させることを目的とする。

2. 高レベルの抽象的指示の理解と実行：

従来のロボットシステムは、具体的な動作指示を必要とし、「コーヒーを作って」などの抽象的な指示を理解し、適切なサブタスクに分解して実行することが困難であった。本発明は、人間の自然言語による高レベルの抽象的指示を理解し、適切なサブタスクに分解して実行する能力を向上させることを目的とする。

3. 視覚および力覚フィードバックの効果的な統合：
従来のロボットシステムでは、視覚フィードバックに重点が置かれ、力覚フィードバックの活用が不十分であった。また、これらのフィードバックを統合して活用する機構が限られていた。本発明は、視覚および力覚フィードバックを効果的に統合し、環境との相互作用の精度と適応能力を向上させることを目的とする。

4. 環境の変化や不確実性への適応能力の強化：
従来のロボットシステムは、環境の変化や不確実性に対する適応能力が限られており、予期せぬ状況（例：物体の移動、障害物の出現）に対応することが困難であった。本発明は、リアルタイムでのフィードバックを取り入れ、環境の変化や不確実性に対する適応能力を強化することを目的とする。

5. 知識ベースの効率的な活用と拡張性の確保：
従来のロボットシステムでは、知識ベースの活用が限られており、新しい知識の追加や更新が困難であった。また、知識ベースの拡大に伴うパフォーマンスの低下が課題であった。本発明は、検索強化生成（RAG）技術を用いて、知識ベースを効率的に活用し、システムの拡張性を確保することを目的とする。

6. 長期的なタスクの計画と実行：
従来のロボットシステムは、短期的なタスクの実行に焦点が当てられ、複数のサブタスクから構成される長期的なタスクの計画と実行が困難であった。本発明は、サブタスク間の依存関係を考慮した長期的なタスクの計画と実行能力を向上させることを目的とする。

7. 創造的な動作生成：
従来のロボットシステムは、事前にプログラムされた動作パターンに依存し、新しい状況や要求に対応した創造的な動作生成が限られていた。本発明は、言語モデルと視覚生成モデルを組み合わせることで、新しい動作パターンを生成する能力を向上させることを目的とする。

8. 安全性と信頼性の確保：
ロボットシステムの高度化に伴い、安全性と信頼性の確保が重要な課題となっている。本発明は、安全制約を組み込み、システムの安全性と信頼性を確保することを目的とする。

9. マルチモーダル情報の統合と処理：
従来のロボットシステムでは、異なるモダリティ（視覚、力覚、言語など）からの情報を効果的に統合し、処理する能力が限られていた。本発明は、マルチモーダル情報を統合し、より豊かな環境理解と適応的な行動生成を実現することを目的とする。

10. 人間との自然なインタラクション：
従来のロボットシステムでは、人間との自然なインタラクションが限られていた。本発明は、自然言語理解と生成、非言語的コミュニケーション（ジェスチャー、表情など）の認識と生成を通じて、人間との自然なインタラクションを実現することを目的とする。

これらの課題を解決することにより、本発明は、人間と協働し、複雑なタスクを実行できる「知的なロボット」の実現に貢献することを目指す。

## 5. 課題を解決するための手段

本発明は、マルチモーダルフィードバック統合型知識検索強化ロボット制御システム（以下、本システム）を提供する。本システムは、大規模言語モデル（LLM）、検索強化生成（RAG）インフラストラクチャ、視覚システム、力覚モジュール、音声認識・合成モジュール、およびロボット制御システムを統合したものである。

本システムの主要な構成要素は以下の通りである：

1. 言語処理コンポーネント：
言語処理コンポーネントは、ユーザーのクエリと環境データを処理し、複雑なタスクを実行可能な一連のステップに分解する役割を担う。このコンポーネントは、大規模言語モデル（例：GPT-4）を使用して、自然言語の理解と生成、タスクの分解と計画、コードの生成を行う。また、検索強化生成（RAG）技術を用いて、キュレートされた知識ベースから関連する例を動的に選択し、適応させる。

言語処理コンポーネントの主要な機能は以下の通りである：
- 自然言語の理解と解釈：ユーザーの指示を理解し、その意図と目標を抽出する。
- タスクの分解と計画：複雑なタスクを実行可能なサブタスクに分解し、サブタスク間の依存関係を考慮した計画を立てる。
- コード生成：サブタスクの実行に必要なコードを生成し、ロボット制御システムに送信する。
- 知識検索と適応：RAGを用いて、知識ベースから関連する例を検索し、現在のタスクに適応させる。
- マルチモーダル情報の統合：視覚、力覚、音声などの異なるモダリティからの情報を統合し、より豊かな環境理解を実現する。
- 対話管理：ユーザーとの対話を管理し、必要に応じて質問や確認を行う。

言語処理コンポーネントは、サブタスク間の依存関係を条件付き確率として表現し、タスクの計画と実行を最適化する。例えば、P(L2A, L2B|L1)は、タスクL1の成功実行後にタスクL2AまたはL2Bに進む可能性を指定する。これにより、タスクの進行状況に応じて、次のサブタスクを動的に選択することができる。

2. 視覚システム：
視覚システムは、環境の3次元表現を生成し、物体の位置を特定する役割を担う。このシステムは、深度カメラ（例：Azure Kinect DK）を用いて環境の3次元情報を取得し、言語-視覚モジュール（例：Grounded-Segment-Anything）を通じて、言語指示に基づいて物体を識別する。

視覚システムの主要な機能は以下の通りである：
- 環境の3次元表現の生成：深度カメラを用いて、環境の3次元表現を生成する。
- 物体の検出とセグメンテーション：言語-視覚モジュールを用いて、物体を検出し、セグメンテーションを行う。
- 物体のポーズ推定：検出された物体の位置と姿勢を推定する。
- 物体の追跡：物体の移動を追跡し、リアルタイムで位置情報を更新する。
- オクルージョン処理：物体が他の物体に隠れた場合でも、その位置を推定する。
- 環境変化の検出：環境の変化（例：物体の移動、新しい物体の出現）を検出する。

視覚システムは、物体のセグメンテーションとポーズ推定を行い、ロボットの把持や操作に必要な情報を提供する。また、環境の変化（例：物体の移動）をリアルタイムで検出し、ロボットの動作に反映させる。

3. 力覚モジュール：
力覚モジュールは、ロボットのエンドエフェクタが受ける力を測定し、物体操作の精度を向上させる役割を担う。このモジュールは、多軸力覚センサー（例：ATI力覚センサー）を用いて、ロボットのエンドエフェクタが受ける力とトルクを測定する。

力覚モジュールの主要な機能は以下の通りである：
- 力とトルクの測定：多軸力覚センサーを用いて、ロボットのエンドエフェクタが受ける力とトルクを測定する。
- 力の変換と解釈：測定された力をロボットのベース座標系に変換し、その意味を解釈する。
- 力制御：物体との相互作用において、適切な力を加えるよう制御する。
- 接触検出：物体との接触を検出し、その状態を判断する。
- 物体特性の推定：力覚フィードバックから物体の重さ、硬さ、摩擦などの特性を推定する。
- 異常検出：予期せぬ力の変化を検出し、安全対策を講じる。

力覚モジュールは、力覚フィードバックを利用して、物体操作（例：液体の注入量、引き出しの開閉）の精度を向上させる。また、視覚情報が限られた状況（例：視界が遮られた場合）でも、力覚フィードバックを用いて操作を継続することができる。

4. 音声認識・合成モジュール：
音声認識・合成モジュールは、ユーザーの音声指示を認識し、システムの応答を音声で出力する役割を担う。このモジュールは、音声認識エンジンと音声合成エンジンから構成される。

音声認識・合成モジュールの主要な機能は以下の通りである：
- 音声認識：ユーザーの音声指示を認識し、テキストに変換する。
- 音声合成：システムの応答をテキストから音声に変換する。
- 話者識別：複数のユーザーの声を識別し、個人に適応した対応を行う。
- 感情認識：ユーザーの音声から感情を認識し、適切な対応を行う。
- 環境音認識：環境音（例：警告音、ドアの開閉音）を認識し、状況を理解する。
- 音声対話管理：ユーザーとの音声対話を管理し、自然な会話を実現する。

音声認識・合成モジュールは、ユーザーとの自然なインタラクションを実現し、ハンズフリーでの操作を可能にする。また、視覚情報と組み合わせることで、マルチモーダルな対話を実現する。

5. ロボット制御システム：
ロボット制御システムは、言語処理、視覚システム、力覚モジュール、音声認識・合成モジュールからのフィードバックを統合し、ロボットの動作を制御する役割を担う。このシステムは、ROS（Robot Operating System）を基盤とし、安全制約を組み込んだ制御機構を提供する。

ロボット制御システムの主要な機能は以下の通りである：
- マルチモーダルフィードバックの統合：異なるモダリティからのフィードバックを統合し、ロボットの動作に反映させる。
- 動作計画と実行：タスクの実行に必要な動作を計画し、実行する。
- 安全制約の実装：最大速度や力の制限、作業空間の境界などの安全制約を実装する。
- 障害物回避：環境内の障害物を検出し、回避する経路を計画する。
- エラー検出と回復：動作中のエラーを検出し、適切な回復処理を行う。
- 適応制御：環境の変化や不確実性に適応して、制御パラメータを調整する。

ロボット制御システムは、安全制約を組み込み、最大速度や力の制限、作業空間の境界を設定する。これにより、システムの安全性と信頼性を確保する。また、リアルタイムでのフィードバックに基づいて、動作を調整することができる。

6. 知識ベース：

知識ベースは、検証済みの低次および高次のアクションの例を含む、キュレートされたデータベースである。このデータベースは、コード例、モーションプリミティブ、タスク分解の例などを体系的に整理している。

知識ベースの主要な構成要素は以下の通りである：
- 基本的なモーションプリミティブ：直線運動、回転運動、グリッパーの開閉などの基本的な動作。
- 物体操作のモーションプリミティブ：把持、配置、持ち上げ、運搬などの物体操作に関する動作。
- 特殊なモーションプリミティブ：液体の注入、粉末のすくい取り、引き出しの開閉、描画などの特殊な動作。
- 高次のタスク例：コーヒーの作成、皿の装飾、物体の受け渡しなどの複雑なタスク。
- エラー処理と回復例：動作中のエラーに対処し、回復するための例。
- 環境適応例：環境の変化や不確実性に適応するための例。

知識ベースは、既知の不確実性を組み込んだモーション例を含み、多様なシナリオに対応することができる。また、新しい知識の追加や更新が容易な構造を持ち、システムの拡張性を確保している。

7. マルチモーダル統合モジュール：
マルチモーダル統合モジュールは、視覚、力覚、音声などの異なるモダリティからの情報を統合し、より豊かな環境理解と適応的な行動生成を実現する役割を担う。このモジュールは、各モダリティからの情報を適切に重み付けし、統合する機構を提供する。

マルチモーダル統合モジュールの主要な機能は以下の通りである：
- モダリティ間の情報統合：異なるモダリティからの情報を統合し、一貫した環境表現を生成する。
- モダリティの重み付け：タスクや状況に応じて、各モダリティの重要度を動的に調整する。
- クロスモーダル学習：異なるモダリティ間の関係を学習し、情報の相互補完を実現する。
- マルチモーダル推論：統合された情報に基づいて、環境や状況に関する推論を行う。
- 不確実性の管理：各モダリティの不確実性を考慮し、より信頼性の高い情報を優先する。
- 欠損情報の補完：一部のモダリティからの情報が欠損した場合、他のモダリティからの情報を用いて補完する。

マルチモーダル統合モジュールは、例えば、視覚情報から物体の位置を特定し、力覚情報から物体の重さを推定し、これらの情報を統合して適切な把持力と位置を決定するといった処理を行う。これにより、より精密で適応的な物体操作が可能となる。

本システムの動作フローは以下の通りである：

1. ユーザーが高レベルの指示（例：「コーヒーを作って」）を音声または文字で与える。

2. 音声認識・合成モジュールが音声指示をテキストに変換し、言語処理コンポーネントに送信する。

3. 言語処理コンポーネントが指示を解釈し、環境データ（視覚センサーからの画像）と組み合わせて、タスクを一連のサブタスク（例：「マグカップを見つける」「コーヒーをすくう」「お湯を注ぐ」）に分解する。

4. 各サブタスクに対して、RAGを用いて知識ベースから関連する例を検索する。この際、ベクトル埋め込みとコサイン類似度に基づいて、最も関連性の高い例を選択する。

5. 検索された例を基に、言語モデルが実行可能なコードを生成する。このコードには、視覚および力覚フィードバックを利用するための関数呼び出しが含まれる。

6. 生成されたコードがロボット制御システムに送信され、実行される。

7. ロボットが視覚、力覚、音声などのマルチモーダルフィードバックを利用しながら、生成されたコードに基づいてタスクを実行する。
  - 視覚フィードバックを用いて、物体の位置を特定し、把持や操作の計画を立てる。
  - 力覚フィードバックを用いて、物体との相互作用（例：液体の注入量、引き出しの開閉）を制御する。
  - 音声フィードバックを用いて、ユーザーとの対話を継続し、必要に応じて質問や確認を行う。

8. マルチモーダル統合モジュールが、異なるモダリティからの情報を統合し、より豊かな環境理解と適応的な行動生成を実現する。

9. 環境の変化や不確実性に対して、リアルタイムでフィードバックを取り入れ、動作を調整する。
  - 物体が移動した場合、視覚フィードバックを用いて新しい位置を特定し、動作を調整する。
  - 予期せぬ障害物が現れた場合、回避行動を取る。
  - 力覚フィードバックが予想と異なる場合（例：引き出しが予想よりも重い）、力の調整を行う。
  - ユーザーが新しい指示を与えた場合、タスクの計画を更新する。

10. タスクの完了後、結果をユーザーに報告し、次の指示を待つ。

本システムの特徴的な点は、以下の通りである：

1. RAGを用いた知識ベースの効率的な活用：
  本システムは、RAGを用いて知識ベースを効率的に活用することにより、システムの拡張性を確保している。従来のアプローチでは、知識ベースの内容を静的に言語モデルのコンテキストウィンドウに組み込んでいたため、知識ベースの拡大に伴いパフォーマンスが低下する問題があった。本システムでは、RAGを用いて知識ベースから関連する例を動的に選択するため、知識ベースが拡大してもパフォーマンスを維持できる。

2. マルチモーダルフィードバックの統合：
  本システムは、視覚、力覚、音声などの異なるモダリティからのフィードバックを統合することで、より豊かな環境理解と適応的な行動生成を実現している。各モダリティは異なる種類の情報を提供し、それらを適切に統合することで、単一のモダリティでは捉えられない複雑な状況に対応することができる。

3. 創造的な動作生成：
  本システムは、言語モデルと視覚生成モデルを組み合わせることで、新しい動作パターンを生成することができる。例えば、DALL-Eなどの視覚生成モデルを用いて、キーワードに基づいてシルエットを生成し、そのアウトラインを抽出して物理的な表面に描画するといった創造的なタスクを実行することができる。

4. 安全制約の組み込み：
  本システムは、安全制約を組み込むことで、システムの安全性と信頼性を確保している。最大速度や力の制限、作業空間の境界などの制約は、基本的なモーションプリミティブにコード化されているため、言語モデルのエラーがこれらの制約をオーバーライドすることはない。

5. 自然なユーザーインタラクション：
  本システムは、音声認識・合成モジュールを通じて、ユーザーとの自然なインタラクションを実現している。ユーザーは自然言語で指示を与えることができ、システムも自然言語で応答することができる。また、タスクの実行中も、ユーザーとの対話を継続し、必要に応じて質問や確認を行うことができる。

6. ハードウェア非依存の設計：

本システムは、特定のハードウェアに依存しない設計となっており、様々なロボットプラットフォームに適用可能である。また、RAGのインフラストラクチャも柔軟に選択できるため、Haystackやvebraなどのオープンソースツールを使用することも、OpenAIのRAGプロセスを使用することも可能である。

これらの特徴により、本システムは予測不可能な環境下での複雑なタスク実行能力を向上させ、高レベルの抽象的指示の理解と実行、視覚および力覚フィードバックの効果的な統合、環境の変化や不確実性への適応能力の強化、知識ベースの効率的な活用と拡張性の確保、長期的なタスクの計画と実行、創造的な動作生成、安全性と信頼性の確保、マルチモーダル情報の統合と処理、人間との自然なインタラクションといった課題を解決することができる。

## 6. 発明の効果

本発明により、以下の効果が得られる：

1. 高レベルの抽象的指示の理解と実行：
本システムは、ユーザーの自然言語による高レベルの抽象的指示（例：「コーヒーを作って」「疲れているので、ホットドリンクを作って」）を理解し、適切なサブタスクに分解して実行することができる。これにより、ユーザーは具体的な動作指示を与えることなく、目的を伝えるだけでタスクを実行させることが可能となる。例えば、「疲れていて、すぐにケーキを食べる友人が来るので、ホットドリンクを作って、皿にランダムな動物を描いてください」という複雑な指示に対して、システムは「疲れている人のためのホットドリンク」としてコーヒーを選択し、マグカップを見つけ、コーヒーをすくい、お湯を注ぎ、さらに皿に動物の絵を描くという一連のサブタスクに分解して実行することができる。

この効果は、大規模言語モデル（LLM）の自然言語理解能力と、検索強化生成（RAG）技術による知識ベースの効率的な活用によって実現される。LLMは、文脈や意図を理解し、抽象的な指示を具体的なサブタスクに分解することができる。また、RAGを用いることで、知識ベースから関連する例を検索し、現在のタスクに適応させることができる。これにより、システムは様々な抽象的指示に対応し、適切なサブタスクに分解して実行することができる。

具体的な例として、ユーザーが「朝食を準備して」と指示した場合、システムは環境データ（例：冷蔵庫の中身、調理器具の有無）を考慮して、「トーストを作る」「卵を調理する」「コーヒーを入れる」などのサブタスクに分解し、それぞれのサブタスクを適切な順序で実行することができる。また、「部屋を片付けて」という指示に対しては、「床に散らかった物を拾う」「テーブルの上を整理する」「ゴミを捨てる」などのサブタスクに分解して実行することができる。

このような高レベルの抽象的指示の理解と実行能力は、ロボットの使いやすさと実用性を大幅に向上させる。ユーザーは、専門的な知識や詳細な指示を必要とせず、自然な言葉でロボットに指示を与えることができる。これにより、高齢者や障害者、子供など、様々なユーザーがロボットを容易に使用することができるようになる。

2. 長期的なタスクの完遂：
本システムは、複数のサブタスクから構成される長期的なタスクを一貫して実行することができる。サブタスク間の依存関係を条件付き確率として表現し、タスクの計画と実行を最適化することで、複雑なタスクを効率的に完遂することが可能となる。例えば、コーヒーを作るタスクでは、マグカップが見つからない場合に引き出しを開けるという依存関係を考慮した計画を立てることができる。また、タスクの進行状況に応じて計画を調整し、予期せぬ状況（例：コーヒー豆が不足している場合）に対応することができる。

この効果は、言語処理コンポーネントのタスク分解と計画能力、およびロボット制御システムの適応制御能力によって実現される。言語処理コンポーネントは、タスクを実行可能なサブタスクに分解し、サブタス

ク間の依存関係を考慮した計画を立てる。また、ロボット制御システムは、環境の変化や不確実性に適応して、制御パラメータを調整する。これにより、システムは長期的なタスクを一貫して実行し、予期せぬ状況にも対応することができる。

具体的な例として、「夕食を準備して」という長期的なタスクを考えると、システムはこれを「食材を冷蔵庫から取り出す」「野菜を洗う」「野菜を切る」「調理する」「食器を用意する」「料理を盛り付ける」などの多数のサブタスクに分解する。各サブタスクには依存関係があり、例えば「野菜を切る」は「野菜を洗う」が完了した後に実行する必要がある。システムはこれらの依存関係を考慮して計画を立て、効率的にタスクを実行する。また、途中で食材が不足していることが判明した場合、システムは計画を調整し、代替の食材を使用するか、ユーザーに確認を取るなどの対応を行うことができる。

このような長期的なタスクの完遂能力は、ロボットの実用性と有用性を大幅に向上させる。ロボットは単純な動作の繰り返しだけでなく、複雑で多段階のタスクを一貫して実行することができるようになる。これにより、家事支援、介護支援、製造業での複雑な組立作業など、様々な応用分野でロボットの活用が進むことが期待される。

3. マルチモーダルフィードバックの統合と活用：
本システムは、視覚、力覚、音声などの異なるモダリティからのフィードバックを統合することで、より豊かな環境理解と適応的な行動生成を実現している。視覚フィードバックを用いて物体の位置や形状を認識し、力覚フィードバックを用いて物体との物理的相互作用を制御し、音声フィードバックを用いてユーザーとの対話を継続することができる。これにより、単一のモダリティでは捉えられない複雑な状況に対応することが可能となる。

この効果は、視覚システム、力覚モジュール、音声認識・合成モジュール、およびマルチモーダル統合モジュールの連携によって実現される。視覚システムは物体の検出とセグメンテーション、ポーズ推定を行い、力覚モジュールは力とトルクの測定、力制御を行い、音声認識・合成モジュールは音声認識と合成、対話管理を行う。マルチモーダル統合モジュールは、これらの情報を統合し、一貫した環境表現を生成する。これにより、システムはより豊かな環境理解と適応的な行動生成を実現することができる。

具体的な例として、液体を注ぐタスクを考えると、システムは視覚フィードバックを用いてカップの位置を特定し、力覚フィードバックを用いて注入量を制御する。カップが移動した場合、視覚フィードバックを用いて新しい位置を特定し、動作を調整する。また、注入中にカップが視界から隠れた場合でも、力覚フィードバックを用いて注入量を継続的に監視し、適切なタイミングで注入を停止することができる。さらに、ユーザーが「もう少し注いで」と音声で指示した場合、音声フィードバックを用いてその指示を理解し、追加の液体を注ぐことができる。

このようなマルチモーダルフィードバックの統合と活用能力は、ロボットの環境理解と適応能力を大幅に向上させる。ロボットは単一のセンサーの限界を超え、人間のように複数の感覚を統合して環境を理解し、適応的に行動することができるようになる。これにより、予測不可能な環境下での複雑なタスク実行が可能となり、ロボットの応用範囲が大幅に拡大する。

4. 環境の変化や不確実性への適応：
本システムは、リアルタイムでフィードバックを取り入れ、環境の変化や不確実性に対する適応能力を向上させることができる。物体が移動した場合、視覚フィードバックを用いて新しい位置を特定し、動作を調整することができる。また、予期せぬ障害物が現れた場合、回避行動を取ることができる。さらに、力覚フィードバックが予想と異なる場合（例：引き出しが予想よりも重い）、力の調整を行うことができる。これにより、予測不可能な環境下でも、タスクを継続して実行することが可能となる。

この効果は、視覚システムの環境変化の検出能力、力覚モジュールの力制御能力、およびロボット制御システムの適応制御能力によって実現される。視覚システムは環境の変化をリアルタイムで検出し、力覚モ

ジュールは物体との相互作用を適切に制御する。ロボット制御システムは、これらのフィードバックに基づいて、制御パラメータを動的に調整する。これにより、システムは環境の変化や不確実性に適応して、タスクを継続して実行することができる。

具体的な例として、テーブルの上の物体を片付けるタスクを考えると、システムは視覚フィードバックを用いて物体の位置を特定し、把持して指定された場所に配置する。途中で物体が移動した場合、システムは視覚フィードバックを用いて新しい位置を特定し、動作計画を調整する。また、物体が予想よりも重い場合、システムは力覚フィードバックを用いてその重さを検知し、把持力を増加させるか、両手で持ち上げるなどの対応を行うことができる。さらに、配置場所に障害物が現れた場合、システムは視覚フィードバックを用いてその障害物を検知し、回避するか、別の場所に配置するなどの対応を行うことができる。

このような環境の変化や不確実性への適応能力は、ロボットの実世界での有用性を大幅に向上させる。実世界は常に変化し、不確実性に満ちているため、環境の変化や不確実性に適応できるロボットは、より多くの実用的なタスクを実行することができる。これにより、家庭、医療、製造業など、様々な分野でのロボットの活用が進むことが期待される。

5. 知識ベースの効率的な活用と拡張性：
本システムは、RAGを用いて知識ベースを効率的に活用し、システムの拡張性を確保することができる。従来のアプローチでは、知識ベースの内容を静的に言語モデルのコンテキストウィンドウに組み込んでいたため、知識ベースの拡大に伴いパフォーマンスが低下する問題があった。本システムでは、RAGを用いて知識ベースから関連する例を動的に選択するため、知識ベースが拡大してもパフォーマンスを維持できる。これにより、新しい知識の追加や更新が容易となり、システムの能力を継続的に向上させることが可能となる。実験結果によれば、RAGを使用することで、GPT-4の忠実性スコアが0.74から0.88に向上し、GPT-3.5-turboの忠実性スコアが0.78から0.86に向上することが確認されている。

この効果は、言語処理コンポーネントのRAG機能によって実現される。RAGは、ユーザーのクエリと現在のタスク状態をベクトル埋め込みに変換し、知識ベースの各チャンクとのコサイン類似度を計算して、最も関連性の高いチャンクを選択する。選択されたチャンクは言語モデルの入力に追加され、より正確で関連性の高い応答を生成することができる。これにより、システムは知識ベースを効率的に活用し、拡張性を確保することができる。

具体的な例として、システムに新しいタスク（例：「パンケーキを作る」）を追加する場合を考えると、従来のアプローチでは、パンケーキ作りに関する全ての知識を言語モデルのコンテキストウィンドウに組み込む必要があり、既存の知識と合わせるとコンテキストウィンドウの制限を超える可能性があった。本システムでは、パンケーキ作りに関する知識を知識ベースに追加するだけで、RAGが必要に応じて関連する知識を検索し、言語モデルに提供する。これにより、知識ベースが拡大しても、システムのパフォーマンスを維持することができる。

このような知識ベースの効率的な活用と拡張性は、ロボットの学習能力と適応能力を大幅に向上させる。ロボットは新しい知識を継続的に追加し、様々なタスクに対応することができるようになる。これにより、ロボットの応用範囲が拡大し、より多くの実用的なタスクを実行することが可能となる。

6. 創造的な動作生成：
本システムは、言語モデルと視覚生成モデルを組み合わせることで、新しい動作パターンを生成することができる。例えば、DALL-Eなどの視覚生成モデルを用いて、キーワード（例：「ランダムな鳥」「ランダムな植物」）に基づいてシルエットを生成し、そのアウトラインを抽出して物理的な表面（例：皿）に描画することができる。これにより、ユーザーの指示に基づいて、芸術的な動作（例：描画、装飾）を実行することが可能となる。また、この技術はケーキの装飾やラテアートなどの応用にも拡張できる。

この効果は、言語処理コンポーネントの創造的な動作生成能力と、視覚生成モデルの画像生成能力によって実現される。言語処理コンポーネントは、ユーザーの指示からキーワードを抽出し、視覚生成モデルに入力する。視覚生成モデルは、キーワードに基づいて画像を生成し、その画像から描画軌跡を抽出する。ロボット制御システムは、抽出された軌跡に沿って、ロボットの動作を制御する。これにより、システムは創造的な動作を生成し、実行することができる。

具体的な例として、ユーザーが「皿に猫の絵を描いて」と指示した場合、システムは「猫」というキーワードを抽出し、DALL-Eに入力する。DALL-Eは猫のシルエット画像を生成し、システムはその画像からアウトラインを抽出する。抽出されたアウトラインは、皿の寸法に合わせて変換され、ロボットはペンを用いてそのアウトラインに沿って描画を行う。また、「ケーキにハッピーバースデーと書いて花で飾って」という指示に対しては、システムは「ハッピーバースデー」と「花」というキーワードを抽出し、テキストと花の装飾を組み合わせた描画を生成し、実行することができる。

このような創造的な動作生成能力は、ロボットの表現力と応用範囲を大幅に拡大する。ロボットは単純な反復作業だけでなく、芸術的な表現や個性的な創作活動も行うことができるようになる。これにより、エンターテイメント、教育、芸術など、新たな分野でのロボットの活用が進むことが期待される。

7. 安全性と信頼性の向上：
本システムは、安全制約を組み込むことで、システムの安全性と信頼性を向上させることができる。最大速度や力の制限、作業空間の境界などの制約は、基本的なモーションプリミティブにコード化されているため、言語モデルのエラーがこれらの制約をオーバーライドすることはない。例えば、線形速度は±0.05 m/s以内に、角速度は±60°/s以内に制限され、エンドエフェクタの力も20Nに制限される。また、エンドエフェクタは事前定義された作業空間の境界内に制限される。これにより、ロボットの動作が安全かつ信頼性の高いものとなる。

この効果は、ロボット制御システムの安全制約実装によって実現される。ロボット制御システムは、最大速度や力の制限、作業空間の境界などの安全制約を実装し、これらの制約を常に監視する。制約に違反する動作が検出された場合、システムは自動的に動作を停止または修正する。これにより、システムの安全性と信頼性を確保することができる。

具体的な例として、ロボットが物体を把持して移動する際、言語モデルが生成したコードが誤って高速な動作を指示した場合でも、ロボット制御システムの安全制約により、動作速度は安全な範囲内に制限される。また、ロボットが作業空間の境界に近づいた場合、システムは自動的に動作を減速または停止し、境界を超えないようにする。さらに、物体との相互作用において、過剰な力が加わる可能性がある場合、システムは力を制限し、物体や環境の損傷を防ぐ。

このような安全性と信頼性の向上は、ロボットの実用化と普及に不可欠である。安全で信頼性の高いロボットは、人間と共存する環境で安心して使用することができる。これにより、家庭、医療、教育など、人間と密接に関わる分野でのロボットの活用が進むことが期待される。

8. 人間との自然なインタラクション：
本システムは、音声認識・合成モジュールを通じて、ユーザーとの自然なインタラクションを実現することができる。ユーザーは自然言語で指示を与えることができ、システムも自然言語で応答することができる。また、タスクの実行中も、ユーザーとの対話を継続し、必要に応じて質問や確認を行うことができる。これにより、ユーザーとロボットの間のコミュニケーションがより円滑になり、ユーザーエクスペリエンスが向上する。

この効果は、音声認識・合成モジュールの音声対話管理能力と、言語処理コンポーネントの対話管理能力によって実現される。音声認識・合成モジュールは、ユーザーの音声指示を認識し、システムの応答を音声

で出力する。言語処理コンポーネントは、ユーザーとの対話を管理し、必要に応じて質問や確認を行う。これにより、システムはユーザーとの自然なインタラクションを実現することができる。

具体的な例として、ユーザーが「コーヒーを作って」と指示した場合、システムは「かしこまりました。コーヒーを作ります。ミルクや砂糖は必要ですか？」と応答し、ユーザーの好みを確認することができる。また、タスクの実行中に問題が発生した場合（例：コーヒー豆が不足している）、システムは「コーヒー豆が少ないようです。このまま続けますか、それとも別のドリンクを作りますか？」と質問し、ユーザーの指示を仰ぐことができる。さらに、タスクの完了後には「コーヒーができました。他に何かお手伝いできることはありますか？」と応答し、次の指示を待つことができる。

このような人間との自然なインタラクション能力は、ロボットの使いやすさと受容性を大幅に向上させる。ユーザーは特別な訓練や知識がなくても、自然な対話を通じてロボットを操作することができる。これにより、高齢者や障害者、子供など、様々なユーザーがロボットを容易に使用することができるようになる。

9. 拡張性と柔軟性：
本システムは、ハードウェアに依存しない設計となっており、様々なロボットプラットフォームに適用可能である。また、RAGのインフラストラクチャも柔軟に選択できるため、Haystackやvebraなどのオープンソースツールを使用することも、OpenAIのRAGプロセスを使用することも可能である。さらに、知識ベースの内容も柔軟に拡張できるため、新しいタスクや環境に対応するための知識を追加することができる。これにより、システムの適用範囲を継続的に拡大することが可能となる。

この効果は、システムのモジュラー設計と標準化されたインターフェースによって実現される。各コンポーネント（言語処理、視覚システム、力覚モジュールなど）は独立して動作し、標準化されたインターフェースを通じて通信する。これにより、特定のコンポーネントを異なる実装に置き換えても、システム全体が機能し続けることができる。また、知識ベースも標準化された形式で整理されているため、新しい知識を容易に追加することができる。

具体的な例として、本システムを異なるロボットプラットフォーム（例：Universal Robotsの協働ロボット、Boston Dynamicsの四足歩行ロボット）に適用する場合、ロボット制御システムのインターフェースを対象プラットフォームに合わせて調整するだけで、他のコンポーネントはそのまま使用することができる。また、RAGのインフラストラクチャを変更する場合（例：OpenAIのRAGプロセスからHaystackに変更する場合）、言語処理コンポーネントのRAG関連部分を調整するだけで、他のコンポーネントはそのまま使用することができる。さらに、新しいタスク（例：「洗濯物を畳む」）に対応するために、知識ベースに関連する例を追加するだけで、システムはそのタスクを実行することができるようになる。

このような拡張性と柔軟性は、システムの持続的な発展と幅広い応用を可能にする。システムは様々なハードウェアプラットフォームに適用でき、様々なタスクに対応することができる。これにより、システムの適用範囲が大幅に拡大し、多様な産業分野での活用が進むことが期待される。

10. 省エネルギーと環境負荷の低減：
本システムは、効率的なタスク実行により、エネルギー消費と環境負荷を低減することができる。例えば、Kinova Gen3ロボットアームの消費電力は約36W、NVIDIA RTX 2080 GPUの消費電力は約225Wであり、4分間のタスク実行における二酸化炭素排出量は約7gと推定される。これは、人間が同様のタスクを実行する場合と比較して、エネルギー効率が高いことを示している。また、タスクの最適化により、無駄な動作を減らし、さらにエネルギー効率を向上させることが可能である。

この効果は、システムの効率的なタスク計画と実行能力によって実現される。システムは、タスクを最適な順序で実行し、無駄な動作を最小限に抑えることができる。また、環境の変化や不確実性に適応することで、再試行や修正の必要性を減らし、エネルギー効率を向上させることができる。

具体的な例として、複数の物体を片付けるタスクを考えると、システムは物体の位置と目的地を考慮して、最適な順序で物体を移動させることができる。これにより、移動距離と時間を最小化し、エネルギー消費を削減することができる。また、物体の把持に失敗した場合、システムは視覚および力覚フィードバックを用いて把持位置と力を調整し、再試行の成功率を高めることができる。これにより、再試行の回数を減らし、エネルギー効率を向上させることができる。

このような省エネルギーと環境負荷の低減は、持続可能な社会の実現に貢献する。エネルギー効率の高いロボットシステムは、限られたエネルギー資源を効率的に活用し、環境への負荷を最小限に抑えることができる。これにより、環境に配慮した形でのロボット技術の普及が進むことが期待される。

これらの効果により、本発明は、人間と協働し、複雑なタスクを実行できる「知的なロボット」の実現に貢献することができる。特に、家庭や医療現場などの予測不可能な環境下での支援、製造業における柔軟な生産、サービス業における顧客対応など、様々な分野での応用が期待される。

## 7. 発明を実施するための形態

以下、本発明の実施形態について詳細に説明する。

### システム構成
本システムは、以下のハードウェアおよびソフトウェアコンポーネントから構成される：

1. ロボットアーム：
本実施形態では、7自由度のKinova Gen3ロボットアームを使用する。このロボットアームは、高い自由度と精密な制御能力を持ち、複雑な操作タスクに適している。7つの回転関節により、人間の腕に近い動作範囲と柔軟性を実現している。各関節には高精度のエンコーダーが搭載されており、関節角度を正確に測定することができる。また、各関節にはトルクセンサーが内蔵されており、外部からの力を検知することができる。これにより、力制御や衝突検知などの高度な機能を実現している。

Kinova Gen3ロボットアームの主要な仕様は以下の通りである：
- 自由度：7（肩3、肘1、手首3）
- 最大リーチ：902mm
- 可搬重量：4kg
- 繰り返し精度：±0.1mm
- 最大速度：0.5m/s
- 消費電力：36W（通常動作時）
- 重量：8.2kg
- 通信インターフェース：イーサネット、USB

ロボットアームは、ROS（Robot Operating System）を通じて制御される。ROSは、ロボット開発のためのオープンソースのソフトウェアフレームワークであり、様々なハードウェアとの連携や、複雑なロボットシステムの構築に適している。本システムでは、Kinova ROS Kortexライブラリを使用して、Kinova Gen3ロボットアームとの通信を確立する。

2. グリッパー：
ロボットアームの先端には、Robotiq 2F-140mmグリッパーを取り付ける。このグリッパーは、幅広いオブジェクトを把持することができ、最大開口幅は140mmである。また、把持力を調整することができるため、繊細なオブジェクト（例：卵、ガラス製品）から堅固なオブジェクト（例：工具、容器）まで、様々なオブ

ジェクトを適切な力で把持することができる。グリッパーの制御は、ROSを通じて行われ、開閉速度や把持力を指定することができる。

Robotiq 2F-140mmグリッパーの主要な仕様は以下の通りである：
- 最大開口幅：140mm
- 把持力：10-125N（調整可能）
- 閉鎖速度：20-150mm/s（調整可能）
- 重量：1kg
- 消費電力：5W（通常動作時）
- 通信インターフェース：USB、RS-485

グリッパーは、ROSのアクションサーバーを通じて制御される。アクションサーバーは、グリッパーの開閉、把持力の調整、状態のモニタリングなどの機能を提供する。グリッパーの状態（開閉位置、把持力など）は、50Hzの周波数で更新される。

3. 視覚センサー：
環境の3次元表現を生成するために、Azure Kinect DK深度カメラを使用する。このカメラは、640×576ピクセルの解像度で30fpsのフレームレートで動作し、深度情報とRGB画像を同時に取得することができる。深度センサーは、Time-of-Flight方式を採用しており、高精度な深度測定が可能である。また、広角レンズを搭載しているため、広い視野角（水平120度、垂直120度）を確保している。カメラの校正には、14cmのAprilTagを使用し、カメラとロボットのベースの間の位置合わせを行う。これにより、10^-6未満の精度で物体の位置検出が可能となる。

Azure Kinect DK深度カメラの主要な仕様は以下の通りである：
- RGB解像度：3840×2160（4K）、2560×1440（2K）、1920×1080（FHD）、1280×720（HD）
- 深度解像度：640×576（30fps）、512×512（30fps）、1024×1024（15fps）
- 深度モード：NFOV Unbinned（近距離、高解像度）、NFOV 2x2 Binned（近距離、低解像度）、WFOV 2x2 Binned（広角、低解像度）
- 深度範囲：0.5-5.46m（NFOV Unbinned）、0.25-2.88m（WFOV 2x2 Binned）
- 視野角：水平120度、垂直120度
- 通信インターフェース：USB 3.0

カメラからの画像データは、ROSのトピックとして公開され、視覚システムによって処理される。視覚システムは、Grounded-Segment-Anythingを用いて、物体の検出とセグメンテーションを行う。検出された物体のポーズ（位置と姿勢）は、約1/3Hzの周波数で更新される。

4. 力覚センサー：
ロボットのエンドエフェクタが受ける力を測定するために、ATI多軸力覚センサーを使用する。このセンサーは、6軸（3軸の力と3軸のトルク）の測定が可能であり、100Hzのサンプリングレートで動作する。測定精度はフルスケールの約2%であり、微細な力の変化も検出することができる。センサーの校正は、重力の影響を補正するために、外部力がない状態でセンサーがゼロを示すように調整する。これにより、エンドエフェクタに加わる外部力を正確に予測できる。校正プロセスは、一つの軸でセンサーをゼロにし、センサーを回転させ、次の軸でゼロにするという手順で行われる。

ATI多軸力覚センサーの主要な仕様は以下の通りである：
- 測定軸：6軸（Fx、Fy、Fz、Tx、Ty、Tz）
- 測定範囲：Fx, Fy: ±65N、Fz: ±200N、Tx, Ty, Tz: ±5Nm
- 分解能：Fx, Fy: 0.025N、Fz: 0.05N、Tx, Ty, Tz: 0.001Nm

- サンプリングレート：100Hz
- 通信インターフェース：USB、イーサネット

力覚センサーからのデータは、ROSのトピックとして公開され、力覚モジュールによって処理される。力覚モジュールは、測定された力をロボットのベース座標系に変換し、その意味を解釈する。変換は以下の式で行われる：

Fglobal = Tend_effector_to_robot_base × Flocal

ここで、Fglobalはロボットのベース座標系での力ベクトル、Tend_effector_to_robot_baseはエンドエフェクタの座標系からロボットのベース座標系への変換行列、Flocalはエンドエフェクタの局所座標系での力ベクトルである。

5. 音声認識・合成デバイス：
ユーザーとの音声対話を実現するために、マイクロフォンとスピーカーを搭載したデバイス（例：Amazon Echo、Google Home、または専用のマイクロフォンとスピーカー）を使用する。このデバイスは、ユーザーの音声指示を認識し、システムの応答を音声で出力する。音声認識と合成は、クラウドベースのサービス（例：Amazon Alexa、Google Assistant）または、ローカルで実行される音声認識・合成エンジン（例：Mozilla DeepSpeech、Festival）を使用して行われる。

音声認識・合成デバイスの主要な仕様は以下の通りである：
- マイクロフォン：遠距離音声認識が可能なアレイマイクロフォン
- スピーカー：クリアな音声出力が可能な高品質スピーカー
- 通信インターフェース：Wi-Fi、Bluetooth、USB
- 対応言語：日本語、英語、その他の主要言語
- 音声認識精度：90%以上（一般的な会話）
- 音声合成品質：自然な抑揚と発音

音声認識・合成デバイスからのデータは、ROSのトピックとして公開され、音声認識・合成モジュールによって処理される。音声認識・合成モジュールは、ユーザーの音声指示をテキストに変換し、言語処理コンポーネントに送信する。また、言語処理コンポーネントからのテキスト応答を音声に変換し、ユーザーに出力する。

6. コンピュータ：
システムの処理を行うために、Intel Core i9プロセッサとNVIDIA RTX 2080 GPU搭載のデスクトップコンピュータを使用する。このコンピュータは、ロボットアーム、視覚センサー、力覚センサー、音声認識・合成デバイスとイーサネットケーブルまたはUSBケーブルで接続されている。GPUは、視覚処理や言語モデルの実行に使用され、高速な処理を実現している。また、十分なメモリ（RAM 32GB以上）を搭載しており、大規模なデータ処理や複雑な計算を行うことができる。

コンピュータの主要な仕様は以下の通りである：
- プロセッサ：Intel Core i9-9900K（8コア、16スレッド、3.6GHz）
- GPU：NVIDIA RTX 2080（8GB VRAM）
- メモリ：32GB DDR4 RAM
- ストレージ：1TB NVMe SSD
- OS：Ubuntu 20.04 LTS
- ネットワーク：ギガビットイーサネット、Wi-Fi 6

コンピュータ上では、ROS、視覚処理ライブラリ（OpenCV、PCL）、機械学習フレームワーク（PyTorch、TensorFlow）、言語モデルAPI（OpenAI API）などのソフトウェアが実行される。これらのソフトウェアは、システムの各コンポーネント（言語処理、視覚システム、力覚モジュールなど）を実装するために使用される。

7. オペレーティングシステム：
システムのソフトウェア基盤として、Ubuntu 20.04を使用する。このオペレーティングシステムは、ROSとの互換性が高く、安定した動作を実現している。また、多くのロボット関連のライブラリやツールがサポートされており、開発環境として適している。

Ubuntu 20.04の主要な特徴は以下の通りである：
- カーネル：Linux 5.4
- デスクトップ環境：GNOME 3.36
- パッケージマネージャ：APT
- サポート期間：5年（2025年4月まで）
- セキュリティ：定期的なセキュリティアップデート
- ハードウェア互換性：幅広いハードウェアをサポート

Ubuntuは、ROSの公式サポートOSであり、ROSのインストールと設定が容易である。また、多くのロボット関連のライブラリやツールがUbuntu向けにパッケージ化されており、開発環境の構築が容易である。

8. ロボット制御ソフトウェア：
ロボットの制御には、ROS（Robot Operating System）を使用する。ROSは、ロボット開発のためのオープンソースのソフトウェアフレームワークであり、様々なハードウェアとの連携や、複雑なロボットシステムの構築に適している。本システムでは、Kinova ROS Kortexライブラリを使用して、Kinova Gen3ロボットアームとの通信を確立する。ROSのノード間通信機能を活用して、言語処理、視覚システム、力覚モジュールからのフィードバックを統合し、ロボットの動作を制御する。

ROSの主要な機能は以下の通りである：
- ノード間通信：トピック、サービス、アクションを通じたノード間の通信
- 分散コンピューティング：複数のコンピュータにまたがるロボットシステムの構築
- ハードウェア抽象化：様々なハードウェアを統一的に扱うためのインターフェース
- デバッグツール：ロボットシステムのデバッグを支援するツール（RViz、rqt）
- シミュレーション：Gazeboなどのシミュレータとの連携
- パッケージ管理：ロボット関連のソフトウェアパッケージの管理

ROSは、トピック、サービス、アクションという3つの通信方式を提供している。トピックは、非同期の一方向通信に使用され、センサーデータの配信などに適している。サービスは、同期のリクエスト-レスポンス通信に使用され、計算や問い合わせなどに適している。アクションは、長時間実行されるタスクの制御に使用され、ロボットの動作制御などに適している。本システムでは、これらの通信方式を適切に組み合わせて、各コンポーネント間の通信を実現している。

9. 言語モデル：
ユーザーのクエリと環境データを処理するために、GPT-4などの大規模言語モデルを使用する。GPT-4は、自然言語理解と生成の能力が高く、複雑な指示を理解し、適切なコードを生成することができる。本システムでは、GPT-4のAPIを通じて、言語処理を行う。また、RAGを実装するために、OpenAIのRAGプロセスを使用するか、Haystackやvebraなどのオープンソースツールを使用する。

GPT-4の主要な特徴は以下の通りである：

- パラメータ数：非公開（GPT-3は1750億）
- コンテキストウィンドウ：8192トークン（約32,000単語）
- 言語理解能力：複雑な指示や文脈を理解し、適切に応答
- コード生成能力：様々なプログラミング言語のコードを生成
- 推論能力：論理的推論、常識的推論、数学的推論などを実行
- 多言語対応：英語、日本語、中国語、スペイン語など多くの言語をサポート

GPT-4のAPIは、以下のパラメータを設定することができる：
- モデル：gpt-4、gpt-4-0613など
- 最大トークン数：生成するテキストの最大長
- 温度：生成の多様性を制御（0-2、低いほど決定的）
- トップP：生成のバリエーションを制御（0-1、低いほど決定的）
- 頻度ペナルティ：繰り返しを抑制（0-2、高いほど抑制）
- プレゼンスペナルティ：新しいトピックの導入を促進（0-2、高いほど促進）

本システムでは、以下の設定を使用する：
- モデル：gpt-4-0613
- 最大トークン数：8192
- 温度：0.7（創造性と一貫性のバランスを取るため）
- トップP：0.95（多様な応答を生成するため）
- 頻度ペナルティ：0.0（繰り返しを抑制するため）
- プレゼンスペナルティ：0.0（新しいトピックの導入を促進するため）

10. 視覚処理モジュール：
物体の識別とセグメンテーションには、Grounded-Segment-Anythingなどの言語-視覚モデルを使用する。このモデルは、言語指示に基づいて物体を識別し、セグメンテーションマスクを生成することができる。また、MobileSAMを使用して、セグメント化されたマスクを作成し、検出された物体を包含する3次元ボクセルを作成する。これにより、物体のポーズを抽出し、ロボットの把持や操作に使用することができる。

Grounded-Segment-Anythingの主要な特徴は以下の通りである：
- 言語-視覚モデル：言語指示に基づいて物体を識別
- セグメンテーション：物体の詳細なセグメンテーションマスクを生成
- ゼロショット転送：訓練データにない物体でも識別可能
- リアルタイム処理：高速な処理が可能（約0.3秒/フレーム）
- 高精度：COCO zero-shot転送ベンチマークで52.5のAP（平均精度）

視覚処理モジュールは、以下の処理を行う：
- 画像の前処理：ノイズ除去、コントラスト調整など
- 物体検出：言語指示に基づいて物体を検出
- セグメンテーション：検出された物体のセグメンテーションマスクを生成
- 3次元再構成：深度情報とセグメンテーションマスクを組み合わせて、物体の3次元モデルを作成
- ポーズ推定：物体の位置と姿勢を推定
- 追跡：物体の移動を追跡

視覚処理モジュールは、ROSのノードとして実装され、カメラからの画像データを処理し、検出された物体のポーズをトピックとして公開する。これにより、ロボット制御システムは物体の位置と姿勢を把握し、適切な把持や操作を行うことができる。

11. 知識ベース：
　システムの知識ベースには、検証済みの低次および高次のアクションの例を含む、キュレートされたデータベースを使用する。このデータベースには、コード例、モーションプリミティブ、タスク分解の例などが体系的に整理されている。また、既知の不確実性を組み込んだモーション例も含まれており、多様なシナリオに対応することができる。知識ベースの形式は、マークダウンファイルや構造化されたJSONなど、RAGシステムと互換性のある形式を選択する。

　知識ベースの主要な構成要素は以下の通りである：
　- 基本的なモーションプリミティブ：直線運動、回転運動、グリッパーの開閉などの基本的な動作
　- 物体操作のモーションプリミティブ：把持、配置、持ち上げ、運搬などの物体操作に関する動作
　- 特殊なモーションプリミティブ：液体の注入、粉末のすくい取り、引き出しの開閉、描画などの特殊な動作
　- 高次のタスク例：コーヒーの作成、皿の装飾、物体の受け渡しなどの複雑なタスク
　- エラー処理と回復例：動作中のエラーに対処し、回復するための例
　- 環境適応例：環境の変化や不確実性に適応するための例

　各例には、以下の情報が含まれる：
　- コード：実行可能なPythonコード
　- 入力パラメータ：コードの実行に必要なパラメータ（例：目標位置、速度、力）
　- 出力：コードの実行結果（例：成功/失敗、エラーメッセージ）
　- 前提条件：コードの実行に必要な前提条件（例：特定の物体が存在すること、ロボットが特定の初期状態にあること）
　- 後続条件：コードの実行後に期待される状態（例：物体が特定の位置に配置されていること、ロボットが特定の状態にあること）
　- 不確実性：コードの実行に関連する不確実性（例：物体の位置の不確かさ、力の制御の精度）
　- 対処方法：不確実性に対処するための方法（例：視覚フィードバックの活用、力覚フィードバックの活用）

　知識ベースは、RAGシステムによって検索され、関連する例が言語モデルに提供される。これにより、言語モデルは現在のタスクに適した応答を生成することができる。

### 言語処理コンポーネント
言語処理コンポーネントは、ユーザーのクエリと環境データを処理し、複雑なタスクを実行可能な一連のステップに分解する役割を担う。

#### 言語モデルの選択と設定
本実施形態では、GPT-4を言語モデルとして使用する。GPT-4は、自然言語理解と生成の能力が高く、複雑な指示を理解し、適切なコードを生成することができる。GPT-4のAPIを通じて、言語処理を行う。APIの設定パラメータは以下の通りである：

- モデル：gpt-4-0613
- 最大トークン数：8192
- 温度：0.7（創造性と一貫性のバランスを取るため）
- トップP：0.95（多様な応答を生成するため）
- 頻度ペナルティ：0.0（繰り返しを抑制するため）
- プレゼンスペナルティ：0.0（新しいトピックの導入を促進するため）

これらのパラメータは、タスクの性質や要求される創造性のレベルに応じて調整することができる。例えば、より決定的な応答が必要な場合は、温度とトップPを低く設定し、より創造的な応答が必要な場合は、これらのパラメータを高く設定することができる。

GPT-4への入力（プロンプト）は、以下の要素から構成される：
- システムメッセージ：モデルの役割や動作の制約を指定
- ユーザーのクエリ：ユーザーからの指示や質問
- 環境データ：視覚センサーからの画像や環境の状態
- 知識ベースからの関連例：RAGによって検索された関連例

システムメッセージの例：
```
あなたはロボット制御システムのアシスタントです。ユーザーの指示と環境データに基づいて、ロボットが実行すべきタスクを分析し、適切なPythonコードを生成してください。コードは、ROS環境で実行され、視覚および力覚フィードバックを利用します。安全性を最優先し、最大速度や力の制限、作業空間の境界などの制約を遵守してください。
```

このようなプロンプト設計により、GPT-4は適切な役割を理解し、安全で効果的なコードを生成することができる。

#### タスク分解のプロセス
言語処理コンポーネントは、ユーザーの高レベルの指示（例：「コーヒーを作って」）を受け取り、環境データ（視覚センサーからの画像）と組み合わせて、タスクを一連のサブタスクに分解する。このプロセスは以下の手順で行われる：

1. ユーザーの指示を解析し、主要な目標（例：「コーヒーを作る」）を特定する。
2. 環境データを分析し、利用可能なリソース（例：マグカップ、コーヒー豆、ケトル）を特定する。
3. 目標を達成するために必要なサブタスクを特定する。例えば、「コーヒーを作る」というタスクは、以下のサブタスクに分解される：
   - マグカップを見つける
   - コーヒー豆を見つける
   - ケトルを見つける
   - マグカップを適切な位置に置く
   - コーヒー豆をすくう
   - コーヒー豆をマグカップに入れる
   - ケトルからお湯を注ぐ
4. サブタスク間の依存関係を特定する。例えば、「コーヒー豆をマグカップに入れる」は「マグカップを適切な位置に置く」に依存する。
5. 各サブタスクに対して、実行条件と成功基準を定義する。例えば、「マグカップを見つける」というサブタスクの成功基準は、「マグカップの位置が特定され、ロボットがマグカップを把持できる状態になっていること」である。

このタスク分解のプロセスは、条件付き確率の形式で表現される。例えば、P(L2A, L2B|L1)は、タスクL1の成功実行後にタスクL2AまたはL2Bに進む可能性を指定する。これにより、タスクの進行状況に応じて、次のサブタスクを動的に選択することができる。

具体的には、以下のような条件付き確率が定義される：
- P(マグカップを見つける|初期状態) = 0.9：初期状態からマグカップを見つける確率は90%

- P(引き出しを開ける|マグカップを見つけられない) = 0.8：マグカップを見つけられない場合、引き出しを開ける確率は80%
- P(マグカップを見つける|引き出しを開ける) = 0.7：引き出しを開けた後、マグカップを見つける確率は70%

これらの条件付き確率は、知識ベースの例や過去の経験に基づいて設定される。また、タスクの実行中に得られるフィードバックに基づいて、動的に更新することもできる。

#### 検索強化生成（RAG）の実装
本システムでは、RAGを用いて知識ベースから関連する例を検索し、言語モデルの出力を改善する。RAGの実装には、以下の手順を採用する：

1. クエリの埋め込み：ユーザーのクエリと現在のタスク状態を、ベクトル埋め込みに変換する。埋め込みモデルとしては、OpenAIのtext-embedding-ada-002や、HuggingFaceのSentence-BERTなどを使用する。
2. チャンキング：知識ベースを意味のある単位（チャンク）に分割する。チャンクのサイズは、文脈の保持と検索効率のバランスを考慮して決定する。
3. チャンクの埋め込み：各チャンクをベクトル埋め込みに変換する。
4. 類似度計算：クエリの埋め込みと各チャンクの埋め込みの間のコサイン類似度を計算する。
5. 上位チャンクの選択：類似度に基づいて、上位k個のチャンクを選択する。kの値は、タスクの複雑さと知識ベースの規模に応じて調整する。
6. コンテキスト拡張：選択されたチャンクを、言語モデルへの入力に追加する。
7. 応答生成：拡張されたコンテキストを用いて、言語モデルが応答（コード）を生成する。

本実施形態では、OpenAIのRAGプロセスを使用し、キュレートされた知識ベースをマークダウンファイルとして整理する。ただし、本システムのフレームワークでは、Haystackやvebraなどのツールを使用した他のRAGアプローチも利用可能である。これらのツールを使用する場合、以下のコンポーネントを選択することができる：

- ドキュメントストア：知識ベースの保存と組織化の方法（例：マークダウンファイル、Elasticsearch）
- エンベッダー：テキストをベクトル表現に変換するモデル（例：OpenAI Embeddings、Sentence-BERT）
- リトリーバー：クエリに関連するドキュメントを検索する方法（例：ベクトル検索、キーワード検索、ハイブリッド検索）
- チャンキング技術：ドキュメントを意味のある単位に分割する方法（例：固定サイズ、段落ベース、セマンティックチャンキング）
- 言語モデル：最終的な応答を生成するモデル（例：GPT-4、GPT-3.5-turbo、Zephyr-7B-beta）

RAGの実装例として、以下のようなPythonコードが考えられる：

```python
from openai import OpenAI
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# OpenAI APIクライアントの初期化
client = OpenAI(api_key="your-api-key")

# 知識ベースのチャンクとその埋め込みを保存する辞書
knowledge_base = {}

# 知識ベースの初期化（マークダウンファイルからチャンクを抽出し、埋め込みを計算）
def initialize_knowledge_base(markdown_file):
    with open(markdown_file, 'r') as f:
        content = f.read()
```

```
    # マークダウンを意味のあるチャンクに分割
    chunks = split_into_chunks(content)

    # 各チャンクの埋め込みを計算
    for chunk in chunks:
        embedding = get_embedding(chunk)
        knowledge_base[chunk] = embedding

# テキストの埋め込みを取得
def get_embedding(text):
    response = client.embeddings.create(
        model="text-embedding-ada-002",
        input=text
    )
    return response.data[0].embedding

# クエリに関連するチャンクを検索
def retrieve_relevant_chunks(query, k=5):
    # クエリの埋め込みを取得
    query_embedding = get_embedding(query)

    # 各チャンクとクエリの類似度を計算
    similarities = {}
    for chunk, chunk_embedding in knowledge_base.items():
        similarity = cosine_similarity([query_embedding], [chunk_embedding])[0][0]
        similarities[chunk] = similarity

    # 類似度に基づいて上位k個のチャンクを選択
    sorted_chunks = sorted(similarities.items(), key=lambda x: x[1], reverse=True)
    top_k_chunks = [chunk for chunk, _ in sorted_chunks[:k]]

    return top_k_chunks

# RAGを用いた応答生成
def generate_response_with_rag(query, system_message, k=5):
    # 関連するチャンクを検索
    relevant_chunks = retrieve_relevant_chunks(query, k)

    # コンテキストを拡張
    context = "\n\n".join(relevant_chunks)

    # 言語モデルに入力を送信
    response = client.chat.completions.create(
        model="gpt-4-0613",
        messages=[
            {"role": "system", "content": system_message},
            {"role": "user", "content": f"Query: {query}\n\nRelevant Information:\n{context}"}
        ],
        max_tokens=8192,
        temperature=0.7,
        top_p=0.95,
        frequency_penalty=0.0,
        presence_penalty=0.0
    )

    return response.choices[0].message.content
```

このようなRAG実装により、システムは知識ベースから関連する例を効率的に検索し、言語モデルの出力を改善することができる。

#### コード生成と実行
言語処理コンポーネントは、RAGを通じて取得した関連例を基に、実行可能なコードを生成する。生成されたコードは、ROSと互換性のあるPython（v.3.8）で記述され、以下の要素を含む：

1. 必要なライブラリとモジュールのインポート
2. ROSノードの初期化
3. 視覚および力覚フィードバックを取得するための関数呼び出し
4. タスク実行のためのメイン関数
5. エラー処理とリカバリーメカニズム
6. 安全制約の実装

生成されたコードの例：

```python
#!/usr/bin/env python3

# 必要なライブラリとモジュールのインポート
import rospy
import numpy as np
from std_msgs.msg import String
from geometry_msgs.msg import Pose, Twist
from sensor_msgs.msg import Image, JointState
from kortex_driver.srv import *
from kortex_driver.msg import *
from cv_bridge import CvBridge
import tf2_ros
import tf2_geometry_msgs
from tf.transformations import quaternion_from_euler

# ROSノードの初期化
rospy.init_node('coffee_making_node', anonymous=True)

# グローバル変数
robot_name = "my_gen3"
bridge = CvBridge()
tf_buffer = tf2_ros.Buffer()
tf_listener = tf2_ros.TransformListener(tf_buffer)

# サービスクライアントの設定
base_service_name = '/' + robot_name + '/base/base_service'
base_client = rospy.ServiceProxy(base_service_name, Base_ClearFaults)

# 視覚フィードバックを取得する関数
def get_object_pose(object_name):
    try:
        # オブジェクトの検出結果を取得
        object_pose = rospy.wait_for_message('/vision/object_poses', Pose, timeout=5.0)
        return object_pose
    except rospy.ROSException:
        rospy.logerr(f"Failed to get pose for object: {object_name}")
        return None

# 力覚フィードバックを取得する関数
def get_force_feedback():
    try:
        # 力覚センサーの読み取り値を取得
        force_data = rospy.wait_for_message('/force_torque_sensor/wrench', WrenchStamped, timeout=5.0)
```

```
        return force_data.wrench
    except rospy.ROSException:
        rospy.logerr("Failed to get force feedback")
        return None

# グリッパーを制御する関数
def control_gripper(position, speed=0.1, force=0.8):
    try:
        # グリッパーの制御サービスを呼び出す
        service_name = '/' + robot_name + '/base/send_gripper_command'
        gripper_client = rospy.ServiceProxy(service_name, SendGripperCommand)

        # グリッパーコマンドの設定
        gripper_command = SendGripperCommandRequest()
        gripper_command.input.gripper.finger[0].value = position
        gripper_command.input.mode = GripperMode.GRIPPER_POSITION

        # グリッパーコマンドの送信
        gripper_client(gripper_command)

        return True
    except rospy.ServiceException as e:
        rospy.logerr(f"Failed to control gripper: {e}")
        return False

# ロボットを移動する関数
def move_robot(target_pose, speed=0.1):
    try:
        # カートesian速度コマンドのパブリッシャーを設定
        velocity_pub = rospy.Publisher('/' + robot_name + '/in/cartesian_velocity', TwistCommand, queue_size=10)

        # 現在の位置を取得
        current_pose = rospy.wait_for_message('/' + robot_name + '/base/tool_pose', Pose, timeout=5.0)

        # 目標位置への方向ベクトルを計算
        direction = np.array([
            target_pose.position.x - current_pose.position.x,
            target_pose.position.y - current_pose.position.y,
            target_pose.position.z - current_pose.position.z
        ])

        # 方向ベクトルの正規化
        distance = np.linalg.norm(direction)
        if distance > 0:
            direction = direction / distance

        # 速度コマンドの設定
        twist_cmd = TwistCommand()
        twist_cmd.reference_frame = CartesianReferenceFrame.CARTESIAN_REFERENCE_FRAME_BASE
        twist_cmd.twist.linear_x = direction[0] * speed
        twist_cmd.twist.linear_y = direction[1] * speed
        twist_cmd.twist.linear_z = direction[2] * speed

        # 目標位置に到達するまで速度コマンドを送信
        rate = rospy.Rate(10)  # 10Hz
        while distance > 0.01:
            # 現在の位置を取得
            current_pose = rospy.wait_for_message('/' + robot_name + '/base/tool_pose', Pose, timeout=5.0)

            # 目標位置への方向ベクトルを再計算
```

```
            direction = np.array([
                target_pose.position.x - current_pose.position.x,
                target_pose.position.y - current_pose.position.y,
                target_pose.position.z - current_pose.position.z
            ])

            # 方向ベクトルの正規化
            distance = np.linalg.norm(direction)
            if distance > 0:
                direction = direction / distance

            # 速度コマンドの更新
            twist_cmd.twist.linear_x = direction[0] * min(speed, distance)
            twist_cmd.twist.linear_y = direction[1] * min(speed, distance)
            twist_cmd.twist.linear_z = direction[2] * min(speed, distance)

            # 速度コマンドの送信
            velocity_pub.publish(twist_cmd)

            rate.sleep()

        # 停止コマンドの送信
        twist_cmd.twist.linear_x = 0.0
        twist_cmd.twist.linear_y = 0.0
        twist_cmd.twist.linear_z = 0.0
        velocity_pub.publish(twist_cmd)

        return True
    except rospy.ROSException as e:
        rospy.logerr(f"Failed to move robot: {e}")
        return False

# マグカップを見つける関数
def find_mug():
    rospy.loginfo("Finding mug...")

    # マグカップの位置を取得
    mug_pose = get_object_pose("mug")

    if mug_pose is None:
        rospy.logwarn("Mug not found in the current view. Searching in drawers...")
        # 引き出しを開ける処理を呼び出す
        open_drawer()
        # 再度マグカップの位置を取得
        mug_pose = get_object_pose("mug")

        if mug_pose is None:
            rospy.logerr("Failed to find mug even after opening drawer.")
            return None

    rospy.loginfo(f"Mug found at position: {mug_pose.position.x}, {mug_pose.position.y}, {mug_pose.position.z}")
    return mug_pose

# 引き出しを開ける関数
def open_drawer():
    rospy.loginfo("Opening drawer...")

    # 引き出しのハンドルの位置を取得
    handle_pose = get_object_pose("drawer_handle")

    if handle_pose is None:
        rospy.logerr("Failed to find drawer handle.")
```

```
    return False

# ハンドルに近づく
approach_pose = Pose()
approach_pose.position.x = handle_pose.position.x - 0.1
approach_pose.position.y = handle_pose.position.y
approach_pose.position.z = handle_pose.position.z
approach_pose.orientation = handle_pose.orientation

move_robot(approach_pose)

# ハンドルを把持
move_robot(handle_pose)
control_gripper(0.5) # グリッパーを閉じる

# 力覚フィードバックを取得
initial_force = get_force_feedback()

if initial_force is None:
    rospy.logerr("Failed to get force feedback.")
    control_gripper(1.0) # グリッパーを開く
    return False

# 引き出しを引く
pull_pose = Pose()
pull_pose.position.x = handle_pose.position.x - 0.3
pull_pose.position.y = handle_pose.position.y
pull_pose.position.z = handle_pose.position.z
pull_pose.orientation = handle_pose.orientation

# 力覚フィードバックを監視しながら引く
try:
    velocity_pub = rospy.Publisher('/' + robot_name + '/in/cartesian_velocity', TwistCommand, queue_size=10)

    twist_cmd = TwistCommand()
    twist_cmd.reference_frame = CartesianReferenceFrame.CARTESIAN_REFERENCE_FRAME_BASE
    twist_cmd.twist.linear_x = -0.05 # ゆっくりと引く


    rate = rospy.Rate(10) # 10Hz

    # 引き出しが開くまで、または力が閾値を超えるまで引く
    max_force = 10.0 # 最大力（N）
    max_duration = rospy.Duration(5.0) # 最大引く時間
    start_time = rospy.Time.now()

    while (rospy.Time.now() - start_time) < max_duration:
        # 力覚フィードバックを取得
        force = get_force_feedback()

        if force is None:
            break

        # 力が閾値を超えた場合、引くのを止める
        if abs(force.force.x) > max_force:
            rospy.logwarn("Force threshold exceeded. Stopping pull.")
            break

        # 速度コマンドの送信
        velocity_pub.publish(twist_cmd)
```

```
            rate.sleep()

        # 停止コマンドの送信
        twist_cmd.twist.linear_x = 0.0
        velocity_pub.publish(twist_cmd)

    except rospy.ROSException as e:
        rospy.logerr(f"Failed to pull drawer: {e}")
        control_gripper(1.0) # グリッパーを開く
        return False

    # グリッパーを開く
    control_gripper(1.0)

    rospy.loginfo("Drawer opened successfully.")
    return True

# コーヒーをすくう関数
def scoop_coffee():
    rospy.loginfo("Scooping coffee...")

    # コーヒー容器の位置を取得
    coffee_container_pose = get_object_pose("coffee_container")

    if coffee_container_pose is None:
        rospy.logerr("Failed to find coffee container.")
        return False

    # スプーンの位置を取得
    spoon_pose = get_object_pose("spoon")

    if spoon_pose is None:
        rospy.logerr("Failed to find spoon.")
        return False

    # スプーンを把持
    move_robot(spoon_pose)
    control_gripper(0.5) # グリッパーを閉じる

    # スプーンを持ち上げる
    lift_pose = Pose()
    lift_pose.position.x = spoon_pose.position.x
    lift_pose.position.y = spoon_pose.position.y
    lift_pose.position.z = spoon_pose.position.z + 0.1
    lift_pose.orientation = spoon_pose.orientation

    move_robot(lift_pose)

    # コーヒー容器の上に移動
    above_container_pose = Pose()
    above_container_pose.position.x = coffee_container_pose.position.x
    above_container_pose.position.y = coffee_container_pose.position.y
    above_container_pose.position.z = coffee_container_pose.position.z + 0.1
    above_container_pose.orientation = spoon_pose.orientation

    move_robot(above_container_pose)

    # スプーンをコーヒー容器に挿入
    scoop_pose = Pose()
    scoop_pose.position.x = coffee_container_pose.position.x
    scoop_pose.position.y = coffee_container_pose.position.y
```

```
    scoop_pose.position.z = coffee_container_pose.position.z + 0.02
    scoop_pose.orientation = spoon_pose.orientation

    move_robot(scoop_pose)

    # スプーンを水平に動かしてコーヒーをすくう
    scoop_motion_pose = Pose()
    scoop_motion_pose.position.x = coffee_container_pose.position.x + 0.05
    scoop_motion_pose.position.y = coffee_container_pose.position.y
    scoop_motion_pose.position.z = coffee_container_pose.position.z + 0.02
    scoop_motion_pose.orientation = spoon_pose.orientation

    move_robot(scoop_motion_pose)

    # スプーンを持ち上げる
    lift_after_scoop_pose = Pose()
    lift_after_scoop_pose.position.x = scoop_motion_pose.position.x
    lift_after_scoop_pose.position.y = scoop_motion_pose.position.y
    lift_after_scoop_pose.position.z = scoop_motion_pose.position.z + 0.1
    lift_after_scoop_pose.orientation = spoon_pose.orientation

    move_robot(lift_after_scoop_pose)

    rospy.loginfo("Coffee scooped successfully.")
    return True

# お湯を注ぐ関数
def pour_water(mug_pose):
    rospy.loginfo("Pouring water...")

    # ケトルの位置を取得
    kettle_pose = get_object_pose("kettle")

    if kettle_pose is None:
        rospy.logerr("Failed to find kettle.")
        return False

    # ケトルを把持
    move_robot(kettle_pose)
    control_gripper(0.5) # グリッパーを閉じる

    # ケトルを持ち上げる
    lift_pose = Pose()
    lift_pose.position.x = kettle_pose.position.x
    lift_pose.position.y = kettle_pose.position.y
    lift_pose.position.z = kettle_pose.position.z + 0.1
    lift_pose.orientation = kettle_pose.orientation

    move_robot(lift_pose)

    # マグカップの上に移動
    above_mug_pose = Pose()
    above_mug_pose.position.x = mug_pose.position.x
    above_mug_pose.position.y = mug_pose.position.y
    above_mug_pose.position.z = mug_pose.position.z + 0.2
    above_mug_pose.orientation = kettle_pose.orientation

    move_robot(above_mug_pose)

    # ケトルを傾ける
    pour_orientation = quaternion_from_euler(0, 0.5, 0)  # 約30度傾ける
    pour_pose = Pose()
```

```
    pour_pose.position = above_mug_pose.position
    pour_pose.orientation.x = pour_orientation[0]
    pour_pose.orientation.y = pour_orientation[1]
    pour_pose.orientation.z = pour_orientation[2]
    pour_pose.orientation.w = pour_orientation[3]

    # 力覚フィードバックを監視しながら注ぐ
    try:
        # 初期の力を取得
        initial_force = get_force_feedback()

        if initial_force is None:
            rospy.logerr("Failed to get initial force feedback.")
            return False

        # ケトルを傾ける
        move_robot(pour_pose)

        # 一定時間待機（お湯が注がれる）
        rospy.sleep(3.0)

        # 現在の力を取得
        current_force = get_force_feedback()

        if current_force is None:
            rospy.logerr("Failed to get current force feedback.")
            return False

        # 力の変化から注がれた量を推定
        force_change = current_force.force.z - initial_force.force.z
        poured_amount = force_change / 9.8  # 力の変化を質量に変換（概算）


        rospy.loginfo(f"Estimated poured amount: {poured_amount} kg")

        # ケトルを元の向きに戻す
        move_robot(above_mug_pose)

    except rospy.ROSException as e:
        rospy.logerr(f"Failed to pour water: {e}")
        return False

    # ケトルを元の位置に戻す
    move_robot(lift_pose)
    move_robot(kettle_pose)

    # グリッパーを開く
    control_gripper(1.0)

    rospy.loginfo("Water poured successfully.")
    return True

# メイン関数
def make_coffee():
    rospy.loginfo("Starting coffee making process...")

    try:
        # マグカップを見つける
        mug_pose = find_mug()

        if mug_pose is None:
            rospy.logerr("Failed to find mug. Aborting coffee making.")
```

```
            return False

        # コーヒーをすくう
        if not scoop_coffee():
            rospy.logerr("Failed to scoop coffee. Aborting coffee making.")
            return False

        # マグカップにコーヒーを入れる
        # （この部分は省略されていますが、スプーンをマグカップの上に移動し、傾けてコーヒーを入れる処理が必要です）

        # お湯を注ぐ
        if not pour_water(mug_pose):
            rospy.logerr("Failed to pour water. Aborting coffee making.")
            return False

        rospy.loginfo("Coffee made successfully!")
        return True

    except Exception as e:
        rospy.logerr(f"An error occurred during coffee making: {e}")
        return False

# メイン実行
if __name__ == '__main__':
    try:
        # ロボットの初期化
        rospy.wait_for_service(base_service_name, timeout=10)
        base_client()

        # コーヒーを作る
        make_coffee()

    except rospy.ROSInterruptException:
        pass
```

このようなコードは、RAGを通じて取得した関連例を基に生成され、ROSと互換性のある形式で記述されている。コードには、視覚および力覚フィードバックを利用するための関数呼び出しが含まれており、環境の変化や不確実性に対応することができる。また、安全制約（速度制限、力の制限など）も組み込まれている。

生成されたコードは、セキュアな環境で実行される。この環境は、事前定義された関数のみにアクセスできるように制限されており、システムの安全性を確保している。また、コードの実行前に、構文チェックとセキュリティチェックを行い、潜在的な問題を検出する。

### 視覚システム
視覚システムは、環境の3次元表現を生成し、物体の位置を特定する役割を担う。

#### カメラのセットアップと校正
本実施形態では、Azure Kinect DK深度カメラを使用する。カメラの設定は以下の通りである：

- 解像度：640×576ピクセル
- フレームレート：30fps
- 深度モード：NFOV Unbinned（近距離、高解像度）
- カラーフォーマット：BGRA

- 露出時間：自動調整

カメラの校正には、14cmのAprilTagを使用する。AprilTagは、既知のサイズと形状を持つマーカーであり、カメラとロボットのベースの間の位置合わせに使用される。校正プロセスは以下の手順で行われる：

1. AprilTagをロボットの作業空間内の既知の位置に配置する。
2. カメラでAprilTagを撮影し、その位置とサイズを検出する。
3. 検出された位置とサイズから、カメラの内部パラメータ（焦点距離、主点）と外部パラメータ（位置、姿勢）を計算する。
4. 計算されたパラメータを用いて、カメラ座標系からロボットのベース座標系への変換行列を求める。

この校正により、10^-6未満の精度で物体の位置検出が可能となる。変換行列は以下の式で表される：

PR = TAR × (TCA × PC)

ここで、PCはカメラ座標系での点、TCAはカメラ座標系からAprilTag座標系への変換行列、TARはAprilTag座標系からロボットのベース座標系への変換行列、PRはロボットのベース座標系での点である。

校正プロセスは、ROSのtf2ライブラリを用いて実装される。tf2は、異なる座標系間の変換を管理するためのライブラリであり、時間的に変化する変換も扱うことができる。

#### 物体検出とセグメンテーション
物体の検出とセグメンテーションには、Grounded-Segment-Anythingを使用する。このモデルは、言語指示に基づいて物体を識別し、セグメンテーションマスクを生成することができる。処理フローは以下の通りである：

1. カメラから取得した画像を、Grounded-Segment-Anythingモデルに入力する。
2. モデルは、言語指示（例：「白いマグカップ」「黒いケトル」）に基づいて、対応する物体を検出する。
3. 検出された物体に対して、バウンディングボックスを生成する。
4. MobileSAMを使用して、セグメント化されたマスクを作成する。
5. 深度情報と組み合わせて、検出された物体を包含する3次元ボクセルを作成する。
6. ボクセルから物体のポーズ（位置と姿勢）を抽出する。

このプロセスにより、物体の3次元位置と姿勢を特定し、ロボットの把持や操作に使用することができる。物体の検出精度は、物体の種類や環境条件によって異なるが、実験結果によれば、白いマグカップの検出精度は約70%、黒いケトルの検出精度は約79%、手の検出精度は約53%であった。

Grounded-Segment-Anythingの実装は、以下のようなPythonコードで行われる：

```python
import torch
import numpy as np
import cv2
from PIL import Image
from groundingdino.util.inference import load_model, load_image, predict
from segment_anything import sam_model_registry, SamPredictor
import rospy
from sensor_msgs.msg import Image as RosImage
from geometry_msgs.msg import Pose
from cv_bridge import CvBridge

class ObjectDetector:
    def __init__(self):
        # Grounding DINOモデルの読み込み
```

```
        self.grounding_dino_model = load_model("groundingdino/config/GroundingDINO_SwinT_OGC.py",
"groundingdino/weights/groundingdino_swint_ogc.pth")

        # SAMモデルの読み込み
        self.sam = sam_model_registry["vit_h"](checkpoint="sam_vit_h_4b8939.pth")
        self.sam_predictor = SamPredictor(self.sam)

        # ROS関連の設定
        self.bridge = CvBridge()
        self.image_sub = rospy.Subscriber("/camera/color/image_raw", RosImage, self.image_callback)
        self.depth_sub = rospy.Subscriber("/camera/depth/image_raw", RosImage, self.depth_callback)
        self.pose_pub = rospy.Publisher("/vision/object_poses", Pose, queue_size=10)

        # 最新の画像と深度情報を保存する変数
        self.latest_image = None
        self.latest_depth = None

        # 検出対象のクラス
        self.classes = ["mug", "kettle", "spoon", "coffee_container", "drawer_handle"]

    def image_callback(self, msg):
        # ROSのImage型をOpenCVの画像に変換
        self.latest_image = self.bridge.imgmsg_to_cv2(msg, "bgr8")

    def depth_callback(self, msg):
        # ROSのImage型をOpenCVの深度画像に変換
        self.latest_depth = self.bridge.imgmsg_to_cv2(msg, "32FC1")

    def detect_objects(self, text_prompt):
        if self.latest_image is None or self.latest_depth is None:
            rospy.logwarn("No image or depth data available.")
            return None

        # 画像の前処理
        image_pil = Image.fromarray(self.latest_image)
        image_tensor, _ = load_image(image_pil)

        # Grounding DINOで物体検出
        boxes, logits, phrases = predict(
            model=self.grounding_dino_model,
            image=image_tensor,
            caption=text_prompt,
            box_threshold=0.3,
            text_threshold=0.25
        )

        # 検出結果がない場合
        if len(boxes) == 0:
            rospy.logwarn(f"No objects detected for prompt: {text_prompt}")
            return None

        # SAMでセグメンテーション
        self.sam_predictor.set_image(self.latest_image)

        result_poses = []

        for box, logit, phrase in zip(boxes, logits, phrases):
            # バウンディングボックスの座標を取得
            x0, y0, x1, y1 = box

            # SAMでセグメンテーション
```

```
            sam_mask, _, _ = self.sam_predictor.predict(
                box=np.array([x0, y0, x1, y1]),
                multimask_output=False
            )

            # マスクを用いて深度情報から3D位置を計算
            mask = sam_mask[0].astype(np.uint8)
            masked_depth = self.latest_depth.copy()
            masked_depth[mask == 0] = 0

            # 有効な深度値を持つピクセルを抽出
            valid_depth = masked_depth[masked_depth > 0]

            if len(valid_depth) == 0:
                rospy.logwarn(f"No valid depth data for object: {phrase}")
                continue

            # 深度の中央値を計算（ノイズに強い）
            median_depth = np.median(valid_depth)

            # バウンディングボックスの中心を計算
            center_x = (x0 + x1) / 2
            center_y = (y0 + y1) / 2

            # 画像座標系から3D座標系への変換
            # （カメラの内部パラメータが必要）
            fx = 500  # カメラの焦点距離x（実際の値に置き換える）
            fy = 500  # カメラの焦点距離y（実際の値に置き換える）
            cx = 320  # 主点x（実際の値に置き換える）
            cy = 288  # 主点y（実際の値に置き換える）

            # 3D座標を計算
            z = median_depth
            x = (center_x - cx) * z / fx
            y = (center_y - cy) * z / fy

            # Poseメッセージを作成
            pose = Pose()
            pose.position.x = x
            pose.position.y = y
            pose.position.z = z

            # 姿勢は単純化のため単位四元数とする
            pose.orientation.x = 0.0
            pose.orientation.y = 0.0
            pose.orientation.z = 0.0
            pose.orientation.w = 1.0

            # 結果を追加
            result_poses.append((phrase, pose, logit.item()))

        # 信頼度でソート
        result_poses.sort(key=lambda x: x[2], reverse=True)

        # 結果を公開
        for phrase, pose, _ in result_poses:
            rospy.loginfo(f"Detected {phrase} at position: {pose.position.x}, {pose.position.y}, {pose.position.z}")
            self.pose_pub.publish(pose)
```

```
        return result_poses

    def detect_specific_object(self, object_name):
        # 特定の物体を検出するためのプロンプトを作成
        text_prompt = object_name

        # 物体検出を実行
        results = self.detect_objects(text_prompt)

        if results is None or len(results) == 0:
            return None

        # 最も信頼度の高い結果を返す
        return results[0][1]  # Poseを返す
```

このコードは、ROSのノードとして実装され、カメラからの画像と深度情報を処理して、物体のポーズを検出し、ROSのトピックとして公開する。検出された物体のポーズは、ロボット制御システムによって利用され、物体の把持や操作に使用される。

#### オクルージョンと不確実性の処理
視覚システムは、オクルージョン（物体の一部が他の物体に隠れる状態）や不確実性に対処するメカニズムを備えている。オクルージョンの処理には、以下のアプローチを採用する：

1. 物体の部分的な可視性に基づく検出：物体の一部が見えている場合でも、その特徴に基づいて物体を検出する。
2. 時間的な追跡：物体が一時的に隠れた場合でも、過去の位置情報に基づいて追跡を継続する。
3. 複数視点からの観測：可能であれば、異なる角度からの観測を組み合わせて、オクルージョンの影響を軽減する。

不確実性の処理には、以下のアプローチを採用する：

1. 確率的表現：物体の位置と姿勢を確率分布として表現し、不確実性を明示的に考慮する。
2. フィルタリング：カルマンフィルタなどの手法を用いて、ノイズの影響を軽減し、より安定した推定を行う。
3. アクティブ知覚：不確実性が高い場合、ロボットが能動的に視点を変えるなどの行動を取り、より多くの情報を収集する。

実験結果によれば、オクルージョン率が20%〜30%の場合、白いマグカップの検出成功率は約90%であったが、オクルージョン率が80%〜90%になると、検出成功率は約20%に低下した。このような状況では、力覚フィードバックなどの他のモダリティを活用することが重要となる。

オクルージョンと不確実性の処理は、以下のようなPythonコードで実装される：

```python
import numpy as np
import rospy
from geometry_msgs.msg import Pose
from filterpy.kalman import KalmanFilter

class ObjectTracker:
    def __init__(self, object_name):
        self.object_name = object_name

        # カルマンフィルタの初期化
```

```
        self.kf = KalmanFilter(dim_x=6, dim_z=3)  # 状態: [x, y, z, vx, vy, vz], 観測: [x, y, z]

        # 状態遷移行列
        dt = 0.1  # 時間ステップ
        self.kf.F = np.array([
            [1, 0, 0, dt, 0, 0],
            [0, 1, 0, 0, dt, 0],
            [0, 0, 1, 0, 0, dt],
            [0, 0, 0, 1, 0, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 0, 1]
        ])

        # 観測行列
        self.kf.H = np.array([
            [1, 0, 0, 0, 0, 0],
            [0, 1, 0, 0, 0, 0],
            [0, 0, 1, 0, 0, 0]
        ])

        # 測定ノイズ
        self.kf.R = np.eye(3) * 0.01

        # プロセスノイズ
        self.kf.Q = np.eye(6) * 0.01

        # 初期状態
        self.kf.x = np.zeros(6)

        # 初期共分散
        self.kf.P = np.eye(6) * 1000

        # 最後に物体が見えた時間
        self.last_seen_time = None

        # 物体が見えなくなってから追跡を続ける最大時間（秒）
        self.max_tracking_duration = 3.0

        # 物体の可視性
        self.is_visible = False

        # ROS関連の設定
        self.pose_sub = rospy.Subscriber(f"/vision/object_poses/{object_name}", Pose, self.pose_callback)
        self.tracked_pose_pub = rospy.Publisher(f"/vision/tracked_object_poses/{object_name}", Pose, queue_size=10)

        # タイマーで定期的に状態を更新
        self.timer = rospy.Timer(rospy.Duration(dt), self.update)

    def pose_callback(self, msg):
        # 物体が検出された場合
        self.is_visible = True
        self.last_seen_time = rospy.Time.now()

        # 観測値を設定
        z = np.array([
            msg.position.x,
            msg.position.y,
            msg.position.z
        ])
```

```
        # カルマンフィルタを更新
        self.kf.update(z)

    def update(self, event):
        # 状態を予測
        self.kf.predict()

        # 物体が見えなくなってからの経過時間を計算
        if self.last_seen_time is not None:
            elapsed_time = (rospy.Time.now() - self.last_seen_time).to_sec()

            # 一定時間以上見えない場合、追跡を停止
            if elapsed_time > self.max_tracking_duration:
                self.is_visible = False

        # 追跡結果を公開
        if self.last_seen_time is not None:
            pose = Pose()
            pose.position.x = self.kf.x[0]
            pose.position.y = self.kf.x[1]
            pose.position.z = self.kf.x[2]

            # 姿勢は単純化のため単位四元数とする
            pose.orientation.x = 0.0
            pose.orientation.y = 0.0
            pose.orientation.z = 0.0
            pose.orientation.w = 1.0

            # 不確実性の情報を追加（ROSのPoseメッセージには直接含められないため、別のトピックで公開するか、拡張する必要がある）
            uncertainty = np.sqrt(np.diag(self.kf.P)[:3]) # 位置の不確実性（標準偏差）

            rospy.logdebug(f"Tracked {self.object_name} at position: {pose.position.x}, {pose.position.y}, {pose.position.z}")
            rospy.logdebug(f"Uncertainty: {uncertainty}")

            self.tracked_pose_pub.publish(pose)

class ActivePerception:
    def __init__(self, object_detector, robot_controller):
        self.object_detector = object_detector
        self.robot_controller = robot_controller

        # 観測位置のリスト（ロボットのベース座標系）
        self.observation_poses = [
            # 正面からの観測
            {'position': [0.5, 0.0, 0.5], 'orientation': [0.0, 0.0, 0.0, 1.0]},
            # 左からの観測
            {'position': [0.5, 0.3, 0.5], 'orientation': [0.0, 0.0, 0.3826834, 0.9238795]},
            # 右からの観測
            {'position': [0.5, -0.3, 0.5], 'orientation': [0.0, 0.0, -0.3826834, 0.9238795]},
            # 上からの観測
            {'position': [0.5, 0.0, 0.7], 'orientation': [0.0, 0.3826834, 0.0, 0.9238795]}
        ]

    def find_object_with_active_perception(self, object_name, max_attempts=4):
        rospy.loginfo(f"Searching for {object_name} with active perception...")

        # 各観測位置から物体を探す
```

```
        for i, pose in enumerate(self.observation_poses[:max_attempts]):
            rospy.loginfo(f"Moving to observation pose {i+1}/{len(self.observation_poses)}")

            # ロボットを観測位置に移動
            target_pose = Pose()
            target_pose.position.x = pose['position'][0]
            target_pose.position.y = pose['position'][1]
            target_pose.position.z = pose['position'][2]
            target_pose.orientation.x = pose['orientation'][0]
            target_pose.orientation.y = pose['orientation'][1]
            target_pose.orientation.z = pose['orientation'][2]
            target_pose.orientation.w = pose['orientation'][3]

            self.robot_controller.move_to_pose(target_pose)

            # 物体を検出
            object_pose = self.object_detector.detect_specific_object(object_name)

            if object_pose is not None:
                rospy.loginfo(f"Found {object_name} at position: {object_pose.position.x}, {object_pose.position.y},
{object_pose.position.z}")
                return object_pose

        rospy.logwarn(f"Failed to find {object_name} after {max_attempts} attempts")
        return None
```

このコードは、カルマンフィルタを用いた物体追跡と、アクティブ知覚による物体探索を実装している。物体追跡では、物体が一時的に隠れた場合でも、過去の位置情報と速度情報に基づいて追跡を継続する。また、アクティブ知覚では、物体が見つからない場合、ロボットが異なる視点から観測を行い、物体を探索する。

### 力覚モジュール
力覚モジュールは、ロボットのエンドエフェクタが受ける力を測定し、物体操作の精度を向上させる役割を担う。

#### 力覚センサーの校正
力覚センサーの校正は、重力の影響を補正するために、外部力がない状態でセンサーがゼロを示すように調整する。これにより、エンドエフェクタに加わる外部力を正確に予測できる。校正プロセスは、以下の手順で行われる：

1. 一つの軸でセンサーをゼロにする。
2. センサーを回転させる。
3. 次の軸でセンサーをゼロにする。
4. すべての軸で同様のプロセスを繰り返す。

校正後、ローカルな力をグローバル平面に変換して、異なる回転でのエンドエフェクタに加わる上向きの力を推定する。変換は以下の式で行われる：

$$F_{global} = T_{end\_effector\_to\_robot\_base} \times F_{local}$$

ここで、$F_{global}$はロボットのベース座標系での力ベクトル、$T_{end_effector_to_robot_base}$はエンドエフェクタの座標系からロボットのベース座標系への変換行列、$F_{local}$はエンドエフェクタの局所座標系での力ベクトルである。

力覚センサーの校正は、以下のようなPythonコードで実装される：

```python
import numpy as np
import rospy
import tf2_ros
import tf2_geometry_msgs
from geometry_msgs.msg import WrenchStamped, TransformStamped
from sensor_msgs.msg import JointState

class ForceSensorCalibrator:
    def __init__(self):
        # ROS関連の設定
        self.force_sub = rospy.Subscriber("/force_torque_sensor/raw", WrenchStamped, self.force_callback)
        self.joint_sub = rospy.Subscriber("/joint_states", JointState, self.joint_callback)
        self.calibrated_force_pub = rospy.Publisher("/force_torque_sensor/calibrated", WrenchStamped, queue_size=10)

        # TF関連の設定
        self.tf_buffer = tf2_ros.Buffer()
        self.tf_listener = tf2_ros.TransformListener(self.tf_buffer)

        # 校正用のパラメータ
        self.gravity_compensation = np.zeros(6)  # [fx, fy, fz, tx, ty, tz]
        self.is_calibrated = False

        # 最新の力覚データ
        self.latest_force_data = None

        # 最新の関節状態
        self.latest_joint_state = None

    def force_callback(self, msg):
        # 最新の力覚データを保存
        self.latest_force_data = msg

        # 校正済みの場合、補正した力を公開
        if self.is_calibrated and self.latest_joint_state is not None:
            calibrated_force = self.calibrate_force(msg)
            self.calibrated_force_pub.publish(calibrated_force)

    def joint_callback(self, msg):
        # 最新の関節状態を保存
        self.latest_joint_state = msg

    def calibrate(self):
        rospy.loginfo("Starting force sensor calibration...")

        if self.latest_force_data is None or self.latest_joint_state is None:
            rospy.logwarn("No force data or joint state available. Cannot calibrate.")
            return False

        # 各軸で校正を行う
        axes = ["x", "y", "z"]

        for axis_idx, axis in enumerate(axes):
            rospy.loginfo(f"Calibrating {axis} axis...")

            # 現在の力を記録
            current_force = np.array([
                self.latest_force_data.wrench.force.x,
                self.latest_force_data.wrench.force.y,
                self.latest_force_data.wrench.force.z,
                self.latest_force_data.wrench.torque.x,
                self.latest_force_data.wrench.torque.y,
```

```
                self.latest_force_data.wrench.torque.z
            ])

            # 重力補償値を更新
            self.gravity_compensation[axis_idx] = current_force[axis_idx]

            rospy.loginfo(f"{axis} axis calibrated. Offset: {self.gravity_compensation[axis_idx]}")

            # ロボットを回転させて次の軸を校正する準備
            # （実際のコードでは、ロボットを適切に回転させる処理が必要）
            rospy.sleep(2.0) # 回転後の安定を待つ

        rospy.loginfo("Force sensor calibration completed.")
        rospy.loginfo(f"Gravity compensation values: {self.gravity_compensation}")

        self.is_calibrated = True
        return True

    def calibrate_force(self, force_msg):
        # 力覚データをNumPy配列に変換
        force_array = np.array([
            force_msg.wrench.force.x,
            force_msg.wrench.force.y,
            force_msg.wrench.force.z,
            force_msg.wrench.torque.x,
            force_msg.wrench.torque.y,
            force_msg.wrench.torque.z
        ])

        # 重力補償を適用
        calibrated_force_array = force_array - self.gravity_compensation

        # エンドエフェクタからベース座標系への変換を取得
        try:
            transform = self.tf_buffer.lookup_transform(
                "base_link",
                force_msg.header.frame_id,
                rospy.Time(0),
                rospy.Duration(1.0)
            )

            # 回転行列を抽出
            q = transform.transform.rotation
            rotation_matrix = self.quaternion_to_rotation_matrix(q)

            # 力とトルクを変換
            force_local = calibrated_force_array[:3]
            torque_local = calibrated_force_array[3:]

            force_global = rotation_matrix @ force_local
            torque_global = rotation_matrix @ torque_local

            # 変換された力とトルクでメッセージを作成
            calibrated_msg = WrenchStamped()
            calibrated_msg.header = force_msg.header
            calibrated_msg.header.frame_id = "base_link"

            calibrated_msg.wrench.force.x = force_global[0]
            calibrated_msg.wrench.force.y = force_global[1]
            calibrated_msg.wrench.force.z = force_global[2]

            calibrated_msg.wrench.torque.x = torque_global[0]
```

```
            calibrated_msg.wrench.torque.y = torque_global[1]
            calibrated_msg.wrench.torque.z = torque_global[2]

            return calibrated_msg

        except (tf2_ros.LookupException, tf2_ros.ConnectivityException, tf2_ros.ExtrapolationException) as e:
            rospy.logwarn(f"Failed to transform force: {e}")

            # 変換できない場合、重力補償のみ適用したメッセージを返す
            calibrated_msg = WrenchStamped()
            calibrated_msg.header = force_msg.header

            calibrated_msg.wrench.force.x = calibrated_force_array[0]
            calibrated_msg.wrench.force.y = calibrated_force_array[1]
            calibrated_msg.wrench.force.z = calibrated_force_array[2]

            calibrated_msg.wrench.torque.x = calibrated_force_array[3]
            calibrated_msg.wrench.torque.y = calibrated_force_array[4]
            calibrated_msg.wrench.torque.z = calibrated_force_array[5]

            return calibrated_msg

    def quaternion_to_rotation_matrix(self, q):
        # 四元数から回転行列への変換
        x, y, z, w = q.x, q.y, q.z, q.w

        rotation_matrix = np.array([
            [1 - 2*y*y - 2*z*z, 2*x*y - 2*z*w, 2*x*z + 2*y*w],
            [2*x*y + 2*z*w, 1 - 2*x*x - 2*z*z, 2*y*z - 2*x*w],
            [2*x*z - 2*y*w, 2*y*z + 2*x*w, 1 - 2*x*x - 2*y*y]
        ])

        return rotation_matrix
```

このコードは、力覚センサーの校正と、ローカル座標系からグローバル座標系への力の変換を実装している。校正プロセスでは、各軸の重力補償値を計算し、これを用いて力覚データを補正する。また、TFライブラリを用いて、エンドエフェクタの座標系からロボットのベース座標系への変換を行い、力とトルクをグローバル座標系で表現する。

#### 力覚フィードバックの活用
力覚フィードバックは、以下のような様々なタスクで活用される：

1. 液体の注入量の制御：
  液体を注ぐ際、力覚フィードバックを用いて注入量を制御する。静的平衡条件を仮定し、低速での操作を維持することで、力と質量の関係を利用して流量を推定する。数学的には、以下の式で表される：

$$F_{up} \approx mg$$
$$\Delta F_{up} \approx \Delta mg$$

  ここで、$F_{up}$は上向きの力、$m$は質量、$g$は重力加速度、$\Delta F_{up}$は力の変化、$\Delta m$は質量の変化である。この関係を用いて、力の変化から注入量を推定することができる。

  実験結果によれば、ピッチ速度が4 m/sの場合、注入精度は100gあたり約5.4gであった。ただし、ピッチ速度が増加すると精度は低下し、30 m/sでは誤差が約20g/sに達した。これは、静的平衡の仮定が崩れ、注入媒体と容器の質量分布が測定精度に影響するためである。

2. 引き出しの開閉：

引き出しを開閉する際、力覚フィードバックを用いて適切な力を加える。引き出しの重さや摩擦などの特性に応じて、力の大きさと方向を調整する。例えば、引き出しが重い場合や引っかかっている場合、より大きな力を加える必要がある。また、引き出しの動きを検知し、開閉の進行状況をモニタリングする。

3. 物体の置き方：
物体を置く際、力覚フィードバックを用いて接触を検知し、適切な力で物体を置く。例えば、マグカップを置く際、上向きの力のピークを成功した配置の指標として使用する。これにより、物体を安定して置くことができ、落下や転倒を防ぐことができる。

4. ペンの圧力制御：
描画タスクでは、力覚フィードバックを用いてペンの圧力を制御する。均一な圧力を維持することで、一貫した線の太さと質を確保することができる。また、表面の硬さや摩擦などの特性に応じて、圧力を調整することができる。

システムは、3軸に沿って力ベクトルを継続的に管理し、知識ベース内の基準に基づいて加える力を調整する。LLMは、特定のダウンストリームタスクの要件を満たすために、必要な力の大きさと方向を動的に選択する。例えば、知識ベースには、オブジェクトの特性やタスクの要求に応じて適用される様々な力の大きさが指定されている場合がある。このアプローチにより、システムは幅広い運用基準に合わせて自律的に行動を調整することができる。

力覚フィードバックの活用は、以下のようなPythonコードで実装される：

```python
import numpy as np
import rospy
from geometry_msgs.msg import WrenchStamped, Pose, Twist
from std_msgs.msg import Float64

class ForceController:
    def __init__(self):
        # ROS関連の設定
        self.force_sub = rospy.Subscriber("/force_torque_sensor/calibrated", WrenchStamped, self.force_callback)
        self.velocity_pub = rospy.Publisher("/robot/cartesian_velocity_command", Twist, queue_size=10)
        self.poured_amount_pub = rospy.Publisher("/pouring/amount", Float64, queue_size=10)

        # 最新の力覚データ
        self.latest_force = None

        # 注入量の推定用パラメータ
        self.initial_force = None
        self.gravity = 9.8  # m/s^2

        # 力制御のパラメータ
        self.force_threshold = 10.0  # N
        self.contact_threshold = 1.0  # N
        self.max_velocity = 0.05  # m/s

    def force_callback(self, msg):
        # 最新の力覚データを保存
        self.latest_force = msg

    def start_pouring(self):
        rospy.loginfo("Starting pouring...")

        if self.latest_force is None:
            rospy.logwarn("No force data available. Cannot start pouring.")
            return False
```

```
    # 初期の力を記録
    self.initial_force = np.array([
        self.latest_force.wrench.force.x,
        self.latest_force.wrench.force.y,
        self.latest_force.wrench.force.z
    ])

    rospy.loginfo(f"Initial force: {self.initial_force}")

    return True

def update_pouring(self, target_amount):
    if self.latest_force is None or self.initial_force is None:
        rospy.logwarn("No force data or initial force available. Cannot update pouring.")
        return 0.0, False

    # 現在の力を取得
    current_force = np.array([
        self.latest_force.wrench.force.x,
        self.latest_force.wrench.force.y,
        self.latest_force.wrench.force.z
    ])

    # 力の変化から注入量を推定
    force_change = current_force[2] - self.initial_force[2]  # z軸方向の力の変化
    poured_amount = force_change / self.gravity  # kg

    # 注入量を公開
    amount_msg = Float64()
    amount_msg.data = poured_amount
    self.poured_amount_pub.publish(amount_msg)

    rospy.logdebug(f"Estimated poured amount: {poured_amount} kg")

    # 目標量に達したかどうかを判定
    is_complete = poured_amount >= target_amount

    return poured_amount, is_complete

def open_drawer_with_force_control(self, direction, max_distance=0.3, max_duration=5.0):
    rospy.loginfo("Opening drawer with force control...")

    if self.latest_force is None:
        rospy.logwarn("No force data available. Cannot open drawer.")
        return False

    # 初期の力を記録
    initial_force = np.array([
        self.latest_force.wrench.force.x,
        self.latest_force.wrench.force.y,
        self.latest_force.wrench.force.z
    ])

    # 引き出しを引く方向のベクトル（正規化）
    direction_norm = np.linalg.norm(direction)
    if direction_norm > 0:
        direction = direction / direction_norm

    # 速度コマンドの設定
    twist = Twist()
    twist.linear.x = direction[0] * self.max_velocity
```

```
        twist.linear.y = direction[1] * self.max_velocity
        twist.linear.z = direction[2] * self.max_velocity

        # 引き出しを引く
        start_time = rospy.Time.now()
        rate = rospy.Rate(10)  # 10Hz

        moved_distance = 0.0
        last_time = start_time

        while (rospy.Time.now() - start_time).to_sec() < max_duration and moved_distance < max_distance:
            if self.latest_force is None:
                rospy.logwarn("Lost force data during drawer opening.")
                break

            # 現在の力を取得
            current_force = np.array([
                self.latest_force.wrench.force.x,
                self.latest_force.wrench.force.y,
                self.latest_force.wrench.force.z
            ])

            # 力の大きさを計算
            force_magnitude = np.linalg.norm(current_force - initial_force)

            # 力が閾値を超えた場合、速度を調整
            if force_magnitude > self.force_threshold:
                rospy.logwarn(f"Force threshold exceeded: {force_magnitude} N")

                # 速度を減少
                scale_factor = self.force_threshold / force_magnitude
                twist.linear.x = direction[0] * self.max_velocity * scale_factor
                twist.linear.y = direction[1] * self.max_velocity * scale_factor
                twist.linear.z = direction[2] * self.max_velocity * scale_factor
            else:
                # 通常速度
                twist.linear.x = direction[0] * self.max_velocity
                twist.linear.y = direction[1] * self.max_velocity
                twist.linear.z = direction[2] * self.max_velocity

            # 速度コマンドを送信
            self.velocity_pub.publish(twist)

            # 移動距離を更新
            current_time = rospy.Time.now()
            dt = (current_time - last_time).to_sec()
            moved_distance += self.max_velocity * dt
            last_time = current_time

            rate.sleep()

        # 停止コマンドを送信
        twist.linear.x = 0.0
        twist.linear.y = 0.0
        twist.linear.z = 0.0
        self.velocity_pub.publish(twist)

        rospy.loginfo(f"Drawer opening completed. Moved distance: {moved_distance} m")

        return moved_distance > 0.1  # 10cm以上動いたら成功とみなす

    def place_object_with_force_control(self, target_pose, approach_distance=0.1, contact_force=2.0):
```

```
rospy.loginfo("Placing object with force control...")

if self.latest_force is None:
    rospy.logwarn("No force data available. Cannot place object.")
    return False

# アプローチ位置を計算
approach_pose = Pose()
approach_pose.position.x = target_pose.position.x
approach_pose.position.y = target_pose.position.y
approach_pose.position.z = target_pose.position.z + approach_distance
approach_pose.orientation = target_pose.orientation

# アプローチ位置に移動
# （実際のコードでは、ロボットを移動させる処理が必要）
rospy.sleep(1.0) # 移動完了を待つ

# 初期の力を記録
initial_force = np.array([
    self.latest_force.wrench.force.x,
    self.latest_force.wrench.force.y,
    self.latest_force.wrench.force.z
])

# 物体を下げる
twist = Twist()
twist.linear.z = -0.01  # ゆっくりと下げる

rate = rospy.Rate(10)  # 10Hz
contact_detected = False

while not contact_detected:
    if self.latest_force is None:
        rospy.logwarn("Lost force data during object placement.")
        break

    # 現在の力を取得
    current_force = np.array([
        self.latest_force.wrench.force.x,
        self.latest_force.wrench.force.y,
        self.latest_force.wrench.force.z
    ])

    # 力の変化を計算
    force_change = np.linalg.norm(current_force - initial_force)

    # 接触を検出
    if force_change > self.contact_threshold:
        rospy.loginfo(f"Contact detected: {force_change} N")
        contact_detected = True
        break

    # 速度コマンドを送信
    self.velocity_pub.publish(twist)

    rate.sleep()

# 停止コマンドを送信
twist.linear.z = 0.0
self.velocity_pub.publish(twist)
```

```
    if contact_detected:
        # 目標の接触力まで押し付ける
        twist.linear.z = -0.005 # さらにゆっくりと下げる

        target_force_reached = False

        while not target_force_reached:
            if self.latest_force is None:
                rospy.logwarn("Lost force data during force application.")
                break

            # 現在の力を取得
            current_force = np.array([
                self.latest_force.wrench.force.x,
                self.latest_force.wrench.force.y,
                self.latest_force.wrench.force.z
            ])

            # 力の変化を計算
            force_change = np.linalg.norm(current_force - initial_force)

            # 目標の力に達したかチェック
            if force_change >= contact_force:
                rospy.loginfo(f"Target force reached: {force_change} N")
                target_force_reached = True
                break

            # 速度コマンドを送信
            self.velocity_pub.publish(twist)

            rate.sleep()

        # 停止コマンドを送信
        twist.linear.z = 0.0
        self.velocity_pub.publish(twist)

        rospy.loginfo("Object placed successfully.")
        return True
    else:
        rospy.logwarn("Failed to detect contact. Object placement may have failed.")
        return False

def control_pen_pressure(self, target_force=1.0, tolerance=0.2):
    if self.latest_force is None:
        rospy.logwarn("No force data available. Cannot control pen pressure.")
        return False

    # 現在の力を取得
    current_force = np.array([
        self.latest_force.wrench.force.x,
        self.latest_force.wrench.force.y,
        self.latest_force.wrench.force.z
    ])

    # z軸方向の力（ペンの圧力）
    pen_pressure = abs(current_force[2])

    # 目標の力との差を計算
    force_error = target_force - pen_pressure

    # 許容範囲内かチェック
```

```
        if abs(force_error) <= tolerance:
            return True

        # 力を調整するための速度コマンド
        twist = Twist()

        if force_error > 0:
            # 圧力を増加
            twist.linear.z = -0.005 * force_error
        else:
            # 圧力を減少
            twist.linear.z = 0.005 * abs(force_error)

        # 速度コマンドを送信
        self.velocity_pub.publish(twist)

        return False
```

このコードは、力覚フィードバックを活用した様々なタスク（液体の注入量の制御、引き出しの開閉、物体の置き方、ペンの圧力制御）を実装している。各タスクでは、力覚フィードバックを用いて、適切な力を加えたり、力の変化から状態を推定したりする。例えば、液体の注入量の制御では、力の変化から注入量を推定し、目標量に達したら注入を停止する。また、引き出しの開閉では、力が閾値を超えた場合に速度を調整し、引き出しを安全に開ける。

### 音声認識・合成モジュール
音声認識・合成モジュールは、ユーザーの音声指示を認識し、システムの応答を音声で出力する役割を担う。

#### 音声認識
音声認識は、ユーザーの音声指示をテキストに変換するプロセスである。本システムでは、クラウドベースの音声認識サービス（例：Google Cloud Speech-to-Text）またはローカルで実行される音声認識エンジン（例：Mozilla DeepSpeech）を使用する。音声認識プロセスは以下の手順で行われる：

1. マイクロフォンから音声信号を取得する。
2. 音声信号をデジタル化し、前処理（ノイズ除去、正規化など）を行う。
3. 音声認識エンジンに音声データを送信し、テキストに変換する。
4. 認識結果を言語処理コンポーネントに送信する。

音声認識の精度は、環境ノイズ、話者の発音、方言、言語モデルの品質などの要因に影響される。本システムでは、ロボットの動作音や環境ノイズに対応するために、ノイズキャンセリング技術やビームフォーミング技術を活用する。

#### 音声合成
音声合成は、システムの応答をテキストから音声に変換するプロセスである。本システムでは、クラウドベースの音声合成サービス（例：Google Cloud Text-to-Speech）またはローカルで実行される音声合成エンジン（例：Festival、eSpeak）を使用する。音声合成プロセスは以下の手順で行われる：

1. 言語処理コンポーネントからテキスト応答を受け取る。
2. テキストを音声合成エンジンに送信し、音声データに変換する。
3. 音声データをスピーカーから出力する。

音声合成の品質は、合成エンジンの性能、声の選択、韻律（イントネーション、アクセント、リズム）の制御などの要因に影響される。本システムでは、自然な抑揚と発音を持つ音声を生成するために、最新の音声合成技術を活用する。

#### 音声対話管理
音声対話管理は、ユーザーとの音声対話を管理し、自然な会話を実現するプロセスである。本システムでは、以下の機能を提供する：

1. 話者識別：複数のユーザーの声を識別し、個人に適応した対応を行う。
2. 感情認識：ユーザーの音声から感情を認識し、適切な対応を行う。
3. 環境音認識：環境音（例：警告音、ドアの開閉音）を認識し、状況を理解する。
4. 対話状態管理：対話の文脈を管理し、適切な応答を生成する。
5. 割り込み処理：ユーザーの割り込みを検出し、適切に対応する。

音声対話管理は、言語処理コンポーネントと連携して、ユーザーとの自然なインタラクションを実現する。

音声認識・合成モジュールは、以下のようなPythonコードで実装される：

```python
import speech_recognition as sr
import pyttsx3
import numpy as np
import rospy
from std_msgs.msg import String
from google.cloud import speech
from google.cloud import texttospeech

class SpeechModule:
    def __init__(self, use_cloud=True):
        # ROS関連の設定
        self.speech_pub = rospy.Publisher("/speech/recognized", String, queue_size=10)
        self.text_sub = rospy.Subscriber("/speech/synthesize", String, self.synthesize_callback)

        # 音声認識関連の設定
        self.recognizer = sr.Recognizer()
        self.microphone = sr.Microphone()

        # 環境ノイズの調整
        with self.microphone as source:
            self.recognizer.adjust_for_ambient_noise(source)

        # クラウドサービスを使用するかどうか
        self.use_cloud = use_cloud

        if use_cloud:
            # Google Cloud Speech-to-Text クライアントの初期化
            self.speech_client = speech.SpeechClient()

            # Google Cloud Text-to-Speech クライアントの初期化
            self.tts_client = texttospeech.TextToSpeechClient()
        else:
            # ローカルの音声合成エンジンの初期化
            self.engine = pyttsx3.init()

            # 音声の設定
            voices = self.engine.getProperty('voices')
            self.engine.setProperty('voice', voices[0].id)  # 0: 男性, 1: 女性
```

```
        self.engine.setProperty('rate', 150)  # 話速
        self.engine.setProperty('volume', 0.8)  # 音量

    # 音声認識のタイマー
    self.timer = rospy.Timer(rospy.Duration(0.1), self.recognize_speech)

    # 話者識別のための話者プロファイル
    self.speaker_profiles = {}

    # 対話状態
    self.dialogue_state = {
        "context": [],
        "last_query": "",
        "last_response": "",
        "is_listening": True
    }

def recognize_speech(self, event=None):
    if not self.dialogue_state["is_listening"]:
        return

    try:
        with self.microphone as source:
            audio = self.recognizer.listen(source, timeout=1.0, phrase_time_limit=5.0)

        if self.use_cloud:
            # Google Cloud Speech-to-Text を使用
            audio_data = audio.get_raw_data()

            audio_config = speech.RecognitionConfig(
                encoding=speech.RecognitionConfig.AudioEncoding.LINEAR16,
                sample_rate_hertz=16000,
                language_code="ja-JP",
                enable_automatic_punctuation=True,
                model="command_and_search"
            )

            response = self.speech_client.recognize(
                config=audio_config,
                audio=speech.RecognitionAudio(content=audio_data)
            )

            if response.results:
                text = response.results[0].alternatives[0].transcript
                confidence = response.results[0].alternatives[0].confidence

                rospy.loginfo(f"Recognized: {text} (confidence: {confidence})")

                if confidence > 0.7:
                    # 認識結果を公開
                    msg = String()
                    msg.data = text
                    self.speech_pub.publish(msg)

                    # 対話状態を更新
                    self.dialogue_state["last_query"] = text
                    self.dialogue_state["context"].append({"role": "user", "content": text})
            else:
                rospy.logdebug("No speech recognized.")
        else:
            # SpeechRecognition のローカル認識を使用
            try:
```

```
                text = self.recognizer.recognize_google(audio, language="ja-JP")

                rospy.loginfo(f"Recognized: {text}")

                # 認識結果を公開
                msg = String()
                msg.data = text
                self.speech_pub.publish(msg)

                # 対話状態を更新
                self.dialogue_state["last_query"] = text
                self.dialogue_state["context"].append({"role": "user", "content": text})
            except sr.UnknownValueError:
                rospy.logdebug("Google Speech Recognition could not understand audio")
            except sr.RequestError as e:
                rospy.logerr(f"Could not request results from Google Speech Recognition service; {e}")

        except Exception as e:
            rospy.logerr(f"Error in speech recognition: {e}")

    def synthesize_callback(self, msg):
        text = msg.data

        rospy.loginfo(f"Synthesizing speech: {text}")

        # 対話状態を更新
        self.dialogue_state["is_listening"] = False
        self.dialogue_state["last_response"] = text
        self.dialogue_state["context"].append({"role": "assistant", "content": text})

        if self.use_cloud:
            # Google Cloud Text-to-Speech を使用
            synthesis_input = texttospeech.SynthesisInput(text=text)

            voice = texttospeech.VoiceSelectionParams(
                language_code="ja-JP",
                ssml_gender=texttospeech.SsmlVoiceGender.NEUTRAL
            )

            audio_config = texttospeech.AudioConfig(
                audio_encoding=texttospeech.AudioEncoding.LINEAR16
            )

            response = self.tts_client.synthesize_speech(
                input=synthesis_input,
                voice=voice,
                audio_config=audio_config
            )

            # 音声データを一時ファイルに保存
            with open("output.wav", "wb") as out:
                out.write(response.audio_content)

            # 音声を再生
            import subprocess
            subprocess.call(["aplay", "output.wav"])
        else:
            # ローカルの音声合成エンジンを使用
            self.engine.say(text)
            self.engine.runAndWait()

        # 対話状態を更新
        self.dialogue_state["is_listening"] = True
```

```
    def identify_speaker(self, audio):
        # 話者識別の実装
        # （実際のコードでは、話者識別のためのアルゴリズムが必要）
        return "unknown"

    def recognize_emotion(self, audio):
        # 感情認識の実装
        # （実際のコードでは、感情認識のためのアルゴリズムが必要）
        return "neutral"

    def recognize_environmental_sound(self, audio):
        # 環境音認識の実装
        # （実際のコードでは、環境音認識のためのアルゴリズムが必要）
        return []
```

このコードは、音声認識・合成モジュールの基本的な機能を実装している。音声認識では、SpeechRecognitionライブラリまたはGoogle Cloud Speech-to-Textを使用して、ユーザーの音声をテキストに変換する。音声合成では、pyttsx3ライブラリまたはGoogle Cloud Text-to-Speechを使用して、テキストを音声に変換する。また、対話状態を管理し、ユーザーとの自然な会話を実現するための機能も提供している。

### ロボット制御システム
ロボット制御システムは、言語処理、視覚システム、力覚モジュール、音声認識・合成モジュールからのフィードバックを統合し、ロボットの動作を制御する役割を担う。

#### ROSの設定と操作
本実施形態では、ROSを用いてロボットを制御する。ROSの設定と操作は以下の手順で行われる：

1. Kinova ROS Kortexドライバーを起動し、ROSネットワークとKinova Gen3ロボットの間の通信を確立する。このノードは、ROSネットワーク内で通信を可能にし、サブスクライバーがアクセスできるいくつかのトピックを公開し、ロボットの構成を変更するために呼び出すことができるサービスを提供する。ベースジョイントは40Hzの周波数で更新される。

2. Robotiq 2F-140mmグリッパーノードを50Hzで起動する。このノードは、USB接続を介してグリッパーとの通信リンクを設定し、グリッパーの正確な制御と操作データの交換を可能にするアクションサーバーを開始する。

3. 視覚モジュールノードを起動する。「classes」変数を使用して、環境内の選択されたオブジェクトのターゲットポーズを識別する。この変数は動的に更新でき、シーンの変化に適応できる。オブジェクトのポーズ座標は、約1/3Hzで公開される。これは主に、Grounding DINOがオブジェクトを検出しバウンディングボックスを確立するための処理時間によるものである。

4. 力覚ノードを100Hzで起動し、ATI力覚トランスデューサーにローカライズされた多軸力覚および回転力の読み取り値を提供する。読み取り値は、四元数ベースの3×3回転行列を使用してロボットのグローバルベースフレームに合わせて変換され、固定された自由度にわたる過去5つの時間ステップにわたる生の値と平均値を提供する。

5. 音声認識・合成ノードを起動し、ユーザーの音声指示を認識し、システムの応答を音声で出力する。

6. ROSが言語処理、視覚システム、力覚モジュール、音声認識・合成モジュールからのマルチモーダルフィードバックデータを継続的に処理する。

ROSの設定と操作は、以下のようなlaunchファイルで実装される：

```xml
<launch>
  <!-- Kinova Gen3ロボットドライバーの起動 -->
  <include file="$(find kortex_driver)/launch/kortex_driver.launch">
    <arg name="robot_name" value="my_gen3"/>
    <arg name="ip_address" value="192.168.1.10"/>
    <arg name="cyclic_data_publish_rate" value="40"/>
  </include>

  <!-- Robotiqグリッパードライバーの起動 -->
  <node name="robotiq_2f_140_driver" pkg="robotiq_2f_gripper_control" type="Robotiq2FGripperRtuNode.py" output="screen">
    <param name="comport" value="/dev/ttyUSB0"/>
    <param name="baud" value="115200"/>
  </node>

  <!-- Azure Kinectドライバーの起動 -->
  <include file="$(find azure_kinect_ros_driver)/launch/driver.launch">
    <arg name="color_resolution" value="1080P"/>
    <arg name="depth_mode" value="NFOV_UNBINNED"/>
    <arg name="fps" value="30"/>
  </include>

  <!-- 視覚システムの起動 -->
  <node name="object_detector" pkg="my_robot_vision" type="object_detector.py" output="screen">
    <param name="model_path" value="$(find my_robot_vision)/models"/>
    <param name="classes" value="mug,kettle,spoon,coffee_container,drawer_handle"/>
  </node>

  <!-- 力覚センサードライバーの起動 -->
  <node name="force_sensor_driver" pkg="ati_force_sensor" type="force_sensor_driver" output="screen">
    <param name="ip_address" value="192.168.1.11"/>
    <param name="port" value="49152"/>
    <param name="frame_id" value="force_sensor_link"/>
    <param name="sample_rate" value="100"/>
  </node>

  <!-- 力覚モジュールの起動 -->
  <node name="force_controller" pkg="my_robot_force" type="force_controller.py" output="screen"/>

  <!-- 音声認識・合成モジュールの起動 -->
  <node name="speech_module" pkg="my_robot_speech" type="speech_module.py" output="screen">
    <param name="use_cloud" value="true"/>
  </node>

  <!-- 言語処理コンポーネントの起動 -->
  <node name="language_processor" pkg="my_robot_language" type="language_processor.py" output="screen">
    <param name="api_key" value="your-api-key"/>
    <param name="model" value="gpt-4-0613"/>
  </node>

  <!-- マルチモーダル統合モジュールの起動 -->
  <node name="multimodal_integrator" pkg="my_robot_integration" type="multimodal_integrator.py" output="screen"/>

  <!-- ロボット制御システムの起動 -->
  <node name="robot_controller" pkg="my_robot_control" type="robot_controller.py" output="screen"/>
</launch>
```

```

このlaunchファイルは、システムの各コンポーネント（Kinova Gen3ロボットドライバー、Robotiqグリッパードライバー、Azure Kinectドライバー、視覚システム、力覚モジュール、音声認識・合成モジュール、言語処理コンポーネント、マルチモーダル統合モジュール、ロボット制御システム）を起動し、それらの間の通信を確立する。

#### 安全制約の実装
ロボットの動作は、速度を制御する6自由度のツイストコマンドと、開閉のための可変速度および力のグリッパー手順に基づいている。これにより、最大速度や力の制限、作業空間の境界などのハードコードされた安全制約を統合できる。具体的な制約は以下の通りである：

1. 線形速度は±0.05 m/s以内に制限される。
2. 角速度は±60°/s以内に制限される。
3. エンドエフェクタの力は20Nに制限される。
4. エンドエフェクタは事前定義された作業空間の境界（x = [0.0, 1.1]、y = [-0.3, 0.3]、z = [0, 1.0]）内に制限される。

これらの制約は基本的なモーションプリミティブにコード化されているため、言語モデルのエラーがこれらの制約をオーバーライドすることはない。また、エンドエフェクタの位置は、10Hzの周波数でパブリッシャーによって将来の時間ステップでチェックされる。

安全制約の実装は、以下のようなPythonコードで行われる：

```python
import numpy as np
import rospy
from geometry_msgs.msg import Twist, Pose
from std_msgs.msg import Bool

class SafetyConstraints:
    def __init__(self):
        # 安全制約のパラメータ
        self.max_linear_velocity = 0.05  # m/s
        self.max_angular_velocity = 1.047  # rad/s (60 deg/s)
        self.max_force = 20.0  # N

        # 作業空間の境界
        self.workspace_bounds = {
            'x': [0.0, 1.1],
            'y': [-0.3, 0.3],
            'z': [0.0, 1.0]
        }

        # ROS関連の設定
        self.velocity_sub = rospy.Subscriber("/robot/cartesian_velocity_command_unsafe", Twist, self.velocity_callback)
        self.pose_sub = rospy.Subscriber("/robot/cartesian_pose", Pose, self.pose_callback)
        self.force_sub = rospy.Subscriber("/force_torque_sensor/calibrated", WrenchStamped, self.force_callback)

        self.velocity_pub = rospy.Publisher("/robot/cartesian_velocity_command", Twist, queue_size=10)
        self.safety_violation_pub = rospy.Publisher("/robot/safety_violation", Bool, queue_size=10)

        # 最新の状態
        self.latest_pose = None
        self.latest_force = None

        # 安全性チェックのタイマー
```
```

```
    self.timer = rospy.Timer(rospy.Duration(0.1), self.check_safety)

def velocity_callback(self, msg):
    # 速度コマンドを安全制約に従って制限
    safe_velocity = self.apply_velocity_constraints(msg)

    # 安全な速度コマンドを公開
    self.velocity_pub.publish(safe_velocity)

def pose_callback(self, msg):
    # 最新のポーズを保存
    self.latest_pose = msg

def force_callback(self, msg):
    # 最新の力を保存
    self.latest_force = msg

def apply_velocity_constraints(self, velocity):
    # 速度コマンドのコピーを作成
    safe_velocity = Twist()

    # 線形速度の制限
    safe_velocity.linear.x = np.clip(velocity.linear.x, -self.max_linear_velocity, self.max_linear_velocity)
    safe_velocity.linear.y = np.clip(velocity.linear.y, -self.max_linear_velocity, self.max_linear_velocity)
    safe_velocity.linear.z = np.clip(velocity.linear.z, -self.max_linear_velocity, self.max_linear_velocity)

    # 角速度の制限
    safe_velocity.angular.x = np.clip(velocity.angular.x, -self.max_angular_velocity, self.max_angular_velocity)
    safe_velocity.angular.y = np.clip(velocity.angular.y, -self.max_angular_velocity, self.max_angular_velocity)
    safe_velocity.angular.z = np.clip(velocity.angular.z, -self.max_angular_velocity, self.max_angular_velocity)

    return safe_velocity

def check_workspace_bounds(self, pose):
    # 作業空間の境界をチェック
    x_in_bounds = self.workspace_bounds['x'][0] <= pose.position.x <= self.workspace_bounds['x'][1]
    y_in_bounds = self.workspace_bounds['y'][0] <= pose.position.y <= self.workspace_bounds['y'][1]
    z_in_bounds = self.workspace_bounds['z'][0] <= pose.position.z <= self.workspace_bounds['z'][1]

    return x_in_bounds and y_in_bounds and z_in_bounds

def check_force_limits(self, force):
    # 力の制限をチェック
    force_magnitude = np.sqrt(
        force.wrench.force.x**2 +
        force.wrench.force.y**2 +
        force.wrench.force.z**2
    )

    return force_magnitude <= self.max_force

def check_safety(self, event=None):
    if self.latest_pose is None or self.latest_force is None:
        return

    # 作業空間の境界をチェック
    workspace_safe = self.check_workspace_bounds(self.latest_pose)

    # 力の制限をチェック
    force_safe = self.check_force_limits(self.latest_force)
```

```
    # 安全性違反があるかどうかを公開
    safety_violation = not (workspace_safe and force_safe)

    msg = Bool()
    msg.data = safety_violation
    self.safety_violation_pub.publish(msg)

    if safety_violation:
        # 安全性違反がある場合、ロボットを停止
        stop_velocity = Twist()
        self.velocity_pub.publish(stop_velocity)

        rospy.logwarn("Safety violation detected! Robot stopped.")

        if not workspace_safe:
            rospy.logwarn("Workspace bounds violated.")

        if not force_safe:
            rospy.logwarn("Force limits exceeded.")
```

このコードは、ロボットの安全制約を実装している。速度コマンドを安全制約に従って制限し、作業空間の境界と力の制限をチェックする。安全性違反が検出された場合、ロボットを停止し、警告メッセージを表示する。これにより、ロボットの動作が安全かつ信頼性の高いものとなる。

#### フィードバックループの実装
ロボット制御システムは、視覚および力覚フィードバックを利用して、リアルタイムで動作を調整する。フィードバックループの実装は以下の通りである：

1. 視覚フィードバック：
 視覚システムからの情報を用いて、物体の位置と姿勢を継続的に更新する。物体が移動した場合、新しい位置を特定し、動作計画を調整する。また、予期せぬ障害物が現れた場合、回避行動を計画する。視覚フィードバックは、特に物体の把持や配置など、位置精度が重要なタスクで重要である。

2. 力覚フィードバック：
 力覚センサーからの情報を用いて、物体との相互作用を制御する。例えば、液体を注ぐ際には、力の変化から注入量を推定し、目標量に達したら注入を停止する。また、引き出しを開ける際には、力の大きさと方向を調整して、スムーズな開閉を実現する。力覚フィードバックは、特に視覚情報が限られた状況（例：視界が遮られた場合）で重要である。

3. 音声フィードバック：
 音声認識・合成モジュールからの情報を用いて、ユーザーとの対話を継続する。ユーザーの指示に応じて、タスクの計画を更新したり、追加の情報を提供したりする。音声フィードバックは、特にユーザーとの自然なインタラクションを実現する上で重要である。

4. マルチモーダル統合：
 視覚、力覚、音声などの異なるモダリティからの情報を統合し、より豊かな環境理解と適応的な行動生成を実現する。例えば、視覚情報から物体の位置を特定し、力覚情報から物体の重さを推定し、これらの情報を統合して適切な把持力と位置を決定する。

フィードバックループは、40Hzでのエンドエフェクタの位置（p）と姿勢（q）の更新を含み、ロボットが外乱（例：ユーザーによるカップの移動）に応答できるようにする。

フィードバックループの実装は、以下のようなPythonコードで行われる：

```python
import numpy as np
import rospy
import tf2_ros
from geometry_msgs.msg import Pose, Twist, WrenchStamped
from std_msgs.msg import String
from sensor_msgs.msg import JointState

class FeedbackController:
    def __init__(self):
        # ROS関連の設定
        self.pose_sub = rospy.Subscriber("/robot/cartesian_pose", Pose, self.pose_callback)
        self.joint_sub = rospy.Subscriber("/joint_states", JointState, self.joint_callback)
        self.force_sub = rospy.Subscriber("/force_torque_sensor/calibrated", WrenchStamped, self.force_callback)
        self.object_pose_sub = rospy.Subscriber("/vision/tracked_object_poses/target", Pose, self.object_pose_callback)
        self.speech_sub = rospy.Subscriber("/speech/recognized", String, self.speech_callback)

        self.velocity_pub = rospy.Publisher("/robot/cartesian_velocity_command_unsafe", Twist, queue_size=10)
        self.speech_pub = rospy.Publisher("/speech/synthesize", String, queue_size=10)

        # TF関連の設定
        self.tf_buffer = tf2_ros.Buffer()
        self.tf_listener = tf2_ros.TransformListener(self.tf_buffer)

        # 最新の状態
        self.latest_pose = None
        self.latest_joints = None
        self.latest_force = None
        self.target_object_pose = None
        self.latest_speech = None

        # タスク状態
        self.task_state = {
            "current_task": None,
            "target_pose": None,
            "is_tracking": False,
            "is_force_controlled": False,
            "is_speech_controlled": False
        }

        # フィードバックループのタイマー
        self.timer = rospy.Timer(rospy.Duration(0.025), self.feedback_loop)  # 40Hz

    def pose_callback(self, msg):
        # 最新のポーズを保存
        self.latest_pose = msg

    def joint_callback(self, msg):
        # 最新の関節状態を保存
        self.latest_joints = msg

    def force_callback(self, msg):
        # 最新の力を保存
        self.latest_force = msg

    def object_pose_callback(self, msg):
        # 最新の対象物体のポーズを保存
        self.target_object_pose = msg

    def speech_callback(self, msg):
        # 最新の音声認識結果を保存
        self.latest_speech = msg.data
```

```
    # 音声コマンドを処理
    self.process_speech_command(msg.data)

def process_speech_command(self, command):
    # 音声コマンドを処理
    if "停止" in command or "止まれ" in command or "stop" in command:
        # ロボットを停止
        self.stop_robot()
        self.speak("停止します")
    elif "続けて" in command or "再開" in command or "continue" in command:
        # タスクを再開
        self.task_state["is_tracking"] = True
        self.speak("タスクを再開します")
    elif "もう少し" in command and "注いで" in command:
        # 追加で注ぐ
        self.speak("追加で注ぎます")
        # （実際のコードでは、追加で注ぐための処理が必要）
    elif "十分" in command or "enough" in command:
        # 注ぐのを停止
        self.speak("注ぐのを停止します")
        # （実際のコードでは、注ぐのを停止するための処理が必要）

def speak(self, text):
    # 音声合成メッセージを公開
    msg = String()
    msg.data = text
    self.speech_pub.publish(msg)

def stop_robot(self):
    # ロボットを停止
    self.task_state["is_tracking"] = False
    self.task_state["is_force_controlled"] = False

    # 停止コマンドを送信
    twist = Twist()
    self.velocity_pub.publish(twist)

def track_object(self):
    if self.latest_pose is None or self.target_object_pose is None:
        return

    # 対象物体への方向ベクトルを計算
    direction = np.array([
        self.target_object_pose.position.x - self.latest_pose.position.x,
        self.target_object_pose.position.y - self.latest_pose.position.y,
        self.target_object_pose.position.z - self.latest_pose.position.z
    ])

    # 距離を計算
    distance = np.linalg.norm(direction)

    # 方向ベクトルの正規化
    if distance > 0:
        direction = direction / distance
```

```
    # 速度コマンドの設定
    twist = Twist()

    # 対象物体に近づく速度を設定
    max_velocity = 0.05  # m/s
    velocity_scale = min(distance, 1.0)  # 距離に応じて速度をスケーリング

    twist.linear.x = direction[0] * max_velocity * velocity_scale
    twist.linear.y = direction[1] * max_velocity * velocity_scale
    twist.linear.z = direction[2] * max_velocity * velocity_scale

    # 速度コマンドを送信
    self.velocity_pub.publish(twist)

def force_control(self):
    if self.latest_force is None or self.task_state["target_pose"] is None:
        return

    # 目標位置への方向ベクトルを計算
    direction = np.array([
        self.task_state["target_pose"].position.x - self.latest_pose.position.x,
        self.task_state["target_pose"].position.y - self.latest_pose.position.y,
        self.task_state["target_pose"].position.z - self.latest_pose.position.z
    ])

    # 距離を計算
    distance = np.linalg.norm(direction)

    # 方向ベクトルの正規化
    if distance > 0:
        direction = direction / distance

    # 力の大きさを計算
    force_magnitude = np.sqrt(
        self.latest_force.wrench.force.x**2 +
        self.latest_force.wrench.force.y**2 +
        self.latest_force.wrench.force.z**2
    )

    # 速度コマンドの設定
    twist = Twist()

    # 力に応じて速度を調整
    max_velocity = 0.05  # m/s
    force_threshold = 5.0  # N

    if force_magnitude < force_threshold:
        # 力が閾値未満の場合、目標位置に向かって移動
        velocity_scale = min(distance, 1.0)  # 距離に応じて速度をスケーリング

        twist.linear.x = direction[0] * max_velocity * velocity_scale
        twist.linear.y = direction[1] * max_velocity * velocity_scale
        twist.linear.z = direction[2] * max_velocity * velocity_scale
    else:
        # 力が閾値以上の場合、力の方向と反対方向に移動
        force_direction = np.array([
            self.latest_force.wrench.force.x,
            self.latest_force.wrench.force.y,
            self.latest_force.wrench.force.z
        ])
```

```
            force_direction = force_direction / np.linalg.norm(force_direction)

            twist.linear.x = -force_direction[0] * max_velocity * 0.5
            twist.linear.y = -force_direction[1] * max_velocity * 0.5
            twist.linear.z = -force_direction[2] * max_velocity * 0.5

        # 速度コマンドを送信
        self.velocity_pub.publish(twist)

    def feedback_loop(self, event=None):
        # 視覚フィードバック
        if self.task_state["is_tracking"] and self.target_object_pose is not None:
            self.track_object()

        # 力覚フィードバック
        if self.task_state["is_force_controlled"] and self.latest_force is not None:
            self.force_control()

        # 音声フィードバック
        if self.task_state["is_speech_controlled"] and self.latest_speech is not None:
            # （実際のコードでは、音声フィードバックに基づく処理が必要）
            pass

    def set_tracking_target(self, object_name):
        # 追跡対象を設定
        self.task_state["is_tracking"] = True
        self.task_state["current_task"] = f"tracking_{object_name}"

        # （実際のコードでは、object_nameに対応するトピックを購読するための処理が必要）

    def set_force_control_target(self, target_pose):
        # 力制御の目標位置を設定
        self.task_state["is_force_controlled"] = True
        self.task_state["target_pose"] = target_pose
        self.task_state["current_task"] = "force_control"

    def enable_speech_control(self):
        # 音声制御を有効化
        self.task_state["is_speech_controlled"] = True
        self.task_state["current_task"] = "speech_control"
```

このコードは、視覚、力覚、音声フィードバックを統合したフィードバックループを実装している。視覚フィードバックでは、対象物体の位置を追跡し、それに向かって移動する。力覚フィードバックでは、力の大きさと方向に応じて速度を調整する。音声フィードバックでは、ユーザーの音声コマンドに応じてタスクを制御する。これらのフィードバックを統合することで、環境の変化や不確実性に適応しながら、タスクを継続して実行することができる。

### マルチモーダル統合モジュール
マルチモーダル統合モジュールは、視覚、力覚、音声などの異なるモダリティからの情報を統合し、より豊かな環境理解と適応的な行動生成を実現する役割を担う。

#### モダリティ間の情報統合
マルチモーダル統合モジュールは、各モダリティからの情報を適切に重み付けし、統合する機構を提供する。例えば、視覚情報から物体の位置を特定し、力覚情報から物体の重さを推定し、これらの情報を統合して適切な把持力と位置を決定する。また、音声情報からユーザーの意図を理解し、視覚情報と組み合わせて適切な対象物体を特定する。

モダリティ間の情報統合は、以下のようなPythonコードで実装される：

```python
import numpy as np
import rospy
from geometry_msgs.msg import Pose, WrenchStamped
from std_msgs.msg import String, Float64MultiArray
from sensor_msgs.msg import Image
from cv_bridge import CvBridge

class MultimodalIntegrator:
    def __init__(self):
        # ROS関連の設定
        self.vision_sub = rospy.Subscriber("/vision/object_poses", Pose, self.vision_callback)
        self.force_sub = rospy.Subscriber("/force_torque_sensor/calibrated", WrenchStamped, self.force_callback)
        self.speech_sub = rospy.Subscriber("/speech/recognized", String, self.speech_callback)

        self.integrated_info_pub = rospy.Publisher("/multimodal/integrated_info", Float64MultiArray, queue_size=10)

        # 最新のモーダル情報
        self.latest_vision = None
        self.latest_force = None
        self.latest_speech = None

        # 統合情報
        self.integrated_info = {
            "object_position": None,
            "object_weight": None,
            "user_intent": None,
            "confidence": {
                "vision": 0.0,
                "force": 0.0,
                "speech": 0.0
            }
        }

        # モダリティの重み
        self.modality_weights = {
            "vision": 0.5,
            "force": 0.3,
            "speech": 0.2
        }

        # 統合タイマー
        self.timer = rospy.Timer(rospy.Duration(0.1), self.integrate_modalities)

    def vision_callback(self, msg):
        # 最新の視覚情報を保存
        self.latest_vision = msg

        # 視覚情報から物体の位置を抽出
        object_position = np.array([
            msg.position.x,
            msg.position.y,
            msg.position.z
        ])

        # 統合情報を更新
        self.integrated_info["object_position"] = object_position
        self.integrated_info["confidence"]["vision"] = 0.8  # 視覚情報の信頼度（例）
```

```
def force_callback(self, msg):
    # 最新の力覚情報を保存
    self.latest_force = msg

    # 力覚情報から物体の重さを推定
    force_magnitude = np.sqrt(
        msg.wrench.force.x**2 +
        msg.wrench.force.y**2 +
        msg.wrench.force.z**2
    )

    # 重力加速度を考慮して重さを推定
    estimated_weight = force_magnitude / 9.8  # kg

    # 統合情報を更新
    self.integrated_info["object_weight"] = estimated_weight
    self.integrated_info["confidence"]["force"] = 0.7  # 力覚情報の信頼度（例）

def speech_callback(self, msg):
    # 最新の音声情報を保存
    self.latest_speech = msg.data

    # 音声情報からユーザーの意図を抽出
    user_intent = self.extract_user_intent(msg.data)

    # 統合情報を更新
    self.integrated_info["user_intent"] = user_intent
    self.integrated_info["confidence"]["speech"] = 0.6  # 音声情報の信頼度（例）

def extract_user_intent(self, speech_text):
    # 音声テキストからユーザーの意図を抽出
    intent = {
        "action": None,
        "object": None,
        "location": None,
        "modifier": None
    }

    # 簡単な意図抽出（実際のコードでは、より高度な自然言語処理が必要）
    if "取って" in speech_text or "持って" in speech_text:
        intent["action"] = "pick"
    elif "置いて" in speech_text:
        intent["action"] = "place"
    elif "注いで" in speech_text:
        intent["action"] = "pour"

    if "カップ" in speech_text or "マグ" in speech_text:
        intent["object"] = "mug"
    elif "ケトル" in speech_text:
        intent["object"] = "kettle"
    elif "スプーン" in speech_text:
        intent["object"] = "spoon"

    if "テーブル" in speech_text:
        intent["location"] = "table"
    elif "引き出し" in speech_text:
        intent["location"] = "drawer"
```

```
        if "ゆっくり" in speech_text:
            intent["modifier"] = "slow"
        elif "速く" in speech_text:
            intent["modifier"] = "fast"

        return intent

    def integrate_modalities(self, event=None):
        # 各モダリティの情報を統合

        # 統合された情報を公開
        integrated_data = []

        if self.integrated_info["object_position"] is not None:
            integrated_data.extend(self.integrated_info["object_position"])
        else:
            integrated_data.extend([0.0, 0.0, 0.0])

        if self.integrated_info["object_weight"] is not None:
            integrated_data.append(self.integrated_info["object_weight"])
        else:
            integrated_data.append(0.0)

        # ユーザーの意図を数値化
        if self.integrated_info["user_intent"] is not None:
            action_code = 0.0
            if self.integrated_info["user_intent"]["action"] == "pick":
                action_code = 1.0
            elif self.integrated_info["user_intent"]["action"] == "place":
                action_code = 2.0
            elif self.integrated_info["user_intent"]["action"] == "pour":
                action_code = 3.0

            integrated_data.append(action_code)
        else:
            integrated_data.append(0.0)

        # 信頼度情報を追加
        integrated_data.append(self.integrated_info["confidence"]["vision"])
        integrated_data.append(self.integrated_info["confidence"]["force"])
        integrated_data.append(self.integrated_info["confidence"]["speech"])

        # 統合情報を公開
        msg = Float64MultiArray()
        msg.data = integrated_data
        self.integrated_info_pub.publish(msg)

    def update_modality_weights(self, vision_weight, force_weight, speech_weight):
        # モダリティの重みを更新
        total_weight = vision_weight + force_weight + speech_weight

        if total_weight > 0:
            self.modality_weights["vision"] = vision_weight / total_weight
            self.modality_weights["force"] = force_weight / total_weight
            self.modality_weights["speech"] = speech_weight / total_weight
        else:
            rospy.logwarn("Total weight is zero or negative. Weights not updated.")
```

このコードは、視覚、力覚、音声の各モダリティからの情報を統合し、より豊かな環境理解を実現するマルチモーダル統合モジュールを実装している。視覚情報から物体の位置を抽出し、力覚情報から物体の重さを

推定し、音声情報からユーザーの意図を抽出する。これらの情報を統合し、各モダリティの信頼度を考慮して、より正確な環境理解を実現する。

#### クロスモーダル学習
クロスモーダル学習は、異なるモダリティ間の関係を学習し、情報の相互補完を実現するプロセスである。例えば、視覚情報と力覚情報の関係を学習することで、視覚情報から物体の重さを推定したり、力覚情報から物体の形状を推定したりすることができる。また、音声情報と視覚情報の関係を学習することで、音声指示から対象物体を特定することができる。

クロスモーダル学習は、以下のようなPythonコードで実装される：

```python
import numpy as np
import rospy
from geometry_msgs.msg import Pose, WrenchStamped
from std_msgs.msg import String, Float64MultiArray
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.svm import SVR

class CrossModalLearner:
    def __init__(self):
        # ROS関連の設定
        self.vision_sub = rospy.Subscriber("/vision/object_poses", Pose, self.vision_callback)
        self.force_sub = rospy.Subscriber("/force_torque_sensor/calibrated", WrenchStamped, self.force_callback)
        self.speech_sub = rospy.Subscriber("/speech/recognized", String, self.speech_callback)

        self.predicted_weight_pub = rospy.Publisher("/multimodal/predicted_weight", Float64MultiArray, queue_size=10)

        # 学習データ
        self.vision_data = []
        self.force_data = []
        self.speech_data = []

        # 学習モデル
        self.vision_to_force_model = RandomForestRegressor()
        self.force_to_vision_model = RandomForestRegressor()
        self.speech_to_vision_model = None  # 自然言語処理には別のアプローチが必要

        # 学習済みフラグ
        self.is_trained = False

        # 学習タイマー
        self.timer = rospy.Timer(rospy.Duration(60.0), self.train_models)  # 60秒ごとに学習

    def vision_callback(self, msg):
        # 視覚情報をデータセットに追加
        vision_features = [
            msg.position.x,
            msg.position.y,
            msg.position.z,
            msg.orientation.x,
            msg.orientation.y,
            msg.orientation.z,
            msg.orientation.w
        ]

        self.vision_data.append(vision_features)
```

```
    # 学習済みの場合、視覚情報から力を予測
    if self.is_trained:
        predicted_force = self.predict_force_from_vision(vision_features)
        self.publish_predicted_weight(predicted_force)

def force_callback(self, msg):
    # 力覚情報をデータセットに追加
    force_features = [
        msg.wrench.force.x,
        msg.wrench.force.y,
        msg.wrench.force.z,
        msg.wrench.torque.x,
        msg.wrench.torque.y,
        msg.wrench.torque.z
    ]

    self.force_data.append(force_features)

def speech_callback(self, msg):
    # 音声情報をデータセットに追加
    # （実際のコードでは、テキストを特徴ベクトルに変換する処理が必要）
    self.speech_data.append(msg.data)

def train_models(self, event=None):
    # データが十分にある場合、モデルを学習
    if len(self.vision_data) > 10 and len(self.force_data) > 10:
        rospy.loginfo("Training cross-modal models...")

        # 最新のデータを使用
        recent_vision_data = np.array(self.vision_data[-100:])
        recent_force_data = np.array(self.force_data[-100:])

        # 視覚情報から力覚情報を予測するモデルを学習
        self.vision_to_force_model.fit(recent_vision_data, recent_force_data)

        # 力覚情報から視覚情報を予測するモデルを学習
        self.force_to_vision_model.fit(recent_force_data, recent_vision_data)

        self.is_trained = True
        rospy.loginfo("Cross-modal models trained successfully.")
    else:
        rospy.logwarn("Not enough data to train cross-modal models.")

def predict_force_from_vision(self, vision_features):
    # 視覚情報から力覚情報を予測
    vision_features_array = np.array([vision_features])
    predicted_force = self.vision_to_force_model.predict(vision_features_array)[0]

    return predicted_force

def predict_vision_from_force(self, force_features):
    # 力覚情報から視覚情報を予測
    force_features_array = np.array([force_features])
    predicted_vision = self.force_to_vision_model.predict(force_features_array)[0]

    return predicted_vision

def publish_predicted_weight(self, predicted_force):
    # 予測された力から重さを計算
    force_magnitude = np.sqrt(
        predicted_force[0]**2 +
```

```
            predicted_force[1]**2 +
            predicted_force[2]**2
        )

        # 重力加速度を考慮して重さを推定
        estimated_weight = force_magnitude / 9.8  # kg

        # 予測された重さを公開
        msg = Float64MultiArray()
        msg.data = [estimated_weight]
        self.predicted_weight_pub.publish(msg)
```

このコードは、視覚情報と力覚情報の関係を学習し、一方から他方を予測するクロスモーダル学習を実装している。視覚情報から力覚情報を予測するモデルと、力覚情報から視覚情報を予測するモデルを学習する。これにより、一方のモダリティの情報が欠損した場合でも、他方のモダリティから補完することができる。

#### マルチモーダル推論

マルチモーダル推論は、統合された情報に基づいて、環境や状況に関する推論を行うプロセスである。例えば、視覚情報と力覚情報を統合して、物体の材質や中身を推定したり、視覚情報と音声情報を統合して、ユーザーの意図を推定したりする。

マルチモーダル推論は、以下のようなPythonコードで実装される：

```python
import numpy as np
import rospy
from geometry_msgs.msg import Pose, WrenchStamped
from std_msgs.msg import String, Float64MultiArray
from sklearn.ensemble import RandomForestClassifier

class MultimodalReasoner:
    def __init__(self):
        # ROS関連の設定
        self.integrated_info_sub = rospy.Subscriber("/multimodal/integrated_info", Float64MultiArray,
self.integrated_info_callback)

        self.object_property_pub = rospy.Publisher("/multimodal/object_property", String, queue_size=10)
        self.user_intent_pub = rospy.Publisher("/multimodal/user_intent", String, queue_size=10)

        # 物体の特性分類器
        self.material_classifier = RandomForestClassifier()
        self.content_classifier = RandomForestClassifier()

        # 物体の特性データベース
        self.material_database = {
            "plastic": {"weight_range": [0.05, 0.2], "hardness": "medium"},
            "ceramic": {"weight_range": [0.2, 0.5], "hardness": "high"},
            "metal": {"weight_range": [0.3, 1.0], "hardness": "high"},
            "glass": {"weight_range": [0.2, 0.6], "hardness": "high"},
            "wood": {"weight_range": [0.1, 0.4], "hardness": "medium"}
        }

        self.content_database = {
            "empty": {"weight_change": [0.0, 0.1]},
            "water": {"weight_change": [0.1, 1.0]},
            "coffee": {"weight_change": [0.1, 0.3]},
            "sugar": {"weight_change": [0.1, 0.2]}
        }

        # ユーザー意図の解釈ルール
```

```
        self.intent_rules = {
            "pick": {
                "mug": "pick up the mug",
                "kettle": "pick up the kettle",
                "spoon": "pick up the spoon"
            },
            "place": {
                "mug": "place the mug",
                "kettle": "place the kettle",
                "spoon": "place the spoon"
            },
            "pour": {
                "kettle": "pour water from the kettle"
            }
        }

    def integrated_info_callback(self, msg):
        # 統合情報を解析
        integrated_data = msg.data

        # 物体の位置
        object_position = integrated_data[0:3]

        # 物体の重さ
        object_weight = integrated_data[3]

        # ユーザーの意図
        action_code = integrated_data[4]

        # 信頼度
        vision_confidence = integrated_data[5]
        force_confidence = integrated_data[6]
        speech_confidence = integrated_data[7]

        # 物体の材質を推定
        material = self.infer_material(object_weight)

        # 物体の中身を推定
        content = self.infer_content(object_weight)

        # ユーザーの意図を解釈
        user_intent = self.interpret_user_intent(action_code)

        # 推論結果を公開
        self.publish_object_property(material, content)
        self.publish_user_intent(user_intent)

    def infer_material(self, weight):
        # 重さから物体の材質を推定
        for material, properties in self.material_database.items():
            if properties["weight_range"][0] <= weight <= properties["weight_range"][1]:
                return material

        return "unknown"

    def infer_content(self, weight):
        # 重さの変化から物体の中身を推定
        # （実際のコードでは、重さの変化を追跡する必要がある）
        weight_change = 0.2  # 例
```

```
        for content, properties in self.content_database.items():
            if properties["weight_change"][0] <= weight_change <= properties["weight_change"][1]:
                return content

        return "unknown"

    def interpret_user_intent(self, action_code):
        # アクションコードからユーザーの意図を解釈
        action = "unknown"
        if action_code == 1.0:
            action = "pick"
        elif action_code == 2.0:
            action = "place"
        elif action_code == 3.0:
            action = "pour"

        object_type = "mug"  # 例（実際のコードでは、物体の種類を推定する必要がある）

        if action in self.intent_rules and object_type in self.intent_rules[action]:
            return self.intent_rules[action][object_type]
        else:
            return "unknown intent"

    def publish_object_property(self, material, content):
        # 物体の特性を公開
        msg = String()
        msg.data = f"Material: {material}, Content: {content}"
        self.object_property_pub.publish(msg)

    def publish_user_intent(self, user_intent):
        # ユーザーの意図を公開
        msg = String()
        msg.data = user_intent
        self.user_intent_pub.publish(msg)
```

このコードは、統合された情報に基づいて、物体の材質や中身、ユーザーの意図などを推論するマルチモーダル推論を実装している。物体の重さから材質を推定し、重さの変化から中身を推定する。また、アクションコードと物体の種類からユーザーの意図を解釈する。これにより、より豊かな環境理解と適応的な行動生成が可能となる。

## 8. 産業上の利用可能性

本発明は、以下の産業分野での利用が期待される：

### 1. 家庭用サービスロボット

家庭環境は予測不可能で多様であり、ロボットが様々なタスクを実行するためには、高い適応能力と知能が必要である。本システムは、以下のような家庭用サービスロボットの応用に適している：

1. 家事支援：
   - 調理補助：食材の準備、調理プロセスの支援、食器の配置など
   - 清掃：床掃除、表面の拭き取り、物の整理整頓など
   - 洗濯：洗濯物の仕分け、洗濯機への投入、乾燥した衣類の折りたたみなど
   - 食器洗い：食器の収集、洗浄、収納など

2. 高齢者・障害者支援：

- 日常生活支援：食事、着替え、入浴などの支援
- 薬の管理：適切なタイミングでの薬の提供、残量の管理など
- 移動支援：物の取得、ドアの開閉、移動の補助など
- 安全監視：転倒検知、異常行動の検出、緊急時の通報など

3. 子育て支援：
- おもちゃの片付け：散らかったおもちゃの収集と収納
- 教育支援：対話型の学習支援、知育玩具の操作など
- 安全監視：危険な行動の検出、危険物の除去など

本システムの高レベル指示理解能力と環境適応能力は、家庭環境の多様性と予測不可能性に対応するのに適している。また、視覚および力覚フィードバックの統合により、繊細な操作（例：食器の扱い、衣類の折りたたみ）も可能となる。さらに、音声対話機能により、ユーザーとの自然なインタラクションが実現され、使いやすさが向上する。

例えば、高齢者の自宅では、「お薬の時間ですよ。薬を持ってきましょうか？」と声をかけ、承諾を得てから薬と水を準備し、服薬を支援することができる。また、「今日の夕食は何がいいですか？」と尋ね、希望に応じて調理の準備を手伝うことができる。このような自然な対話と適応的な支援は、高齢者の自立した生活を支える上で重要である。

### 2. 製造業
製造業では、多品種少量生産や柔軟な生産ラインの需要が高まっており、適応能力の高いロボットシステムが求められている。本システムは、以下のような製造業の応用に適している：

1. 組立作業：
- 複雑な部品の組立：形状や配置が異なる部品の組立
- 精密組立：微細な部品の位置合わせと組立
- フレキシブル組立：製品バリエーションに応じた組立プロセスの適応

2. 品質検査：
- 視覚検査：製品の外観検査、欠陥検出など
- 機能検査：製品の動作確認、性能テストなど
- 寸法測定：製品の寸法精度の確認など

3. 柔軟な生産ライン：
- 製品切り替え：異なる製品への迅速な切り替え
- プロセス適応：製造プロセスの変更への適応
- 協働作業：人間との協働による複雑なタスクの実行

本システムのRAGを用いた知識ベースの効率的な活用により、多様な製品や製造プロセスに対応することができる。また、視覚および力覚フィードバックの統合により、精密な組立や検査が可能となる。さらに、音声対話機能により、作業者との円滑なコミュニケーションが実現され、協働作業の効率が向上する。

例えば、自動車部品の組立ラインでは、「次の製品はモデルBの組立です。準備はいいですか？」と作業者に確認し、必要な部品や工具を準備することができる。また、「この部品の取り付け方を教えてください」という作業者の質問に対して、手順を説明したり、デモンストレーションを行ったりすることができる。このような柔軟な対応と協働作業は、多品種少量生産や複雑な組立作業において重要である。

### 3. 医療・ヘルスケア

医療・ヘルスケア分野では、高い精度と安全性が求められるとともに、患者ごとに異なる状況に適応する必要がある。本システムは、以下のような医療・ヘルスケアの応用に適している：

1. 手術支援：
 - 器具の受け渡し：外科医に適切な器具を適切なタイミングで提供
 - 視野の確保：内視鏡やカメラの位置調整による視野の確保
 - 組織の保持：手術中の組織の安定した保持

2. リハビリテーション支援：
 - 運動支援：患者の状態に応じた運動支援
 - 進捗モニタリング：リハビリの進捗状況の記録と分析
 - フィードバック提供：患者へのリアルタイムフィードバックの提供

3. 患者ケア：
 - 食事支援：患者の状態に応じた食事の提供
 - 投薬管理：適切なタイミングでの薬の提供
 - 移動支援：ベッドからの起き上がり、歩行の補助など

本システムの高い安全性と適応能力は、患者ごとに異なる状況に対応するのに適している。また、視覚および力覚フィードバックの統合により、繊細な操作（例：器具の受け渡し、患者との物理的相互作用）も可能となる。さらに、音声対話機能により、患者との自然なコミュニケーションが実現され、ケアの質が向上する。

例えば、リハビリテーション施設では、「今日の調子はいかがですか？」と患者に尋ね、状態に応じてリハビリプログラムを調整することができる。また、「もう少し力を入れてみましょう」「その調子です、素晴らしい」などの声かけを行いながら、患者の運動をサポートすることができる。このような個別化されたケアと励ましは、リハビリテーションの効果を高める上で重要である。

### 4. 小売・サービス業
小売・サービス業では、顧客との対話や商品の取り扱いなど、多様なタスクが求められる。本システムは、以下のような小売・サービス業の応用に適している：

1. 商品管理：
 - 商品陳列：商品の適切な配置と整理
 - 在庫確認：棚卸し、在庫状況の確認など
 - 商品補充：在庫切れ商品の補充など

2. 顧客対応：
 - 案内：店舗内の案内、商品の場所の案内など
 - 情報提供：商品情報の提供、質問への回答など
 - 注文受付：顧客の注文の受付と処理など

3. フードサービス：
 - 調理補助：食材の準備、調理プロセスの支援など
 - 盛り付け：料理の盛り付け、装飾など
 - 配膳：料理の運搬、テーブルへの配置など

本システムの自然言語理解能力と創造的な動作生成能力は、顧客との対話や芸術的な盛り付けなどに適している。また、視覚および力覚フィードバックの統合により、繊細な商品の取り扱いや料理の盛り付けが可能

となる。さらに、音声対話機能により、顧客との自然なコミュニケーションが実現され、サービスの質が向上する。

例えば、カフェでは、「いらっしゃいませ。何をお探しですか？」と顧客に声をかけ、希望に応じてメニューを説明したり、おすすめを提案したりすることができる。また、「このコーヒーはどのように淹れられていますか？」という質問に対して、コーヒーの淹れ方や特徴を説明することができる。さらに、ラテアートなどの創造的な装飾を施したドリンクを提供することもできる。このような個別化されたサービスと創造的な表現は、顧客満足度を高める上で重要である。

### 5. 教育・研究
教育・研究分野では、柔軟な対応と創造的な問題解決が求められる。本システムは、以下のような教育・研究の応用に適している：

1. 実験支援：
   - 実験準備：試薬の準備、装置のセットアップなど
   - データ収集：実験データの記録、サンプルの収集など
   - 実験実行：定型的な実験手順の自動化など

2. 教育用ロボット：
   - 対話型学習：質問への回答、学習内容の説明など
   - 実演：実験や操作の実演、手順の説明など
   - 評価：学習者の理解度の評価、フィードバックの提供など

3. 研究開発支援：
   - プロトタイピング：新しいアイデアの迅速な実装と検証
   - データ分析：実験データの処理と分析
   - 文献調査：関連研究の検索と要約など

本システムのRAGを用いた知識ベースの効率的な活用により、多様な実験や研究テーマに対応することができる。また、創造的な動作生成能力は、新しい実験手法の開発や教育コンテンツの作成に役立つ。さらに、音声対話機能により、学習者との自然なコミュニケーションが実現され、学習効果が向上する。

例えば、化学実験室では、「この実験の手順を教えてください」という学生の質問に対して、手順を説明しながら実演することができる。また、「この反応はなぜ起こるのですか？」という質問に対して、化学反応のメカニズムを説明することができる。さらに、「この実験の結果を分析してください」という要求に対して、データを収集し、統計的な分析を行うことができる。このような対話型の学習支援と実践的な実演は、学習者の理解を深める上で重要である。

### 6. 災害対応・危険環境
災害現場や危険環境では、人間が立ち入ることが困難な場所での作業が必要となる。本システムは、以下のような災害対応・危険環境の応用に適している：

1. 災害救助：
   - 探索：被災者の探索、状況の確認など
   - 救助：瓦礫の除去、被災者の救出など
   - 物資輸送：食料、水、医薬品などの輸送

2. 危険物処理：
   - 検査：危険物の検出、状態の確認など
   - 回収：危険物の安全な回収と保管

- 処理：危険物の無害化、処分など

3. 極限環境での作業：
- 深海作業：海底での調査、機器の設置など
- 宇宙作業：宇宙ステーションでの実験、修理など
- 原子力施設：放射線環境下での点検、修理など

本システムの高い適応能力と自律性は、予測困難な災害現場や危険環境での作業に適している。また、視覚および力覚フィードバックの統合により、複雑な環境での操作や精密な作業が可能となる。さらに、音声対話機能により、遠隔操作者との円滑なコミュニケーションが実現され、作業の効率と安全性が向上する。

例えば、原子力施設の事故現場では、「この配管の状態を確認してください」という遠隔操作者の指示に対して、配管を視覚的に検査し、損傷の有無や程度を報告することができる。また、「このバルブを慎重に閉めてください」という指示に対して、適切な力でバルブを操作することができる。このような遠隔作業と状況報告は、人間が立ち入ることが危険な環境での作業において重要である。

### 7. エンターテイメント・芸術
エンターテイメント・芸術分野では、創造性と表現力が求められる。本システムは、以下のようなエンターテイメント・芸術の応用に適している：

1. パフォーマンス：
- ダンス：音楽に合わせた動きの生成と実行
- 演劇：感情表現、対話、物語の進行など
- 音楽：楽器の演奏、歌唱など

2. 創作活動：
- 描画：絵画、イラスト、グラフィックデザインなど
- 彫刻：3次元造形、立体アートなど
- 工芸：陶芸、織物、木工など

3. インタラクティブアート：
- 観客参加型作品：観客の動きや声に反応する作品
- メディアアート：デジタル技術を活用した芸術表現
- 環境アート：空間全体を活用した芸術体験

本システムの創造的な動作生成能力と自然言語理解能力は、芸術的な表現や観客とのインタラクションに適している。また、視覚および力覚フィードバックの統合により、繊細な芸術的操作（例：絵画、彫刻）が可能となる。さらに、音声対話機能により、観客との自然なコミュニケーションが実現され、パフォーマンスの没入感が向上する。

例えば、インタラクティブアート展示では、「あなたの好きな色は何ですか？」と観客に尋ね、回答に基づいて即興の絵を描くことができる。また、「どんな気分ですか？」という質問に対して、観客の感情に合わせた音楽や動きを生成することができる。このような観客との対話と即興的な創作は、芸術体験の個別化と没入感を高める上で重要である。

これらの産業分野において、本システムは従来のロボットシステムでは困難であった複雑なタスクの実行を可能にし、生産性の向上、安全性の確保、新しいサービスの創出などに貢献することが期待される。特に、予測不可能な環境下での適応能力と、高レベルの抽象的指示の理解能力は、様々な応用シナリオで価値を生み出すことができる。また、マルチモーダルフィードバックの統合と活用により、より豊かな環境理解と適

応的な行動生成が実現され、人間との自然なインタラクションが可能となる。これにより、ロボットは単なる道具ではなく、人間の協働パートナーとして機能することができる。

### 8. 農業・食品産業
農業・食品産業では、環境条件の変化や作物・食材の個体差に対応する必要がある。本システムは、以下のような農業・食品産業の応用に適している：

1. 精密農業：
  - 作物管理：作物の状態監視、水やり、施肥など
  - 収穫：熟度判断、適切な力での収穫、選別など
  - 環境制御：温度、湿度、光量などの最適化

2. 食品加工：
  - 原材料処理：洗浄、皮むき、切断、分離など
  - 調理：混合、加熱、冷却、発酵管理など
  - 包装：適切な量の充填、シール、ラベリングなど

3. 品質管理：
  - 外観検査：形状、色、傷の検出など
  - 物性測定：硬さ、粘度、水分含有量など
  - 成分分析：栄養素、添加物、異物の検出など

本システムの視覚および力覚フィードバックの統合により、作物や食材の状態に応じた適切な処理が可能となる。また、RAGを用いた知識ベースの効率的な活用により、多様な作物や食材に対応することができる。さらに、環境の変化や不確実性への適応能力により、屋外環境や変動する条件下でも安定した作業が可能となる。

例えば、トマト農園では、「これらのトマトの熟度を確認し、収穫可能なものを選んでください」という指示に対して、視覚情報から熟度を判断し、適切な力で収穫することができる。また、「この区画の水やりが必要かどうか判断してください」という指示に対して、土壌の状態や天候条件を考慮して判断することができる。このような精密な管理と判断は、農業の生産性と品質向上に貢献する。

食品加工工場では、「この野菜を均一な大きさに切り分けてください」という指示に対して、視覚情報から野菜の形状を認識し、適切な切断位置と力を調整することができる。また、「このソースの粘度を確認し、必要に応じて調整してください」という指示に対して、力覚フィードバックから粘度を推定し、調整することができる。このような精密な加工と品質管理は、食品の一貫性と安全性確保に重要である。

### 9. 物流・倉庫管理
物流・倉庫管理分野では、多様な形状や重量の商品を扱い、効率的な保管と出荷が求められる。本システムは、以下のような物流・倉庫管理の応用に適している：

1. 商品ピッキング：
  - 商品識別：バーコード、QRコード、外観による商品の識別
  - 適応的把持：商品の形状、重量、硬さに応じた把持方法の調整
  - 梱包：商品の適切な配置、緩衝材の挿入、箱の封緘など

2. 在庫管理：
  - 棚卸し：商品の自動カウント、位置確認、状態チェック
  - 商品配置：効率的な保管レイアウトの計画と実行
  - 在庫補充：在庫切れの検出、補充オーダーの生成

3. 物流最適化：
  - ルート計画：効率的な移動経路の計画と実行
  - 積載最適化：トラックや配送箱への効率的な商品配置
  - 配送準備：配送順序に応じた商品の仕分けと準備

本システムの視覚および力覚フィードバックの統合により、多様な商品の適切な取り扱いが可能となる。また、RAGを用いた知識ベースの効率的な活用により、新しい商品や梱包方法にも対応することができる。さらに、環境の変化や不確実性への適応能力により、混雑した倉庫環境や変動する注文パターンにも対応できる。

例えば、Eコマース倉庫では、「この注文リストの商品をピッキングして梱包してください」という指示に対して、各商品を識別し、適切な方法で把持して梱包することができる。また、「この棚の在庫状況を確認してください」という指示に対して、商品をスキャンし、在庫数と位置を報告することができる。このような効率的なピッキングと在庫管理は、物流コストの削減と顧客満足度の向上に貢献する。

### 10. スマートシティ・公共サービス
スマートシティや公共サービス分野では、市民の多様なニーズに対応し、安全で効率的な都市環境を維持する必要がある。本システムは、以下のようなスマートシティ・公共サービスの応用に適している：

1. 公共空間管理：
  - 清掃：道路、公園、公共施設などの清掃
  - 点検：インフラ設備、公共施設の状態点検
  - 修繕：小規模な修繕作業、メンテナンス

2. 市民サービス：
  - 案内：観光案内、施設案内、道案内など
  - 情報提供：イベント情報、交通情報、天気情報など
  - 支援：高齢者や障害者の移動支援、荷物運搬など

3. 安全管理：
  - 監視：異常行動の検出、危険状況の認識
  - 通報：緊急事態の通報、関係機関への連絡
  - 初期対応：初期消火、避難誘導、応急処置など

本システムの高レベル指示理解能力と環境適応能力は、多様な公共空間と市民ニーズに対応するのに適している。また、視覚および力覚フィードバックの統合により、繊細な作業（例：ゴミの分別、設備点検）も可能となる。さらに、音声対話機能により、市民との自然なコミュニケーションが実現され、サービスの質が向上する。

例えば、公共施設では、「この建物の案内をしてください」という来訪者の要求に対して、施設の概要を説明し、目的の場所まで案内することができる。また、「この設備の状態を点検してください」という管理者の指示に対して、設備を視覚的に検査し、異常の有無を報告することができる。このような対話型の案内と点検は、公共サービスの効率化と質の向上に貢献する。

公園や広場では、「このエリアの清掃をしてください」という指示に対して、ゴミを識別し、適切な方法で回収することができる。また、「この噴水の水圧を確認してください」という指示に対して、噴水の状態を観察し、異常があれば報告することができる。このような日常的な管理と点検は、公共空間の美観と安全性の維持に重要である。

### 11. 宇宙探査・極限環境
宇宙探査や極限環境での作業では、高い自律性と適応能力が求められる。本システムは、以下のような宇宙探査・極限環境の応用に適している：

1. 宇宙探査：
   - 惑星表面探査：地形調査、サンプル採取、実験実施
   - 宇宙ステーション作業：機器の設置、修理、メンテナンス
   - 宇宙建設：構造物の組立、資源利用施設の建設

2. 深海探査：
   - 海底調査：地形マッピング、生態系観察、資源探査
   - 海底作業：ケーブル敷設、設備設置、サンプル採取
   - 海中構造物点検：油田プラットフォーム、橋脚、パイプラインの点検

3. 極地探査：
   - 氷床調査：氷層構造分析、コア採取、気象観測
   - 極地基地支援：物資運搬、設備メンテナンス、研究支援
   - 環境モニタリング：生態系観察、汚染物質検出、気候変動データ収集

本システムの高い自律性と適応能力は、通信遅延や制限された環境での作業に適している。また、視覚および力覚フィードバックの統合により、未知の環境での安全な操作が可能となる。さらに、RAGを用いた知識ベースの効率的な活用により、予期せぬ状況にも対応することができる。

例えば、火星探査では、「この岩石のサンプルを採取してください」という地球からの指示に対して、岩石の硬さや形状に応じた適切な方法でサンプルを採取することができる。また、「この地形を調査し、安全な経路を見つけてください」という指示に対して、周囲の地形を分析し、障害物を避けながら安全に移動することができる。このような自律的な判断と適応は、通信遅延のある遠隔環境での探査に不可欠である。

宇宙ステーションでは、「この機器の修理をサポートしてください」という宇宙飛行士の要求に対して、必要な工具を準備し、修理手順を説明しながら作業を支援することができる。また、「この実験装置をセットアップしてください」という指示に対して、装置の各部品を適切に配置し、接続することができる。このような精密な作業と支援は、限られたリソースと厳しい安全要件のある宇宙環境で重要である。

### 12. 教育・訓練シミュレーション
教育・訓練シミュレーション分野では、リアルな環境と状況を再現し、効果的な学習体験を提供する必要がある。本システムは、以下のような教育・訓練シミュレーションの応用に適している：

1. 医療訓練：
   - 手術シミュレーション：外科手術の手順、技術の練習
   - 救急対応訓練：緊急医療処置、トリアージの実践
   - 患者ケア訓練：診察、看護、リハビリテーション技術の習得

2. 技術訓練：
   - 工業技術：機械操作、組立、修理、検査の訓練
   - 建設技術：重機操作、測量、施工技術の習得
   - 科学実験：化学実験、物理実験、生物実験の実践

3. 災害対応訓練：
   - 避難誘導：災害時の避難計画、誘導技術の訓練
   - 救助活動：被災者の探索、救出、応急処置の実践

- 危機管理：指揮系統、情報収集、意思決定の訓練

本システムの視覚および力覚フィードバックの統合により、リアルな触感と視覚体験を提供することができる。また、自然言語理解能力により、学習者との対話型の指導が可能となる。さらに、環境の変化や不確実性への適応能力により、様々なシナリオや難易度に対応することができる。

例えば、医学教育では、「この患者の診察を行ってください」という指示に対して、バーチャル患者の症状を表現し、学生の質問に応答することができる。また、「この手術手順を実演してください」という要求に対して、手術の各ステップを説明しながら実演し、学生が実践する際にはフィードバックを提供することができる。このような対話型のシミュレーションと即時フィードバックは、医療技術の習得に効果的である。

災害対応訓練では、「この建物で火災が発生しました。避難誘導を行ってください」というシナリオに対して、火災の拡大や煙の充満などの状況変化を再現し、訓練者の判断と行動に応じて状況を変化させることができる。また、「この瓦礫の下に被災者がいます。救出してください」というシナリオでは、瓦礫の安定性や被災者の状態に応じた適切な救出方法を訓練することができる。このような動的なシミュレーションと状況対応は、実践的な災害対応能力の向上に貢献する。

## 9. 実施例

以下、本発明の具体的な実施例について説明する。なお、以下の実験は、New York General Group社のCategorical AIを使い行われた。Categorical AIは、Anthropic社によって動作するClaude-3.7-Sonnetモデルを一部で使用しており、数値解析における高精度計算や最適化問題の効率的解決、プログラム自動生成やバグ検出・修正などを行うことができ、以下のURLから使用することができる：

https://www.newyorkgeneralgroup.com/ouraimodels

本実施例では、提案したマルチモーダルフィードバック統合型知識検索強化ロボット制御システムの有効性を詳細に検証するために、Gymnasium（旧OpenAI Gym）を用いた高度なシミュレーション実験を行った。Gymnasiumは、強化学習アルゴリズムの開発とベンチマークのための標準的なインターフェースを提供するPythonライブラリであり、様々な環境でのロボット制御タスクをシミュレートすることができる。本実験では、Gymnasiumの機能を大幅に拡張し、マルチモーダルフィードバックと検索強化生成（RAG）を統合した高度なカスタム環境を構築した。

### シミュレーション環境の構築

シミュレーション環境は、実際のキッチン環境を精密に模擬したものであり、「コーヒーを作る」というタスクを実行するための多様なオブジェクト（マグカップ、コーヒー豆、ケトル、スプーン、引き出し、冷蔵庫、食器棚など）が配置されている。環境内のオブジェクトは、位置、姿勢、物理的特性（重さ、摩擦係数、弾性係数、熱伝導率など）を持ち、高度な物理シミュレーションに基づいて相互作用する。また、液体（水、コーヒー）や粉末（コーヒー豆）などの非剛体物質のシミュレーションも実装し、より現実的なタスク実行を可能にした。

環境の構築には、Gymnasiumの基本クラスを継承したカスタム環境クラス「KitchenEnvironment」を実装した。このクラスは、以下の特徴を持つ：

1. 観測空間（Observation Space）：
   - 視覚情報：RGB画像（1920×1080ピクセル、24ビット色深度）と深度マップ（1920×1080ピクセル、16ビット深度）
   - 力覚情報：6次元ベクトル（3次元の力と3次元のトルク）

- 音声情報：テキスト形式の音声認識結果と音響特徴量（周波数スペクトル）
- 環境状態：オブジェクトの位置、姿勢、物理的状態（温度、含水量など）

2. 行動空間（Action Space）：
- ロボットの関節角度（7自由度）
- グリッパーの開閉状態（連続値）
- エンドエフェクタの位置と姿勢（6次元）
- 力制御パラメータ（最大力、コンプライアンスなど）
- 高レベルアクション（「把持する」「持ち上げる」「注ぐ」など）

3. 物理シミュレーション：
- 剛体力学：PyBulletを用いた高精度な剛体シミュレーション
- 流体力学：SPH（Smoothed Particle Hydrodynamics）法を用いた液体シミュレーション
- 熱力学：熱伝導と対流のシミュレーション（コーヒーの温度変化など）
- 接触力学：摩擦、弾性、粘性などの接触特性のシミュレーション

4. 環境の不確実性：
- オブジェクトの初期位置のランダム化（標準偏差5cm）
- 物理特性のばらつき（重さ±10%、摩擦係数±20%など）
- センサーノイズ（視覚：ガウシアンノイズσ=0.01、力覚：ガウシアンノイズσ=0.1N）
- アクチュエータの遅延（10-50ms）と不確実性（指令値の±2%）
- 環境の動的変化（他のエージェントによるオブジェクトの移動など）

5. 報酬関数：
- タスク完了報酬：サブタスクごとの報酬（+10）とタスク全体の完了報酬（+100）
- 効率性報酬：タスク完了時間に反比例する報酬（最大+50）
- 安全性報酬：過度な力や速度に対するペナルティ（最大-50）
- 精度報酬：液体の注入量の精度に比例する報酬（最大+30）

環境は、Python 3.9上で実装され、PyBullet 3.2.1、NumPy 1.22.3、OpenCV 4.6.0などの主要ライブラリを使用している。シミュレーションは、Intel Core i9-12900K CPU、NVIDIA RTX 3090 GPU、64GB RAMを搭載したワークステーションで実行され、リアルタイムファクター0.8（実時間の0.8倍の速度）でシミュレーションが進行する。

### システムの詳細実装

提案システムは、モジュール化されたアーキテクチャで実装され、各コンポーネントは独立して開発・テストされた後、統合された。システムの全体アーキテクチャは、ROS2（Robot Operating System 2）に似た発行-購読（Publish-Subscribe）パターンに基づいており、各コンポーネント間の通信は非同期メッセージングで行われる。主要コンポーネントの詳細実装は以下の通りである：

#### 1. 言語処理コンポーネント

言語処理コンポーネントは、自然言語理解、タスク分解、コード生成、検索強化生成（RAG）の機能を提供する。実装詳細は以下の通りである：

- 言語モデル：GPT-4（OpenAI API経由）を主要な言語モデルとして使用。バックアップとして、ローカルで実行可能なLlama 2（70B）モデルも実装。
- エンベディングモデル：文章のベクトル表現にはOpenAIのtext-embedding-ada-002（1536次元）を使用。

- RAG実装：Langchainフレームワークを用いて実装。ドキュメントチャンキング（チャンクサイズ512トークン、オーバーラップ50トークン）、ベクトル化、Faissを用いた近似最近傍検索を実装。
- プロンプトエンジニアリング：タスク分解、コード生成、エラー処理のための専用プロンプトテンプレートを開発。プロンプトには、システムの状態、環境の記述、過去の行動履歴、エラー情報などが含まれる。
- タスク分解アルゴリズム：階層的タスク分解を実装。高レベルタスクを中レベルタスクに、中レベルタスクを低レベルタスクに分解する3層構造。各レベル間の依存関係は有向非巡回グラフ（DAG）で表現。
- 条件付き確率モデル：サブタスク間の依存関係を条件付き確率P(T_j|T_i)として表現。確率値は知識ベースの例から学習され、ベイジアンネットワークとして実装。
- コード生成：Python AST（Abstract Syntax Tree）を用いた安全なコード生成と検証。生成されたコードは、サンドボックス環境で実行前に静的解析と動的チェックを実施。
- エラー処理：エラー検出、診断、回復のための専用モジュール。エラーパターンのデータベースと、過去のエラー回復事例を活用。

言語処理コンポーネントは、40Hzの周期で環境状態を監視し、必要に応じてタスク計画を更新する。タスク分解とコード生成は非同期で実行され、結果がキューに格納される。システムは、現在の環境状態に最も適したコードを選択し、実行する。

#### 2. 視覚システム

視覚システムは、環境の3次元表現を生成し、オブジェクトの検出、セグメンテーション、ポーズ推定を行う。実装詳細は以下の通りである：

- オブジェクト検出：YOLOv7をベースとしたリアルタイム物体検出器（mAP 0.56@0.5IoU）。30種類のキッチンオブジェクトに対して微調整。検出速度は30FPS。
- インスタンスセグメンテーション：Mask R-CNN（ResNet-101バックボーン）とSegment Anythingモデルを組み合わせたハイブリッドアプローチ。セグメンテーション精度はmIoU 0.78。
- 言語-視覚モジュール：CLIP（Contrastive Language-Image Pretraining）モデルを用いた言語指示に基づくオブジェクト検出。「赤いマグカップ」「大きなケトル」などの属性付き指示に対応。
- 3次元再構成：深度マップとセグメンテーションマスクを組み合わせた3次元点群生成。RANSAC（Random Sample Consensus）アルゴリズムを用いた平面・曲面フィッティング。
- ポーズ推定：ICP（Iterative Closest Point）アルゴリズムと事前学習された形状モデルを用いた6DoFポーズ推定。推定精度は平均で位置±5mm、姿勢±3度。
- 時間的追跡：カルマンフィルタとハンガリアンアルゴリズムを用いたマルチオブジェクト追跡。オクルージョン（遮蔽）に対して最大5秒間の追跡維持。
- 不確実性推定：各検出とポーズ推定に対する確率分布（多変量ガウス分布）の推定。不確実性が高い場合、アクティブ知覚（能動的な視点変更）をトリガー。
- 視覚的注意機構：タスク関連性に基づく視覚的注意の動的制御。現在のサブタスクに関連するオブジェクトに優先的に注意を向ける。

視覚システムは、20Hzの周期で画像処理を実行し、検出結果とポーズ推定をパブリッシュする。計算負荷の高い処理（3次元再構成など）は、必要に応じて低い周波数（5Hz）で実行される。

#### 3. 力覚モジュール

力覚モジュールは、ロボットのエンドエフェクタが受ける力とトルクを測定し、物体操作の精度を向上させる。実装詳細は以下の通りである：

- 力覚フィルタリング：カルマンフィルタと移動平均フィルタを組み合わせたハイブリッドフィルタリング。高周波ノイズの除去と急激な変化の検出を両立。

- 校正アルゴリズム：重力補償と温度ドリフト補正を含む自動校正プロセス。6軸力覚センサーの各軸を独立に校正し、クロストーク効果を最小化。
- 接触検出：適応的閾値に基づく接触イベントの検出。接触の種類（衝突、スライディング、把持）を分類するSVMモデル。
- 物体特性推定：力覚フィードバックから物体の重さ、硬さ、摩擦係数などを推定するベイジアンモデル。推定精度は重さ±5%、硬さ±10%、摩擦±15%。
- 力制御アルゴリズム：インピーダンス制御、ハイブリッド位置/力制御、適応制御を状況に応じて切り替えるメタ制御器。制御周波数は1kHz。
- 液体操作モデル：液体の注入量を力の変化から推定するモデル。静的平衡条件と動的効果を考慮した二段階推定アルゴリズム。推定精度は±5ml（100mlの注入時）。
- 異常検出：期待される力パターンからの逸脱を検出する異常検出器。マハラノビス距離に基づくアノマリースコアを計算。
- 力覚メモリ：過去の力覚パターンを記憶し、類似状況での適切な力制御パラメータを想起するアソシアティブメモリ。

力覚モジュールは、1kHzの高周波数でデータ取得と基本的なフィルタリングを行い、100Hzの周波数で高次処理（特性推定、異常検出など）を実行する。力覚データは、リングバッファに保存され、過去5秒間のデータにアクセスできる。

#### 4. 音声認識・合成モジュール

音声認識・合成モジュールは、ユーザーとの自然な対話を実現する。実装詳細は以下の通りである：

- 音声認識エンジン：Whisper Large v3モデルをベースとした音声認識システム。日本語と英語のバイリンガル認識に対応。単語誤り率（WER）は日本語5.2%、英語3.8%。
- 音声合成エンジン：FastSpeech 2モデルとHiFi-GANボコーダを組み合わせた高品質音声合成システム。自然性MOS（Mean Opinion Score）4.3/5.0。
- 話者識別：x-vectorとPLDAを用いた話者識別システム。20人の話者を98%の精度で識別。新規話者の登録機能あり。
- 感情認識：音声の韻律特徴とスペクトル特徴を用いた7種類の感情（喜び、悲しみ、怒り、恐れ、嫌悪、驚き、中立）の認識。認識精度は72%。
- 環境音認識：YAMNetをベースとした環境音認識システム。キッチン関連の50種類の音（沸騰音、タイマー音、切断音など）を認識。認識精度は85%。
- 対話管理：階層型有限状態機械とRASA対話管理フレームワークを組み合わせたハイブリッド対話管理システム。コンテキスト維持、修復戦略、確認要求などの機能を実装。
- 適応的音声処理：環境ノイズに応じて認識パラメータを調整する適応アルゴリズム。SNR（Signal-to-Noise Ratio）の低い環境でも安定した認識を実現。
- マルチモーダル対話：視覚情報と音声情報を統合した対話システム。「あれを取って」などの指示語を含む発話に対して、視覚的文脈を考慮した解釈を行う。

音声認識・合成モジュールは、16kHzのサンプリングレートで音声を取得し、リアルタイムで認識処理を行う。認識結果は100ms以内に得られ、対話管理システムに送られる。音声合成は、テキスト入力から200ms以内に合成音声を生成する。

#### 5. マルチモーダル統合モジュール

マルチモーダル統合モジュールは、視覚、力覚、音声などの異なるモダリティからの情報を統合し、より豊かな環境理解と適応的な行動生成を実現する。実装詳細は以下の通りである：

- 確率的統合フレームワーク：ベイジアンネットワークとパーティクルフィルタを組み合わせた確率的統合フレームワーク。各モダリティの不確実性を明示的に考慮。
- 動的重み付けアルゴリズム：各モダリティの信頼度に基づく動的重み付け。信頼度は、センサーの状態、環境条件、タスク要件に基づいて計算。
- クロスモーダル学習：変分オートエンコーダ（VAE）を用いたクロスモーダル表現学習。一方のモダリティから他方のモダリティを予測するモデルを訓練。
- マルチモーダル注意機構：Transformerアーキテクチャに基づくクロスモーダル注意機構。各モダリティ間の関連性を学習し、情報統合を最適化。
- 欠損モダリティ補完：一部のモダリティが利用できない場合（センサー故障、オクルージョンなど）、他のモダリティから欠損情報を予測するメカニズム。
- 時間的統合：異なる時間スケールのモダリティ（高速な力覚、低速な視覚など）を統合するための時間的アライメントアルゴリズム。
- 不確実性伝播：各モダリティの不確実性を統合プロセス全体で伝播させ、最終的な状態推定の不確実性を定量化。
- マルチモーダル記憶：過去のマルチモーダル経験を記憶し、類似状況での適切な統合戦略を想起するエピソード記憶システム。

マルチモーダル統合モジュールは、50Hzの周期で実行され、各モダリティからの最新情報を統合し、環境の状態推定を更新する。統合結果は、ロボット制御システムと言語処理コンポーネントに送信される。

#### 6. ロボット制御システム

ロボット制御システムは、言語処理、視覚システム、力覚モジュール、音声認識・合成モジュール、マルチモーダル統合モジュールからのフィードバックを統合し、ロボットの動作を制御する。実装詳細は以下の通りである：

- 階層制御アーキテクチャ：タスク計画層、動作計画層、軌道生成層、低レベル制御層の4層構造。各層は異なる時間スケールで動作（0.1Hz〜1kHz）。
- 動作プリミティブライブラリ：40種類の基本動作（直線移動、回転、把持、持ち上げ、注ぐなど）を実装。各プリミティブはパラメータ化され、状況に応じて調整可能。
- 軌道最適化：モデル予測制御（MPC）と最適制御理論に基づく軌道最適化。エネルギー効率、滑らかさ、安全性を考慮した多目的最適化。
- 適応制御：環境とタスクに応じて制御パラメータを調整する適応制御アルゴリズム。モデル同定と制御パラメータ調整を並行して実行。
- 安全制約実装：速度制限（±0.05m/s）、力制限（±20N）、作業空間制限（x=[0.0,1.1], y=[-0.3,0.3], z=[0,1.0]）などの安全制約を実装。制約違反の検出と回避メカニズムを含む。
- 障害物回避：動的障害物を考慮した実時間障害物回避アルゴリズム。潜在的場（Potential Field）法とRRT*（Rapidly-exploring Random Tree Star）を組み合わせたハイブリッドアプローチ。
- 失敗検出と回復：動作実行中の失敗を検出し、適切な回復戦略を実行するメカニズム。失敗パターンのデータベースと回復戦略のライブラリを実装。
- 学習型制御：模倣学習と強化学習を組み合わせた学習型制御器。デモンストレーションからの初期ポリシー学習と、実行経験からのポリシー改善を実装。

ロボット制御システムは、異なる周波数で動作する複数のループから構成される。低レベル制御ループは1kHzで実行され、軌道追従と安全監視を担当する。動作計画と軌道生成は50Hzで実行され、環境の変化に応じて計画を更新する。タスク計画は必要に応じて更新され、通常は0.1-1Hzの頻度で実行される。

#### 7. 知識ベース

知識ベースは、検証済みの低次および高次のアクションの例を含むキュレートされたデータベースである。実装詳細は以下の通りである：

- 知識表現：知識は、構造化JSONフォーマットで表現。各エントリには、タスク記述、前提条件、後続条件、実行コード、成功/失敗事例、不確実性情報などが含まれる。
- 階層的組織化：知識は3層の階層構造（高次タスク、中次タスク、低次タスク）で組織化。階層間のリンクにより、タスク分解と合成が容易。
- インデックス化：複数の観点（タスク種類、オブジェクト種類、環境条件など）からのインデックス化。Faissを用いた高速ベクトル検索を実装。
- 不確実性モデリング：各アクション例に対する不確実性情報（成功確率、期待誤差など）を記録。ベイジアンモデルを用いた不確実性の定量化。
- 自動拡張機能：タスク実行の成功事例と失敗事例を自動的に知識ベースに追加する機能。重複検出と品質評価メカニズムを含む。
- バージョン管理：知識ベースの変更履歴を管理するバージョン管理システム。変更の追跡、ロールバック、分岐管理などの機能を実装。
- 分散アクセス：複数のエージェントが同時にアクセスできる分散知識ベース。一貫性維持と競合解決メカニズムを実装。
- セマンティック検索：自然言語クエリに基づくセマンティック検索機能。エンベディングベースの類似度計算と、構造化クエリ言語を組み合わせたハイブリッド検索。

知識ベースは、MongoDB（ドキュメントストア）とFaiss（ベクトルインデックス）を組み合わせたハイブリッドデータベースとして実装。現在、知識ベースには以下のエントリが含まれる：
- 基本動作プリミティブ：100種類（直線移動、回転、把持など）
- 物体操作プリミティブ：80種類（持ち上げ、配置、注ぐなど）
- 特殊操作プリミティブ：50種類（液体注入、粉末すくい取り、引き出し開閉など）
- 中レベルタスク：60種類（コーヒー準備、トースト作成など）
- 高レベルタスク：30種類（朝食準備、ケーキ作りなど）
- エラー処理と回復：40種類（物体落下、液体こぼれなど）
- 環境適応例：35種類（オブジェクト位置変化、障害物出現など）

### 実験設定

実験では、「コーヒーを作る」というタスクを対象とし、以下の詳細条件でシステムの性能を評価した：

#### 1. 環境の不確実性レベル

環境の不確実性を以下の3レベルで設定し、各レベルでシステムの性能を評価した：

- 低不確実性（Level 1）：
  - オブジェクト位置のランダム化：標準偏差2cm
  - 物理特性のばらつき：重さ±5%、摩擦係数±10%
  - センサーノイズ：視覚（σ=0.005）、力覚（σ=0.05N）
  - すべてのオブジェクトが視界内に配置

- 中不確実性（Level 2）：
  - オブジェクト位置のランダム化：標準偏差5cm
  - 物理特性のばらつき：重さ±10%、摩擦係数±20%
  - センサーノイズ：視覚（σ=0.01）、力覚（σ=0.1N）
  - 一部のオブジェクト（マグカップまたはコーヒー豆）が引き出し内に配置

- 高不確実性（Level 3）：
  - オブジェクト位置のランダム化：標準偏差10cm
  - 物理特性のばらつき：重さ±20%、摩擦係数±30%
  - センサーノイズ：視覚（σ=0.02）、力覚（σ=0.2N）
  - 複数のオブジェクト（マグカップ、コーヒー豆）が引き出しや食器棚内に配置
  - 動的障害物（移動するオブジェクト）の導入
  - 一時的なセンサー障害（視覚または力覚の一時的な情報喪失）

#### 2. 指示の抽象度レベル

ユーザーからの指示の抽象度を以下の3レベルで設定し、各レベルでシステムの理解能力と実行能力を評価した：

- 高抽象度（Level A）：
  -「コーヒーを作って」
  -「朝のドリンクを用意して」
  -「カフェインが必要だ」

- 中抽象度（Level B）：
  -「マグカップにコーヒーを入れて」
  -「コーヒー豆を挽いて、お湯を注いで」
  -「ドリップコーヒーを準備して」

- 低抽象度（Level C）：
  -「マグカップを取って、コーヒー豆をすくって、ケトルからお湯を注いで」
  -「赤いマグカップを右の棚から取り出し、コーヒー豆を2スプーン入れ、80度のお湯を150ml注いで」
  -「以下の手順でコーヒーを作って：1. マグカップを準備、2. コーヒー豆を入れる、3. お湯を注ぐ」

#### 3. フィードバック条件

異なるフィードバック条件でシステムの性能を評価するため、以下の条件を設定した：

- 単一モダリティ：
  - 視覚のみ：力覚と音声フィードバックを無効化
  - 力覚のみ：視覚と音声フィードバックを無効化
  - 音声のみ：視覚と力覚フィードバックを無効化

- デュアルモダリティ：
  - 視覚+力覚：音声フィードバックを無効化
  - 視覚+音声：力覚フィードバックを無効化
  - 力覚+音声：視覚フィードバックを無効化

- フルモダリティ：
  - 視覚+力覚+音声：すべてのフィードバックを有効化

- 適応的統合：
  - 状況に応じて各モダリティの重みを動的に調整
  - 不確実性に基づく最適なモダリティ選択
  - アクティブ知覚による情報収集

#### 4. RAG構成

検索強化生成（RAG）の効果を評価するため、以下の構成を比較した：

- RAGなし：
  - 知識ベースを使用せず、言語モデルの一般知識のみに依存

- 基本RAG：
  - 単純なコサイン類似度に基づく検索
  - 上位5件の検索結果を使用
  - 固定チャンクサイズ（512トークン）

- 拡張RAG：
  - ハイブリッド検索（コサイン類似度+BM25）
  - 動的検索数（タスクの複雑さに応じて3-10件）
  - 適応的チャンキング（セマンティックチャンキング）
  - 再ランキングメカニズム

- 適応的RAG：
  - コンテキスト対応クエリ拡張
  - マルチホップ検索（初期検索結果に基づく追加検索）
  - フィードバック情報を考慮した検索
  - 実行履歴に基づく知識ベース更新

#### 5. 評価指標

システムの性能を多角的に評価するため、以下の指標を測定した：

- タスク完了率：タスク全体とサブタスクの完了率
- 実行時間：タスク全体とサブタスクの実行時間
- 精度指標：
  - 液体注入精度：目標量との誤差（ml）
  - 物体配置精度：目標位置との誤差（mm）
  - 力制御精度：目標力との誤差（N）
- 効率指標：
  - エネルギー消費：ロボットの関節トルクの二乗和
  - 動作滑らかさ：加速度の二乗和
  - 計算効率：CPU/GPU使用率と処理時間
- 安全指標：
  - 最大力：タスク実行中の最大接触力
  - 速度違反：安全速度制限を超えた回数
  - 衝突回数：環境との意図しない接触回数
- 適応指標：
  - 回復率：エラーからの回復成功率
  - 探索効率：隠れたオブジェクトの発見率
  - 環境変化対応：動的変化への適応成功率

実験は、各条件の組み合わせ（不確実性レベル×指示抽象度×フィードバック条件×RAG構成）で20回ずつ実行し、結果の統計的有意性を確保した。また、実験条件の順序はランダム化し、学習効果によるバイアスを最小化した。

### 実験結果と詳細分析

#### タスク完了率の包括的分析

「コーヒーを作る」タスクの完了率を、異なる実験条件で詳細に分析した結果を以下に示す：

1. 不確実性レベルとフィードバック条件の交互作用：

提案システム（フルモダリティ+適応的RAG）の不確実性レベル別タスク完了率：
- 低不確実性：95%（20回中19回成功）
- 中不確実性：85%（20回中17回成功）
- 高不確実性：70%（20回中14回成功）

フィードバック条件別の高不確実性環境でのタスク完了率：
- 視覚のみ：45%（20回中9回成功）
- 力覚のみ：30%（20回中6回成功）
- 音声のみ：15%（20回中3回成功）
- 視覚+力覚：65%（20回中13回成功）
- 視覚+音声：50%（20回中10回成功）
- 力覚+音声：35%（20回中7回成功）
- フルモダリティ：70%（20回中14回成功）
- 適応的統合：75%（20回中15回成功）

これらの結果から、環境の不確実性が高まるにつれてタスク完了率が低下するが、マルチモーダルフィードバックの統合によってその影響が緩和されることが明らかになった。特に、視覚と力覚の組み合わせが効果的であり、適応的統合アプローチがさらに性能を向上させることが確認された。

統計分析（二元配置ANOVA）の結果、不確実性レベルの主効果（$F(2,456)=78.3$, $p<0.001$）、フィードバック条件の主効果（$F(7,456)=62.5$, $p<0.001$）、および両者の交互作用（$F(14,456)=12.7$, $p<0.001$）がすべて有意であった。これは、フィードバック条件の効果が不確実性レベルによって異なることを示している。

2. 指示抽象度とRAG構成の交互作用：

指示抽象度別のタスク完了率（フルモダリティ条件）：
- 高抽象度（「コーヒーを作って」）：
  - RAGなし：50%（20回中10回成功）
  - 基本RAG：70%（20回中14回成功）
  - 拡張RAG：80%（20回中16回成功）
  - 適応的RAG：85%（20回中17回成功）

- 中抽象度（「マグカップにコーヒーを入れて」）：
  - RAGなし：65%（20回中13回成功）
  - 基本RAG：75%（20回中15回成功）
  - 拡張RAG：85%（20回中17回成功）
  - 適応的RAG：90%（20回中18回成功）

- 低抽象度（詳細な手順）：
  - RAGなし：80%（20回中16回成功）
  - 基本RAG：85%（20回中17回成功）
  - 拡張RAG：90%（20回中18回成功）
  - 適応的RAG：95%（20回中19回成功）

これらの結果から、指示の抽象度が高いほどRAGの効果が顕著であることが明らかになった。特に、高抽象度の指示では、RAGなしの場合と適応的RAGの場合で35%もの差が生じた。これは、抽象的な指示を具体的なサブタスクに分解する際に、RAGが提供する構造化された知識が重要な役割を果たすことを示している。

統計分析（二元配置ANOVA）の結果、指示抽象度の主効果（$F(2,228)=45.2$, $p<0.001$）、RAG構成の主効果（$F(3,228)=38.7$, $p<0.001$）、および両者の交互作用（$F(6,228)=8.3$, $p<0.001$）がすべて有意であった。

3. 失敗モードの詳細分析：

タスク失敗の原因を詳細に分析した結果、以下のパターンが明らかになった：

- オブジェクト検出失敗：28%（全失敗の中で）
  - 主な原因：視覚的類似性、部分的オクルージョン、照明条件
  - 改善策：マルチビュー統合、アクティブ知覚、時間的一貫性の強化

- 把持失敗：22%
  - 主な原因：滑りやすい表面、不安定な姿勢、重量誤推定
  - 改善策：適応的把持力制御、マルチフィンガー把持、事前把持姿勢最適化

- 液体操作失敗：18%
  - 主な原因：流量推定誤差、容器の傾き制御不良、温度変化
  - 改善策：閉ループ流量制御、視覚-力覚統合強化、予測モデルの精緻化

- 計画失敗：15%
  - 主な原因：不完全なタスク分解、依存関係の誤認識、環境変化への不適応
  - 改善策：階層的計画の強化、動的再計画、コンテキスト対応RAG

- システム統合失敗：10%
  - 主な原因：モジュール間通信遅延、非同期処理の問題、リソース競合
  - 改善策：メッセージングシステムの最適化、優先度ベースのスケジューリング、分散処理

- その他：7%
  - シミュレーション特有の問題、ランダムな障害など

この分析から、視覚的認識と物体操作（特に液体操作）が主要な失敗原因であることが明らかになった。これらの問題に対処するために、マルチモーダルフィードバックの統合と適応的制御が特に重要であることが示唆された。

#### タスク分解と実行の詳細分析

「コーヒーを作る」タスクの分解と実行プロセスを詳細に分析した結果を以下に示す：

1. タスク分解の質的評価：

「コーヒーを作って」という高抽象度指示に対する、異なるRAG構成でのタスク分解例：

- RAGなし：
 1. マグカップを見つける
 2. コーヒー豆を見つける
 3. お湯を準備する
 4. コーヒーを入れる
 （不完全な分解、具体的なサブタスクの欠如）

- 基本RAG：
 1. マグカップを見つける
 2. コーヒー豆を見つける
 3. ケトルを見つける
 4. マグカップを適切な位置に置く
 5. コーヒー豆をマグカップに入れる
 6. お湯を注ぐ
 （基本的なサブタスクを含むが、詳細さに欠ける）

- 適応的RAG：
 1. マグカップを見つける（必要に応じて引き出しや食器棚を探索）
 2. コーヒー豆を見つける（必要に応じて容器を開ける）
 3. スプーンを見つける
 4. ケトルを見つける（必要に応じてお湯を沸かす）
 5. マグカップを安定した場所に置く
 6. スプーンでコーヒー豆を適量すくう
 7. コーヒー豆をマグカップに入れる
 8. ケトルからお湯を適量注ぐ（温度と量を監視）
 9. 必要に応じてかき混ぜる
 （詳細かつ条件付きの行動を含む包括的な分解）

適応的RAGを用いた場合、環境の状態に応じた条件付き行動（「必要に応じて...」）が含まれており、不確実性への対応力が高いことが確認された。また、「スプーンを見つける」など、他のアプローチでは見落とされがちな重要なサブタスクも含まれていた。

2. サブタスク実行の成功率：

各サブタスクの成功率（適応的RAG、フルモダリティ条件）：
- マグカップを見つける：90%
 - 通常位置：95%
 - 引き出し内：85%
 - 食器棚内：80%
- コーヒー豆を見つける：92%
- スプーンを見つける：95%
- ケトルを見つける：98%
- マグカップを置く：97%
- コーヒー豆をすくう：88%
- コーヒー豆を入れる：93%
- お湯を注ぐ：85%
 - 精度（目標量との誤差）：±8ml

これらの結果から、「マグカップを見つける」（特に隠れている場合）と「お湯を注ぐ」が比較的難しいサブタスクであることが明らかになった。これらのサブタスクでは、マルチモーダルフィードバックの統合が特に重要であった。

3. サブタスク間の依存関係処理：

条件付き確率モデルを用いたサブタスク間の依存関係の例：
- P(マグカップを引き出しで探す | マグカップが通常位置にない) = 0.85
- P(マグカップを食器棚で探す | マグカップが引き出しにもない) = 0.90
- P(お湯を沸かす | ケトルのお湯が冷たい) = 0.95
- P(コーヒー容器を開ける | コーヒー豆が見えない) = 0.88

これらの条件付き確率は、RAGを通じて取得された知識と、実行経験から学習されたパターンに基づいて設定された。システムは、これらの確率に基づいて動的に計画を調整し、環境の状態変化に適応することができた。

4. 実行時間分析：

各サブタスクの平均実行時間（秒）：
- マグカップを見つける：12.3（標準位置）、28.7（隠れている場合）
- コーヒー豆を見つける：10.5
- スプーンを見つける：8.2
- ケトルを見つける：9.1
- マグカップを置く：5.4
- コーヒー豆をすくう：15.8
- コーヒー豆を入れる：7.3
- お湯を注ぐ：22.6

全タスクの平均実行時間：95.7秒（並列実行なし）、78.3秒（並列実行あり）

並列実行の例：「ケトルからお湯を沸かしている間に、マグカップとコーヒー豆を準備する」など、独立したサブタスクを並行して実行することで、全体の実行時間を短縮することができた。

#### マルチモーダルフィードバックの詳細分析

マルチモーダルフィードバックの統合と効果を詳細に分析した結果を以下に示す：

1. モダリティ統合パターンの分析：

異なるサブタスクにおける各モダリティの貢献度（重み）：

- マグカップを見つける：
  - 視覚：0.75
  - 力覚：0.05
  - 音声：0.20
  （視覚が主要、音声による位置情報の補完）

- 引き出しを開ける：
  - 視覚：0.40
  - 力覚：0.55

  - 音声：0.05
  （力覚が主要、視覚による位置調整の補助）

- コーヒー豆をすくう：
  - 視覚：0.60
  - 力覚：0.35
  - 音声：0.05
  （視覚と力覚の協調）

- お湯を注ぐ：
  - 視覚：0.30
  - 力覚：0.65
  - 音声：0.05
  （力覚が主要、視覚による位置調整の補助）

これらの重みは、適応的統合アルゴリズムによって動的に調整された。各モダリティの信頼度（センサーの状態、環境条件など）と、タスクの要件に基づいて最適な重みが決定された。

2. クロスモーダル予測の精度：

一方のモダリティから他方のモダリティを予測する精度：

- 視覚→力覚予測：
  - 物体重量予測：±15%（視覚特徴からの推定）
  - 表面摩擦予測：±25%（視覚テクスチャからの推定）

- 力覚→視覚予測：
  - 物体形状予測：60%正確さ（触覚探索からの推定）
  - 液体レベル予測：±12%（重量変化からの推定）

- 音声→視覚予測：
  - オブジェクト位置予測：45%正確さ（音源定位からの推定）
  - イベント検出：75%正確さ（特徴的な音からのイベント推定）

これらの予測は、クロスモーダル学習モデル（変分オートエンコーダベース）によって生成された。モデルは、過去の経験から異なるモダリティ間の関係を学習し、一方のモダリティが利用できない場合に他方から予測することができた。

3. モダリティ欠損時の性能：

一部のモダリティが利用できない場合のタスク完了率（中不確実性環境）：

- すべてのモダリティ利用可能：85%
- 視覚欠損（一時的）：
  - 補完なし：40%
  - クロスモーダル補完あり：65%
- 力覚欠損（一時的）：
  - 補完なし：55%
  - クロスモーダル補完あり：70%
- 音声欠損（一時的）：

- 補完なし：75%
- クロスモーダル補完あり：80%

これらの結果から、クロスモーダル補完メカニズムが、センサー障害や一時的な情報喪失に対するロバスト性を大幅に向上させることが確認された。特に、視覚情報の欠損時に力覚と音声からの情報を用いた補完が効果的であった。

4. 時間的統合の効果：

異なる時間スケールのモダリティ統合による効果：

- 高速フィードバック（1kHz）：力覚制御の安定性向上
  - 力制御誤差：±0.5N（時間的統合あり）vs ±1.2N（なし）
  - 接触検出遅延：5ms（あり）vs 25ms（なし）

- 中速フィードバック（30Hz）：視覚追跡の精度向上
  - 位置推定誤差：±3mm（時間的統合あり）vs ±8mm（なし）
  - 動的オブジェクト追跡成功率：85%（あり）vs 60%（なし）

- 低速フィードバック（5Hz）：計画更新の効率向上
  - 環境変化への適応時間：1.2秒（時間的統合あり）vs 3.5秒（なし）
  - 再計画成功率：90%（あり）vs 75%（なし）

時間的統合アルゴリズムは、異なる周波数のフィードバックを適切に統合し、高速な反応と長期的な一貫性を両立させることができた。特に、力覚の高速フィードバックと視覚の中速フィードバックを組み合わせることで、物体操作の精度と安定性が向上した。

#### RAGの詳細分析

検索強化生成（RAG）の効果と動作メカニズムを詳細に分析した結果を以下に示す：

1. 検索精度と関連性の分析：

異なるRAG構成での検索精度（関連性スコア、0-1スケール）：

- 基本RAG（コサイン類似度のみ）：
  - 平均関連性スコア：0.68
  - 上位5件中の高関連文書（スコア>0.8）の数：1.8件

- 拡張RAG（ハイブリッド検索）：
  - 平均関連性スコア：0.79
  - 上位5件中の高関連文書の数：3.2件

- 適応的RAG（コンテキスト対応）：
  - 平均関連性スコア：0.85
  - 上位5件中の高関連文書の数：4.1件

関連性スコアは、人間の評価者によって事前に評価された基準との一致度に基づいて計算された。適応的RAGは、クエリ拡張とコンテキスト情報の活用により、より関連性の高い文書を検索することができた。

2. 知識ベースサイズとRAG性能の関係：

知識ベースのサイズを変化させた場合のタスク完了率（高不確実性環境）：

- 50エントリ：55%
- 100エントリ：62%
- 200エントリ：68%
- 400エントリ：73%
- 800エントリ：75%
- 1600エントリ：76%

これらの結果から、知識ベースのサイズが増加するにつれてタスク完了率が向上するが、約800エントリ以上では性能の向上が緩やかになることが確認された。これは、知識の量だけでなく質と多様性も重要であることを示唆している。

3. RAGの計算効率：

異なるRAG構成での計算時間（ミリ秒）：

- 基本RAG：
  - エンベディング計算：45ms
  - ベクトル検索：15ms
  - コンテキスト構築：10ms
  - 合計：70ms

- 拡張RAG：
  - エンベディング計算：45ms
  - ハイブリッド検索：25ms
  - 再ランキング：20ms
  - コンテキスト構築：15ms
  - 合計：105ms

- 適応的RAG：
  - クエリ拡張：15ms
  - エンベディング計算：45ms
  - ハイブリッド検索：25ms
  - マルチホップ検索：40ms
  - 再ランキング：20ms
  - コンテキスト構築：20ms
  - 合計：165ms

これらの計算時間は、Intel Core i9-12900K CPU、32GB RAM、NVIDIA RTX 3090 GPUを搭載したワークステーションでの測定値である。エンベディング計算はGPUで並列処理され、検索はCPUで実行された。

適応的RAGは最も計算コストが高いが、キャッシング機構（類似クエリの結果を再利用）を導入することで、平均計算時間を約85msに削減することができた。

4. RAGの具体的な貢献例：

RAGが特に効果的だった具体的なケース：

- 隠れたオブジェクトの探索戦略：
 「マグカップが見つからない」状況で、RAGは過去の類似事例から「引き出しや食器棚を探す」という戦略を提供。探索成功率が40%から85%に向上。

- 液体注入の精度向上：
 「お湯を注ぐ」タスクで、RAGは類似の液体注入例から適切な注ぎ角度と速度を提供。注入精度が±15mlから±8mlに向上。

- エラー回復戦略：
 「コーヒー豆をこぼした」状況で、RAGは類似のエラー回復例から「スプーンで集める」という戦略を提供。回復成功率が50%から85%に向上。

- 未知オブジェクトの操作：
 初めて遭遇する「フレンチプレス」に対して、RAGは類似のコーヒーメーカー操作例から適切な操作方法を提供。タスク完了率が30%から70%に向上。

これらの例は、RAGが単なる情報検索ではなく、過去の経験から学んだ知識を新しい状況に適応させる能力を持つことを示している。

5. 知識転移の分析：

類似タスク間での知識転移の効果：

- 「コーヒーを作る」→「紅茶を入れる」：
 - 転移なし（RAGなし）：60%成功率
 - 転移あり（適応的RAG）：85%成功率

- 「マグカップを取る」→「グラスを取る」：
 - 転移なし：75%成功率
 - 転移あり：95%成功率

- 「引き出しを開ける」→「冷蔵庫を開ける」：
 - 転移なし：65%成功率
 - 転移あり：90%成功率

これらの結果から、RAGを通じた知識転移が、新しいタスクの学習と実行を大幅に効率化することが確認された。特に、操作対象は異なるが基本的な動作パターンが類似するタスク間での転移効果が顕著であった。

#### 安全性と効率性の詳細分析

システムの安全性と効率性を詳細に分析した結果を以下に示す：

1. 安全制約の有効性：

安全制約の有無によるインシデント発生率の比較：

- 安全制約なし：
 - 過度な力（>20N）：15.3%のタスク実行で発生
 - 高速動作（>0.5m/s）：22.7%のタスク実行で発生
 - 作業空間外への移動：8.5%のタスク実行で発生
 - 衝突：18.9%のタスク実行で発生

- 安全制約あり：
  - 過度な力：0.5%のタスク実行で発生（97%削減）
  - 高速動作：0.8%のタスク実行で発生（96%削減）
  - 作業空間外への移動：0%のタスク実行で発生（100%削減）
  - 衝突：3.2%のタスク実行で発生（83%削減）

安全制約の実装により、ほとんどのインシデントが防止されたことが確認された。残存するインシデントは主に、動的環境変化（予期せぬ物体の移動など）に起因するものであった。

2. エネルギー効率の分析：

異なるアプローチでのエネルギー消費（関節トルクの二乗和、相対単位）：

- ベースライン（RAGなし、単一モダリティ）：1.00（基準）
- RAGあり、単一モダリティ：0.85（15%削減）
- RAGなし、マルチモダリティ：0.80（20%削減）
- RAGあり、マルチモダリティ：0.65（35%削減）

エネルギー効率向上の主な要因：
- より滑らかな軌道計画（加速度変化の最小化）
- 適切な把持力の選択（過剰な力の回避）
- 効率的な動作順序（無駄な動きの削減）
- 予測的制御（先読みによる急激な修正の回避）

マルチモーダルフィードバックとRAGの組み合わせにより、最も高いエネルギー効率が達成された。これは、適切なフィードバックと知識の活用が、より効率的な動作計画と実行につながることを示している。

3. 計算効率の分析：

システムコンポーネントごとの計算負荷（CPU/GPU使用率）：

- 言語処理コンポーネント：
  - CPU：15-25%
  - GPU：60-80%（言語モデル推論時）
  - メモリ：4-6GB

- 視覚システム：
  - CPU：10-15%
  - GPU：30-50%（物体検出・セグメンテーション時）
  - メモリ：2-3GB

- 力覚モジュール：
  - CPU：5-8%
  - GPU：< 5%
  - メモリ：0.5-1GB

- マルチモーダル統合モジュール：
  - CPU：8-12%
  - GPU：10-20%

  - メモリ：1-2GB

- ロボット制御システム：
  - CPU：10-15%
  - GPU：< 5%
  - メモリ：1-2GB

全体のピーク使用率：
- CPU：45-60%
- GPU：70-90%
- メモリ：12-18GB

これらの測定値から、言語モデルの推論と視覚処理が最も計算負荷の高いコンポーネントであることが確認された。最適化の余地としては、モデルの量子化、バッチ処理の導入、計算リソースの動的割り当てなどが考えられる。

4. レイテンシ分析：

エンドツーエンドのレイテンシ（指示入力からロボット動作開始までの時間）：

- 高抽象度指示（「コーヒーを作って」）：
  - RAGなし：3.2秒
  - 適応的RAG：3.8秒

- 中抽象度指示（「マグカップにコーヒーを入れて」）：
  - RAGなし：2.1秒
  - 適応的RAG：2.5秒

- 低抽象度指示（詳細な手順）：
  - RAGなし：1.3秒
  - 適応的RAG：1.6秒

コンポーネント別のレイテンシ内訳（適応的RAG、高抽象度指示の場合）：
- 音声認識：0.3秒
- 言語理解：0.5秒
- RAG処理：0.6秒
- タスク分解：1.2秒
- コード生成：0.8秒
- 実行準備：0.4秒

RAGを使用すると若干のレイテンシ増加が見られるが、タスク完了率の向上と実行時間の短縮によって相殺される。また、キャッシング機構とプリフェッチ最適化により、RAGのレイテンシを約40%削減することができた。

5. リアルタイム性能の分析：

環境変化に対する応答時間：

- 物体移動の検出と計画更新：
  - 視覚のみ：450ms

- マルチモーダル：320ms

- 予期せぬ障害物への対応：
 - 検出から回避行動開始まで：180ms
 - 完全な軌道再計画：520ms

- ユーザー指示変更への対応：
 - 音声認識から計画更新まで：850ms
 - 新計画の実行開始まで：1200ms

これらの応答時間は、人間のリアクションタイム（視覚刺激に対して約250ms、聴覚刺激に対して約170ms）と比較して十分に高速であり、自然なインタラクションと安全な操作を実現できることが確認された。

### 結論と今後の展望

Gymnasiumを用いた詳細なシミュレーション実験の結果、提案したマルチモーダルフィードバック統合型知識検索強化ロボット制御システムは、以下の点で高い有効性を示した：

1. 高レベル指示理解と実行能力：
 提案システムは、「コーヒーを作って」などの抽象的な指示を理解し、適切なサブタスクに分解して実行することができた。特に、RAGを用いた知識ベースの活用により、抽象的指示の理解と実行能力が大幅に向上した（RAGなしの50%から適応的RAGの85%へ）。

2. マルチモーダルフィードバックの統合効果：
 視覚、力覚、音声の統合により、単一モダリティよりも大幅に高いタスク完了率が達成された（高不確実性環境で単一モダリティの最大45%から、マルチモーダルの75%へ）。特に、視覚と力覚の組み合わせが効果的であり、適応的統合アプローチがさらに性能を向上させた。

3. 環境の不確実性への適応能力：
 提案システムは、オブジェクト位置のランダム化、物理特性のばらつき、センサーノイズなどの不確実性に対して高い適応能力を示した。特に、高不確実性環境でも70%のタスク完了率を達成し、マルチモーダルフィードバックとRAGの組み合わせが適応能力の向上に大きく寄与した。

4. 知識ベースの効率的活用と知識転移：
 RAGを通じた知識ベースの効率的活用により、タスクの成功率が向上した。また、類似タスク間での知識転移が確認され、新しいタスクの学習と実行が効率化された（転移なしの60-75%から転移ありの85-95%へ）。

5. 安全性と効率性の向上：
 安全制約の実装により、過度な力や速度、作業空間外への移動、衝突などのインシデントが大幅に削減された（83-100%削減）。また、マルチモーダルフィードバックとRAGの組み合わせにより、エネルギー効率が35%向上した。

これらの結果から、提案システムが予測不可能な環境下での複雑なタスク実行に高い有効性を持つことが確認された。特に、マルチモーダルフィードバックの統合とRAGの組み合わせが、環境理解と適応能力の向上に大きく寄与することが明らかになった。

今後の研究課題と展望としては、以下の点が挙げられる：

1. 実機ロボットでの検証：

シミュレーション環境で確認された有効性を、実機ロボットを用いて検証する。シミュレーションと実世界のギャップ（sim-to-real gap）に対処するための技術開発が必要である。

2. より複雑なタスクへの拡張：
複数のオブジェクトを操作する複雑なタスク（例：料理の調理、物体の組み立て）への適用を検討する。特に、長期的な計画能力と適応能力の評価が重要である。

3. マルチモーダル学習の強化：
異なるモダリティ間の関係をより効果的に学習するための手法の開発。特に、少量のデータからの効率的な学習と、モダリティ間の因果関係の理解が課題である。

4. 知識ベースの自動拡張と最適化：
実行経験からの自動的な知識獲得と、知識ベースの最適化手法の開発。特に、矛盾する知識の解決と、知識の一般化能力の向上が重要である。

5. ユーザーとの自然なインタラクションの強化：
より自然な対話と協調作業を実現するためのインタラクション技術の開発。特に、ユーザーの意図理解と、適切なフィードバック提供が課題である。

6. 分散処理と計算効率の向上：
大規模言語モデルの推論を含む計算負荷の高い処理を、分散システムで効率的に実行するための技術開発。特に、エッジデバイスでの低レイテンシ処理が重要である。

7. 倫理的・法的考慮事項の検討：
ロボットの自律的判断と行動に関する倫理的・法的枠組みの検討。特に、安全性保証と責任分担の明確化が必要である。

これらの課題に取り組むことで、より高度な知能と適応能力を持つロボットシステムの実現が期待される。特に、人間と協働し、複雑なタスクを安全かつ効率的に実行できるロボットの開発は、産業、医療、家庭など様々な分野での応用可能性を広げるものである。

## 10. 要約

【課題】予測不可能な環境下での複雑なタスク実行能力の向上、高レベルの抽象的指示の理解と実行、マルチモーダルフィードバックの効果的な統合、環境の変化や不確実性への適応能力の強化、知識ベースの効率的な活用と拡張性の確保。
【解決手段】大規模言語モデル、検索強化生成技術、視覚システム、力覚モジュール、音声認識・合成モジュール、マルチモーダル統合モジュール、およびロボット制御システムを統合したロボット制御システムを提供する。言語処理コンポーネントが高レベル指示を理解し、検索強化生成技術により知識ベースから関連例を動的に選択・適応させる。視覚システムは環境の3次元表現を生成し、力覚モジュールはエンドエフェクタの力を測定する。マルチモーダル統合モジュールが異なるモダリティからの情報を統合し、環境理解と適応的な行動生成を実現する。これにより、予測不可能な環境下での複雑なタスク実行と環境変化への適応が可能となる。

## 11. 先行技術文献

【非特許文献】Mon-Williams, R., Li, G., Long, R. et al. Embodied large language models enable robots to complete complex tasks in unpredictable environments. Nat Mach Intell (2025). https://doi.org/10.1038/s42256-025-01005-x